# Exploring .NET MAUI: MVVM, DI, and Commanding

Up to this point in this article series on .NET MAUI ,you created a set of typical business application input pages and learned about the many different controls you can use for data input. Data binding is a great feature of .NET MAUI to help you eliminate C# code in your applications. In this article, you’ll learn the Model-View-View-Model (MVVM) and Dependency Injection (DI) design patterns to create reusable, maintainable, and testable applications. You’ll learn to eliminate code in your code-behind by taking advantage of Commanding. You’ll also learn how to apply Commanding while keeping your various components reusable across other types of applications. Finally, you’ll learn how to keep your MauiProgram class maintainable by employing extension methods.

## Introduction to Model-View-View-Model (MVVM)

You’ve been using an entity class to supply the controls with data, but an entity class shouldn’t be used for this purpose. A **View Model** class is used to bind data to the controls on the UI. View models have properties used to set which buttons are enabled/visible, which menus are enabled/visible, and for posting informational and error messages. View model classes have methods to load data into entities or save data. View model classes may expose an entity object or have properties to expose only those properties of the entity object needed for the UI.

The whole point of using the MVVM design pattern is better reusability, maintainability, and testability of classes. Design your classes and assemblies in such a manner that you achieve the ability to reuse those classes across multiple projects, as shown in **Figure 1**. The "View Layer" such as WPF, Blazor, .NET MAUI, etc., should know how to use the properties and methods of the view model classes. However, the "View Model Layer" should not know about the view layer. The view model classes should know which repositories (Data Layer) and Model (Entity Layer) classes it can use to fill in its properties. However, the repositories and entities shouldn’t know anything about the view model classes or the view layer. It’s important to keep the classes in the view model, model, and data layer assemblies' technology-agnostic. In other words, those assemblies should be simple class libraries and have no references to any specific front-end technology to achieve maximum reusability.

A diagram of a data flow

Description automatically generated

**Figure 1:** Use the MVVM design pattern for better reusability and maintainability.

### Add More Properties to the Common Base Class

For the application being developed in this article, a few view model classes are needed. Typically, you create a view model class for each view/page you create in your application. Almost all pages in your application need to display information, error, and validation messages to the user. Let's add some properties to the CommonBase class to hold that data. Open the **BaseClasses\CommonBase.cs** file and add a couple of Using statements.

using System.ComponentModel.DataAnnotations.Schema;

using System.Text.Json.Serialization;

Add a string property named **InfoMessage** to hold informational messages, and another string property named **LastErrorMessage** to hold the last error message to display to the user. The **LastException** property keeps track of the last exception generated. The constant REPO\_NOT\_SET is used when you use a repository class to get data from a data store. This constant provides the error message to display if you forget to set the repository object into a class that needs it. You’ll see examples of these later, but for now, add the code shown in **Listing 1** to the CommonBase.cs file.

Notice that each property raises the PropertyChanged event when a new value is set into the property. Two attributes are added to each property [NotMapped] and [JsonIgnore]. The [NotMapped] attribute is used because each entity class inherits from this class and if you use the Entity Framework, you don’t want it thinking it needs to locate a column with any of these property names. The [JsonIgnore] attribute is optional, but you typically don’t want to send these properties as JSON when serializing and deserializing an entity object through a Web API.

## Create a View Model Base Class

Like you did with the entity classes, create a view model base class that each view model inherits from. Right mouse-click on the **Common.Library\BaseClasses** folder and add a new class named **ViewModelBase** and replace the entire contents of the new file with the code shown in **Listing 2**.

This class inherits from the CommonBase class, then adds an additional property and method. The **RowsAffected** property is set after each method that either retrieves or modifies data in a data store. This can be useful to display to the user how many items were found when searching, or how many items were modified. The **PublishException**() method is where you write code to perform logging of the exception.

### Create View Model Layer Class Library

As you did with the User class, create a new Class Library project into which you place all your view model classes. Right mouse-click on the **Solution** and add a new Class Library project named **AdventureWorks.ViewModelLayer**. Delete the **Class1.cs** file as this file is not used. Right mouse-click on the **Dependencies** folder in this new **AdventureWorks.ViewModelLayer** project and add a project reference to the **Common.Library** project and to the **AdventureWorks.EntityLayer** project.

### Add a User View Model Class

Right mouse-click on the **AdventureWorks.ViewModelLayer** project and add a new folder named **ViewModelClasses**. Right mouse-click on the **ViewModelClasses** folder and add a new class named **UserViewModel**. Replace the entire contents of this new file with the code shown in **Listing 3**.

The UserViewModel class (**Listing 3**) inherits from the ViewModelBase class you just created. It contains a property named **CurrentEntity** that’s a User data type. The CurrentEntity property is used to bind each property in the User class to the appropriate controls on the UserDetailView page. There are three methods in this view model class; **GetAsync**(), **GetAsync**(**id**), and **SaveAsync**(). These methods do not currently interact with any data store, but later, you’ll hook these up to different repository classes that get data from different data stores.

The GetAsync() method returns a list of users that can be bound to a collection-type view on the UserList page. For now, this method returns an empty list, but you’ll add a list of user data soon. The GetAsync(id) method sets the CurrentEntity property to a single user object, then returns that object. The user object is hard coded currently, but later this data will come from a data store. The SaveAsync() method will eventually be called from the Save button on the user page, and for now, just returns a new user.

