Microsoft

Enterprise Application Patterns using .NET MAUI

Michael Stonis

DOWNLOAD available at: https://aka.ms/maui-ebook

EDITION v2.0

PUBLISHED BY

Microsoft Developer Division, .NET, and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Authors:

Michael Stonis, Mobile Software Architect, Eight-Bot

Reviewers:

James Montemagno, Principal Lead Program Manager, Microsoft Corp.

David Pine, Developer Relations, Microsoft Corp.

Acknowledgments

This book originated from the excellent Enterprise Application Patterns using Xamarin.Forms eBook by David Britch and Javier Suarez Ruiz. Without their hard work, detailed information, and excellent examples, this book would not be possible.

Introduction

Enterprise applications face a number of difficult problems to solve including ever changing business requirements, the need for quick turn around time, support for multiple platforms, and integration with multiple systems. Due to the varying nature of these problems, it's important that our application's architecture allows it to be modular, modifiable and extensible over time.

This book takes provides real world solutions for addressing these issues when building an enterprise application using .NET MAUI. This book uses a pre-built .NET MAUI application that serves as the front-end of an online eCommerce application as a reference and a guide for common enterprise design patterns. This book covers topics such as the MVVM pattern, dependency injection, navigation, configuration, the loose-coupling of components and additional enterprise concerns. The content of this book is helpful for anyone looking to build a new application for this business or looking to solve the problems of applications that evolve over time.

Who should use the book

This book is for .NET MAUI developers that are already familiar with the framework, but that are looking for guidance on architecture and implementation when building enterprise applications. This book can help developers solve common problems using tried and true patterns.

How to use the book

This book focuses on building cross-platform enterprise apps using .NET MAUI. As such, it should be read in its entirety to provide a foundation of understanding such apps and their technical considerations. The book, 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. You can find the sample app on GitHub.

What this book doesn't cover

This book is aimed at readers who are already familiar with .NET MAUI. It does cover some concepts of .NET MAUI to help better illustrate the topic, but it does not cover most controls and concepts in any detail. For general guidance on building a new .NET MAUI app, please refer to the Building your first app guide in the .NET MAUI documentation.

Additional resources

For official .NET MAUI content, see .NET MAUI docs. .NET MAUI is developed as an open-source project and is available on GitHub at dotnet/maui. For code samples developed with .NET MAUI, see the dotnet/maui-samples repo.

Contents

Purpose

This eBook provides guidance on building cross-platform enterprise apps using .NET MAUI. .NET MAUI is a cross-platform UI toolkit that allows developers to easily create native user interface layouts that can be shared across platforms, including iOS, macOS, Android, and Windows. 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 .NET MAUI 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 eShop multi-platform app, and source code for the eShop reference app. The eShop multi-platform app is a cross-platform enterprise app developed using .NET MAUI, which connects to a series of containerized microservices known as the eShop reference app. However, the eShop multi-platform 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 .NET MAUI. For a detailed introduction to .NET MAUI, see the .NET MAUI documentation.

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 .NET MAUI.

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 .NET MAUI.

How to use this guide

This guide focuses on building cross-platform enterprise apps using .NET MAUI. As such, it should be read in its entirety to provide a foundation of understanding such apps and their technical considerations. The guide and 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 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 .NET MAUI. Having everybody working from a common set of terminologies and underlying principles will help ensure a consistent application of architectural patterns and practices.

Introduction to .NET MAUI

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 .NET MAUI 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 .NET MAUI. The guidance aims to help to produce adaptable, maintainable, and testable code, by addressing common .NET MAUI enterprise app development scenarios, and by separating the concerns of presentation, presentation logic, and entities through support for the Model-View-ViewModel (MVVM) pattern.

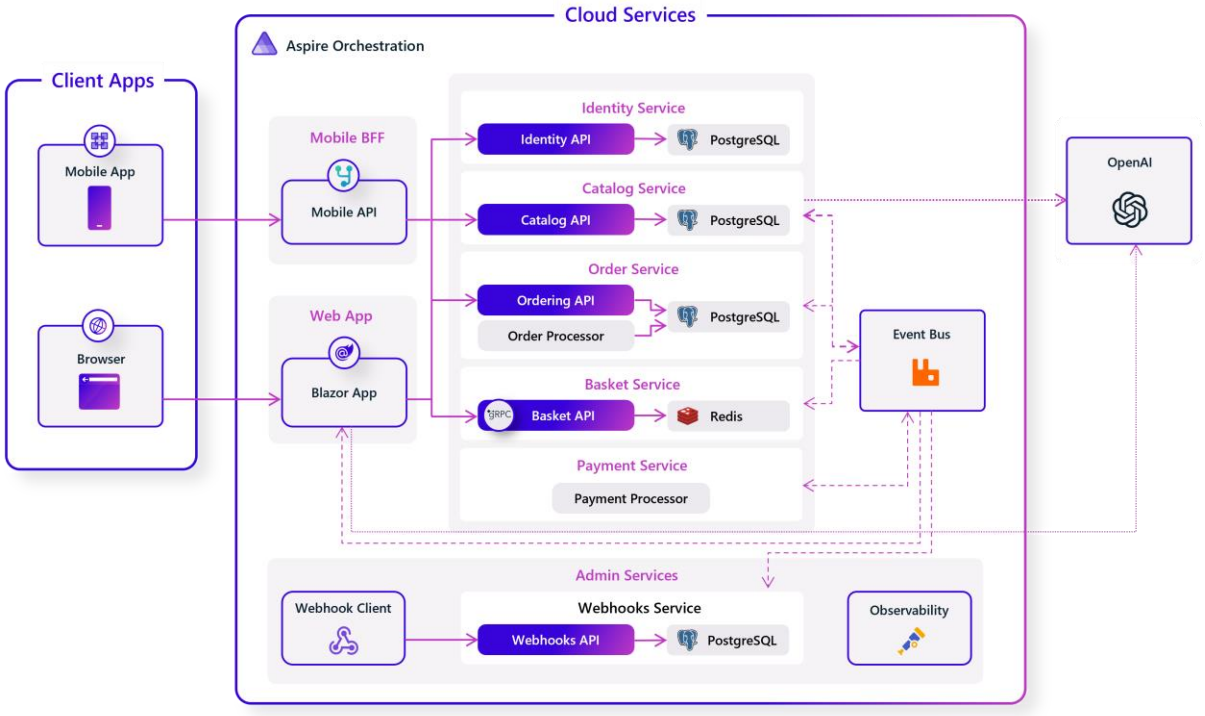
Sample application

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

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

Sample application architecture

Below is a high-level overview of the architecture of the sample application.



The sample application ships with:

- .NET Aspire App Hosting & Orchestration
- An Blazor web application developed with ASP.NET Core.
- A multi-platform app developed with .NET MAUI, which supports iOS, Android, macOS via Mac Catalyst, and Windows.

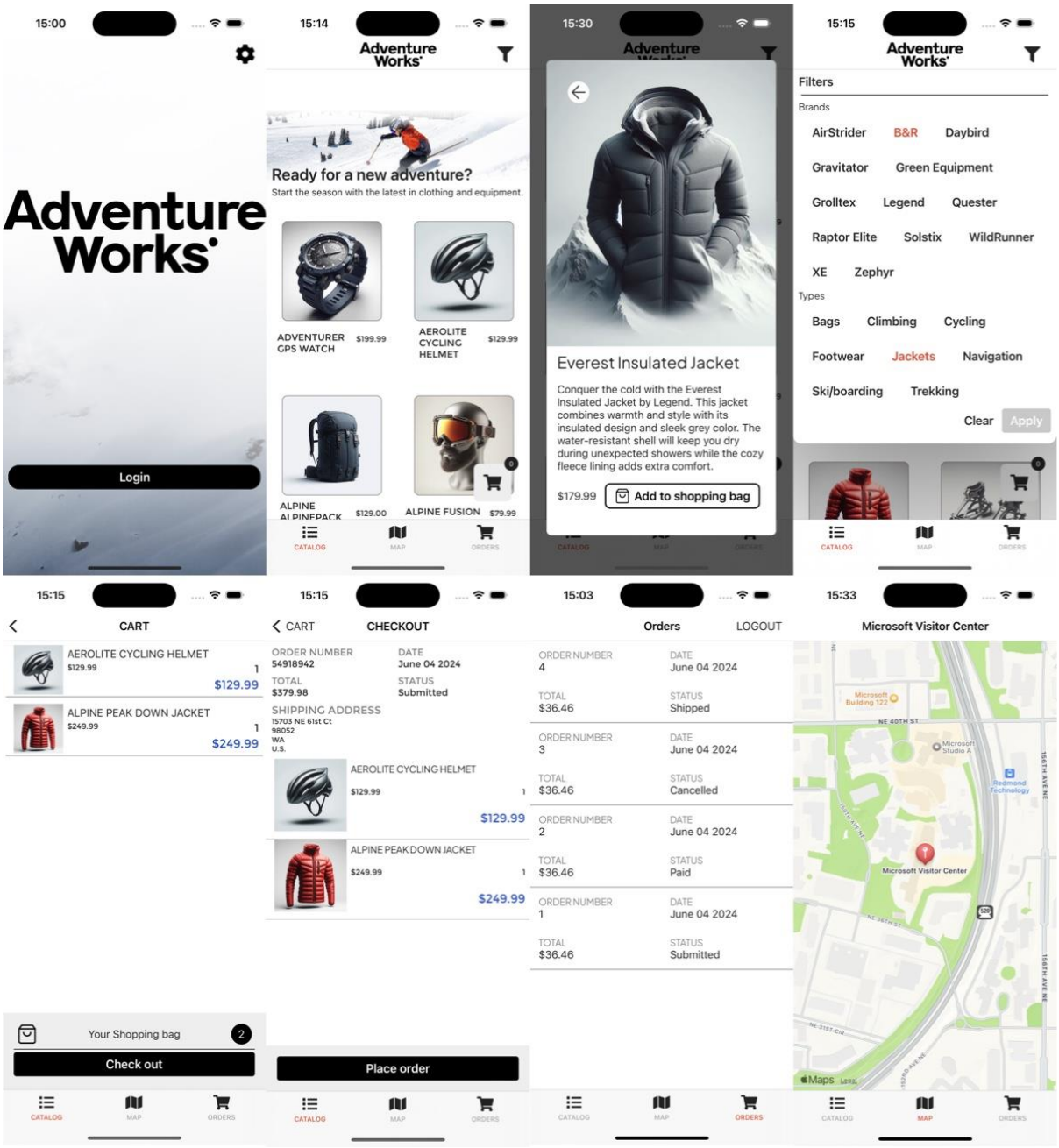
The sample application includes the following backend services:

- 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, and are deployed as unique containers with .NET Aspire. Collectively, these backend services are referred to as the eShop reference application. Client apps communicate with the backend services through a Representational State Transfer (REST) web interface. For more information about microservices and conainers, see Containerized microservices.

Multi-Platform app

This guide focuses on building cross-platform enterprise apps using .NET MAUI, and uses the eShop multi-platform app as an example. The image below shows the pages from the eShop multi-platform app that provide the functionality outlined earlier.



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

The eShop multi-platform app exercises the following .NET MAUI functionality:

- XAML
- Controls
- Bindings
- Converters
- Styles
- Animations
- Commands
- Behaviors
- Triggers
- Effects
- Custom Controls

For more information about this functionality, see the .NET MAUI documentation.

In addition, unit tests are provided for some of the classes in the eShop multi-platform app.

Multi-Platform app solution

The eShop multi-platform app solution organizes the source code and other resources into a multiple projects. All of the core mobile components are contained in a singular project named eShopContainers. This is a feature introduced with .NET 6 that allows a project to target multiple outputs which helps eliminate the need for multiple platform projects that we would have used in Xamarin.Forms and earlier .NET versions. An additional project is included for unit testing.

While this project has all of its components stored in a singular project, it is worth considering separating it into multiple projects based on your needs. For example, if you have multiple implementations of service providers based off of a service with their own dependencies, it may make sense to break those service provider implementations out into their own separate project. Good candidates for project separation include shared models, service implementations, api client components, database or caching layers. Any place where you feel that the business could re-use a component in another project is a potential candidate for separation. These projects can then be packaged via NuGet for easy distribution and versioning.

All of the projects use folders to organize the source code and other resources into categories. The classes from the eShop multi-platform app can be re-used in any .NET MAUI app with little or no modification.

eShop project

The eShop 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.

Folder	Description
Controls	Contains custom controls used by the app.
Converters	Contains value converters that apply custom logic to a binding.
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.

Summary

Microsoft's cross-platform multi-platform 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, macOS, 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.

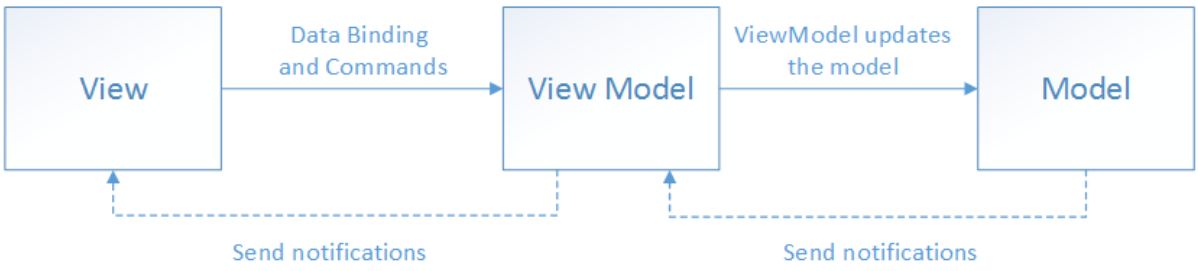
Model-View-ViewModel (MVVM)

The .NET MAUI developer experience typically involves creating a user interface in XAML, and then adding code-behind that operates on the user interface. Complex maintenance issues can arise as apps are modified and grow in size and scope. 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 MVVM pattern helps cleanly separate an application's business and presentation logic from its user interface (UI). Maintaining a clean separation between application logic and the UI helps address numerous development issues and makes an application easier to test, maintain, and evolve. It can also significantly improve code re-use opportunities and allows developers and UI designers to collaborate more easily 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. The diagram below shows the relationships between the three components.



In addition to understanding the responsibilities of each component, it's also important to understand how they interact. 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 an existing model implementation 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 prevents you from making 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 view model and model code, provided that the view is implemented entirely in XAML or C#. 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 development. 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 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 .NET MAUI application, a view is typically a ContentPage-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 the 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 the 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.

Multi-platform 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 to 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 .NET MAUI. 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 Microsoft Maui's 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 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>
    <!-- Omitted for brevity... -->
</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, resulting 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.

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.

