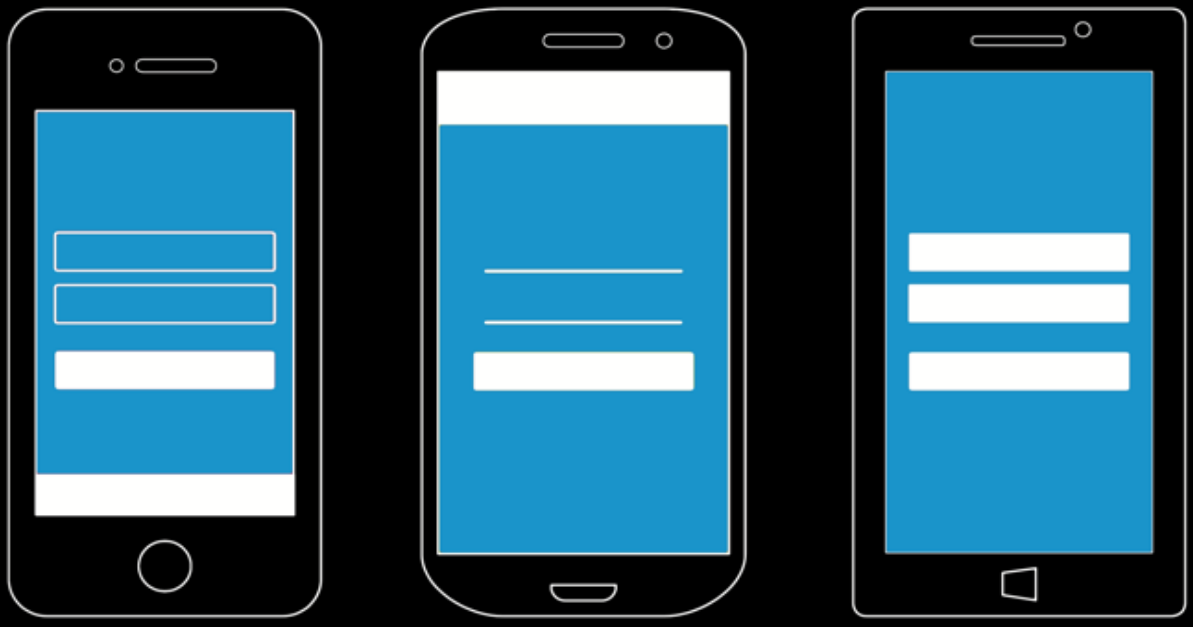
Early Draft

Microsoft

Developing Enterprise Mobile Applications with Xamarin.Forms



David Britch
Microsoft Corp.

PUBLISHED BY

DevDiv, .NET and Visual Studio product teams
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Author: David Britch
Participants and reviewers: Craig Dunn
Editor:

Contents

Preface

Introduction

MVVM

The Xamarin.Forms developer experience typically involves creating a user interface in XAML, and then adding code-behind that operates on the user interface. As apps are modified, and grow in size and scope, complex maintenance issues can arise. These issues include the tight coupling between the UI controls and the business logic, which increases the cost of making UI modifications, and the difficulty of unit testing such code.

The Model-View-ViewModel (MVVM) pattern helps to cleanly separate the business and presentation logic of an application from its user interface (UI). Maintaining a clean separation between application logic and the UI helps to address numerous development issues and can make an application easier to test, maintain, and evolve. It can also greatly improve code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts of an app.

The MVVM pattern

There are three core components in the MVVM pattern: the model, the view, and the view model. Each serves a distinct purpose. Figure 2-1 shows the relationships between the three components.

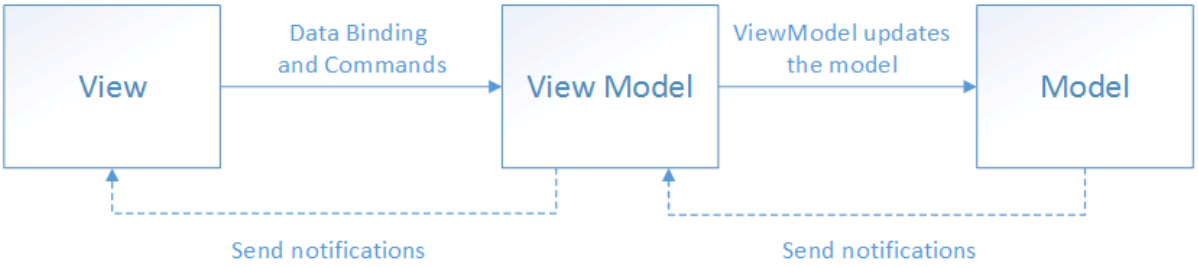


Figure 2-1: The MVVM pattern.

In addition to understanding the responsibilities of each components, it's also important to understand how they interact with each other. At a high level, the view "knows about" the view model, and the view model "knows about" the model, but the model is unaware of the view model, and the view model is unaware of the view. Therefore, the view model isolates the view from the model, and allows the model to evolve independently of the view.

The benefits of using the MVVM pattern are as follows:

- Developers can create unit tests for the view model and the model, without using the view. The unit tests for the view model can exercise exactly the same functionality as used by the view.
- The app UI can be redesigned without touching the code, provided that the view is implemented entirely in XAML. Therefore, a new version of the view should work the existing view model.
- Designers and developers can work independently and concurrently on their components during the development process. Designers can focus on the view, while developers can work on the view model and model components.

- If there's an existing model implementation that encapsulates existing business logic, it can be difficult or risky to change it. In this scenario, the view model acts as an adapter for the model classes and enables you to avoid making any major changes to the model code.

The key to using MVVM effectively lies in understanding how to factor app code into the correct classes, and in understanding how the classes interact. The following sections discuss the responsibilities of each of the classes in the MVVM pattern.

View

The view is responsible for defining the structure, layout, and appearance of what the user sees on screen. Ideally, each view is defined in XAML, with a limited code-behind that does not contain business logic. However, in some cases, the code-behind may contain UI logic that implements visual behavior that is difficult to express in XAML, such as animations.

In a Xamarin.Forms application, a view is typically a `Page`-derived or `ContentView`-derived class. However, views can also be represented by a data template, which specifies the UI elements to be used to visually represent an object when it's displayed. A data templates as a view does not have any code-behind, and is designed to bind to a specific view model type.

> **Tip:** Avoid enabling and disabling UI elements in the code-behind
>
> Ensure that view models are responsible for defining logical state changes that affect some aspect of the view's display, such as whether a command is available, or an indication that an operation is pending. Therefore, enable and disable UI elements by binding to view model properties, rather than enabling and disabling them in code-behind.

There are several options for executing code on the view model in response to interactions on the view, such as a button click or item selection. If a control supports commands, the control's `Command` property can be data-bound to an `ICommand` property on the view model. When the control's command is invoked, the code in the view model will be executed. In addition to commands, behaviors can be attached to an object in the view and can listen for either a command to be invoked or event to be raised. In response, the behavior can then invoke an `ICommand` on the view model or a method on the view model.

ViewModel

The view model implements properties and commands to which the view can data bind to, and notifies the view of any state changes through change notification events. The properties and commands that the view model provides define the functionality to be offered by the UI, but the view determines how that functionality is to be displayed.

> **Tip:** Keep the UI responsive with asynchronous operations
>
> Mobile apps should keep the UI thread unblocked to improve the user's perception of performance. Therefore, in the view model, use asynchronous methods for I/O operations and raise events to asynchronously notify views of property changes.

The view model is also responsible for coordinating the view's interactions with any model classes that are required. There's typically a one-to-many relationship between the view model and the model classes. The view model may choose to expose model classes directly to the view so that controls in the view can data bind directly to them. In this case, the model classes will need to be designed to support data binding and change notification events.

Each view model provides data from a model in a form that the view can easily consume. To accomplish this, the view model sometimes performs data conversion. Placing this data conversion in

the view model is a good idea because it provides properties that the view can bind to. For example, the view model may combine the value of two properties to make it easier for display by the view.

> **Tip:** Centralize data conversions in a conversion layer
>
> It's also possible to use converters as a separate data conversion layer that sits between the view model and the view. This can be necessary, for example, when data requires special formatting that the view model doesn't provide.

In order for the view model to participate in two-way data binding with the view, its properties must raise the `PropertyChanged` event. View models satisfy this requirement by implementing the `INotifyPropertyChanged` interface, and raising the `PropertyChanged` event when a property is changed.

For collections, the view-friendly `ObservableCollection<T>` is provided. This collection implements collection changed notification, relieving the developer from having to implement the `INotifyCollectionChanged` interface on collections.

Model

Model classes are non-visual classes that encapsulate the app's data. Therefore, the model can be thought of as representing the app's domain model, which usually includes a data model along with business and validation logic. Examples of model objects include data transfer objects (DTOs), Plain Old CLR Objects (POCOs), and generated entity and proxy objects.

Model classes are typically used in conjunction with a service or repository that encapsulates data access and caching.

Connecting view models to views

MVVM uses the data-binding capabilities of Xamarin.Forms to connect view models to views. There are many approaches that can be used to construct views and view models and associate them at runtime. However, all share the same aim, which is for the view to have a view model assigned to its `BindingContext` property.

There are two categories of approaches for constructing views and view models, and associating them at runtime. They are known as view first composition, and view model first composition, and deciding whether an app will construct views or view models first is an issue of preference and complexity.

With view first composition the app is conceptually composed of views which connect to the view models they depend on. The primary benefit of this approach is that it makes it easy to construct loosely coupled, unit testable apps because the view models have no dependence on the views themselves. It's also easy to understand the structure of the app by following its visual structure, rather than having to track code execution to understand how classes are created and associated. In addition, view first construction aligns with the Xamarin.Forms navigation system that's responsible for constructing pages when navigation occurs, which makes a view model first composition complex and misaligned with the platform.

View model first composition feels more natural to some developers, since the view creation can be abstracted away, allowing them to focus on the logical non-UI structure of the app. However, this approach is often complex and it can become difficult to understand how the various parts of the app are created and associated.

> **Tip:** Keep view models and views independent

> The binding of views to a property in a data source should be the view's principal dependency on its corresponding view model. Specifically, don't reference view types from view models. If you follow the principles outlined here, you'll have the ability to test view models in isolation, therefore reducing the likelihood of software defects by limiting scope.

The following sections discuss the main approaches to connecting view models to views.