## Use the User View Model on XAML

It’s now time to change the user detail view to use the UserViewModel class instead of the User class. Right mouse-click on the **Dependencies** folder in this new **AdventureWorks.MAUI** project and add a project reference to the **AdventureWorks.ViewModelLayer** project. Open the **Views\UserDetailView.xaml** file and **change** the XML namespace "vm" to point to the ViewModelLayer assembly, as shown in the following code snippet.

xmlns:vm="clr-namespace:

AdventureWorks.**ViewModelLayer**;

assembly=AdventureWorks.**ViewModelLayer**"

Change the **x:DataType** attribute on the ContentPage to use the UserViewModel instead of the User class.

x:DataType="vm:User**ViewModel**"

Remove the **<vm:User x:Key="viewModel" …>** from the <ContentPage.Resources> element as you’re now going to use the hard-coded data returned from the GetAsync(id) method in the UserViewModel class. Remove the **BindingContext** attribute from the <Border> element as the BindingContent is going to be set in the code behind. Your <Border> element should now look like the following XAML.

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

### Change All Bindings to Use the CurrentEntity Property

Because you’ve now wrapped up the User object within the CurrentEntity property on the UserViewModel class, change all the Binding markup extensions to use **CurrentEntity** before the property names, as shown in the following XAML. You should be able to do this with a search and replace in the Visual Studio editor.

{Binding **CurrentEntity.**PROPERTY\_NAME}

### Modify the Code Behind

Open the **Views\UserDetailView.xaml.cs** file and **change** the **using** statement that points to the EntityLayer to point to the ViewModelLayer instead, as shown in the following code snippet:

using AdventureWorks.ViewModelLayer;

Remove the line of code from the constructor that set the ViewModel to the value coming from the **Resources** collection. That was the view model created in the XAML that’s no longer there. Your constructor should now look like the following:

public UserDetailView() {

InitializeComponent();

}

Change the **\_ViewModel** variable to a **UserViewModel** data type as shown in the following code snippet. Create a **new** instance of the UserViewModel on the same line as the declaration.

private readonly **UserViewModel** \_ViewModel = new();

Modify the OnAppearing() event procedure to look like the following code. Make sure you add the **async** keyword to the OnAppearing() event procedure. Set the BindingContext for the entire ContentPage to the **\_ViewModel** variable. Call the GetAsync(id) method on the view model asynchronously to have the single instance of the User object placed into the CurrentEntity property.

protected **async** override void OnAppearing() {

base.OnAppearing();

**// Set the BindingContext to the ViewModel**

**BindingContext = \_ViewModel;**

**// Retrieve a User**

**await \_ViewModel.GetAsync(1);**

}

### Try It Out

Run the application and click on **Users > Navigate to Detail**. You should see that the hard-coded User data from the view model class is now bound to the controls on this user detail page.

## Load Phone Picker Using MVVM

If you remember, the phone type list is a XAML resource located in the AppStyles.xaml file. Let's create a list of phone types in the user view model class so that list can be bound on the page. Open the **ViewModelClasses\UserViewModel.cs** file and add a new private variable that’s an **ObservableCollection** of strings. An ObservableCollection in .NET raises the PropertyChanged event whenever the collection is modified in any manner. This collection type should be used for all collections when working with .NET MAUI or WPF.

private ObservableCollection<string>

\_PhoneTypesList = [];

Let's expose this list of phone types so it can be bound on the user detail page. Add a public property named **PhoneTypesList** to the UserViewModel class as shown in the code snippet below.

public ObservableCollection<string>

PhoneTypesList {

get { return \_PhoneTypesList; }

set {

\_PhoneTypesList = value;

RaisePropertyChanged(nameof(PhoneTypesList));

}

}

In the UserViewModel class create a method named **GetPhoneTypes**() to load the list of phone types, as shown in the following code. All data is hard coded for now, but later in this article series, you’ll get data from a data store.

#region GetPhoneTypes Method

public async

Task<ObservableCollection<string>>

GetPhoneTypes() {

PhoneTypesList = await Task.FromResult(

new ObservableCollection<string>(

["Home", "Mobile", "Work", "Other"])

);

return PhoneTypesList;

}

#endregion

### Modify the User Detail View

Now that you have a method to load phone types, that method needs to be called before the user detail page is displayed. Open the **Views\UserDetailView.xaml.cs** file and add a call to the GetPhoneTypes() method in the **OnAppearing** event procedure, as shown in the following code.

protected async override void OnAppearing() {

base.OnAppearing();

// Set the Page BindingContext

BindingContext = ViewModel;

// Get the Phone Types

await \_ViewModel.GetPhoneTypes();

// Retrieve a User

await \_ViewModel.GetAsync(1);

}

Because you’re now getting the phone types from the view model class, you no longer need the XAML array. Open the **Resources\Styles\AppStyles.xaml** file and delete the **<x:Array x:Key="phoneTypes" …>** element from the <ResourceDictionary> element. Open the **Views\UserDetailView.xaml** file and locate the <Picker> and change it to look like the following XAML.

<Picker VerticalTextAlignment="Center"

SelectedItem="{Binding CurrentEntity.PhoneType}"

**ItemsSource="{Binding PhoneTypesList}"** />

The **SelectedItem** property is bound to the **PhoneType** property on the user object so if the selection in the Picker control is changed, the **PhoneType** property in the user object is updated with the new value. The **ItemsSource** property is changed from the StaticResource markup extension to use the Binding markup extension that points to the observable collection of phone types in the UserViewModel class.

### Try It Out

Run the application, click on **User > Navigate to Detail** and you should still see the same list of phone types, and the picker should be positioned on the value for the user read from the view model class.

