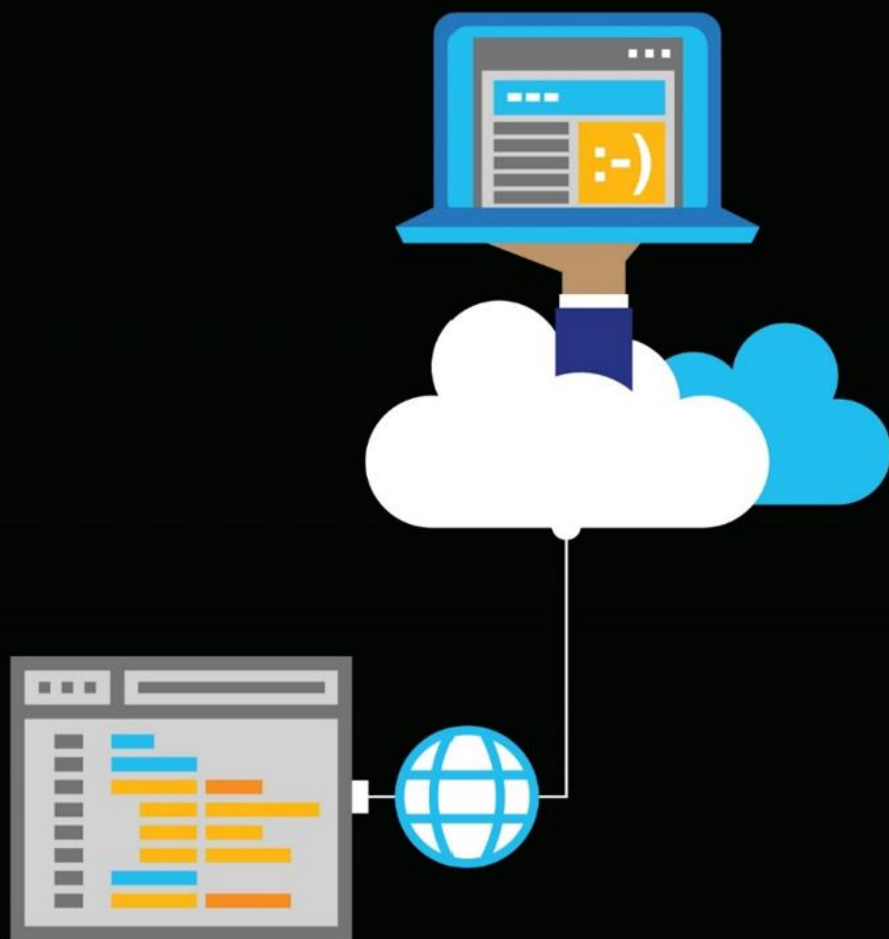


Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure



Steve “ardalis” Smith

EDITION v7.0 - Updated to ASP.NET Core 7.0

Refer [changelog](#) for the book updates and community contributions.

PUBLISHED BY

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

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2023 by Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

This book is provided “as-is” and expresses the author’s views and opinions. The views, opinions, and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at <https://www.microsoft.com> on the “Trademarks” webpage are trademarks of the Microsoft group of companies.

Mac and macOS are trademarks of Apple Inc.

The Docker whale logo is a registered trademark of Docker, Inc. Used by permission.

All other marks and logos are property of their respective owners.

Author:

Steve “ardalis” Smith - Software Architect and Trainer - [Ardalis.com](https://ardalis.com)

Editors:

Maira Wenzel

Action links

- This e-book is also available in a PDF format (English version only) [Download](#)
- Clone/Fork the reference application [eShopOnWeb on GitHub](#)

Introduction

.NET 7 and ASP.NET Core offer several advantages over traditional .NET development. You should use .NET 7 for your server applications if some or all of the following are important to your application’s success:

- Cross-platform support.

- Use of microservices.
- Use of Docker containers.
- High performance and scalability requirements.
- Side-by-side versioning of .NET versions by application on the same server.

Traditional .NET 4.x apps can and do support many of these requirements, but ASP.NET Core and .NET 7 have been optimized to offer improved support for the above scenarios.

More and more organizations are choosing to host their web applications in the cloud using services like Microsoft Azure. You should consider hosting your application in the cloud if the following are important to your application or organization:

- Reduced investment in data center costs (hardware, software, space, utilities, server management, etc.)
- Flexible pricing (pay based on usage, not for idle capacity).
- Extreme reliability.
- Improved app mobility; easily change where and how your app is deployed.
- Flexible capacity; scale up or down based on actual needs.

Building web applications with ASP.NET Core, hosted in Azure, offers many competitive advantages over traditional alternatives. ASP.NET Core is optimized for modern web application development practices and cloud hosting scenarios. In this guide, you'll learn how to architect your ASP.NET Core applications to best take advantage of these capabilities.

Version

This guide has been revised to cover **.NET 7.0** version along with many additional updates related to the same "wave" of technologies (that is, Azure and additional third-party technologies) coinciding in time with the .NET 7.0 release. That's why the book version has also been updated to version **7.0**.

Purpose

This guide provides end-to-end guidance on building *monolithic* web applications using ASP.NET Core and Azure. In this context, "monolithic" refers to the fact that these applications are deployed as a single unit, not as a collection of interacting services and applications. In some contexts, the term *monolith* may be used as a pejorative, but in the vast majority of situations a single application is much easier to build, deploy, and debug than an app composed of many different services, while still achieving the business requirements.

This guide is complementary to [".NET Microservices. Architecture for Containerized .NET Applications"](#), which focuses more on Docker, microservices, and deployment of containers to host enterprise applications.

.NET Microservices. Architecture for Containerized .NET Applications

- **e-book**
<https://aka.ms/MicroservicesEbook>
- **Sample Application**
<https://aka.ms/microservicesarchitecture>

Who should use this guide

The audience for this guide is mainly developers, development leads, and architects who are interested in building modern web applications using Microsoft technologies and services in the cloud.

A secondary audience is technical decision makers who are already familiar ASP.NET or Azure and are looking for information on whether it makes sense to upgrade to ASP.NET Core for new or existing projects.

How you can use this guide

This guide has been condensed into a relatively small document that focuses on building web applications with modern .NET technologies and Azure. As such, it can be read in its entirety to provide a foundation of understanding such applications and their technical considerations. The guide, along with its sample application, can also serve as a starting point or reference. Use the associated sample application as a template for your own applications, or to see how you might organize your application's component parts. Refer back to the guide's principles and coverage of architecture and technology options and decision considerations when you're weighing these choices for your own application.

Feel free to forward this guide to your team to help ensure a common understanding of these considerations and opportunities. Having everybody working from a common set of terminology and underlying principles helps ensure consistent application of architectural patterns and practices.

References

- **Choosing between .NET and .NET Framework for server apps**
<https://learn.microsoft.com/dotnet/standard/choosing-core-framework-server>

Contents

| | |
|-----------------------------------------------------------------------|----|
| Characteristics of Modern Web Applications | 1 |
| Reference application: eShopOnWeb | 1 |
| Reference Application | 2 |
| Cloud-hosted and scalable | 2 |
| Cross platform | 2 |
| Modular and loosely coupled | 3 |
| Easily tested with automated tests | 3 |
| Traditional and SPA behaviors supported | 3 |
| Simple development and deployment | 4 |
| Traditional ASP.NET and Web Forms | 4 |
| Blazor | 4 |
| References – Modern Web Applications | 4 |
| Choose Between Traditional Web Apps and Single Page Apps (SPAs) | 6 |
| Blazor | 7 |
| When to choose traditional web apps | 7 |
| When to choose SPAs | 8 |
| References – SPA Frameworks | 8 |
| When to choose Blazor | 9 |
| Decision table | 9 |
| Other considerations | 9 |
| Architectural principles | 11 |
| Common design principles | 11 |
| Separation of concerns | 11 |
| Encapsulation | 11 |
| Dependency inversion | 12 |
| Explicit dependencies | 14 |
| Single responsibility | 14 |
| Don't repeat yourself (DRY) | 15 |

| | |
|-------------------------------------------------------|----|
| Persistence ignorance | 15 |
| Bounded contexts | 16 |
| Additional resources | 16 |
| Common web application architectures | 17 |
| What is a monolithic application? | 17 |
| All-in-one applications | 17 |
| What are layers? | 18 |
| Traditional “N-Layer” architecture applications | 19 |
| Clean architecture | 23 |
| Organizing code in Clean Architecture | 28 |
| Monolithic applications and containers | 30 |
| Monolithic application deployed as a container | 31 |
| Docker support | 33 |
| Troubleshooting Docker problems | 34 |
| Other web application architectural styles | 34 |
| References – Common web architectures | 34 |
| Common client-side web technologies | 36 |
| HTML | 36 |
| CSS | 36 |
| CSS preprocessors | 37 |
| JavaScript | 38 |
| Legacy web apps with jQuery | 38 |
| jQuery vs a SPA Framework | 38 |
| Angular SPAs | 39 |
| React | 40 |
| Vue | 40 |
| Blazor WebAssembly | 41 |
| Choosing a SPA Framework | 41 |
| References – Client Web Technologies | 42 |
| Develop ASP.NET Core MVC apps | 43 |
| MVC and Razor Pages | 43 |

| | |
|-------------------------------------------------------|----|
| Why Razor Pages? | 44 |
| When to use MVC | 44 |
| Mapping requests to responses | 44 |
| Keeping controllers under control | 46 |
| References – Mapping Requests to Responses | 48 |
| Working with dependencies..... | 48 |
| Declare your dependencies..... | 49 |
| Structuring the application | 50 |
| Feature organization..... | 51 |
| APIs and Blazor applications..... | 53 |
| Cross-cutting concerns | 54 |
| References – Structuring applications | 57 |
| Security | 57 |
| Identity | 57 |
| Authentication..... | 60 |
| References – Authentication..... | 61 |
| Authorization | 61 |
| References – Security..... | 64 |
| Client communication | 64 |
| References – Client Communication | 65 |
| Domain-driven design – Should you apply it? | 65 |
| When should you apply DDD..... | 66 |
| When shouldn't you apply DDD | 66 |
| References – Domain-Driven Design | 67 |
| Deployment..... | 67 |
| References – Deployment..... | 68 |
| Working with Data in ASP.NET Core Apps..... | 69 |
| Entity Framework Core (for relational databases)..... | 69 |
| The DbContext | 69 |
| Configuring EF Core..... | 70 |
| Fetching and storing Data | 71 |
| Fetching related data..... | 72 |

| | |
|----------------------------------------------------------------------|-----|
| Encapsulating data | 73 |
| Resilient connections | 74 |
| References – Entity Framework Core | 76 |
| EF Core or micro-ORM? | 76 |
| SQL or NoSQL | 77 |
| Azure Cosmos DB | 78 |
| Other persistence options | 79 |
| Caching | 80 |
| ASP.NET Core response caching | 80 |
| Data caching | 81 |
| Getting data to Blazor WebAssembly apps | 83 |
| Test ASP.NET Core MVC apps | 85 |
| Kinds of automated tests | 85 |
| Unit tests | 85 |
| Integration tests | 85 |
| Functional tests | 86 |
| Testing Pyramid | 86 |
| What to test | 87 |
| Organizing test projects | 88 |
| Test naming | 89 |
| Unit testing ASP.NET Core apps | 91 |
| Integration testing ASP.NET Core apps | 92 |
| Functional testing ASP.NET Core apps | 92 |
| References – Test ASP.NET Core MVC apps | 95 |
| Development process for Azure | 96 |
| Vision | 96 |
| Development environment for ASP.NET Core apps | 96 |
| Development tools choices: IDE or editor | 96 |
| Development workflow for Azure-hosted ASP.NET Core apps | 97 |
| Initial setup | 97 |
| Workflow for developing Azure-hosted ASP.NET Core applications | 98 |
| References | 100 |

| | |
|---------------------------------------------------------------|-----|
| Azure hosting recommendations for ASP.NET Core web apps | 101 |
| Web applications | 101 |
| App Service Web Apps | 101 |
| App Service Web Apps for Containers | 103 |
| Azure Kubernetes Service | 106 |
| Azure Virtual Machines | 107 |
| Logical processes | 107 |
| Data | 107 |
| Architecture recommendations..... | 108 |

Characteristics of Modern Web Applications

"... with proper design, the features come cheaply. This approach is arduous, but continues to succeed."

- *Dennis Ritchie*

Modern web applications have higher user expectations and greater demands than ever before. Today's web apps are expected to be available 24/7 from anywhere in the world, and usable from virtually any device or screen size. Web applications must be secure, flexible, and scalable to meet spikes in demand. Increasingly, complex scenarios should be handled by rich user experiences built on the client using JavaScript, and communicating efficiently through web APIs.

ASP.NET Core is optimized for modern web applications and cloud-based hosting scenarios. Its modular design enables applications to depend on only those features they actually use, improving application security and performance while reducing hosting resource requirements.

Reference application: eShopOnWeb

This guidance includes a reference application, *eShopOnWeb*, that demonstrates some of the principles and recommendations. The application is a simple online store, which supports browsing through a catalog of shirts, coffee mugs, and other marketing items. The reference application is deliberately simple in order to make it easy to understand.

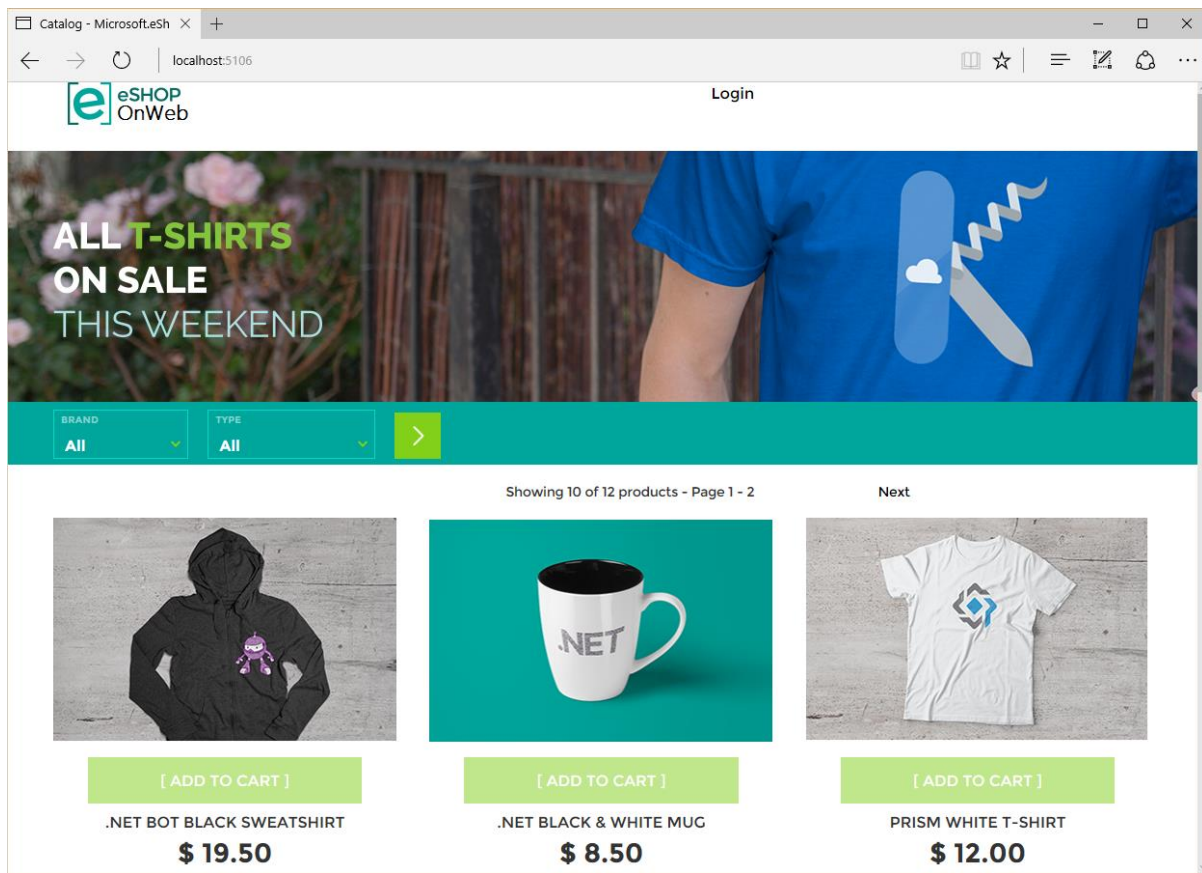


Figure 2-1. eShopOnWeb

Reference Application

- **eShopOnWeb**
<https://github.com/dotnet/eShopOnWeb>

Cloud-hosted and scalable

ASP.NET Core is optimized for the cloud (public cloud, private cloud, any cloud) because it is low-memory and high-throughput. The smaller footprint of ASP.NET Core applications means you can host more of them on the same hardware, and you pay for fewer resources when using pay-as-you-go cloud hosting services. The higher-throughput means you can serve more customers from an application given the same hardware, further reducing the need to invest in servers and hosting infrastructure.

Cross platform

ASP.NET Core is cross-platform and can run on Linux, macOS, and Windows. This capability opens up many new options for both the development and deployment of apps built with ASP.NET Core.

Docker containers - both Linux and Windows - can host ASP.NET Core applications, allowing them to take advantage of the benefits of [containers and microservices](#).

Modular and loosely coupled

NuGet packages are first-class citizens in .NET Core, and ASP.NET Core apps are composed of many libraries through NuGet. This granularity of functionality helps ensure apps only depend on and deploy functionality they actually require, reducing their footprint and security vulnerability surface area.

ASP.NET Core also fully supports [dependency injection](#), both internally and at the application level. Interfaces can have multiple implementations that can be swapped out as needed. Dependency injection allows apps to loosely couple to those interfaces, rather than specific implementations, making them easier to extend, maintain, and test.

Easily tested with automated tests

ASP.NET Core applications support unit testing, and their loose coupling and support for dependency injection makes it easy to swap infrastructure concerns with fake implementations for test purposes. ASP.NET Core also ships with a TestServer that can be used to host apps in memory. Functional tests can then make requests to this in-memory server, exercising the full application stack (including middleware, routing, model binding, filters, etc.) and receiving a response, all in a fraction of the time it would take to host the app on a real server and make requests through the network layer. These tests are especially easy to write, and valuable, for APIs, which are increasingly important in modern web applications.

Traditional and SPA behaviors supported

Traditional web applications have involved little client-side behavior, but instead have relied on the server for all navigation, queries, and updates the app might need to make. Each new operation made by the user would be translated into a new web request, with the result being a full page reload in the end user's browser. Classic Model-View-Controller (MVC) frameworks typically follow this approach, with each new request corresponding to a different controller action, which in turn would work with a model and return a view. Some individual operations on a given page might be enhanced with AJAX (Asynchronous JavaScript and XML) functionality, but the overall architecture of the app used many different MVC views and URL endpoints. In addition, ASP.NET Core MVC also supports Razor Pages, a simpler way to organize MVC-style pages.

Single Page Applications (SPAs), by contrast, involve very few dynamically generated server-side page loads (if any). Many SPAs are initialized within a static HTML file that loads the necessary JavaScript libraries to start and run the app. These apps make heavy usage of web APIs for their data needs and can provide much richer user experiences. Blazor WebAssembly provides a means of building SPAs using .NET code, which then runs in the client's browser.

Many web applications involve a combination of traditional web application behavior (typically for content) and SPAs (for interactivity). ASP.NET Core supports both MVC (Views or Page based) and web APIs in the same application, using the same set of tools and underlying framework libraries.

Simple development and deployment

ASP.NET Core applications can be written using simple text editors and command-line interfaces, or full-featured development environments like Visual Studio. Monolithic applications are typically deployed to a single endpoint. Deployments can easily be automated to occur as part of a continuous integration (CI) and continuous delivery (CD) pipeline. In addition to traditional CI/CD tools, Microsoft Azure has integrated support for git repositories and can automatically deploy updates as they are made to a specified git branch or tag. Azure DevOps provides a full-featured CI/CD build and deployment pipeline, and GitHub Actions provide another option for projects hosted there.

Traditional ASP.NET and Web Forms

In addition to ASP.NET Core, traditional ASP.NET 4.x continues to be a robust and reliable platform for building web applications. ASP.NET supports MVC and Web API development models, as well as Web Forms, which is well suited to rich page-based application development and features a rich third-party component ecosystem. Microsoft Azure has great longstanding support for ASP.NET 4.x applications, and many developers are familiar with this platform.

Blazor

Blazor is included with ASP.NET Core 3.0 and later. It provides a new mechanism for building rich interactive web client applications using Razor, C#, and ASP.NET Core. It offers another solution to consider when developing modern web applications. There are two versions of Blazor to consider: server-side and client-side.

Server-side Blazor was released in 2019 with ASP.NET Core 3.0. As its name implies, it runs on the server, rendering changes to the client document back to the browser over the network. Server-side Blazor provides a rich client experience without requiring client-side JavaScript and without requiring separate page loads for each client page interaction. Changes in the loaded page are requested from and processed by the server and then sent back to the client using SignalR.

Client-side Blazor, released in 2020, eliminates the need to render changes on the server. Instead, it leverages WebAssembly to run .NET code within the client. The client can still make API calls to the server if needed to request data, but all client-side behavior runs in the client via WebAssembly, which is already supported by all major browsers and is just a JavaScript library.

References – Modern Web Applications

- **Introduction to ASP.NET Core**
<https://learn.microsoft.com/aspnet/core/>

- **Testing in ASP.NET Core**
<https://learn.microsoft.com/aspnet/core/testing/>
- **Blazor - Get Started**
<https://blazor.net/docs/get-started.html>

Choose Between Traditional Web Apps and Single Page Apps (SPAs)

“Atwood’s Law: Any application that can be written in JavaScript, will eventually be written in JavaScript.”

