Microsoft

Enterprise Application Patterns using Xamarin.Forms

David Britch

PUBLISHED BY

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

Author: David Britch
Developer: Javier Suarez Ruiz (Plain Concepts)
Participants and reviewers: Craig Dunn, Tom Opgenorth
Editor: John Meade (Populus Group)

Contents

Preface ..... **iv**
- **Purpose** ..... **iv**
  - What's left out of this guide's scope iv
  - Who should use this guide iv
  - How to use this guide v

Introduction **1**
- **Sample application** **2**
  - Sample application architecture 2
  - Mobile app 4
  - **eShopOnContainers.Core project** 5
  - **Platform projects** 6
- **Summary** **6**

MVVM **7**
- **The MVVM pattern** **7**
  - View 8
  - ViewModel 8
  - Model 9
- **Connecting view models to views** **9**
  - Creating a view model declaratively 10
  - Creating a view model programmatically 10
  - Creating a view defined as a data template 11
  - Automatically creating a view model with a view model locator 11
- **Updating views in response to changes in the underlying view model or model** **12**
- **UI interaction using commands and behaviors** **13**
  - Implementing commands 14
  - Implementing behaviors 15
- **Summary** **17**

Dependency injection **18**
- **Introduction to dependency injection** **18**
- **Registration** **20**
- **Resolution** **22**
- **Managing the lifetime of resolved objects** **22**
- **Summary** **23**

Communicating between loosely coupled components 24
Introduction to MessagingCenter 24
Defining a message 26
Publishing a message 26
Subscribing to a message 27
Unsubscribing from a message 27
Summary 27

Navigation 28
Navigating between pages 29
Creating the NavigationService instance 29
Handling navigation requests 30
Navigating when the app is launched 32
Passing parameters during navigation 33
Invoking navigation using behaviors 34
Confirming or cancelling navigation 34
Summary 35

Validation 36
Specifying validation rules 37
Adding validation rules to a property 38
Triggering validation 39
Triggering validation manually 39
Triggering validation when properties change 40
Displaying validation errors 40
Highlighting a control that contains invalid data 41
Displaying error messages 44
Summary 45

Configuration management 46
Creating a settings class 46
Adding a setting 47
Data binding to user settings 48
Summary 50

Containerized microservices 51
Microservices 52
Containerization 53
Communication between client and microservices 55
Communication between microservices 56
Summary 58

Authentication and authorization 59
Authentication 59
Issuing bearer tokens using IdentityServer 4 60
Adding IdentityServer to a web application 60
Configuring IdentityServer 61

Performing authentication 64
Authorization 69
Configuring IdentityServer to perform authorization 70
Making access requests to APIs 71
Summary 71

Accessing remote data 73
Introduction to Representational State Transfer 73
Consuming RESTful APIs 74
Making web requests 74
Caching data 81
Managing data expiration 82
Caching images 82
Increasing resilience 83
Retry pattern 83
Circuit breaker pattern 84
Summary 85

Unit testing 86
Dependency injection and unit testing 86
Testing MVVM applications 87
Testing asynchronous functionality 88
Testing INotifyPropertyChanged implementations 88
Testing message-based communication 89
Testing exception handling 89
Testing validation 90
Summary 91

Preface

Purpose

This eBook provides guidance on building cross-platform enterprise apps using Xamarin.Forms. Xamarin.Forms is a cross-platform UI toolkit that allows developers to easily create native user interface layouts that can be shared across platforms, including iOS, Android, and the Universal Windows Platform (UWP). It provides a comprehensive solution for Business to Employee (B2E), Business to Business (B2B), and Business to Consumer (B2C) apps, providing the ability to share code across all target platforms and helping to lower the total cost of ownership (TCO).

The guide provides architectural guidance for developing adaptable, maintainable, and testable Xamarin.Forms enterprise apps. Guidance is provided on how to implement MVVM, dependency injection, navigation, validation, and configuration management, while maintaining loose coupling. In addition, there's also guidance on performing authentication and authorization with IdentityServer, accessing data from containerized microservices, and unit testing.

The guide comes with source code for the eShopOnContainers mobile app, and source code for the eShopOnContainers reference app. The eShopOnContainers mobile app is a cross-platform enterprise app developed using Xamarin.Forms, which connects to a series of containerized microservices known as the eShopOnContainers reference app. However, the eShopOnContainers mobile app can be configured to consume data from mock services for those who wish to avoid deploying the containerized microservices.

What's left out of this guide's scope

This guide is aimed at readers who are already familiar with Xamarin.Forms. For a detailed introduction to Xamarin.Forms, see the Xamarin.Forms documentation on the Xamarin Developer Center, and Creating Mobile Apps with Xamarin.Forms.

The guide is complementary to .NET Microservices: Architecture for Containerized .NET Applications, which focuses on developing and deploying containerized microservices. Other guides worth reading include Architecting and Developing Modern Web Applications with ASP.NET Core and Microsoft Azure, Containerized Docker Application Lifecycle with Microsoft Platform and Tools, and Microsoft Platform and Tools for Mobile App Development.

Who should use this guide

The audience for this guide is mainly developers and architects who would like to learn how to architect and implement cross-platform enterprise apps using Xamarin.Forms.

A secondary audience is technical decision makers who would like to receive an architectural and technology overview before deciding on what approach to select for cross-platform enterprise app development using Xamarin.Forms.

How to use this guide

This guide focuses on building cross-platform enterprise apps using Xamarin.Forms. As such, it should be read in its entirety to provide a foundation of understanding such apps and their technical considerations. The guide, along with its sample app, can also serve as a starting point or reference for creating a new enterprise app. Use the associated sample app as a template for the new app, or to see how to organize an app's component parts. Then, refer back to this guide for architectural guidance.

Feel free to forward this guide to team members to help ensure a common understanding of cross-platform enterprise app development using Xamarin.Forms. Having everybody working from a common set of terminologies and underlying principles will help ensure a consistent application of architectural patterns and practices.

CHAPTER 1

Introduction

Regardless of platform, developers of enterprise apps face several challenges:

- App requirements that can change over time.
- New business opportunities and challenges.
- Ongoing feedback during development that can significantly affect the scope and requirements of the app.

With these in mind, it's important to build apps that can be easily modified or extended over time. Designing for such adaptability can be difficult as it requires an architecture that allows individual parts of the app to be independently developed and tested in isolation without affecting the rest of the app.

Many enterprise apps are sufficiently complex to require more than one developer. It can be a significant challenge to decide how to design an app so that multiple developers can work effectively on different pieces of the app independently, while ensuring that the pieces come together seamlessly when integrated into the app.

The traditional approach to designing and building an app results in what is referred to as a *monolithic* app, where components are tightly coupled with no clear separation between them. Typically, this monolithic approach leads to apps that are difficult and inefficient to maintain, because it can be difficult to resolve bugs without breaking other components in the app, and it can be difficult to add new features or to replace existing features.

An effective remedy for these challenges is to partition an app into discrete, loosely coupled components that can be easily integrated together into an app. Such an approach offers several benefits:

- It allows individual functionality to be developed, tested, extended, and maintained by different individuals or teams.
- It promotes reuse and a clean separation of concerns between the app's horizontal capabilities, such as authentication and data access, and the vertical capabilities, such as app specific business functionality. This allows the dependencies and interactions between app components to be more easily managed.
- It helps maintain a separation of roles by allowing different individuals, or teams, to focus on a specific task or piece of functionality according to their expertise. In particular, it provides a cleaner separation between the user interface and the app's business logic.

However, there are many issues that must be resolved when partitioning an app into discrete, loosely coupled components. These include:

- Deciding how to provide a clean separation of concerns between the user interface controls and their logic. One of the most important decisions when creating a Xamarin.Forms enterprise app is whether to place business logic in code-behind files, or whether to create a clean separation of concerns between the user interface controls and their logic, in order to

make the app more maintainable and testable. For more information, see Model-View-ViewModel.

- Determining whether to use a dependency injection container. Dependency injection containers reduce the dependency coupling between objects by providing a facility to construct instances of classes with their dependencies injected, and manage their lifetime based on the configuration of the container. For more information, see Dependency injection.
- Choosing between platform provided eventing and loosely coupled message-based communication between components that are inconvenient to link by object and type references. For more information, see Introduction to Communicating between loosely coupled components.
- Deciding how to navigate between pages, including how to invoke navigation, and where navigation logic should reside. For more information, see Navigation.
- Determining how to validate user input for correctness. The decision must include how to validate user input, and how to notify the user about validation errors. For more information, see Validation.
- Deciding how to perform authentication, and how to protect resources with authorization. For more information, see Authentication and authorization.
- Determining how to access remote data from web services, including how to reliably retrieve data, and how to cache data. For more information, see Accessing remote data.
- Deciding how to test the app. For more information, see Unit testing.

This guide provides guidance on these issues, and focuses on the core patterns and architecture for building a cross-platform enterprise app using Xamarin.Forms. The guidance aims to help to produce adaptable, maintainable, and testable code, by addressing common Xamarin.Forms enterprise app development scenarios, and by separating the concerns of presentation, presentation logic, and entities through support for the Model-View-ViewModel (MVVM) pattern.

Sample application

This guide includes a sample application, eShopOnContainers, that's an online store that includes the following functionality:

- Authenticating and authorizing against a backend service.
- Browsing a catalog of shirts, coffee mugs, and other marketing items.
- Filtering the catalog.
- Ordering items from the catalog.
- Viewing the user's order history.
- Configuration of settings.

Sample application architecture

Figure 1-1 provides a high-level overview of the architecture of the sample application.

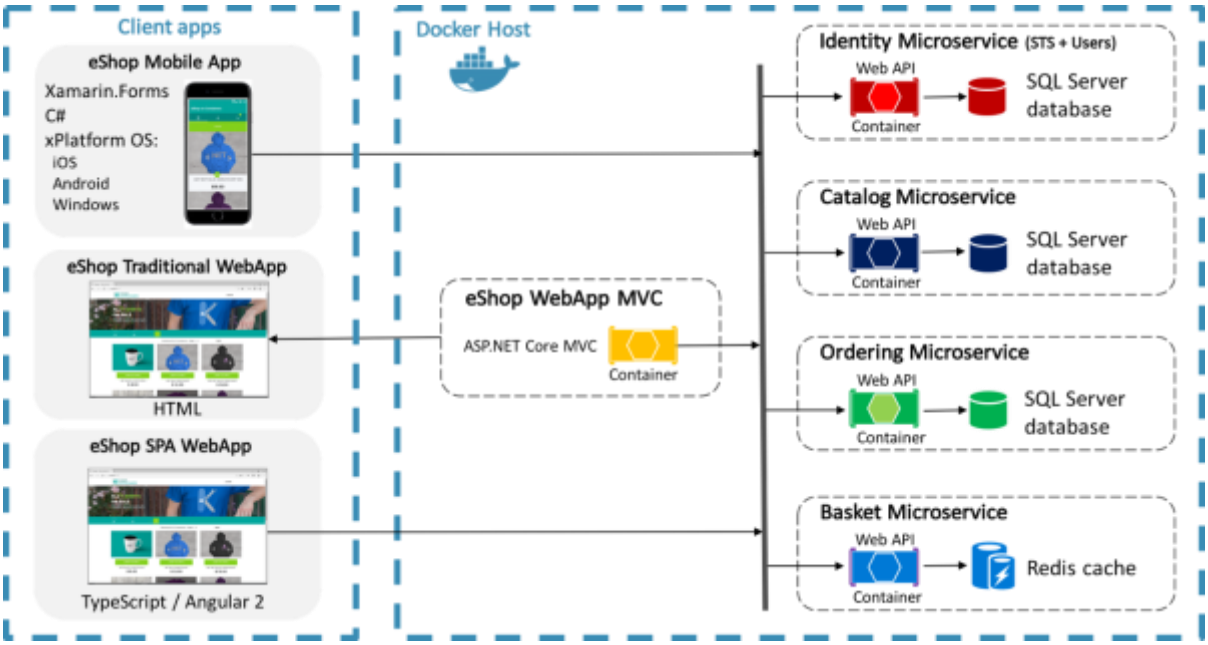

Figure 1-1: eShopOnContainers high-level architecture

The sample application ships with three client apps:

- An MVC application developed with ASP.NET Core.
- A Single Page Application (SPA) developed with Angular 2 and Typescript. This approach for web applications avoids performing a round-trip to the server with each operation.
- A mobile app developed with Xamarin.Forms, which supports iOS, Android, and the Universal Windows Platform (UWP).

For information about the web applications, see Architecting and Developing Modern Web Applications with ASP.NET Core and Microsoft Azure.

The sample application includes the following backend services:

- An identity microservice, which uses ASP.NET Core Identity and IdentityServer.
- A catalog microservice, which is a data-driven create, read, update, delete (CRUD) service that consumes an SQL Server database using EntityFramework Core.
- An ordering microservice, which is a domain-driven service that uses domain-driven design patterns.
- A basket microservice, which is a data-driven CRUD service that uses Redis Cache.

These backend services are implemented as microservices using ASP.NET Core MVC, and are deployed as unique containers within a single Docker host. Collectively, these backend services are referred to as the eShopOnContainers reference application. Client apps communicate with the backend services through a Representational State Transfer (REST) web interface. For more information about microservices and Docker, see Containerized microservices.

For information about the implementation of the backend services, see .NET Microservices: Architecture for Containerized .NET Applications.

Mobile app

This guide focuses on building cross-platform enterprise apps using Xamarin.Forms, and uses the eShopOnContainers mobile app as an example. Figure 1-2 shows the pages from the eShopOnContainers mobile app that provide the functionality outlined earlier.

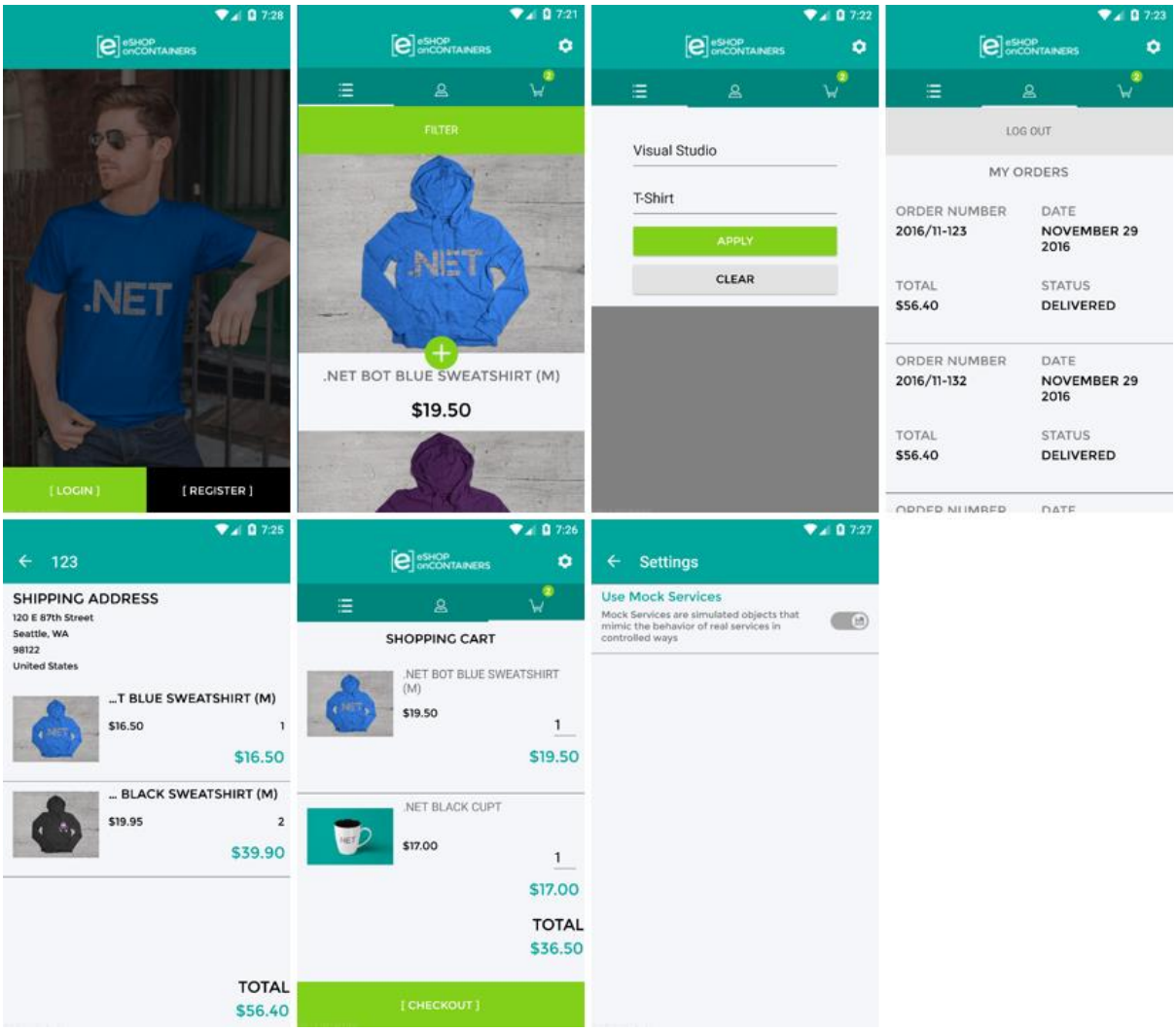

Figure 1-2: The eShopOnContainers mobile app

The mobile app consumes the backend services provided by the eShopOnContainers reference application. However, it can be configured to consume data from mock services for those who wish to avoid deploying the backend services.

The eShopOnContainers mobile app exercises the following Xamarin.Forms functionality:

- XAML
- Controls
- Bindings
- Converters
- Styles

- Animations
- Commands
- Behaviors
- Triggers
- Effects
- Custom Renderers
- MessagingCenter
- Custom Controls

For more information about this functionality, see the Xamarin.Forms documentation on the Xamarin Developer Center, and Creating Mobile Apps with Xamarin.Forms.

In addition, unit tests are provided for some of the classes in the eShopOnContainers mobile app.