Creating a view model declaratively

The simplest approach is for the view to declaratively instantiate its corresponding view model in XAML. When the view is constructed, the corresponding view model object will also be constructed. This approach is demonstrated in the following code example:

```
<ContentPage ... xmlns:local="clr-namespace:eShop">
    <ContentPage.BindingContext>
        <local:LoginViewModel />
    </ContentPage.BindingContext>
    ...
</ContentPage>
```

When the `ContentPage` is created, an instance of the `LoginViewModel` is automatically constructed and set as the view's `BindingContext`.

This declarative construction and assignment of the view model by the view has the advantage that it's simple, but has the disadvantage that it requires a default (parameter-less) constructor in the view model.

Creating a view model programmatically

A view can have code in the code-behind file that results in the view model being assigned to its BindingContext property. This is often accomplished in the view's constructor, as shown in the following code example:

```
public LoginView()
{
    InitializeComponent();
    BindingContext = new LoginViewModel(navigationService);
}
```

The programmatic construction and assignment of the view model within the view's code-behind has the advantage that it's simple. However, the main disadvantage of this approach is that the view needs to provide the view model with any required dependencies. Using a dependency injection container can help to main loose coupling between the view and view model. For more information, see Introduction to dependency injection.

Creating a view defined as a data template

A view can be defined as a data template and associated with a view model type. Data templates can be defined as resources, or they can be defined inline within the control that will display the view model. The content of the control is the view model instance, and the data template is used to visually represent it. This technique is an example of a situation in which the view model is instantiated first, followed by the creation of the view.

Automatically creating a view model with a view model locator

A view model locator is a class that manages the instantiation of view models and their association to views. In the eShopOnContainers mobile app, the `ViewModelLocator` class, has an attached property, `AutoWireViewModel`, that's used to associate view models with views. In the view's XAML this attached property is set to `true` to indicate that the view model should be automatically connected to the view, as shown in the following code example:

```
viewModelBase:ViewModelLocator.AutoWireViewModel="true"
```

The `AutoWireViewModel` property is a bindable property that's initialized to `false`, and when its value changes the `OnAutoWireViewModelChanged` event handler is called. This method resolves the view model for the view. The following code example shows how this is achieved:

```
private static void OnAutoWireViewModelChanged(BindableObject bindable, object oldValue, object n
ewValue)
{
    var view = bindable as Element;
    if (view == null)
    {
        return;
    }

    var viewType = view.GetType();
    var viewName = viewType.FullName.Replace(".Views.", ".ViewModels.");
    var viewAssemblyName = viewType.GetTypeInfo().Assembly.FullName;
    var viewModelName = string.Format(CultureInfo.InvariantCulture, "{0}Model, {1}", viewName, vi
ewAssemblyName);

    var viewModelType = Type.GetType(viewModelName);
    if (viewModelType == null)
    {
        return;
    }
    var viewModel = _unityContainer.Resolve(viewModelType);
    view.BindingContext = viewModel;
}
```

The `OnAutoWireViewModelChanged` method attempts to resolve the view model using a convention-based approach. This convention assumes that:

- View models are in the same assembly as view types.
- Views are in a .Views child namespace.
- View models are in a .ViewModels child namespace.
- View model names correspond with view names and end with "ViewModel".

Finally, the `OnAutoWireViewModelChanged` method sets the `BindingContext` of the view type to the resolved view model type. For more information about resolving the view model type, see Resolution.

This approach has the advantage that an app has a single class that is responsible for the instantiation of view models and their connection to views.

Tip: Use a view model locator for ease of substitution

A view model locator can also be used as a point of substitution for alternate implementations of dependencies, such as for unit testing or design time data.

Updating views in response to changes in the underlying view model or model

All view model and model classes that are accessible to a view should implement the `INotifyPropertyChanged` interface. Implementing this interface in a view model or model class allows the class to provide change notifications to any data-bound controls in the view when the underlying property value changes. The eShopOnContainers mobile app uses the `ExtendedBindableObject` class to provide change notifications , which is shown in the following code example:

```
public abstract class ExtendedBindableObject : BindableObject
{
    public void RaisePropertyChanged<T>(Expression<Func<T>> property)
    {
        var name = GetMemberInfo(property).Name;
        OnPropertyChanged(name);
    }

    private MemberInfo GetMemberInfo(Expression expression)
    {
        ...
    }
}
```

Xamarin.Form's `BindableObject` class implements the `INotifyPropertyChanged` interface, and provides an `OnPropertyChanged` method. The ExtendedBindableObject class provides the `RaisePropertyChanged` method to invoke property change notification, and in doing so uses the functionality provided by the `BindableObject` class.

Each view model class in the eShopOnContainers mobile app derives from the `ViewModelBase` class, which in turn derives from the `ExtendedBindableObject` class. Therefore, each view model class uses the RaisePropertyChanged method in the ExtendedBindableObject class to provide property change notification. The following code example shows how the eShopOnContainers mobile app invokes property change notification by using a lambda expression:

```
public bool IsLogin
{
    get
    {
        return _isLogin;
    }
    set
    {
        _isLogin = value;
        RaisePropertyChanged(() => IsLogin);
    }
}
```

Note that using a lambda expression in this way involves a small performance cost because the lambda expression has to be evaluated for each call. Although the performance cost is small and would not normally impact an app, the costs can accrue when there are many change notifications.

However, the benefit of this approach is that it provides compile-time type safety and refactoring support when renaming properties.

Tip: Architect apps for the correct use of property change notification.

Always implement the `INotifyPropertyChanged` interface on any view model or model classes that are accessible to the view.

Always raise a `PropertyChanged` event if a public property's value changes. Do not assume that raising the `PropertyChanged` event can be ignored because of knowledge of how XAML binding occurs.

Always raise a `PropertyChanged` event for any calculated properties whose values are used by other properties in the view model or model.

Always raise the `PropertyChanged` event at the end of the method that makes a property change, or when the object is known to be in a safe state. Raising the event interrupts the operation by invoking the event's handlers synchronously. If this happens in the middle of an operation, it may expose the object to callback functions when it is in an unsafe, partially update state. In addition, it's possible for cascading changes to be triggered by `PropertyChanged` events. Cascading changes generally require updates to be complete before the cascading change is safe to execute.

Never raise a `PropertyChanged` event if the property does not change. This means that you must compare the old and new values before raising the `PropertyChanged` event.

Never raise the `PropertyChanged` event during a view model's constructor if you are initializing a property. Data-bound controls in the view will not have subscribed to receive change notifications at this point.

Never raise more than one `PropertyChanged` event with the same property name argument within a single synchronous invocation of a public method of a class. For example, given a `NumberOfItems` property whose backing store is the `_numberOfItems` field, if a method increments `_numberOfItems` fifty times during the execution of a loop, it should only raise property change notification on the `NumberOfItems` property once, after all the work is complete. For asynchronous methods, raise the `PropertyChanged` event for a given property name in each synchronous segment of an asynchronous continuation chain.

UI interaction using commands and behaviors

In mobile apps, actions are typically invoked in response to a user action, such as a button click, that can be implemented by creating an event handler in the code-behind file. However, in the MVVM pattern, the responsibility for implementing the action lies with the view model, and placing code in the code-behind should be avoided.

Commands provide a convenient way to represent actions that can be bound to controls in the UI. They encapsulate the code that implements the action, and help to keep it decoupled from its visual representation in the view. Xamarin.Forms includes controls that can be declaratively connected to a command, and these controls will invoke the command when the user interacts with the control.

Behaviors also allow controls to be declaratively connected to a command. However, behaviors can be used to invoke an action that's associated with a range of events raised by a control. Therefore, behaviors address many of the same scenarios as command-enabled controls, while providing a greater degree of flexibility and control. In addition, behaviors can also be used to associate command objects or methods with controls that were not specifically designed to interact with commands.

Implementing commands

View models typically expose command properties, for binding from the view, that are object instances that implement the `ICommand` interface. A number of Xamarin.Forms controls provide a `Command` property, which can be data bound to an `ICommand` object provided by the view model. The `ICommand` interface defines an `Execute` method, which encapsulates the operation itself, a `CanExecute` method, which indicates whether the command can be invoked, and a `CanExecuteChanged` event that occurs when changes occur that effect whether the command should execute. The `Command` and `Command<T>` classes, provided by Xamarin.Forms, implement the `ICommand` interface, where `T` is the type of the arguments to `Execute` and `CanExecute`.

Within a view model, there should be an object of type `Command` or `Command<T>` for each public property in the view model of type `ICommand`. The `Command` or `Command<T>` constructor requires an `Action` callback object, that's called when the `ICommand.Execute` method is invoked. The `CanExecute` method is an optional constructor parameter, and is a `Func` that returns a `bool`.

The following code shows how a `Command` instance, which represents a register command, is constructed by specifying a delegate to the `Register` view model method:

```
public ICommand RegisterCommand => new Command(Register);
```

The command is exposed to the view through a property that returns a reference to an `ICommand`. When the `Execute` method is called on the `Command` object, it simply forwards the call to the method in the view model via the delegate that was specified in the `Command` constructor.

An asynchronous method can be invoked by a command by using the `async` and `await` keywords when specifying the command's `Execute` delegate. This indicates that the callback is a `Task` and should be awaited. For example, the following code shows how a `Command` instance, which represents a sign-in command, is constructed by specifying a delegate to the `SignInAsync` view model method:

```
public ICommand SignInCommand => new Command(async () => await SignInAsync());
```

Parameters can be passed to the `Execute` and `CanExecute` actions by using the `Command<T>` class to instantiate the command. For example, the following code shows how a `Command<T>` instance is used to indicate that the `NavigateAsync` method will require an argument of type `string`:

```
public ICommand NavigateCommand => new Command<string>(NavigateAsync);
```

In both the `Command` and `Command<T>` classes, the delegate to the `CanExecute` method in each constructor is optional. If a delegate isn't specified, the `Command` will return `true` for `CanExecute`. However, the view model can indicate a change in the command's `CanExecute` status by calling the `ChangeCanExecute` method on the `Command` object. This causes the `CanExecuteChanged` event to be raised. Any controls in the UI that are bound to the command will then update their enabled status to reflect the availability of the data-bound command.