A simple way to provide this functionality would be to create an extension of the BindableObject class. In this example, the ExtendedBindableObject class provides 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)
    {
        // Omitted for brevity ...
    }
}
```

.NET MAUI'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.

View model classes can then derive 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 eShop multi-platform app invokes property change notification by using a lambda expression:

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

```
    }
}
```

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 typically 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.

MVVM Frameworks

The MVVM pattern is well established in .NET, and the community has created many frameworks which help ease this development. Each framework provides a different set of features, but it is standard for them to provide a common view model with an implementation of the INotifyPropertyChanged interface. Additional features of MVVM frameworks include custom commands, navigation helpers, dependency injection/service locator components, and UI platform integration. While it is not necessary to use these frameworks, they can speed up and standardize your development. The eShop multi-platform app uses the .NET Community MVVM Toolkit. When choosing a framework, you should consider your application's needs and your team's strengths. The list below includes some of the more common MVVM frameworks for .NET MAUI.

- .NET Community MVVM Toolkit
- ReactiveUI
- Prism Library

UI interaction using commands and behaviors

In multi-platform 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. This way, your view models become more portable to new platforms, as they do not have a direct dependency on events provided by the platform's UI framework. .NET MAUI 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 public properties, for binding from the view, which implement the ICommand interface. Many .NET MAUI controls and gestures provide a Command property, which can be data bound to an ICommand object provided by the view model. The button control is one of the most commonly used controls, providing a command property that executes when the button is clicked.

> **Note**
>
> While it's possible to expose the actual implementation of the ICommand interface that your view model uses (for example, Command<T> or RelayCommand), it is recommended to expose your commands publicly as ICommand. This way, if you ever need to change the implementation at a later date, it can easily be swapped out.

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. In most cases, we will only supply the Execute method for our commands. For a more detailed overview of ICommand, refer to the Commanding documentation for .NET MAUI.

Provided with .NET MAUI are the Command and Command<T> classes that implement the ICommand interface, where T is the type of the arguments to Execute and CanExecute. Command and Command<T> are basic implementations that provide the minimal set of functionality needed for the ICommand interface.

> **Note**
>
> Many MVVM frameworks offer more feature rich implementations of the ICommand interface.

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 eShop multi-platform app uses the RelayCommand and AsyncRelayCommand. The primary benefit for modern applications is that the AsyncRelayCommand provides better functionality for asynchronous operations.

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 { get; }
```

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 an ICommand instance, which represents a sign-in command, is constructed by specifying a delegate to the SignInAsync view model method:

```
public ICommand SignInCommand { get; }
...
SignInCommand = new AsyncRelayCommand(async () => await SignInAsync());
```

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

```
public ICommand NavigateCommand { get; }

...
NavigateCommand = new AsyncRelayCommand<string>(NavigateAsync);
```

In both the RelayCommand and RelayCommand<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 UI controls 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 CommandParameter, 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 typically 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.

A .NET MAUI 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 eShop multi-platform 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 .NET MAUI 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.

The eShop multi-platform app includes an EventToCommandBehavior class which is provided by the MAUI Community toolkit. EventToCommandBehavior executes a command in response to an event occurring. This class derives from the BaseBehavior<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:

```
/// <summary>
/// The <see cref="EventToCommandBehavior"/> is a behavior that allows the user to invoke a
<see cref="ICommand"/> through an event. It is designed to associate Commands to events
exposed by controls that were not designed to support Commands. It allows you to map any
arbitrary event on a control to a Command.
/// </summary>
public class EventToCommandBehavior : BaseBehavior<VisualElement>
{
    // Omitted for brevity...

    /// <inheritdoc/>
    protected override void OnAttachedTo(VisualElement bindable)
    {
        base.OnAttachedTo(bindable);
        RegisterEvent();
    }

    /// <inheritdoc/>
    protected override void OnDetachingFrom(VisualElement bindable)
    {
        UnregisterEvent();
        base.OnDetachingFrom(bindable);
    }

    static void OnEventNamePropertyChanged(BindableObject bindable, object oldValue, object
newValue)
        => ((EventToCommandBehavior)bindable).RegisterEvent();

    void RegisterEvent()
    {
```

```
        UnregisterEvent();

        var eventName = EventName;
        if (View is null || string.IsNullOrWhiteSpace(eventName))
        {
            return;
        }

        eventInfo = View.GetType()?.GetRuntimeEvent(eventName) ??
            throw new ArgumentException($"{nameof(EventToCommandBehavior)}: Couldn't
resolve the event.", nameof(EventName));

        ArgumentNullException.ThrowIfNull(eventInfo.EventHandlerType);
        ArgumentNullException.ThrowIfNull(eventHandlerMethodInfo);

        eventHandler = eventHandlerMethodInfo.CreateDelegate(eventInfo.EventHandlerType,
this) ??
            throw new ArgumentException($"{nameof(EventToCommandBehavior)}: Couldn't create
event handler.", nameof(EventName));

        eventInfo.AddEventHandler(View, eventHandler);
    }

    void UnregisterEvent()
    {
        if (eventInfo is not null && eventHandler is not null)
        {
            eventInfo.RemoveEventHandler(View, eventHandler);
        }

        eventInfo = null;
        eventHandler = null;
    }

    /// <summary>
    /// Virtual method that executes when a Command is invoked
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="eventArgs"></param>
    [Microsoft.Maui.Controls.Internals.Preserve(Conditional = true)]
    protected virtual void OnTriggerHandled(object? sender = null, object? eventArgs =
null)
    {
        var parameter = CommandParameter
            ?? EventArgsConverter?.Convert(eventArgs, typeof(object), null, null);

        var command = Command;
        if (command?.CanExecute(parameter) ?? false)
        {
            command.Execute(parameter);
        }
    }
}
```

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 OnTriggerHandled 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 LoginView uses the EventToCommandBehavior to execute the ValidateCommand when the user changes the value of their password, as shown in the following code:

```
<Entry
    IsPassword="True"
    Text="{Binding Password.Value, Mode=TwoWay}">
    <!-- Omitted for brevity... -->
    <Entry.Behaviors>
        <mct:EventToCommandBehavior
            EventName="TextChanged"
            Command="{Binding ValidateCommand}" />
    </Entry.Behaviors>
    <!-- Omitted for brevity... -->
</Entry>
```

At runtime, the EventToCommandBehavior will respond to interaction with the Entry. When a user types into the Entry field, the TextChanged event will fire, which will execute the ValidateCommand in the LoginViewModel. By default, the event arguments for the event are passed to the command. If needed, the EventArgsConverter property can be used to convert the EventArgs provided by the event into a value that the command expects as input.

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

Summary

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

Using the MVVM pattern, the UI of the app and the underlying presentation and business logic are 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 known as *constructor injection*. The dependencies the object needs are injected into the constructor.

By specifying dependencies as interface types, dependency injection enables decoupling 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:

```
private readonly ISettingsService _settingsService;
private readonly IAppEnvironmentService _appEnvironmentService;

public ProfileViewModel(
    IAppEnvironmentService appEnvironmentService,
    IDialogService dialogService,
    INavigationService navigationService,
    ISettingsService settingsService)
    : base(dialogService, navigationService, settingsService)
{
    _appEnvironmentService = appEnvironmentService;
    _settingsService = settingsService;

    // Omitted for brevity
}
```

The ProfileViewModel constructor receives multiple interface object instances as arguments injected by another class. The only dependency in the ProfileViewModel class is on the interface types. Therefore, the ProfileViewModel class doesn't have any knowledge of the class that's responsible for instantiating the interface objects. The class that's responsible for instantiating the interface objects, 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 object 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.

There are several advantages to using a dependency injection container:

- A container removes the need for a class to locate its dependencies and manage its lifetimes.
- A container allows the 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 .NET MAUI app that uses MVVM, a dependency injection container will typically be used for registering and resolving views, registering and resolving view models, and for registering services and injecting them into view models.

There are many dependency injection containers available in .NET; the eShop multi-platform app uses Microsoft.Extensions.DependencyInjection to manage the instantiation of views, view models, and service classes in the app. Microsoft.Extensions.DependencyInjection 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 Microsoft.Extensions.DependencyInjection, see Dependency injection in .NET.

In .NET MAUI, the MauiProgram class will call into the CreateMauiApp method to create a MauiAppBuilder object. The MauiAppBuilder object has a Services property of type IServiceCollection, which provides a place to register our components, such as views, view models, and services for dependency injection. Any components registered with the Services property will be provided to the dependency injection container when the MauiAppBuilder.Build method is called.

At runtime, the container must know which implementation of the services are being requested in order to instantiate them for the requested objects. In the eShop multi-platform app, the IAppEnvironmentService, IDialogService , INavigationService, and ISettingsService interfaces need to be resolved before it can instantiate a ProfileViewModel object. This involves the container performing the following actions:

- Deciding how to instantiate an object that implements the interface. This is known as *registration*.
- Instantiating the object that implements the required 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 any short-lived interface implementations if other classes do not share the same instance.

Registration

Before dependencies can be injected into an object, the types of the dependencies must first be registered with the container. Registering a type 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. This is known as transient registration. 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.

> **Note**
>
> 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 requiring dependency injection should be performed in a single method in an app. This method should be invoked early in the app's lifecycle to ensure it is aware of the dependencies between its classes. The eShop multi-platform app performs this the MauiProgram.CreateMauiApp method. The following code example shows how the eShop multi-platform app declares the CreateMauiApp in the MauiProgram class:

```
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
        => MauiApp.CreateBuilder()
            .UseMauiApp<App>()
            // Omitted for brevity
            .RegisterAppServices()
            .RegisterViewModels()
            .RegisterViews()
            .Build();
}
```

The MauiApp.CreateBuilder method creates a MauiAppBuilder object that we can use to register our dependencies. Many dependencies in the eShop multi-platform app need to be registered, so the extension methods RegisterAppServices, RegisterViewModels, and RegisterViews were created to help provide an organized and maintainable registration workflow. The following code shows the RegisterViewModels method:

```
public static MauiAppBuilder RegisterViewModels(this MauiAppBuilder mauiAppBuilder)
{
    mauiAppBuilder.Services.AddSingleton<ViewModels.MainViewModel>();
    mauiAppBuilder.Services.AddSingleton<ViewModels.LoginViewModel>();
    mauiAppBuilder.Services.AddSingleton<ViewModels.BasketViewModel>();
    mauiAppBuilder.Services.AddSingleton<ViewModels.CatalogViewModel>();
    mauiAppBuilder.Services.AddSingleton<ViewModels.ProfileViewModel>();
```

```
    mauiAppBuilder.Services.AddTransient<ViewModels.CheckoutViewModel>();
    mauiAppBuilder.Services.AddTransient<ViewModels.OrderDetailViewModel>();
    mauiAppBuilder.Services.AddTransient<ViewModels.SettingsViewModel>();
    mauiAppBuilder.Services.AddTransient<ViewModels.CampaignViewModel>();
    mauiAppBuilder.Services.AddTransient<ViewModels.CampaignDetailsViewModel>();

    return mauiAppBuilder;
}
```

This method receives an instance of MauiAppBuilder, and we can use the Services property to register our view models. Depending on the needs of your application, you may need to add services with different lifetimes. The following table provides information on when you may want to choose these different registration lifetimes:

Method	Description
AddSingleton<T>	Will create a single instance of the object which will be remain for the lifetime of the application.
AddTransient<T>	Will create a new instance of the object when requested during resolution. Transient objects do not have a pre-defined lifetime, but will typically follow the lifetime of their host.

Note

The view models do not inherit from an interface, so they only need their concrete type provided to the AddSingleton<T> and AddTransient<T> methods.

The CatalogViewModel is used near the application's root and should always be available, so registering it with AddSingleton<T> is beneficial. Other view models, such as CheckoutViewModel and OrderDetailViewModel are situationally navigated to or are used later in the application. Suppose you know that you have a component that may not always be used. In that case, if it is memory or computationally intensive or requires just-in-time data, it may be a better candidate for AddTransient<T> registration.

Another common way to add services is using the AddSingleton<TService, TImplementation> and AddTransient<TService, TImplementation> methods. These methods take two input types: the interface definition and the concrete implementation. This type of registration is best for cases where you are implementing services based on interfaces. In the code example below, we register our ISettingsService interface using the SettingsService implementation:

```
public static MauiAppBuilder RegisterAppServices(this MauiAppBuilder mauiAppBuilder)
{
    mauiAppBuilder.Services.AddSingleton<ISettingsService, SettingsService>();
    // Omitted for brevity...
}
```

Once all services have been registered, the MauiAppBuilder.Build method should be called to create our MauiApp and populate our dependency injection container with all the registered services.

Important

Once the Build method has been called, the dependency injection container is immutable and can no longer be updated or modified. Ensure that all services that you need within your application have been registered before you call Build.

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 has been registered as transient, the container returns a new instance and doesn't maintain a reference to it.

.NET MAUI offers a number of ways to resolve registered components based on your needs. The most direct way to gain access to the dependency injection container is from an Element using the Handler.MauiContext.Services. An example of this is shown below:

```
var settingsService = this.Handler.MauiContext.Services.GetServices<ISettingsService>();
```

This can be helpful if you need to resolve a service from within an Element or from outside of the constructor of your Element.

Caution

There is a possibility that the Handler property of your Element may be null, so be aware that you may need to handle those situations. For more information, please refer to Handler lifecycle on the Microsoft Documentation Center.

If using the Shell control for .NET MAUI, it will implicitly call into the dependency injection container to create our objects during navigation. When setting up our Shell control, the Routing.RegisterRoute method will tie a route path to a View as shown in the example below:

```
Routing.RegisterRoute("Filter", typeof(FiltersView));
```

During Shell navigation, it will look for registrations of the FiltersView, and if any are found, it will create that view and inject any dependencies into the constructor. As shown in the code example below, the CatalogViewModel will be injected into the FiltersView:

```
namespace eShop.Views;

public partial class FiltersView : ContentPage
{
```

```
    public FiltersView(CatalogViewModel viewModel)
    {
        BindingContext = viewModel;

        InitializeComponent();
    }
}
```

Tip

The dependency injection container is great for creating view model instances. If a view model has dependencies, it will handle the creation and injection of any required services. Just make sure that you register your view models and any dependencies that they may have with the CreateMauiApp method in the MauiProgram class.

Summary

Dependency injection enables the 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.

Microsoft.Extensions.DependencyInjection 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.

Communicating between loosely coupled components

The publish-subscribe pattern is a messaging pattern in which publishers send messages without knowing any receivers, known as subscribers. Similarly, subscribers listen for specific messages, without knowing any publishers.

Events in .NET implement the publish-subscribe pattern and are the most simple 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 MVVM Toolkit Messenger

The MVVM Toolkit IMessenger interface describes 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 direct reference to each other, helping to reduce dependencies between components, while also allowing components to be independently developed and tested.

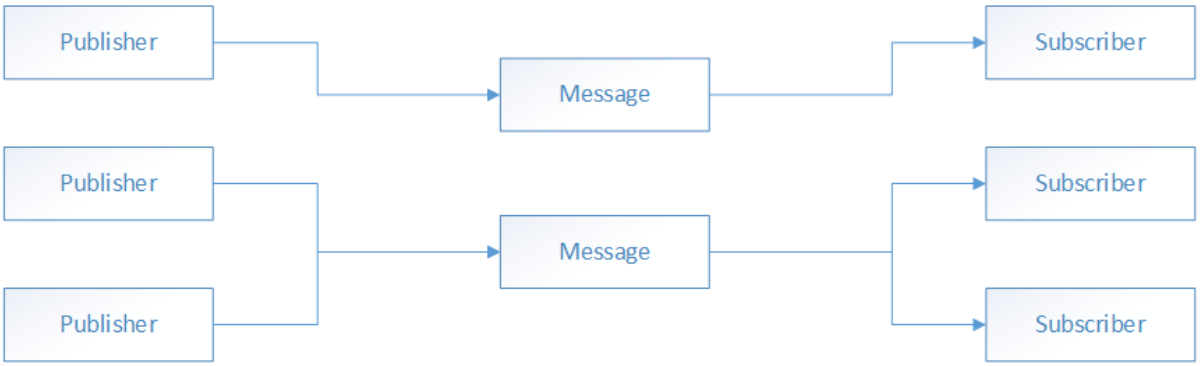
Note

The MVVM Toolkit Messenger is part of the CommunityToolkit.Mvvm package. For information on how to add the package to your project, see Introduction to the MVVM Toolkit on the Microsoft Developer Center.

Warning

.NET MAUI contains a built-in MessagingCenter class that's no longer recommended for use. Use the MVVM Toolkit Messenger instead.