Mobile app solution

The eShopOnContainers mobile app solution organizes the source code and other resources into projects. All of the projects use folders to organize the source code and other resources into categories. The following table outlines the projects that make up the eShopOnContainers mobile app:

Project	Description
eShopOnContainers.Core	This project is the portable class library (PCL) project that contains the shared code and shared UI.
eShopOnContainers.Droid	This project holds Android specific code and is the entry point for the Android app.
eShopOnContainers.iOS	This project holds iOS specific code and is the entry point for the iOS app.
eShopOnContainers.UWP	This project holds Universal Windows Platform (UWP) specific code and is the entry point for the Windows app.
eShopOnContainers.TestRunner.Droid	This project is the Android test runner for the eShopOnContainers.UnitTests project.
eShopOnContainers.TestRunner.iOS	This project is the iOS test runner for the eShopOnContainers.UnitTests project.
eShopOnContainers.TestRunner.Windows	This project is the Universal Windows Platform test runner for the eShopOnContainers.UnitTests project.
eShopOnContainers.UnitTests	This project contains unit tests for the eShopOnContainers.Core project.

The classes from the eShopOnContainers mobile app can be re-used in any Xamarin.Forms app with little or no modification.

eShopOnContainers.Core project

The eShopOnContainers.Core PCL project contains the following folders:

Folder	Description
Animations	Contains classes that enable animations to be consumed in XAML.
Behaviors	Contains behaviors that are exposed to view classes.
Controls	Contains custom controls used by the app.
Converters	Contains value converters that apply custom logic to a binding.
Effects	Contains the `EntryLineColorEffect` class, which is used to change the border color of specific `Entry` controls.
Exceptions	Contains the custom `ServiceAuthenticationException`.
Extensions	Contains extension methods for the `VisualElement` and `IEnumerable<T>` classes.
Helpers	Contains helper classes for the app.
Models	Contains the model classes for the app.
Properties	Contains `AssemblyInfo.cs`, a .NET assembly metadata file.
Services	Contains interfaces and classes that implement services that are provided to the app.
Triggers	Contains the `BeginAnimation` trigger, which is used to invoke an animation in XAML.
Validations	Contains classes involved in validating data input.
ViewModels	Contains the application logic that's exposed to pages.
Views	Contains the pages for the app.

Platform projects

The platform projects contain effect implementations, custom renderer implementations, and other platform-specific resources.

Summary

Xamarin's cross-platform mobile app development tools and platforms provide a comprehensive solution for B2E, B2B, and B2C mobile client apps, providing the ability to share code across all target platforms (iOS, Android, and Windows) and helping to lower the total cost of ownership. Apps can share their user interface and app logic code, while retaining the native platform look and feel.

Developers of enterprise apps face several challenges that can alter the architecture of the app during development. Therefore, it's important to build an app so that it can be modified or extended over time. Designing for such adaptability can be difficult, but typically involves partitioning an app into discrete, loosely coupled components that can be easily integrated together into an app.

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.

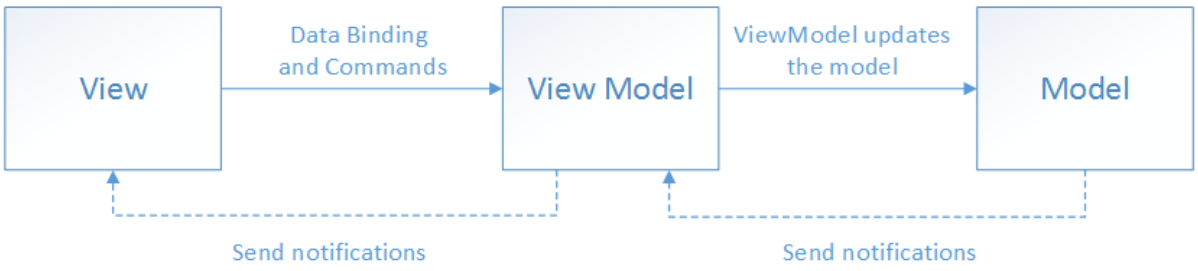

Figure 2-1: The MVVM pattern

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:

- 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.

- 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 with 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.

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 might 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 template 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 aspects 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 might 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 might combine the values 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 services or repositories that encapsulate data access and caching.

Connecting view models to views

View models can be connected to views by using the data-binding capabilities of Xamarin.Forms. There are many approaches that can be used to construct views and view models and associate them at runtime. These approaches fall into two categories, known as view first composition, and view model first composition. Choosing between view first composition and view model first composition is an issue of preference and complexity. However, all approaches share the same aim, which is for the view to have a view model assigned to its `BindingContext` property.

With view first composition the app is conceptually composed of views that 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.

With view model first composition the app is conceptually composed of view models, with a service being responsible for locating the view for a view model. 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. In addition, it allows view models to be created by other view models. 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, such as `Button` and `ListView`, from view models. By following the principles outlined here, view models can be tested 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 maintain loose coupling between the view and view model. For more information, see 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 custom 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, viewAssemblyName);

    var viewModelType = Type.GetType(viewModelName);
    if (viewModelType == null)
    {
        return;
    }
    var viewModel = _container.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.

App's should be architected for the correct use of property change notification, by meeting the following requirements:

- Always raising 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 raising a `PropertyChanged` event for any calculated properties whose values are used by other properties in the view model or model.
- Always raising 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 might expose the object to callback functions when it is in an unsafe, partially updated 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 raising 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 raising 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 raising 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.

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.

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 affect 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 about this behavior, 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 named '{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> action)
    {
        ...
    }

    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

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, 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 registration through assembly scanning.

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 Autofac to manage the instantiation of view model and service classes in the app. Autofac facilitates building loosely coupled apps, and provides all of the features commonly found in dependency injection containers, including methods to register type mappings and object instances, resolve objects, manage object lifetimes, and inject dependent objects into constructors of objects that it resolves. For more information about Autofac, see Autofac on readthedocs.io.

In Autofac, the `IContainer` interface provides the dependency injection container. Figure 3-1 shows the dependencies when using this container, which instantiates an `IOrderService` object and injects it into the `ProfileViewModel` class.

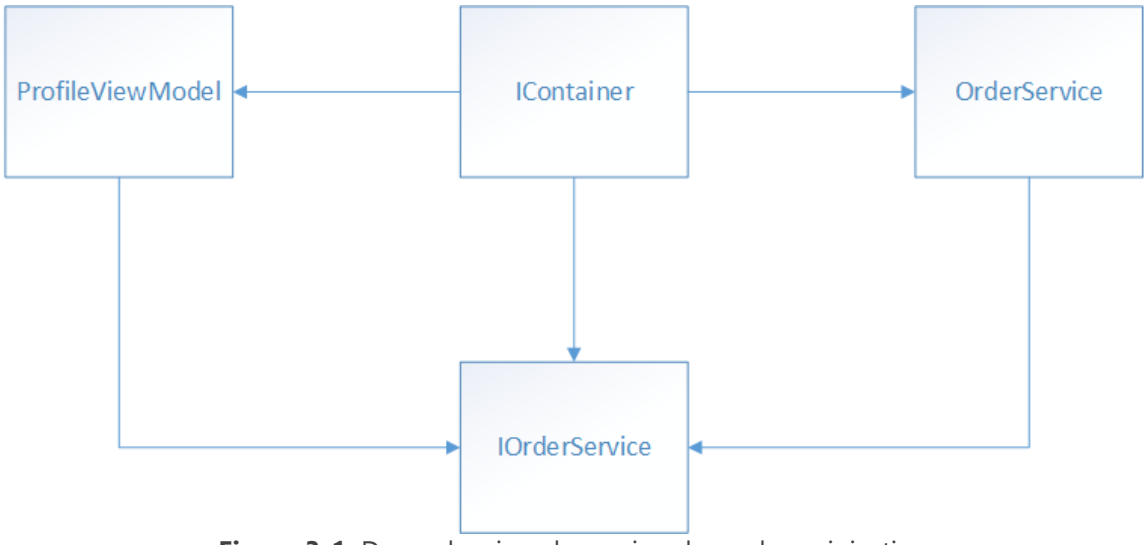

Figure 3-1: Dependencies when using dependency injection

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 might 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 might not make sense to put it in the container. In addition, if a class has a single set of dependencies that are integral to the type and will never change, it might 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 builds the `IContainer` 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 declares the `IContainer` object in the `ViewModelLocator` class:

```
private static IContainer _container;
```

Types and instances are registered in the `RegisterDependencies` method in the `ViewModelLocator` class. This is achieved by first creating a `ContainerBuilder` instance, which is demonstrated in the following code example:

```
var builder = new ContainerBuilder();
```

Types and instances are then registered with the `ContainerBuilder` object, and the following code example demonstrates the most common form of type registration:

```
builder.RegisterType<RequestProvider>().As<IRequestProvider>();
```

The `RegisterType` method shown here 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:

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

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

Autofac also allows 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:

```
builder.RegisterType<OrderService>().As<IOrderService>().SingleInstance();
```

The `RegisterType` method shown here maps an interface type to a concrete type. The `SingleInstance` method configures the registration so that every dependent object receives the same shared instance. Therefore, only a single `OrderService` instance will exist in the container, which is shared by objects that require an injection of an `IOrderService` through a constructor.

Instance registration can also be performed with the `RegisterInstance` method, which is demonstrated in the following code example:

```
builder.RegisterInstance(new OrderMockService()).As<IOrderService>();
```

The `RegisterInstance` method shown here creates a new `OrderMockService` instance and registers it with the container. Therefore, only a single `OrderMockService` instance exists in the container, which is shared by objects that require an injection of an `IOrderService` through a constructor.

Following type and instance registration, the `IContainer` object must be built, which is demonstrated in the following code example:

```
_container = builder.Build();
```

Invoking the `Build` method on the `ContainerBuilder` instance builds a new dependency injection container that contains the registrations that have been made.

> **Tip**: Consider an `IContainer` as being immutable
>
> While Autofac provides an `Update` method to update registrations in an existing container, calling this method should be avoided where possible. There are risks to modifying a container after it's been built, particularly if the container has been used. For more information, see Consider a Container as Immutable on readthedocs.io.

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 is called for, the container creates it if required, and maintains a reference to it.
3. If the type hasn't 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 `RequestProvider` type that was previously registered with Autofac can be resolved:

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

In this example, Autofac 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 = _container.Resolve(viewModelType);
```

In this example, Autofac 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, Autofac 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 Autofac 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 Autofac 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.

An Autofac instance scope can be used to specify the singleton behavior for an object that Autofac creates from a registered type. Autofac instance scopes manage the object lifetimes instantiated by the container. The default instance scope for the `RegisterType` method is the `InstancePerDependency` scope. However, the `SingleInstance` scope 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 Autofac is instructed to create a singleton instance of the `NavigationService` class:

```
builder.RegisterType<NavigationService>().As<INavigationService>().SingleInstance();
```

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 `SingleInstance` scope disposes created objects when the container is disposed.

Autofac includes additional instance scopes. For more information, see Instance Scope on readthedocs.io.

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.

Autofac facilitates building loosely coupled apps, and provides all of the features commonly found in dependency injection containers, 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 in which 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:

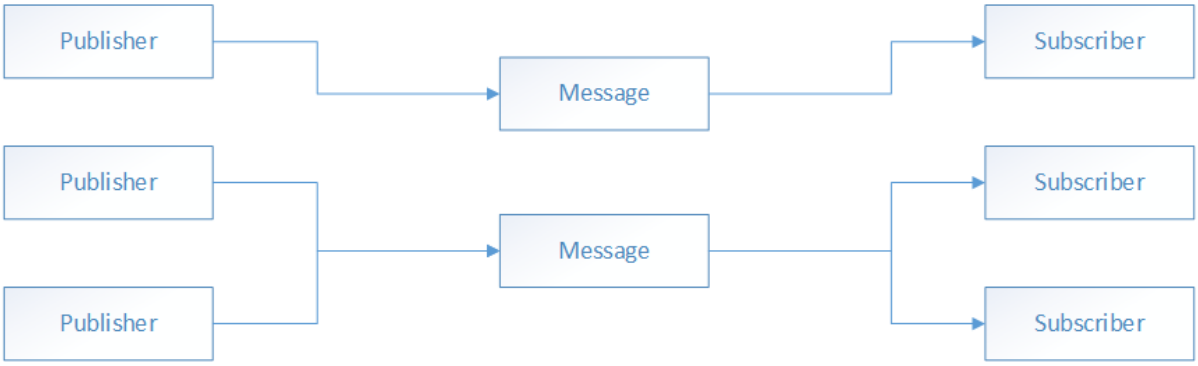

Figure 4-1: Multicast publish-subscribe functionality

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 the message.

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 performing UI updates
>
> 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 of 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, MessageKeys.AddProduct, async (sender, arg) =>
{
    BadgeCount++;

    await AddCatalogItemAsync(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 might 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 must be met:

- 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 might 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 might 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 might 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 might 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 might be the simplest approach, it is not easily testable through unit tests. Placing navigation logic in view model classes means that the logic can be exercised through unit 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 might 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
{
    ViewModelBase PreviousPageViewModel { get; }
    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.

In addition, the `INavigationService` interface specifies that an implementing class must provide a `PreviousPageViewModel` property. This property returns the view model type associated with the previous page in 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 because it's not required.

Creating the NavigationService instance

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

```
builder.RegisterType<NavigationService>().As<INavigationService>().SingleInstance();
```

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 Autofac 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 Autofac dependency injection container into each view model class.

Handling navigation requests

Xamarin.Forms provides the `NavigationPage` class, which implements a hierarchical navigation experience in which 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, viewModelAssemblyName);
    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 might 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 Autofac 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 Autofac 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 that 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 a method that can be overridden. 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 might need to interact with the user during a navigation operation, so that the user can confirm or cancel navigation. This might 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 to 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 or not navigation is invoked.

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 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 for input that 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 of why the data is invalid. Figure 6-1 shows the classes involved in performing validation in the eShopOnContainers mobile app.

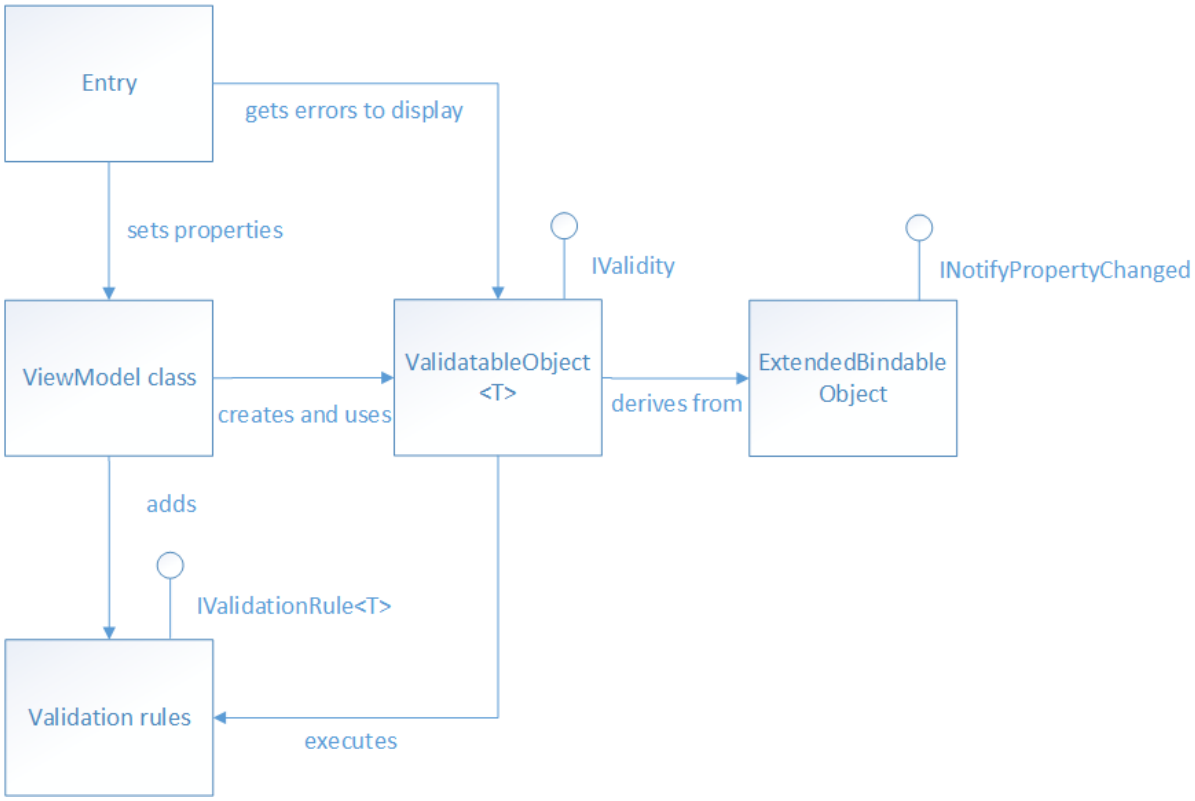

Figure 6-1: Validation classes in the eShopOnContainers mobile app

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 or not the entered data is valid.

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 or not the `value` argument is a valid email address. 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 is 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.

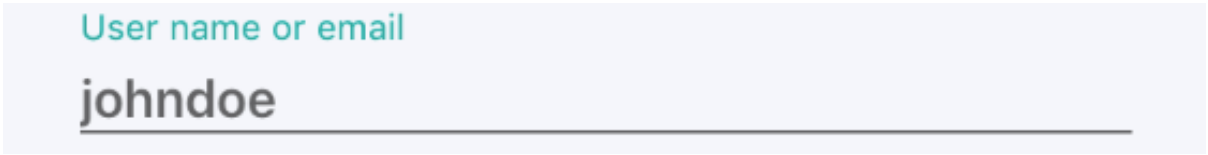

Figure 6-3: Black line indicating no validation error

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.

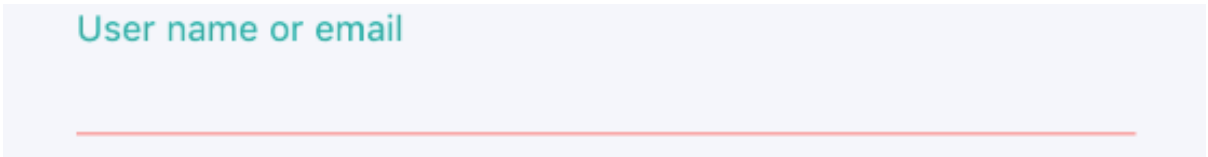

Figure 6-4: Red line indicating validation error

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.

Configuration management

Settings allow the separation of data that configures the behavior of an app from the code, allowing the behavior to be changed without rebuilding the app. There are two types of settings: app settings, and user settings.

App settings are data that an app creates and manages. It can include data such as fixed web service endpoints, API keys, and runtime state. App settings are tied to the existence of the app and are only meaningful to that app.

User settings are the customizable settings of an app that affect the behavior of the app and don't require frequent re-adjustment. For example, an app might let the user specify where to retrieve data from, and how to display it on the screen.

Xamarin.Forms includes a persistent dictionary that can be used to store settings data. This dictionary can be accessed using the `Application.Current.Properties` property, and any data that's placed into it is saved when the app goes into a sleep state, and is restored when the app resumes or starts up again. In addition, the `Application` class also has a `SavePropertiesAsync` method that allows an app to have its settings saved when required. For more information about this dictionary, see Properties Dictionary on the Xamarin Developer Center.

A downside to storing data using the Xamarin.Forms persistent dictionary is that it's not easily data bound to. Therefore, the eShopOnContainers mobile app uses the Xam.Plugins.Settings library, available from NuGet. This library provides a consistent, type-safe, cross-platform approach for persisting and retrieving app and user settings, while using the native settings management provided by each platform. In addition, it's straightforward to use data binding to access settings data exposed by the library.

> **Note:** While the Xam.Plugin.Settings library can store both app and user settings, it makes no distinction between the two.

Creating a settings class

When using the Xam.Plugins.Settings library, a single `static` class should be created that will contain the app and user settings required by the app. The following code example shows the `Settings` class in the eShopOnContainers mobile app:

```
public static class Settings
{
    private static ISettings AppSettings
    {
        get
        {
            return CrossSettings.Current;
        }
    }
    ...
}
```

Settings can be read and written through the `ISettings` API, which is provided by the Xam.Plugins.Settings library. This library provides a singleton that can be used to access the API, `CrossSettings.Current`, and an app's settings class should expose this singleton via an `ISettings` property.

> **Note:** Using directives for the `Plugin.Settings` and `Plugin.Settings.Abstractions` namespaces should be added to a class that requires access to the Xam.Plugins.Settings library types.

Adding a setting

Each setting consists of a key, a default value, and a property. The following code example shows all three items for a user setting that represents the base URL for the online services that the eShopOnContainers mobile app connects to:

```
public static class Settings
{
    ...
    private const string IdUrlBase = "url_base";
    private static readonly string UrlBaseDefault = GlobalSetting.Instance.BaseEndpoint;
    ...

    public static string UrlBase
    {
        get
        {
            return AppSettings.GetValueOrDefault<string>(IdUrlBase, UrlBaseDefault);
        }
        set
        {
            AppSettings.AddOrUpdateValue<string>(IdUrlBase, value);
        }
    }
}
```

The key is always a `const string` that defines the key name, with the default value for the setting being a `static readonly` value of the required type. Providing a default value ensures that a valid value is available if an unset setting is retrieved.

The `UrlBase static` property uses two methods from the `ISettings` API to read or write the setting value. The `ISettings.GetValueOrDefault` method is used to retrieve a setting's value from platform-specific storage. If no value is defined for the setting, its default value is retrieved instead. Similarly, the `ISettings.AddOrUpdateValue` method is used to persist a setting's value to platform-specific storage.

Rather that define a default value inside the `Settings` class, the `UrlBaseDefault` string obtains its value from the `GlobalSetting` class. The following code example shows the `BaseEndpoint` property and `UpdateEndpoint` method in this class:

```
public class GlobalSetting
{
    ...
    public string BaseEndpoint
    {
        get { return _baseEndpoint; }
        set
        {
            _baseEndpoint = value;
            UpdateEndpoint(_baseEndpoint);
        }
    }
    ...

    private void UpdateEndpoint(string baseEndpoint)
    {
        RegisterWebsite = string.Format("{0}:5105/Account/Register", baseEndpoint);
        CatalogEndpoint = string.Format("{0}:5101", baseEndpoint);
        OrdersEndpoint = string.Format("{0}:5102", baseEndpoint);
        BasketEndpoint = string.Format("{0}:5103", baseEndpoint);
        IdentityEndpoint = string.Format("{0}:5105/connect/authorize", baseEndpoint);
        UserInfoEndpoint = string.Format("{0}:5105/connect/userinfo", baseEndpoint);
        TokenEndpoint = string.Format("{0}:5105/connect/token", baseEndpoint);
        LogoutEndpoint = string.Format("{0}:5105/connect/endsession", baseEndpoint);
        IdentityCallback = string.Format("{0}:5105/xamarincallback", baseEndpoint);
        LogoutCallback = string.Format("{0}:5105/Account/Redirecting", baseEndpoint);
    }
}
```

Each time the `BaseEndpoint` property is set, the `UpdateEndpoint` method is called. This method updates a series of properties, all of which are based on the `UrlBase` user setting that's provided by the `Settings` class that represent different endpoints that the eShopOnContainers mobile app connects to.

Data binding to user settings

In the eShopOnContainers mobile app, the `SettingsView` exposes two user settings. These settings allow configuration of whether the app should retrieve data from microservices that are deployed as Docker containers, or whether the app should retrieve data from mock services that don't require an internet connection. When choosing to retrieve data from containerized microservices, a base endpoint URL for the microservices must be specified. Figure 7-1 shows the `SettingsView` when the user has chosen to retrieve data from containerized microservices.

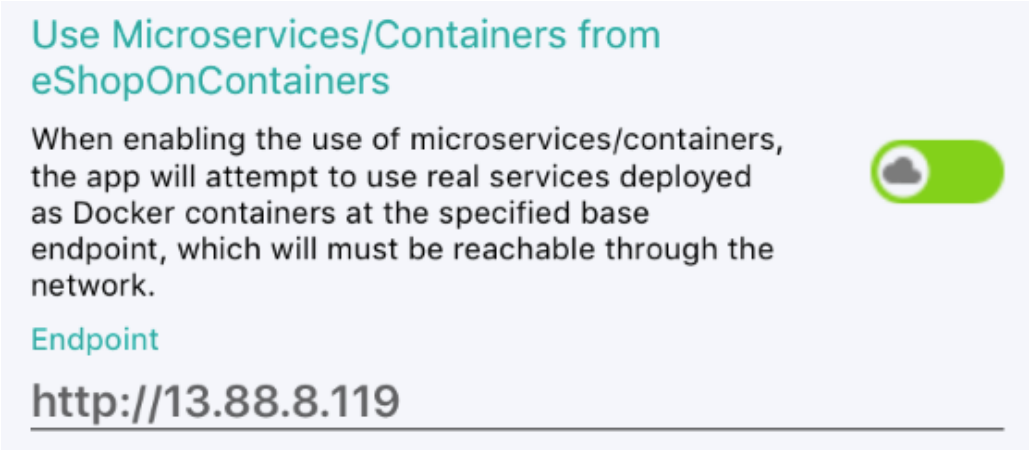

Figure 7-1: User settings exposed by the eShopOnContainers mobile app

Data binding can be used to retrieve and set settings exposed by the `Settings` class. This is achieved by controls on the view binding to view model properties that in turn access properties in the `Settings` class, and raising a property changed notification if the settings value has changed. For 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.

The following code example shows the `Entry` control from the `SettingsView` that allows the user to enter a base endpoint URL for the containerized microservices:

```
<Entry Text="{Binding Endpoint, Mode=TwoWay}" />
```

This `Entry` control binds to the `Endpoint` property of the `SettingsViewModel` class, using a two-way binding. The following code example shows the `Endpoint` property:

```
public string Endpoint
{
    get { return _endpoint; }
    set
    {
        _endpoint = value;

        if(!string.IsNullOrEmpty(_endpoint))
        {
            UpdateEndpoint(_endpoint);
        }

        RaisePropertyChanged(() => Endpoint);
    }
}
```

When the Endpoint property is set the `UpdateEndpoint` method is called, provided that the supplied value is valid, and property changed notification is raised. The following code example shows the `UpdateEndpoint` method:

```
private void UpdateEndpoint(string endpoint)
{
    Settings.UrlBase = endpoint;
}
```

This method updates the `UrlBase` property in the `Settings` class with the base endpoint URL value entered by the user, which causes it to be persisted to platform-specific storage.

When the `SettingsView` is navigated to, the `InitializeAsync` method in the `SettingsViewModel` class is executed. The following code example shows this method:

```
public override Task InitializeAsync(object navigationData)
{
    ...
    Endpoint = Settings.UrlBase;
    ...
}
```

The method sets the `Endpoint` property to the value of the `UrlBase` property in the `Settings` class. Accessing the `UrlBase` property causes the Xam.Plugins.Settings library to retrieve the settings value from platform-specific storage. For information about how the `InitializeAsync` method is invoked, see Passing parameters during navigation.

This mechanism ensures that whenever a user navigates to the `SettingsView`, user settings are retrieved from platform-specific storage and presented through data binding. Then, if the user changes the settings values, data binding ensures that they are immediately persisted back to platform-specific storage.

Summary

Settings allow the separation of data that configures the behavior of an app from the code, allowing the behavior to be changed without rebuilding the app. App settings are data that an app creates and manages, and user settings are the customizable settings of an app that affect the behavior of the app and don't require frequent re-adjustment.

The Xam.Plugins.Settings library provides a consistent, type-safe, cross-platform approach for persisting and retrieving app and user settings, and data binding can be used to access settings created with the library.

Containerized microservices

Developing client-server applications has resulted in a focus on building tiered applications that use specific technologies in each tier. Such applications are often referred to as *monolithic* applications, and are packaged onto hardware pre-scaled for peak loads. The main drawbacks of this development approach are the tight coupling between components within each tier, that individual components can't be easily scaled, and the cost of testing. A simple update can have unforeseen effects on the rest of the tier, and so a change to an application component requires its entire tier to be retested and redeployed.

Particularly concerning in the age of the cloud, is that individual components can't be easily scaled. A monolithic application contains domain-specific functionality, and is typically divided by functional layers such as front end, business logic, and data storage. A monolithic application is scaled by cloning the entire application onto multiple machines, as illustrated in Figure 8-1.

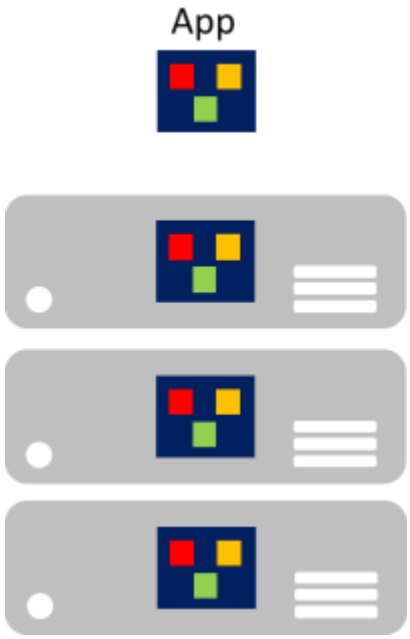

Figure 8-1: Monolithic application scaling approach

Microservices

Microservices offer a different approach to application development and deployment, an approach that's suited to the agility, scale, and reliability requirements of modern cloud applications. A microservices application is decomposed into independent components that work together to deliver the application's overall functionality. The term microservice emphasizes that applications should be composed of services small enough to reflect independent concerns, so that each microservice implements a single function. In addition, each microservice has well-defined contracts so that other microservices can communicate and share data with it. Typical examples of microservices include shopping carts, inventory processing, purchase subsystems, and payment processing.

Microservices can scale-out independently, as compared to giant monolithic applications that scale together. This means that a specific functional area, that requires more processing power or network bandwidth to support demand, can be scaled rather than unnecessarily scaling-out other areas of the application. Figure 8-2 illustrates this approach, where microservices are deployed and scaled independently, creating instances of services across machines.

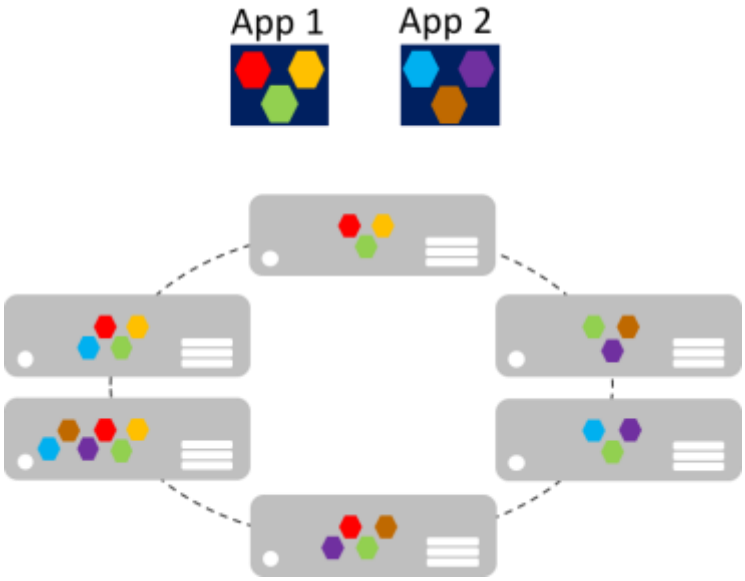

Figure 8-2: Microservices application scaling approach

Microservice scale-out can be nearly instantaneous, allowing an application to adapt to changing loads. For example, a single microservice in the web-facing functionality of an application might be the only microservice in the application that needs to scale out to handle additional incoming traffic.

The classic model for application scalability is to have a load-balanced, stateless tier with a shared external datastore to store persistent data. Stateful microservices manage their own persistent data, usually storing it locally on the servers on which they are placed, to avoid the overhead of network access and complexity of cross-service operations. This enables the fastest possible processing of data and can eliminate the need for caching systems. In addition, scalable stateful microservices usually partition data among their instances, in order to manage data size and transfer throughput beyond which a single server can support.

Microservices also support independent updates. This loose coupling between microservices provides a rapid and reliable application evolution. Their independent, distributed nature supports rolling updates, where only a subset of instances of a single microservice will update at any given time. Therefore, if a problem is detected, a buggy update can be rolled back, before all instances update with the faulty code or configuration. Similarly, microservices typically use schema versioning, so that

clients see a consistent version when updates are being applied, regardless of which microservice instance is being communicated with.

Therefore, microservice applications have many benefits over monolithic applications:

- Each microservice is relatively small, easy to manage and evolve.
- Each microservice can be developed and deployed independently of other services.
- Each microservice can be scaled-out independently. For example, a catalog service or shopping basket service might need to be scaled-out more than an ordering service. Therefore, the resulting infrastructure will more efficiently consume resources when scaling out.
- Each microservice isolates any issues. For example, if there is an issue in a service it only impacts that service. The other services can continue to handle requests.
- Each microservice can use the latest technologies. Because microservices are autonomous and run side-by-side, the latest technologies and frameworks can be used, rather than being forced to use an older framework that might be used by a monolithic application.

However, a microservice based solution also has potential drawbacks:

- Choosing how to partition an application into microservices can be challenging, as each microservice has to be completely autonomous, end-to-end, including responsibility for its data sources.
- Developers must implement inter-service communication, which adds complexity and latency to the application.
- Atomic transactions between multiple microservices usually aren't possible. Therefore, business requirements must embrace eventual consistency between microservices.
- In production, there is an operational complexity in deploying and managing a system compromised of many independent services.
- Direct client-to-microservice communication can make it difficult to refactor the contracts of microservices. For example, over time how the system is partitioned into services might need to change. A single service might split into two or more services, and two services might merge. When clients communicate directly with microservices, this refactoring work can break compatibility with client apps.

Containerization

Containerization is an approach to software development in which an application and its versioned set of dependencies, plus its environment configuration abstracted as deployment manifest files, are packaged together as a container image, tested as a unit, and deployed to a host operating system.

A container is an isolated, resource controlled, and portable operating environment, where an application can run without touching the resources of other containers, or the host. Therefore, a container looks and acts like a newly installed physical computer or a virtual machine.

There are many similarities between containers and virtual machines, as illustrated in Figure 8-3.

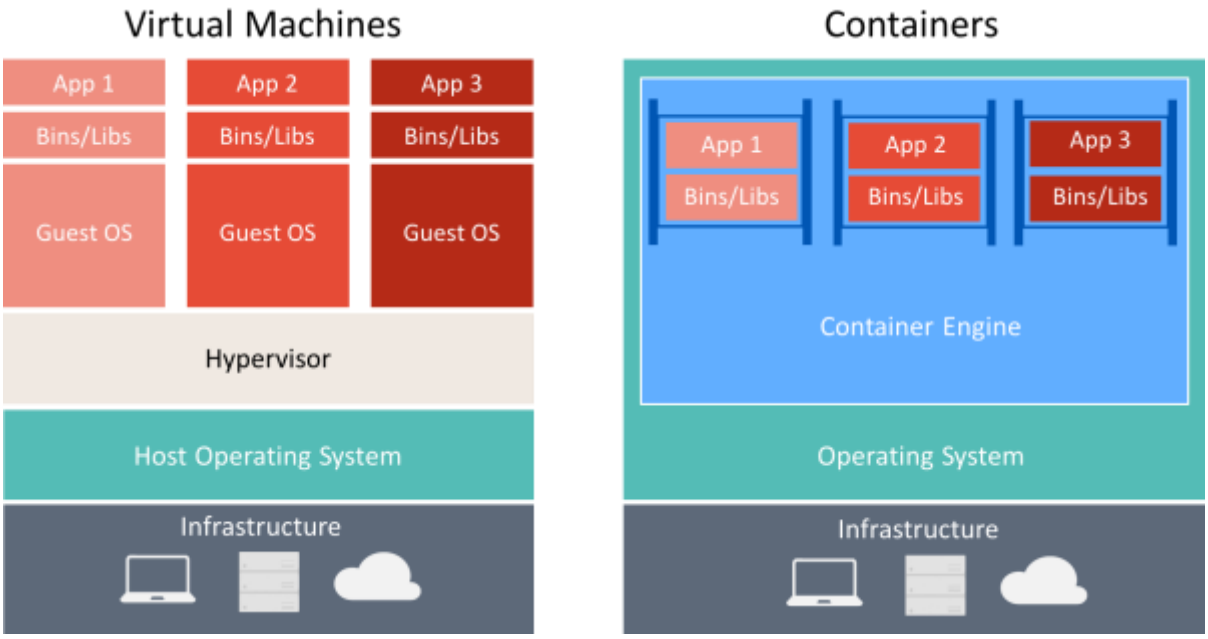

Figure 8-3: Comparison of virtual machines and containers

A container runs an operating system, has a file system, and can be accessed over a network as if it were a physical or virtual machine. However, the technology and concepts used by containers are very different from virtual machines. Virtual machines include the applications, the required dependencies, and a full guest operating system. Containers include the application and its dependencies, but share the operating system with other containers, running as isolated processes on the host operating system (aside from Hyper-V containers which run inside of a special virtual machine per container). Therefore, containers share resources and typically require fewer resources than virtual machines.

The advantage of a container-oriented development and deployment approach is that it eliminates most of the issues that arise from inconsistent environment setups and the problems that come with them. In addition, containers permit fast application scale-up functionality by instancing new containers as required.

The key concepts when creating and working with containers are:

- Container Host: The physical or virtual machine configured to host containers. The container host will run one or more containers.
- Container Image: An image consists of a union of layered filesystems stacked on top of each other, and is the basis of a container. An image does not have state and it never changes as it's deployed to different environments.
- Container: A container is a runtime instance of an image.
- Container OS Image: Containers are deployed from images. The container operating system image is the first layer in potentially many image layers that make up a container. A container operating system is immutable, and can't be modified.
- Container Repository: Each time a container image is created, the image and its dependencies are stored in a local repository. These images can be reused many times on the container host. The container images can also be stored in a public or private registry, such as Docker Hub, so that they can be used across different container hosts.

Enterprises are increasingly adopting containers when implementing microservice based applications, and Docker has become the standard container implementation that has been adopted by most software platforms and cloud vendors.

The eShopOnContainers reference application uses Docker to host four containerized back-end microservices, as illustrated in Figure 8-4.

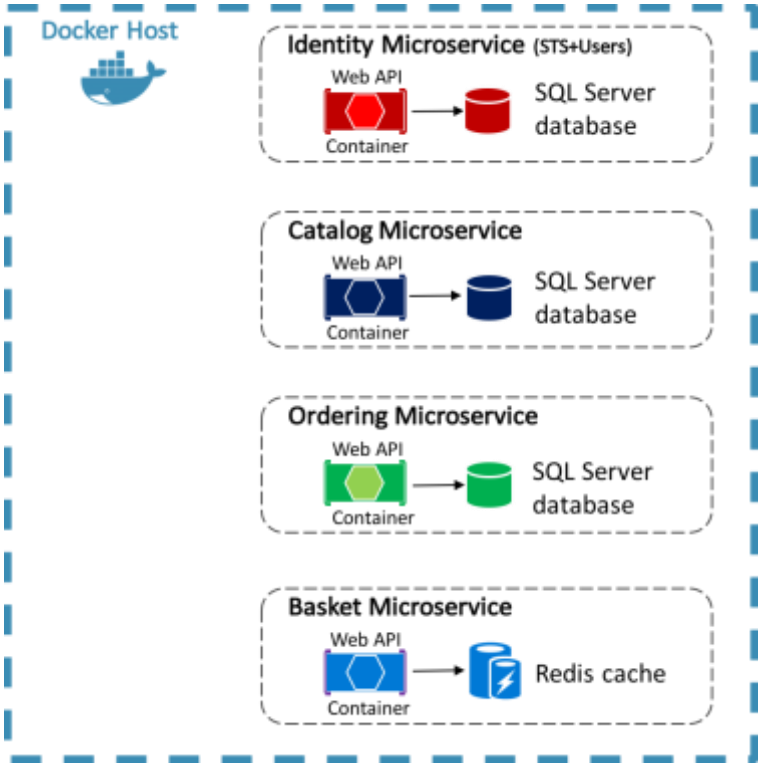

Figure 8-4: eShopOnContainers reference application back-end microservices

The architecture of the back-end services in the reference application is decomposed into multiple autonomous sub-systems in the form of collaborating microservices and containers. Each microservice provides a single area of functionality: an identity service, a catalog service, an ordering service, and a basket service.

Each microservice has its own database, allowing it to be fully decoupled from the other microservices. Where necessary, consistency between databases from different microservices is achieved using application-level events. For more information, see Communication between microservices.

For more information about the reference application, see .NET Microservices: Architecture for Containerized .NET Applications.

Communication between client and microservices

The eShopOnContainers mobile app communicates with the containerized back-end microservices using *direct client-to-microservice* communication, which is shown in Figure 8-5.

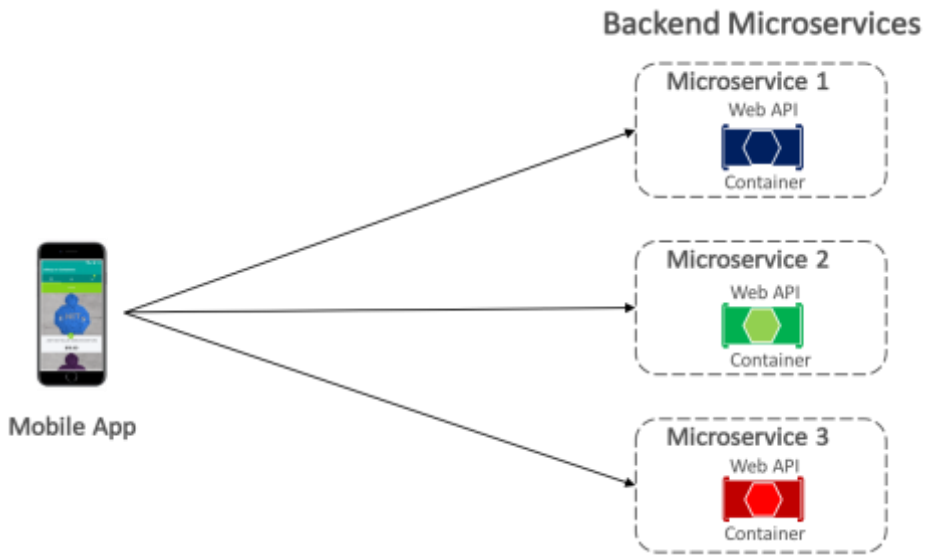

Figure 8-5: Direct client-to-microservice communication

With direct client-to-microservice communication, the mobile app makes requests to each microservice directly through its public endpoint, with a different TCP port per microservice. In production, the endpoint would typically map to the microservice's load balancer, which distributes requests across the available instances.

> **Tip:** Consider using API gateway communication
>
> Direct client-to-microservice communication can have drawbacks when building a large and complex microservice based application, but it's more than adequate for a small application. When designing a large microservice based application with tens of microservices, consider using API gateway communication. For more information, see .NET Microservices: Architecture for Containerized .NET Applications.

Communication between microservices

A microservices based application is a distributed system, potentially running on multiple machines. Each service instance is typically a process. Therefore, services must interact using an inter-process communication protocol, such as HTTP, TCP, Advanced Message Queuing Protocol (AMQP), or binary protocols, depending on the nature of each service.

The two common approaches for microservice-to-microservice communication are HTTP based REST communication when querying for data, and lightweight asynchronous messaging when communicating updates across multiple microservices.

Asynchronous messaging based event-driven communication is critical when propagating changes across multiple microservices. With this approach, a microservice publishes an event when something notable happens, for example, when it updates a business entity. Other microservices subscribe to these events. Then, when a microservice receives an event, it updates its own business entities, which might in turn lead to more events being published. This publish-subscribe functionality is usually achieved with an event bus.

An event bus allows publish-subscribe communication between microservices, without requiring the components to be explicitly aware of each other, as shown in Figure 8-6.

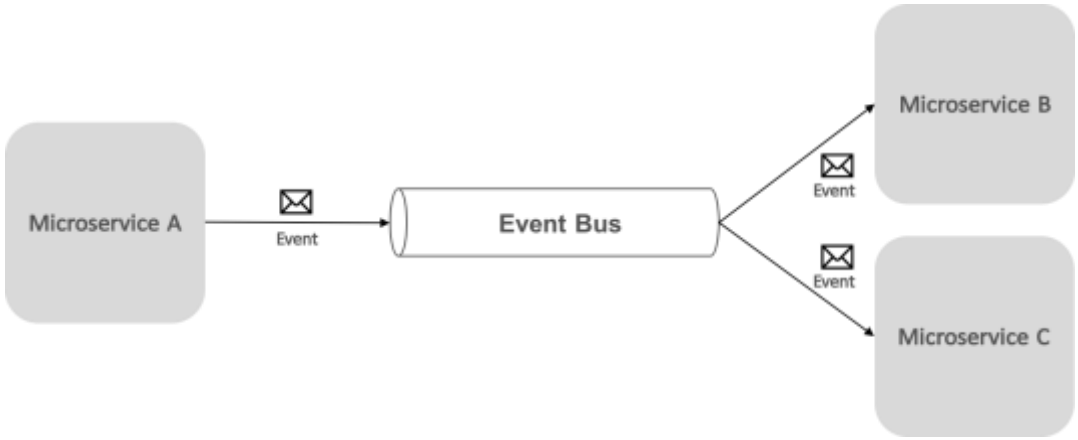

Figure 8-6: Publish-subscribe with an event bus

From an application perspective, the event bus is simply a publish-subscribe channel exposed via an interface. However, the way the event bus is implemented can vary. For example, an event bus implementation could use RabbitMQ, Azure Service Bus, or other service buses such as NServiceBus and MassTransit. Figure 8-7 shows how an event bus is used in the eShopOnContainers reference application.

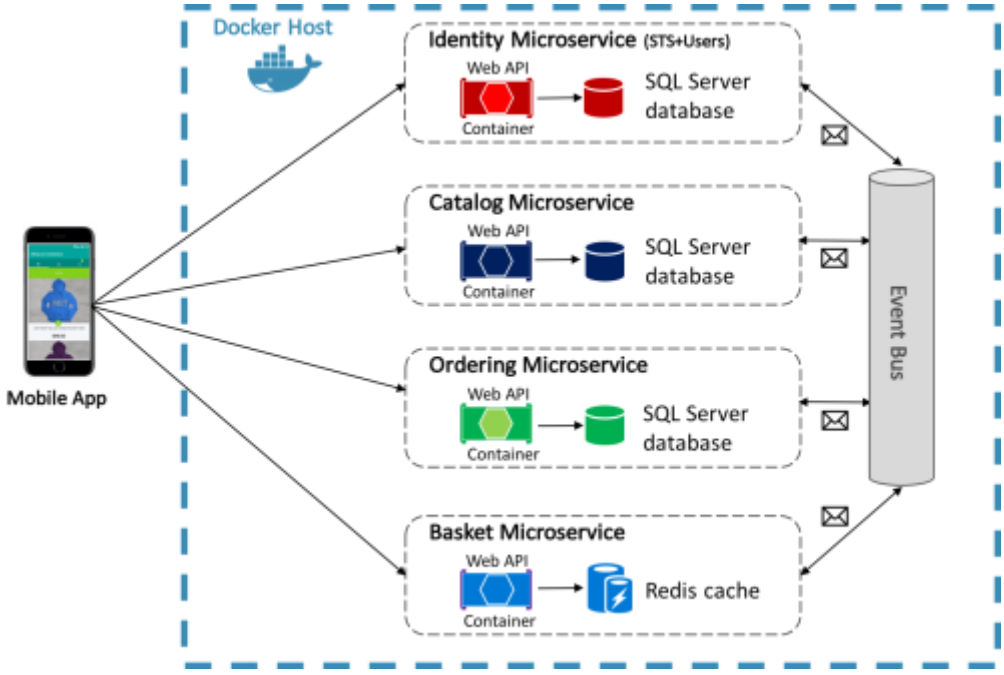

Figure 8-7: Asynchronous event-driven communication in the reference application

The eShopOnContainers event bus, implemented using RabbitMQ, provides one-to-many asynchronous publish-subscribe functionality. This means that after publishing an event, there can be multiple subscribers listening for the same event. Figure 8-9 illustrates this relationship.

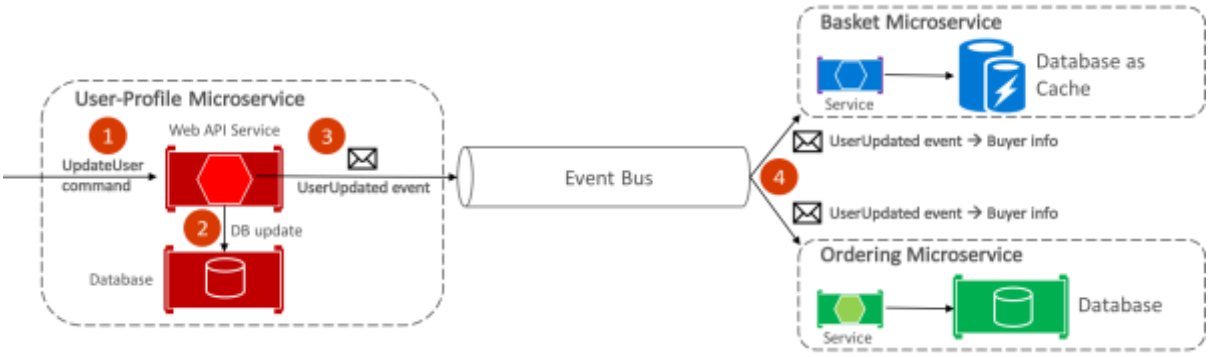

Figure 8-9: One-to-many communication

This one-to-many communication approach uses events to implement business transactions that span multiple services, ensuring eventual consistency between the services. An eventual-consistent transaction consists of a series of distributed steps. Therefore, when the user-profile microservice receives the `UpdateUser` command, it updates the user's details in its database and publishes the `UserUpdated` event to the event bus. Both the basket microservice and the ordering microservice have subscribed to receive this event, and in response update their buyer information in their respective databases.

> **Note:** The eShopOnContainers event bus, implemented using RabbitMQ, is intended to be used only as a proof of concept. For production systems, alternative event bus implementations should be considered.

For information about the event bus implementation, see .NET Microservices: Architecture for Containerized .NET Applications.

Summary

Microservices offer an approach to application development and deployment that's suited to the agility, scale, and reliability requirements of modern cloud applications. One of the main advantages of microservices is that they can be scaled-out independently, which means that a specific functional area can be scaled that requires more processing power or network bandwidth to support demand, without unnecessarily scaling areas of the application that are not experiencing increased demand.

A container is an isolated, resource controlled, and portable operating environment, where an application can run without touching the resources of other containers, or the host. Enterprises are increasingly adopting containers when implementing microservice based applications, and Docker has become the standard container implementation that has been adopted by most software platforms and cloud vendors.

Authentication and authorization

Authentication is the process of obtaining identification credentials such as name and password from a user, and validating those credentials against an authority. If the credentials are valid, the entity that submitted the credentials is considered an authenticated identity. Once an identity has been authenticated, an authorization process determines whether that identity has access to a given resource.

There are many approaches to integrating authentication and authorization into a Xamarin.Forms app that communicates with an ASP.NET MVC web application, including using ASP.NET Core Identity, external authentication providers such as Microsoft, Google, Facebook, or Twitter, and authentication middleware. The eShopOnContainers mobile app performs authentication and authorization with a containerized identity microservice that uses IdentityServer 4. The mobile app requests security tokens from IdentityServer, either for authenticating a user or for accessing a resource. For IdentityServer to issue tokens on behalf of a user, the user must sign-in to IdentityServer. However, IdentityServer doesn't provide a user interface or database for authentication. Therefore, in the eShopOnContainers reference application, ASP.NET Core Identity is used for this purpose.

Authentication

Authentication is required when an application needs to know the identity of the current user. ASP.NET Core's primary mechanism for identifying users is the ASP.NET Core Identity membership system, which stores user information in a data store configured by the developer. Typically, this data store will be an EntityFramework store, though custom stores or third party packages can be used to store identity information in Azure storage, DocumentDB, or other locations.

For authentication scenarios that make use of a local user data store, and that persist identity information between requests via cookies (as is typical in ASP.NET MVC web applications), ASP.NET Core Identity is a suitable solution. However, cookies are not always a natural means of persisting and transmitting data. For example, an ASP.NET Core web application that exposes RESTful endpoints that are accessed from a mobile app will typically need to use bearer token authentication, since cookies can't be used in this scenario. However, bearer tokens can easily be retrieved and included in the authorization header of web requests made from the mobile app.

Issuing bearer tokens using IdentityServer 4

IdentityServer 4 is an open source OpenID Connect and OAuth 2.0 framework for ASP.NET Core, which can be used for many authentication and authorization scenarios including issuing security tokens for local ASP.NET Core Identity users.

> **Note:** OpenID Connect and OAuth 2.0 are very similar, while having different responsibilities.
>
> OpenID Connect is an authentication layer on top of the OAuth 2.0 protocol. OAuth 2 is a protocol that allows applications to request access tokens from a security token service and use them to communicate with APIs. This delegation reduces complexity in both client applications and APIs since authentication and authorization can be centralized.
>
> The combination of OpenID Connect and OAuth 2.0 combine the two fundamental security concerns of authentication and API access, and IdentityServer 4 is an implementation of these protocols.

In applications that use direct client-to-microservice communication, such as the eShopOnContainers reference application, a dedicated authentication microservice acting as a Security Token Service (STS) can be used to authenticate users, as shown in Figure 9-1. For more information about direct client-to-microservice communication, see Communication between client and microservices.

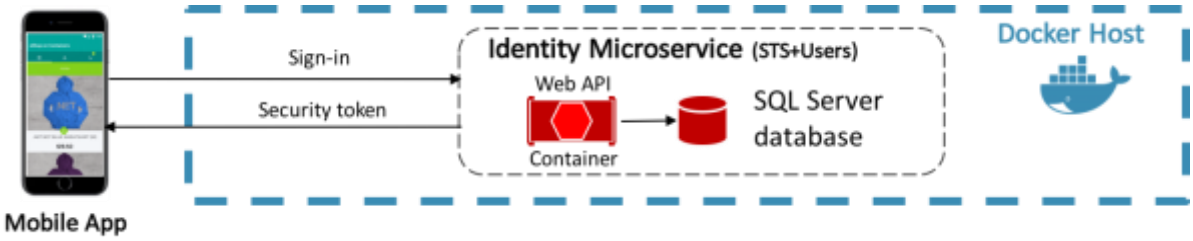

Figure 9-1: Authentication by a dedicated authentication microservice

The eShopOnContainers mobile app communicates with the identity microservice, which uses IdentityServer 4 to perform authentication, and access control for APIs. Therefore, the mobile app requests tokens from IdentityServer, either for authenticating a user or for accessing a resource:

- Authenticating users with IdentityServer is achieved by the mobile app requesting an *identity* token, which represents the outcome of an authentication process. At a bare minimum, it contains an identifier for the user, and information about how and when the user authenticated. It can also contain additional identity data.
- Accessing a resource with IdentityServer is achieved by the mobile app requesting an *access* token, which allows access to an API resource. Clients request access tokens and forward them to the API. Access tokens contain information about the client, and the user (if present). APIs then use that information to authorize access to their data.

> **Note**: A client must be registered with IdentityServer before it can request tokens.

Adding IdentityServer to a web application

In order for an ASP.NET Core web application to use IdentityServer 4, it must be added to the web application's Visual Studio solution. For more information, see Setup and Overview in the IdentityServer documentation.

Once IdentityServer is included in the web application's Visual Studio solution, it must be added to the web application's HTTP request processing pipeline, so that it can serve requests to OpenID Connect and OAuth 2.0 endpoints. This is achieved in the `Configure` method in the web application's `Startup` class, as demonstrated in the following code example:

```
public void Configure(
    IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    ...
    app.UseIdentity();
    ...
}
```

Order matters in the web application's HTTP request processing pipeline. Therefore, IdentityServer must be added to the pipeline before the UI framework that implements the login screen.

Configuring IdentityServer

IdentityServer should be configured in the `ConfigureServices` method in the web application's `Startup` class by calling the `services.AddIdentityServer` method, as demonstrated in the following code example from the eShopOnContainers reference application:

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddIdentityServer(x => x.IssuerUri = "null")
        .AddSigningCredential(Certificate.Get())
        .AddAspNetIdentity<ApplicationUser>()
        .AddConfigurationStore(builder =>
            builder.UseSqlServer(connectionString, options =>
                options.MigrationsAssembly(migrationsAssembly)))
        .AddOperationalStore(builder =>
            builder.UseSqlServer(connectionString, options =>
                options.MigrationsAssembly(migrationsAssembly)))
        .Services.AddTransient<IProfileService, ProfileService>();
}
```

After calling the `services.AddIdentityServer` method, additional fluent APIs are called to configure the following:

- Credentials used for signing.
- API and identity resources that users might request access to.
- Clients that will be connecting to request tokens.
- ASP.NET Core Identity.