Invoking commands from a view

The following code example shows how a `Grid` in the `LoginView` binds to the `RegisterCommand` in the `LoginViewModel` class by using a `TapGestureRecognizer` instance:

```
<Grid Grid.Column="1" HorizontalOptions="Center">
    <Label Text="REGISTER" TextColor="Gray"/>
    <Grid.GestureRecognizers>
        <TapGestureRecognizer Command="{Binding RegisterCommand}" NumberOfTapsRequired="1" />
```

```
    </Grid.GestureRecognizers>
</Grid>
```

A command parameter can also be optionally defined using the `CommandParameter` property. The type of the expected argument is specified in the `Execute` and `CanExecute` target methods. The `TapGestureRecognizer` will automatically invoke the target command when the user interacts with the attached control. The command parameter, if provided, will be passed as the argument to the command's `Execute` delegate.

Implementing behaviors

Behaviors allow functionality to be added to UI controls without having to subclass them. Instead, the functionality is implemented in a behavior class and attached to the control as if it was part of the control itself. Behaviors enable you to implement code that you would normally have to write as code-behind, because it directly interacts with the API of the control, in such a way that it can be concisely attached to the control, and packaged for reuse across more than one view or app. In the context of MVVM, behaviors are a useful approach for connecting controls to commands.

A behavior that's attached to a control through attached properties is known as an *attached behavior*. The behavior can then use the exposed API of the element to which it is attached to add functionality to that control or other controls in the visual tree of the view. The eShopOnContainers mobile app contains the `LineColorBehavior` class, which is an attached behavior. For more information, see Displaying validation errors.

A Xamarin.Forms behavior is a class that derives from the `Behavior` or `Behavior<T>` class, where T is the type of the control to which the behavior should apply. These classes provide `OnAttachedTo` and `OnDetachingFrom` methods, which should be overridden to provide logic that will be executed when the behavior is attached to and detached from controls.

In the eShopOnContainers mobile app, the `BindableBehavior<T>` class derives from the `Behavior<T>` class. The purpose of the `BindableBehavior<T>` class is to provide a base class for Xamarin.Forms behaviors that require the `BindingContext` of the behavior to be set to the attached control.

The `BindableBehavior<T>` class provides an overridable `OnAttachedTo` method that sets the `BindingContext` of the behavior, and an overridable `OnDetachingFrom` method that cleans up the `BindingContext`. In addition, the class stores a reference to the attached control in the `AssociatedObject` property.

The eShopOnContainers mobile app includes an `EventToCommandBehavior` class, which executes a command in response to an event occurring. This class derives from the `BindableBehavior<View>` class so that the behavior can bind to and execute an `ICommand` specified by a `Command` property when the behavior is consumed. The following code example shows the `EventToCommandBehavior` class:

```
public class EventToCommandBehavior : BindableBehavior<View>
{
    ...
    protected override void OnAttachedTo(View visualElement)
    {
        base.OnAttachedTo(visualElement);

        var events = AssociatedObject.GetType().GetRuntimeEvents().ToArray();
        if (events.Any())
        {
            _eventInfo = events.FirstOrDefault(e => e.Name == EventName);
            if (_eventInfo == null)
```

```
                throw new ArgumentException(String.Format("EventToCommand: Can't find any event n
amed '{0}' on attached type", EventName));

            AddEventHandler(_eventInfo, AssociatedObject, OnFired);
        }
    }

    protected override void OnDetachingFrom(View view)
    {
        if (_handler != null)
            _eventInfo.RemoveEventHandler(AssociatedObject, _handler);

        base.OnDetachingFrom(view);
    }

    private void AddEventHandler(EventInfo eventInfo, object item, Action<object, EventArgs> acti
on)
    {
        ...
    }

    private void OnFired(object sender, EventArgs eventArgs)
    {
        ...
    }
}
```

The `OnAttachedTo` and `OnDetachingFrom` methods are used to register and deregister an event handler for the event defined in the `EventName` property. Then, when the event fires, the `OnFired` method is invoked, which executes the command.

The advantage of using the `EventToCommandBehavior` to execute a command when an event fires, is that commands can be associated with controls that weren't designed to interact with commands. In addition, this moves event-handling code to view models, where it can be unit tested.

Invoking behaviors from a view

The `EventToCommandBehavior` is particularly useful for attaching a command to a control that doesn't support commands. For example, the `ProfileView` uses the `EventToCommandBehavior` to execute the `OrderDetailCommand` when the `ItemTapped` event fires on the `ListView` that lists the user's orders, as shown in the following code:

```
<ListView>
    <ListView.Behaviors>
        <behaviors:EventToCommandBehavior
            EventName="ItemTapped"
            Command="{Binding OrderDetailCommand}"
            EventArgsConverter="{StaticResource ItemTappedEventArgsConverter}" />
    </ListView.Behaviors>
    ...
</ListView>
```

At runtime, the `EventToCommandBehavior` will respond to interaction with the `ListView`. When an item is selected in the `ListView`, the `ItemTapped` event will fire, which will execute the `OrderDetailCommand` in the `ProfileViewModel`. By default, the event arguments for the event are passed to the command. This data is converted as it's passed between source and target by the converter specified in the `EventArgsConverter` property, which returns the `Item` of the `ListView` from the `ItemTappedEventArgs`. Therefore, when the `OrderDetailCommand` is executed, the selected `Order` is passed as a parameter to the registered `Action`.

For more information about behaviors, see Behaviors on the Xamarin Developer Center.

Summary

The Model-View-ViewModel (MVVM) pattern helps to cleanly separate the business and presentation logic of an application from its user interface (UI). Maintaining a clean separation between application logic and the UI helps to address numerous development issues and can make an application easier to test, maintain, and evolve. It can also greatly improve code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts of an app.

Using the MVVM pattern, the UI of the app and the underlying presentation and business logic is separated into three separate classes: the view, which encapsulates the UI and UI logic; the view model, which encapsulates presentation logic and state; and the model, which encapsulates the app's business logic and data.

Dependency Injection

Typically, a class constructor is invoked when instantiating an object, and any values that the object needs are passed as arguments to the constructor. This is an example of dependency injection, and specifically is known as *constructor injection*. The dependencies the object needs are injected into the constructor.

By specifying dependencies as interface types, dependency injection enables decoupling of the concrete types from the code that depends on these types. It generally uses a container that holds a list of registrations and mappings between interfaces and abstract types, and the concrete types that implement or extend these types.

There are also other types of dependency injection, such as *property setter injection*, and *method call injection*, but they are less commonly seen. Therefore, this chapter will focus solely on performing constructor injection with a dependency injection container.

Introduction to dependency injection

If a class doesn't directly instantiate the objects that it needs, another class must take on this responsibility. Factories, service locators, and dependency injection are all approaches for moving the responsibility for instantiating and managing objects to another class.

A common feature of factories and service locators is that it's an object's responsibility to resolve its own dependencies by requesting instances of the types that it needs. For example, Figure 3-1 shows the dependencies in the factory pattern, where the factory instantiates an `OrderService` object on behalf of the `ProfileViewModel` class.

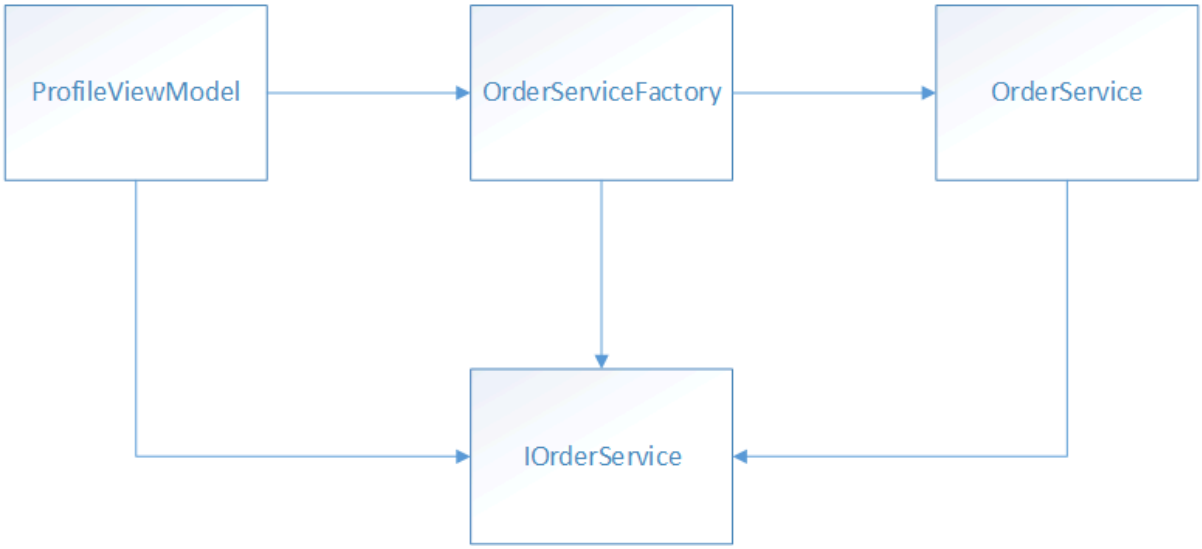


Figure 3-1: Dependencies in the factory pattern

In this example, the `ProfileViewModel` class relies on the `OrderServiceFactory` class to create the instance of the `OrderService` instance on its behalf. Therefore, the `ProfileViewModel` class has a dependency on the class that's responsible for creating the object it wants to use, as well as the `IOrderService` interface type. In addition, if each view model class in an app uses a factory class to create its dependencies, the dependencies will be hidden over multiple classes, making them harder to test.

Dependency injection is a specialized version of the Inversion of Control (IoC) pattern, where the concern being inverted is the process of obtaining the required dependency. With dependency injection, just like with the factory pattern, another class is responsible for injecting dependencies into an object at runtime. The following code example shows how the `ProfileViewModel` class is structured when using dependency injection:

```
public class ProfileViewModel : ViewModelBase
{
    private IOrderService _orderService;

    public ProfileViewModel(IOrderService orderService)
    {
        _orderService = orderService;
    }
    ...
}
```

The `ProfileViewModel` constructor receives an `IOrderService` instance as an argument, injected by another class. The only dependency in the `ProfileViewModel` class is on the interface type. Therefore, the `ProfileViewModel` class doesn't have any knowledge of the class that's responsible for instantiating the `IOrderService` object. The class that's responsible for instantiating the `IOrderService` object, and inserting it into the `ProfileViewModel` class, is known as the *dependency injection container*.

Dependency injection containers reduce the coupling between objects by providing a facility to instantiate class instances and manage their lifetime based on the configuration of the container. During the objects creation, the container injects any dependencies that the object requires into it. If those dependencies have not yet been created, the container creates and resolves their dependencies first.