## Introduction to Dependency Injection

I want you to think about the <ShellContent> elements you created in the AppShell.xaml file. Each ShellContent object uses a **DataTemplate** markup extension to which you pass the name of a ContentPage you want to be displayed when clicked upon. You don’t have to write any C# code to create the ContentPage; the .NET MAUI navigation engine takes care of all the details of page creation.

Think of some of the most common things you do when you’re writing a typical business application. You probably use a tool like Log4Net or Serilog to record exceptions, information, and debug messages. You probably have an Entity Framework (EF) database context object for working with your database. You might have a set of repository classes that are responsible for working with each table in your database. You might also have a configuration manager class to retrieve application-wide settings from a JSON file, or maybe from a database.

The reason I’m pointing out these items is that I want you think about how many times in your application you might be instantiating your logging, your database context, your repository, or your configuration classes. If you have even a medium size application, the answer is that you’re probably instantiating these classes in hundreds of locations throughout your code base. Now, think of the maintenance nightmare you’d encounter if you needed to replace the logging class with a different one, or that all your repository classes can't use a database context anymore, but need to make Web API calls. If you don't program with Dependency Injection, a change like this could cause hundreds of hours of re-work.

Dependency Injection (DI) helps you remove coupling between classes. In order to use DI, create an interface (or base class) for each of your logging, configuration, repository, database context, and other classes to implement or inherit. Now, instead of creating an instance of Serilog in each class that needs to perform logging, register Serilog with a DI service in the MauiProgram class. Most loggers today implement the ILogger interface. In each class that needs to use logging, you add the ILogger interface as an argument to the constructor of that class. You do the same process with repository, configuration, and even the EF DbContext class, as shown in **Figure 2**.

A screenshot of a computer

Description automatically generated

**Figure 2:** Services are placed into a DI container waiting to be referenced by an instance of a class.

Register each interface and the corresponding class with the **Services** collection of the **MauiAppBuilder** object in the CreateMauiApp() method in the MauiProgram class, as shown in the following code snippet:

builder.Services.AddScoped

<IRepository<User>, UserRepository>();

builder.Services.AddScoped

<IRepository<PhoneType>, PhoneTypeRepository>();

In this code, you specify a service lifetime (Scoped, Singleton, Transient) and pass the interface followed by the class that implements that interface. Service lifetimes will be explained shortly. Each of these classes are placed into the DI container and are then ready to be used. A class is used when an instance of a class is created by .NET MAUI and that class has a constructor expecting a parameter of an interface type registered in the DI container.

In addition to classes that implement interfaces you also register view models and views if they need to receive any of the classes from DI to do their work. For example, if the constructor of the UserDetailView has an IRepository<User> as an argument (shown in the following code) the navigation engine asks the DI container to look up the interface to see if it’s registered. If it is, an instance of the UserRepository class is created by the DI engine and passed to the **repo** argument when the UserDetailView class is instantiated.

public UserDetailView(IRepository<User> repo) {

InitializeComponent();

// DO SOMETHING WITH THE REPOSITORY CLASS

}

### Use Interfaces for Repository Classes

In a typical business application, you build a set of repository classes to talk to a data store to get and modify data. Each repository class needs certain standard methods such as Get(), Get(id), Insert(), Update(), Delete(), etc., that manipulate the data in a single table of your data store. Each repository class implements the interface by creating each method to perform the specified logic. If you have three tables in your application, such as Product, User, and PhoneTypes, you create three repository classes to implement each of the methods for performing CRUD logic.

For each repository class, you might start out using EF and a DbContext object to manipulate the data in each of these tables. However, when testing applications, you might want to use mock data instead. If you use .NET MAUI for a cross-platform application, each repository class should call Web APIs to perform the data manipulation. Each of these different types of classes implements the same interface, but each performs it differently.

Place each of these different repository implementations into different assemblies (**Figure 3**) and you reference just one of those assemblies at a time in your .NET MAUI program (**Figure 4**). The repository classes from the referenced assembly are the ones registered with the DI container in the MauiProgram class. As long as each view model and/or view only uses the repository interface, it doesn’t matter which assembly you use, as they are completely interchangeable.

A group of orange rectangular objects with white text

Description automatically generated

**Figure 3:** Use an interface to create different implementations of a service.

A diagram of a computer program

Description automatically generated with medium confidence

**Figure 4:** The .NET MAUI application selects one assembly with the implementation of an interface to use.

### DI Service Lifetimes

The **Services** collection on the MauiAppBuilder object has three different methods you can use to register classes into the DI container; AddSingleton(), AddScoped(), and AddTransient(). When you add a class as a **Singleton,** the first time that class is requested by a constructor, the object is created and has a scope of the entire lifetime of the application. If another class requests that class, the exact same instance of the object first created is used. Because a .NET MAUI application is inherently single user, this type of lifetime is fine for views and view model classes.

When you add a class as **Scoped**, an instance of that class is created within the scope of the object that created it. When the object that created it is destroyed, this object is also destroyed. This type of lifetime is good for repository objects because if you have a DbContext object, you don't want that object to stay around any longer than necessary. This type of lifetime is appropriate for views and view models.

A class added as **Transient** is like a scoped object, but there’s no pre-defined lifetime. It may be released anytime, and if requested again, a new instance is created. This type of lifetime is good for classes that log data or read configuration items.