The IMessenger interface allows for multicast publish-subscribe functionality. This means that there can be multiple publishers that publish a single message, and there can be multiple subscribers listening to the same message. The image below illustrates this relationship:



There are two implementations of the IMessenger interface that come with the CommunityToolkit.Mvvm package. The WeakReferenceMessenger uses weak references which can result in easier cleanup for message subscribers. This is a good option if your subscribers do not have a clearly defined lifecycle. The StrongReferenceMessenger uses strong references which can result in better performance and a more clearly controlled lifetime of the subscription. If you have a workflow with a very controlled lifetime (for example, a subscription that is bound to a page's OnAppearing and OnDisappearing methods), the StrongReferenceManager may be a better option, if performance is a concern. Both of these implementations are available with default implementations ready to use by referencing either WeakReferenceMessenger.Default or StrongReferenceMessenger.Default.

Note

While the IMessenger interface 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.

The eShop multi-platform app uses the WeakReferenceMessenger class to communicate between loosely coupled components. The app defines a single message named AddProductMessage. The AddProductMessage is published by the CatalogViewModel class when an item is added to the shopping basket. In return, the CatalogView class subscribes to the message and uses this to highlight the product adds with animation in response.

In the eShop multi-platform app, WeakReferenceMessenger is used to update the UI in response to an action occurring in another class. Therefore, messages are published from the thread that the class is executing on, with subscribers receiving the message on the same thread.

Tip

Marshal to the UI or main thread when performing UI updates. If updates to user interfaces are not made on this thread, it can cause the application to crash or become unstable.

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 MainThread.BeginInvokeOnMainThread method.

For more information about Messenger, see Messenger on the Microsoft Developer Center.

Defining a message

IMessenger messages are custom objects that provide custom payloads. The following code example shows the AddProductMessage message defined within the eShop multi-platform app:

```
public class AddProductMessage : ValueChangedMessage<int>
{
    public AddProductMessage(int count) : base(count)
    {
    }
}
```

The base class is defined using ValueChangedMessage<T> where T can be of any type needed to pass data. Both message publishers and subscribers can expect messages of a specific type (for example, AddProductMessage). This can help ensure that both parties have agreed to a messaging contract and that the data provided with that contract will be consistent. Additionally, this approach provides compile-time type safety and refactoring support.

Publishing a message

To publish a message, we will need to use the IMessenger.Send method. This can be accessed most commonly through WeakReferenceMessenger.Default.Send or StrongReferenceMessenger.Default.Send. The message sent can be of any object type. The following code example demonstrates publishing the AddProduct message:

```
WeakReferenceMessenger.Default.Send(new Messages.AddProductMessage(BadgeCount));
```

In this example, the Send method specifies provides a new instance of the AddProductMessage object for downstream subscribers to receive. An additional second token parameter can be added to use when multiple different subscribers need to receive messages of the same type without receiving the wrong message.

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.

Subscribing to a message

Subscribers can register to receive a message using one of the IMessenger.Register<T> overloads. The following code example demonstrates how the eShop multi-platform app subscribes to, and processes, the AddProductMessage message:

```
WeakReferenceMessenger.Default
    .Register<CatalogView, Messages.AddProductMessage>(
        this,
        async (recipient, message) =>
        {
            await recipient.Dispatcher.DispatchAsync(
                async () =>
                {
                    await recipient.badge.ScaleTo(1.2);
                    await recipient.badge.ScaleTo(1.0);
                });
        });
```

In the preceding example, the Register method subscribes to the AddProductMessage 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.

Note

Avoid the use of this within your callback delegate to avoid capturing that object within the delegate. This can help improve performance. Instead, use the recipient parameter.

If payload data is supplied, 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.

Unsubscribing from a message

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

```
WeakReferenceMessenger.Default.Unregister<Messages.AddProductMessage>(this);
```

Note

In this example, it isn't fully necessary to call Unregister as the WeakReferenceMessenger will allow unused objects to be garbage collected. If the StrongReferenceMessenger were used, it would be advised to call Unregister for any subscriptions that are no longer in use.

In this example, the Unsubscribe method syntax specifies the type argument of the message and the recipient object that is listening for messages.

Summary

The MVVM Toolkit IMessenger interface describes 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

.NET MAUI 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:

- Identifying the view to be navigated to using an approach that does not introduce tight coupling and dependencies between views.
- Coordinating 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.
- Determining how to cleanly separate the navigational behavior of the app across the views and view-models. The MVVM pattern separates the app's UI and its presentation and business logic, but it doesn't provide a direct mechanism for tying them together. However, the navigation behavior of an app will often span the UI and presentation 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.
- Determining 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.
- Coordinating navigation to ensure that specific 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 navigation service class named MauiNavigationService that's used to perform view-model-first page navigation.

Note

The MauiNavigationService used by the app is simplistic and does not cover all possible navigation types. The types of navigation needed by your application may require additional functionality.

Navigating between pages

Navigation logic can reside in a view's code-behind or a data-bound view-model. While placing navigation logic in a view might be the most straightforward approach, it is not easily testable through unit tests. Putting navigation logic in view-model classes means that the logic can be verified 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 navigation service 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 MauiNavigationService presented here specifies the view-model type as the target to navigate to.

The eShop multi-platform app uses the MauiNavigationService class to provide view-model-first navigation. This class implements the INavigationService interface, which is shown in the following code example:

```
public interface INavigationService
{
    Task InitializeAsync();

    Task NavigateToAsync(string route, IDictionary<string, object> routeParameters = null);

    Task PopAsync();
}
```

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(string route, IDictionary<string, object> routeParameters = null)	Performs hierarchical navigation to a specified page using a registered navigation route. Can optionally pass named route parameters to use for processing on the destination page
PopAsync	Removes the current page from the navigation stack.

Note

An INavigationService interface would usually also specify a GoBackAsync method, which is used to programmatically return to the previous page in the navigation stack. However, this method is missing from the eShop multi-platform app because it's not required.

Creating the MauiNavigationService instance

The MauiNavigationService class, which implements the INavigationService interface, is registered as a singleton with the dependency injection container in the MauiProgram.CreateMauiApp() method, as demonstrated in the following code example:

```
mauiAppBuilder.Services.AddSingleton<INavigationService, MauiNavigationService>();;
```

The INavigationService interface can then be resolved by adding it to the constructor of our views and view-models, as demonstrated in the following code example:

```
public AppShell(INavigationService navigationService)
```

This returns a reference to the MauiNavigationService object that's stored in the dependency injection container.

The ViewModelBase class stores the MauiNavigationService 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.

Handling navigation requests

.NET MAUI provides multiple ways to navigate within an application. The traditional way to navigate is with the NavigationPage class, which implements a hierarchical navigation experience in which the user can navigate through pages, forward and backward, as desired. The eShop app uses the Shell component as the root container for the application and as a navigation host. For more information about Shell navigation, see Shell Navigation on the Microsoft Developer Center.

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

```
await NavigationService.NavigateToAsync("//Main");
```

The following code example shows the NavigateToAsync method provided by the MauiNavigationService class:

```
public Task NavigateToAsync(string route, IDictionary<string, object> routeParameters =
null)
{
    return
        routeParameters != null
            ? Shell.Current.GoToAsync(route, routeParameters)
            : Shell.Current.GoToAsync(route);
}
```

The .NET MAUI Shell control is already familiar with route-based navigation, so the NavigateToAsync method works to mask this functionality. The NavigateToAsync method allows 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.

Important

There are multiple ways to perform navigation in .NET MAUI. The MauiNavigationService is specifically build to work with Shell. If you are using a NavigationPage or TabbedPage or a different navigation mechanism, this routing service would have to be updated to work using those components.

In order to register routes for the MauiNavigationService we need to supply route information from XAML or in the code-behind. The following example shows registration of routes via XAML.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:views="clr-namespace:eShop.Views"
    x:Class="eShop.AppShell">

    <!-- Omitted for brevity -->

    <FlyoutItem >
        <ShellContent x:Name="login" ContentTemplate="{DataTemplate views:LoginView}"
Route="Login" />
    </FlyoutItem>

    <TabBar x:Name="main" Route="Main">
        <ShellContent Title="CATALOG" Route="Catalog" Icon="{StaticResource
CatalogIconImageSource}" ContentTemplate="{DataTemplate views:CatalogView}" />
        <ShellContent Title="PROFILE" Route="Profile" Icon="{StaticResource
ProfileIconImageSource}" ContentTemplate="{DataTemplate views:ProfileView}" />
    </TabBar>
</Shell>
```

In this example, the ShellContent and TabBar user interface objects are setting their Route property. This is the preferred method of registering routes for user interface objects that are controlled by a Shell.

If we have objects that will be added to the navigation stack at a later time, then we will need to add those via code-behind. The following example show registration of routes in code-behind.

```
Routing.RegisterRoute("Filter", typeof(FiltersView));
Routing.RegisterRoute("Basket", typeof(BasketView));
```

In code-behind, we will call the Routing.RegisterRoute method which takes a route name as the first parameter and a view type as the second parameter. When a view-model uses the NavigationService property to navigate, the application's Shell object will look for registered routes and push them onto the navigation stack.

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

Navigating when the app is launched

When the app is launched, a Shell object is set as the root view of the application. Once set, the Shell will be used to control route registration and will be present at the root of our application going forward. Once the Shell has been created, we can wait for it to be attached to the application using the OnParentSet method to initialize our navigation route. The following code example shows this method:

```
protected override async void OnParentSet()
{
    base.OnParentSet();

    if (Parent is not null)
    {
        await _navigationService.InitializeAsync();
    }
}
```

The method uses an instance of INavigationService which is provided the constructor from dependency injection and invokes its InitializeAsync method.

The following code example shows the implementation of the MauiNavigationService.InitializeAsync method:

```
public Task InitializeAsync()
{
    return NavigateToAsync(string.IsNullOrEmpty(_settingsService.AuthAccessToken)
        ? "//Login"
        : "//Main/Catalog");
}
```

The //Main/Catalog route is navigated to if the app has a cached access token, which is used for authentication. Otherwise, the //Login route is navigated to.

Passing parameters during navigation

The NavigateToAsync method, specified by the INavigationService interface, enables navigation data to be specified as an IDictionary<string, object> of data 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)
{
    if (order is null)
    {
        return;
    }

    await NavigationService.NavigateToAsync(
        "OrderDetail",
```

```
        new Dictionary<string, object>{ { "OrderNumber", order.OrderNumber } });
}
```

This method invokes navigation to the OrderDetail route, passing order number information the order that the user selected. When the dependency injection framework creates the OrderDetailView for the OrderDetail route along with the OrderDetailViewModel class which is assigned to the view's BindingContext. The OrderDetailViewModel has an attribute added to it that allows it to receive data from the navigation service as shown in the code example below.

```
[QueryProperty(nameof(OrderNumber), "OrderNumber")]
public class OrderDetailViewModel : ViewModelBase
{
    public int OrderNumber { get; set; }
}
```

The QueryProperty attribute allows us to provide a parameter for a property to map values to and a key to find values from the query parameters dictionary. In this example, the key "OrderNumber" and order number value were provided during the NavigateToAsync call. The view-model found the "OrderNumber" key and mapped the value to the OrderNumber property. The OrderNumber property can then be used at a later time 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)
{
    // Omitted for brevity.
    if (!string.IsNullOrWhiteSpace(accessToken))
    {
```

```
        _settingsService.AuthAccessToken = accessToken;
        _settingsService.AuthIdToken = authResponse.IdentityToken;
        await NavigationService.NavigateToAsync("//Main/Catalog");
    }
}
```

This method invokes NavigationService route the application to the //Main/Catalog route.

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

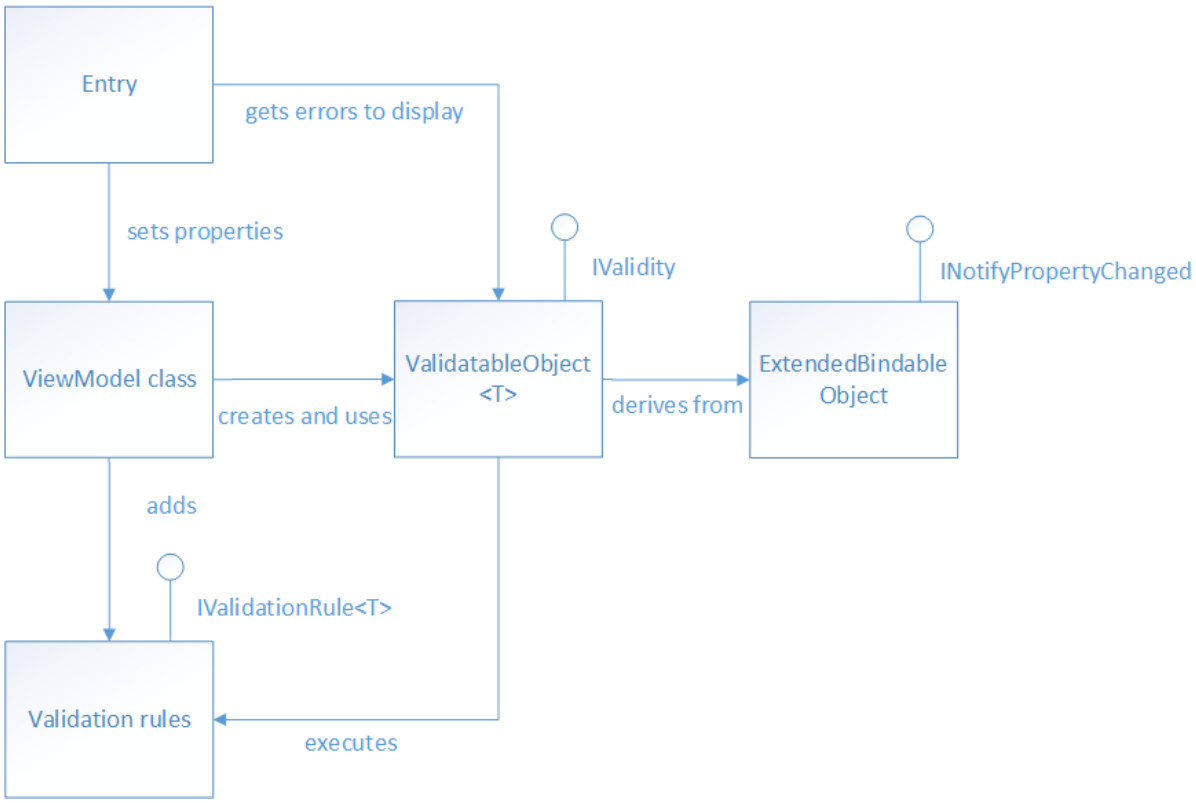
.NET MAUI includes support for page navigation, which typically results from the user's interaction with the UI, or from the app itself, as a result of internal logic-driven state changes. However, navigation can be complex to implement in apps that use the MVVM pattern.

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. Proper validation enforces business rules and could help to prevent an attacker from injecting malicious data.

In the context of the Model-View-ViewModel (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 eShop multi-platform 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. The image below shows the classes involved in performing validation in the eShop multi-platform 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 the validation succeeded or failed. The following code shows the implementation of the ValidatableObject<T>:

```
using CommunityToolkit.Mvvm.ComponentModel;
namespace eShop.Validations;
public class ValidatableObject<T> : ObservableObject, IValidity
{
    private IEnumerable<string> _errors;
    private bool _isValid;
    private T _value;
    public List<IValidationRule<T>> Validations { get; } = new();
    public IEnumerable<string> Errors
    {
        get => _errors;
        private set => SetProperty(ref _errors, value);
    }
    public bool IsValid
    {
        get => _isValid;
        private set => SetProperty(ref _isValid, value);
    }
    public T Value
    {
        get => _value;
        set => SetProperty(ref _value, value);
    }
    public ValidatableObject()
    {
        _isValid = true;
        _errors = Enumerable.Empty<string>();
    }
    public bool Validate()
    {
        Errors = Validations
            ?.Where(v => !v.Check(Value))
            ?.Select(v => v.ValidationMessage)
            ?.ToArray()
            ?? Enumerable.Empty<string>();
        IsValid = !Errors.Any();
        return IsValid;
    }
}
```

Property change notification is provided by the ObservableObject 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 eShop multi-platform app:

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

    public bool Check(T value) =>
        value is string str && !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 eShop multi-platform app, the following code example shows a validation rule for validating email addresses:

```
public class EmailRule<T> : IValidationRule<T>
{
    private readonly Regex _regex = new(@"^([w.-]+)@([w-]+)((.(w){2,3})+)$");

    public string ValidationMessage { get; set; }

    public bool Check(T value) =>
        value is string str && _regex.IsMatch(str);
}
```

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 against Regex.IsMatch.

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 eShop multi-platform 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; private set; }
public ValidatableObject<string> Password { get; private set; }
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 eShop multi-platform 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 eShop multi-platform 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 = _validations
        ?.Where(v => !v.Check(Value))
        ?.Select(v => v.ValidationMessage)
        ?.ToArray()
        ?? Enumerable.Empty<string>();

    IsValid = !Errors.Any();

    return IsValid;
}
```

This method 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:

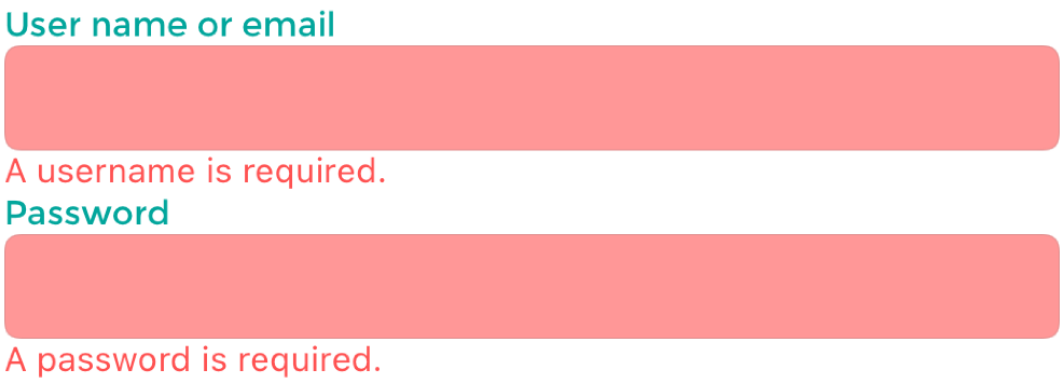
```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    <Entry.Behaviors>
        <behaviors:EventToCommandBehavior
            EventName="TextChanged"
            Command="{Binding ValidateUserNameCommand}" />
    </Entry.Behaviors>
</Entry>
```

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.

Displaying validation errors

The eShop multi-platform app notifies the user of any validation errors by highlighting the control that contains the invalid data with a red background, 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 background changes back to the default state and the error message is removed. The image below shows the LoginView in the eShop multi-platform app when validation errors are present.



Highlighting a control that contains invalid data

.NET MAUI offers a number of ways to present validation information to end-users, but one of the most straight-forward ways is through the use of Triggers. Triggers provide us a way to change the state of our controls, typically for appearance, based on an event or data change that occurs for a control. For validation, we will be using a DataTrigger which will listen to changes raised from a bound property and respond to the changes. The Entry controls on the LoginView are setup using the following code:

```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    <Entry.Style>
        <OnPlatform x:TypeArguments="Style">
            <On Platform="iOS, Android" Value="{StaticResource EntryStyle}" />
            <On Platform="WinUI" Value="{StaticResource WinUIEntryStyle}" />
        </OnPlatform>
    </Entry.Style>
    <Entry.Behaviors>
        <mct:EventToCommandBehavior
            EventName="TextChanged"
            Command="{Binding ValidateCommand}" />
    </Entry.Behaviors>
    <Entry.Triggers>
        <DataTrigger
            TargetType="Entry"
            Binding="{Binding UserName.IsValid}"
            Value="False">
            <Setter Property="BackgroundColor" Value="{StaticResource ErrorColor}" />
        </DataTrigger>
    </Entry.Triggers>
</Entry>
```

The DataTrigger specifies the following properties:

Property	Description
TargetType	The control type that the trigger belongs to.
Binding	The data Binding markup which will provide change notifications and value for the trigger condition.
Value	The data value to specify when the trigger's condition has been met.

For this Entry, we will be listening for changes to the LoginViewModel.UserName.IsValid property. Each time this property raises a change, the value will be compared against the Value property set in the DataTrigger. If the values are equal, then the trigger condition will be met and any Setter objects provided to the DataTrigger will be executed. This control has a single Setter object that updates the BackgroundColor property to a custom color defined using the StaticResource markup. When a Trigger condition is no longer met, the control will revert the properties set by the Setter object to their previous state. For more information about Triggers, see .NET MAUI Docs: Triggers.

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 IEnumerable<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 eShop multi-platform 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.

Application settings 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 core functionality and are only meaningful to that app.

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

Creating a Settings Interface

While the preferences manager can be used directly in your application, it does come with the drawback of making your application tightly coupled to the preferences manager implementation. This coupling means that creating unit tests or extending the functionality of preferences management will be limited since your application will not have a direct way to intercept the behavior. To address this concern, an interface can be created to work as a proxy for preferences management. The interface will allow us to supply an implementation that fits our needs. For example, when writing a unit test, we may want to set specific settings, and the interface will give us an easy way to consistently set this data for the test. The following code example shows the ISettingsService interface in the eShop multi-platform app:

```
namespace eShop.Services.Settings;

public interface ISettingsService
{
    string AuthAccessToken { get; set; }
    string AuthIdToken { get; set; }
    bool UseMocks { get; set; }
    string IdentityEndpointBase { get; set; }
    string GatewayShoppingEndpointBase { get; set; }
    string GatewayMarketingEndpointBase { get; set; }
    bool UseFakeLocation { get; set; }
    string Latitude { get; set; }
    string Longitude { get; set; }
```

```
    bool AllowGpsLocation { get; set; }
}
```

Adding Settings

.NET MAUI includes a preferences manager that provides a way to store runtime settings for a user. This feature can be accessed from anywhere within your application using the Microsoft.Maui.Storage.Preferences class. The preferences manager 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. For more information, see the Preferences on the Microsoft Developer Center.

Tip

Preferences is meant for storing relatively small data. If you need to store larger or more complex data, consider using a local database or filesystem to store the data.

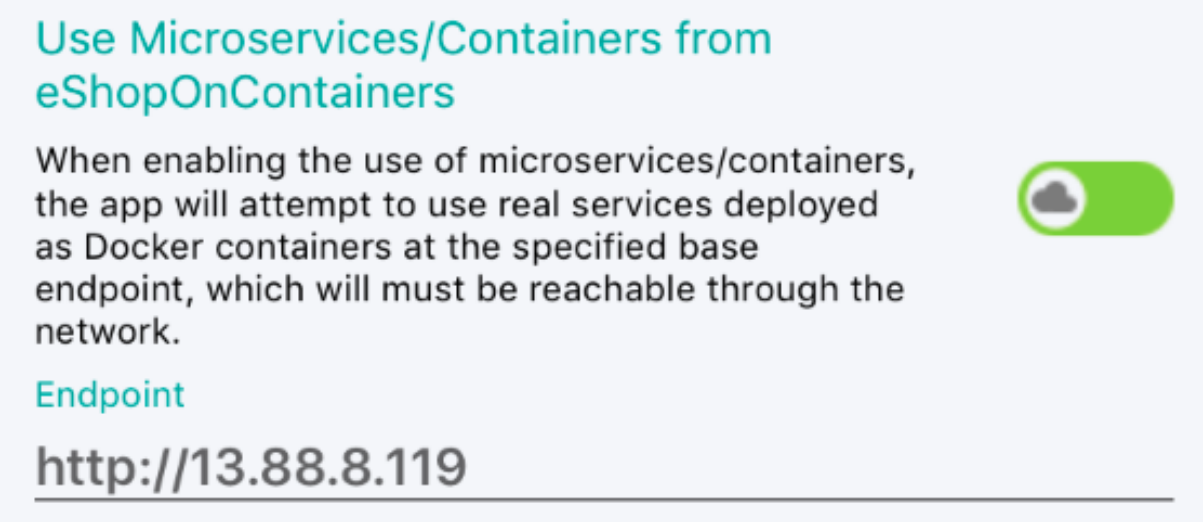
Our application will use the Preferences class need to implement the ISettingsService interface. The code below shows how the eShop multi-platform app's SettingsService implements the AuthTokenAccess and UseMocks properties:

```
public sealed class SettingsService : ISettingsService
{
    private const string AccessToken = "access_token";
    private const string AccessTokenDefault = string.Empty;

    private const string IdUseMocks = "use_mocks";
    private const bool UseMocksDefault = true;

    public string AuthAccessToken
    {
        get => Preferences.Get(AccessToken, AccessTokenDefault);
        set => Preferences.Set(AccessToken, value);
    }

    public bool UseMocks
    {
        get => Preferences.Get(IdUseMocks, UseMocksDefault);
        set => Preferences.Set(IdUseMocks, value);
    }
}
```

Each setting consists of a private key, a private default value, and a public property. The key is always a const string that defines a unique name, with the default value for the setting being a static read-only or constant value of the required type. Providing a default value ensures that a valid value is available if an unset setting is retrieved. This service implementation can be provided via dependency injection to our application for use in view-models or other services throughout the application.

Data binding to user settings

In the eShop multi-platform app, the SettingsView exposes multiple settings the user can configure at runtime. These settings include allowing configuration of whether the app should retrieve data from microservices deployed as Docker containers or whether the app should retrieve data from mock services that don't require an internet connection. When retrieving data from containerized microservices, a base endpoint URL for the microservices must be specified. The image below shows the SettingsView when the user has chosen to retrieve data from containerized microservices.



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

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

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

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

```
private readonly ISettingsService _settingsService;

private string _identityEndpoint;

public SettingsViewModel(
    ILocationService locationService, IAppEnvironmentService appEnvironmentService,
    IDialogService dialogService, INavigationService navigationService, ISettingsService
settingsService)
    : base(dialogService, navigationService, settingsService)
{
    _settingsService = settingsService;

    _identityEndpoint = _settingsService.IdentityEndpointBase;
}

public string IdentityEndpoint
```

```
{
    get => _identityEndpoint;
    set
    {
        SetProperty(ref _identityEndpoint, value);

        if (!string.IsNullOrWhiteSpace(value))
        {
            UpdateIdentityEndpoint();
        }
    }
}
```

When the IdentityEndpoint property is set, the UpdateIdentityEndpoint method is called, provided that the supplied value is valid. The following code example shows the UpdateIdentityEndpoint method:

```
private void UpdateIdentityEndpoint()
{
    _settingsService.IdentityEndpointBase = _identityEndpoint;
}
```

This method updates the IdentityEndpointBase property in the ISettingService interface implementation with the base endpoint URL value entered by the user. If the SettingsService class is provided as the implementation for _settingsService, the value will persist to platform-specific storage.

Summary

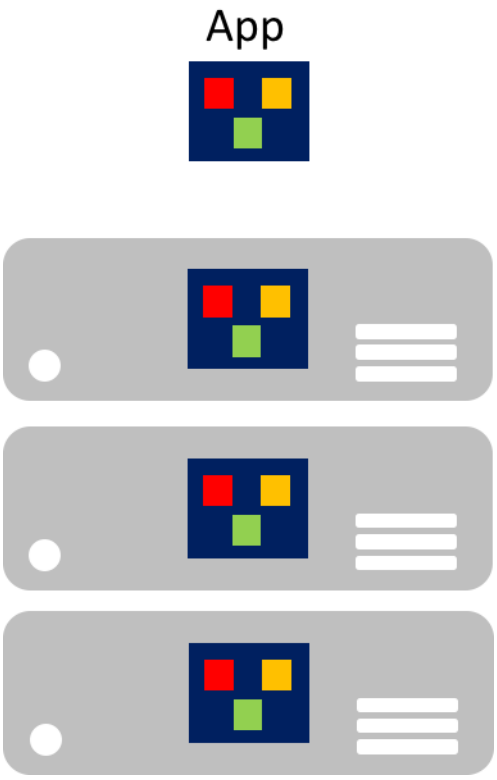
Settings allow the separation of data that configures the behavior of an app from the code, allowing the behavior to be changed without rebuilding the app. App settings are data that an app creates and manages, and user settings are the customizable settings of an app that affect the app's behavior and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

The Microsoft.Maui.Storage.Preferences class provides a consistent, type-safe, cross-platform approach for persisting and retrieving app and user settings.

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* 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, so a change to an application component requires its entire tier to be retested and redeployed.

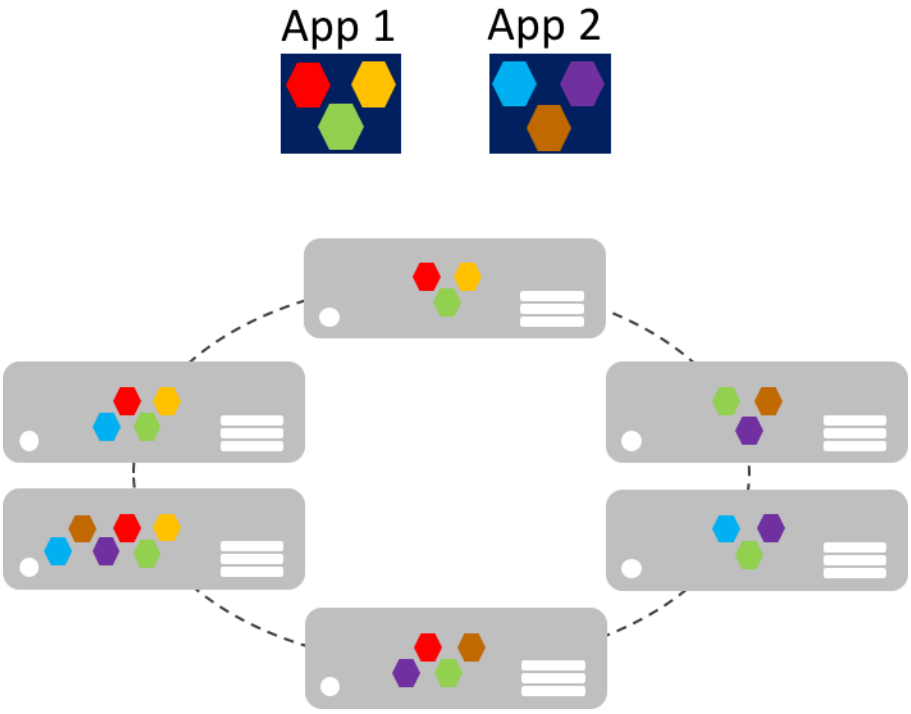
Particularly concerning, in the age of the cloud, is that individual components can't be easily scaled. A monolithic application contains domain-specific functionality and is typically divided by functional layers such as front-end, business logic, and data storage. The image below illustrates that a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



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 split 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 particular concerns, so each microservice implements a single function. In addition, each microservice has well-defined contracts with which other microservices communicate and share data. Typical examples of microservices include shopping carts, inventory processing, purchase subsystems, and payment processing.