Note: Dependency injection can also be implemented manually using factories. However, using a container provides additional capabilities such as lifetime management, and interception.

There are several advantages to using a dependency injection container:

- A container removes the need for a class to locate its dependencies and manage their lifetimes.
- A container allows mapping of implemented dependencies without affecting the class.
- A container facilitates testability by allowing dependencies to be mocked.
- A container increases maintainability by allowing new classes to be easily added to the app.

In the context of a Xamarin.Forms app that uses MVVM, a dependency injection container will typically be used for registering and resolving view models, and for registering services and injecting them into view models.

There are many dependency injection containers available, with the eShopOnContainers mobile app using Unity to manage the instantiation of view model and service classes in the app. Unity is a lightweight, extensible dependency injection container that facilitates building loosely coupled apps. It provides all of the features commonly found in dependency injection mechanisms, including methods to register type mappings and object instances, resolve objects, manage object lifetimes, and inject dependent objects into constructors of objects it resolves. For more information about Unity, see Developer's Guide to Dependency Injection using Unity on MSDN.

In Unity, the `UnityContainer` class provides the dependency injection container. Figure 3-2 shows the dependencies when using this container, which instantiates an `IOrderService` object and injects it into the `ProfileViewModel` class.

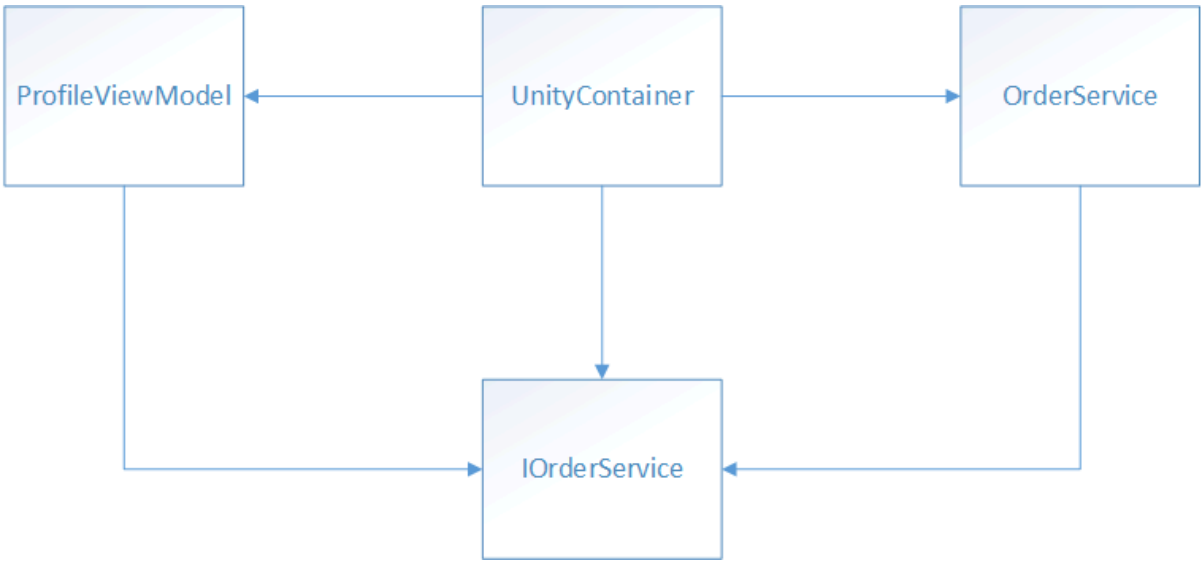


Figure 3-2: Dependencies when using dependency injection

At runtime, the container must know which implementation of the `IOrderService` interface it should instantiate, before it can instantiate a `ProfileViewModel` object. This involves:

- The container deciding how to instantiate an object that implements the `IOrderService` interface. This is known as *registration*.
- The container instantiating the object that implements the `IOrderService` interface, and the `ProfileViewModel` object. This is known as *resolution*.

Eventually, the app will finish using the `ProfileViewModel` object and it will become available for garbage collection. At this point, the garbage collector should dispose of the `IOrderService` instance if other classes do not share the same instance.

> **Tip:** Write container-agnostic code
>
> Always try to write container-agnostic code to decouple the app from the specific dependency container being used.

Registration

Before dependencies can be injected into an object, the types of the dependencies must first be registered with the container. Registering a type typically involves passing the container an interface and a concrete type that implements the interface.

There are two ways of registering types and objects in the container through code:

- Register a type or mapping with the container. When required, the container will build an instance of the specified type.
- Register an existing object in the container as a singleton. When required, the container will return a reference to the existing object.

> **Tip:** Dependency injection containers are not always suitable
>
> Dependency injection introduces additional complexity and requirements that may not be appropriate or useful to small apps.
>
> If a class does not have any dependencies, or is not a dependency for other types, it may not make sense to put it in the container.
>
> If a class has a single set of dependencies that are integral to the type and will never change, it may not make sense to put it in the container.

The registration of types that require dependency injection should be performed in a single method in an app, and this method should be invoked early in the app's lifecycle to ensure that the app is aware of the dependencies between its classes. In the eShopOnContainers mobile app this is performed by the `ViewModelLocator` class, which instantiates the `UnityContainer` object and is the only class in the app that holds a reference to that object. The following code example shows how the eShopOnContainers mobile app creates the Unity container in the `ViewModelLocator` class:

```
private static readonly IUnityContainer _unityContainer = new UnityContainer();
```

Types and instances are then registered in the `Initialize` and `UpdateDependencies` methods in the `ViewModelLocator` class.

The simplest type of registration is instance registration, where the container is responsible for maintaining a reference to a singleton instance of a type. For example, the following code example shows how the eShopOnContainers mobile app registers the concrete type to use when a `ProfileViewModel` instance requires an `IOrderService` instance:

```
_unityContainer.RegisterInstance<IOrderService>(new OrderService(requestProvider));
```

The `RegisterInstance` method shown here creates a new `OrderService` instance and registers it with the container. Therefore, only a single `OrderService` instance exists in the container, which is shared by objects that require an injection of an `IOrderService` through a constructor. The `requestProvider` variable is a resolved instance of a previously registered `RequestProvider` concrete type, which is shown in the following code example:

```
_unityContainer.RegisterType<IRequestProvider, RequestProvider>();
```

The `RegisterType` method shown here is the most common type registration, which maps an interface type to a concrete type. It tells the container to instantiate a `RequestProvider` object when it instantiates an object that requires an injection of an `IRequestProvider` through a constructor.

Concrete types can also be registered directly without a mapping from an interface type, as shown in the following code example:

```
_unityContainer.RegisterType<ProfileViewModel>();
```

When the `ProfileViewModel` type is resolved, the container will inject its required dependencies.

Resolution

After a type is registered, it can be resolved or injected as a dependency. When a type is being resolved and the container needs to create a new instance, it injects any dependencies into the instance.

Generally, when a type is resolved, one of three things happens:

1. If the type hasn't been registered, the container throws an exception.
2. If the type has been registered as a singleton, the container returns the singleton instance. If this is the first time the type was called for, the container creates it if required, and maintains a reference to it.
3. If the type has not been registered as a singleton, the container returns a new instance, and doesn't maintain a reference to it.

The following code example shows how the eShopOnContainers mobile app resolves the `RequestProvider` type that was previously registered with Unity:

```
var requestProvider = _unityContainer.Resolve<IRequestProvider>();
```

In this example, Unity is asked to resolve the concrete type for the `IRequestProvider` type, along with any dependencies. Typically, the `Resolve` method is called when an instance of a specific type is required. For information about controlling the lifetime of resolved objects, see Managing the lifetime of resolved objects.

The following code example shows how the eShopOnContainers mobile app instantiates view model types and their dependencies:

```
var viewModel = _unityContainer.Resolve(viewModelType);
```

In this example, Unity is asked to resolve the view model type for a requested view model, and the container will also resolve any dependencies. When resolving the `ProfileViewModel` type, the

dependency to resolve is an `IOrderService` object. Therefore, Unity first constructs an `OrderService` object and then passes it to the constructor of the `ProfileViewModel` class. For more information about how the eShopOnContainers mobile app constructs view models and associates them to views, see Automatically creating a view model with a view model locator.

> **Note:** Registering and resolving types with a container has a performance cost because of the container's use of reflection for creating each type, especially if dependencies are being reconstructed for each page navigation in the app. If there are many or deep dependencies, the cost of creation can increase significantly.

Managing the lifetime of resolved objects

After registering a type, the default behavior for Unity is to create a new instance of the registered type each time the type is resolved, or when the dependency mechanism injects instances into other classes. In this scenario, the container doesn't hold a reference to the resolved object. However, when registering an instance, the default behavior for Unity is to manage the lifetime of the object as a singleton. Therefore, the instance remains in scope while the container is in scope, and is disposed when the container goes out of scope and is garbage collected, or when code explicitly disposes the container.

A Unity *lifetime manager* can be used to specify the singleton behavior for an object that Unity creates from a registered type. Unity lifetime managers manage the object lifetimes instantiated by the container. The default lifetime manager for the `RegisterType` method is the `TransientLifetimeManager`, and the default lifetime manager for the `RegisterInstance` method is the `ContainerControllerLifetimeManager`. The `ContainerControlledLifetimeManager` class can be used with the `RegisterType` method, so that the container creates or returns a singleton instance of a type when calling the `Resolve` method. The following code example shows how Unity is instructed to create a singleton instance of the `NavigationService` class:

```
_unityContainer.RegisterType<INavigationService, NavigationService>(new ContainerControlledLifeti
meManager());
```

The first time that the `INavigationService` interface is resolved, the container creates a new `NavigationService` object and maintains a reference to it. On any subsequent resolutions of the `INavigationService` interface, the container returns a reference to the `NavigationService` object that was previously created.

> **Note:** The `ContainerControlledLifetimeManager` disposes created objects when the container is disposed.

Unity includes additional lifetime managers. For more information, see Lifetime Management on MSDN.

Summary

Dependency injection enables decoupling of concrete types from the code that depends on these types. It typically uses a container that holds a list of registrations and mappings between interfaces and abstract types, and the concrete types that implement or extend these types.

Unity is a lightweight, extensible dependency injection container that facilitates building loosely coupled apps. It provides the features commonly found in dependency injection mechanisms,

including methods to register type mappings and object instances, resolve objects, manage object lifetimes, and inject dependent objects into constructors of objects it resolves.