Tip: Dynamically load the IdentityServer 4 configuration

IdentityServer 4's APIs allow for configuring IdentityServer from an in-memory list of configuration objects. In the eShopOnContainers reference application, these in-memory collections are hard-coded into the application. However, in production scenarios they can be loaded dynamically from a configuration file or from a database.

For information about configuring IdentityServer to use ASP.NET Core Identity, see Using ASP.NET Core Identity in the IdentityServer documentation.

Configuring API resources

When configuring API resources, the `AddInMemoryApiResources` method expects an `IEnumerable<ApiResource>` collection. The following code example shows the `GetApis` method that provides this collection in the eShopOnContainers reference application:

```
public static IEnumerable<ApiResource> GetApis()
{
    return new List<ApiResource>
    {
        new ApiResource("orders", "Orders Service"),
        new ApiResource("basket", "Basket Service")
    };
}
```

This method specifies that IdentityServer should protect the `orders` and `basket` APIs. Therefore, IdentityServer managed access tokens will be required when making calls to these APIs. For more information about the `ApiResource` type, see API Resource in the IdentityServer 4 documentation.

Configuring identity resources

When configuring identity resources, the `AddInMemoryIdentityResources` method expects an `IEnumerable<IdentityResource>` collection. Identity resources are data such as user ID, name, or email address. Each identity resource has a unique name, and arbitrary claim types can be assigned to it, which will then be included in the identity token for the user. The following code example shows the `GetResources` method that provides this collection in the eShopOnContainers reference application:

```
public static IEnumerable<IdentityResource> GetResources()
{
    return new List<IdentityResource>
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Profile()
    };
}
```

The OpenID Connect specification specifies some standard identity resources. The minimum requirement is that support is provided for emitting a unique ID for users. This is achieved by exposing the `IdentityResources.OpenId` identity resource.

> **Note:** The `IdentityResources` class supports all of the scopes defined in the OpenID Connect specification (openid, email, profile, telephone, and address).

IdentityServer also supports defining custom identity resources. For more information, see Defining custom identity resources in the IdentityServer documentation. For more information about the `IdentityResource` type, see Identity Resource in the IdentityServer 4 documentation.

Configuring clients

Clients are applications that can request tokens from IdentityServer. Typically, the following settings must be defined for each client as a minimum:

- A unique client ID.
- The allowed interactions with the token service (known as the grant type).
- The location where identity and access tokens are sent to (known as a redirect URI).

- A list of resources that the client is allowed access to (known as scopes).

When configuring clients, the `AddInMemoryClients` method expects an `IEnumerable<Client>` collection. The following code example shows the configuration for the eShopOnContainers mobile app in the `GetClients` method that provides this collection in the eShopOnContainers reference application:

```
public static IEnumerable<Client> GetClients(Dictionary<string,string> clientsUrl)
{
    return new List<Client>
    {
        ...
        new Client
        {
            ClientId = "xamarin",
            ClientName = "eShop Xamarin OpenId Client",
            AllowedGrantTypes = GrantTypes.Hybrid,
            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },
            RedirectUris = { clientsUrl["Xamarin"] },
            RequireConsent = false,
            RequirePkce = true,
            PostLogoutRedirectUris = { $"{clientsUrl["Xamarin"]}/Account/Redirecting" },
            AllowedCorsOrigins = { "http://eshopxamarin" },
            AllowedScopes = new List<string>
            {
                IdentityServerConstants.StandardScopes.OpenId,
                IdentityServerConstants.StandardScopes.Profile,
                IdentityServerConstants.StandardScopes.OfflineAccess,
                "orders",
                "basket"
            },
            AllowOfflineAccess = true,
            AllowAccessTokensViaBrowser = true
        },
        ...
    };
}
```

This configuration specifies data for the following properties:

- `ClientId`: A unique ID for the client.
- `ClientName`: The client display name, which is used for logging and the consent screen.
- `AllowedGrantTypes`: Specifies how a client wants to interact with IdentityServer. For more information see Configuring the authentication flow.
- `ClientSecrets`: Specifies the client secret credentials that are used when requesting tokens from the token endpoint.
- `RedirectUris`: Specifies the allowed URIs to which to return tokens or authorization codes.
- `RequireConsent`: Specifies whether a consent screen is required.
- `RequirePkce`: Specifies whether clients using an authorization code must send a proof key.
- `PostLogoutRedirectUris`: Specifies the allowed URIs to redirect to after logout.

- `AllowedCorsOrigins`: Specifies the origin of the client so that IdentityServer can allow cross-origin calls from the origin.
- `AllowedScopes`: Specifies the resources the client has access to. By default, a client has no access to any resources.
- `AllowOfflineAccess`: Specifies whether the client can request refresh tokens.

Configuring the authentication flow

The authentication flow between a client and IdentityServer can be configured by specifying the grant types in the `Client.AllowedGrantTypes` property. The OpenID Connect and OAuth 2.0 specifications define a number of authentication flows, including:

- Implicit. This flow is optimized for browser-based applications and should be used either for user authentication-only, or authentication and access token requests. All tokens are transmitted via the browser, and therefore advanced features like refresh tokens are not permitted.
- Authorization code. This flow provides the ability to retrieve tokens on a back channel, as opposed to the browser front channel, while also supporting client authentication.
- Hybrid. This flow is a combination of the implicit and authorization code grant types. The identity token is transmitted via the browser channel and contains the signed protocol response along with other artifacts such as the authorization code. After successful validation of the response, the back channel should be used to retrieve the access and refresh token.

> **Tip:** Use the hybrid authentication flow
>
> The hybrid authentication flow mitigates a number of attacks that apply to the browser channel, and is the recommended flow for native applications that want to retrieve access tokens (and possibly refresh tokens).

For more information about authentication flows, see Grant Types in the IdentityServer 4 documentation.

Performing authentication

For IdentityServer to issue tokens on behalf of a user, the user must sign-in to IdentityServer. However, IdentityServer doesn't provide a user interface or database for authentication. Therefore, in the eShopOnContainers reference application, ASP.NET Core Identity is used for this purpose.

The eShopOnContainers mobile app authenticates with IdentityServer with the hybrid authentication flow, which is illustrated in Figure 9-2.

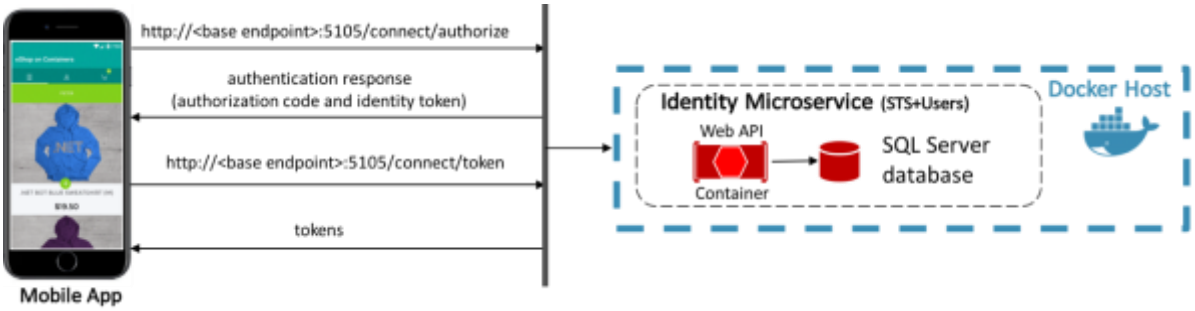

Figure 9-2: High-level overview of the sign-in process

A sign-in request is made to `<base endpoint>:5105/connect/authorize`. Following successful authentication, IdentityServer returns an authentication response containing an authorization code and an identity token. The authorization code is then sent to `<base endpoint>:5105/connect/token`, which responds with access, identity, and refresh tokens.

The eShopOnContainers mobile app signs-out of IdentityServer by sending a request to `http://<base endpoint>:5105/connect/endsession`, with additional parameters. After sign-out occurs, IdentityServer responds by sending a post logout redirect URI back to the mobile app. Figure 9-3 illustrates this process.

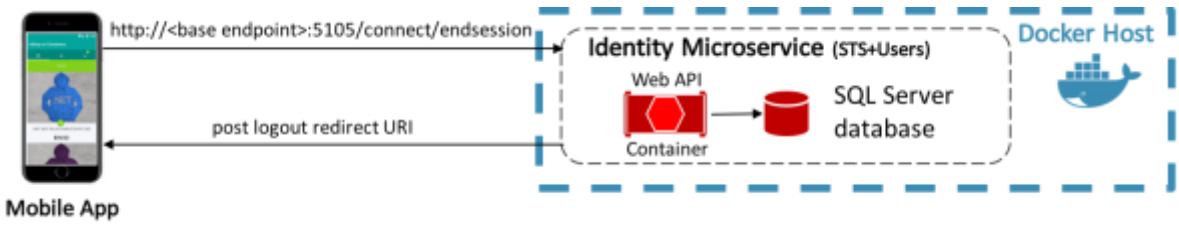

Figure 9-3: High-level overview of the sign-out process

In the eShopOnContainers mobile app, communication with IdentityServer is performed by the `IdentityService` class, which implements the `IIdentityService` interface. This interface specifies that the implementing class must provide `CreateAuthorizationRequest`, `CreateLogoutRequest`, and `GetTokenAsync` methods.

Signing-in

When the user taps the **LOGIN** button on the `LoginView`, the `SignInCommand` in the `LoginViewModel` class is executed, which in turn executes the `SignInAsync` method. The following code example shows this method:

```
private async Task SignInAsync()
{
    ...
    LoginUrl = _identityService.CreateAuthorizationRequest();
    IsLogin = true;
    ...
}
```

This method invokes the `CreateAuthorizationRequest` method in the `IdentityService` class, which is shown in the following code example:

```
public string CreateAuthorizationRequest()
{
    // Create URI to authorization endpoint
    var authorizeRequest = new AuthorizeRequest(GlobalSetting.Instance.IdentityEndpoint);

    // Dictionary with values for the authorize request
    var dic = new Dictionary<string, string>();
    dic.Add("client_id", GlobalSetting.Instance.ClientId);
    dic.Add("client_secret", GlobalSetting.Instance.ClientSecret);
    dic.Add("response_type", "code id_token");
    dic.Add("scope", "openid profile basket orders locations marketing offline_access");
    dic.Add("redirect_uri", GlobalSetting.Instance.IdentityCallback);
    dic.Add("nonce", Guid.NewGuid().ToString("N"));
    dic.Add("code_challenge", CreateCodeChallenge());
    dic.Add("code_challenge_method", "S256");
```

```
    // Add CSRF token to protect against cross-site request forgery attacks.
    var currentCSRFToken = Guid.NewGuid().ToString("N");
    dic.Add("state", currentCSRFToken);

    var authorizeUri = authorizeRequest.Create(dic);
    return authorizeUri;
}
```

This method creates the URI for IdentityServer's authorization endpoint, with the required parameters. The authorization endpoint is at `/connect/authorize` on port 5105 of the base endpoint exposed as a user setting. For more information about user settings, see Configuration management.

Note: The attack surface of the eShopOnContainers mobile app is reduced by implementing the Proof Key for Code Exchange (PKCE) extension to OAuth. PKCE protects the authorization code from being used if it's intercepted. This is achieved by the client generating a secret verifier, a hash of which is passed in the authorization request, and which is presented unhashed when redeeming the authorization code. For more information about PKCE, see Proof Key for Code Exchange by OAuth Public Clients on the Internet Engineering Task Force web site.

The returned URI is stored in the `LoginUrl` property of the `LoginViewModel` class. When the `IsLogin` property becomes `true`, the `WebView` in the `LoginView` becomes visible. The `WebView` data binds its `Source` property to the `LoginUrl` property of the `LoginViewModel` class, and so makes a sign-in request to IdentityServer when the `LoginUrl` property is set to IdentityServer's authorization endpoint. When IdentityServer receives this request and the user isn't authenticated, the `WebView` will be redirected to the configured login page, which is shown in Figure 9-4.

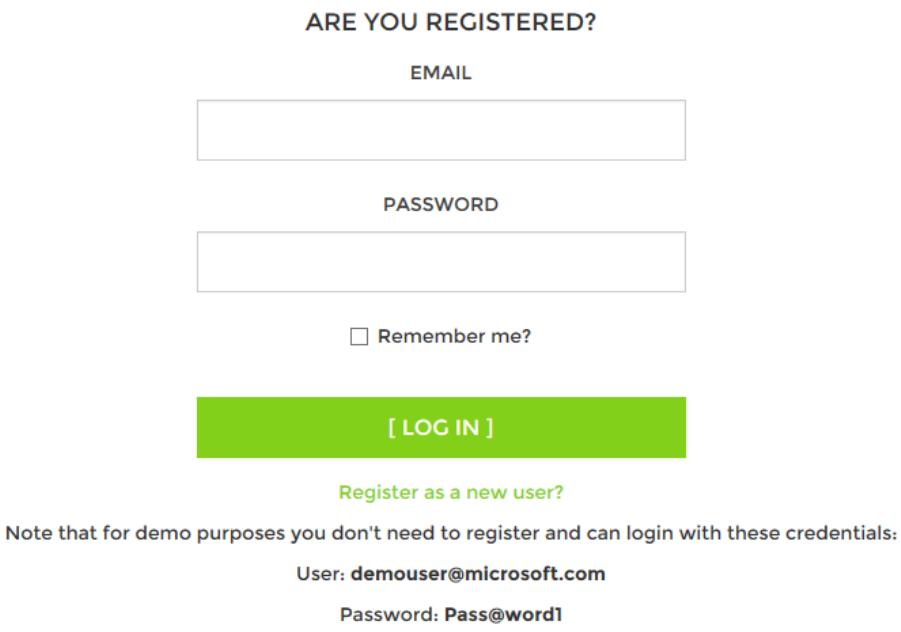

Figure 9-4: Login page displayed by the WebView

Once login is completed, the `WebView` will be redirected to a return URI. This `WebView` navigation will cause the `NavigateAsync` method in the `LoginViewModel` class to be executed, which is shown in the following code example:

```
private async Task NavigateAsync(string url)
{
    ...
    var authResponse = new AuthorizeResponse(url);
    if (!string.IsNullOrWhiteSpace(authResponse.Code))
    {
        var userToken = await _identityService.GetTokenAsync(authResponse.Code);
        string accessToken = userToken.AccessToken;

        if (!string.IsNullOrWhiteSpace(accessToken))
        {
            Settings.AuthAccessToken = accessToken;
            Settings.AuthIdToken = authResponse.IdentityToken;

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

This method parses the authentication response that's contained in the return URI, and provided that a valid authorization code is present, it makes a request to IdentityServer's token endpoint, passing the authorization code, the PKCE secret verifier, and other required parameters. The token endpoint is at `/connect/token` on port 5105 of the base endpoint exposed as a user setting. For more information about user settings, see Configuration management.

> **Tip**: Validate return URIs
>
> Although the eShopOnContainers mobile app doesn't validate the return URI, the best practice is to validate that the return URI refers to a known location in order to prevent open-redirect attacks.

If the token endpoint receives a valid authorization code and PKCE secret verifier, it responds with an access token, identity token, and refresh token. The access token (which allows access to API resources) and identity token are then stored as application settings, and page navigation is performed. Therefore, the overall effect in the eShopOnContainers mobile app is this: provided that users are able to successfully authenticate with IdentityServer, they are navigated to the `MainView` page, which is a `TabbedPage` that displays the `CatalogView` as its selected tab.

For information about page navigation, see Navigation. For information about how `WebView` navigation causes a view model method to be executed, see Invoking navigation using behaviors. For information about application settings, see Configuration management.

> **Note:** The eShopOnContainers also allows a mock sign-in when the app is configured to use mock services in the `SettingsView`. In this mode, the app doesn't communicate with IdentityServer, instead allowing the user to sign-in using any credentials.

Signing-out

When the user taps the **LOG OUT** button in the `ProfileView`, the `LogoutCommand` in the `ProfileViewModel` class is executed, which in turn executes the `LogoutAsync` method. This method performs page navigation to the `LoginView` page, passing a `LogoutParameter` instance set to `true` as a parameter. For more information about passing parameters during page navigation, see Passing parameters during navigation.

When a view is created and navigated to, the `InitializeAsync` method of the view's associated view model is executed, which then executes the `Logout` method of the `LoginViewModel` class, which is shown in the following code example:

```
private void Logout()
{
    var authIdToken = Settings.AuthIdToken;
    var logoutRequest = _identityService.CreateLogoutRequest(authIdToken);

    if (!string.IsNullOrEmpty(logoutRequest))
    {
        // Logout
        LoginUrl = logoutRequest;
    }
    ...
}
```

This method invokes the `CreateLogoutRequest` method in the `IdentityService` class, passing the identity token retrieved from application settings as a parameter. For more information about application settings, see Configuration management. The following code example shows the `CreateLogoutRequest` method:

```
public string CreateLogoutRequest(string token)
{
    ...
    return string.Format("{0}?id_token_hint={1}&post_logout_redirect_uri={2}",
        GlobalSetting.Instance.LogoutEndpoint,
        token,
        GlobalSetting.Instance.LogoutCallback);
}
```

This method creates the URI to IdentityServer's end session endpoint, with the required parameters. The end session endpoint is at `/connect/endsession` on port 5105 of the base endpoint exposed as a user setting. For more information about user settings, see Configuration management.

The returned URI is stored in the `LoginUrl` property of the `LoginViewModel` class. While the `IsLogin` property is `true`, the `WebView` in the `LoginView` is visible. The `WebView` data binds its `Source` property to the `LoginUrl` property of the `LoginViewModel` class, and so makes a sign-out request to IdentityServer when the `LoginUrl` property is set to IdentityServer's end session endpoint. When IdentityServer receives this request, provided that the user is signed-in, sign-out occurs. Authentication is tracked with a cookie managed by the cookie authentication middleware from ASP.NET Core. Therefore, signing out of IdentityServer removes the authentication cookie and sends a post logout redirect URI back to the client.

In the mobile app, the `WebView` will be redirected to the post logout redirect URI. This `WebView` navigation will cause the `NavigateAsync` method in the `LoginViewModel` class to be executed, which is shown in the following code example:

```
private async Task NavigateAsync(string url)
{
    ...
    Settings.AuthAccessToken = string.Empty;
    Settings.AuthIdToken = string.Empty;
    IsLogin = false;
    LoginUrl = _identityService.CreateAuthorizationRequest();
    ...
}
```

This method clears both the identity token and the access token from application settings, and sets the `IsLogin` property to `false`, which causes the `WebView` on the `LoginView` page to become invisible. Finally, the `LoginUrl` property is set to the URI of IdentityServer's authorization endpoint, with the required parameters, in preparation for the next time the user initiates a sign-in.

For information about page navigation, see Navigation. For information about how `WebView` navigation causes a view model method to be executed, see Invoking navigation using behaviors. For information about application settings, see Configuration management.

> **Note:** The eShopOnContainers also allows a mock sign-out when the app is configured to use mock services in the `SettingsView`. In this mode, the app doesn't communicate with IdentityServer, and instead clears any stored tokens from application settings.

Authorization

After authentication, ASP.NET Core web APIs often need to authorize access, which allows a service to make APIs available to some authenticated users, but not to all.

Restricting access to an ASP.NET Core MVC route can be achieved by applying an `Authorize` attribute to a controller or action, which limits access to the controller or action to authenticated users, as shown in the following code example:

```
[Authorize]
public class BasketController : Controller
{
    ...
}
```

If an unauthorized user attempts to access a controller or action that's marked with the `Authorize` attribute, the MVC framework returns a 401 (unauthorized) HTTP status code.

> **Note:** Parameters can be specified on the `Authorize` attribute to restrict an API to specific users. For more information, see Authorization on the Microsoft Documentation Center.

IdentityServer can be integrated into the authorization workflow so that the access tokens it provides control authorization. This approach is shown in Figure 9-5.

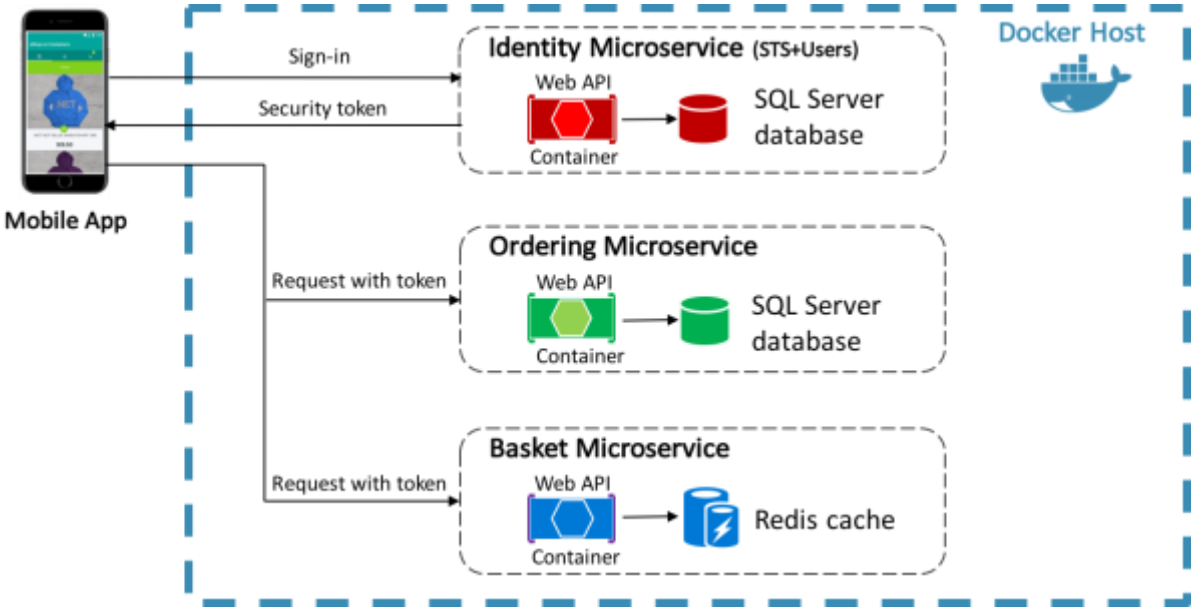

Figure 9-5: Authorization by access token

The eShopOnContainers mobile app communicates with the identity microservice and requests an access token as part of the authentication process. The access token is then forwarded to the APIs exposed by the ordering and basket microservices as part of the access requests. Access tokens contain information about the client, and the user. APIs then use that information to authorize access to their data. For information about how to configure IdentityServer to protect APIs, see Configuring API resources.

Configuring IdentityServer to perform authorization

To perform authorization with IdentityServer, its authorization middleware must be added to the web application's HTTP request pipeline. The middleware is added in the `ConfigureAuth` method in the web application's `Startup` class, which is invoked from the `Configure` method, and is demonstrated in the following code example from the eShopOnContainers reference application:

```
protected virtual void ConfigureAuth(IApplicationBuilder app)
{
    var identityUrl = Configuration.GetValue<string>("IdentityUrl");
    app.UseIdentityServerAuthentication(new IdentityServerAuthenticationOptions
    {
        Authority = identityUrl.ToString(),
        ScopeName = "basket",
        RequireHttpsMetadata = false
    });
}
```

This method ensures that the API can only be accessed with a valid access token. The middleware validates the incoming token to ensure that it's sent from a trusted issuer, and validates that the token is valid to be used with the API that receives it. Therefore, browsing to the ordering or basket controller will return a 401 (unauthorized) HTTP status code, indicating that an access token is required.

Note: IdentityServer's authorization middleware must be added to the web application's HTTP request pipeline before adding MVC with `app.UseMvc()` or `app.UseMvcWithDefaultRoute()`.

Making access requests to APIs

When making requests to the ordering and basket microservices, the access token, obtained from IdentityServer during the authentication process, must be included in the request, as shown in the following code example:

```
var authToken = Settings.AuthAccessToken;
Order = await _ordersService.GetOrderAsync(Convert.ToInt32(order.OrderNumber), authToken);
```

The access token is stored as an application setting, and is retrieved from platform-specific storage and included in the call to the `GetOrderAsync` method in the `OrderService` class.

Similarly, the access token must be included when sending data to an IdentityServer protected API, as shown in the following code example:

```
var authToken = Settings.AuthAccessToken;
await _basketService.UpdateBasketAsync(new CustomerBasket
{
    BuyerId = userInfo.UserId,
    Items = BasketItems.ToList()
}, authToken);
```

The access token is retrieved from platform-specific storage and included in the call to the `UpdateBasketAsync` method in the `BasketService` class.

The `RequestProvider` class, in the eShopOnContainers mobile app, uses the `HttpClient` class to make requests to the RESTful APIs exposed by the eShopOnContainers reference application. When making requests to the ordering and basket APIs, which require authorization, a valid access token must be included with the request. This is achieved by adding the access token to the headers of the `HttpClient` instance, as demonstrated in the following code example:

```
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);
```

The `DefaultRequestHeaders` property of the `HttpClient` class exposes the headers that are sent with each request, and the access token is added to the `Authorization` header prefixed with the string `Bearer`. When the request is sent to a RESTful API, the value of the `Authorization` header is extracted and validated to ensure that it's sent from a trusted issuer, and used to determine whether the user has permission to invoke the API that receives it.

For more information about how the eShopOnContainers mobile app makes web requests, see Accessing remote data.

Summary

There are many approaches to integrating authentication and authorization into a Xamarin.Forms app that communicates with an ASP.NET MVC web application. The eShopOnContainers mobile app performs authentication and authorization with a containerized identity microservice that uses IdentityServer 4. IdentityServer is an open source OpenID Connect and OAuth 2.0 framework for ASP.NET Core that integrates with ASP.NET Core Identity to perform bearer token authentication.

The mobile app requests security tokens from IdentityServer, either for authenticating a user or for accessing a resource. When accessing a resource, an access token must be included in the request to APIs that require authorization. IdentityServer's middleware validates incoming access tokens to ensure that they are sent from a trusted issuer, and that they are valid to be used with the API that receives them.

Accessing remote data

Many modern web-based solutions make use of web services, hosted by web servers, to provide functionality for remote client applications. The operations that a web service exposes constitute a web API.

Client apps should be able to utilize the web API without knowing how the data or operations that the API exposes are implemented. This requires that the API abides by common standards that enable a client app and web service to agree on which data formats to use, and the structure of the data that is exchanged between client apps and the web service.

Introduction to Representational State Transfer

Representational State Transfer (REST) is an architectural style for building distributed systems based on hypermedia. A primary advantage of the REST model is that it's based on open standards and doesn't bind the implementation of the model or the client apps that access it to any specific implementation. Therefore, a REST web service could be implemented using Microsoft ASP.NET Core MVC, and client apps could be developing using any language and toolset that can generate HTTP requests and parse HTTP responses.

The REST model uses a navigational scheme to represent objects and services over a network, referred to as resources. Systems that implement REST typically use the HTTP protocol to transmit requests to access these resources. In such systems, a client app submits a request in the form of a URI that identifies a resource, and an HTTP method (such as GET, POST, PUT, or DELETE) that indicates the operation to be performed on that resource. The body of the HTTP request contains any data required to perform the operation.

> **Note:** REST defines a stateless request model. Therefore, HTTP requests must be independent and might occur in any order.

The response from a REST request makes use of standard HTTP status codes. For example, a request that returns valid data should include the HTTP response code 200 (OK), while a request that fails to find or delete a specified resource should return a response that includes the HTTP status code 404 (Not Found).

A RESTful web API exposes a set of connected resources, and provides the core operations that enable an app to manipulate those resources and easily navigate between them. For this reason, the URIs that constitute a typical RESTful web API are oriented towards the data that it exposes, and use the facilities provided by HTTP to operate on this data.

The data included by a client app in an HTTP request, and the corresponding response messages from the web server, could be presented in a variety of formats, known as media types. When a client app sends a request that returns data in the body of a message, it can specify the media types it can handle in the `Accept` header of the request. If the web server supports this media type, it can reply with a response that includes the `Content-Type` header that specifies the format of the data in the body of the message. It's then the responsibility of the client app to parse the response message and interpret the results in the message body appropriately.

For more information about REST, see API design and API implementation on Microsoft Docs.

Consuming RESTful APIs

The eShopOnContainers mobile app uses the Model-View-ViewModel (MVVM) pattern, and the model elements of the pattern represent the domain entities used in the app. The controller and repository classes in the eShopOnContainers reference application accept and return many of these model objects. Therefore, they are used as data transfer objects (DTOs) that hold all the data that is passed between the mobile app and the containerized microservices. The main benefit of using DTOs to pass data to and receive data from a web service is that by transmitting more data in a single remote call, the app can reduce the number of remote calls that need to be made.

Making web requests

The eShopOnContainers mobile app uses the `HttpClient` class to make requests over HTTP, with JSON being used as the media type. This class provides functionality for asynchronously sending HTTP requests and receiving HTTP responses from a URI identified resource. The `HttpResponseMessage` class represents an HTTP response message received from a REST API after an HTTP request has been made. It contains information about the response, including the status code, headers, and any body. The `HttpContent` class represents the HTTP body and content headers, such as `Content-Type` and `Content-Encoding`. The content can be read using any of the `ReadAs` methods, such as `ReadAsStringAsync` and `ReadAsByteArrayAsync`, depending on the format of the data.

Making a GET request

The `CatalogService` class is used to manage the data retrieval process from the catalog microservice. In the `RegisterDependencies` method in the `ViewModelLocator` class, the `CatalogService` class is registered as a type mapping against the `ICatalogService` type with the Autofac dependency injection container. Then, when an instance of the `CatalogViewModel` class is created, its constructor accepts an `ICatalogService` type, which Autofac resolves, returning an instance of the `CatalogService` class. For more information about dependency injection, see Introduction to dependency injection.

Figure 10-1 shows the interaction of classes that read catalog data from the catalog microservice for displaying by the `CatalogView`.

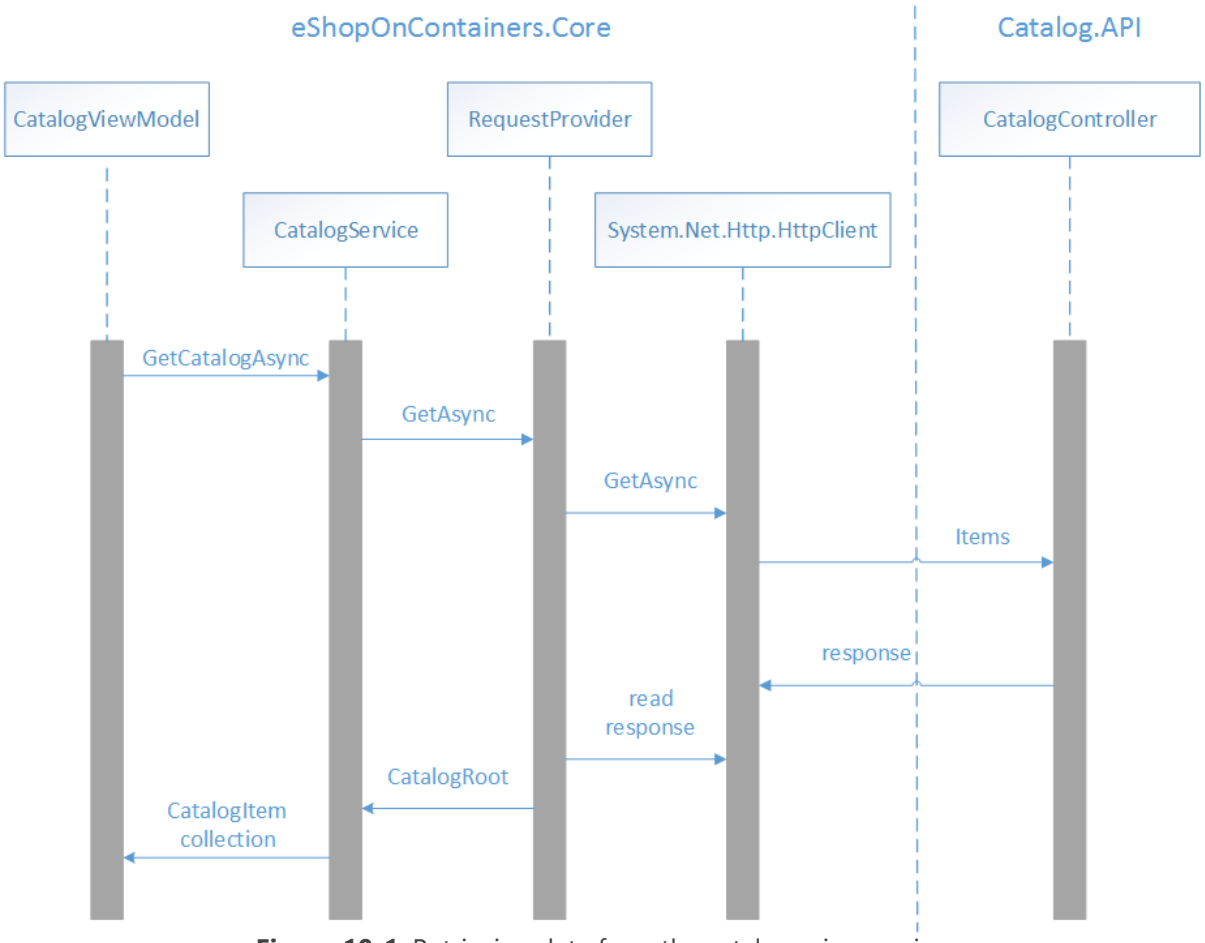

Figure 10-1: Retrieving data from the catalog microservice

When the `CatalogView` is navigated to, the `OnInitialize` method in the `CatalogViewModel` class is called. This method retrieves catalog data from the catalog microservice, as demonstrated in the following code example:

```
public override async Task InitializeAsync(object navigationData)
{
    ...
    Products = await _productsService.GetCatalogAsync();
    ...
}
```

This method calls the `GetCatalogAsync` method of the `CatalogService` instance that was injected into the `CatalogViewModel` by Autofac. The following code example shows the `GetCatalogAsync` method:

```
public async Task<ObservableCollection<CatalogItem>> GetCatalogAsync()
{
    UriBuilder builder = new UriBuilder(GlobalSetting.Instance.CatalogEndpoint);
    builder.Path = "api/v1/catalog/items";
    string uri = builder.ToString();