Microservices can scale independently 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 application areas. The image below illustrates this approach, where microservices are deployed and scaled independently, creating instances of services across machines.



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 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 helps 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:

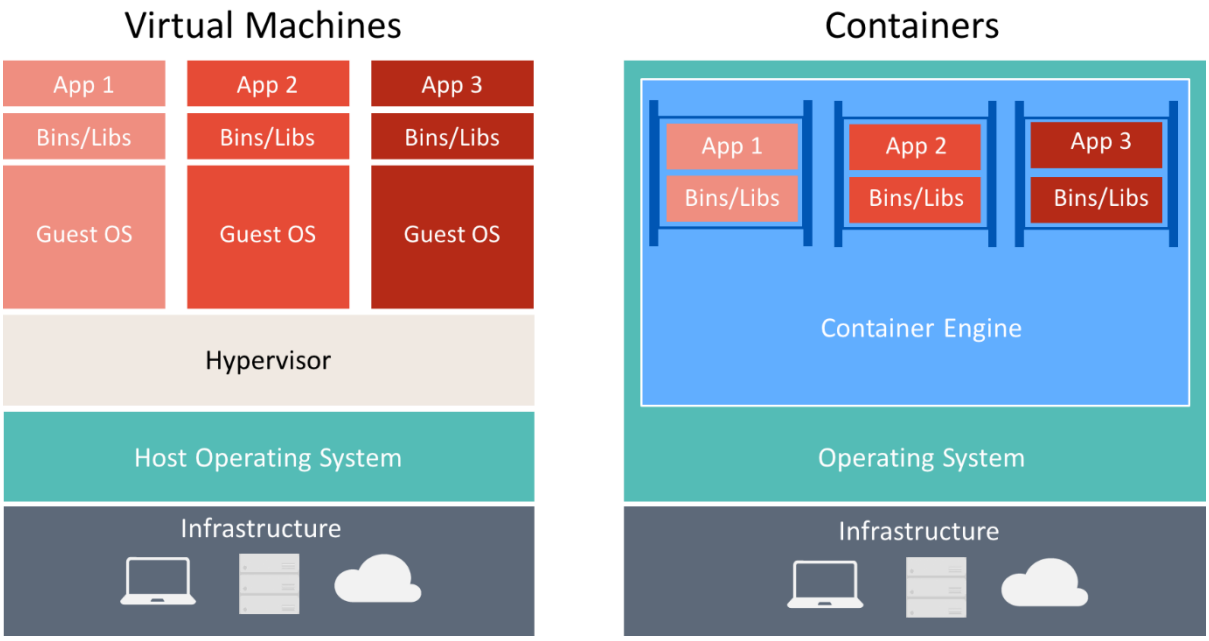
- Choosing how to partition an application into microservices can be challenging, as each microservice has to be completely autonomous, end-to-end, including responsibility for its data sources.
- Developers must implement inter-service communication, which adds complexity and latency to the application.
- 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 below.



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.

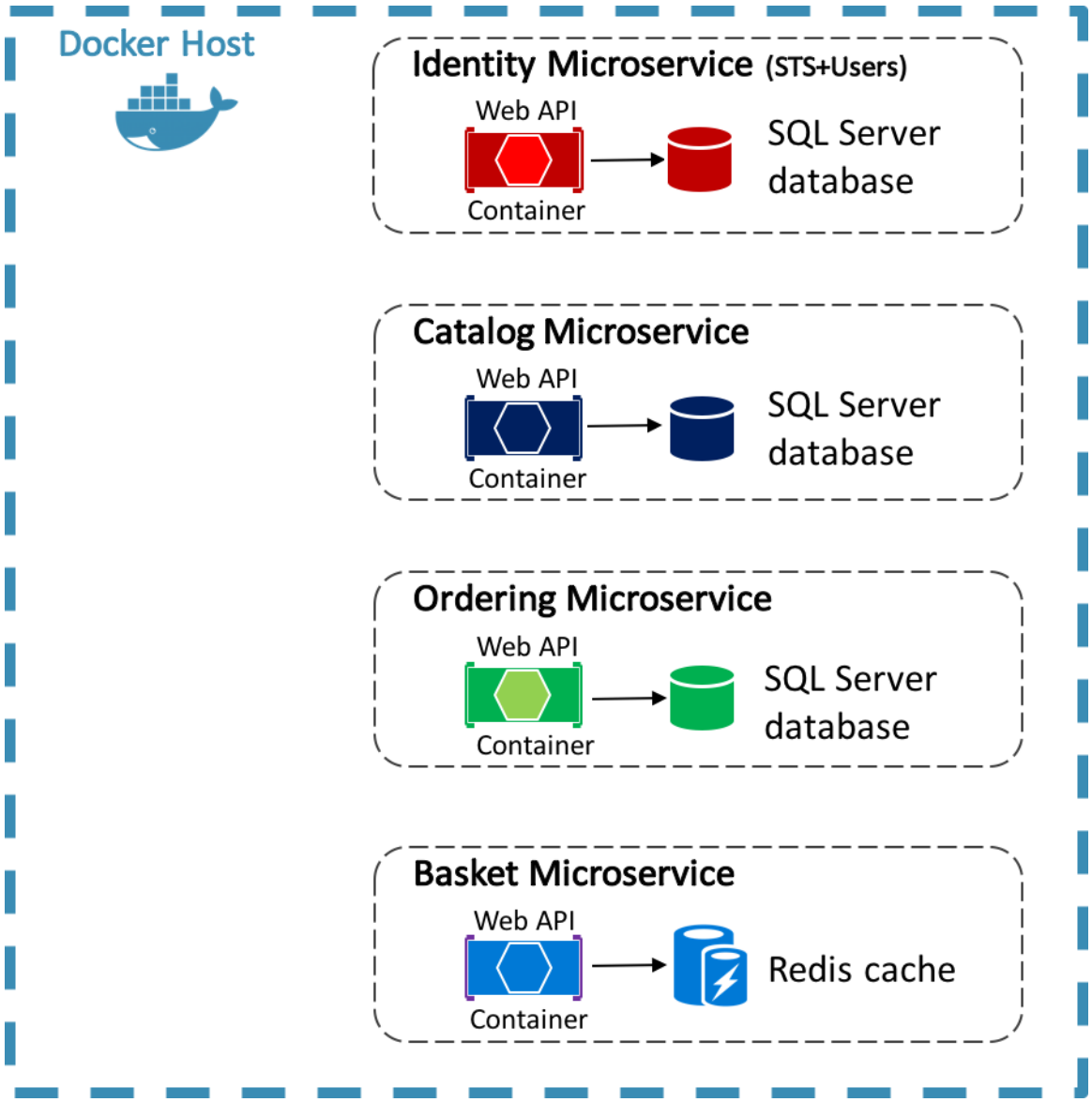
The key concepts when creating and working with containers are:

Concept	Description
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.

Concept	Description
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 eShop reference application uses Docker to host four containerized back-end microservices, as illustrated in the diagram below.

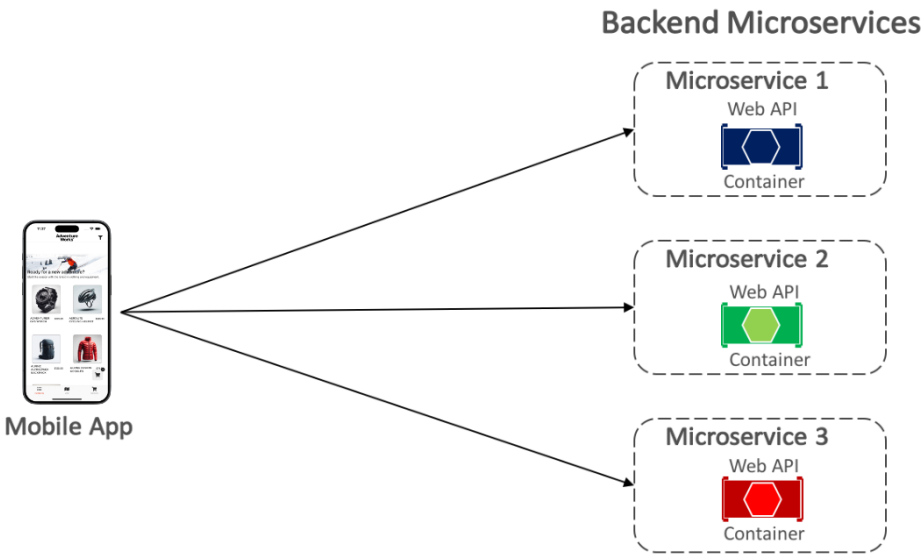


The architecture of the back-end services in the reference application is decomposed into multiple autonomous sub-systems in the form of collaborating microservices and containers. Each microservice provides a single area of functionality: an identity service, a catalog service, an ordering service,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

Communication between client and microservices

The eShop multi-platform app communicates with the containerized back-end microservices using *direct client-to-microservice* communication, as shown below.



With direct client-to-microservice communication, the multi-platform 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. Consider using API gateway communication when designing a large microservice-based application with tens of microservices.

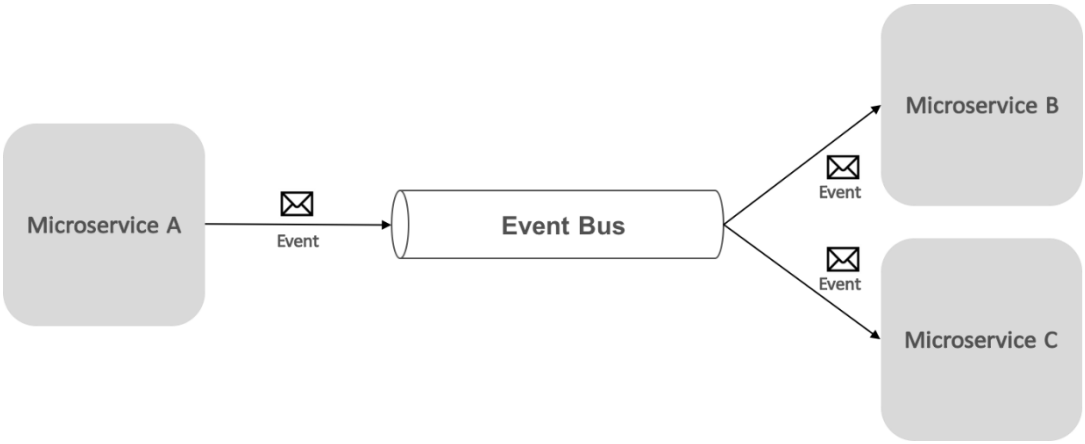
Communication between microservices

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.

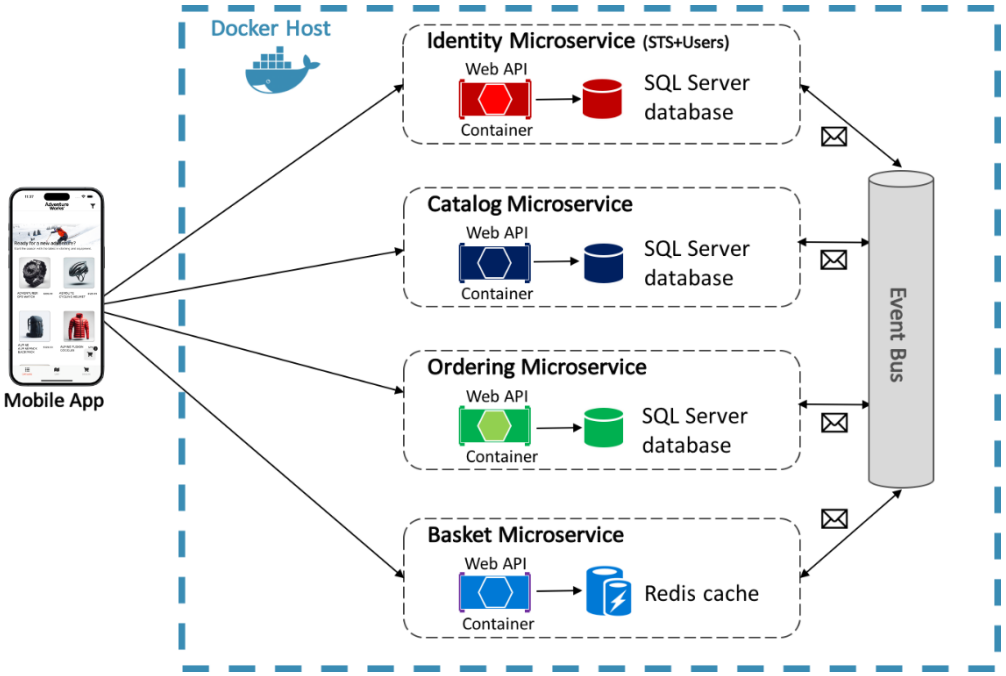
The two common approaches for microservice-to-microservice communication are HTTP-based REST communication when querying for data, and lightweight asynchronous messaging when communicating updates across multiple microservices.

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.

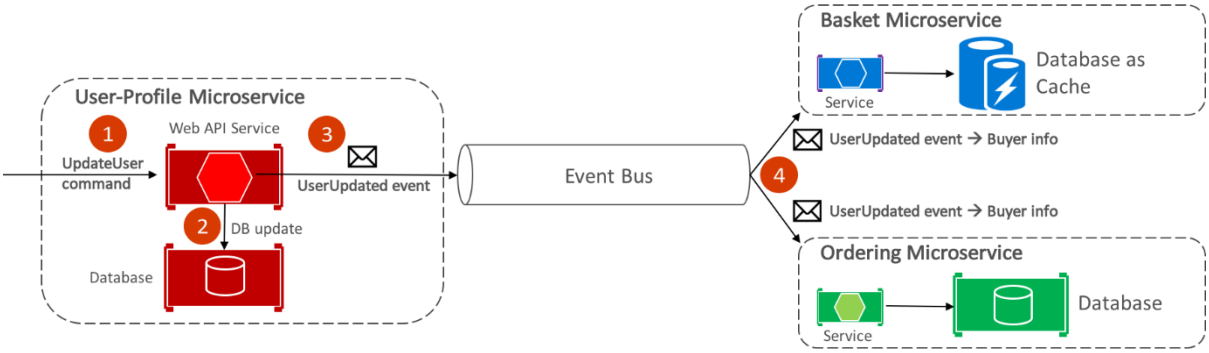
An event bus allows publish-subscribe communication between microservices without requiring the components to be explicitly aware of each other, as shown below.



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. The diagram below shows how an event bus is used in the eShop reference application.



The eShop 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. The diagram below illustrates this relationship.



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.

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 most software platforms and cloud vendors have adopted.

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, 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 eShop multi-platform 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 eShop 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 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.