CHAPTER 4

Communicating between loosely coupled components

The publish-subscribe pattern is a messaging pattern where publishers send messages without having knowledge of any receivers, known as subscribers. Similarly, subscribers listen for specific messages, without having knowledge of any publishers.

Events in .NET implement the publish-subscribe pattern, and are the most simple and straightforward approach for a communication layer between components if loose coupling is not required, such as a control and the page that contains it. However, the publisher and subscriber lifetimes are coupled by object references to each other, and the subscriber type must have a reference to the publisher type. This can create memory management issues, especially when there are short lived objects that subscribe to an event of a static or long lived object. If the event handler isn't removed, the subscriber will be kept alive by the reference to it in the publisher, and this will prevent or delay the garbage collection of the subscriber.

Introduction to MessagingCenter

The Xamarin.Forms `MessagingCenter` class implements the publish-subscribe pattern, allowing message-based communication between components that are inconvenient to link by object and type references. This mechanism allows publishers and subscribers to communicate without having a reference to each other, helping to reduce dependencies between components, while also allowing components to be independently developed and tested.

The `MessagingCenter` class provides multicast publish-subscribe functionality. This means that there can be multiple publishers that publish a single message, and there can be multiple subscribers listening for the same message. Figure 4-1 illustrates this relationship:

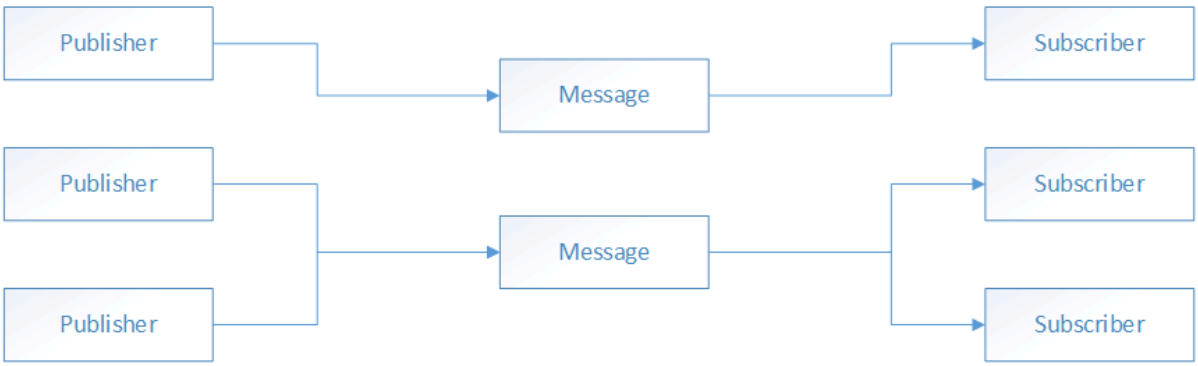


Figure 4-1: Multicast publish-subscribe functionality

Publishers send messages using the `MessagingCenter.Send` method, while subscribers listen for messages using the `MessagingCenter.Subscribe` method. In addition, subscribers can also unsubscribe from message subscriptions, if required, with the `MessagingCenter.Unsubscribe` method.

Internally, the `MessagingCenter` class uses weak references. This means that it will not keep objects alive, and will allow them to be garbage collected. Therefore, it should only be necessary to unsubscribe from a message when a class no longer wishes to receive it.

The eShopOnContainers mobile app uses the `MessagingCenter` class to communicate between loosely coupled components. The app defines three messages:

- The `AddProduct` message is published by the `CatalogViewModel` class when an item is added to the shopping basket. In return, the `BasketViewModel` class subscribes to the message and increments the number of items in the shopping basket in response. In addition, the `BasketViewModel` class also unsubscribes from this message.
- The `Filter` message is published by the `CatalogViewModel` class when the user applies a brand or type filter to the items displayed from the catalogue. In return, the `CatalogView` class subscribes to the message and updates the UI so that only items that match the filter criteria are displayed.
- The `ChangeTab` message is published by the `MainViewModel` class when the `CheckoutViewModel` navigates to the `MainViewModel` following the successful creation and submission of a new order. In return, the `MainView` class subscribes to the message and updates the UI so that the **My profile** tab is active, to show the user's orders.

> **Note:** While the `MessagingCenter` class permits communication between loosely-coupled classes, it does not offer the only architectural solution to this issue. For example, communication between a view model and a view can also be achieved by the binding engine and through property change notifications. In addition, communication between two view models can also be achieved by passing data during navigation.

In the eShopOnContainers mobile app, `MessagingCenter` is used to update in the UI in response to an action occurring in another class. Therefore, messages are published on the UI thread, with subscribers receiving the message on the same thread.

> **Tip:** Marshal to the UI thread when required

If a message that's sent from a background thread is required to update the UI, process the message on the UI thread in the subscriber by invoking the `Device.BeginInvokeOnMainThread` method.

For more information about `MessagingCenter`, see MessagingCenter on the Xamarin Developer Center.

Defining a message

`MessagingCenter` messages are strings that are used to identify messages. The following code example shows the messages defined within the eShopOnContainers mobile app:

```
public class MessengerKeys
{
    // Add product to basket
    public const string AddProduct = "AddProduct";

    // Filter
    public const string Filter = "Filter";

    // Change selected Tab programmatically
    public const string ChangeTab = "ChangeTab";
}
```

In this example, messages are defined using constants. The advantage of this approach is that it provides compile-time type safety and refactoring support.

Publishing a message

Publishers notify subscribers of a message with one the `MessagingCenter.Send` overloads. The following code example demonstrates publishing the `AddProduct` message:

```
MessagingCenter.Send(this, MessengerKeys.AddProduct, catalogItem);
```

In this example, the `Send` method specifies three arguments:

- The first argument specifies the sender class. The sender class must be specified by any subscribers who wish to receive the message.
- The second argument specifies the message.
- The third argument specifies the payload data to be sent to the subscriber. In this case the payload data is a `CatalogItem` instance.

The `Send` method will publish the message, and its payload data, using a fire-and-forget approach. Therefore, the message is sent even if there are no subscribers registered to receive the message. In this situation, the sent message is ignored.

Note: The `MessagingCenter.Send` method can use generic parameters to control how messages are delivered. Therefore, multiple messages that share a message identity but send different payload data types can be received by different subscribers.

Subscribing to a message

Subscribers can register to receive a message using one of the `MessagingCenter.Subscribe` overloads. The following code example demonstrates how the eShopOnContainers mobile app subscribes to and processes the `AddProduct` message:

```
MessagingCenter.Subscribe<CatalogViewModel, CatalogItem>(this, MessengerKeys.AddProduct, (sender,
 arg) =>
{
    BadgeCount++;
    AddCatalogItem(arg);
});
```

In this example, the `Subscribe` method subscribes to the `AddProduct` message, and executes a callback delegate in response to receiving the message. This callback delegate, specified as a lambda expression, executes code that updates the UI.

> **Tip:** Consider using immutable payload data
>
> Don't attempt to modify the payload data from within a callback delegate because several threads could be accessing the received data simultaneously. In this scenario, the payload data should be immutable to avoid concurrency errors.

A subscriber may not need to handle every instance of a published message, and this can be controlled by the generic type arguments that are specified on the `Subscribe` method. In this example, the subscriber will only receive `AddProduct` messages that are sent from the `CatalogViewModel` class, whose payload data is a `CatalogItem` instance.

Unsubscribing from a message

Subscribers can unsubscribe from messages they no longer want to receive. This is achieved with one of the `MessagingCenter.Unsubscribe` overloads, as demonstrated in the following code example:

```
MessagingCenter.Unsubscribe<CatalogViewModel, CatalogItem>(this, MessengerKeys.AddProduct);
```

In this example, the `Unsubscribe` method syntax reflects the type arguments specified when subscribing to receive the `AddProduct` message.

Summary

The Xamarin.Forms `MessagingCenter` class implements the publish-subscribe pattern, allowing message-based communication between components that are inconvenient to link by object and type references. This mechanism allows publishers and subscribers to communicate without having a reference to each other, helping to reduce dependencies between components, while also allowing components to be independently developed and tested.

Navigation

Xamarin.Forms includes support for page navigation, which typically results from the user's interaction with the UI or from the app itself as a result of internal logic-driven state changes. However, navigation can be complex to implement in apps that use the Model-View-ViewModel (MVVM) pattern, as the following challenges need addressing:

- How to identify the view to be navigated to, using an approach that does not introduce tight coupling and dependencies between views.
- How to coordinate the process by which the view to be navigated to is instantiated and initialized. When using MVVM, the view and view model need to be instantiated and associated with each other via the view's binding context. When an app is using a dependency injection container, the instantiation of views and view models may require a specific construction mechanism.
- Whether to perform view-first navigation, or view model-first navigation. With view-first navigation, the page to navigate to refers to the name of the view type. During navigation, the specified view is instantiated, along with its corresponding view model and other dependent services. An alternative approach is to use view model-first navigation, where the page to navigate to refers to the name of the view model type.
- How to cleanly separate the navigational behavior of the app across the views and view models. The MVVM pattern provides a separation between the app's UI and its presentation and business logic. However, the navigation behavior of an app will often span the UI and presentations parts of the app. The user will often initiate navigation from a view, and the view will be replaced as a result of the navigation. However, navigation may often also need to be initiated or coordinated from within the view model.
- How to pass parameters during navigation, for initialization purposes. For example, if the user navigates to a view to update order details, the order data will have to be passed to the view so that it can display the correct data.
- How to co-ordinate navigation, to ensure that certain business rules are obeyed. For example, users may be prompted before navigating away from a view so that they can correct any invalid data or be prompted to submit or discard any data changes that were made within the view.

This chapter addresses these challenges by presenting a `NavigationService` class that's used to perform view model-first page navigation.

> **Note:** The `NavigationService` used by the app is designed only to perform hierarchical navigation between `ContentPage` instances. Using the service to navigate between other page types may result in unexpected behavior.

Navigating between pages

Navigation logic can reside in a view's code-behind, or in a data bound view model. While placing navigation logic in a view may be the simplest approach, it is not easily testable through automated tests. Placing navigation logic in view model classes means that the logic can be exercised through automated tests. In addition, the view model can then implement logic to control navigation to ensure that certain business rules are enforced. For example, an app may not allow the user to navigate away from a page without first ensuring that the entered data is valid.

A `NavigationService` class is typically invoked from view models, in order to promote testability. However, navigating to views from view models would require the view models to reference views, and particularly views that the active view model isn't associated with, which is not recommended. Therefore, the `NavigationService` presented here specifies the view model type as the target to navigate to.