- Jeff Atwood

There are two general approaches to building web applications today: traditional web applications that perform most of the application logic on the server, and single-page applications (SPAs) that perform most of the user interface logic in a web browser, communicating with the web server primarily using web APIs. A hybrid approach is also possible, the simplest being host one or more rich SPA-like subapplications within a larger traditional web application.

Use traditional web applications when:

- Your application’s client-side requirements are simple or even read-only.
- Your application needs to function in browsers without JavaScript support.
- Your application is public-facing and benefits from search engine discovery and referrals.

Use a SPA when:

- Your application must expose a rich user interface with many features.
- Your team is familiar with JavaScript, TypeScript, or Blazor WebAssembly development.
- Your application must already expose an API for other (internal or public) clients.

Additionally, SPA frameworks require greater architectural and security expertise. They experience greater churn due to frequent updates and new client frameworks than traditional web applications. Configuring automated build and deployment processes and utilizing deployment options like containers may be more difficult with SPA applications than traditional web apps.

Improvements in user experience made possible by the SPA approach must be weighed against these considerations.

Blazor

ASP.NET Core includes a model for building rich, interactive, and composable user interfaces called Blazor. Blazor server-side allows developers to build UI with C# and Razor on the server and for the UI to be interactively connected to the browser in real-time using a persistent SignalR connection. Blazor WebAssembly introduces another option for Blazor apps, allowing them to run in the browser using WebAssembly. Because it's real .NET code running on WebAssembly, you can reuse code and libraries from server-side parts of your application.

Blazor provides a new, third option to consider when evaluating whether to build a purely server-rendered web application or a SPA. You can build rich, SPA-like client-side behaviors using Blazor, without the need for significant JavaScript development. Blazor applications can call APIs to request data or perform server-side operations. They can interoperate with JavaScript where necessary to take advantage of JavaScript libraries and frameworks.

Consider building your web application with Blazor when:

- Your application must expose a rich user interface
- Your team is more comfortable with .NET development than JavaScript or TypeScript development

If you have an existing web forms application you're considering migrating to .NET Core or the latest .NET, you may wish to review the free e-book, [Blazor for Web Forms Developers](#) to see whether it makes sense to consider migrating it to Blazor.

For more information about Blazor, see [Get started with Blazor](#).

When to choose traditional web apps

The following section is a more detailed explanation of the previously stated reasons for picking traditional web applications.

Your application has simple, possibly read-only, client-side requirements

Many web applications are primarily consumed in a read-only fashion by the vast majority of their users. Read-only (or read-mostly) applications tend to be much simpler than those applications that maintain and manipulate a great deal of state. For example, a search engine might consist of a single entry point with a textbox and a second page for displaying search results. Anonymous users can easily make requests, and there is little need for client-side logic. Likewise, a blog or content management system's public-facing application usually consists mainly of content with little client-side behavior. Such applications are easily built as traditional server-based web applications, which perform logic on the web server and render HTML to be displayed in the browser. The fact that each unique page of the site has its own URL that can be bookmarked and indexed by search engines (by default, without having to add this functionality as a separate feature of the application) is also a clear benefit in such scenarios.

Your application needs to function in browsers without JavaScript support

Web applications that need to function in browsers with limited or no JavaScript support should be written using traditional web app workflows (or at least be able to fall back to such behavior). SPAs require client-side JavaScript in order to function; if it's not available, SPAs are not a good choice.

Your team is unfamiliar with JavaScript or TypeScript development techniques

If your team is unfamiliar with JavaScript or TypeScript, but is familiar with server-side web application development, then they will probably be able to deliver a traditional web app more quickly than a SPA. Unless learning to program SPAs is a goal, or the user experience afforded by a SPA is required, traditional web apps are a more productive choice for teams who are already familiar with building them.

When to choose SPAs

The following section is a more detailed explanation of when to choose a Single Page Applications style of development for your web app.

Your application must expose a rich user interface with many features

SPAs can support rich client-side functionality that doesn't require reloading the page as users take actions or navigate between areas of the app. SPAs can load more quickly, fetching data in the background, and individual user actions are more responsive since full page reloads are rare. SPAs can support incremental updates, saving partially completed forms or documents without the user having to click a button to submit a form. SPAs can support rich client-side behaviors, such as drag-and-drop, much more readily than traditional applications. SPAs can be designed to run in a disconnected mode, making updates to a client-side model that are eventually synchronized back to the server once a connection is re-established. Choose a SPA-style application if your app's requirements include rich functionality that goes beyond what typical HTML forms offer.

Frequently, SPAs need to implement features that are built into traditional web apps, such as displaying a meaningful URL in the address bar reflecting the current operation (and allowing users to bookmark or deep link to this URL to return to it). SPAs also should allow users to use the browser's back and forward buttons with results that won't surprise them.

Your team is familiar with JavaScript and/or TypeScript development

Writing SPAs requires familiarity with JavaScript and/or TypeScript and client-side programming techniques and libraries. Your team should be competent in writing modern JavaScript using a SPA framework like Angular.

References – SPA Frameworks

- **Angular:** <https://angular.io>
- **React:** <https://reactjs.org/>
- **Vue.js:** <https://vuejs.org/>

Your application must already expose an API for other (internal or public) clients

If you're already supporting a web API for use by other clients, it may require less effort to create a SPA implementation that leverages these APIs rather than reproducing the logic in server-side form. SPAs make extensive use of web APIs to query and update data as users interact with the application.

When to choose Blazor

The following section is a more detailed explanation of when to choose Blazor for your web app.

Your application must expose a rich user interface

Like JavaScript-based SPAs, Blazor applications can support rich client behavior without page reloads. These applications are more responsive to users, fetching only the data (or HTML) required to respond to a given user interaction. Designed properly, server-side Blazor apps can be configured to run as client-side Blazor apps with minimal changes once this feature is supported.

Your team is more comfortable with .NET development than JavaScript or TypeScript development

Many developers are more productive with .NET and Razor than with client-side languages like JavaScript or TypeScript. Since the server-side of the application is already being developed with .NET, using Blazor ensures every .NET developer on the team can understand and potentially build the behavior of the front end of the application.

Decision table

The following decision table summarizes some of the basic factors to consider when choosing between a traditional web application, a SPA, or a Blazor app.

| Factor | Traditional Web App | Single Page Application | Blazor App |
|------------------------------------------------------|---------------------|-------------------------|-------------|
| Required Team Familiarity with JavaScript/TypeScript | Minimal | Required | Minimal |
| Support Browsers without Scripting | Supported | Not Supported | Supported |
| Minimal Client-Side Application Behavior | Well-Suited | Overkill | Viable |
| Rich, Complex User Interface Requirements | Limited | Well-Suited | Well-Suited |

Other considerations

Traditional Web Apps tend to be simpler and have better Search Engine Optimization (SEO) criteria than SPA apps. Search engine bots can easily consume content from traditional apps, while support for indexing SPAs may be lacking or limited. If your app benefits from public discovery by search engines, this is often an important consideration.

In addition, unless you've built a management tool for your SPA's content, it may require developers to make changes. Traditional Web Apps are often easier for non-developers to make changes to, since much of the content is simply HTML and may not require a build process to preview or even deploy. If non-developers in your organization are likely to need to maintain the content of the app, a traditional web app may be a better choice.

SPAs shine when the app has complex interactive forms or other user interaction features. For complex apps that require authentication to use, and thus aren't accessible by public search engine spiders, SPAs are a great option in many cases.

Architectural principles

"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."

- *Gerald Weinberg*

You should architect and design software solutions with maintainability in mind. The principles outlined in this section can help guide you toward architectural decisions that will result in clean, maintainable applications. Generally, these principles will guide you toward building applications out of discrete components that are not tightly coupled to other parts of your application, but rather communicate through explicit interfaces or messaging systems.

Common design principles

Separation of concerns

A guiding principle when developing is **Separation of Concerns**. This principle asserts that software should be separated based on the kinds of work it performs. For instance, consider an application that includes logic for identifying noteworthy items to display to the user, and which formats such items in a particular way to make them more noticeable. The behavior responsible for choosing which items to format should be kept separate from the behavior responsible for formatting the items, since these behaviors are separate concerns that are only coincidentally related to one another.

Architecturally, applications can be logically built to follow this principle by separating core business behavior from infrastructure and user-interface logic. Ideally, business rules and logic should reside in a separate project, which should not depend on other projects in the application. This separation helps ensure that the business model is easy to test and can evolve without being tightly coupled to low-level implementation details (it also helps if infrastructure concerns depend on abstractions defined in the business layer). Separation of concerns is a key consideration behind the use of layers in application architectures.

Encapsulation

Different parts of an application should use **encapsulation** to insulate them from other parts of the application. Application components and layers should be able to adjust their internal implementation without breaking their collaborators as long as external contracts are not violated. Proper use of encapsulation helps achieve loose coupling and modularity in application designs, since objects and packages can be replaced with alternative implementations so long as the same interface is maintained.

In classes, encapsulation is achieved by limiting outside access to the class's internal state. If an outside actor wants to manipulate the state of the object, it should do so through a well-defined function (or property setter), rather than having direct access to the private state of the object. Likewise, application components and applications themselves should expose well-defined interfaces for their collaborators to use, rather than allowing their state to be modified directly. This approach frees the application's internal design to evolve over time without worrying that doing so will break collaborators, so long as the public contracts are maintained.

Mutable global state is antithetical to encapsulation. A value fetched from mutable global state in one function cannot be relied upon to have the same value in another function (or even further in the same function). Understanding concerns with mutable global state is one of the reasons programming languages like C# have support for different scoping rules, which are used everywhere from statements to methods to classes. It's worth noting that data-driven architectures which rely on a central database for integration within and between applications are, themselves, choosing to depend on the mutable global state represented by the database. A key consideration in domain-driven design and clean architecture is how to encapsulate access to data, and how to ensure application state is not made invalid by direct access to its persistence format.

Dependency inversion

The direction of dependency within the application should be in the direction of abstraction, not implementation details. Most applications are written such that compile-time dependency flows in the direction of runtime execution, producing a direct dependency graph. That is, if class A calls a method of class B and class B calls a method of class C, then at compile time class A will depend on class B, and class B will depend on class C, as shown in Figure 4-1.

Direct Dependency Graph

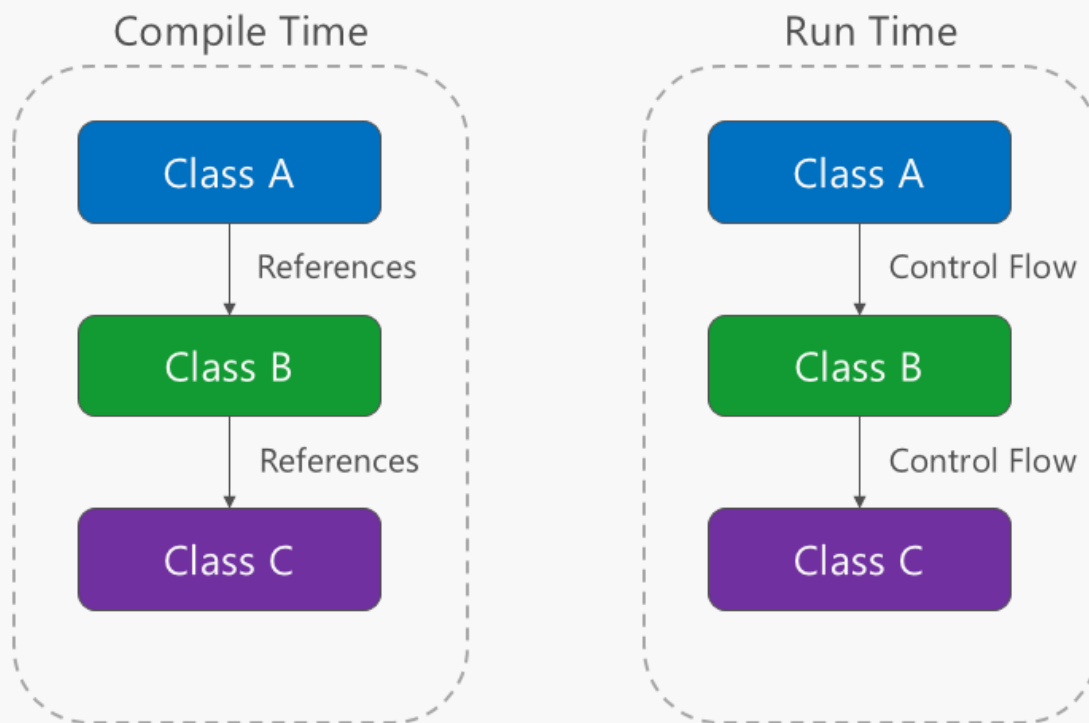


Figure 4-1. Direct dependency graph.

Applying the dependency inversion principle allows A to call methods on an abstraction that B implements, making it possible for A to call B at run time, but for B to depend on an interface controlled by A at compile time (thus, *inverting* the typical compile-time dependency). At run time, the flow of program execution remains unchanged, but the introduction of interfaces means that different implementations of these interfaces can easily be plugged in.

Inverted Dependency Graph

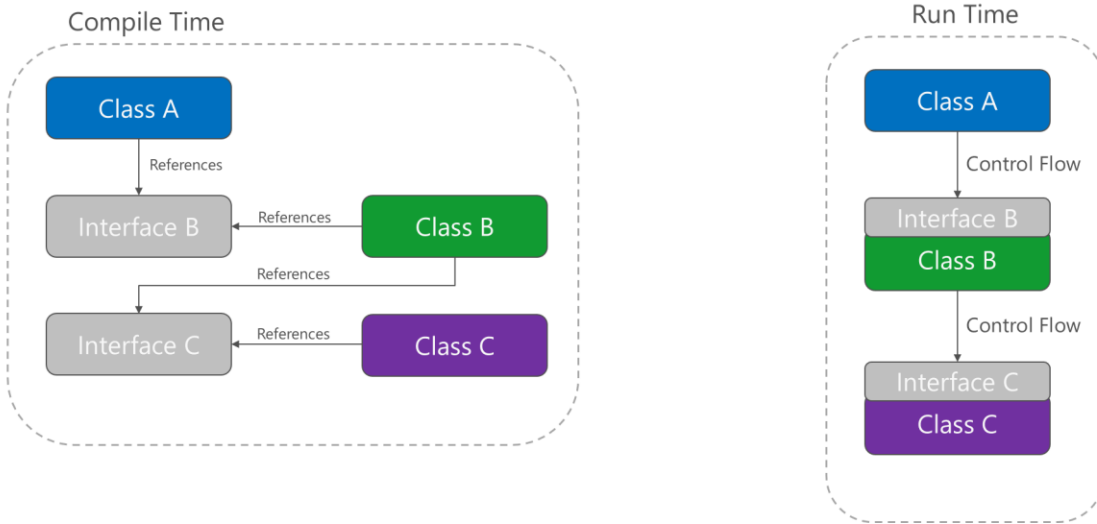


Figure 4-2. Inverted dependency graph.

Dependency inversion is a key part of building loosely coupled applications, since implementation details can be written to depend on and implement higher-level abstractions, rather than the other way around. The resulting applications are more testable, modular, and maintainable as a result. The practice of *dependency injection* is made possible by following the dependency inversion principle.

Explicit dependencies

Methods and classes should explicitly require any collaborating objects they need in order to function correctly. It is called the [Explicit Dependencies Principle](#). Class constructors provide an opportunity for classes to identify the things they need in order to be in a valid state and to function properly. If you define classes that can be constructed and called, but that will only function properly if certain global or infrastructure components are in place, these classes are being *dishonest* with their clients. The constructor contract is telling the client that it only needs the things specified (possibly nothing if the class is just using a parameterless constructor), but then at runtime it turns out the object really did need something else.

By following the explicit dependencies principle, your classes and methods are being honest with their clients about what they need in order to function. Following the principle makes your code more self-documenting and your coding contracts more user-friendly, since users will come to trust that as long as they provide what's required in the form of method or constructor parameters, the objects they're working with will behave correctly at run time.

Single responsibility

The single responsibility principle applies to object-oriented design, but can also be considered as an architectural principle similar to separation of concerns. It states that objects should have only one responsibility and that they should have only one reason to change. Specifically, the only situation in which the object should change is if the manner in which it performs its one responsibility must be

updated. Following this principle helps to produce more loosely coupled and modular systems, since many kinds of new behavior can be implemented as new classes, rather than by adding additional responsibility to existing classes. Adding new classes is always safer than changing existing classes, since no code yet depends on the new classes.

In a monolithic application, we can apply the single responsibility principle at a high level to the layers in the application. Presentation responsibility should remain in the UI project, while data access responsibility should be kept within an infrastructure project. Business logic should be kept in the application core project, where it can be easily tested and can evolve independently from other responsibilities.

When this principle is applied to application architecture and taken to its logical endpoint, you get microservices. A given microservice should have a single responsibility. If you need to extend the behavior of a system, it's usually better to do it by adding additional microservices, rather than by adding responsibility to an existing one.

[Learn more about microservices architecture](#)

Don't repeat yourself (DRY)

The application should avoid specifying behavior related to a particular concept in multiple places as this practice is a frequent source of errors. At some point, a change in requirements will require changing this behavior. It's likely that at least one instance of the behavior will fail to be updated, and the system will behave inconsistently.

Rather than duplicating logic, encapsulate it in a programming construct. Make this construct the single authority over this behavior, and have any other part of the application that requires this behavior use the new construct.

Note

Avoid binding together behavior that is only coincidentally repetitive. For example, just because two different constants both have the same value, that doesn't mean you should have only one constant, if conceptually they're referring to different things. Duplication is always preferable to coupling to the wrong abstraction.

Persistence ignorance

Persistence ignorance (PI) refers to types that need to be persisted, but whose code is unaffected by the choice of persistence technology. Such types in .NET are sometimes referred to as Plain Old CLR Objects (POCOs), because they do not need to inherit from a particular base class or implement a particular interface. Persistence ignorance is valuable because it allows the same business model to be persisted in multiple ways, offering additional flexibility to the application. Persistence choices might change over time, from one database technology to another, or additional forms of persistence might be required in addition to whatever the application started with (for example, using a Redis cache or Azure Cosmos DB in addition to a relational database).

Some examples of violations of this principle include:

- A required base class.
- A required interface implementation.
- Classes responsible for saving themselves (such as the Active Record pattern).
- Required parameterless constructor.
- Properties requiring virtual keyword.
- Persistence-specific required attributes.

The requirement that classes have any of the above features or behaviors adds coupling between the types to be persisted and the choice of persistence technology, making it more difficult to adopt new data access strategies in the future.

Bounded contexts

Bounded contexts are a central pattern in Domain-Driven Design. They provide a way of tackling complexity in large applications or organizations by breaking it up into separate conceptual modules. Each conceptual module then represents a context that is separated from other contexts (hence, bounded), and can evolve independently. Each bounded context should ideally be free to choose its own names for concepts within it, and should have exclusive access to its own persistence store.

At a minimum, individual web applications should strive to be their own bounded context, with their own persistence store for their business model, rather than sharing a database with other applications. Communication between bounded contexts occurs through programmatic interfaces, rather than through a shared database, which allows for business logic and events to take place in response to changes that take place. Bounded contexts map closely to microservices, which also are ideally implemented as their own individual bounded contexts.

Additional resources

- [Principles](#)
- [Bounded Context](#)

Common web application architectures

"If you think good architecture is expensive, try bad architecture." - Brian Foote and Joseph Yoder

Most traditional .NET applications are deployed as single units corresponding to an executable or a single web application running within a single IIS appdomain. This approach is the simplest deployment model and serves many internal and smaller public applications very well. However, even given this single unit of deployment, most non-trivial business applications benefit from some logical separation into several layers.

What is a monolithic application?

A monolithic application is one that is entirely self-contained, in terms of its behavior. It may interact with other services or data stores in the course of performing its operations, but the core of its behavior runs within its own process and the entire application is typically deployed as a single unit. If such an application needs to scale horizontally, typically the entire application is duplicated across multiple servers or virtual machines.

All-in-one applications

The smallest possible number of projects for an application architecture is one. In this architecture, the entire logic of the application is contained in a single project, compiled to a single assembly, and deployed as a single unit.

A new ASP.NET Core project, whether created in Visual Studio or from the command line, starts out as a simple "all-in-one" monolith. It contains all of the behavior of the application, including presentation, business, and data access logic. Figure 5-1 shows the file structure of a single-project app.

VS Solution Structure

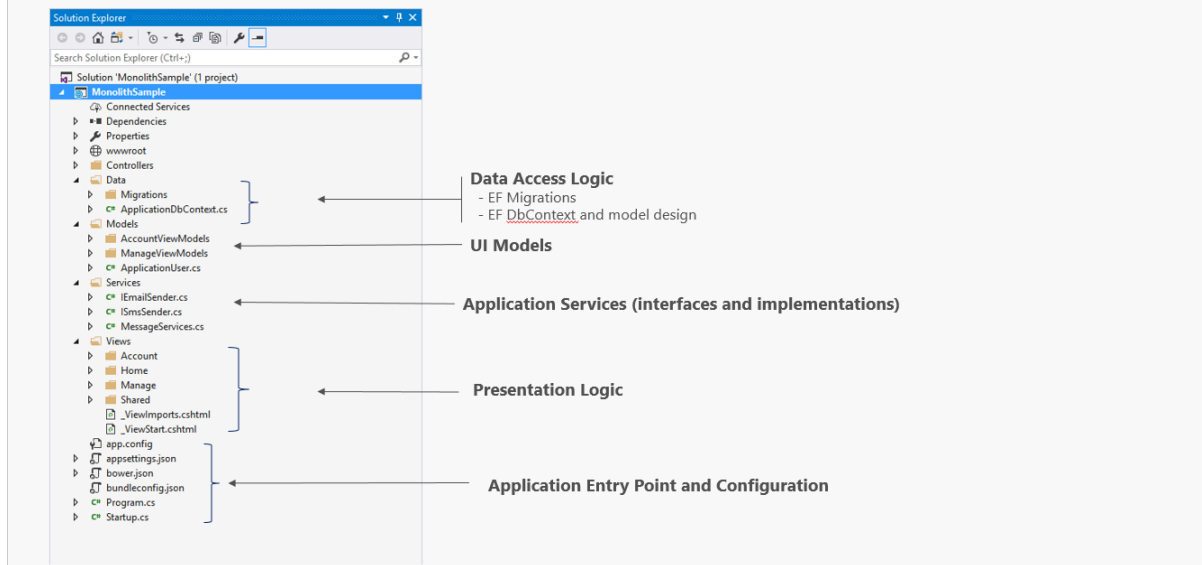


Figure 5-1. A single project ASP.NET Core app.