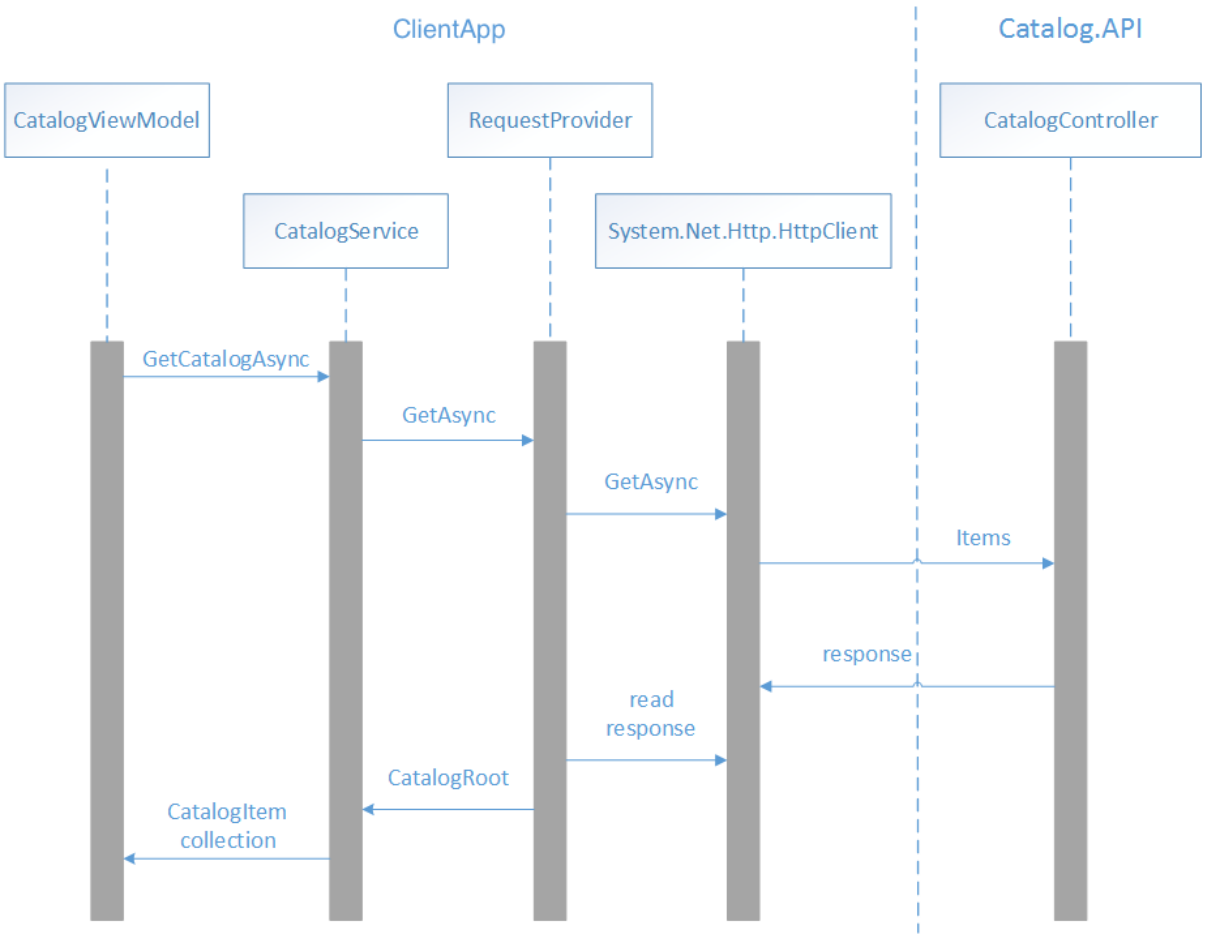
Making web requests

The eShop multi-platform 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 RegisterViewModels method in the MauiProgram class, the CatalogService class is registered as a type mapping against the ICatalogService type with the dependency injection container. Then, when an instance of the CatalogViewModel class is created, its constructor accepts an ICatalogService type, which the dependency injection container resolves, returning an instance of the CatalogService class. For more information about dependency injection, see Dependency Injection.

The image below shows the interaction of classes that read catalog data from the catalog microservice for displaying by the CatalogView.



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()
{
    Products = await _productsService.GetCatalogAsync();
}
```

This method calls the GetCatalogAsync method of the CatalogService instance that was injected into the CatalogViewModel by the dependency injection container. 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;
}
```

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 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 = GetOrCreateHttpClient(token);
    HttpResponseMessage response = await httpClient.GetAsync(uri);

    await HandleResponse(response);
    TResult result = await response.Content.ReadFromJsonAsync<TResult>();

    return result;
}
```

This method calls the GetOrCreateHttpClient 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 GetOrCreateHttpClient method is shown in the following code example:

```
private readonly Lazy<HttpClient> _httpClient =
    new Lazy<HttpClient>(
        () =>
        {
            var httpClient = new HttpClient();
            httpClient.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
            return httpClient;
        },
        LazyThreadSafetyMode.ExecutionAndPublication);

private HttpClient GetOrCreateHttpClient(string token = "")
    {
        var httpClient = _httpClient.Value;

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

```
        return httpClient;
    }
```

This method uses creates a new instance or retrieves a cached 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 GetOrCreateHttpClient 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.

Tip

It is highly recommended to cache and reuse instances of the HttpClient for better application performance. Creating a new HttpClient for each operation can lead to issue with socket exhaustion. For more information, see HttpClient Instancing on the Microsoft Developer Center.

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);
}
```

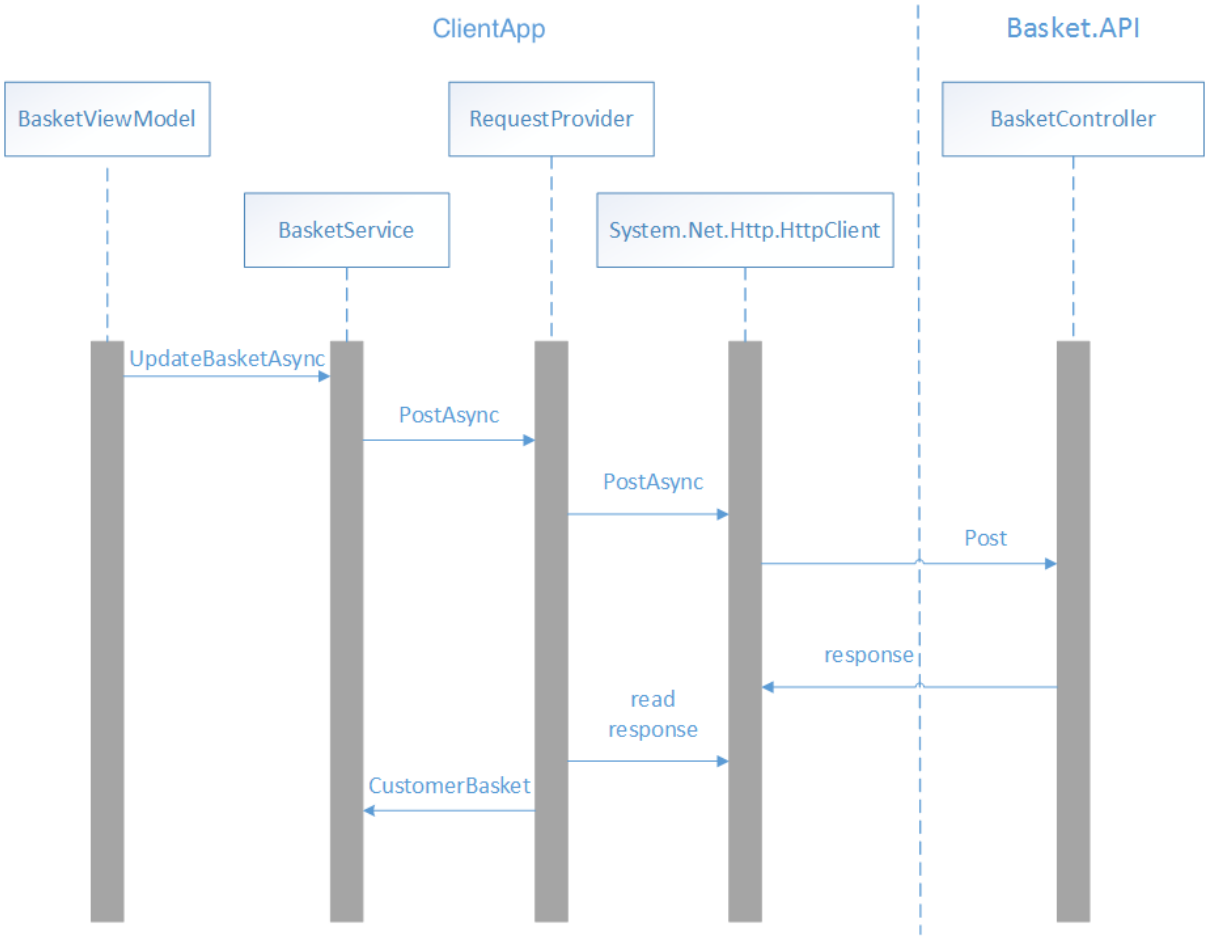
This method retrieves the catalog data from the SQL database using EntityFramework, and returns it as a response message that includes a success HTTP status code, and a collection of JSON formatted CatalogItem instances.

Making a POST request

The BasketService class is used to manage the data retrieval and update process with the basket microservice. In the RegisterAppServices method in the MauiProgram class, the BasketService class is registered as a type mapping against the IBasketService type with the dependency injection container. Then, when an instance of the BasketViewModel class is created, its constructor accepts an

IBasketService type, which the dependency injection container resolves, returning an instance of the BasketService class. For more information about dependency injection, see Dependency Injection.

The image below shows the interaction of classes that send the basket data displayed by the BasketView, 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()
{
    // Omitted for brevity...

    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 the dependency injection container. 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 = GetOrCreateHttpClient(token);

    var content = new StringContent(JsonSerializer.Serialize(data));
    content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
    HttpResponseMessage response = await httpClient.PostAsync(uri, content);

    await HandleResponse(response);
    TResult result = await response.Content.ReadFromJsonAsync<TResult>();

    return result;
}
```

This method calls the GetOrCreateHttpClient 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 GetOrCreateHttpClient method, see Making a GET request.

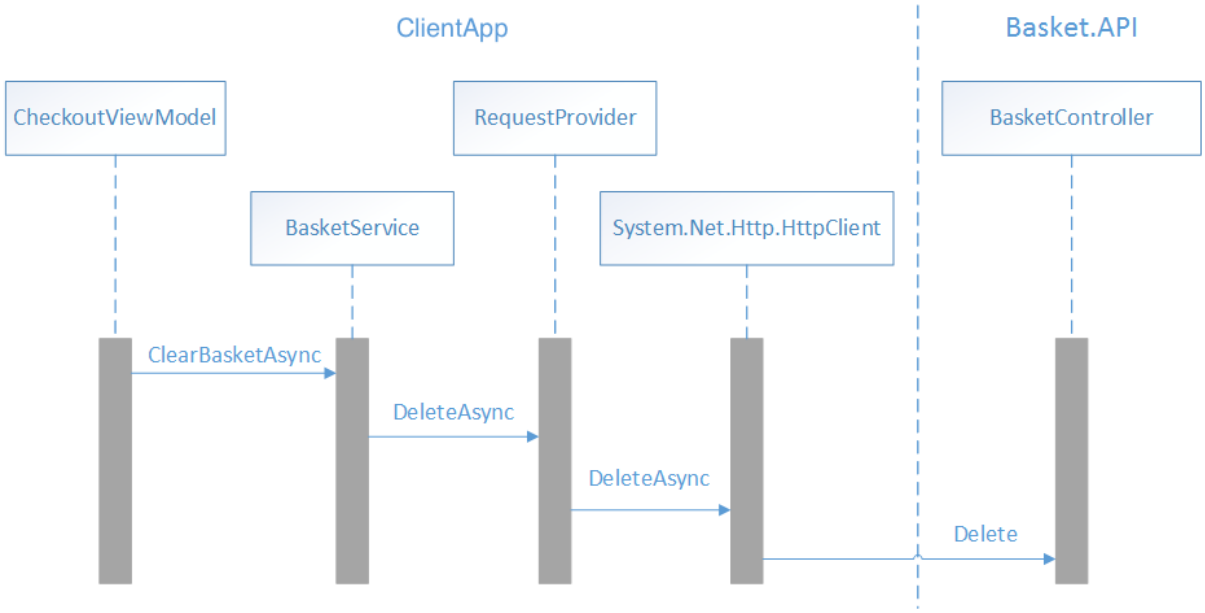
When the PostAsync method in the RequestProvider class calls HttpClient.PostAsync, the Post method in the BasketController class in the Basket.API project is invoked, which is shown in the following code example:

```
[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

The image below shows the interactions of classes that delete basket data from the basket microservice, for the CheckoutView.



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()
{
    // Omitted for brevity...

    await _basketService.ClearBasketAsync(
        _shippingAddress.Id.ToString(), authToken);
}
```

This method calls the ClearBasketAsync method of the BasketService instance that was injected into the CheckoutViewModel by the dependency injection container. The following method shows the ClearBasketAsync method:

```
public async Task ClearBasketAsync(string guidUser, string token)
{
    UriBuilder builder = new(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 = GetOrCreateHttpClient(token);
    await httpClient.DeleteAsync(uri);
}
```

This method calls the GetOrCreateHttpClient 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 GetOrCreateHttpClient 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 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 eShop 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 eShop multi-platform app uses a private cache, where data is held locally on the device that's running an instance of the app.

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 eShop multi-platform app consumes remote product images that benefit from being cached. These images are displayed by the Image control. The .NET MAUI Image control supports caching of

downloaded images which has caching 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 Microsoft Developer Center.

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. Instead, prefer an exponential backoff.

Use a finite number of retries, or implement the Circuit Breaker pattern to allow a service to recover.

The eShop reference application does implement the retry pattern.

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 eShop multi-platform app does not currently implement the circuit breaker pattern. However, the eShop does.

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.

CHAPTER 11

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. The entity that submitted the credentials is considered an authenticated identity if the credentials are valid. Once an identity has been established, an authorization process determines whether that identity has access to a given resource.

There are many approaches to integrating authentication and authorization into a .NET MAUI app that communicates with an ASP.NET web application, including using ASP.NET Core Identity, external authentication providers such as Microsoft, Google, Facebook, or Twitter, and authentication middleware. The eShop multi-platform app performs authentication and authorization with a containerized identity microservice that uses IdentityServer. The app requests security tokens from IdentityServer to authenticate a user or access 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 eShop reference application, ASP.NET Core Identity is used for this purpose.

Authentication

Authentication is required when an application needs to know the current user's identity. 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 use a local user datastore and persist identity information between requests via cookies (as is typical in ASP.NET 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 an 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 app.

Issuing bearer tokens using IdentityServer

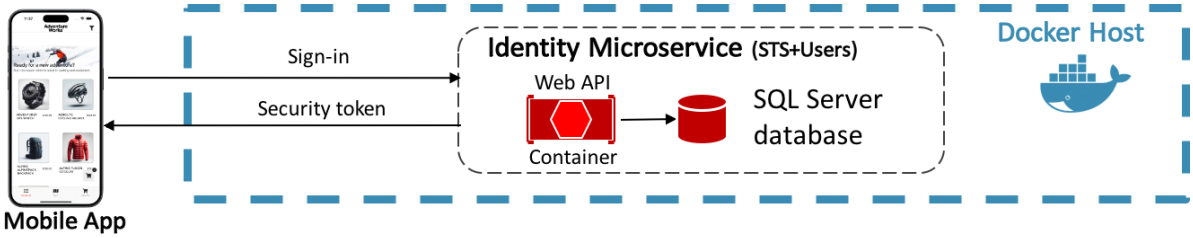
IdentityServer is an open-source OpenID Connect and OAuth 2.0 framework for ASP.NET Core, which can be used for many authentication and authorization scenarios, including issuing security tokens for local ASP.NET Core Identity users.

> **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.

OpenID Connect and OAuth 2.0 combine the two fundamental security concerns of authentication and API access, and IdentityServer is an implementation of these protocols.

In applications that use direct client-to-microservice communication, such as the eShop reference application, a dedicated authentication microservice acting as a Security Token Service (STS) can be used to authenticate users, as shown in the following diagram. For more information about direct client-to-microservice communication, see Microservices.