The eShopOnContainers mobile app uses the `NavigationService` class to provide view model-first navigation. This class implements the `INavigationService` interface, which is shown in the following code example:

```
public interface INavigationService
{
    Task InitializeAsync();
    Task NavigateToAsync<TViewModel>() where TViewModel : ViewModelBase;
    Task NavigateToAsync<TViewModel>(object parameter) where TViewModel : ViewModelBase;
    Task RemoveLastFromBackStackAsync();
    Task RemoveBackStackAsync();
}
```

This interface specifies that an implementing class must provide the following methods:

Method	Purpose
`InitializeAsync`	Performs navigation to one of two pages when the app is launched.
`NavigateToAsync<T>`	Performs hierarchical navigation to a specified page.
`NavigateToAsync<T>(parameter)`	Performs hierarchical navigation to a specified page, passing a parameter.
`RemoveLastFromBackStackAsync`	Removes the previous page from the navigation stack.
`RemoveBackStackAsync`	Removes all the previous pages from the navigation stack.

Note: An `INavigationService` interface would usually also specify a `GoBackAsync` method, which is used to programmatically return to the previous page in the navigation stack. However, this method is missing from the eShopOnContainers mobile app as it's not required.

Creating the NavigationService instance

The `NavigationService` class, which implements the `INavigationService` interface, is registered as a singleton with the Unity dependency injection container, as demonstrated in the following code example:

```
_unityContainer.RegisterType<INavigationService, NavigationService>(new ContainerControlledLifeti
meManager());
```

The `INavigationService` interface is resolved in the `ViewModelBase` class constructor, as demonstrated in the following code example:

```
NavigationService = ViewModelLocator.Resolve<INavigationService>();
```

This returns a reference to the `NavigationService` object that's stored in the Unity dependency injection container, which is created by the `InitNavigation` method in the `App` class. For more information, see Navigating when the app is launched.

The `ViewModelBase` class stores the `NavigationService` instance in a `NavigationService` property, of type `INavigationService`. Therefore, all view model classes, which derive from the `ViewModelBase` class, can use the `NavigationService` property to access the methods specified by the `INavigationService` interface. This avoids the overhead of injecting the `NavigationService` object from the Unity dependency injection container into each view model class.

Handling navigation requests

Xamarin.Forms provides the `NavigationPage` class, which implements a hierarchical navigation experience where the user is able to navigate through pages, forwards and backwards, as desired. For more information about hierarchical navigation, see Hierarchical Navigation on the Xamarin Developer Center.

Rather than use the `NavigationPage` class directly, the eShopOnContainers app wraps the `NavigationPage` class in the `CustomNavigationView` class, as shown in the following code example:

```
public partial class CustomNavigationView : NavigationPage
{
    public CustomNavigationView() : base()
    {
        InitializeComponent();
    }

    public CustomNavigationView(Page root) : base(root)
    {
        InitializeComponent();
    }
}
```

The purpose of this wrapping is for ease of styling the `NavigationPage` instance, inside the XAML file for the class.

Navigation is performed inside view model classes by invoking one of the `NavigateToAsync` methods, specifying the view model type for the page being navigated to, as demonstrated in the following code example:

```
await NavigationService.NavigateToAsync<MainViewModel>();
```

The following code example shows the `NavigateToAsync` methods provided by the `NavigationService` class:

```
public Task NavigateToAsync<TViewModel>() where TViewModel : ViewModelBase
{
    return InternalNavigateToAsync(typeof(TViewModel), null);
}

public Task NavigateToAsync<TViewModel>(object parameter) where TViewModel : ViewModelBase
```

```
{
    return InternalNavigateToAsync(typeof(TViewModel), parameter);
}
```

Each method allows any view model class that derives from the `ViewModelBase` class to perform hierarchical navigation, by invoking the `InternalNavigateToAsync` method. In addition, the second `NavigateToAsync` method enables navigation data to be specified as an argument that's passed to the view model being navigated to, where it's typically used to perform initialization. For more information, see Passing parameters during navigation.

The `InternalNavigateToAsync` method executes the navigation request, and is shown in the following code example:

```
private async Task InternalNavigateToAsync(Type viewModelType, object parameter)
{
    Page page = CreatePage(viewModelType, parameter);

    if (page is LoginView)
    {
        Application.Current.MainPage = new CustomNavigationView(page);
    }
    else
    {
        var navigationPage = Application.Current.MainPage as CustomNavigationView;
        if (navigationPage != null)
        {
            await navigationPage.PushAsync(page);
        }
        else
        {
            Application.Current.MainPage = new CustomNavigationView(page);
        }
    }

    await (page.BindingContext as ViewModelBase).InitializeAsync(parameter);
}

private Type GetPageTypeForViewModel(Type viewModelType)
{
    var viewName = viewModelType.FullName.Replace("Model", string.Empty);
    var viewModelAssemblyName = viewModelType.GetTypeInfo().Assembly.FullName;
    var viewAssemblyName = string.Format(CultureInfo.InvariantCulture, "{0}, {1}", viewName, view
ModelAssemblyName);
    var viewType = Type.GetType(viewAssemblyName);
    return viewType;
}

private Page CreatePage(Type viewModelType, object parameter)
{
    Type pageType = GetPageTypeForViewModel(viewModelType);
    if (pageType == null)
    {
        throw new Exception($"Cannot locate page type for {viewModelType}");
    }

    Page page = Activator.CreateInstance(pageType) as Page;
    return page;
}
```

The `InternalNavigateToAsync` method performs navigation to a view model by first calling the `CreatePage` method. This method locates the view that corresponds to the specified view model type,

and creates and returns an instance of this view type. Locating the view that corresponds to the view model type uses a convention-based approach, which assumes that:

- Views are in the same assembly as view model types.
- Views are in a .Views child namespace.
- View models are in a .ViewModels child namespace.
- View names correspond to view model names, with "Model" removed.

When a view is instantiated, it's associated with its corresponding view model. For more information about how this occurs, see Automatically creating a view model with a view model locator.

If the view being created is a `LoginView`, it's wrapped inside a new instance of the `CustomNavigationView` class and assigned to the `Application.Current.MainPage` property. Otherwise, the `CustomNavigationView` instance is retrieved, and provided that it isn't `null`, the `PushAsync` method is invoked to push the view being created onto the navigation stack. However, If the retrieved `CustomNavigationView` instance is `null`, the view being created is wrapped inside a new instance of the `CustomNavigationView` class and assigned to the `Application.Current.MainPage` property. This mechanism ensures that during navigation, pages are added correctly to the navigation stack both when it's empty, and when it contains data.

> **Tip:** Consider caching pages
>
> Page caching results in memory consumption for views that are not currently displayed. However, without page caching it does mean that XAML parsing and construction of the page and its view model will occur every time a new page is navigated to, which can have a performance impact for a complex page. For a well-designed page that does not use an excessive number of controls, the performance should be sufficient. However, page caching may help if slow page loading times are encountered.

After the view is created and navigated to, the `InitializeAsync` method of the view's associated view model is executed. For more information, see Passing parameters during navigation.

Navigating when the app is launched

When the app is launched, the `InitNavigation` method in the `App` class is invoked. The following code example shows this method:

```
private Task InitNavigation()
{
    var navigationService = ViewModelLocator.Resolve<INavigationService>();
    return navigationService.InitializeAsync();
}
```

The method creates a new `NavigationService` object in the Unity dependency injection container, and returns a reference to it, before invoking its `InitializeAsync` method.

> **Note:** When the `INavigationService` interface is resolved by the `ViewModelBase` class, the container returns a reference to the `NavigationService` object that was created when the `InitNavigation` method is invoked.

The following code example shows the `NavigationService` `InitializeAsync` method:

```
public Task InitializeAsync()
{
```

```
    if(string.IsNullOrEmpty(Settings.AuthAccessToken))
        return NavigateToAsync<LoginViewModel>();
    else
        return NavigateToAsync<MainViewModel>();
}
```

The `MainView` is navigated to if the app has a cached access token, which is used for authentication. Otherwise, the `LoginView` is navigated to.

For more information about the Unity dependency injection container, see Introduction to dependency injection.

Passing parameters during navigation

One of the `NavigateToAsync` methods, specified by the `INavigationService` interface, enables navigation data to be specified as an argument that's passed to the view model being navigated to, where it's typically used to perform initialization.

For example, the `ProfileViewModel` class contains an `OrderDetailCommand` that's executed when the user selects an order on the `ProfileView` page. In turn, this executes the `OrderDetailAsync` method, which is shown in the following code example:

```
private async Task OrderDetailAsync(Order order)
{
    await NavigationService.NavigateToAsync<OrderDetailViewModel>(order);
}
```

This method invokes navigation to the `OrderDetailViewModel`, passing an `Order` instance that represents the order the user selected on the `ProfileView` page. When the `NavigationService` class creates the `OrderDetailView`, the `OrderDetailViewModel` class is instantiated and assigned to the view's `BindingContext`. After navigating to the `OrderDetailView`, the `InternalNavigateToAsync` method executes the `InitializeAsync` method of the view's associated view model.

The `InitializeAsync` method is defined in the `ViewModelBase` class as an overridable method. This method specifies an `object` argument that represents the data to be passed to a view model during a navigation operation. Therefore, view model classes that want to receive data from a navigation operation provide their own implementation of the `InitializeAsync` method, to perform the required initialization. The following code example shows the `InitializeAsync` method from the `OrderDetailViewModel` class:

```
public override async Task InitializeAsync(object navigationData)
{
    if (navigationData is Order)
    {
        ...
        Order = await _ordersService.GetOrderAsync(Convert.ToInt32(order.OrderNumber), authToken)
;
        ...
    }
}
```

This method retrieves the `Order` instance that was passed into the view model during the navigation operation, and uses it to retrieve the full order details from the `OrderService` instance.

Invoking navigation using behaviors

Navigation is usually triggered from a view by a user interaction. For example, the `LoginView` performs navigation following successful authentication. The following code example shows how the navigation is invoked by a behavior:

```
<WebView ...>
    <WebView.Behaviors>
        <behaviors:EventToCommandBehavior
            EventName="Navigating"
            EventArgsConverter="{StaticResource WebNavigatingEventArgsConverter}"
            Command="{Binding NavigateCommand}" />
    </WebView.Behaviors>
</WebView>
```

At runtime, the `EventToCommandBehavior` will respond to interaction with the `WebView`. When the `WebView` navigates to a web page, the `Navigating` event will fire, which will execute the `NavigateCommand` in the `LoginViewModel`. By default, the event arguments for the event are passed to the command. This data is converted as it's passed between source and target by the converter specified in the `EventArgsConverter` property, which returns the `Url` from the `WebNavigatingEventArgs`. Therefore, when the `NavigationCommand` is executed, the `Url` of the web page is passed as a parameter to the registered `Action`.