    CatalogRoot catalog = await _requestProvider.GetAsync<CatalogRoot>(uri);
    ...
```

```
    return catalog?.Data.ToObservableCollection();
}
```

This method builds the URI that identifies the resource the request will be sent to, and uses the `RequestProvider` class to invoke the GET HTTP method on the resource, before returning the results to the `CatalogViewModel`. The `RequestProvider` class contains functionality that submits a request in the form of a URI that identifies a resource, an HTTP method that indicates the operation to be performed on that resource, and a body that contains any data required to perform the operation. For information about how the `RequestProvider` class is injected into the `CatalogService` class, see Introduction to dependency injection.

The following code example shows the `GetAsync` method in the `RequestProvider` class:

```
public async Task<TResult> GetAsync<TResult>(string uri, string token = "")
{
    HttpClient httpClient = CreateHttpClient(token);
    HttpResponseMessage response = await httpClient.GetAsync(uri);

    await HandleResponse(response);
    string serialized = await response.Content.ReadAsStringAsync();

    TResult result = await Task.Run(() =>
        JsonConvert.DeserializeObject<TResult>(serialized, _serializerSettings));

    return result;
}
```

This method calls the `CreateHttpClient` method, which returns an instance of the `HttpClient` class with the appropriate headers set. It then submits an asynchronous GET request to the resource identified by the URI, with the response being stored in the `HttpResponseMessage` instance. The `HandleResponse` method is then invoked, which throws an exception if the response doesn't include a success HTTP status code. Then the response is read as a string, converted from JSON to a `CatalogRoot` object, and returned to the `CatalogService`.

The `CreateHttpClient` method is shown in the following code example:

```
private HttpClient CreateHttpClient(string token = "")
{
    var httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json"));

    if (!string.IsNullOrEmpty(token))
    {
        httpClient.DefaultRequestHeaders.Authorization =
            new AuthenticationHeaderValue("Bearer", token);
    }
    return httpClient;
}
```

This method creates a new instance of the `HttpClient` class, and sets the `Accept` header of any requests made by the `HttpClient` instance to `application/json`, which indicates that it expects the content of any response to be formatted using JSON. Then, if an access token was passed as an argument to the `CreateHttpClient` method, it's added to the `Authorization` header of any requests made by the `HttpClient` instance, prefixed with the string `Bearer`. For more information about authorization, see Authorization.

When the `GetAsync` method in the `RequestProvider` class calls `HttpClient.GetAsync`, the `Items` method in the `CatalogController` class in the `Catalog.API` project is invoked, which is shown in the following code example:

```
[HttpGet]
[Route("[action]")]
public async Task<IActionResult> Items(
    [FromQuery]int pageSize = 10, [FromQuery]int pageIndex = 0)
{
    var totalItems = await _catalogContext.CatalogItems
        .LongCountAsync();

    var itemsOnPage = await _catalogContext.CatalogItems
        .OrderBy(c=>c.Name)
        .Skip(pageSize * pageIndex)
        .Take(pageSize)
        .ToListAsync();

    itemsOnPage = ComposePicUri(itemsOnPage);
    var model = new PaginatedItemsViewModel<CatalogItem>(
        pageIndex, pageSize, totalItems, itemsOnPage);

    return Ok(model);
}
```

This method retrieves the catalog data from the SQL database using EntityFramework, and returns it as a response message that includes a success HTTP status code, and a collection of JSON formatted `CatalogItem` instances.

Making a POST request

The `BasketService` class is used to manage the data retrieval and update process with the basket microservice. In the `RegisterDependencies` method in the `ViewModelLocator` class, the `BasketService` class is registered as a type mapping against the `IBasketService` type with the Autofac dependency injection container. Then, when an instance of the `BasketViewModel` class is created, its constructor accepts an `IBasketService` type, which Autofac resolves, returning an instance of the `BasketService` class. For more information about dependency injection, see Introduction to dependency injection.

Figure 10-2 shows the interaction of classes that send the basket data displayed by the `BasketView`, to the basket microservice.

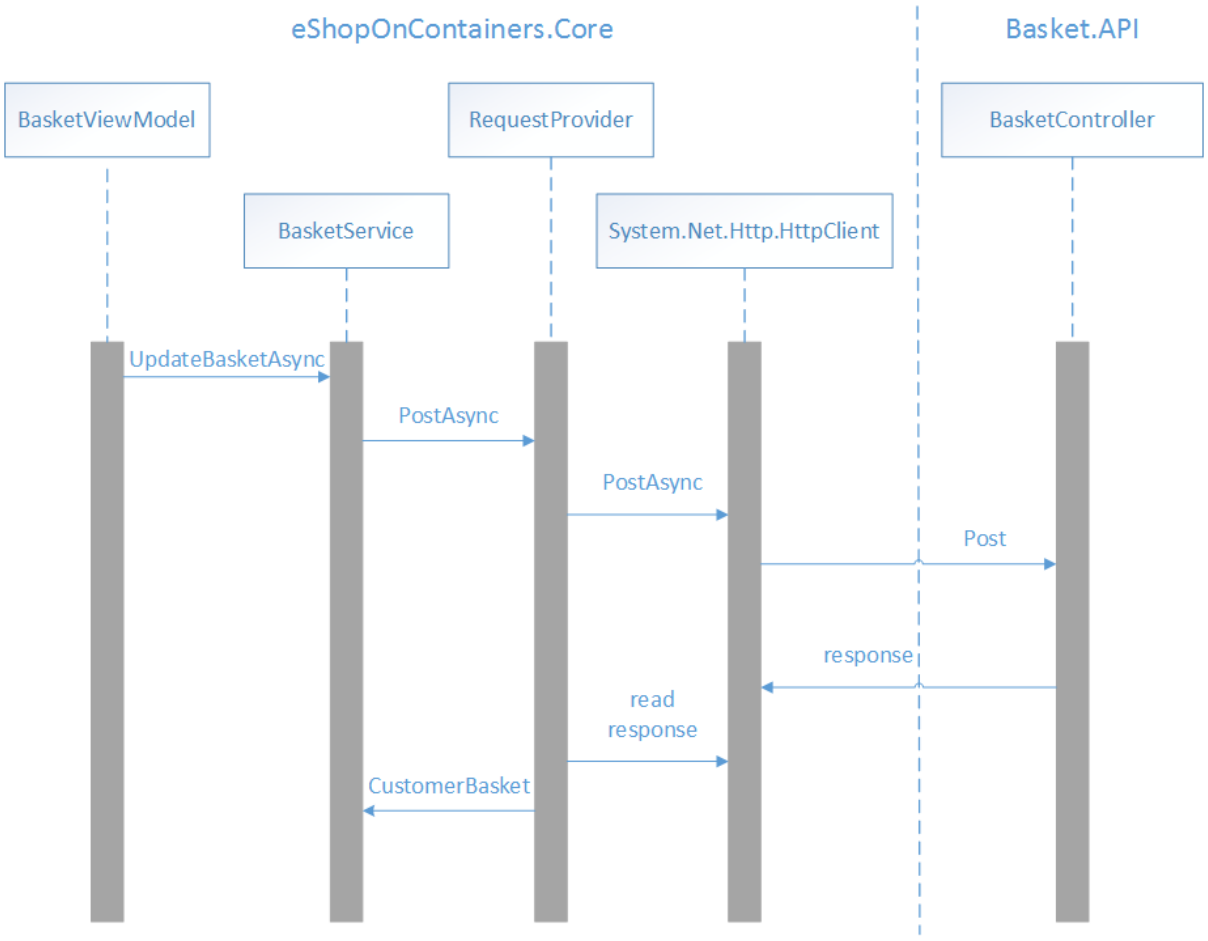

Figure 10-2: Sending data to the basket microservice

When an item is added to the shopping basket, the `ReCalculateTotalAsync` method in the `BasketViewModel` class is called. This method updates the total value of items in the basket, and sends the basket data to the basket microservice, as demonstrated in the following code example:

```
private async Task ReCalculateTotalAsync()
{
    ...
    await _basketService.UpdateBasketAsync(new CustomerBasket
    {
        BuyerId = userInfo.UserId,
        Items = BasketItems.ToList()
    }, authToken);
}
```

This method calls the `UpdateBasketAsync` method of the `BasketService` instance that was injected into the `BasketViewModel` by Autofac. The following method shows the `UpdateBasketAsync` method:

```
public async Task<CustomerBasket> UpdateBasketAsync(CustomerBasket customerBasket, string token)
{
    UriBuilder builder = new UriBuilder(GlobalSetting.Instance.BasketEndpoint);
    string uri = builder.ToString();
```

```
    var result = await _requestProvider.PostAsync(uri, customerBasket, token);
    return result;
}
```

This method builds the URI that identifies the resource the request will be sent to, and uses the `RequestProvider` class to invoke the POST HTTP method on the resource, before returning the results to the `BasketViewModel`. Note that an access token, obtained from IdentityServer during the authentication process, is required to authorize requests to the basket microservice. For more information about authorization, see Authorization.

The following code example shows one of the `PostAsync` methods in the `RequestProvider` class:

```
public async Task<TResult> PostAsync<TResult>(
    string uri, TResult data, string token = "", string header = "")
{
    HttpClient httpClient = CreateHttpClient(token);
    ...
    var content = new StringContent(JsonConvert.SerializeObject(data));
    content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
    HttpResponseMessage response = await httpClient.PostAsync(uri, content);

    await HandleResponse(response);
    string serialized = await response.Content.ReadAsStringAsync();

    TResult result = await Task.Run(() =>
        JsonConvert.DeserializeObject<TResult>(serialized, _serializerSettings));

    return result;
}
```

This method calls the `CreateHttpClient` method, which returns an instance of the `HttpClient` class with the appropriate headers set. It then submits an asynchronous POST request to the resource identified by the URI, with the serialized basket data being sent in JSON format, and the response being stored in the `HttpResponseMessage` instance. The `HandleResponse` method is then invoked, which throws an exception if the response doesn't include a success HTTP status code. Then, the response is read as a string, converted from JSON to a `CustomerBasket` object, and returned to the `BasketService`. For more information about the CreateHttpClient method, see Making a GET request.

When the `PostAsync` method in the `RequestProvider` class calls `HttpClient.PostAsync`, the `Post` method in the `BasketController` class in the `Basket.API` project is invoked, which is shown in the following code example:

```
[HttpPost]
public async Task<IActionResult> Post([FromBody]CustomerBasket value)
{
    var basket = await _repository.UpdateBasketAsync(value);
    return Ok(basket);
}
```

This method uses an instance of the `RedisBasketRepository` class to persist the basket data to the Redis cache, and returns it as a response message that includes a success HTTP status code, and a JSON formatted `CustomerBasket` instance.

Making a DELETE request

Figure 10-3 shows the interactions of classes that delete basket data from the basket microservice, for the `CheckoutView`.

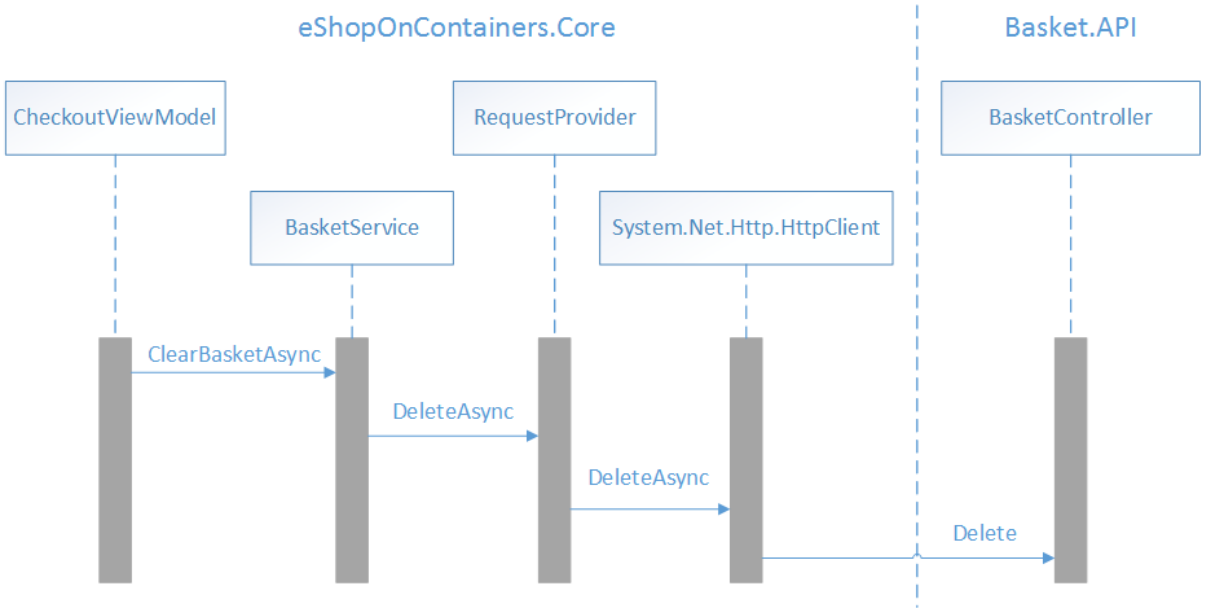

Figure 10-3: Deleting data from the basket microservice

When the checkout process is invoked, the `CheckoutAsync` method in the `CheckoutViewModel` class is called. This method creates a new order, before clearing the shopping basket, as demonstrated in the following code example:

```
private async Task CheckoutAsync()
{
    ...
    await _basketService.ClearBasketAsync(_shippingAddress.Id.ToString(), authToken);
    ...
}
```

This method calls the `ClearBasketAsync` method of the `BasketService` instance that was injected into the `CheckoutViewModel` by Autofac. The following method shows the `ClearBasketAsync` method:

```
public async Task ClearBasketAsync(string guidUser, string token)
{
    UriBuilder builder = new UriBuilder(GlobalSetting.Instance.BasketEndpoint);
    builder.Path = guidUser;
    string uri = builder.ToString();
    await _requestProvider.DeleteAsync(uri, token);
}
```

This method builds the URI that identifies the resource that the request will be sent to, and uses the `RequestProvider` class to invoke the DELETE HTTP method on the resource. Note that an access token, obtained from IdentityServer during the authentication process, is required to authorize requests to the basket microservice. For more information about authorization, see Authorization.

The following code example shows the `DeleteAsync` method in the `RequestProvider` class:

```
public async Task DeleteAsync(string uri, string token = "")
{
    HttpClient httpClient = CreateHttpClient(token);
    await httpClient.DeleteAsync(uri);
}
```

This method calls the `CreateHttpClient` method, which returns an instance of the `HttpClient` class with the appropriate headers set. It then submits an asynchronous DELETE request to the resource identified by the URI. For more information about the `CreateHttpClient` method, see Making a GET request.

When the `DeleteAsync` method in the `RequestProvider` class calls `HttpClient.DeleteAsync`, the `Delete` method in the `BasketController` class in the `Basket.API` project is invoked, which is shown in the following code example:

```
[HttpDelete("{id}")]
public void Delete(string id)
{
    _repository.DeleteBasketAsync(id);
}
```

This method uses an instance of the `RedisBasketRepository` class to delete the basket data from the Redis cache.

Caching data

The performance of an app can be improved by caching frequently accessed data to fast storage that's located close to the app. If the fast storage is located closer to the app than the original source, then caching can significantly improve response times when retrieving data.

The most common form of caching is read-through caching, where an app retrieves data by referencing the cache. If the data isn't in the cache, it's retrieved from the data store and added to the cache. Apps can implement read-through caching with the cache-aside pattern. This pattern determines whether the item is currently in the cache. If the item isn't in the cache, it's read from the data store and added to the cache. For more information, see the Cache-Aside pattern on Microsoft Docs.

> **Tip:** Cache data that's read frequently and that changes infrequently
>
> This data can be added to the cache on demand the first time it is retrieved by an app. This means that the app needs to fetch the data only once from the data store, and that subsequent access can be satisfied by using the cache.

Distributed applications, such as the eShopOnContainers reference application, should provide either or both of the following caches:

- A shared cache, which can be accessed by multiple processes or machines.
- A private cache, where data is held locally on the device running the app.

The eShopOnContainers mobile app uses a private cache, where data is held locally on the device that's running an instance of the app. For information about the cache used by the eShopOnContainers reference application, see .NET Microservices: Architecture for Containerized .NET Applications.

> **Tip:** Think of the cache as a transient data store that could disappear at any time
>
> Ensure that data is maintained in the original data store as well as the cache. The chances of losing data are then minimized if the cache becomes unavailable.

Managing data expiration

It's impractical to expect that cached data will always be consistent with the original data. Data in the original data store might change after it's been cached, causing the cached data to become stale. Therefore, apps should implement a strategy that helps to ensure that the data in the cache is as up-to-date as possible, but can also detect and handle situations that arise when the data in the cache has become stale. Most caching mechanisms enable the cache to be configured to expire data, and hence reduce the period for which data might be out of date.

> **Tip:** Set a default expiration time when configuring a cache
>
> Many caches implement expiration, which invalidates data and removes it from the cache if it's not accessed for a specified period. However, care must be taken when choosing the expiration period. If it's made too short, data will expire too quickly and the benefits of caching will be reduced. If it's made too long, the data risks becoming stale. Therefore, the expiration time should match the pattern of access for apps that use the data.

When cached data expires, it should be removed from the cache, and the app must retrieve the data from the original data store and place it back into the cache.

It's also possible that a cache might fill up if data is allowed to remain for too long a period. Therefore, requests to add new items to the cache might be required to remove some items in a process known as *eviction*. Caching services typically evict data on a least-recently-used basis. However, there are other eviction policies, including most-recently-used, and first-in-first-out. For more information, see Caching Guidance on Microsoft Docs.

Caching images

The eShopOnContainers mobile app consumes remote product images that benefit from being cached. These images are displayed by the `Image` control, and the `CachedImage` control provided by the FFImageLoading library.

The Xamarin.Forms `Image` control supports caching of downloaded images. Caching is enabled by default, and will store the image locally for 24 hours. In addition, the expiration time can be configured with the `CacheValidity` property. For more information, see Downloaded Image Caching on the Xamarin Developer Center.

FFImageLoading's `CachedImage` control is a replacement for the Xamarin.Forms `Image` control, providing additional properties that enable supplementary functionality. Amongst this functionality, the control provides configurable caching, while supporting error and loading image placeholders. The following code example shows how the eShopOnContainers mobile app uses the `CachedImage` control in the `ProductTemplate`, which is the data template used by the `ListView` control in the `CatalogView`:

```
<ffimageloading:CachedImage
    Grid.Row="0"
    Source="{Binding PictureUri}"
    Aspect="AspectFill">
    <ffimageloading:CachedImage.LoadingPlaceholder>
        <OnPlatform
            x:TypeArguments="ImageSource"
```

```
            iOS="default_product"
            Android="default_product"
            WinPhone="Assets/default_product.png"/>
    </ffimageloading:CachedImage.LoadingPlaceholder>
    <ffimageloading:CachedImage.ErrorPlaceholder>
        <OnPlatform
            x:TypeArguments="ImageSource"
            iOS="noimage"
            Android="noimage"
            WinPhone="Assets/noimage.png"/>
    </ffimageloading:CachedImage.ErrorPlaceholder>
</ffimageloading:CachedImage>
```

The `CachedImage` control sets the `LoadingPlaceholder` and `ErrorPlaceholder` properties to platform-specific images. The `LoadingPlaceholder` property specifies the image to be displayed while the image specified by the `Source` property is retrieved, and the `ErrorPlaceholder` property specifies the image to be displayed if an error occurs when attempting to retrieve the image specified by the `Source` property.

As the name implies, the `CachedImage` control caches remote images on the device for the time specified by the value of the `CacheDuration` property. When this property value isn't explicitly set, the default value of 30 days is applied.

Increasing resilience

All apps that communicate with remote services and resources must be sensitive to transient faults. Transient faults include the momentary loss of network connectivity to services, the temporary unavailability of a service, or timeouts that arise when a service is busy. These faults are often self-correcting, and if the action is repeated after a suitable delay it's likely to succeed.

Transient faults can have a huge impact on the perceived quality of an app, even if it has been thoroughly tested under all foreseeable circumstances. To ensure that an app that communicates with remote services operates reliably, it must be able to do all of the following:

- Detect faults when they occur, and determine if the faults are likely to be transient.
- Retry the operation if it determines that the fault is likely to be transient and keep track of the number of times the operation was retried.
- Use an appropriate retry strategy, which specifies the number of retries, the delay between each attempt, and the actions to take after a failed attempt.

This transient fault handling can be achieved by wrapping all attempts to access a remote service in code that implements the retry pattern.

Retry pattern

If an app detects a failure when it tries to send a request to a remote service, it can handle the failure in any of the following ways:

- Retrying the operation. The app could retry the failing request immediately.
- Retrying the operation after a delay. The app should wait for a suitable amount of time before retrying the request.
- Cancelling the operation. The application should cancel the operation and report an exception.

The retry strategy should be tuned to match the business requirements of the app. For example, it's important to optimize the retry count and retry interval to the operation being attempted. If the operation is part of a user interaction, the retry interval should be short and only a few retries attempted to avoid making users wait for a response. If the operation is part of a long running workflow, where cancelling or restarting the workflow is expensive or time-consuming, it's appropriate to wait longer between attempts and to retry more times.

> **Note:** An aggressive retry strategy with minimal delay between attempts, and a large number of retries, could degrade a remote service that's running close to or at capacity. In addition, such a retry strategy could also affect the responsiveness of the app if it's continually trying to perform a failing operation.

If a request still fails after a number of retries, it's better for the app to prevent further requests going to the same resource and to report a failure. Then, after a set period, the app can make one or more requests to the resource to see if they're successful. For more information, see Circuit breaker pattern.

> **Tip:** Never implement an endless retry mechanism
>
> Use a finite number of retries, or implement the Circuit Breaker pattern to allow a service to recover.

The eShopOnContainers mobile app does not currently implement the retry pattern when making RESTful web requests. However, the `CachedImage` control, provided by the FFImageLoading library supports transient fault handling by retrying image loading. If image loading fails, further attempts will be made. The number of attempts is specified by the `RetryCount` property, and retries will occur after a delay specified by the `RetryDelay` property. If these property values aren't explicitly set, their default values are applied – 3 for the `RetryCount` property, and 250ms for the `RetryDelay` property. For more information about the `CachedImage` control, see Caching images.

The eShopOnContainers reference application does implement the retry pattern. For more information, including a discussion of how to combine the retry pattern with the `HttpClient` class, see .NET Microservices: Architecture for Containerized .NET Applications.

For more information about the retry pattern, see the Retry pattern on Microsoft Docs.

Circuit breaker pattern

In some situations, faults can occur due to anticipated events that take longer to fix. These faults can range from a partial loss of connectivity to the complete failure of a service. In these situations, it's pointless for an app to retry an operation that's unlikely to succeed, and instead should accept that the operation has failed and handle this failure accordingly.

The circuit breaker pattern can prevent an app from repeatedly trying to execute an operation that's likely to fail, while also enabling the app to detect whether the fault has been resolved.

> **Note:** The purpose of the circuit breaker pattern is different from the retry pattern. The retry pattern enables an app to retry an operation in the expectation that it'll succeed. The circuit breaker pattern prevents an app from performing an operation that's likely to fail.

A circuit breaker acts as a proxy for operations that might fail. The proxy should monitor the number of recent failures that have occurred, and use this information to decide whether to allow the operation to proceed, or to return an exception immediately.

The eShopOnContainers mobile app does not currently implement the circuit breaker pattern. However, the eShopOnContainers does. For more information, see .NET Microservices: Architecture for Containerized .NET Applications.

Tip: Combine the retry and circuit breaker patterns

An app can combine the retry and circuit breaker patterns by using the retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault is not transient.

For more information about the circuit breaker pattern, see the Circuit Breaker pattern on Microsoft Docs.

Summary

Many modern web-based solutions make use of web services, hosted by web servers, to provide functionality for remote client applications. The operations that a web service exposes constitute a web API, and client apps should be able to utilize the web API without knowing how the data or operations that the API exposes are implemented.

The performance of an app can be improved by caching frequently accessed data to fast storage that's located close to the app. Apps can implement read-through caching with the cache-aside pattern. This pattern determines whether the item is currently in the cache. If the item isn't in the cache, it's read from the data store and added to the cache.

When communicating with web APIs, apps must be sensitive to transient faults. Transient faults include the momentary loss of network connectivity to services, the temporary unavailability of a service, or timeouts that arise when a service is busy. These faults are often self-correcting, and if the action is repeated after a suitable delay, then it's likely to succeed. Therefore, apps should wrap all attempts to access a web API in code that implements a transient fault handling mechanism.

Unit testing

Mobile apps have unique problems that desktop and web-based applications don't have to worry about. Mobile users will differ by the devices that they use, by network connectivity, by the availability of services, and a range of other factors. Therefore, mobile apps should be tested as they will be used in the real world in order to improve their quality, reliability, and performance. There are many types of testing that should be performed on an app, including unit testing, integration testing, and user interface testing, with unit testing being the most common form of testing.

A unit test takes a small unit of the app, typically a method, isolates it from the remainder of the code, and verifies that it behaves as expected. Its goal is to check that each unit of functionality performs as expected, so that errors don't propagate throughout the app. Detecting a bug where it occurs is more efficient that observing the effect of a bug indirectly at a secondary point of failure.

Unit testing has the greatest effect on code quality when it's an integral part of the software development workflow. As soon as a method has been written, unit tests should be written that verify the behavior of the method in response to standard, boundary, and incorrect cases of input data, and that check any explicit or implicit assumptions made by the code. Alternatively, with test driven development, unit tests are written before the code. In this scenario, unit tests act as both design documentation and functional specifications.

> **Note:** Unit tests are very effective against regression – that is, functionality that used to work but has been disturbed by a faulty update.

Unit tests typically use the arrange-act-assert pattern:

- The *arrange* section of the unit test method initializes objects and sets the value of the data that is passed to the method under test.
- The *act* section invokes the method under test with the required arguments.
- The *assert* section verifies that the action of the method under test behaves as expected.

Following this pattern ensures that unit tests are readable and consistent.

Dependency injection and unit testing

One of the motivations for adopting a loosely-coupled architecture is that it facilitates unit testing. One of the types registered with Autofac is the `OrderService` class. The following code example shows an outline of this class:

```
public class OrderDetailViewModel : ViewModelBase
{
    private IOrderService _ordersService;

    public OrderDetailViewModel(IOrderService ordersService)
```

```
    {
        _ordersService = ordersService;
    }
    ...
}
```

The `OrderDetailViewModel` class has a dependency on the `IOrderService` type which the container resolves when it instantiates a `OrderDetailViewModel` object. However, rather than create an `OrderService` object to unit test the `OrderDetailViewModel` class, instead, replace the `OrderService` object with a mock for the purpose of the tests. Figure 10-1 illustrates this relationship.

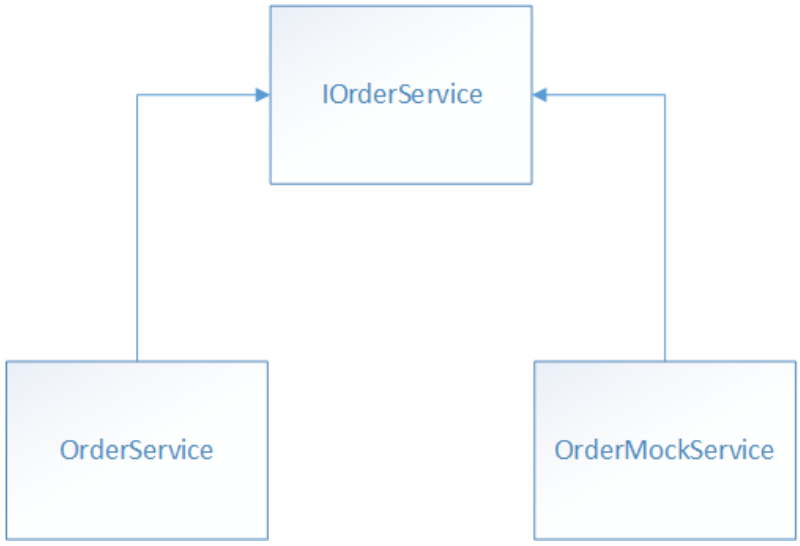

Figure 10-1: Classes that implement the IOrderService interface

This approach allows the `OrderService` object to be passed into the `OrderDetailViewModel` class at runtime, and in the interests of testability, it allows the `OrderMockService` class to be passed into the `OrderDetailViewModel` class at test time. The main advantage of this approach is that it enables unit tests to be executed without requiring unwieldy resources such as web services, or databases.

Testing MVVM applications

Testing models and view models from MVVM applications is identical to testing any other classes, and the same tools and techniques – such as unit testing and mocking, can be used. However, there are some patterns that are typical to model and view model classes, that can benefit from specific unit testing techniques.

> **Tip:** Test one thing with each unit test
>
> Don't be tempted to make a unit test exercise more than one aspect of the unit's behavior. Doing so leads to tests that are difficult to read and update. It can also lead to confusion when interpreting a failure.

The eShopOnContainers mobile app uses xUnit to perform unit testing, which supports two different types of unit tests:

- Facts are tests that are always true, which test invariant conditions.
- Theories are tests that are only true for a particular set of data.

The unit tests included with the eShopOnContainers mobile app are fact tests, and so each unit test method is decorated with the `[Fact]` attribute.

> **Note:** xUnit tests are executed by a test runner. To execute the test runner, run the eShopOnContainers.TestRunner project for the required platform.

Testing asynchronous functionality

When implementing the MVVM pattern, view models usually invoke operations on services, often asynchronously. Tests for code that invokes these operations typically use mocks as replacements for the actual services. The following code example demonstrates testing asynchronous functionality by passing a mock service into a view model:

```
[Fact]
public async Task OrderPropertyIsNotNullAfterViewModelInitializationTest()
{
    var orderService = new OrderMockService();
    var orderViewModel = new OrderDetailViewModel(orderService);

    var order = await orderService.GetOrderAsync(1, GlobalSetting.Instance.AuthToken);
    await orderViewModel.InitializeAsync(order);

    Assert.NotNull(orderViewModel.Order);
}
```

This unit test checks that the `Order` property of the `OrderDetailViewModel` instance will have a value after the `InitializeAsync` method has been invoked. The `InitializeAsync` method is invoked when the view model's corresponding view is navigated to. For more information about navigation, see Navigation.

When the `OrderDetailViewModel` instance is created, it expects an `OrderService` instance to be specified as an argument. However, the `OrderService` retrieves data from a web service. Therefore, an `OrderMockService` instance, which is a mock version of the `OrderService` class, is specified as the argument to the `OrderDetailViewModel` constructor. Then, when the view model's `InitializeAsync` method is invoked, which invokes `IOrderService` operations, mock data is retrieved rather than communicating with a web service.

Testing INotifyPropertyChanged implementations

Implementing the `INotifyPropertyChanged` interface allows views to react to changes that originate from view models and models. These changes are not limited to data shown in controls – they are also used to control the view, such as view model states that cause animations to be started or controls to be disabled.

Properties that can be updated directly by the unit test can be tested by attaching an event handler to the `PropertyChanged` event and checking whether the event is raised after setting a new value for the property. The following code example shows such a test:

```
[Fact]
public async Task SettingOrderPropertyShouldRaisePropertyChanged()
{
    bool invoked = false;
    var orderService = new OrderMockService();
    var orderViewModel = new OrderDetailViewModel(orderService);

    orderViewModel.PropertyChanged += (sender, e) =>
    {
```

```
        if (e.PropertyName.Equals("Order"))
            invoked = true;
    };
    var order = await orderService.GetOrderAsync(1, GlobalSetting.Instance.AuthToken);
    await orderViewModel.InitializeAsync(order);

    Assert.True(invoked);
}
```

This unit test invokes the `InitializeAsync` method of the `OrderViewModel` class, which causes its `Order` property to be updated. The unit test will pass, provided that the `PropertyChanged` event is raised for the `Order` property.

Testing message-based communication

View models that use the `MessagingCenter` class to communicate between loosely-coupled classes can be unit tested by subscribing to the message being sent by the code under test, as demonstrated in the following code example:

```
[Fact]
public void AddCatalogItemCommandSendsAddProductMessageTest()
{
    bool messageReceived = false;
    var catalogService = new CatalogMockService();
    var catalogViewModel = new CatalogViewModel(catalogService);

    Xamarin.Forms.MessagingCenter.Subscribe<CatalogViewModel, CatalogItem>(
        this, MessageKeys.AddProduct, (sender, arg) =>
    {
        messageReceived = true;
    });
    catalogViewModel.AddCatalogItemCommand.Execute(null);

    Assert.True(messageReceived);
}
```

This unit test checks that the `CatalogViewModel` publishes the `AddProduct` message in response to its `AddCatalogItemCommand` being executed. Because the `MessagingCenter` class supports multicast message subscriptions, the unit test can subscribe to the `AddProduct` message and execute a callback delegate in response to receiving it. This callback delegate, specified as a lambda expression, sets a `boolean` field that's used by the `Assert` statement to verify the behavior of the test.

Testing exception handling

Unit tests can also be written that check that specific exceptions are thrown for invalid actions or inputs, as demonstrated in the following code example:

```
[Fact]
public void InvalidEventNameShouldThrowArgumentExceptionText()
{
    var behavior = new MockEventToCommandBehavior
    {
        EventName = "OnItemTapped"
    };
    var listView = new ListView();

    Assert.Throws<ArgumentException>(() => listView.Behaviors.Add(behavior));
}
```

This unit test will throw an exception, because the `ListView` control does not have an event named `OnItemTapped`. The `Assert.Throws<T>` method is a generic method where `T` is the type of the expected exception. The argument passed to the `Assert.Throws<T>` method is a lambda expression that will throw the exception. Therefore, the unit test will pass provided that the lambda expression throws an `ArgumentException`.

> **Tip**: Avoid writing unit tests that examine exception message strings
>
> Exception message strings might change over time, and so unit tests that rely on their presence are regarded as brittle.

Testing validation

There are two aspects to testing the validation implementation: testing that any validation rules are correctly implemented, and testing that the `ValidatableObject<T>` class performs as expected.

Validation logic is usually simple to test, because it is typically a self-contained process where the output depends on the input. There should be tests on the results of invoking the `Validate` method on each property that has at least one associated validation rule, as demonstrated in the following code example:

```
[Fact]
public void CheckValidationPassesWhenBothPropertiesHaveDataTest()
{
    var mockViewModel = new MockViewModel();
    mockViewModel.Forename.Value = "John";
    mockViewModel.Surname.Value = "Smith";

    bool isValid = mockViewModel.Validate();

    Assert.True(isValid);
}
```

This unit test checks that validation succeeds when the two `ValidatableObject<T>` properties in the `MockViewModel` instance both have data.

As well as checking that validation succeeds, validation unit tests should also check the values of the `Value`, `IsValid`, and `Errors` property of each `ValidatableObject<T>` instance, to verify that the class performs as expected. The following code example demonstrates a unit test that does this:

```
[Fact]
public void CheckValidationFailsWhenOnlyForenameHasDataTest()
{
    var mockViewModel = new MockViewModel();
    mockViewModel.Forename.Value = "John";

    bool isValid = mockViewModel.Validate();

    Assert.False(isValid);
    Assert.NotNull(mockViewModel.Forename.Value);
    Assert.Null(mockViewModel.Surname.Value);
    Assert.True(mockViewModel.Forename.IsValid);
    Assert.False(mockViewModel.Surname.IsValid);
    Assert.Empty(mockViewModel.Forename.Errors);
    Assert.NotEmpty(mockViewModel.Surname.Errors);
}
```

This unit test checks that validation fails when the `Surname` property of the `MockViewModel` doesn't have any data, and the `Value`, `IsValid`, and Errors property of each `ValidatableObject<T>` instance are correctly set.

Summary

A unit test takes a small unit of the app, typically a method, isolates it from the remainder of the code, and verifies that it behaves as expected. Its goal is to check that each unit of functionality performs as expected, so that errors don't propagate throughout the app.

The behavior of an object under test can be isolated by replacing dependent objects with mock objects that simulate the behavior of the dependent objects. This enables unit tests to be executed without requiring unwieldy resources such as web services, or databases.

Testing models and view models from MVVM applications is identical to testing any other classes, and the same tools and techniques can be used.

Visit our .NET Architecture Center
www.microsoft.com/net/architecture