## Add an IRepository Interface

Right mouse-click on the **Common.Library** project and add a new folder named **Interfaces**. Right mouse-click on the new **Interfaces** folder and add a new file named **IRepository**. Replace the entire contents of the new file with the following code:

using System.Collections.ObjectModel;

namespace Common.Library;

public interface IRepository<TEntity> {

Task<ObservableCollection<TEntity>> GetAsync();

Task<TEntity?> GetAsync(int id);

}

The generic IRepository<TEntity> interface requires you to identify the Entity class name when declaring a variable. For example, IRepository<User> or IRepository<Product> identifies the type of object(s) returned from the GetAsync() or GetAsync(id) methods. All the Repository classes you build in this article series are going to be asynchronous, so all method names are going to reflect this. Other methods to support data modification will be added later to this interface.

### Use the IRepository Interface in the User View Model Class

Let's now use the IRepository interface in the user view model class. The Common.Library project is already referenced from the view AdventureWorks.ViewModelLayer project, so the IRepository interface is available to use. Open the **ViewModelClasses\UserViewModel.cs** file and add a new private variable that is of the type IRepository<User>, as shown in the following line of code:

private readonly IRepository<User>? \_Repository;

Add two constructors, one empty, and the other injects the IRepository<User> interface, as shown in the following code. Assign the **repo** variable to the private read-only **\_Repository** variable. You haven’t built this repository class yet, but you don't need it to create the appropriate code in the user view model class to make calls to the methods defined in the interface.

#region Constructors

public UserViewModel() {

}

public UserViewModel(IRepository<User> repo)

: base() {

\_Repository = repo;

}

#endregion

### Use the IRepository Interface in View Model Classes

Modify the **GetAsync(id)** method (**Listing 4**) to retrieve a single user by calling the GetAsync(id) method of the **\_Repository** variable. Of course, you’re going to ensure that this object is of the type UserRepository a little later. In the code shown in **Listing 4,** check to see if the **\_Repository** variable is not null. If it’s not null, make a call to the GetAsync(id) on the repository class. The value returned is placed into the **CurrentEntity** property. Once the **CurrentEntity** property is set and the PropertyChanged event fires, the UI re-reads the data and updates all bindings on the page.

Also notice that the code sets the **LastErrorMessage** property if the **\_Repository** variable has not been set. The **InfoMessage** property is set to one of two messages depending on whether the user ID was located or not. If mock data is returned because the **\_Repository** variable was null, then the **InfoMessage** property is set to tell the user that mock data was used. Finally, the **RowsAffected** property is set to a value of one (1).

### Try It Out

Run the application and click on **Users < Navigate to Detail** to see the hard-coded user coming from this new method. The messages do not show up anywhere, but those will be added later in this article series.

## Create a Data Layer Class Library

As stated before, you may want to have different data stores from which you get data. I’d highly recommend that you create different class library projects for each of these different data stores. In this article, you’re going to use mock data. Right mouse-click on the **Solution** and add a new **Class Library** project named **AdventureWorks.DataLayer.Mock**. Delete the **Class1.cs** file, as this file isn’t needed. Right mouse-click on the **Dependencies** folder in the **AdventureWorks.DataLayer.Mock** project and add two project references to the **AdventureWorks.EntityLayer** and **Common.Library** class libraries.

### Add a User Repository Class

Right mouse-click on the new **AdventureWorks.DataLayer.Mock** project and add a folder named **RepositoryClasses**. Right mouse-click on the **RepositoryClasses** folder and add a new class named **UserRepository**. Replace the entire contents of this new file with the code shown in **Listing 5**. The UserRepository class implements the IRepository interface and thus contains the implementation of the two methods GetAsync() and GetAsync(id). The GetAsync() method creates a list of three users and returns them as an ObservableCollection of User objects. The GetAsync(id) method retrieves the list of users from the GetAsync() method, then applies the LINQ Where() method to the list to locate the ID passed in as the parameter.

### Add Dependency in MAUI Project

As mentioned previously, you need one concrete implementation of a data layer in your .NET MAUI application. Right mouse-click on the **Dependencies** folder in the **AdventureWorks.MAUI** project and a project reference to **AdventureWorks.DataLayer.Mock** project. Do **NOT** add a reference to the DataLayer.Mock project to the view model layer. All the view model class needs to know is the interface that the repository classes are using. The concrete implementation of the repository classes is created by .NET MAUI's DI engine and those implementations are injected into the view models.

### Inject View Models into Views

You now have the IRepository<User> being passed to the constructor of the UserViewModel class. You need to pass an instance of the UserViewModel class to the UserDetailView page so the page can bind to the properties of the view model class, and make calls to any methods. Open the **Views\UserDetailView.xaml.cs** file and modify the constructor to accept the view model class and to assign that variable to the **\_ViewModel** variable.

public partial class UserDetailView

: ContentPage {

public UserDetailView(**UserViewModel viewModel**) {

InitializeComponent();

**\_ViewModel = viewModel;**

}

// REST OF THE CODE HERE

}

## Register Classes with DI Container

The responsibility for creating an instance of the UserDetailView is through the .NET MAUI navigation system. So, how do the UserViewModel and the UserRepository classes get passed to the UserDetailView and the UserViewModel respectively? Register these classes in the MauiProgram class using the MauiAppBuilder object. Open the **MauiProgram.cs** file and add some new using statements.

using AdventureWorks.DataLayer;

using AdventureWorks.EntityLayer;

using AdventureWorks.MAUI.Views;

using AdventureWorks.ViewModelLayer;

using Common.Library;

Just below the **builder.UseMauiApp<App>()**, add the following code to inject the repository, the view model, and the view into the DI container.

// Add classes for use in Dependency Injection

builder.Services.AddScoped<IRepository<User>,

UserRepository>();

builder.Services.AddScoped<UserViewModel>();

builder.Services.AddScoped<UserDetailView>();

### Try It Out

Run the application and click on the **User > Navigate to Detail** menu to see the user associated with the **UserId** property equal to one (1) in the user repository appear. Stop the application and try using one of the other user ID values such as 2 or 3.