In a single project scenario, separation of concerns is achieved through the use of folders. The default template includes separate folders for MVC pattern responsibilities of Models, Views, and Controllers, as well as additional folders for Data and Services. In this arrangement, presentation details should be limited as much as possible to the Views folder, and data access implementation details should be limited to classes kept in the Data folder. Business logic should reside in services and classes within the Models folder.

Although simple, the single-project monolithic solution has some disadvantages. As the project's size and complexity grows, the number of files and folders will continue to grow as well. User interface (UI) concerns (models, views, controllers) reside in multiple folders, which aren't grouped together alphabetically. This issue only gets worse when additional UI-level constructs, such as Filters or ModelBinders, are added in their own folders. Business logic is scattered between the Models and Services folders, and there's no clear indication of which classes in which folders should depend on which others. This lack of organization at the project level frequently leads to [spaghetti code](#).

To address these issues, applications often evolve into multi-project solutions, where each project is considered to reside in a particular *layer* of the application.

What are layers?

As applications grow in complexity, one way to manage that complexity is to break up the application according to its responsibilities or concerns. This approach follows the separation of concerns principle and can help keep a growing codebase organized so that developers can easily find where certain functionality is implemented. Layered architecture offers a number of advantages beyond just code organization, though.

By organizing code into layers, common low-level functionality can be reused throughout the application. This reuse is beneficial because it means less code needs to be written and because it can allow the application to standardize on a single implementation, following the [don't repeat yourself \(DRY\)](#) principle.

With a layered architecture, applications can enforce restrictions on which layers can communicate with other layers. This architecture helps to achieve encapsulation. When a layer is changed or replaced, only those layers that work with it should be impacted. By limiting which layers depend on which other layers, the impact of changes can be mitigated so that a single change doesn't impact the entire application.

Layers (and encapsulation) make it much easier to replace functionality within the application. For example, an application might initially use its own SQL Server database for persistence, but later could choose to use a cloud-based persistence strategy, or one behind a web API. If the application has properly encapsulated its persistence implementation within a logical layer, that SQL Server-specific layer could be replaced by a new one implementing the same public interface.

In addition to the potential of swapping out implementations in response to future changes in requirements, application layers can also make it easier to swap out implementations for testing purposes. Instead of having to write tests that operate against the real data layer or UI layer of the application, these layers can be replaced at test time with fake implementations that provide known responses to requests. This approach typically makes tests much easier to write and much faster to run when compared to running tests against the application's real infrastructure.

Logical layering is a common technique for improving the organization of code in enterprise software applications, and there are several ways in which code can be organized into layers.

Note

Layers represent logical separation within the application. In the event that application logic is physically distributed to separate servers or processes, these separate physical deployment targets are referred to as *tiers*. It's possible, and quite common, to have an N-Layer application that is deployed to a single tier.

Traditional "N-Layer" architecture applications

The most common organization of application logic into layers is shown in Figure 5-2.

Application Layers

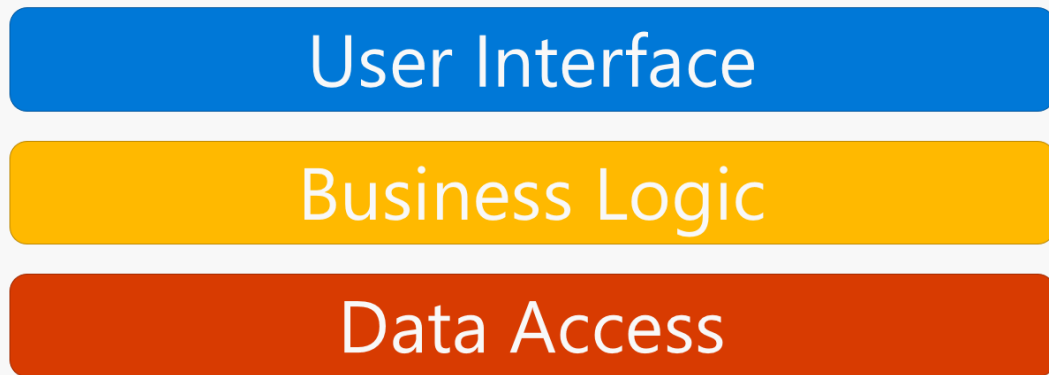


Figure 5-2. Typical application layers.

These layers are frequently abbreviated as UI, BLL (Business Logic Layer), and DAL (Data Access Layer). Using this architecture, users make requests through the UI layer, which interacts only with the BLL. The BLL, in turn, can call the DAL for data access requests. The UI layer shouldn't make any requests to the DAL directly, nor should it interact with persistence directly through other means. Likewise, the BLL should only interact with persistence by going through the DAL. In this way, each layer has its own well-known responsibility.

One disadvantage of this traditional layering approach is that compile-time dependencies run from the top to the bottom. That is, the UI layer depends on the BLL, which depends on the DAL. This means that the BLL, which usually holds the most important logic in the application, is dependent on data access implementation details (and often on the existence of a database). Testing business logic in such an architecture is often difficult, requiring a test database. The dependency inversion principle can be used to address this issue, as you'll see in the next section.

Figure 5-3 shows an example solution, breaking the application into three projects by responsibility (or layer).

VS Solution Structure

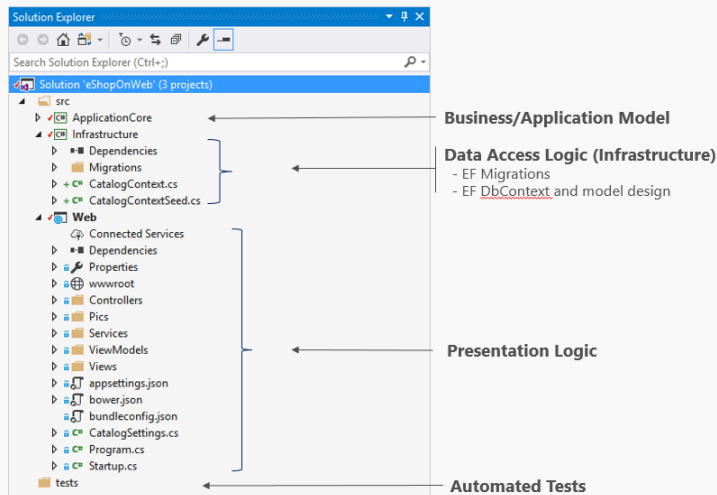


Figure 5-3. A simple monolithic application with three projects.

Although this application uses several projects for organizational purposes, it's still deployed as a single unit and its clients will interact with it as a single web app. This allows for very simple deployment process. Figure 5-4 shows how such an app might be hosted using Azure.

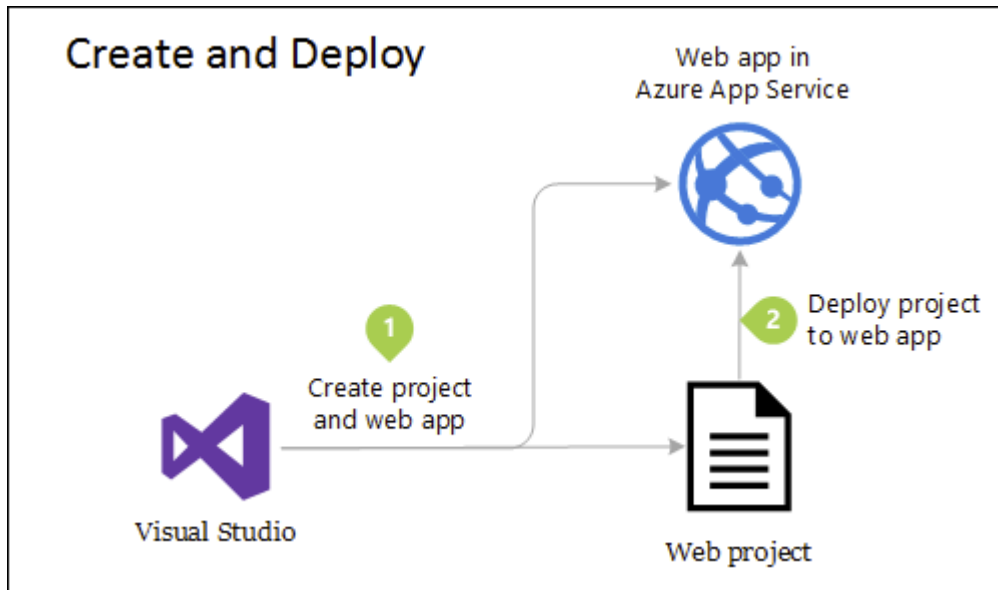


Figure 5-4. Simple deployment of Azure Web App

As application needs grow, more complex and robust deployment solutions may be required. Figure 5-5 shows an example of a more complex deployment plan that supports additional capabilities.

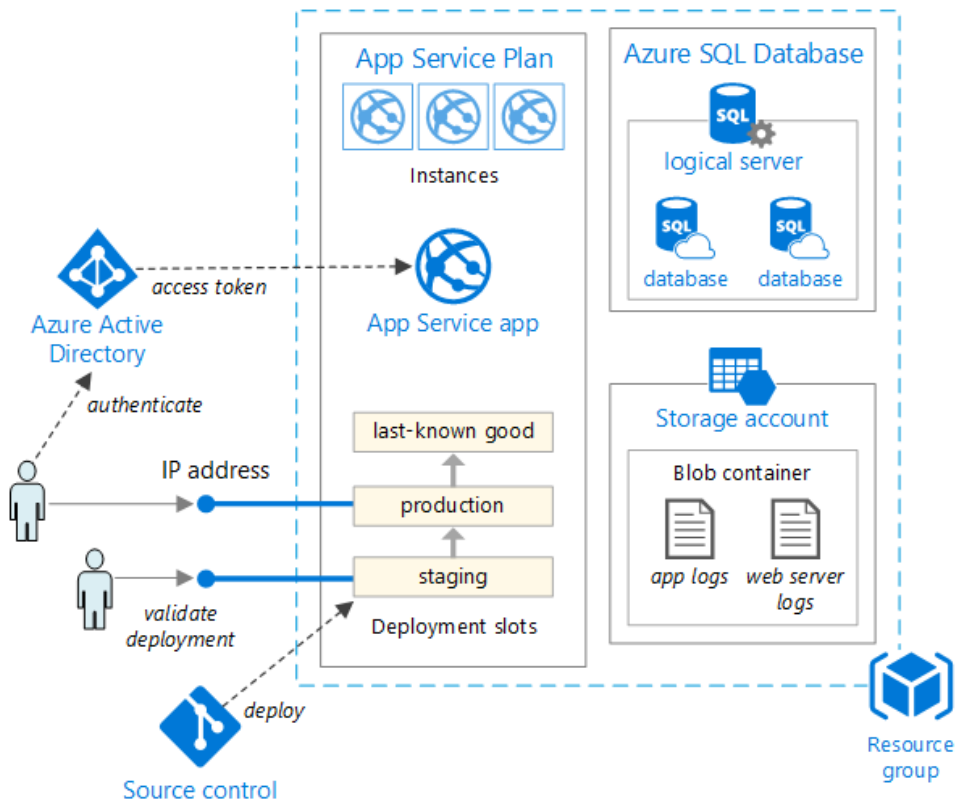


Figure 5-5. Deploying a web app to an Azure App Service

Internally, this project's organization into multiple projects based on responsibility improves the maintainability of the application.

This unit can be scaled up or out to take advantage of cloud-based on-demand scalability. Scaling up means adding additional CPU, memory, disk space, or other resources to the server(s) hosting your app. Scaling out means adding additional instances of such servers, whether these are physical servers, virtual machines, or containers. When your app is hosted across multiple instances, a load balancer is used to assign requests to individual app instances.

The simplest approach to scaling a web application in Azure is to configure scaling manually in the application's App Service Plan. Figure 5-6 shows the appropriate Azure dashboard screen to configure how many instances are serving an app.

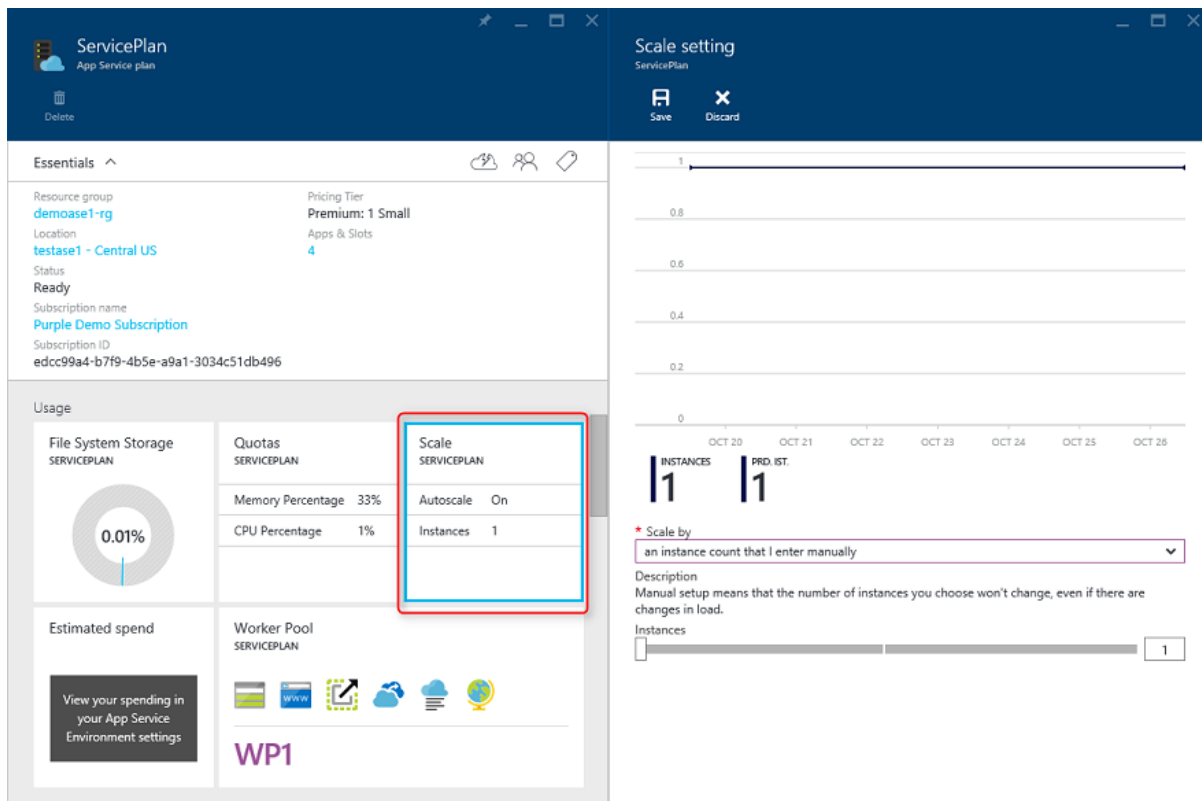


Figure 5-6. App Service Plan scaling in Azure.

Clean architecture

Applications that follow the Dependency Inversion Principle as well as the Domain-Driven Design (DDD) principles tend to arrive at a similar architecture. This architecture has gone by many names over the years. One of the first names was Hexagonal Architecture, followed by Ports-and-Adapters. More recently, it's been cited as the [Onion Architecture](#) or [Clean Architecture](#). The latter name, Clean Architecture, is used as the name for this architecture in this e-book.

The eShopOnWeb reference application uses the Clean Architecture approach in organizing its code into projects. You can find a solution template you can use as a starting point for your own ASP.NET Core solutions in the [ardalis/cleanarchitecture](#) GitHub repository or by [installing the template from NuGet](#).

Clean architecture puts the business logic and application model at the center of the application. Instead of having business logic depend on data access or other infrastructure concerns, this dependency is inverted: infrastructure and implementation details depend on the Application Core. This functionality is achieved by defining abstractions, or interfaces, in the Application Core, which are then implemented by types defined in the Infrastructure layer. A common way of visualizing this architecture is to use a series of concentric circles, similar to an onion. Figure 5-7 shows an example of this style of architectural representation.

Clean Architecture Layers (Onion view)

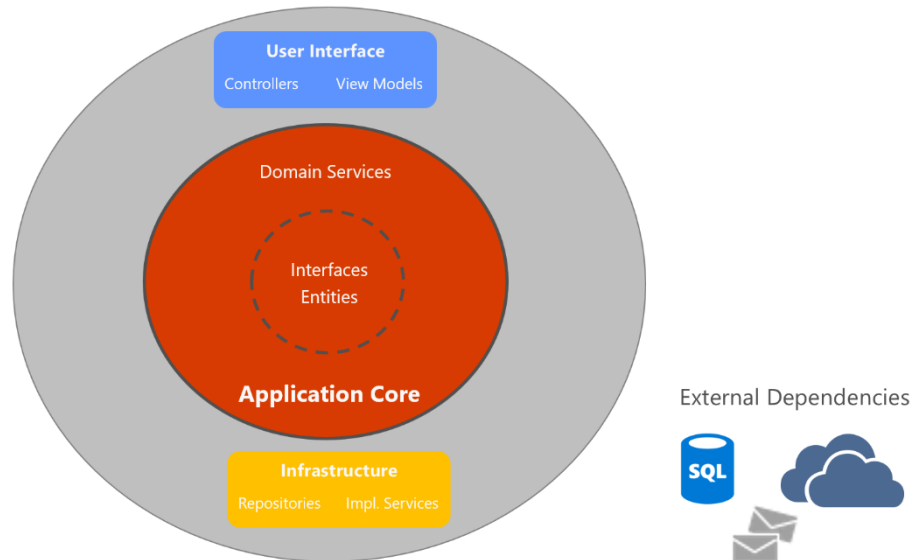


Figure 5-7. Clean Architecture; onion view

In this diagram, dependencies flow toward the innermost circle. The Application Core takes its name from its position at the core of this diagram. And you can see on the diagram that the Application Core has no dependencies on other application layers. The application's entities and interfaces are at the very center. Just outside, but still in the Application Core, are domain services, which typically implement interfaces defined in the inner circle. Outside of the Application Core, both the UI and the Infrastructure layers depend on the Application Core, but not on one another (necessarily).

Figure 5-8 shows a more traditional horizontal layer diagram that better reflects the dependency between the UI and other layers.

Clean Architecture Layers

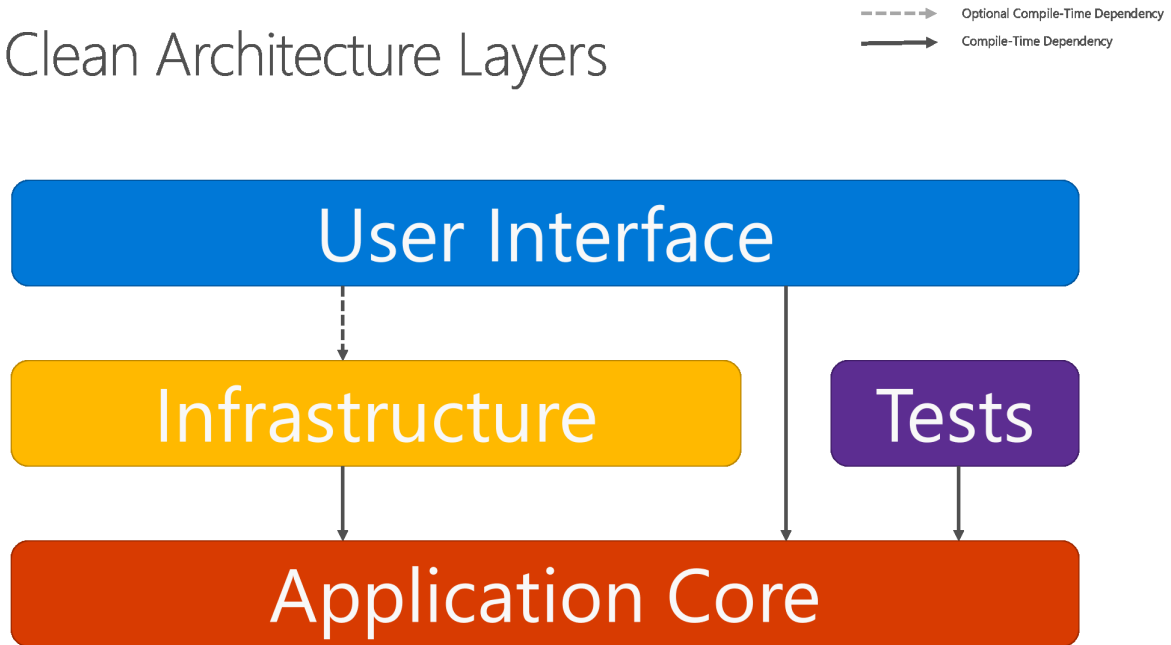


Figure 5-8. Clean Architecture; horizontal layer view

Note that the solid arrows represent compile-time dependencies, while the dashed arrow represents a runtime-only dependency. With the clean architecture, the UI layer works with interfaces defined in the Application Core at compile time, and ideally shouldn't know about the implementation types defined in the Infrastructure layer. At run time, however, these implementation types are required for the app to execute, so they need to be present and wired up to the Application Core interfaces via dependency injection.

Figure 5-9 shows a more detailed view of an ASP.NET Core application's architecture when built following these recommendations.