In turn, the `NavigationCommand` executes the `NavigateAsync` method, which is shown in the following code example:

```
private async Task NavigateAsync(string url)
{
    ...
    await NavigationService.NavigateToAsync<MainViewModel>();
    await NavigationService.RemoveLastFromBackStackAsync();
    ...
}
```

This method invokes navigation to the `MainViewModel`, and following navigation, removes the `LoginView` page from the navigation stack.

Confirming or cancelling navigation

An app may need to interact with the user during a navigation operation, so that the user can confirm or cancel navigation. This may be necessary, for example, when the user attempts to navigate before having fully completed a data entry page. In this situation, an app should provide a notification that allows the user to navigate away from the page, or cancel the navigation operation before it occurs. This can be achieved in a view model class by using the response from a notification to control whether navigation is invoked or not.

Summary

Xamarin.Forms includes support for page navigation, which typically results from the user's interaction with the UI or from the app itself as a result of internal logic-driven state changes. However, navigation can be complex to implement in apps that use the Model-View-ViewModel (MVVM) pattern.

This chapter presented a `NavigationService` class, which is used to perform view model-first navigation from view models. Placing navigation logic in view model classes means that the logic can

be exercised through automated tests. In addition, the view model can then implement logic to control navigation to ensure that certain business rules are enforced.

Validation

Any app that accepts input from users should ensure that the input is valid. An app could, for example, check that the input contains only characters in a particular range, is of a certain length, or matches a particular format. Without validation, a user can supply data that causes the app to fail. Validation enforces business rules, and prevents an attacker from injecting malicious data.

In the context of the Model-ViewModel-Model (MVVM) pattern, a view model or model will often be required to perform data validation and signal any validation errors to the view so that the user can correct them. The eShopOnContainers mobile app performs synchronous client-side validation of view model properties and notifies the user of any validation errors by highlighting the control that contains the invalid data, and by displaying error messages that inform the user why the data is invalid. Figure 6-1 shows the classes involved in performing validation in the eShopOnContainers mobile app.

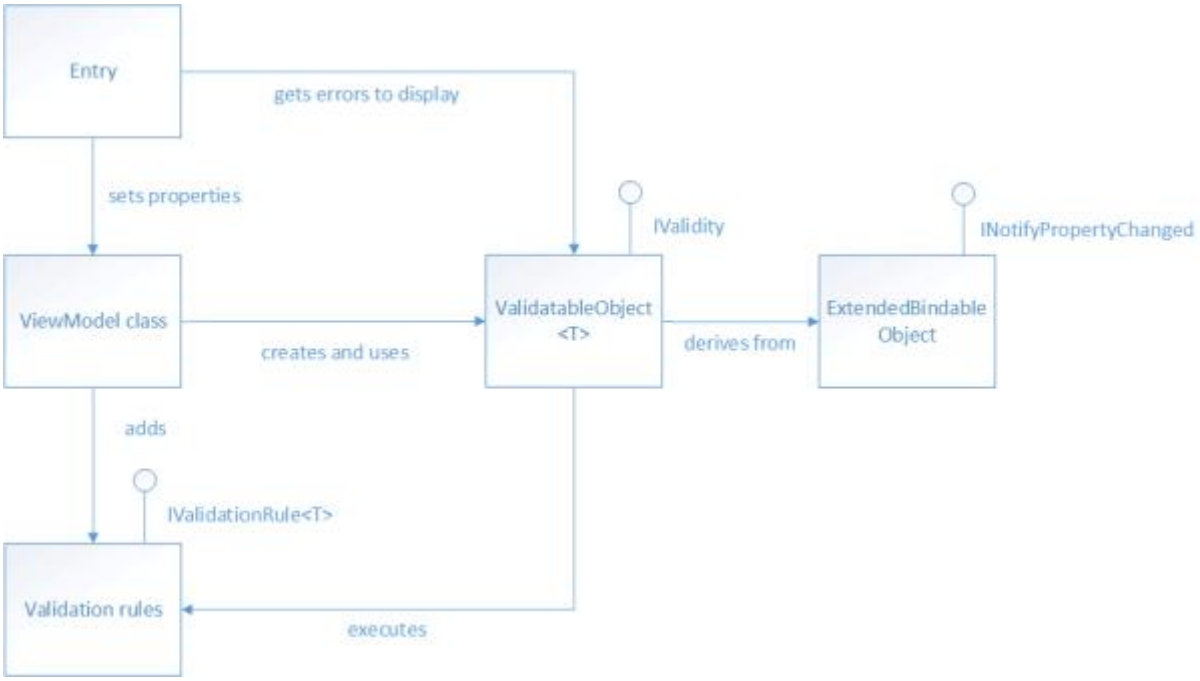


Figure 6-1: Validation classes in the eShopOnContainers mobile app

View model properties that require validation are of type `ValidatableObject<T>`, and each `ValidatableObject<T>` instance has validation rules added to its `Validations` property. Validation is invoked from the view model by calling the `Validate` method of the `ValidatableObject<T>` instance, which retrieves the validation rules and executes them against the `ValidatableObject<T>` `Value` property. Any validation errors are placed into the `Errors` property of the `ValidatableObject<T>` instance, and the `IsValid` property of the `ValidatableObject<T>` instance is updated to indicate whether validation succeeded or failed.

Property change notification is provided by the `ExtendedBindableObject` class, and so an `Entry` control can bind to the `IsValid` property of `ValidatableObject<T>` instance in the view model class to be notified of whether the entered data is valid or not.

Specifying validation rules

Validation rules are specified by creating a class that derives from the `IValidationRule<T>` interface, which is shown in the following code example:

```
public interface IValidationRule<T>
{
    string ValidationMessage { get; set; }
    bool Check(T value);
}
```

This interface specifies that a validation rule class must provide a `boolean Check` method, that is used to perform the required validation, and a `ValidationMessage` property, whose value is the validation error message that will be displayed if validation fails.

The following code example shows the `IsNotNullOrEmptyRule<T>` validation rule, which is used to perform validation of the username and password entered by the user on the `LoginView`, when using mock services in the eShopOnContainers mobile app:

```
public class IsNotNullOrEmptyRule<T> : IValidationRule<T>
{
    public string ValidationMessage { get; set; }

    public bool Check(T value)
    {
        if (value == null)
        {
            return false;
        }

        var str = value as string;
        return !string.IsNullOrWhiteSpace(str);
    }
}
```

The `Check` method returns a `boolean` indicating whether the `value` argument is `null`, empty, or consists only of whitespace characters.

Although not used by the eShopOnContainers mobile app, the following code example shows a validation rule for validating email addresses:

```
public class EmailRule<T> : IValidationRule<T>
{
    public string ValidationMessage { get; set; }

    public bool Check(T value)
    {
        if (value == null)
        {
            return false;
        }

        var str = value as string;
        Regex regex = new Regex(@"^([\w\.\-]+)@([\w\-]+)((\.(\w){2,3})+)$");
```

```
        Match match = regex.Match(str);

        return match.Success;
    }
}
```

The `Check` method returns a `boolean` indicating whether the `value` argument is a valid email address or not. This is achieved by searching the `value` argument for the first occurrence of the regular expression pattern specified in the `Regex` constructor. Whether the regular expression pattern has been found in the input string can be determined by checking the value of the `Match` object's `Success` property.

> **Note:** Property validation can sometimes involve dependent properties. An example of dependent properties occurs when the set of valid values for property A depends on the particular value that has been set in property B. To check that the value of property A is one of the allowed values would involve retrieving the value of property B. In addition, when the value of property B changes, property A would need to be revalidated.

Adding validation rules to a property

In the eShopOnContainers mobile app, view model properties that require validation are declared to be of type `ValidatableObject<T>`, where `T` is the type of the data to be validated. The following code example shows an example of two such properties:

```
public ValidatableObject<string> UserName
{
    get
    {
        return _userName;
    }
    set
    {
        _userName = value;
        RaisePropertyChanged(() => UserName);
    }
}

public ValidatableObject<string> Password
{
    get
    {
        return _password;
    }
    set
    {
        _password = value;
        RaisePropertyChanged(() => Password);
    }
}
```

For validation to occur, validation rules must be added to the `Validations` collection of each `ValidatableObject<T>` instance, as demonstrated in the following code example:

```
private void AddValidations()
{
    _userName.Validations.Add(new IsNotNullOrEmptyRule<string>
    {
```

```
            ValidationMessage = "A username is required."
        });
        _password.Validations.Add(new IsNotNullOrEmptyRule<string>
        {
            ValidationMessage = "A password is required."
        });
    }
```

This method adds the `IsNotNullOrEmptyRule<T>` validation rule to the `Validations` collection of each `ValidatableObject<T>` instance, specifying values for the validation rule's `ValidationMessage` property, which specifies the validation error message that will be displayed if validation fails.

Triggering validation

The validation approach used in the eShopOnContainers mobile app can manually trigger validation of a property, and automatically trigger validation when a property changes.

Triggering validation manually

Validation can be triggered manually for a view model property. For example, this occurs in the eShopOnContainers mobile app when the user taps the **Login** button on the `LoginView`, when using mock services. The command delegate calls the `MockSignInAsync` method in the `LoginViewModel`, which invokes validation by executing the `Validate` method, which is shown in the following code example:

```
private bool Validate()
{
    bool isValidUser = ValidateUserName();
    bool isValidPassword = ValidatePassword();
    return isValidUser && isValidPassword;
}

private bool ValidateUserName()
{
    return _userName.Validate();
}

private bool ValidatePassword()
{
    return _password.Validate();
}
```

The `Validate` method performs validation of the username and password entered by the user on the `LoginView`, by invoking the `Validate` method on each `ValidatableObject<T>` instance. The following code example shows the `Validate` method from the `ValidatableObject<T>` class:

```
public bool Validate()
{
    Errors.Clear();

    IEnumerable<string> errors = _validations
        .Where(v => !v.Check(Value))
        .Select(v => v.ValidationMessage);

    Errors = errors.ToList();
    IsValid = !Errors.Any();
```

```
    return this.IsValid;
}
```

This method clears the `Errors` collection, and then retrieves any validation rules that were added to the object's `Validations` collection. The `Check` method for each retrieved validation rule is executed, and the `ValidationMessage` property value for any validation rule that fails to validate the data is added to the `Errors` collection of the `ValidatableObject<T>` instance. Finally, the `IsValid` property is set, and its value is returned to the calling method, indicating whether validation succeeded or failed.