## Load Phone Types Using Repository Class

Earlier in this article, you loaded the phone types by creating a collection in the UserViewModel class. Instead of hard coding these types in the UserViewModel class, create a PhoneType entity class, and a PhoneTypeRepository class to retrieve the phone type data from a data store. Creating entity and repository classes for phone types allows you to reuse them on other pages.

### Add a PhoneType Entity Class

Right mouse-click on the **EntityClasses** folder in the **AdventureWorks.EntityLayer** project and add a new class named **PhoneType**. Replace the entire contents of this new file with the code shown in **Listing 6**. This class contains a phone type identifier and the description of the phone type. The same design pattern is followed in this class as in the User class. You use private variables for each property, then expose that private variable through getter and setter. Don't forget to add the call to the RaisePropertyChanged() event in the setter. Open the **ViewModelClasses\UserViewModel.cs** file and add a new private variable of the type **PhoneTypeRepository,** as shown in the following line of code.

private readonly IRepository<PhoneType>?

\_PhoneTypeRepository;

Add a third constructor to accept an **IRepository<PhoneType>** object from the DI engine, as shown in the following code.

public UserViewModel(IRepository<User> repo,

IRepository<PhoneType> phoneRepo) : base() {

\_Repository = repo;

\_PhoneTypeRepository = phoneRepo;

}

Locate the GetPhoneTypes() method and modify the code to look like **Listing 7**. The **\_PhoneTypeRepository** private variable injected into the constructor is used to retrieve the list of phone types from the data store used by the PhoneTypeRepository class.

### Add a Phone Type Repository Class

Right mouse-click on the **RepositoryClasses** folder in the **AdventureWorks.DataLayer.Mock** project and add a new class named **PhoneTypeRepository**. Replace the entire contents of this new file with the code shown in **Listing 8**. Once again, a familiar design pattern is applied to the PhoneTypeRepository class. Implement the IRepository interface and create the two methods GetAsync() and GetAsync(id).

### Add Phone Types to DI

Open the **MauiProgram.cs** file in the **AdventureWorks.MAUI** project and add a new service to the DI container to use the **PhoneTypeRepository** class.

### Try It Out

Run the application, click on **User > Navigate to Detail** and you should still see the same list of phone types, and the phone type picker should be positioned on the value for the user read from the repository.

## Eliminate Event Handling with Commanding

The MVVM design pattern eliminates a lot of code from the UI layer by removing code behind pages and moving it down to a view model. On the user detail page, you have two buttons: Save and Cancel. You created an event procedure to handle the click event on the Save button, as shown in the following code.

private void SaveButton\_Clicked(object sender,

EventArgs e) {

// TODO: Respond to the event here

System.Diagnostics.Debugger.Break();

}

The SaveButton\_Clicked event procedure is mapped to the Save button by setting the Clicked attribute as shown below.

<Button Text="Save"

Clicked="SaveButton\_Clicked" />

As you can imagine, as you add more buttons to a page, your code behind can grow significantly. Eliminating code behind helps make your code more portable if you wish to recreate your UI in a different technology. The more code you have in your view model, entity, and repository classes, the more your code is reusable.

The SaveButton\_Clicked event procedure is eliminated by using a technique called **Commanding**. Commanding is where you create a property that is of the type **ICommand**. You then create an instance of this ICommand by instantiating a **Command** class. Once you’ve created this Command property, bind it to the Save button with code that looks like the following:

<Button Text="Save"

Command="{Binding SaveCommand}" />

The ICommand interface is defined in the System.Windows.Input namespace, which is located in the System.ObjectModel assembly. The System.ObjectModel assembly is a part of the Microsoft.NETCore.App framework, so the ICommand can be used in your view model layer assembly if you wish. However, the concrete implementation of the Command class is located in the Microsoft.Maui.Controls project. You don’t want to add this assembly into your view model layer as this couples the view model assembly to .NET MAUI and that can cause problems if you want to reuse the view model assembly in an MVC, a WPF, or a Blazor application.

To take advantage of commanding, create a view model class in the .NET MAUI application that inherits from the view model class in the view model layer project. The view model class created in the .NET MAUI application can implement the commanding, yet all the other reusable functionality is kept in the view model layer project.

### Implement Commanding in Your Project

Right mouse-click on the **AdventureWorks.MAUI** project and add a new folder named **MauiViewModelClasses**. Right mouse-click on the **MauiViewModelClasses** folder and add a new class named **UserViewModel**. Replace the entire contents of this new file with the code shown in **Listing 9**. In the code shown in **Listing 9,** create three constructors just like you did in the UserViewModel in the ViewModelLayer project. Each of these constructors call the base constructor passing in the arguments. The **SaveCommand** property is defined as an ICommand interface. This property is initialized in the Init() method. The Init() method is called from the CommonBase class constructor so it doesn’t need to be called from any of the view model constructors. The first parameter to the Command constructor is any anonymous method that says to call the SaveAsync() method located in the UserViewModel class in the ViewModelLayer project.

### Change Your XAML File to Use the New View Model Class

You need to use the new view model class in your XAML files, instead of the view model class from the view model layer project. Open the **Views\UserDetailView.xaml** file and change the "**xmlns:vm**" namespace to reference the new namespace in the .NET MAUI application, as shown below.

xmlns:vm="clr-namespace:

AdventureWorks.MAUI.MauiViewModelClasses"

Modify the **Save** button and remove the **Clicked** attribute, as shown in the code snippet below. Add a **Command** attribute and set the Binding markup extension to the SaveCommand property you created as show below. When the Save button is clicked upon, the SaveCommand is fired and the SaveAsync() method is invoked.