ASP.NET Core Architecture

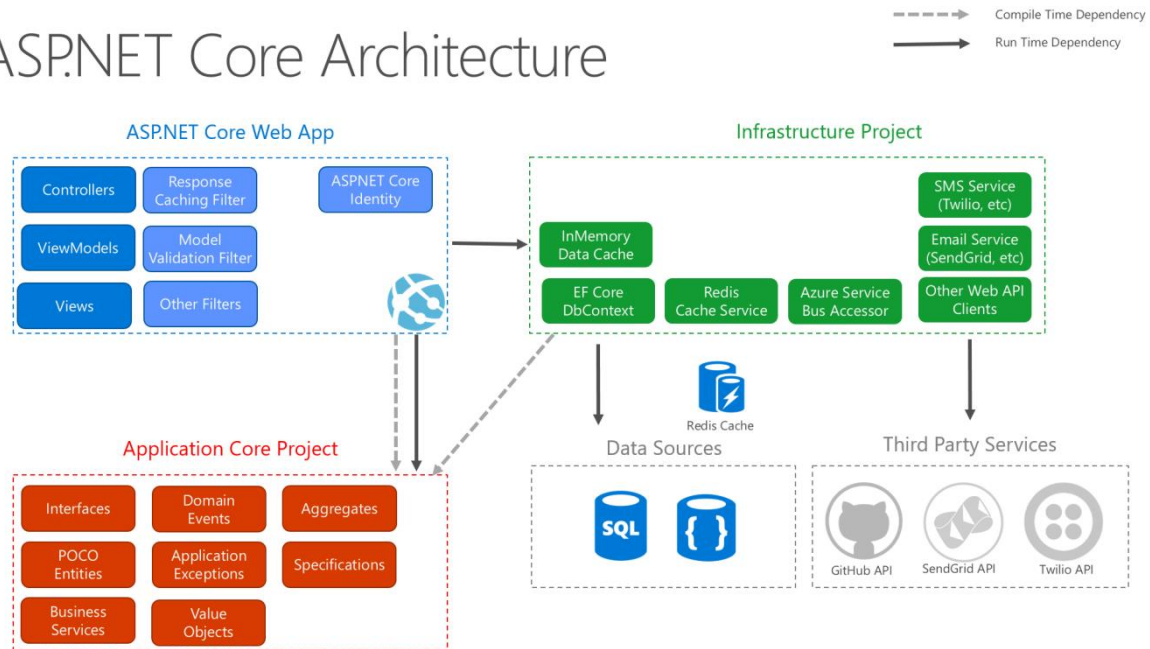


Figure 5-9. ASP.NET Core architecture diagram following Clean Architecture.

Because the Application Core doesn't depend on Infrastructure, it's very easy to write automated unit tests for this layer. Figures 5-10 and 5-11 show how tests fit into this architecture.

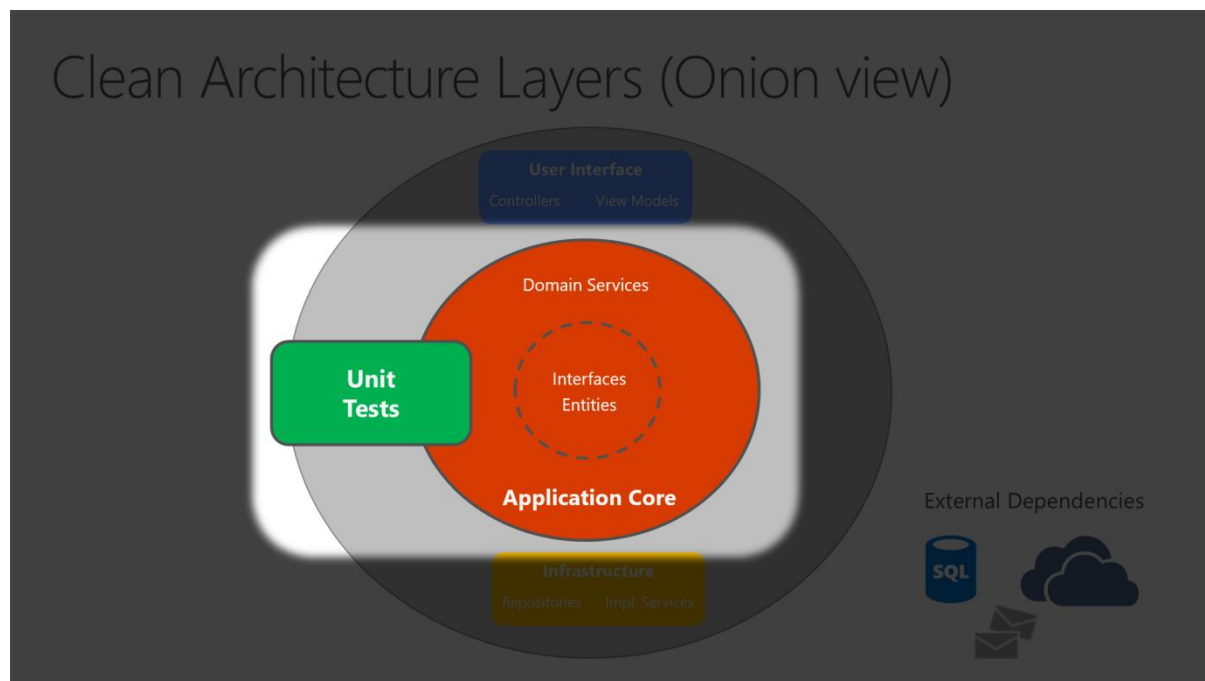


Figure 5-10. Unit testing Application Core in isolation.

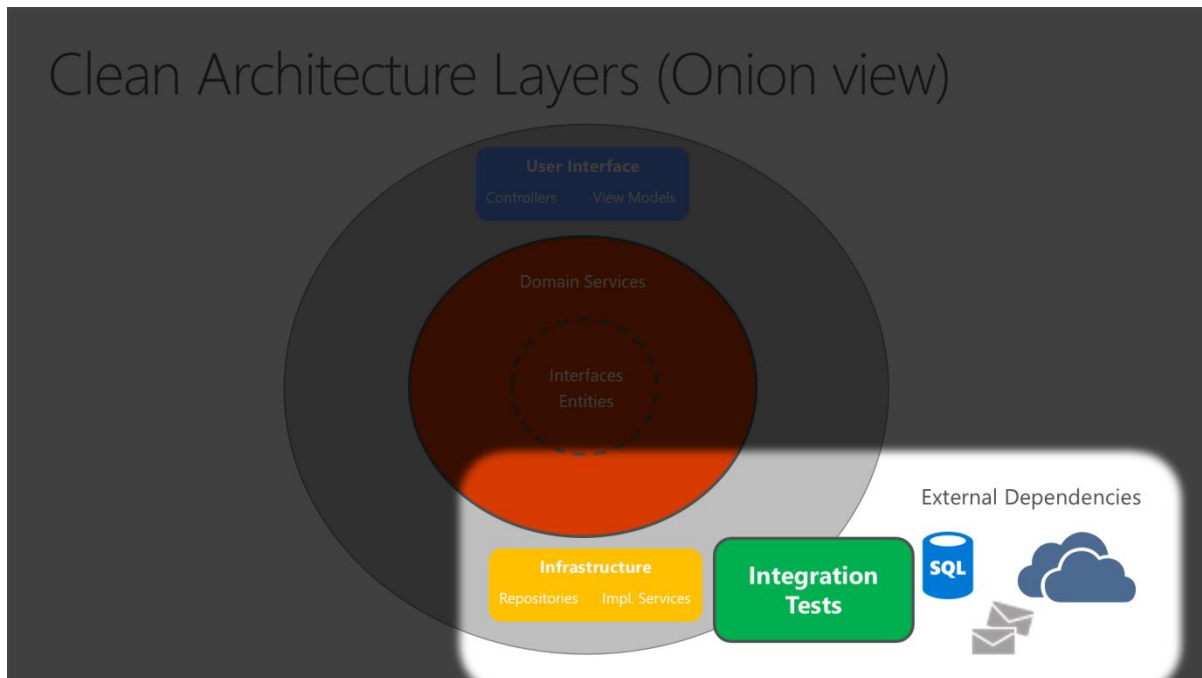


Figure 5-11. Integration testing Infrastructure implementations with external dependencies.

Since the UI layer doesn't have any direct dependency on types defined in the Infrastructure project, it's likewise very easy to swap out implementations, either to facilitate testing or in response to changing application requirements. ASP.NET Core's built-in use of and support for dependency injection makes this architecture the most appropriate way to structure non-trivial monolithic applications.

For monolithic applications, the Application Core, Infrastructure, and UI projects are all run as a single application. The runtime application architecture might look something like Figure 5-12.

ASP.NET Core Architecture

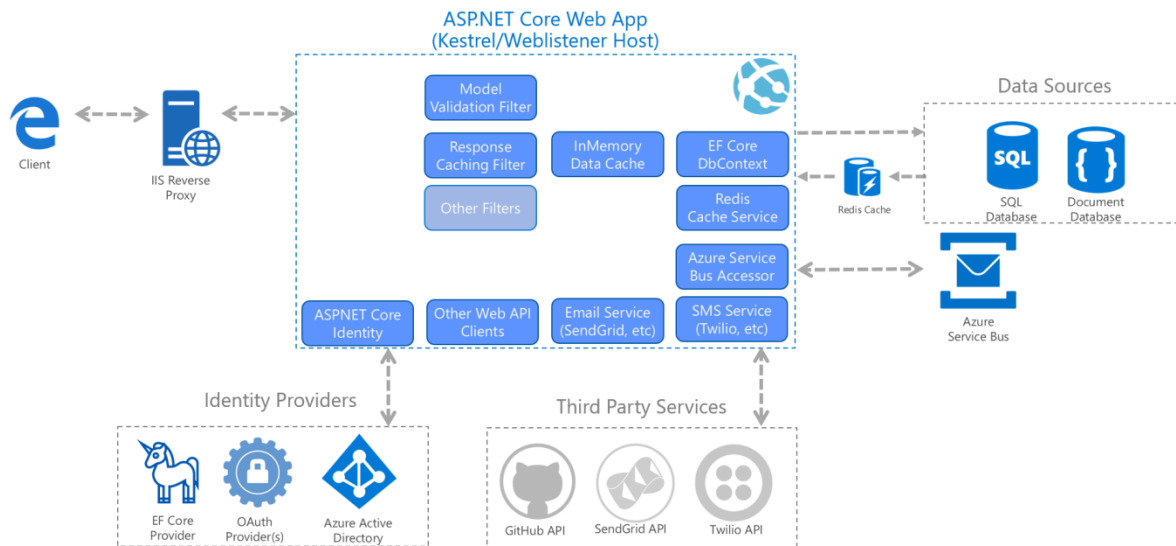


Figure 5-12. A sample ASP.NET Core app's runtime architecture.

Organizing code in Clean Architecture

In a Clean Architecture solution, each project has clear responsibilities. As such, certain types belong in each project and you'll frequently find folders corresponding to these types in the appropriate project.

Application Core

The Application Core holds the business model, which includes entities, services, and interfaces. These interfaces include abstractions for operations that will be performed using Infrastructure, such as data access, file system access, network calls, etc. Sometimes services or interfaces defined at this layer will need to work with non-entity types that have no dependencies on UI or Infrastructure. These can be defined as simple Data Transfer Objects (DTOs).

Application Core types

- Entities (business model classes that are persisted)
- Aggregates (groups of entities)
- Interfaces
- Domain Services
- Specifications
- Custom Exceptions and Guard Clauses
- Domain Events and Handlers

Infrastructure

The Infrastructure project typically includes data access implementations. In a typical ASP.NET Core web application, these implementations include the Entity Framework (EF) DbContext, any EF Core Migration objects that have been defined, and data access implementation classes. The most common way to abstract data access implementation code is through the use of the [Repository design pattern](#).

In addition to data access implementations, the Infrastructure project should contain implementations of services that must interact with infrastructure concerns. These services should implement interfaces defined in the Application Core, and so Infrastructure should have a reference to the Application Core project.

Infrastructure types

- EF Core types (DbContext, Migration)
- Data access implementation types (Repositories)
- Infrastructure-specific services (for example, FileLogger or SmtpNotifier)

UI Layer

The user interface layer in an ASP.NET Core MVC application is the entry point for the application. This project should reference the Application Core project, and its types should interact with infrastructure strictly through interfaces defined in Application Core. No direct instantiation of or static calls to the Infrastructure layer types should be allowed in the UI layer.

UI Layer types

- Controllers
- Custom Filters
- Custom Middleware
- Views
- ViewModels
- Startup

The Startup class or *Program.cs* file is responsible for configuring the application, and for wiring up implementation types to interfaces. The place where this logic is performed is known as the app's *composition root*, and is what allows dependency injection to work properly at run time.

Note

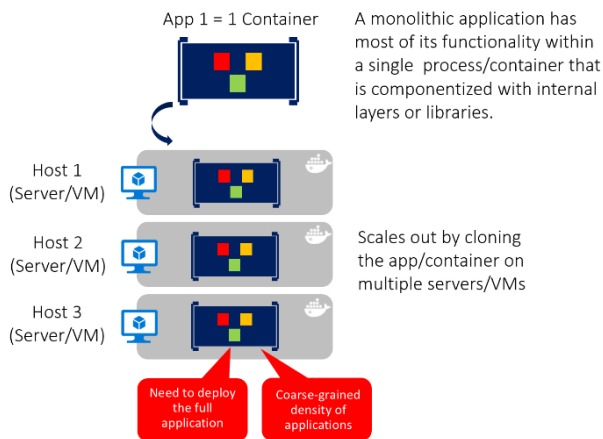
In order to wire up dependency injection during app startup, the UI layer project may need to reference the Infrastructure project. This dependency can be eliminated, most easily by using a custom DI container that has built-in support for loading types from assemblies. For the purposes of this sample, the simplest approach is to allow the UI project to reference the Infrastructure project (but developers should limit actual references to types in the Infrastructure project to the app's composition root).

Monolithic applications and containers

You can build a single and monolithic-deployment based Web Application or Service and deploy it as a container. Within the application, it might not be monolithic but organized into several libraries, components, or layers. Externally, it's a single container with a single process, single web application, or single service.

To manage this model, you deploy a single container to represent the application. To scale, just add additional copies with a load balancer in front. The simplicity comes from managing a single deployment in a single container or VM.

Monolithic Containerized application



You can include multiple components/libraries or internal layers within each container, as illustrated in Figure 5-13. But, following the container principle of “a container does one thing, and does it in one process”, the monolithic pattern might be a conflict.

The downside of this approach comes if/when the application grows, requiring it to scale. If the entire application scales, it's not really a problem. However, in most cases, a few parts of the application are the choke points requiring scaling, while other components are used less.

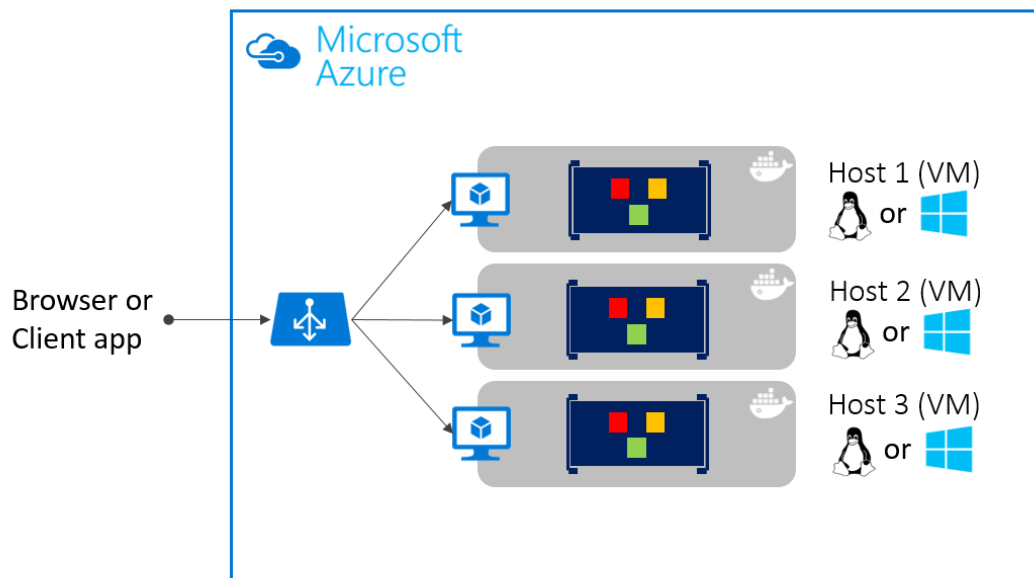
Using the typical eCommerce example, what you likely need to scale is the product information component. Many more customers browse products than purchase them. More customers use their basket than use the payment pipeline. Fewer customers add comments or view their purchase history. And you likely only have a handful of employees, in a single region, that need to manage the content and marketing campaigns. By scaling the monolithic design, all the code is deployed multiple times.

In addition to the “scale everything” problem, changes to a single component require complete retesting of the entire application, and a complete redeployment of all the instances.

The monolithic approach is common, and many organizations are developing with this architectural approach. Many are having good enough results, while others are hitting limits. Many designed their applications in this model, because the tools and infrastructure were too difficult to build service-oriented architectures (SOA), and they didn't see the need until the app grew. If you find you're hitting

the limits of the monolithic approach, breaking up the app to enable it to better leverage containers and microservices may be the next logical step.

Architecture in Docker infrastructure for monolithic applications



Deploying monolithic applications in Microsoft Azure can be achieved using dedicated VMs for each instance. Using [Azure Virtual Machine Scale Sets](#), you can easily scale the VMs. [Azure App Services](#) can run monolithic applications and easily scale instances without having to manage the VMs. Azure App Services can run single instances of Docker containers as well, simplifying the deployment. Using Docker, you can deploy a single VM as a Docker host, and run multiple instances. Using the Azure balancer, as shown in the Figure 5-14, you can manage scaling.

The deployment to the various hosts can be managed with traditional deployment techniques. The Docker hosts can be managed with commands like **docker run** performed manually, or through automation such as Continuous Delivery (CD) pipelines.

Monolithic application deployed as a container

There are benefits of using containers to manage monolithic application deployments. Scaling the instances of containers is far faster and easier than deploying additional VMs. Even when using virtual machine scale sets to scale VMs, they take time to create. When deployed as app instances, the configuration of the app is managed as part of the VM.

Deploying updates as Docker images is far faster and network efficient. Docker Images typically start in seconds, speeding rollouts. Tearing down a Docker instance is as easy as issuing a `docker stop` command, typically completing in less than a second.

As containers are inherently immutable by design, you never need to worry about corrupted VMs, whereas update scripts might forget to account for some specific configuration or file left on the disk.

You can use Docker containers for a monolithic deployment of simpler web applications. This approach improves continuous integration and continuous deployment pipelines and helps achieve deployment-to-production success. No more “It works on my machine, why does it not work in production?”

A microservices-based architecture has many benefits, but those benefits come at a cost of increased complexity. In some cases, the costs outweigh the benefits, so a monolithic deployment application running in a single container or in just a few containers is a better option.

A monolithic application might not be easily decomposable into well-separated microservices. Microservices should work independently of each other to provide a more resilient application. If you can’t deliver independent feature slices of the application, separating it only adds complexity.

An application might not yet need to scale features independently. Many applications, when they need to scale beyond a single instance, can do so through the relatively simple process of cloning that entire instance. The additional work to separate the application into discrete services provides a minimal benefit when scaling full instances of the application is simple and cost-effective.

Early in the development of an application, you might not have a clear idea where the natural functional boundaries are. As you develop a minimum viable product, the natural separation might not yet have emerged. Some of these conditions might be temporary. You might start by creating a monolithic application, and later separate some features to be developed and deployed as microservices. Other conditions might be essential to the application’s problem space, meaning that the application might never be broken into multiple microservices.

Separating an application into many discrete processes also introduces overhead. There’s more complexity in separating features into different processes. The communication protocols become more complex. Instead of method calls, you must use asynchronous communications between services. As you move to a microservices architecture, you need to add many of the building blocks implemented in the microservices version of the `eShopOnContainers` application: event bus handling, message resiliency and retries, eventual consistency, and more.

The much simpler [eShopOnWeb reference application](#) supports single-container monolithic container usage. The application includes one web application that includes traditional MVC views, web APIs, and Razor Pages. Optionally, you can run the application’s Blazor-based admin component, which requires a separate API project to run as well.

The application can be launched from the solution root using the `docker-compose build` and `docker-compose up` commands. This command configures a container for the web instance, using the Dockerfile found in the web project’s root, and runs the container on a specified port. You can download the source for this application from GitHub and run it locally. Even this monolithic application benefits from being deployed in a container environment.

For one, the containerized deployment means that every instance of the application runs in the same environment. This approach includes the developer environment where early testing and development take place. The development team can run the application in a containerized environment that matches the production environment.

In addition, containerized applications scale out at a lower cost. Using a container environment enables greater resource sharing than traditional VM environments.

Finally, containerizing the application forces a separation between the business logic and the storage server. As the application scales out, the multiple containers will all rely on a single physical storage medium. This storage medium would typically be a high-availability server running a SQL Server database.

Docker support

The eShopOnWeb project runs on .NET. Therefore, it can run in either Linux-based or Windows-based containers. Note that for Docker deployment, you want to use the same host type for SQL Server. Linux-based containers allow a smaller footprint and are preferred.

You can use Visual Studio 2017 or later to add Docker support to an existing application by right-clicking on a project in **Solution Explorer** and choosing **Add > Docker Support**. This step adds the files required and modifies the project to use them. The current eShopOnWeb sample already has these files in place.