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

- Authenticating users with IdentityServer is achieved by the multi-platform app requesting an *identity* token, representing an authentication process's outcome. At a minimum, it contains an identifier for the user and information about how and when the user is authenticated. It can also include additional identity data.
- Accessing a resource with IdentityServer is achieved by the multi-platform 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 successfully request tokens. For more information on adding clients, see Defining Clients.

Adding IdentityServer to a web application

In order for an ASP.NET Core web application to use IdentityServer, 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 its HTTP request processing pipeline to serve requests to OpenID Connect and OAuth 2.0 endpoints. This is configured in the Identity.API project's Program.cs, as demonstrated in the following code example:

```
...

app.UseIdentityServer();
```

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 eShop 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 configuration. IdentityServer's APIs allow for configuring IdentityServer from an in-memory list of configuration objects. In the eShop 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 eShop reference application:

```
public static IEnumerable<ApiResource> GetApis()
{
    return new List<ApiResource>
    {
        new ApiScope("orders", "Orders Service"),
        new ApiScope("basket", "Basket Service"),
        new ApiScope("webhooks", "Webhooks registration 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 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 be included in the identity token for the user. The following code example shows the GetResources method that provides this collection in the eShop 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 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 eShop multi-platform app in the GetClients method that provides this collection in the eShop reference application:

```
public static IEnumerable<Client> GetClients(Dictionary<string,string> clientsUrl)
{
    return new List<Client>
    {
        // Omitted for brevity
        new Client
        {
            ClientId = "maui",
            ClientName = "eShop .NET MAUI OpenId Client",
            AllowedGrantTypes = GrantTypes.Hybrid,
            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },
            RedirectUris = { clientsUrl["maui"] },
            RequireConsent = false,
            RequirePkce = true,
            PostLogoutRedirectUris = { $"{clientsUrl["maui"]}/Account/Redirecting" },
            AllowedCorsOrigins = { "http://eshopmaui" },
            AllowedScopes = new List<string>
            {
                IdentityServerConstants.StandardScopes.OpenId,
                IdentityServerConstants.StandardScopes.Profile,
                IdentityServerConstants.StandardScopes.OfflineAccess,
                "orders",
                "basket"
            },
            AllowOfflineAccess = true,
            AllowAccessTokensViaBrowser = true,
            AccessTokenLifetime = 60 * 60 * 2, // 2 hours
            IdentityTokenLifetime = 60 * 60 * 2 // 2 hours
        }
    };
}
```

This configuration specifies data for the following properties:

Property	Description
ClientId	A unique ID for the client.
ClientName	The client display name, which is used for logging and the consent screen.

Property	Description
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.
AllowAccessTokensViaBrowser	Specifies whether the client can receive access tokens from a browser window.
AlwaysIncludeUserClaimsInIdToken	Specifies that the user claims will always be added to the id token. By default, these would have to be retrieved using the userinfo endpoint.
AccessTokenLifetime	Specifies the lifetime of the access token in seconds.
IdentityTokenLifetime	Specifies the lifetime of the identity token in seconds.

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 several authentication flows, including:

Authentication Flow	Description
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 and other artifacts such as the authorization code. After successfully validating the response, the back channel should be used to retrieve the access and refresh token.

Tip

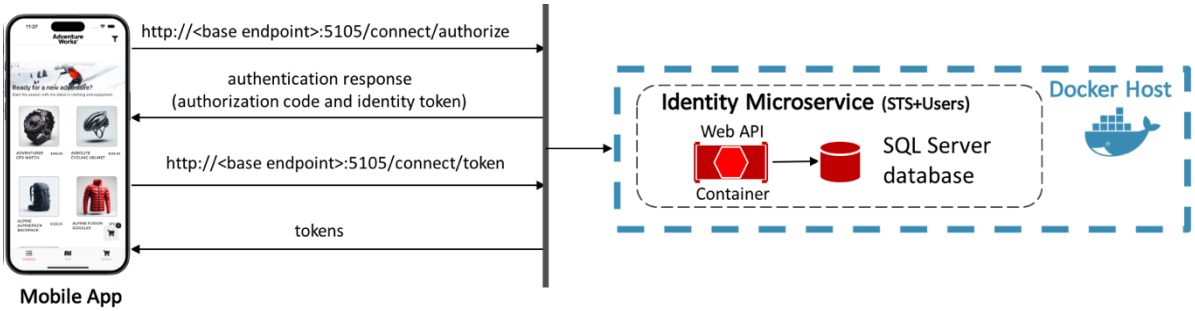
Consider using the hybrid authentication flow. The hybrid authentication flow mitigates a number of attacks that apply to the browser channel, and is the recommended flow for native applications that want to retrieve access tokens (and possibly refresh tokens).

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

Performing authentication

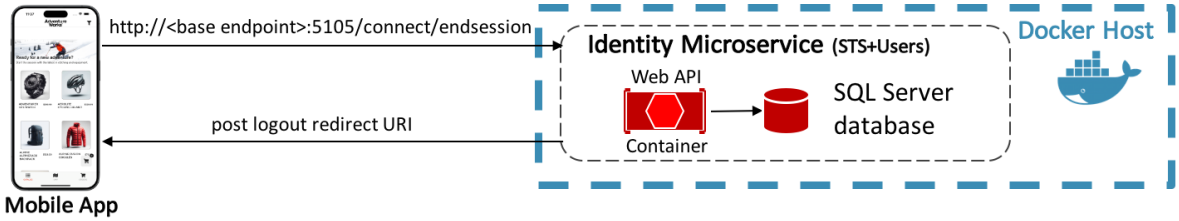
For IdentityServer to issue tokens on behalf of a user, the user must sign in to IdentityServer. However, IdentityServer doesn't provide a user interface or database for authentication. Therefore, in the eShop reference application, ASP.NET Core Identity is used for this purpose.

The eShop multi-platform app authenticates with IdentityServer with the hybrid authentication flow, which is illustrated in the diagram below.



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 sent to <base endpoint>:5105/connect/token, which responds with access, identity, and refresh tokens.

The eShop multi-platform app signs out of IdentityServer by sending a request to <base endpoint>:5105/connect/endsession with additional parameters. After sign-out, IdentityServer responds by sending a post-logout redirecting URI back to the multi-platform app. The diagram below illustrates this process.



In the eShop multi-platform app, communication with IdentityServer is performed by the IdentityService class, which implements the IIdentityService interface. This interface specifies that the implementing class must provide SignInAsync, SignOutAsync, GetUserInfoAsync and GetAuthTokenAsync 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:

```
[RelayCommand]
private async Task SignInAsync()
{
    await IsBusyFor(
        async () =>
        {
            var loginSuccess = await _appEnvironmentService.IdentityService.SignInAsync();

            if (loginSuccess)
            {
                await NavigationService.NavigateToAsync("//Main/Catalog");
            }
```

```
        });
}
```

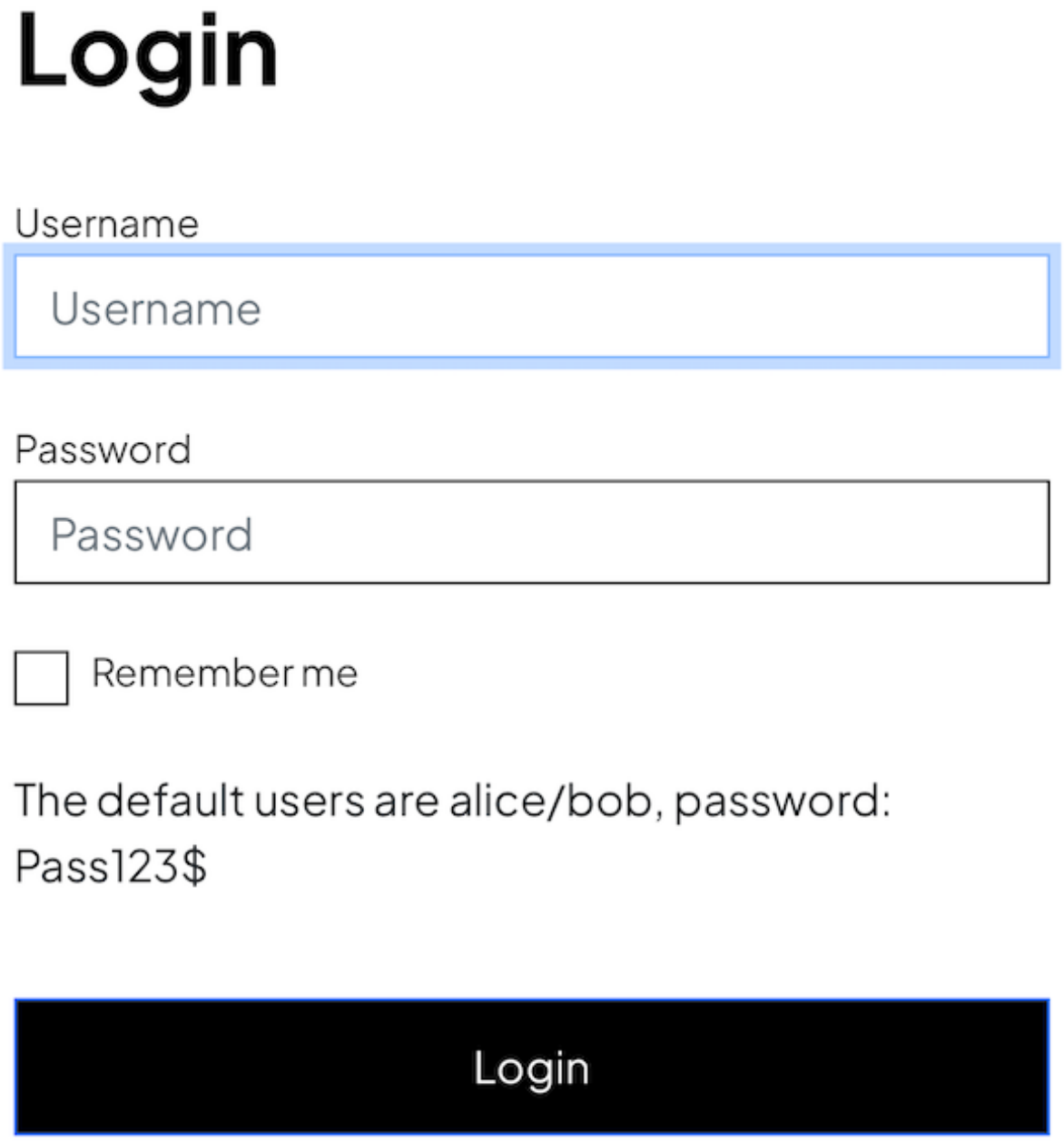
This method invokes the SignInAsync method in the IdentityService class, as shown in the following code example:

```
public async Task<bool> SignInAsync()
{
    var response = await GetClient().LoginAsync(new LoginRequest()).ConfigureAwait(false);

    if (response.IsError)
    {
        return false;
    }

    await _settingsService
        .SetUserTokenAsync(
            new UserToken
            {
                AccessToken = response.AccessToken,
                IdToken = response.IdentityToken,
                RefreshToken = response.RefreshToken,
                ExpiresAt = response.AccessTokenExpiration
            })
        .ConfigureAwait(false);

    return !response.IsError;
}
```

The IdentityService makes use of the OidcClient provided with the IdentityModel.OidcClient NuGet package. This client displays the authentication web view to the user in the application and captures the authentication result. The client connects to 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 eShop multi-platform 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.



If the token endpoint receives valid authentication information, 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 stored as application settings, and page navigation is performed. Therefore, the overall effect in the eShop multi-platform app is this: provided that users are able to successfully authenticate with IdentityServer, they are navigated to the //Main/Catalog route, 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 eShop 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 executes the LogoutAsync method. This method performs page navigation to the LoginView page, passing a Logout query parameter set to true.

That parameter is evaluated in the ApplyQueryAttributes method. If the Logout parameter is present with a true value, the PerformLogoutAsync method of the LoginViewModel class is executed, which is shown in the following code example:

```
private async Task PerformLogoutAsync()
{
    await _appEnvironmentService.IdentityService.SignOutAsync();

    _settingsService.UseFakeLocation = false;

    UserName.Value = string.Empty;
    Password.Value = string.Empty;
}
```

This method invokes the SignOutAsync method in the IdentityService class, which invokes the OidcClient to end the user's session and clears any saved user tokens. For more information about application settings, see Configuration management. The following code example shows the SignOutAsync method:

```
public async Task<bool> SignOutAsync()
{
    var response = await GetClient().LogoutAsync(new
LogoutRequest()).ConfigureAwait(false);

    if (response.IsError)
    {
        return false;
    }

    await _settingsService.SetUserTokenAsync(default);

    return !response.IsError;
}
```

This method uses the OidcClient to call 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. Once the user has successfully signed out, LoginView is presented to the user, and any saved user information will be cleared.

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 eShop 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.

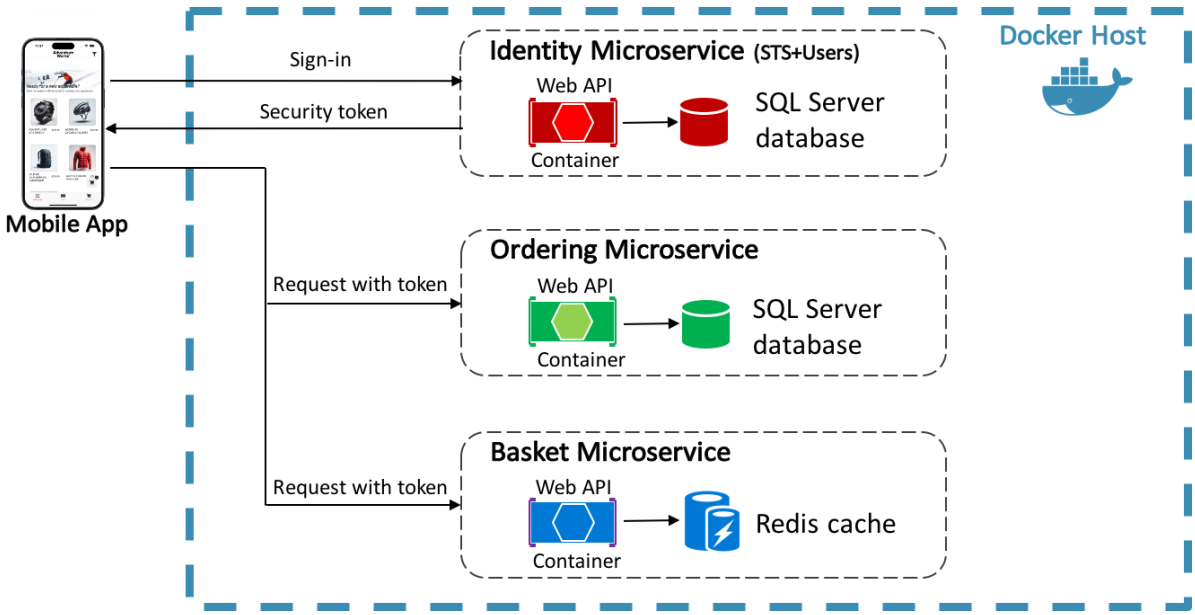
Restricting access to an ASP.NET Core route can be achieved by applying an Authorize attribute to a controller or action, which limits access to the controller or action to authenticated users, as shown in the following code example:

```
[Authorize]
public sealed class BasketController : Controller
{
    // Omitted for brevity
}
```

If an unauthorized user attempts to access a controller or action marked with the Authorize attribute, the API 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 ASP.NET Core Docs: Authorization.

IdentityServer can be integrated into the authorization workflow so that the access tokens provide control authorization. This approach is shown in the diagram below.



The eShop multi-platform 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 AddDefaultAuthentication extension method, which is invoked from the AddApplicationServices method in the Program class and is demonstrated in the following code example from the eShop reference application:

```
public static IServiceCollection AddDefaultAuthentication(this IHostApplicationBuilder
builder)
{
    var services = builder.Services;
    var configuration = builder.Configuration;

    var identitySection = configuration.GetSection("Identity");

    if (!identitySection.Exists())
    {
        // No identity section, so no authentication
        return services;
    }

    // prevent from mapping "sub" claim to nameidentifier.
    JsonWebTokenHandler.DefaultInboundClaimTypeMap.Remove("sub");

    services.AddAuthentication().AddJwtBearer(options =>
    {
```

```
        var identityUrl = identitySection.GetRequiredValue("Url");
        var audience = identitySection.GetRequiredValue("Audience");

        options.Authority = identityUrl;
        options.RequireHttpsMetadata = false;
        options.Audience = audience;
        options.TokenValidationParameters.ValidIssuers = [identityUrl];
        options.TokenValidationParameters.ValidateAudience = false;
    });

    services.AddAuthorization();

    return services;
}
```

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.

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:

```
public async Task CreateOrderAsync(Models.Orders.Order newOrder)
{
    var authToken = await _identityService.GetAuthTokenAsync().ConfigureAwait(false);

    if (string.IsNullOrEmpty(authToken))
    {
        return;
    }

    var uri = $"{UriHelper.CombineUri(_settingsService.GatewayOrdersEndpointBase,
ApiUrlBase)}?api-version=1.0";

    var success = await _requestProvider.PostAsync(uri, newOrder, authToken, "x-
requestid").ConfigureAwait(false);
}
```

The access token is stored with the IIdentityService implementation and can be retrieved using the GetAuthTokenAsync method.

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

```
public async Task ClearBasketAsync()
{
    var authToken = await _identityService.GetAuthTokenAsync().ConfigureAwait(false);

    if (string.IsNullOrEmpty(authToken))
    {
        return;
```

```
    }

    await GetBasketClient().DeleteBasketAsync(new DeleteBasketRequest(),
CreateAuthenticationHeaders(authToken))
        .ConfigureAwait(false);
}
```

The access token is retrieved from the IIdentityService and included in the call to the ClearBasketAsync method in the BasketService class.

The RequestProvider class in the eShop multi-platform app uses the HttpClient class to make requests to the RESTful APIs exposed by the eShop 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", to
ken);
```

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 eShop multi-platform app makes web requests, see Accessing remote data.