<Button Text="Save"

ImageSource="save.png"

ToolTipProperties.Text="Save Data"

ContentLayout="Left"

**Command="{Binding SaveCommand}"** />

### Change the View Model Reference

Open the **MauiProgram.cs** file and remove the modify the DI injection to use the new MauiViewModelClasses.UserViewModel class instead of the view model coming from the view model layer project.

builder.Services.AddScoped<

**MauiViewModelClasses**.UserViewModel>();

### Modify the UserDetailView Code

Open the **Views\UserDetailView.xaml.cs** file and change the **using AdventureWorks.ViewModelLayer;** to use the new namespace, as shown below.

using AdventureWorks.MAUI.MauiViewModelClasses;

Remove the **SaveButton\_Clicked()** event procedure, as this code is no longer needed.

### Try It Out

Run the application, click on the **Users > Navigate to Detail** menu, click on the **Save** button, and you should end up on the Break() method in the SaveAsync() method in the UserViewModel class in the ViewModelLayer project. There are more options in the Command class that you can take advantage of, such as the ability to disable a button through a **CanExecute** property. I don't tend to use this property as I control disabling buttons in my view model by binding a Boolean view model property to the **IsEnabled** property on the buttons. This makes the view model code easier to reuse in technologies that don't support commanding or don't implement commanding in the same manner.

### Advantages of Commanding

Many times, the code in an event handler is just a single line of code calling a method in a view model. By using commanding, you directly call methods in the view model without the event handler code. Commanding thereby eliminates event handling in the user interface of applications making your applications more reusable, easier to test, and more maintainable.

### Disadvantages of Commanding

Although there are great advantages to using commanding, there are some downsides to it as well. First and foremost, commanding isn’t available natively in all UI technologies such as Windows Forms, Web Forms, or WPF. Many of the "Community Toolkits" for these technologies do provide some form of commanding. However, not all these toolkits implement commanding in the same way. You can solve a lot of these pitfalls by using the design pattern I presented in this article. Take advantage of inheritance and just write a little bit of code in the UI layer and leave your view model classes without any ties to a specific UI technology or toolkit.

## Clean Up the MauiProgram Class

Open the **MauiProgram.cs** file in the AdventureWorks.MAUI project and you’ll see that you have quite a few lines just adding repository, view model, and view classes into the DI container. As you can imagine, this list of services can get quite long and thus make your MauiProgram class unmanageable. The **builder.Services** object is an IServiceCollection data type, so let's create a set of extension methods for this type in a separate class.

Right mouse-click on the AdventureWorks.MAUI project and add a new folder named **ExtensionClasses**. Right mouse-click on the **ExtensionClasses** folder and add a new class named **ServiceExtensions**. Replace the entire contents of this new file with the code shown in **Listing 10**.

There’s one public method in the ServiceExtensions class, AddServicesToDIContainer(), that’s called from the MauiProgram.CreateMauiApp() method. This method calls three other private methods to add repository, view model, and view classes. If you have other types of classes to add to the DI container, add additional methods as appropriate. The ServiceExtension class accomplishes the same functionality as the lines of code currently in the MauiProgram.cs file but organizes the adding of services into different methods. Once you have the ServiceExtensions class created, eliminate the lines of code in the MauiProgram class that added classes to the Services collection with the code shown below.

// Add Classes to Dependency Container

builder.Services.AddServicesToDIContainer();

Go to the top of the **MauiProgram.cs** file and eliminate any unnecessary **using** statements.

### Try It Out

Run the application to ensure that everything still works as it did before all the changes you just made.

### Clean Up Warnings in Error List Window

While we’re on the subject of cleaning up, let's also ensure that you’re using the **x:DataType** attribute on all pages and partial pages. If you remember, the x:DataType attribute, when applied on a page, uses compiled bindings on that page. Compiled bindings improve the speed of data binding at runtime by resolving binding expressions at compile-time. Open the **ViewsPartial\HeaderView.xaml** file and add an XML namespace to the ViewsPartial namespace, as shown below.

xmlns:partial="clr-namespace:

AdventureWorks.MAUI.ViewsPartial"

Add the x:DataType attribute, as shown in the following code snippet:

x:DataType="partial:HeaderView"

By adding these two lines, you eliminate some warnings in the Error List window. Not all can be eliminated yet, as you haven’t created a view model for the product data. These warnings will eventually be eliminated as you continue on with this article series.

## Summary

Using the Model-View-View-Model and the Dependency Injection design patterns helps you create reusable, maintainable, and testable code. Always be thinking how to keep common code in separate assemblies for maximum reusability. Commanding is a great way to reduce code behind, but ensures that you keep the code in the UI layer and not in your generic class libraries. Keep your MauiProgram.cs file organized by creating extension methods in a separate class. In the next article, you’ll learn to display lists of data, select an item from the list, and navigate to the detail page for that item.

Getting the Sample Code

You can download the sample code for this article by visiting [www.CODEMag.com](www.CODEMag.com%20) under the issue and article, or by visiting <www.pdsa.com/downloads>. Select "Articles" from the Category drop-down. Then select "Exploring .NET MAUI: MVVM, DI, and Commanding" 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](mailto: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>) on topics ranging from C#, LINQ, JavaScript, Angular, MVC, WPF, XML, jQuery, and Bootstrap. Contact Paul at [psheriff@pdsa.com](mailto:psheriff@pdsa.com).

Listing 1: Add message properties to the CommonBase class.