The solution-level docker-compose.yml file contains information about what images to build and what containers to launch. The file allows you to use the docker-compose command to launch multiple applications at the same time. In this case, it is only launching the Web project. You can also use it to configure dependencies, such as a separate database container.

```
version: '3'

services:
  eshopwebmvc:
    image: eshopwebmvc
    build:
      context: .
      dockerfile: src/Web/Dockerfile
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
    ports:
      - "5106:5106"

networks:
  default:
    external:
      name: nat
```

The docker-compose.yml file references the Dockerfile in the Web project. The Dockerfile is used to specify which base container will be used and how the application will be configured on it. The Web' Dockerfile:

```
FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /app

COPY *.sln .
COPY . .
WORKDIR /app/src/Web
RUN dotnet restore
```

```
RUN dotnet publish -c Release -o out
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS runtime
WORKDIR /app
COPY --from=build /app/src/Web/out ./
ENTRYPOINT ["dotnet", "Web.dll"]
```

Troubleshooting Docker problems

Once you run the containerized application, it continues to run until you stop it. You can view which containers are running with the `docker ps` command. You can stop a running container by using the `docker stop` command and specifying the container ID.

Note that running Docker containers may be bound to ports you might otherwise try to use in your development environment. If you try to run or debug an application using the same port as a running Docker container, you'll get an error stating that the server can't bind to that port. Once again, stopping the container should resolve the issue.

If you want to add Docker support to your application using Visual Studio, make sure Docker Desktop is running when you do so. The wizard won't run correctly if Docker Desktop isn't running when you start the wizard. In addition, the wizard examines your current container choice to add the correct Docker support. If you want to add support for Windows Containers, you need to run the wizard while you have Docker Desktop running with Windows Containers configured. If you want to add support for Linux containers, run the wizard while you have Docker running with Linux containers configured.

Other web application architectural styles

- [Web-Queue-Worker](#): The core components of this architecture are a web front end that serves client requests, and a worker that performs resource-intensive tasks, long-running workflows, or batch jobs. The web front end communicates with the worker through a message queue.
- [N-tier](#): An N-tier architecture divides an application into logical layers and physical tiers.
- [Microservice](#): A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context.

References – Common web architectures

- **The Clean Architecture**
<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>
- **The Onion Architecture**
<https://jeffreypalermo.com/blog/the-onion-architecture-part-1/>
- **The Repository Pattern**
<https://deviq.com/repository-pattern/>
- **Clean Architecture Solution Template**
<https://github.com/ardalis/cleanarchitecture>

- **Architecting Microservices e-book**
<https://aka.ms/MicroservicesEbook>
- **DDD (Domain-Driven Design)**
<https://learn.microsoft.com/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/>

Common client-side web technologies

“Websites should look good from the inside and out.” - *Paul Cookson*

ASP.NET Core applications are web applications and they typically rely on client-side web technologies like HTML, CSS, and JavaScript. By separating the content of the page (the HTML) from its layout and styling (the CSS), and its behavior (via JavaScript), complex web apps can leverage the Separation of Concerns principle. Future changes to the structure, design, or behavior of the application can be made more easily when these concerns are not intertwined.

While HTML and CSS are relatively stable, JavaScript, by means of the application frameworks and utilities developers work with to build web-based applications, is evolving at breakneck speed. This chapter looks at a few ways that JavaScript is used by web developers and provides a high-level overview of the Angular and React client-side libraries.

Note

Blazor provides an alternative to JavaScript frameworks for building rich, interactive client user interfaces.

HTML

HTML is the standard markup language used to create web pages and web applications. Its elements form the building blocks of pages, representing formatted text, images, form inputs, and other structures. When a browser makes a request to a URL, whether fetching a page or an application, the first thing that is returned is an HTML document. This HTML document may reference or include additional information about its look and layout in the form of CSS, or behavior in the form of JavaScript.

CSS

CSS (Cascading Style Sheets) is used to control the look and layout of HTML elements. CSS styles can be applied directly to an HTML element, defined separately on the same page, or defined in a separate file and referenced by the page. Styles cascade based on how they are used to select a given HTML element. For instance, a style might apply to an entire document, but would be overridden by a

style that applied to a particular element. Likewise, an element-specific style would be overridden by a style that applied to a CSS class that was applied to the element, which in turn would be overridden by a style targeting a specific instance of that element (via its ID). Figure 6-1

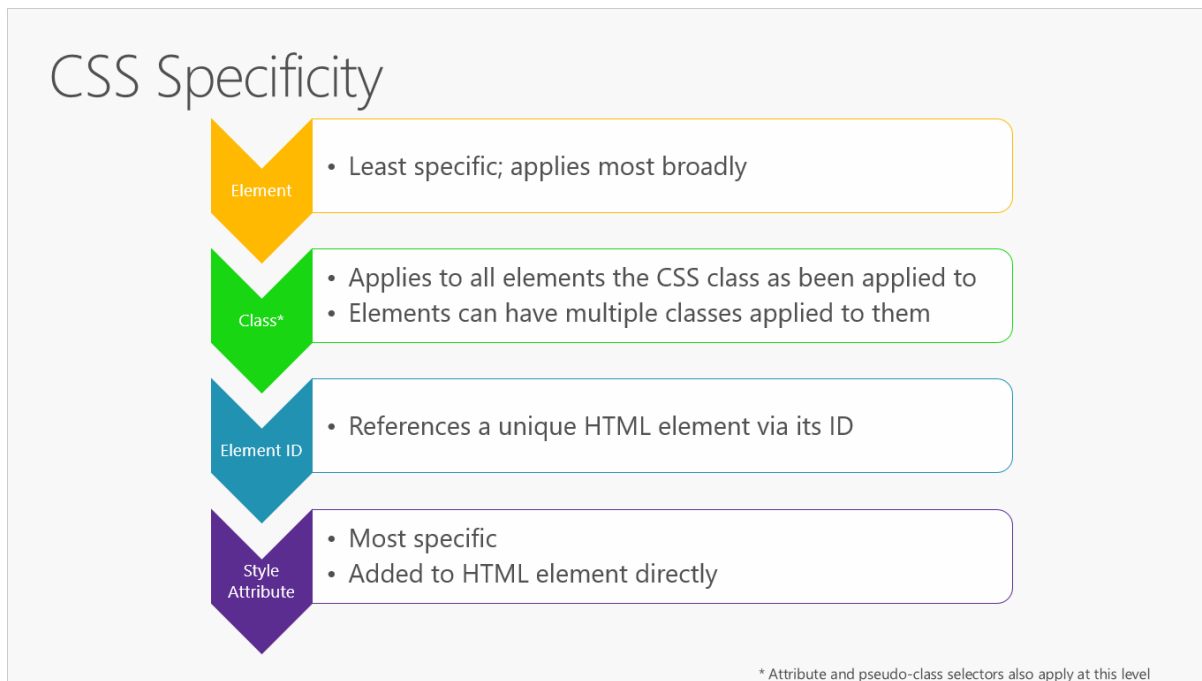


Figure 6-1. CSS Specificity rules, in order.

It's best to keep styles in their own separate stylesheet files, and to use selection-based cascading to implement consistent and reusable styles within the application. Placing style rules within HTML should be avoided, and applying styles to specific individual elements (rather than whole classes of elements, or elements that have had a particular CSS class applied to them) should be the exception, not the rule.

CSS preprocessors

CSS stylesheets lack support for conditional logic, variables, and other programming language features. Thus, large stylesheets often include quite a bit of repetition, as the same color, font, or other setting is applied to many different variations of HTML elements and CSS classes. CSS preprocessors can help your stylesheets follow the [DRY principle](#) by adding support for variables and logic.

The most popular CSS preprocessors are Sass and LESS. Both extend CSS and are backward compatible with it, meaning that a plain CSS file is a valid Sass or LESS file. Sass is Ruby-based and LESS is JavaScript based, and both typically run as part of your local development process. Both have command-line tools available, as well as built-in support in Visual Studio for running them using Gulp or Grunt tasks.

JavaScript

JavaScript is a dynamic, interpreted programming language that has been standardized in the ECMAScript language specification. It is the programming language of the web. Like CSS, JavaScript can be defined as attributes within HTML elements, as blocks of script within a page, or in separate files. Just like CSS, it's recommended to organize JavaScript into separate files, keeping it separated as much as possible from the HTML found on individual web pages or application views.

When working with JavaScript in your web application, there are a few tasks that you'll commonly need to perform:

- Selecting an HTML element and retrieving and/or updating its value.
- Querying a Web API for data.
- Sending a command to a Web API (and responding to a callback with its result).
- Performing validation.

You can perform all of these tasks with JavaScript alone, but many libraries exist to make these tasks easier. One of the first and most successful of these libraries is jQuery, which continues to be a popular choice for simplifying these tasks on web pages. For Single Page Applications (SPAs), jQuery doesn't provide many of the desired features that Angular and React offer.

Legacy web apps with jQuery

Although ancient by JavaScript framework standards, jQuery continues to be a commonly used library for working with HTML/CSS and building applications that make AJAX calls to web APIs. However, jQuery operates at the level of the browser document object model (DOM), and by default offers only an imperative, rather than declarative, model.

For example, imagine that if a textbox's value exceeds 10, an element on the page should be made visible. In jQuery, this functionality would typically be implemented by writing an event handler with code that would inspect the textbox's value and set the visibility of the target element based on that value. This process is an imperative, code-based approach. Another framework might instead use databinding to bind the visibility of the element to the value of the textbox declaratively. This approach would not require writing any code, but instead only requires decorating the elements involved with data binding attributes. As client-side behaviors grow more complex, data binding approaches frequently result in simpler solutions with less code and conditional complexity.

jQuery vs a SPA Framework

| Factor | jQuery | Angular |
|--------------------------|--------|---------|
| Abstracts the DOM | Yes | Yes |
| AJAX Support | Yes | Yes |
| Declarative Data Binding | No | Yes |
| MVC-style Routing | No | Yes |

| Factor | jQuery | Angular |
|-------------------|--------|---------|
| Templating | No | Yes |
| Deep-Link Routing | No | Yes |

Most of the features jQuery lacks intrinsically can be added with the addition of other libraries. However, a SPA framework like Angular provides these features in a more integrated fashion, since it's been designed with all of them in mind from the start. Also, jQuery is an imperative library, meaning that you need to call jQuery functions in order to do anything with jQuery. Much of the work and functionality that SPA frameworks provide can be done declaratively, requiring no actual code to be written.

Data binding is a great example of this functionality. In jQuery, it usually only takes one line of code to get the value of a DOM element or to set an element's value. However, you have to write this code anytime you need to change the value of the element, and sometimes this will occur in multiple functions on a page. Another common example is element visibility. In jQuery, there might be many different places where you'd write code to control whether certain elements were visible. In each of these cases, when using data binding, no code would need to be written. You'd simply bind the value or visibility of the elements in question to a *viewmodel* on the page, and changes to that viewmodel would automatically be reflected in the bound elements.

Angular SPAs

Angular remains one of the world's most popular JavaScript frameworks. Since Angular 2, the team rebuilt the framework from the ground up (using [TypeScript](#)) and rebranded from the original AngularJS name to Angular. Now several years old, the redesigned Angular continues to be a robust framework for building Single Page Applications.

Angular applications are built from components. Components combine HTML templates with special objects and control a portion of the page. A simple component from Angular's docs is shown here:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`
})

export class AppComponent { name = 'Angular'; }
```

Components are defined using the `@Component` decorator function, which takes in metadata about the component. The `selector` property identifies the ID of the element on the page where this component will be displayed. The `template` property is a simple HTML template that includes a placeholder that corresponds to the component's name property, defined on the last line.

By working with components and templates, instead of DOM elements, Angular apps can operate at a higher level of abstraction and with less overall code than apps written using just JavaScript (also called "vanilla JS") or with jQuery. Angular also imposes some order on how you organize your client-side script files. By convention, Angular apps use a common folder structure, with module and

component script files located in an app folder. Angular scripts concerned with building, deploying, and testing the app are typically located in a higher-level folder.

You can develop Angular apps by using a CLI. Getting started with Angular development locally (assuming you already have git and npm installed) consists of simply cloning a repo from GitHub and running npm install and npm start. Beyond this, Angular ships its own CLI, which can create projects, add files, and assist with testing, bundling, and deployment tasks. This CLI friendliness makes Angular especially compatible with ASP.NET Core, which also features great CLI support.

Microsoft has developed a reference application, eShopOnContainers, which includes an Angular SPA implementation. This app includes Angular modules to manage the online store's shopping basket, load and display items from its catalog, and handling order creation. You can view and download the sample application from [GitHub](#).

React

Unlike Angular, which offers a full Model-View-Controller pattern implementation, React is only concerned with views. It's not a framework, just a library, so to build a SPA you'll need to leverage additional libraries. There are a number of libraries that are designed to be used with React to produce rich single page applications.

One of React's most important features is its use of a virtual DOM. The virtual DOM provides React with several advantages, including performance (the virtual DOM can optimize which parts of the actual DOM need to be updated) and testability (no need to have a browser to test React and its interactions with its virtual DOM).

React is also unusual in how it works with HTML. Rather than having a strict separation between code and markup (with references to JavaScript appearing in HTML attributes perhaps), React adds HTML directly within its JavaScript code as JSX. JSX is HTML-like syntax that can compile down to pure JavaScript. For example:

```
<ul>
{ authors.map(author =>
  <li key={author.id}>{author.name}</li>
)}
</ul>
```

If you already know JavaScript, learning React should be easy. There isn't nearly as much learning curve or special syntax involved as with Angular or other popular libraries.

Because React isn't a full framework, you'll typically want other libraries to handle things like routing, web API calls, and dependency management. The nice thing is, you can pick the best library for each of these, but the disadvantage is that you need to make all of these decisions and verify all of your chosen libraries work well together when you're done. If you want a good starting point, you can use a starter kit like React Slingshot, which prepackages a set of compatible libraries together with React.

Vue

From its getting started guide, "Vue is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The

core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. On the other hand, Vue is perfectly capable of powering sophisticated Single-Page Applications when used in combination with modern tooling and supporting libraries.”

Getting started with Vue simply requires including its script within an HTML file:

```
<!-- development version, includes helpful console warnings -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
With the framework added, you're then able to declaratively render data to the DOM using
Vue's straightforward templating syntax:
<div id="app">
  {{ message }}
</div>
and then adding the following script:
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

This is enough to render "Hello Vue!" on the page. Note, however, that Vue isn't simply rendering the message to the div once. It supports databinding and dynamic updates such that if the value of message changes, the value in the <div> is immediately updated to reflect it.

Of course, this only scratches the surface of what Vue is capable of. It's gained a great deal of popularity in the last several years and has a large community. There's a [huge and growing list of supporting components and libraries](#) that work with Vue to extend it as well. If you're looking to add client-side behavior to your web application or considering building a full SPA, Vue is worth investigating.

Blazor WebAssembly

Unlike other JavaScript frameworks, Blazor WebAssembly is a single-page app (SPA) framework for building interactive client-side web apps with .NET. Blazor WebAssembly uses open web standards without plugins or recompiling code into other languages. Blazor WebAssembly works in all modern web browsers, including mobile browsers.

Running .NET code inside web browsers is made possible by WebAssembly (abbreviated wasm). WebAssembly is a compact bytecode format optimized for fast download and maximum execution speed. WebAssembly is an open web standard and is supported in web browsers without plugins.

WebAssembly code can access the full functionality of the browser via JavaScript, called JavaScript interoperability, often shortened to JavaScript interop or JS interop. .NET code executed via WebAssembly in the browser runs in the browser's JavaScript sandbox with the protections that the sandbox provides against malicious actions on the client machine.

For more information, see [Introduction to ASP.NET Core Blazor](#).

Choosing a SPA Framework

When considering which option will work best to support your SPA, keep in mind the following considerations:

- Is your team familiar with the framework and its dependencies (including TypeScript in some cases)?
- How opinionated is the framework, and do you agree with its default way of doing things?
- Does it (or a companion library) include all of the features your app requires?
- Is it well documented?
- How active is its community? Are new projects being built with it?
- How active is its core team? Are issues being resolved and new versions shipped regularly?

Frameworks continue to evolve with breakneck speed. Use the considerations listed above to help mitigate the risk of choosing a framework you'll later regret being dependent upon. If you're particularly risk-averse, consider a framework that offers commercial support and/or is being developed by a large enterprise.

References – Client Web Technologies

- **HTML and CSS**
<https://www.w3.org/standards/webdesign/htmlcss>
- **Sass vs. LESS**
<https://www.keycdn.com/blog/sass-vs-less/>
- **Styling ASP.NET Core Apps with LESS, Sass, and Font Awesome**
<https://learn.microsoft.com/aspnet/core/client-side/less-sass-fa>
- **Client-Side Development in ASP.NET Core**
<https://learn.microsoft.com/aspnet/core/client-side/>
- **jQuery**
<https://jquery.com/>
- **Angular**
<https://angular.io/>
- **React**
<https://reactjs.org/>
- **Vue**
<https://vuejs.org/>
- **Angular vs React vs Vue: Which Framework to Choose in 2020**
<https://www.codeinwp.com/blog/angular-vs-vue-vs-react/>
- **The Top JavaScript Frameworks for Front-End Development in 2020**
<https://www.freecodecamp.org/news/complete-guide-for-front-end-developers-javascript-frameworks-2019/>

Develop ASP.NET Core MVC apps

"It's not important to get it right the first time. It's vitally important to get it right the last time." - Andrew Hunt and David Thomas

ASP.NET Core is a cross-platform, open-source framework for building modern cloud-optimized web applications. ASP.NET Core apps are lightweight and modular, with built-in support for dependency injection, enabling greater testability and maintainability. Combined with MVC, which supports building modern web APIs in addition to view-based apps, ASP.NET Core is a powerful framework with which to build enterprise web applications.

MVC and Razor Pages

ASP.NET Core MVC offers many features that are useful for building web-based APIs and apps. The term MVC stands for "Model-View-Controller", a UI pattern that breaks up the responsibilities of responding to user requests into several parts. In addition to following this pattern, you can also implement features in your ASP.NET Core apps as Razor Pages.

Razor Pages are built into ASP.NET Core MVC, and use the same features for routing, model binding, filters, authorization, etc. However, instead of having separate folders and files for Controllers, Models, Views, etc. and using attribute-based routing, Razor Pages are placed in a single folder ("/Pages"), route based on their relative location in this folder, and handle requests with handlers instead of controller actions. As a result, when working with Razor Pages, all of the files and classes you need are typically colocated, not spread throughout the web project.

Learn more about [how MVC, Razor Pages, and related patterns are applied in the eShopOnWeb sample application](#).

When you create a new ASP.NET Core App, you should have a plan in mind for the kind of app you want to build. When creating a new project, in your IDE or using the dotnet new CLI command, you will choose from several templates. The most common project templates are Empty, Web API, Web App, and Web App (Model-View-Controller). Although you can only make this decision when you first create a project, it's not an irrevocable decision. The Web API project uses standard Model-View-Controller controllers – it just lacks Views by default. Likewise, the default Web App template uses Razor Pages, and so also lacks a Views folder. You can add a Views folder to these projects later to support view-based behavior. Web API and Model-View-Controller projects don't include a Pages

folder by default, but you can add one later to support Razor Pages-based behavior. You can think of these three templates as supporting three different kinds of default user interaction: data (web API), page-based, and view-based. However, you can mix and match any or all of these templates within a single project if you wish.

Why Razor Pages?

Razor Pages is the default approach for new web applications in Visual Studio. Razor Pages offers a simpler way of building page-based application features, such as non-SPA forms. Using controllers and views, it was common for applications to have very large controllers that worked with many different dependencies and view models and returned many different views. This resulted in more complexity and often resulted in controllers that didn't follow the Single Responsibility Principle or Open/Closed Principles effectively. Razor Pages addresses this issue by encapsulating the server-side logic for a given logical "page" in a web application with its Razor markup. A Razor Page that has no server-side logic can only consist of a Razor file (for instance, "Index.cshtml"). However, most non-trivial Razor Pages will have an associated page model class, which by convention is named the same as the Razor file with a ".cs" extension (for example, "Index.cshtml.cs").

A Razor Page's page model combines the responsibilities of an MVC controller and a viewmodel. Instead of handling requests with controller action methods, page model handlers like "OnGet()" are executed, rendering their associated page by default. Razor Pages simplifies the process of building individual pages in an ASP.NET Core app, while still providing all the architectural features of ASP.NET Core MVC. They're a good default choice for new page-based functionality.

When to use MVC

If you're building web APIs, the MVC pattern makes more sense than trying to use Razor Pages. If your project will only expose web API endpoints, you should ideally start from the Web API project template. Otherwise, it's easy to add controllers and associated API endpoints to any ASP.NET Core app. Use the view-based MVC approach if you're migrating an existing application from ASP.NET MVC 5 or earlier to ASP.NET Core MVC and you want to do so with the least amount of effort. Once you've made the initial migration, you can evaluate whether it makes sense to adopt Razor Pages for new features or even as a wholesale migration. For more information about porting .NET 4.x apps to .NET 7, see [Porting Existing ASP.NET Apps to ASP.NET Core eBook](#).

Whether you choose to build your web app using Razor Pages or MVC views, your app will have similar performance and will include support for dependency injection, filters, model binding, validation, and so on.

Mapping requests to responses

At its heart, ASP.NET Core apps map incoming requests to outgoing responses. At a low level, this mapping is done with middleware, and simple ASP.NET Core apps and microservices may be comprised solely of custom middleware. When using ASP.NET Core MVC, you can work at a somewhat higher level, thinking in terms of *routes*, *controllers*, and *actions*. Each incoming request is compared with the application's routing table, and if a matching route is found, the associated action

method (belonging to a controller) is called to handle the request. If no matching route is found, an error handler (in this case, returning a `NotFound` result) is called.