Triggering validation when properties change

Validation is also automatically triggered whenever a bound property changes. For example, when a two-way binding in the `LoginView` sets the `UserName` or `Password` property, validation is triggered. The following code example demonstrates how this occurs:

```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    <Entry.Behaviors>
        <behaviors:EventToCommandBehavior
            EventName="TextChanged"
            Command="{Binding ValidateUserNameCommand}" />
    </Entry.Behaviors>
    ...
</Entry>
```

The `Entry` control binds to the `UserName.Value` property of the `ValidatableObject<T>` instance, and the control's `Behaviors` collection has an `EventToCommandBehavior` instance added to it. This behavior executes the `ValidateUserNameCommand` in response to the `TextChanged` event firing on the `Entry`, which is raised when the text in the `Entry` changes. In turn, the `ValidateUserNameCommand` delegate executes the `ValidateUserName` method, which executes the `Validate` method on the `ValidatableObject<T>` instance. Therefore, every time the user enters a character in the `Entry` control for the username, validation of the entered data is performed.

For more information about behaviors, see Implementing behaviors.

Displaying validation errors

The eShopOnContainers mobile app notifies the user of any validation errors by highlighting the control that contains the invalid data with a red line, and by displaying an error message that informs the user why the data is invalid below the control containing the invalid data. When the invalid data is corrected, the line changes to black and the error message is removed. Figure 6-2 shows the `LoginView` in the eShopOnContainers mobile app when validation errors are present.

Figure 6-2: Displaying validation errors during login

Highlighting a control that contains invalid data

The `LineColorBehavior` attached behavior is used to highlight `Entry` controls where validation errors have occurred. The following code example shows how the `LineColorBehavior` attached behavior is attached to an `Entry` control:

```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    <Entry.Style>
        <OnPlatform x:TypeArguments="Style"
          iOS="{StaticResource EntryStyle}"
          Android="{StaticResource EntryStyle}"
          WinPhone="{StaticResource UwpEntryStyle}"/>
    </Entry.Style>
    ...
</Entry>
```

The `Entry` control consumes an explicit style, which is shown in the following code example:

```
<Style x:Key="EntryStyle"
       TargetType="{x:Type Entry}">
    ...
    <Setter Property="behaviors:LineColorBehavior.ApplyLineColor"
            Value="True" />
    <Setter Property="behaviors:LineColorBehavior.LineColor"
            Value="{StaticResource BlackColor}" />
    ...
</Style>
```

This style sets the `ApplyLineColor` and `LineColor` attached properties of the `LineColorBehavior` attached behavior on the `Entry` control. For more information about styles, see Styles on the Xamarin Developer Center.

When the value of the `ApplyLineColor` attached property is set, or changes, the `LineColorBehavior` attached behavior executes the `OnApplyLineColorChanged` method, which is shown in the following code example:

```
public static class LineColorBehavior
{
    ...
    private static void OnApplyLineColorChanged(
                BindableObject bindable, object oldValue, object newValue)
```

```
    {
        var view = bindable as View;
        if (view == null)
        {
            return;
        }

        bool hasLine = (bool)newValue;
        if (hasLine)
        {
            view.Effects.Add(new EntryLineColorEffect());
        }
        else
        {
            var entryLineColorEffectToRemove =
                    view.Effects.FirstOrDefault(e => e is EntryLineColorEffect);
            if (entryLineColorEffectToRemove != null)
            {
                view.Effects.Remove(entryLineColorEffectToRemove);
            }
        }
    }
}
```

The parameters for this method provide the instance of the control that the behavior is attached to, and the old and new values of the `ApplyLineColor` attached property. The `EntryLineColorEffect` class is added to the control's `Effects` collection if the `ApplyLineColor` attached property is `true`, otherwise it's removed from the control's `Effects` collection. For more information about behaviors, see Implementing behaviors.

The `EntryLineColorEffect` subclasses the `RoutingEffect` class, and is shown in the following code example:

```
public class EntryLineColorEffect : RoutingEffect
{
    public EntryLineColorEffect() : base("eShopOnContainers.EntryLineColorEffect")
    {
    }
}
```

The `RoutingEffect` class represents a platform-independent effect that wraps an inner effect that's platform-specific. This simplifies the effect removal process, since there is no compile-time access to the type information for a platform-specific effect. The `EntryLineColorEffect` calls the base class constructor, passing in a parameter consisting of a concatenation of the resolution group name, and the unique ID that's specified on each platform-specific effect class.

The following code example shows the eShopOnContainers.EntryLineColorEffect implementation for iOS:

```
[assembly: ResolutionGroupName("eShopOnContainers")]
[assembly: ExportEffect(typeof(EntryLineColorEffect), "EntryLineColorEffect")]
namespace eShopOnContainers.iOS.Effects
{
    public class EntryLineColorEffect : PlatformEffect
    {
        UITextField control;

        protected override void OnAttached()
        {
```

```
            try
            {
                control = Control as UITextField;
                UpdateLineColor();
            }
            catch (Exception ex)
            {
                Console.WriteLine("Can't set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached()
        {
            control = null;
        }

        protected override void OnElementPropertyChanged(PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(args);

            if (args.PropertyName == LineColorBehavior.LineColorProperty.PropertyName ||
                args.PropertyName == "Height")
            {
                Initialize();
                UpdateLineColor();
            }
        }

        private void Initialize()
        {
            var entry = Element as Entry;
            if (entry != null)
            {
                Control.Bounds = new CGRect(0, 0, entry.Width, entry.Height);
            }
        }

        private void UpdateLineColor()
        {
            BorderLineLayer lineLayer = control.Layer.Sublayers.OfType<BorderLineLayer>()
                                                              .FirstOrDefault();

            if (lineLayer == null)
            {
                lineLayer = new BorderLineLayer();
                lineLayer.MasksToBounds = true;
                lineLayer.BorderWidth = 1.0f;
                control.Layer.AddSublayer(lineLayer);
                control.BorderStyle = UITextBorderStyle.None;
            }

            lineLayer.Frame = new CGRect(0f, Control.Frame.Height-1f, Control.Bounds.Width, 1f);
            lineLayer.BorderColor = LineColorBehavior.GetLineColor(Element).ToCGColor();
            control.TintColor = control.TextColor;
        }

        private class BorderLineLayer : CALayer
        {
        }
    }
}
```

The `OnAttached` method retrieves the native control for the Xamarin.Forms `Entry` control, and updates the line color by calling the `UpdateLineColor` method. The `OnElementPropertyChanged` override responds to bindable property changes on the `Entry` control by updating the line color if the attached `LineColor` property changes, or the `Height` property of the `Entry` changes. For more information about effects, see Effects on the Xamarin Developer Center.

When valid data is entered in the `Entry` control, it will apply a black line to the bottom of the control, to indicate that there is no validation error. Figure 6-3 shows an example of this.

Figure 6-3: Black line indicating no validation error

The `Entry` control also has a `DataTrigger` added to its `Triggers` collection. The following code example shows the `DataTrigger`:

```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    ...
    <Entry.Triggers>
        <DataTrigger
            TargetType="Entry"
            Binding="{Binding UserName.IsValid}"
            Value="False">
            <Setter Property="behaviors:LineColorBehavior.LineColor"
                    Value="{StaticResource ErrorColor}" />
        </DataTrigger>
    </Entry.Triggers>
</Entry>
```

This `DataTrigger` monitors the `UserName.IsValid` property, and if it's value becomes `false`, it executes the `Setter`, which changes the `LineColor` attached property of the `LineColorBehavior` attached behavior to red. Figure 6-4 shows an example of this.

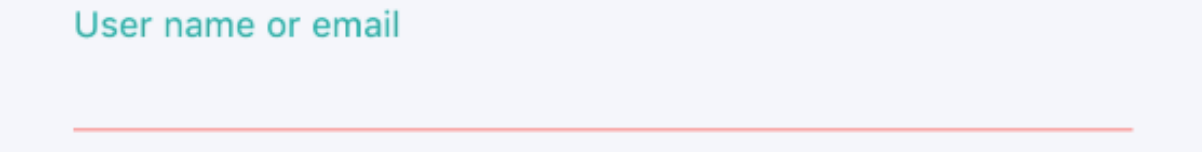


Figure 6-4: Red line indicating validation error

The line in the `Entry` control will remain red while the entered data is invalid, otherwise it will change to black to indicate that the entered data is valid.

For more information about Triggers, see Triggers on the Xamarin Developer Center.

Displaying error messages

The UI displays validation error messages in `Label` controls below each control whose data failed validation. The following code example shows the `Label` that displays a validation error message if the user has not entered a valid username:

```
<Label Text="{Binding UserName.Errors, Converter={StaticResource FirstValidationErrorConverter}"
       Style="{StaticResource ValidationErrorLabelStyle}" />
```

Each `Label` binds to the Errors property of the view model object that's being validated. The `Errors` property is provided by the `ValidatableObject<T>` class, and is of type `List<string>`. Because the `Errors` property can contain multiple validation errors, the `FirstValidationErrorConverter` instance is used to retrieve the first error from the collection, for display.

Summary

The eShopOnContainers mobile app performs synchronous client-side validation of view model properties and notifies the user of any validation errors by highlighting the control that contains the invalid data, and by displaying error messages that inform the user why the data is invalid.

View model properties that require validation are of type `ValidatableObject<T>`, and each `ValidatableObject<T>` instance has validation rules added to its `Validations` property. Validation is invoked from the view model by calling the `Validate` method of the `ValidatableObject<T>` instance, which retrieves the validation rules and executes them against the `ValidatableObject<T>` `Value` property. Any validation errors are placed into the `Errors` property of the `ValidatableObject<T>` instance, and the `IsValid` property of the `ValidatableObject<T>` instance is updated to indicate whether validation succeeded or failed.

CHAPTER 7

Configuration Management

CHAPTER 8

Containerized Microservices

CHAPTER 9

Authentication and Authorization

CHAPTER 10

Accessing Data

CHAPTER 11

Testing