#region Private/Protected Variables

private string \_InfoMessage = string.Empty;

private string \_LastErrorMessage = string.Empty;

private Exception? \_LastException = null;

protected const string REPO\_NOT\_SET =

"The Repository Object is not Set.";

#endregion

#region Public Properties

[NotMapped]

[JsonIgnore]

public string InfoMessage {

get { return \_InfoMessage; }

set {

\_InfoMessage = value;

RaisePropertyChanged(nameof(InfoMessage));

}

}

[NotMapped]

[JsonIgnore]

public string LastErrorMessage {

get { return \_LastErrorMessage; }

set {

\_LastErrorMessage = value;

RaisePropertyChanged(

nameof(LastErrorMessage));

}

}

[NotMapped]

[JsonIgnore]

public Exception? LastException {

get { return \_LastException; }

set {

\_LastException = value;

LastErrorMessage = \_LastException == null

? string.Empty : \_LastException.Message;

RaisePropertyChanged(nameof(LastException));

}

}

#endregion

Listing 2: Create a view model base class from which all your view models inherit.

namespace Common.Library;

public class ViewModelBase : CommonBase {

#region Private Variables

private int \_RowsAffected;

#endregion

#region Public Properties

public int RowsAffected

{

get { return \_RowsAffected; }

set

{

\_RowsAffected = value;

RaisePropertyChanged(nameof(RowsAffected));

}

}

#endregion

#region PublishException Method

protected virtual void PublishException(Exception ex)

{

LastException = ex;

System.Diagnostics.Debug.WriteLine(ex.ToString());

}

#endregion

}

Listing 3: Create a user view model class to which to bind your controls.

using AdventureWorks.EntityLayer;

using Common.Library;

using System.Collections.ObjectModel;

namespace AdventureWorks.ViewModelLayer;

public class UserViewModel : ViewModelBase {

#region Private Variables

private User? \_CurrentEntity = new();

#endregion

#region Public Properties

public User? CurrentEntity {

get { return \_CurrentEntity; }

set {

\_CurrentEntity = value;

RaisePropertyChanged(

nameof(CurrentEntity));

}

}

#endregion

#region GetAsync Method

public async

Task<ObservableCollection<User>> GetAsync()

{

return await Task.FromResult(

new ObservableCollection<User>());

}

#endregion

#region GetAsync(id) Method

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

try {

// TODO: Get a User from a data store

// MOCK Data

CurrentEntity = await

Task.FromResult(new User {

UserId = id,

LoginId = "SallyJones615",

FirstName = "Sally",

LastName = "Jones",

Email = "Sallyj@jones.com",

Phone = "615.987.3456",

PhoneType = "Mobile",

IsFullTime = true,

IsEnrolledIn401k = true,

IsEnrolledInFlexTime = false,

IsEnrolledInHealthCare = true,

IsEnrolledInHSA = false,

IsEmployed = true,

BirthDate =

Convert.ToDateTime("08-13-1989"),

StartTime = new TimeSpan(7, 30, 0)

});

RowsAffected = 1;

}

catch (Exception ex) {

RowsAffected = 0;

PublishException(ex);

}

return CurrentEntity;

}

#endregion

#region SaveAsync Method

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

// TODO: Write code to save data

System.Diagnostics.Debugger.Break();

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

}

#endregion

}

Listing 4: Use a Repository object to retrieve data from a data store.

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

try {

**// Get a User from a data store**

**if (\_Repository != null) {**

**CurrentEntity = await \_Repository.GetAsync(id);**

**if (CurrentEntity == null) {**

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

**}**

**else {**

**InfoMessage = "Found the User";**

**}**

**}**

**else {**

**LastErrorMessage = REPO\_NOT\_SET;**

**InfoMessage = "Found a MOCK User";**

// MOCK Data

CurrentEntity =

await Task.FromResult(new User {

UserId = id,

LoginId = "SallyJones",

FirstName = "Sally",

LastName = "Jones",

Email = "Sallyj@jones.com",

Phone = "615.987.3456",

PhoneType = "Mobile",

IsEnrolledIn401k = true,

IsEnrolledInFlexTime = false,

IsEnrolledInHealthCare = true,

IsEnrolledInHSA = false,

IsEmployed = true,

BirthDate

= Convert.ToDateTime("08-13-1989"),

StartTime = new TimeSpan(7, 30, 0)

});

}

RowsAffected = 1;

}

catch (Exception ex) {

RowsAffected = 0;

PublishException(ex);

}

return CurrentEntity;

}

Listing 5: Create a repository class to return a set of mock user data.

using AdventureWorks.EntityLayer;

using Common.Library;

using System.Collections.ObjectModel;

using System.Data;

namespace AdventureWorks.DataLayer;

/// <summary>

/// Create a set of User mock data

/// </summary>

public partial class UserRepository

: IRepository<User> {

#region GetAsync Method

public async

Task<ObservableCollection<User>>

GetAsync() {

return await Task.FromResult(

new ObservableCollection<User>()