ASP.NET Core MVC apps can use conventional routes, attribute routes, or both. Conventional routes are defined in code, specifying routing *conventions* using syntax like in the example below:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "default", pattern:
        "{controller=Home}/{action=Index}/{id?}");
});
```

In this example, a route named “default” has been added to the routing table. It defines a route template with placeholders for controller, action, and id. The controller and action placeholders have the default specified (Home and Index, respectively), and the id placeholder is optional (by virtue of a “?” applied to it). The convention defined here states that the first part of a request should correspond to the name of the controller, the second part to the action, and then if necessary a third part will represent an ID parameter. Conventional routes are typically defined in one place for the application, such as in *Program.cs* where the request middleware pipeline is configured.

Attribute routes are applied to controllers and actions directly, rather than specified globally. This approach has the advantage of making them much more discoverable when you’re looking at a particular method, but does mean that routing information is not kept in one place in the application. With attribute routes, you can easily specify multiple routes for a given action, as well as combine routes between controllers and actions. For example:

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("")] // Combines to define the route template "Home"
    [Route("Index")] // Combines to define route template "Home/Index"
    [Route("/")] // Does not combine, defines the route template ""
    public IActionResult Index() {}
}
```

Routes can be specified on `[HttpGet]` and similar attributes, avoiding the need to add separate `[Route]` attributes. Attribute routes can also use tokens to reduce the need to repeat controller or action names, as shown below:

```
[Route("[controller]")]
public class ProductsController : Controller
{
    [Route("")] // Matches 'Products'
    [Route("Index")] // Matches 'Products/Index'
    public IActionResult Index() {}
}
```

Razor Pages don’t use attribute routing. You can specify additional route template information for a Razor Page as part of its `@page` directive:

```
@page "{id:int}"
```

In the previous example, the page in question would match a route with an integer id parameter. For example, the *Products.cshtml* page located in the root of */Pages* would respond to requests like this one:

`/Products/123`

Once a given request has been matched to a route, but before the action method is called, ASP.NET Core MVC will perform [model binding](#) and [model validation](#) on the request. Model binding is responsible for converting incoming HTTP data into the .NET types specified as parameters of the action method to be called. For example, if the action method expects an int id parameter, model binding will attempt to provide this parameter from a value provided as part of the request. To do so, model binding looks for values in a posted form, values in the route itself, and query string values. Assuming an id value is found, it will be converted to an integer before being passed into the action method.

After binding the model but before calling the action method, model validation occurs. Model validation uses optional attributes on the model type, and can help ensure that the provided model object conforms to certain data requirements. Certain values may be specified as required, or limited to a certain length or numeric range, etc. If validation attributes are specified but the model does not conform to their requirements, the property `ModelState.IsValid` will be false, and the set of failing validation rules will be available to send to the client making the request.

If you're using model validation, you should be sure to always check that the model is valid before performing any state-altering commands, to ensure your app is not corrupted by invalid data. You can use a [filter](#) to avoid the need to add code for this validation in every action. ASP.NET Core MVC filters offer a way of intercepting groups of requests, so that common policies and cross-cutting concerns can be applied on a targeted basis. Filters can be applied to individual actions, whole controllers, or globally for an application.

For web APIs, ASP.NET Core MVC supports [content negotiation](#), allowing requests to specify how responses should be formatted. Based on headers provided in the request, actions returning data will format the response in XML, JSON, or another supported format. This feature enables the same API to be used by multiple clients with different data format requirements.

Web API projects should consider using the `[ApiController]` attribute, which can be applied to individual controllers, to a base controller class, or to the entire assembly. This attribute adds automatic model validation checking and any action with an invalid model will return a `BadRequest` with the details of the validation errors. The attribute also requires all actions have an attribute route, rather than using a conventional route, and returns more detailed `ProblemDetails` information in response to errors.

Keeping controllers under control

For page-based applications, Razor Pages do a great job of keeping controllers from getting too large. Each individual page is given its own files and classes dedicated just to its handler(s). Prior to the introduction of Razor Pages, many view-centric applications would have large controller classes responsible for many different actions and views. These classes would naturally grow to have many responsibilities and dependencies, making them harder to maintain. If you find your view-based

controllers are growing too large, consider refactoring them to use Razor Pages, or introducing a pattern like a mediator.

The mediator design pattern is used to reduce coupling between classes while allowing communication between them. In ASP.NET Core MVC applications, this pattern is frequently employed to break up controllers into smaller pieces by using *handlers* to do the work of action methods. The popular [MediatR NuGet package](#) is often used to accomplish this. Typically, controllers include many different action methods, each of which may require certain dependencies. The set of all dependencies required by any action must be passed into the controller's constructor. When using MediatR, the only dependency a controller will typically have is an instance of the mediator. Each action then uses the mediator instance to send a message, which is processed by a handler. The handler is specific to a single action and thus only needs the dependencies required by that action. An example of a controller using MediatR is shown here:

```
public class OrderController : Controller
{
    private readonly IMediator _mediator;

    public OrderController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpGet]
    public async Task<IActionResult> MyOrders()
    {
        var viewModel = await _mediator.Send(new GetMyOrders(User.Identity.Name));
        return View(viewModel);
    }
    // other actions implemented similarly
}
```

In the MyOrders action, the call to Send a GetMyOrders message is handled by this class:

```
public class GetMyOrdersHandler : IRequestHandler<GetMyOrders, IEnumerable<OrderViewModel>>
{
    private readonly IOrderRepository _orderRepository;
    public GetMyOrdersHandler(IOrderRepository orderRepository)
    {
        _orderRepository = orderRepository;
    }

    public async Task<IEnumerable<OrderViewModel>> Handle(GetMyOrders request,
        CancellationToken cancellationToken)
    {
        var specification = new CustomerOrdersWithItemsSpecification(request.UserName);
        var orders = await _orderRepository.ListAsync(specification);
        return orders.Select(o => new OrderViewModel
        {
            OrderDate = o.OrderDate,
            OrderItems = o.OrderItems?.Select(oi => new OrderItemViewModel()
            {
                PictureUrl = oi.ItemOrdered.PictureUri,
                ProductId = oi.ItemOrdered.CatalogItemId,
                ProductName = oi.ItemOrdered.ProductName,
                UnitPrice = oi.UnitPrice,
                Units = oi.Units
            })
        });
    }
}
```



```

        }).ToList(),
        OrderNumber = o.Id,
        ShippingAddress = o.ShipToAddress,
        Total = o.Total()
    });
}
}

```

The end result of this approach is for controllers to be much smaller and focused primarily on routing and model binding, while individual handlers are responsible for the specific tasks needed by a given endpoint. This approach can also be achieved without MediatR by using the [ApiEndpoints NuGet package](#), which attempts to bring to API controllers the same benefits Razor Pages brings to view-based controllers.

References – Mapping Requests to Responses

- **Routing to Controller Actions**
<https://learn.microsoft.com/aspnet/core/mvc/controllers/routing>
- **Model Binding**
<https://learn.microsoft.com/aspnet/core/mvc/models/model-binding>
- **Model Validation**
<https://learn.microsoft.com/aspnet/core/mvc/models/validation>
- **Filters**
<https://learn.microsoft.com/aspnet/core/mvc/controllers/filters>
- **ApiController Attribute**
<https://learn.microsoft.com/aspnet/core/web-api/>

Working with dependencies

ASP.NET Core has built-in support for and internally makes use of a technique known as [dependency injection](#). Dependency injection is a technique that enables loose coupling between different parts of an application. Looser coupling is desirable because it makes it easier to isolate parts of the application, allowing for testing or replacement. It also makes it less likely that a change in one part of the application will have an unexpected impact somewhere else in the application. Dependency injection is based on the dependency inversion principle, and is often key to achieving the open/closed principle. When evaluating how your application works with its dependencies, beware of the [static cling](#) code smell, and remember the aphorism “[new is glue](#).”

Static cling occurs when your classes make calls to static methods, or access static properties, which have side effects or dependencies on infrastructure. For example, if you have a method that calls a static method, which in turn writes to a database, your method is tightly coupled to the database. Anything that breaks that database call will break your method. Testing such methods is notoriously difficult, since such tests either require commercial mocking libraries to mock the static calls, or can only be tested with a test database in place. Static calls that don’t have any dependence on infrastructure, especially those calls that are completely stateless, are fine to call and have no impact on coupling or testability (beyond coupling code to the static call itself).

Many developers understand the risks of static cling and global state, but will still tightly couple their code to specific implementations through direct instantiation. “New is glue” is meant to be a reminder of this coupling, and not a general condemnation of the use of the new keyword. Just as with static method calls, new instances of types that have no external dependencies typically do not tightly couple code to implementation details or make testing more difficult. But each time a class is instantiated, take just a brief moment to consider whether it makes sense to hard-code that specific instance in that particular location, or if it would be a better design to request that instance as a dependency.

Declare your dependencies

ASP.NET Core is built around having methods and classes declare their dependencies, requesting them as arguments. ASP.NET applications are typically set up in *Program.cs* or in a Startup class.

Note

Configuring apps completely in *Program.cs* is the default approach for .NET 6 (and later) and Visual Studio 2022 apps. Project templates have been updated to help you get started with this new approach. ASP.NET Core projects can still use a Startup class, if desired.

Configure services in *Program.cs*

For very simple apps, you can wire up dependencies directly in *Program.cs* file using a *WebApplicationBuilder*. Once all needed services have been added, the builder is used to create the app.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();
```

Configure services in *Startup.cs*

The *Startup.cs* is itself configured to support dependency injection at several points. If you’re using a Startup class, you can give it a constructor and it can request dependencies through it, like so:

```
public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(env.ContentRootPath)
            .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);
    }
}
```

The Startup class is interesting in that there are no explicit type requirements for it. It doesn’t inherit from a special Startup base class, nor does it implement any particular interface. You can give it a constructor, or not, and you can specify as many parameters on the constructor as you want. When

the web host you've configured for your application starts, it will call the Startup class (if you've told it to use one), and will use dependency injection to populate any dependencies the Startup class requires. Of course, if you request parameters that aren't configured in the services container used by ASP.NET Core, you'll get an exception, but as long as you stick to dependencies the container knows about, you can request anything you want.

Dependency injection is built into your ASP.NET Core apps right from the start, when you create the Startup instance. It doesn't stop there for the Startup class. You can also request dependencies in the Configure method:

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

The ConfigureServices method is the exception to this behavior; it must take just one parameter of type IServiceCollection. It doesn't really need to support dependency injection, since on the one hand it is responsible for adding objects to the services container, and on the other it has access to all currently configured services via the IServiceCollection parameter. Thus, you can work with dependencies defined in the ASP.NET Core services collection in every part of the Startup class, either by requesting the needed service as a parameter or by working with the IServiceCollection in ConfigureServices.

Note

If you need to ensure certain services are available to your Startup class, you can configure them using an IWebHostBuilder and its ConfigureServices method inside the CreateDefaultBuilder call.

The Startup class is a model for how you should structure other parts of your ASP.NET Core application, from Controllers to Middleware to Filters to your own Services. In each case, you should follow the [Explicit Dependencies Principle](#), requesting your dependencies rather than directly creating them, and leveraging dependency injection throughout your application. Be careful of where and how you directly instantiate implementations, especially services and objects that work with infrastructure or have side effects. Prefer working with abstractions defined in your application core and passed in as arguments to hardcoding references to specific implementation types.

Structuring the application

Monolithic applications typically have a single entry point. In the case of an ASP.NET Core web application, the entry point will be the ASP.NET Core web project. However, that doesn't mean the solution should consist of just a single project. It's useful to break up the application into different layers in order to follow separation of concerns. Once broken up into layers, it's helpful to go beyond folders to separate projects, which can help achieve better encapsulation. The best approach to achieve these goals with an ASP.NET Core application is a variation of the Clean Architecture discussed in chapter 5. Following this approach, the application's solution will comprise separate libraries for the UI, Infrastructure, and ApplicationCore.

In addition to these projects, separate test projects are included as well (Testing is discussed in Chapter 9).

The application's object model and interfaces should be placed in the `ApplicationCore` project. This project will have as few dependencies as possible (and none on specific infrastructure concerns), and the other projects in the solution will reference it. Business entities that need to be persisted are defined in the `ApplicationCore` project, as are services that do not directly depend on infrastructure.

Implementation details, such as how persistence is performed or how notifications might be sent to a user, are kept in the `Infrastructure` project. This project will reference implementation-specific packages such as Entity Framework Core, but should not expose details about these implementations outside of the project. Infrastructure services and repositories should implement interfaces that are defined in the `ApplicationCore` project, and its persistence implementations are responsible for retrieving and storing entities defined in `ApplicationCore`.

The ASP.NET Core UI project is responsible for any UI level concerns, but should not include business logic or infrastructure details. In fact, ideally it shouldn't even have a dependency on the `Infrastructure` project, which will help ensure no dependency between the two projects is introduced accidentally. This can be achieved using a third-party DI container like Autofac, which allows you to define DI rules in Module classes in each project.

Another approach to decoupling the application from implementation details is to have the application call microservices, perhaps deployed in individual Docker containers. This provides even greater separation of concerns and decoupling than leveraging DI between two projects, but has additional complexity.

Feature organization

By default, ASP.NET Core applications organize their folder structure to include Controllers and Views, and frequently ViewModels. Client-side code to support these server-side structures is typically stored separately in the `wwwroot` folder. However, large applications may encounter problems with this organization, since working on any given feature often requires jumping between these folders. This gets more and more difficult as the number of files and subfolders in each folder grows, resulting in a great deal of scrolling through Solution Explorer. One solution to this problem is to organize application code by *feature* instead of by file type. This organizational style is typically referred to as feature folders or [feature slices](#) (see also: [Vertical Slices](#)).

ASP.NET Core MVC supports Areas for this purpose. Using areas, you can create separate sets of Controllers and Views folders (as well as any associated models) in each Area folder. Figure 7-1 shows an example folder structure, using Areas.

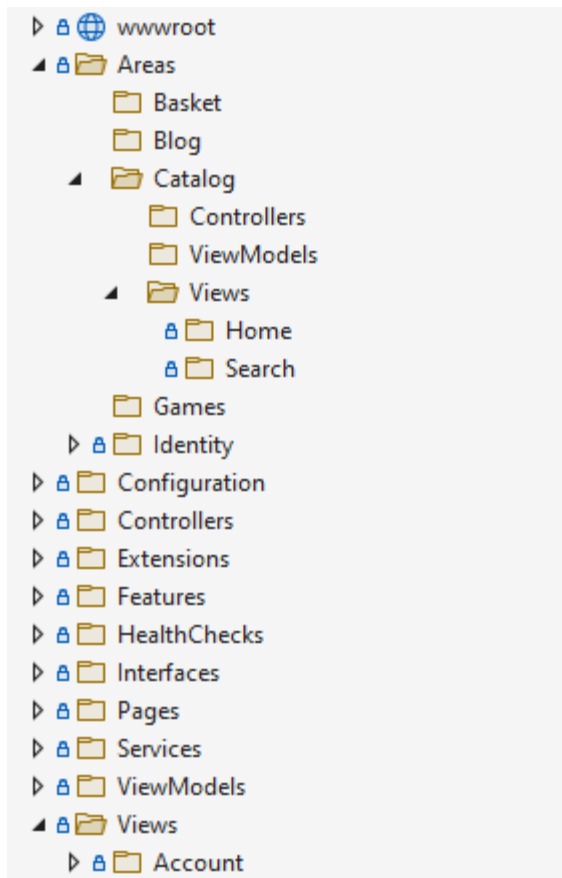


Figure 7-1. Sample Area Organization

When using Areas, you must use attributes to decorate your controllers with the name of the area to which they belong:

```
[Area("Catalog")]
public class HomeController
{
}
You also need to add area support to your routes:
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "areaRoute", pattern:
"{area:exists}/{controller=Home}/{action=Index}/{id?}");
    endpoints.MapControllerRoute(name: "default", pattern:
"{controller=Home}/{action=Index}/{id?}");
});
```

In addition to the built-in support for Areas, you can also use your own folder structure, and conventions in place of attributes and custom routes. This would allow you to have feature folders that didn't include separate folders for Views, Controllers, etc., keeping the hierarchy flatter and making it easier to see all related files in a single place for each feature. For APIs, folders can be used to replace controllers, and each folder can contain all of the API Endpoints and their associated DTOs.

ASP.NET Core uses built-in convention types to control its behavior. You can modify or replace these conventions. For example, you can create a convention that will automatically get the feature name

for a given controller based on its namespace (which typically correlates to the folder in which the controller is located):

```
public class FeatureConvention : IControllerModelConvention
{
    public void Apply(ControllerModel controller)
    {
        controller.Properties.Add("feature",
            GetFeatureName(controller.ControllerType));
    }

    private string GetFeatureName(TypeInfo controllerType)
    {
        string[] tokens = controllerType.FullName.Split('.');
        if (!tokens.Any(t => t == "Features")) return "";
        string featureName = tokens
            .SkipWhile(t => !t.Equals("features",
                StringComparison.CurrentCultureIgnoreCase))
            .Skip(1)
            .Take(1)
            .FirstOrDefault();
        return featureName;
    }
}
```

You then specify this convention as an option when you add support for MVC to your application in `ConfigureServices` (or in `Program.cs`):

```
// ConfigureServices
services.AddMvc(o => o.Conventions.Add(new FeatureConvention()));

// Program.cs
builder.Services.AddMvc(o => o.Conventions.Add(new FeatureConvention()));
```

ASP.NET Core MVC also uses a convention to locate views. You can override it with a custom convention so that views will be located in your feature folders (using the feature name provided by the `FeatureConvention`, above). You can learn more about this approach and download a working sample from the MSDN Magazine article, [Feature Slices for ASP.NET Core MVC](#).

APIs and Blazor applications

If your application includes a set of web APIs, which must be secured, these APIs should ideally be configured as a separate project from your View or Razor Pages application. Separating APIs, especially public APIs, from your server-side web application has a number of benefits. These applications often will have unique deployment and load characteristics. They're also very likely to adopt different mechanisms for security, with standard form-based applications leveraging cookie-based authentication and APIs most likely using token-based authentication.

Additionally, Blazor applications, whether using Blazor Server or Blazor WebAssembly, should be built as separate projects. The applications have different runtime characteristics as well as security models. They're likely to share common types with the server-side web application (or API project), and these types should be defined in a common shared project.

The addition of a Blazor WebAssembly admin interface to `eShopOnWeb` required adding several new projects. The Blazor WebAssembly project itself, `BlazorAdmin`. A new set of public API endpoints, used

by BlazorAdmin and configured to use token-based authentication, is defined in the PublicApi project. And certain shared types used by both of these projects are kept in a new BlazorShared project.

One might ask, why add a separate BlazorShared project when there is already a common ApplicationCore project that could be used to share any types required by both PublicApi and BlazorAdmin? The answer is that this project includes all of the application's business logic and is thus much larger than necessary and also much more likely to need to be kept secure on the server. Remember that any library referenced by BlazorAdmin will be downloaded to users' browsers when they load the Blazor application.

Depending on whether one is using the [Backends-For-Frontends \(BFF\) pattern](#), the APIs consumed by the Blazor WebAssembly app may not share their types 100% with Blazor. In particular, a public API that's meant to be consumed by many different clients may define its own request and result types, rather than sharing them in a client-specific shared project. In the eShopOnWeb sample, the assumption is being made that the PublicApi project is, in fact, hosting a public API, so not all of its request and response types come from the BlazorShared project.

Cross-cutting concerns

As applications grow, it becomes increasingly important to factor out cross-cutting concerns to eliminate duplication and maintain consistency. Some examples of cross-cutting concerns in ASP.NET Core applications are authentication, model validation rules, output caching, and error handling, though there are many others. ASP.NET Core MVC [filters](#) allow you to run code before or after certain steps in the request processing pipeline. For instance, a filter can run before and after model binding, before and after an action, or before and after an action's result. You can also use an authorization filter to control access to the rest of the pipeline. Figures 7-2 shows how request execution flows through filters, if configured.

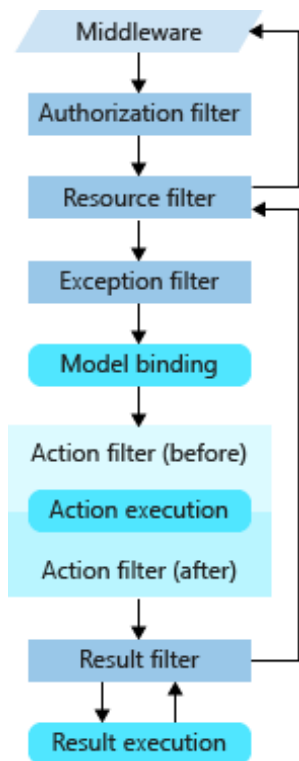


Figure 7-2. Request execution through filters and request pipeline.

Filters are usually implemented as attributes, so you can apply them to controllers or actions (or even globally). When added in this fashion, filters specified at the action level override or build upon filters specified at the controller level, which themselves override global filters. For example, the `[Route]` attribute can be used to build up routes between controllers and actions. Likewise, authorization can be configured at the controller level, and then overridden by individual actions, as the following sample demonstrates:

```

[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous] // overrides the Authorize attribute
    public async Task<IActionResult> Login() {}
    public async Task<IActionResult> ForgotPassword() {}
}
  
```

The first method, `Login`, uses the `[AllowAnonymous]` filter (attribute) to override the `Authorize` filter set at the controller level. The `ForgotPassword` action (and any other action in the class that doesn't have an `AllowAnonymous` attribute) will require an authenticated request.

Filters can be used to eliminate duplication in the form of common error handling policies for APIs. For example, a typical API policy is to return a `NotFound` response to requests referencing keys that do not exist, and a `BadRequest` response if model validation fails. The following example demonstrates these two policies in action:

```

[HttpPut("{id}")]
public async Task<IActionResult> Put(int id, [FromBody]Author author)
{
  // ...
}
  
```



```

    if ((await _authorRepository.ListAsync()).All(a => a.Id != id))
    {
        return NotFound(id);
    }
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    author.Id = id;
    await _authorRepository.UpdateAsync(author);
    return Ok();
}

```

Don't allow your action methods to become cluttered with conditional code like this. Instead, pull the policies into filters that can be applied on an as-needed basis. In this example, the model validation check, which should occur anytime a command is sent to the API, can be replaced by the following attribute:

```

public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(context.ModelState);
        }
    }
}

```

You can add the `ValidateModelAttribute` to your project as a NuGet dependency by including the [Ardalis.ValidateModel](#) package. For APIs, you can use the `ApiController` attribute to enforce this behavior without the need for a separate `ValidateModel` filter.

Likewise, a filter can be used to check if a record exists and return a 404 before the action is executed, eliminating the need to perform these checks in the action. Once you've pulled out common conventions and organized your solution to separate infrastructure code and business logic from your UI, your MVC action methods should be extremely thin:

```

[HttpPut("{id}")]
[ValidateAuthorExists]
public async Task<IActionResult> Put(int id, [FromBody]Author author)
{
    await _authorRepository.UpdateAsync(author);
    return Ok();
}

```

You can read more about implementing filters and download a working sample from the MSDN Magazine article, [Real-World ASP.NET Core MVC Filters](#).

If you find that you have a number of common responses from APIs based on common scenarios like validation errors (Bad Request), resource not found, and server errors, you might consider using a *result* abstraction. The result abstraction would be returned by services consumed by API endpoints, and the controller action or endpoint would use a filter to translate these into `IActionResults`.

References – Structuring applications

- **Areas**
<https://learn.microsoft.com/aspnet/core/mvc/controllers/areas>
- **MSDN Magazine – Feature Slices for ASP.NET Core MVC**
<https://learn.microsoft.com/archive/msdn-magazine/2016/september/asp-net-core-feature-slices-for-asp-net-core-mvc>
- **Filters**
<https://learn.microsoft.com/aspnet/core/mvc/controllers/filters>
- **MSDN Magazine – Real World ASP.NET Core MVC Filters**
<https://learn.microsoft.com/archive/msdn-magazine/2016/august/asp-net-core-real-world-asp-net-core-mvc-filters>
- **Result in eShopOnWeb**
<https://github.com/dotnet-architecture/eShopOnWeb/wiki/Patterns#result>

Security

Securing web applications is a large topic, with many considerations. At its most basic level, security involves ensuring you know who a given request is coming from, and then ensuring that the request only has access to resources it should. Authentication is the process of comparing credentials provided with a request to those in a trusted data store, to see if the request should be treated as coming from a known entity. Authorization is the process of restricting access to certain resources based on user identity. A third security concern is protecting requests from eavesdropping by third parties, for which you should at least [ensure that SSL is used by your application](#).

Identity

ASP.NET Core Identity is a membership system you can use to support login functionality for your application. It has support for local user accounts as well as external login provider support from providers like Microsoft Account, Twitter, Facebook, Google, and more. In addition to ASP.NET Core Identity, your application can use windows authentication, or a third-party identity provider like [Identity Server](#).

ASP.NET Core Identity is included in new project templates if the Individual User Accounts option is selected. This template includes support for registration, login, external logins, forgotten passwords, and additional functionality.

Additional information

ASP.NET Core Web App C# Linux macOS Windows Cloud Service Web

Framework ⓘ
.NET 7.0 (Standard Term Support)

Authentication type ⓘ
None
None
Individual Accounts
Microsoft identity platform
Windows

Linux

☐ Do not use top-level statements ⓘ

Back Create

Figure 7-3. Select Individual User Accounts to have Identity preconfigured.

Identity support is configured in *Program.cs* or *Startup*, and includes configuring services as well as middleware.

Configure Identity in *Program.cs*

In *Program.cs*, you configure services from the *WebApplicationBuilder* instance, and then once the app is created, you configure its middleware. The key points to note are the call to *AddDefaultIdentity* for required services and the *UseAuthentication* and *UseAuthorization* calls which add required middleware.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
    options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
```

```

{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production
    scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

app.Run();

```

Configuring Identity in app startup

```

// Add framework services.
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
builder.Services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
builder.Services.AddMvc();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

```

It's important that `UseAuthentication` and `UseAuthorization` appear before `MapRazorPages`. When configuring Identity services, you'll notice a call to `AddDefaultTokenProviders`. This has nothing to do with tokens that may be used to secure web communications, but instead refers to providers that create prompts that can be sent to users via SMS or email in order for them to confirm their identity.

You can learn more about [configuring two-factor authentication](#) and [enabling external login providers](#) from the official ASP.NET Core docs.

Authentication

Authentication is the process of determining who is accessing the system. If you're using ASP.NET Core Identity and the configuration methods shown in the previous section, it will automatically configure some authentication defaults in the application. However, you can also configure these defaults manually, or override the ones set by AddIdentity. If you're using Identity, it configures cookie-based authentication as the default *scheme*.

In web-based authentication, there are typically up to five actions that may be performed in the course of authenticating a client of a system. These are:

- Authenticate. Use the information provided by the client to create an identity for them to use within the application.
- Challenge. This action is used to require the client to identify themselves.
- Forbid. Inform the client they are forbidden from performing an action.
- Sign-in. Persist the existing client in some way.
- Sign-out. Remove the client from persistence.

There are a number of common techniques for performing authentication in web applications. These are referred to as schemes. A given scheme will define actions for some or all of the above options. Some schemes only support a subset of actions, and may require a separate scheme to perform those it does not support. For example, the OpenId-Connect (OIDC) scheme doesn't support Sign-in or Sign-out, but is commonly configured to use Cookie authentication for this persistence.

In your ASP.NET Core application, you can configure a DefaultAuthenticateScheme as well as optional specific schemes for each of the actions described above. For example, DefaultChallengeScheme, DefaultForbidScheme, etc. Calling AddIdentity<TUser,TRole> configures a number of aspects of the application and adds many required services. It also includes this call to configure the authentication scheme:

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = IdentityConstants.ApplicationScheme;
    options.DefaultChallengeScheme = IdentityConstants.ApplicationScheme;
    options.DefaultSignInScheme = IdentityConstants.ExternalScheme;
});
```

These schemes use cookies for persistence and redirection to login pages for authentication by default. These schemes are appropriate for web applications that interact with users via web browsers, but not recommended for APIs. Instead, APIs will typically use another form of authentication, such as JWT bearer tokens.

Web APIs are consumed by code, such as HttpClient in .NET applications and equivalent types in other frameworks. These clients expect a usable response from an API call, or a status code indicating what, if any, problem has occurred. These clients are not interacting through a browser and do not render or interact with any HTML that an API might return. Thus, it is not appropriate for API endpoints to redirect their clients to login pages if they are not authenticated. Another scheme is more appropriate.

To configure authentication for APIs, you might set up authentication like the following, used by the PublicApi project in the eShopOnWeb reference application:

```
builder.Services
    .AddAuthentication(config =>
    {
        config.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer(config =>
    {
        config.RequireHttpsMetadata = false;
        config.SaveToken = true;
        config.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(key),
            ValidateIssuer = false,
            ValidateAudience = false
        }
    });
```

While it is possible to configure multiple different authentication schemes within a single project, it is much simpler to configure a single default scheme. For this reason, among others, the eShopOnWeb reference application separates its APIs into their own project, PublicApi, separate from the main Web project that includes the application's views and Razor Pages.

Authentication in Blazor apps

Blazor Server applications can leverage the same authentication features as any other ASP.NET Core application. Blazor WebAssembly applications cannot use the built-in Identity and Authentication providers, however, since they run in the browser. Blazor WebAssembly applications can store user authentication status locally and can access claims to determine what actions users should be able to perform. However, all authentication and authorization checks should be performed on the server regardless of any logic implemented inside the Blazor WebAssembly app, since users can easily bypass the app and interact with the APIs directly.

References – Authentication

- **Authentication Actions and Defaults**
<https://stackoverflow.com/a/52493428>
- **Authentication and Authorization for SPAs**
<https://learn.microsoft.com/aspnet/core/security/authentication/identity-api-authorization>
- **ASP.NET Core Blazor Authentication and Authorization**
<https://learn.microsoft.com/aspnet/core/blazor/security/>
- **Security: Authentication and Authorization in ASP.NET Web Forms and Blazor**
<https://learn.microsoft.com/dotnet/architecture/blazor-for-web-forms-developers/security-authentication-authorization>

Authorization

The simplest form of authorization involves restricting access to anonymous users. This functionality can be achieved by applying the [Authorize] attribute to certain controllers or actions. If roles are

being used, the attribute can be further extended to restrict access to users who belong to certain roles, as shown:

```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
}
}
```

In this case, users belonging to either the HRManager or Finance roles (or both) would have access to the SalaryController. To require that a user belong to multiple roles (not just one of several), you can apply the attribute multiple times, specifying a required role each time.

Specifying certain sets of roles as strings in many different controllers and actions can lead to undesirable repetition. At a minimum, define constants for these string literals and use the constants anywhere you need to specify the string. You can also configure authorization policies, which encapsulate authorization rules, and then specify the policy instead of individual roles when applying the [Authorize] attribute:

```
[Authorize(Policy = "CanViewPrivateReport")]
public IActionResult ExecutiveSalaryReport()
{
    return View();
}
```

Using policies in this way, you can separate the kinds of actions being restricted from the specific roles or rules that apply to it. Later, if you create a new role that needs to have access to certain resources, you can just update a policy, rather than updating every list of roles on every [Authorize] attribute.

Claims

Claims are name value pairs that represent properties of an authenticated user. For example, you might store users' employee number as a claim. Claims can then be used as part of authorization policies. You could create a policy called "EmployeeOnly" that requires the existence of a claim called "EmployeeNumber", as shown in this example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddAuthorization(options =>
    {
        options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("EmployeeNumber"));
    });
}
```

This policy could then be used with the [Authorize] attribute to protect any controller and/or action, as described above.

Securing web APIs

Most web APIs should implement a token-based authentication system. Token authentication is stateless and designed to be scalable. In a token-based authentication system, the client must first authenticate with the authentication provider. If successful, the client is issued a token, which is simply

a cryptographically meaningful string of characters. The most common format for tokens is JSON Web Token, or JWT (often pronounced “jot”). When the client then needs to issue a request to an API, it adds this token as a header on the request. The server then validates the token found in the request header before completing the request. Figure 7-4 demonstrates this process.

Token-Based Authentication

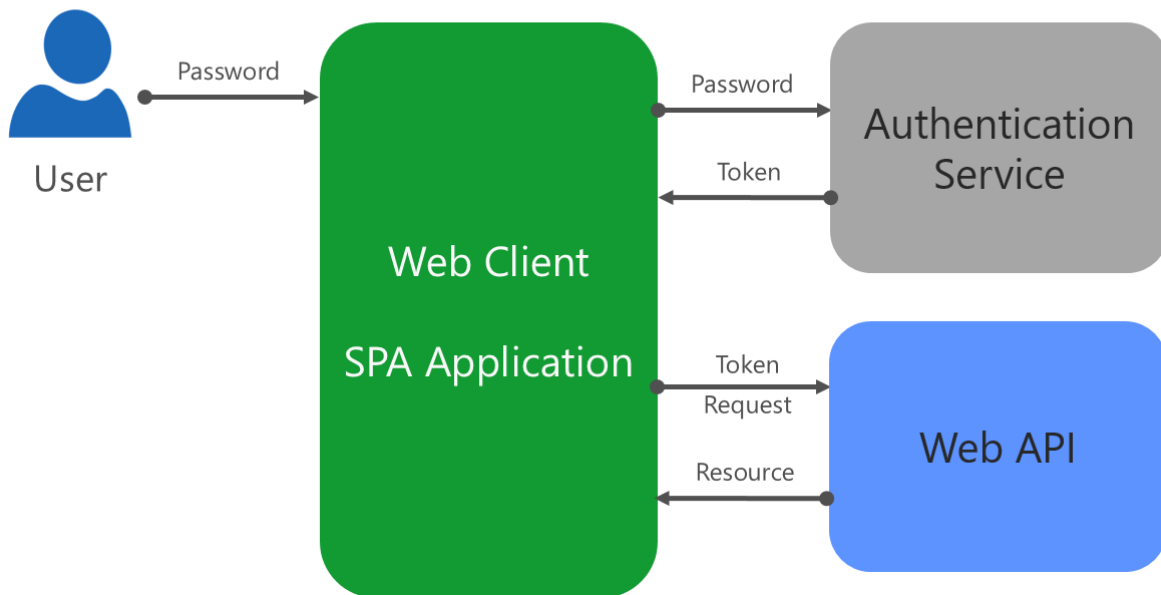


Figure 7-4. Token-based authentication for Web APIs.

You can create your own authentication service, integrate with Azure AD and OAuth, or implement a service using an open-source tool like [IdentityServer](#).

JWT tokens can embed claims about the user, which can be read on the client or server. You can use a tool like [jwt.io](#) to view the contents of a JWT token. Do not store sensitive data like passwords or keys in JWT tokens, since their contents are easily read.

When using JWT tokens with SPA or Blazor WebAssembly applications, you must store the token somewhere on the client and then add it to every API call. This activity is typically done as a header, as the following code demonstrates:

```
// AuthService.cs in BlazorAdmin project of eShopOnWeb
private async Task SetAuthorizationHeader()
{
    var token = await GetToken();
    _httpClient.DefaultRequestHeaders.Authorization = new
    AuthenticationHeaderValue("Bearer", token);
}
```

After calling the above method, requests made with the `_httpClient` will have the token embedded in the request's headers, allowing the server-side API to authenticate and authorize the request.

Custom Security

Caution

As a general rule, avoid implementing your own custom security implementations.

Be especially careful about “rolling your own” implementation of cryptography, user membership, or token generation system. There are many commercial and open-source alternatives available, which will almost certainly have better security than a custom implementation.

References – Security

- **Security Docs Overview**
<https://learn.microsoft.com/aspnet/core/security/>
- **Enforcing SSL in an ASP.NET Core App**
<https://learn.microsoft.com/aspnet/core/security/enforcing-ssl>
- **Introduction to Identity**
<https://learn.microsoft.com/aspnet/core/security/authentication/identity>
- **Introduction to Authorization**
<https://learn.microsoft.com/aspnet/core/security/authorization/introduction>
- **Authentication and Authorization for API Apps in Azure App Service**
<https://learn.microsoft.com/azure/app-service-api/app-service-api-authentication>
- **Identity Server**
<https://github.com/IdentityServer>

Client communication

In addition to serving pages and responding to requests for data via web APIs, ASP.NET Core apps can communicate directly with connected clients. This outbound communication can use a variety of transport technologies, the most common being WebSockets. ASP.NET Core SignalR is a library that makes it simple to add real-time server-to-client communication functionality to your applications. SignalR supports a variety of transport technologies, including WebSockets, and abstracts away many of the implementation details from the developer.

Real-time client communication, whether using WebSockets directly or other techniques, are useful in a variety of application scenarios. Some examples include:

- Live chat room applications
- Monitoring applications
- Job progress updates
- Notifications
- Interactive forms applications

When building client communication into your applications, there are typically two components:

- Server-side connection manager (SignalR Hub, WebSocketManager WebSocketHandler)
- Client-side library

Clients aren't limited to browsers – mobile apps, console apps, and other native apps can also communicate using SignalR/WebSockets. The following simple program echoes all content sent to a chat application to the console, as part of a WebSocketManager sample application:

```
public class Program
{
    private static Connection _connection;
    public static void Main(string[] args)
    {
        StartConnectionAsync();
        _connection.On("receiveMessage", (arguments) =>
        {
            Console.WriteLine($"{arguments[0]} said: {arguments[1]}");
        });
        Console.ReadLine();
        StopConnectionAsync();
    }

    public static async Task StartConnectionAsync()
    {
        _connection = new Connection();
        await _connection.StartConnectionAsync("ws://localhost:65110/chat");
    }

    public static async Task StopConnectionAsync()
    {
        await _connection.StopConnectionAsync();
    }
}
```

Consider ways in which your applications communicate directly with client applications, and consider whether real-time communication would improve your app's user experience.

References – Client Communication

- **ASP.NET Core SignalR**
<https://github.com/dotnet/aspnetcore/tree/main/src/SignalR>
- **WebSocket Manager**
<https://github.com/radu-matei/websocket-manager>

Domain-driven design – Should you apply it?

Domain-Driven Design (DDD) is an agile approach to building software that emphasizes focusing on the *business domain*. It places a heavy emphasis on communication and interaction with business domain expert(s) who can relate to the developers how the real-world system works. For example, if you're building a system that handles stock trades, your domain expert might be an experienced stock broker. DDD is designed to address large, complex business problems, and is often not appropriate for smaller, simpler applications, as the investment in understanding and modeling the domain is not worth it.

When building software following a DDD approach, your team (including non-technical stakeholders and contributors) should develop a *ubiquitous language* for the problem space. That is, the same terminology should be used for the real-world concept being modeled, the software equivalent, and any structures that might exist to persist the concept (for example, database tables). Thus, the concepts described in the ubiquitous language should form the basis for your *domain model*.

Your domain model comprises objects that interact with one another to represent the behavior of the system. These objects may fall into the following categories:

- [Entities](#), which represent objects with a thread of identity. Entities are typically stored in persistence with a key by which they can later be retrieved.
- [Aggregates](#), which represent groups of objects that should be persisted as a unit.
- [Value objects](#), which represent concepts that can be compared on the basis of the sum of their property values. For example, `DateRange` consisting of a start and end date.
- [Domain events](#), which represent things happening within the system that are of interest to other parts of the system.

A DDD domain model should encapsulate complex behavior within the model. Entities, in particular, should not merely be collections of properties. When the domain model lacks behavior and merely represents the state of the system, it is said to be an [anemic model](#), which is undesirable in DDD.

In addition to these model types, DDD typically employs a variety of patterns:

- [Repository](#), for abstracting persistence details.
- [Factory](#), for encapsulating complex object creation.
- [Services](#), for encapsulating complex behavior and/or infrastructure implementation details.
- [Command](#), for decoupling issuing commands and executing the command itself.
- [Specification](#), for encapsulating query details.

DDD also recommends the use of the Clean Architecture discussed previously, allowing for loose coupling, encapsulation, and code that can easily be verified using unit tests.

When should you apply DDD

DDD is well suited to large applications with significant business (not just technical) complexity. The application should require the knowledge of domain experts. There should be significant behavior in the domain model itself, representing business rules and interactions beyond simply storing and retrieving the current state of various records from data stores.

When shouldn't you apply DDD

DDD involves investments in modeling, architecture, and communication that may not be warranted for smaller applications or applications that are essentially just CRUD (create/read/update/delete). If you choose to approach your application following DDD, but find that your domain has an anemic model with no behavior, you may need to rethink your approach. Either your application may not

need DDD, or you may need assistance refactoring your application to encapsulate business logic in the domain model, rather than in your database or user interface.

A hybrid approach would be to only use DDD for the transactional or more complex areas of the application, but not for simpler CRUD or read-only portions of the application. For instance, you don't need the constraints of an Aggregate if you're querying data to display a report or to visualize data for a dashboard. It's perfectly acceptable to have a separate, simpler read model for such requirements.

References – Domain-Driven Design

- **DDD in Plain English (StackOverflow Answer)**
<https://stackoverflow.com/questions/1222392/can-someone-explain-domain-driven-design-ddd-in-plain-english-please/1222488#1222488>

Deployment

There are a few steps involved in the process of deploying your ASP.NET Core application, regardless of where it will be hosted. The first step is to publish the application, which can be done using the `dotnet publish` CLI command. This step will compile the application and place all of the files needed to run the application into a designated folder. When you deploy from Visual Studio, this step is performed for you automatically. The publish folder contains `.exe` and `.dll` files for the application and its dependencies. A self-contained application will also include a version of the .NET runtime. ASP.NET Core applications will also include configuration files, static client assets, and MVC views.

ASP.NET Core applications are console applications that must be started when the server boots and restarted if the application (or server) crashes. A process manager can be used to automate this process. The most common process managers for ASP.NET Core are Nginx and Apache on Linux and IIS or Windows Service on Windows.

In addition to a process manager, ASP.NET Core applications may use a reverse proxy server. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel after some preliminary handling. Reverse proxy servers provide a layer of security for the application. Kestrel also doesn't support hosting multiple applications on the same port, so techniques like host headers cannot be used with it to enable hosting multiple applications on the same port and IP address.

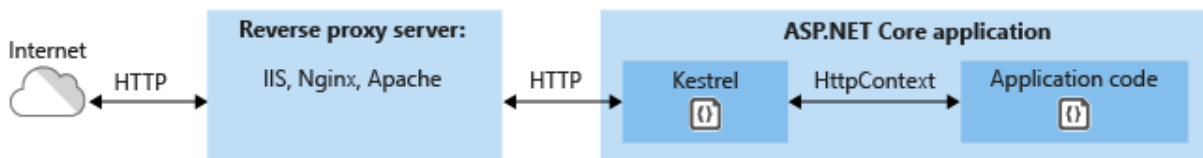


Figure 7-5. ASP.NET hosted in Kestrel behind a reverse proxy server

Another scenario in which a reverse proxy can be helpful is to secure multiple applications using SSL/HTTPS. In this case, only the reverse proxy would need to have SSL configured. Communication between the reverse proxy server and Kestrel could take place over HTTP, as shown in Figure 7-6.

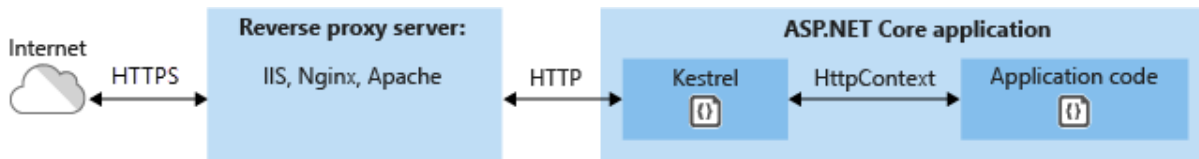


Figure 7-6. ASP.NET hosted behind an HTTPS-secured reverse proxy server

An increasingly popular approach is to host your ASP.NET Core application in a Docker container, which then can be hosted locally or deployed to Azure for cloud-based hosting. The Docker container could contain your application code, running on Kestrel, and would be deployed behind a reverse proxy server, as shown above.