Summary

There are many approaches to integrating authentication and authorization into a .NET MAUI app that communicates with an ASP.NET web application. The eShop multi-platform app performs authentication and authorization with a containerized identity microservice that uses IdentityServer. 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 multi-platform app requests security tokens from IdentityServer to authenticate a user or access 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.

MVVM Toolkit Features

MVVM Toolkit

The Model-View-ViewModel (MVVM) pattern is a great structural basis for creating our applications. In this pattern, the ViewModel becomes the backbone of our application as it provides communication to our front-end user interface and backing components. To provide integration with the user interface, we will rely on the ViewModel's properties and commands. As detailed in Updating views in response to changes in the underlying view model or model, the INotifyPropertyChanged interface on our ViewModel to allows changes to our properties to notify when the value is changed. Implementing all of these features means that our ViewModel can end up becoming very verbose. For example, the following code shows a simple ViewModel with properties that raise changes:

```
public class SampleViewModel : INotifyPropertyChanged
{
    private string _name;
    private int _value;

    public event PropertyChangedEventHandler PropertyChanged;

    public string Name
    {
        get => _name;
        set => SetPropertyValue(ref _name, value);
    }

    public int Value
    {
        get => _value;
        set => SetPropertyValue(ref _value, value);
    }

    protected void SetPropertyValue<T>(ref T storageField, T newValue, [CallerMemberName]
string propertyName = "")
    {
        if (Equals(storageField, newValue))
            return;

        storageField = newValue;
        RaisePropertyChanged(propertyName);
    }

    protected virtual void RaisePropertyChanged([CallerMemberName] string propertyName =
"")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

While some optimizations could be made over time, we will still end up with a fairly verbose set of code for defining our ViewModel. This code can be difficult to maintain and becomes error-prone.

The CommunityToolkit.Mvvm NuGet Package (aka MVVM Toolkit) can be used to help address and simplify these common MVVM patterns. The MVVM Toolkit, along with newer features to the .NET language, allows for simplified logic, easy adoption into a project, and runtime independence. The example below shows the same ViewModel using components that come with the MVVM Toolkit:

```
public partial class SampleViewModel : ObservableObject
{
    [ObservableProperty]
    private string _name;

    [ObservableProperty]
    private int _value;
}
```

Note

The MVVM Toolkit is provided with the CommunityToolkit.Mvvm package. For information on how to add the package to your project, see Introduction to the MVVM Toolkit on the Microsoft Developer Center.

In comparison to the original example, we were able to drastically reduce the overall complexity and simplify the maintainability of our ViewModel. The MVVM Toolkit comes with many pre-built common components and features, such as the ObservableObject shown above, that simplifies and standardizes the code that we have throughout the application.

ObservableObject

The MVVM Toolkit provides ObservableObject which is intended for use as the base of our ViewModel objects or any object that needs to raise change notifications. It implements INotifyPropertyChanged and INotifyPropertyChanging along with helper methods for setting properties and raising changes. Below is an example of a standard ViewModel using ObservableObject:

```
public class SampleViewModel : ObservableObject
{
    private string _name;
    private int _value;

    public string Name
    {
        get => _name;
        set => SetProperty(ref _name, value);
    }

    public int Value
    {
        get => _value;
        set => SetProperty(ref _value, value);
```

```
    }
}
```

ObservableObject handles all of the logic needed for raising change notifications by using the SetProperty method in your property setter. If you have a property that returns a Task<T>, the SetPropertyAndNotifyOnCompletion method can be used to delay publishing a property change until the task has been completed. The methods OnPropertyChanged and OnPropertyChanging that can also be used for raising property changes where needed in your object.

For more detailed information on ObservableObject, see ObservableObject in the MVVM Toolkit Developer Center.

RelayCommand and AsyncRelayCommand

Interaction between .NET MAUI controls (for example, tapping a button or selecting an item from a collection) and the ViewModel is done with the ICommand interface. .NET MAUI comes with a default implementation of ICommand with the Command object. .NET MAUI's Command is fairly basic and lacks support for more advanced features, such as supporting asynchronous work and command execution status.

The MVVM Toolkit comes with two commands, RelayCommand and AsyncRelayCommand. RelayCommand is intended for situations where you have synchronous code to execute and has a fairly similar implementation to the .NET MAUI Command object.

Note

Even though the .NET MAUI Command and RelayCommand are similar, using RelayCommand allows for decoupling your ViewModel from any direct .NET MAUI references. This means that your ViewModel is more portable, leading to easier reuse across projects.

AsyncRelayCommand provides many additional features when working with asynchronous workflows. This is quite common in our ViewModel as we are typically communicating with repositories, APIs, databases, and other systems that utilize async/await. The AsyncRelayCommand constructor takes in an execution task defined as a Func<Task> or a delegate returning Task as part of the constructor. While the execution task is running, AsyncRelayCommand will monitor the state of the task and provides updates using the IsRunning property. The IsRunning property can be bound to the UI which helps manage control states such as showing loading with an ActivityIndicator or disabling/enabling a control. While the execution task is being executed, the Cancel method can be called to attempt cancellation of the execution task, if supported.

By default, AsyncRelayCommand doesn't allow concurrent execution. This is very helpful in situations where a user could unintentionally tap a control multiple times to execute a long-running or costly operation. During task execution, AsyncRelayCommand will automatically call the CanExecuteChanged event. In .NET MAUI, controls that support the Command and CommandParameter properties, such as Button, will listen to this event and automatically enable or disable it during execution. This functionality can be overridden by using a custom canExecute parameter or setting the AsyncRelayCommandOptions.AllowConcurrentExecutions flag in the constructor.

For more detailed information on implementing commands, see the section Implementing commands in the MVVM chapter. Detailed information for the RelayCommand and AsyncRelayCommand is available in the Commanding of the MVVM Toolkit Developer Center.

Source Generators

Using the MVVM Toolkit components out-of-the-box allows you to greatly simplify our ViewModel. The MVVM Toolkit allows you to simplify common code use cases even further by using Source Generators. The MVVM Toolkit source generators look for specific attributes in our code and can generate wrappers for properties and commands.

Important

The MVVM Toolkit Source Generators generate code that is additive to our existing objects. Because of this, any object that is leveraging a source generator will need to be marked as partial.

The MVVM Toolkit ObservableProperty attribute can be applied to fields in objects that inherit from ObservableObject and will wrap a private field with a property that generates changes. The following code shows an example of using the ObservableObject attribute on the _name field:

```
public partial class SampleViewModel : ObservableObject
{
    [ObservableProperty]
    private string _name;
}
```

With the ObservableProperty attribute applied to the _name field, the source generator will run and generate another partial class with the following code:

```
partial class SampleViewModel
{
    public string Name
    {
        get => _name;
        set
        {
            if (!EqualityComparer<string>.Default.Equals(_name, value))
            {
                OnNameChanging(value);
                OnPropertyChanging("Name");
                _name = value;
                OnNameChanged(value);
                OnPropertyChanged("Name");
            }
        }
    }
}
```

The generated SampleViewModel has used the private _name field and generated a new Name property that implements all of the logic needed for raising change notifications.

The MVVM Toolkit RelayCommand attribute can be applied to methods within an ObservableObject and will create a corresponding RelayCommand or AsyncRelayCommand. The following code shows examples of using the RelayCommand attribute:

```
public partial class SampleViewModel : ObservableObject
{
    public INavigationService NavigationService { get; set; }

    [ObservableProperty]
    private string _name;

    [ObservableProperty]
    bool _isValid;

    [RelayCommand]
    private Task SettingsAsync()
    {
        return NavigationService.NavigateToAsync("Settings");
    }

    [RelayCommand]
    private void Validate()
    {
        IsValid = !string.IsNullOrEmpty(Name);
    }
}
```

The RelayCommand applied to the Validate method will generate a RelayCommand validate ValidateCommand because it has a void return and the SettingsAsync method will generate an AsyncRelayCommand named SettingsCommand. The source generator will generate the following code in other partial classes:

```
partial class SampleViewModel
{
    private AsyncRelayCommand? settingsCommand;

    SettingsCommand => settingsCommand ??= new AsyncRelayCommand(SettingsAsync);
}

partial class SampleViewModel
{
    private RelayCommand? validateCommand;

    public IRelayCommand ValidateCommand => validateCommand ??= new RelayCommand(Validate);
}
```

All of the complexity of wrapping our ViewModel's methods with an ICommand implementation has been handled by the source generator.

For more detailed information on MVVM Toolkit Source Generators, see MVVM source generators in the MVVM Toolkit Developer Center.

Summary

The MVVM Toolkit is a great way to standardize and simplify our ViewModel code. The MVVM toolkit offers great implementations of standard MVVM components such as ObservableObject and Async/RelayCommand. The source generators help simplify our ViewModel properties and commands by generating all of the boilerplate code needed for user interface interactions. The MVVM Toolkit offers even more features outside of what has been shown in this chapter. For more information on the MVVM Toolkit, see Introduction to the MVVM Toolkit in the MVVM Toolkit Developer Center.

Unit testing

multi-platform apps experience problems similar to both desktop and web-based applications. Mobile users will differ by their devices, network connectivity, availability of services, and various other factors. Therefore, multi-platform apps should be tested as they would be used in the real world to improve their quality, reliability, and performance. Many types of testing should be performed on an app, including unit testing, integration testing, and user interface testing. Unit testing is the most common form and essential to building high-quality applications.

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 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 most significant effect on code quality when it's an integral part of the software development workflow. Unit tests can act as design documentation and functional specifications for an application. As soon as a method has been written, unit tests should be written that verify the method's behavior in response to standard, boundary, and incorrect input data cases and check any explicit or implicit assumptions made by the code. Alternatively, with test-driven development, unit tests are written before the code. For more information on test-driven development and how to implement it, see Walkthrough: Test-driven development using Test Explorer.

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:

Step	Description
Arrange	Initializes objects and sets the value of the data that is passed to the method under test.
Act	Invokes the method under test with the required arguments.
Assert	Verifies that the action of the method under test behaves as expected.

This pattern ensures that unit tests are readable, self-describing, 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 the dependency injection service is the IAppEnvironmentService interface. The following code example shows an outline of this class:

```
public class OrderDetailViewModel : ViewModelBase
{
    private IAppEnvironmentService _appEnvironmentService;

    public OrderDetailViewModel(
        IAppEnvironmentService appEnvironmentService,
        IDialogService dialogService, INavigationService navigationService,
ISettingsService settingsService)
        : base(dialogService, navigationService, settingsService)
    {
        _appEnvironmentService = appEnvironmentService;
    }
}
```

The OrderDetailViewModel class has a dependency on the IAppEnvironmentService type, which the dependency injection container resolves when it instantiates an OrderDetailViewModel object. However, rather than create an IAppEnvironmentService object which utilizes real servers, devices and configurations to unit test the OrderDetailViewModel class, instead, replace the IAppEnvironmentService object with a mock object for the purpose of the tests. A mock object is one that has the same signature of an object or an interface, but is created in a specific manner to help with unit testing. It is often used with dependency injection to provide specific implementations of interfaces for testing different data and workflow scenarios.

This approach allows the IAppEnvironmentService object to be passed into the OrderDetailViewModel class at runtime, and in the interests of testability, it allows a mock 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 runtime platform features, web services, or databases.

Testing MVVM applications

Testing models and view models from MVVM applications is identical to testing any other class, and uses the same tools and techniques; this includes features such as unit testing and mocking. However, some patterns that are typical to model and view model classes can benefit from specific unit testing techniques.

Tip

Test one thing with each unit test. As the complexity of a test expands, it makes verification of that test more difficult. By limiting a unit test to a single concern, we can ensure that our tests are more repeatable, isolated, and have a smaller execution time. See

Unit testing best practices with .NET for more best practices.

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 eShop multi-platform app uses MSTest to perform unit testing, which supports two different types of unit tests:

Testing Type	Attribute	Description
TestMethod	TestMethod	Defines the actual test method to run..
DataSource	DataSource	Tests that are only true for a particular set of data.

The unit tests included with the eShop multi-platform app are TestMethod, so each unit test method is decorated with the TestMethod attribute. In addition to MSTest there are several other testing frameworks available including NUnit and xUnit.

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:

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

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

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

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:

```
[TestMethod]
public async Task SettingOrderPropertyShouldRaisePropertyChanged()
{
    var 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.IsTrue(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:

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

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

```
    Assert.IsTrue(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:

```
[TestMethod]
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:

```
[TestMethod]
public void CheckValidationPassesWhenBothPropertiesHaveDataTest()
{
```

```
    var mockViewModel = new MockViewModel();
    mockViewModel.Forename.Value = "John";
    mockViewModel.Surname.Value = "Smith";

    var isValid = mockViewModel.Validate();

    Assert.IsTrue(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:

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

    bool isValid = mockViewModel.Validate();

    Assert.IsFalse(isValid);
    Assert.IsNotNull(mockViewModel.Forename.Value);
    Assert.IsNull(mockViewModel.Surname.Value);
    Assert.IsTrue(mockViewModel.Forename.IsValid);
    Assert.IsFalse(mockViewModel.Surname.IsValid);
    Assert.AreEqual(mockViewModel.Forename.Errors.Count(), 0);
    Assert.AreNotEqual(mockViewModel.Surname.Errors.Count(), 0);
}
```

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 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 runtime platform features, 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.