{

new() {

UserId = 1,

LoginId = @"MichaelThomason",

FirstName = @"Michael",

LastName = @"Thomason",

Email = @"MichaelThomason@advworks.com",

Phone = @"615.555.3333",

PhoneType = @"Mobile",

IsFullTime = true,

IsEnrolledIn401k = true,

IsEnrolledInHealthCare = true,

IsEnrolledInHSA = false,

IsEnrolledInFlexTime = false,

IsEmployed = true,

BirthDate = new DateTime(1985, 3, 15),

StartTime = null,

},

new() {

UserId = 2,

LoginId = @"SheilaCleverly",

FirstName = @"Sheila",

LastName = @"Cleverly",

Email = @"SheilaCleverly@advworks.com",

Phone = @"615.123.3456",

PhoneType = @"Other",

IsFullTime = false,

IsEnrolledIn401k = false,

IsEnrolledInHealthCare = true,

IsEnrolledInHSA = false,

IsEnrolledInFlexTime = false,

IsEmployed = true,

BirthDate = new DateTime(1981, 6, 9),

StartTime = new TimeSpan(7, 30, 0),

},

new() {

UserId = 3,

LoginId = @"CatherineAbel",

FirstName = @"Catherine",

LastName = @"Abel",

Email = @"CatherineAbel@advworks.com",

Phone = @"615.998.3332",

PhoneType = @"Mobile",

IsFullTime = true,

IsEnrolledIn401k = true,

IsEnrolledInHealthCare = true,

IsEnrolledInHSA = true,

IsEnrolledInFlexTime = true,

IsEmployed = true,

BirthDate = new DateTime(1979, 4, 5),

StartTime = null,

}

// ADD MORE MOCK DATA HERE

});

}

#endregion

#region GetAsync(id) Method

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

ObservableCollection<User> list =

await GetAsync();

User? entity = list.Where(

row => row.UserId == id).FirstOrDefault();

return entity;

}

#endregion

}

Listing 6: Create a PhoneType entity class.

using Common.Library;

namespace AdventureWorks.EntityLayer;

public class PhoneType : EntityBase {

#region Private Variables

private int \_PhoneTypeId;

private string \_TypeDescription

= string.Empty;

#endregion

#region Public Properties

public int PhoneTypeId {

get { return \_PhoneTypeId; }

set {

\_PhoneTypeId = value;

RaisePropertyChanged(nameof(PhoneTypeId));

}

}

public string TypeDescription {

get { return \_TypeDescription; }

set {

\_TypeDescription = value;

RaisePropertyChanged(

nameof(TypeDescription));

}

}

#endregion

}

Listing 7: Modify the GetPhoneTypes method to use the repository to retrieve the list of phone types.

#region GetPhoneTypes Method

public async Task<ObservableCollection<string>>

GetPhoneTypes() {

if (\_PhoneTypeRepository != null) {

var list =

await \_PhoneTypeRepository.GetAsync();

PhoneTypesList = new

ObservableCollection<string>

(list.Select(row => row.TypeDescription));

}

return PhoneTypesList;

}

#endregion

Listing 8: Add a repository class to work with PhoneType objects.

using AdventureWorks.EntityLayer;

using Common.Library;

using System.Collections.ObjectModel;

namespace AdventureWorks.DataLayer;

/// <summary>

/// Create a set of PhoneType mock data

/// </summary>

public partial class PhoneTypeRepository

: IRepository<PhoneType> {

#region GetAsync Method

public async

Task<ObservableCollection<PhoneType>>

GetAsync() {

return await Task.FromResult(

new ObservableCollection<PhoneType>

{

new() {

PhoneTypeId = 1,

TypeDescription = "Home",

},

new() {

PhoneTypeId = 2,

TypeDescription = "Mobile",

},

new() {

PhoneTypeId = 3,

TypeDescription = "Work",

},

new() {

PhoneTypeId = 4,

TypeDescription = "Other",

}

});

}

#endregion

#region GetAsync(id) Method

public async Task<PhoneType?>

GetAsync(int id) {

ObservableCollection<PhoneType> list

= await GetAsync();

PhoneType? entity = list.Where(

row => row.PhoneTypeId == id)

.FirstOrDefault();

return entity;

}

#endregion

}

Listing 9: Create a Commanding View Model class in your .NET MAUI application.

using AdventureWorks.EntityLayer;

using Common.Library;

using System.Windows.Input;

namespace AdventureWorks.MAUI.MauiViewModelClasses;

public class UserViewModel :

AdventureWorks.ViewModelLayer.UserViewModel {

#region Constructors

public UserViewModel() : base() {

}

public UserViewModel(IRepository<User> repo)

: base(repo) {

}

public UserViewModel(IRepository<User> repo,

IRepository<PhoneType> phoneRepo) :

base(repo, phoneRepo) {

}

#endregion

#region Commands

public ICommand? SaveCommand

{ get; private set; }

#endregion

#region Init Method

public override void Init() {

base.Init();

// Create commands for this view

SaveCommand = new Command(

async () => await SaveAsync());

}

#endregion

}

Listing 10: Add a class for extension methods of the IServicesCollection object.

using AdventureWorks.DataLayer;

using AdventureWorks.EntityLayer;

using AdventureWorks.MAUI.Views;

using AdventureWorks.ViewModelLayer;

using Common.Library;

namespace AdventureWorks.MAUI.ExtensionClasses;

public static class ServiceExtensions {

public static void AddServicesToDIContainer(

this IServiceCollection services) {

// Add Repository Classes

AddRepositoryClasses();

// Add View Model Classes

AddViewModelClasses(services);

// Add View Classes

AddViewClasses(services);

}

private static void AddRepositoryClasses(

IServiceCollection services) {

// Add Repository Classes

services.AddScoped<IRepository<User>,

UserRepository>();

services.AddScoped<IRepository<PhoneType>,

PhoneTypeRepository>();

}

private static void AddViewModelClasses(

IServiceCollection services) {

// Add View Model Classes

services.AddScoped<UserViewModel>();

}

private static void AddViewClasses(

IServiceCollection services) {

// Add View Classes

services.AddScoped<UserDetailView>();

}

}