If you're hosting your application on Azure, you can use Microsoft Azure Application Gateway as a dedicated virtual appliance to provide several services. In addition to acting as a reverse proxy for individual applications, Application Gateway can also offer the following features:

- HTTP load balancing
- SSL offload (SSL only to Internet)
- End to End SSL
- Multi-site routing (consolidate up to 20 sites on a single Application Gateway)
- Web application firewall
- Websocket support
- Advanced diagnostics

Learn more about Azure deployment options in [Chapter 10](#).

References – Deployment

- **Hosting and Deployment Overview**
<https://learn.microsoft.com/aspnet/core/publishing/>
- **When to use Kestrel with a reverse proxy**
<https://learn.microsoft.com/aspnet/core/fundamentals/servers/kestrel#when-to-use-kestrel-with-a-reverse-proxy>
- **Host ASP.NET Core apps in Docker**
<https://learn.microsoft.com/aspnet/core/publishing/docker>
- **Introducing Azure Application Gateway**
<https://learn.microsoft.com/azure/application-gateway/application-gateway-introduction>

Working with Data in ASP.NET Core Apps

“Data is a precious thing and will last longer than the systems themselves.”

Tim Berners-Lee

Data access is an important part of almost any software application. ASP.NET Core supports various data access options, including Entity Framework Core (and Entity Framework 6 as well), and can work with any .NET data access framework. The choice of which data access framework to use depends on the application’s needs. Abstracting these choices from the ApplicationCore and UI projects, and encapsulating implementation details in Infrastructure, helps to produce loosely coupled, testable software.

Entity Framework Core (for relational databases)

If you’re writing a new ASP.NET Core application that needs to work with relational data, then Entity Framework Core (EF Core) is the recommended way for your application to access its data. EF Core is an object-relational mapper (O/RM) that enables .NET developers to persist objects to and from a data source. It eliminates the need for most of the data access code developers would typically need to write. Like ASP.NET Core, EF Core has been rewritten from the ground up to support modular cross-platform applications. You add it to your application as a NuGet package, configure it during app startup, and request it through dependency injection wherever you need it.

To use EF Core with a SQL Server database, run the following dotnet CLI command:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

To add support for an InMemory data source, for testing:

```
dotnet add package Microsoft.EntityFrameworkCore.InMemory
```

The DbContext

To work with EF Core, you need a subclass of [DbContext](#). This class holds properties representing collections of the entities your application will work with. The eShopOnWeb sample includes a `CatalogContext` with collections for items, brands, and types:

```
public class CatalogContext : DbContext
{
    public CatalogContext(DbContextOptions<CatalogContext> options) : base(options)
    {
    }

    public DbSet<CatalogItem> CatalogItems { get; set; }
    public DbSet<CatalogBrand> CatalogBrands { get; set; }
    public DbSet<CatalogType> CatalogTypes { get; set; }
}
```

Your DbContext must have a constructor that accepts DbContextOptions and pass this argument to the base DbContext constructor. If you have only one DbContext in your application, you can pass an instance of DbContextOptions, but if you have more than one you must use the generic DbContextOptions<T> type, passing in your DbContext type as the generic parameter.

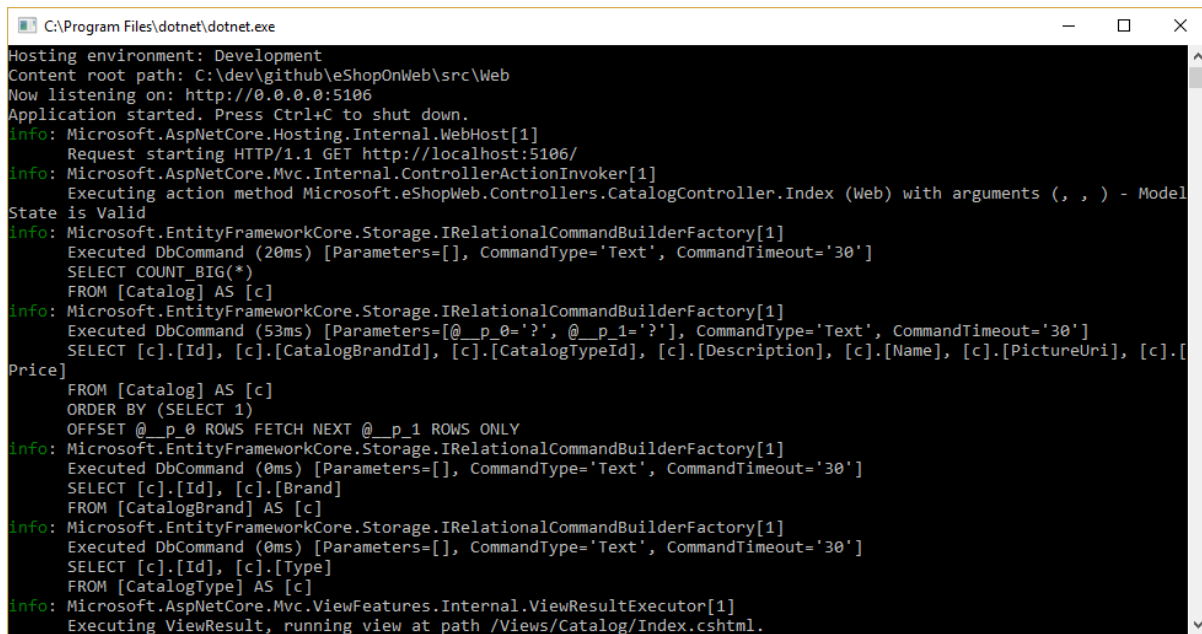
Configuring EF Core

In your ASP.NET Core application, you'll typically configure EF Core in *Program.cs* with your application's other dependencies. EF Core uses a DbContextOptionsBuilder, which supports several helpful extension methods to streamline its configuration. To configure CatalogContext to use a SQL Server database with a connection string defined in Configuration, you would add the following code:

```
builder.Services.AddDbContext<CatalogContext>(
    options => options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection")));
To use the in-memory database:
builder.Services.AddDbContext<CatalogContext>(options =>
    options.UseInMemoryDatabase());
```

Once you have installed EF Core, created a DbContext child type, and added the type to the application's services, you are ready to use EF Core. You can request an instance of your DbContext type in any service that needs it and start working with your persisted entities using LINQ as if they were simply in a collection. EF Core does the work of translating your LINQ expressions into SQL queries to store and retrieve your data.

You can see the queries EF Core is executing by configuring a logger and ensuring its level is set to at least Information, as shown in Figure 8-1.



```
C:\Program Files\dotnet\dotnet.exe
Hosting environment: Development
Content root path: C:\dev\github\eShopOnWeb\src\Web
Now listening on: http://0.0.0.0:5106
Application started. Press Ctrl+C to shut down.
Info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5106/
Info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method Microsoft.eShopWeb.Controllers.CatalogController.Index (Web) with arguments (, , ) - Model
State is Valid
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (20ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT COUNT_BIG(*)
      FROM [Catalog] AS [c]
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (53ms) [Parameters=[@__p_0='?', @__p_1='?'], CommandType='Text', CommandTimeout='30']
      SELECT [c].[Id], [c].[CatalogBrandId], [c].[CatalogTypeId], [c].[Description], [c].[Name], [c].[PictureUri], [c].[
Price]
      FROM [Catalog] AS [c]
      ORDER BY (SELECT 1)
      OFFSET @__p_0 ROWS FETCH NEXT @__p_1 ROWS ONLY
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [c].[Id], [c].[Brand]
      FROM [CatalogBrand] AS [c]
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [c].[Id], [c].[Type]
      FROM [CatalogType] AS [c]
Info: Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.ViewResultExecutor[1]
      Executing ViewResult, running view at path /Views/Catalog/Index.cshtml.
```

Figure 8-1. Logging EF Core queries to the console

Fetching and storing Data

To retrieve data from EF Core, you access the appropriate property and use LINQ to filter the result. You can also use LINQ to perform projection, transforming the result from one type to another. The following example would retrieve `CatalogBrands`, ordered by name, filtered by their `Enabled` property, and projected onto a `SelectListItem` type:

```
var brandItems = await _context.CatalogBrands
    .Where(b => b.Enabled)
    .OrderBy(b => b.Name)
    .Select(b => new SelectListItem {
        Value = b.Id, Text = b.Name })
    .ToListAsync();
```

It's important in the above example to add the call to `ToListAsync` in order to execute the query immediately. Otherwise, the statement will assign an `IQueryable<SelectListItem>` to `brandItems`, which will not be executed until it is enumerated. There are pros and cons to returning `IQueryable` results from methods. It allows the query EF Core will construct to be further modified, but can also result in errors that only occur at run time, if operations are added to the query that EF Core cannot translate. It's generally safer to pass any filters into the method performing the data access, and return back an in-memory collection (for example, `List<T>`) as the result.

EF Core tracks changes on entities it fetches from persistence. To save changes to a tracked entity, you just call the `SaveChangesAsync` method on the `DbContext`, making sure it's the same `DbContext` instance that was used to fetch the entity. Adding and removing entities is directly done on the appropriate `DbSet` property, again with a call to `SaveChangesAsync` to execute the database commands. The following example demonstrates adding, updating, and removing entities from persistence.


```
// create
var newBrand = new CatalogBrand() { Brand = "Acme" };
_context.Add(newBrand);
await _context.SaveChangesAsync();

// read and update
var existingBrand = _context.CatalogBrands.Find(1);
existingBrand.Brand = "Updated Brand";
await _context.SaveChangesAsync();

// read and delete (alternate Find syntax)
var brandToDelete = _context.Find<CatalogBrand>(2);
_context.CatalogBrands.Remove(brandToDelete);
await _context.SaveChangesAsync();
```

EF Core supports both synchronous and async methods for fetching and saving. In web applications, it's recommended to use the async/await pattern with the async methods, so that web server threads are not blocked while waiting for data access operations to complete.

For more information, see [Buffering and Streaming](#).

Fetching related data

When EF Core retrieves entities, it populates all of the properties that are stored directly with that entity in the database. Navigation properties, such as lists of related entities, are not populated and may have their value set to null. This process ensures EF Core is not fetching more data than is needed, which is especially important for web applications, which must quickly process requests and return responses in an efficient manner. To include relationships with an entity using *eager loading*, you specify the property using the Include extension method on the query, as shown:

```
// .Include requires using Microsoft.EntityFrameworkCore
var brandsWithItems = await _context.CatalogBrands
    .Include(b => b.Items)
    .ToListAsync();
```

You can include multiple relationships, and you can also include subrelationships using ThenInclude. EF Core will execute a single query to retrieve the resulting set of entities. Alternately you can include navigation properties of navigation properties by passing a '.'-separated string to the .Include() extension method, like so:

```
.Include("Items.Products")
```

In addition to encapsulating filtering logic, a specification can specify the shape of the data to be returned, including which properties to populate. The eShopOnWeb sample includes several specifications that demonstrate encapsulating eager loading information within the specification. You can see how the specification is used as part of a query here:

```
// Includes all expression-based includes
query = specification.Include.Aggregate(query,
    (current, include) => current.Include(include));

// Include any string-based include statements
query = specification.IncludeStrings.Aggregate(query,
    (current, include) => current.Include(include));
```

Another option for loading related data is to use *explicit loading*. Explicit loading allows you to load additional data into an entity that has already been retrieved. Since this approach involves a separate request to the database, it's not recommended for web applications, which should minimize the number of database round trips made per request.

Lazy loading is a feature that automatically loads related data as it is referenced by the application. EF Core has added support for lazy loading in version 2.1. Lazy loading is not enabled by default and requires installing the `Microsoft.EntityFrameworkCore.Proxies`. As with explicit loading, lazy loading should typically be disabled for web applications, since its use will result in additional database queries being made within each web request. Unfortunately, the overhead incurred by lazy loading often goes unnoticed at development time, when the latency is small and often the data sets used for testing are small. However, in production, with more users, more data, and more latency, the additional database requests can often result in poor performance for web applications that make heavy use of lazy loading.

[Avoid Lazy Loading Entities in Web Applications](#)

It's a good idea to test your application while examining the actual database queries it makes. Under certain circumstances, EF Core may make many more queries or a more expensive query than is optimal for the application. One such problem is known as a [Cartesian Explosion](#). The EF Core team makes available the [AsSplitQuery method](#) as one of several ways to tune runtime behavior.

Encapsulating data

EF Core supports several features that allow your model to properly encapsulate its state. A common problem in domain models is that they expose collection navigation properties as publicly accessible list types. This problem allows any collaborator to manipulate the contents of these collection types, which may bypass important business rules related to the collection, possibly leaving the object in an invalid state. The solution to this problem is to expose read-only access to related collections, and explicitly provide methods defining ways in which clients can manipulate them, as in this example:

```
public class Basket : BaseEntity
{
    public string BuyerId { get; set; }
    private readonly List<BasketItem> _items = new List<BasketItem>();
    public IReadOnlyCollection<BasketItem> Items => _items.AsReadOnly();

    public void AddItem(int catalogItemId, decimal unitPrice, int quantity = 1)
    {
        var existingItem = Items.FirstOrDefault(i => i.CatalogItemId == catalogItemId);
        if (existingItem == null)
        {
            _items.Add(new BasketItem()
            {
                CatalogItemId = catalogItemId,
                Quantity = quantity,
                UnitPrice = unitPrice
            });
        }
        else existingItem.Quantity += quantity;
    }
}
```

This entity type doesn't expose a public List or ICollection property, but instead exposes an IReadOnlyCollection type that wraps the underlying List type. When using this pattern, you can indicate to Entity Framework Core to use the backing field like so:

```
private void ConfigureBasket(EntityTypeBuilder<Basket> builder)
{
    var navigation = builder.Metadata.FindNavigation(nameof(Basket.Items));
    navigation.SetPropertyAccessMode(PropertyAccessMode.Field);
}
```

Another way in which you can improve your domain model is by using value objects for types that lack identity and are only distinguished by their properties. Using such types as properties of your entities can help keep logic specific to the value object where it belongs, and can avoid duplicate logic between multiple entities that use the same concept. In Entity Framework Core, you can persist value objects in the same table as their owning entity by configuring the type as an owned entity, like so:

```
private void ConfigureOrder(EntityTypeBuilder<Order> builder)
{
    builder.OwnsOne(o => o.ShipToAddress);
}
```

In this example, the ShipToAddress property is of type Address. Address is a value object with several properties such as Street and City. EF Core maps the Order object to its table with one column per Address property, prefixing each column name with the name of the property. In this example, the Order table would include columns such as ShipToAddress_Street and ShipToAddress_City. It's also possible to store owned types in separate tables, if desired.

Learn more about owned [entity support in EF Core](#).

Resilient connections

External resources like SQL databases may occasionally be unavailable. In cases of temporary unavailability, applications can use retry logic to avoid raising an exception. This technique is commonly referred to as *connection resiliency*. You can implement your [own retry with exponential backoff](#) technique by attempting to retry with an exponentially increasing wait time, until a maximum retry count has been reached. This technique embraces the fact that cloud resources might intermittently be unavailable for short periods of time, resulting in the failure of some requests.

For Azure SQL DB, Entity Framework Core already provides internal database connection resiliency and retry logic. But you need to enable the Entity Framework execution strategy for each DbContext connection if you want to have resilient EF Core connections.

For instance, the following code at the EF Core connection level enables resilient SQL connections that are retried if the connection fails.

```
builder.Services.AddDbContext<OrderingContext>(options =>
{
    options.UseSqlServer(builder.Configuration["ConnectionString"],
        sqlServerOptionsAction: sqlOptions =>
        {
            sqlOptions.EnableRetryOnFailure(
                maxRetryCount: 5,
```

```

        maxRetryDelay: TimeSpan.FromSeconds(30),
        errorNumbersToAdd: null);
    }
});

```

Execution strategies and explicit transactions using BeginTransaction and multiple DbContexts

When retries are enabled in EF Core connections, each operation you perform using EF Core becomes its own retryable operation. Each query and each call to `SaveChangesAsync` will be retried as a unit if a transient failure occurs.

However, if your code initiates a transaction using `BeginTransaction`, you are defining your own group of operations that need to be treated as a unit; everything inside the transaction has to be rolled back if a failure occurs. You will see an exception like the following if you attempt to execute that transaction when using an EF execution strategy (retry policy) and you include several `SaveChangesAsync` from multiple `DbContext`s in it.

`System.InvalidOperationException`: The configured execution strategy `SqlServerRetryingExecutionStrategy` does not support user initiated transactions. Use the execution strategy returned by `DbContext.Database.CreateExecutionStrategy()` to execute all the operations in the transaction as a retryable unit.

The solution is to manually invoke the EF execution strategy with a delegate representing everything that needs to be executed. If a transient failure occurs, the execution strategy will invoke the delegate again. The following code shows how to implement this approach:

```

// Use of an EF Core resiliency strategy when using multiple DbContexts
// within an explicit transaction
// See:
// https://learn.microsoft.com/ef/core/miscellaneous/connection-resiliency
var strategy = _catalogContext.Database.CreateExecutionStrategy();
await strategy.ExecuteAsync(async () =>
{
    // Achieving atomicity between original Catalog database operation and the
    // IntegrationEventLog thanks to a local transaction
    using (var transaction = _catalogContext.Database.BeginTransaction())
    {
        _catalogContext.CatalogItems.Update(catalogItem);
        await _catalogContext.SaveChangesAsync();

        // Save to EventLog only if product price changed
        if (raiseProductPriceChangedEvent)
        {
            await _integrationEventLogService.SaveEventAsync(priceChangedEvent);
            transaction.Commit();
        }
    }
});

```

The first `DbContext` is the `_catalogContext` and the second `DbContext` is within the `_integrationEventLogService` object. Finally, the `Commit` action would be performed multiple `DbContext`s and using an EF Execution Strategy.

References – Entity Framework Core

- **EF Core Docs** <https://learn.microsoft.com/ef/>
- **EF Core: Related Data** <https://learn.microsoft.com/ef/core/querying/related-data>
- **Avoid Lazy Loading Entities in ASP.NET Applications** <https://ardalis.com/avoid-lazy-loading-entities-in-asp-net-applications>

EF Core or micro-ORM?

While EF Core is a great choice for managing persistence, and for the most part encapsulates database details from application developers, it isn't the only choice. Another popular open-source alternative is [Dapper](#), a so-called micro-ORM. A micro-ORM is a lightweight, less full-featured tool for mapping objects to data structures. In the case of Dapper, its design goals focus on performance, rather than fully encapsulating the underlying queries it uses to retrieve and update data. Because it doesn't abstract SQL from the developer, Dapper is "closer to the metal" and lets developers write the exact queries they want to use for a given data access operation.

EF Core has two significant features it provides which separate it from Dapper but also add to its performance overhead. The first is the translation from LINQ expressions into SQL. These translations are cached, but even so there is overhead in performing them the first time. The second is change tracking on entities (so that efficient update statements can be generated). This behavior can be turned off for specific queries by using the [AsNoTracking](#) extension. EF Core also generates SQL queries that usually are very efficient and in any case perfectly acceptable from a performance standpoint, but if you need fine control over the precise query to be executed, you can pass in custom SQL (or execute a stored procedure) using EF Core, too. In this case, Dapper still outperforms EF Core, but only very slightly. Current performance benchmark data for a variety of data access methods can be found on [the Dapper site](#).

To see how the syntax for Dapper varies from EF Core, consider these two versions of the same method for retrieving a list of items:

```
// EF Core
private readonly CatalogContext _context;
public async Task<IEnumerable<CatalogType>> GetCatalogTypes()
{
    return await _context.CatalogTypes.ToListAsync();
}

// Dapper
private readonly SqlConnection _conn;
public async Task<IEnumerable<CatalogType>> GetCatalogTypesWithDapper()
{
    return await _conn.QueryAsync<CatalogType>("SELECT * FROM CatalogType");
}
```

If you need to build more complex object graphs with Dapper, you need to write the associated queries yourself (as opposed to adding an Include as you would in EF Core). This functionality is supported through various syntaxes, including a feature called Multi Mapping that lets you map individual rows to multiple mapped objects. For example, given a class Post with a property Owner of type User, the following SQL would return all of the necessary data:

```
select * from #Posts p
left join #Users u on u.Id = p.OwnerId
Order by p.Id
```

Each returned row includes both User and Post data. Since the User data should be attached to the Post data via its Owner property, the following function is used:

```
(post, user) => { post.Owner = user; return post; }
```

The full code listing to return a collection of posts with their Owner property populated with the associated user data would be:

```
var sql = @"select * from #Posts p
left join #Users u on u.Id = p.OwnerId
Order by p.Id";
var data = connection.Query<Post, User, Post>(sql,
(post, user) => { post.Owner = user; return post; });
```

Because it offers less encapsulation, Dapper requires developers know more about how their data is stored, how to query it efficiently, and write more code to fetch it. When the model changes, instead of simply creating a new migration (another EF Core feature), and/or updating mapping information in one place in a DbContext, every query that is impacted must be updated. These queries have no compile-time guarantees, so they may break at run time in response to changes to the model or database, making errors more difficult to detect quickly. In exchange for these tradeoffs, Dapper offers extremely fast performance.

For most applications, and most parts of almost all applications, EF Core offers acceptable performance. Thus, its developer productivity benefits are likely to outweigh its performance overhead. For queries that can benefit from caching, the actual query may only be executed a tiny percentage of the time, making relatively small query performance differences moot.

SQL or NoSQL

Traditionally, relational databases like SQL Server have dominated the marketplace for persistent data storage, but they are not the only solution available. NoSQL databases like [MongoDB](#) offer a different approach to storing objects. Rather than mapping objects to tables and rows, another option is to serialize the entire object graph, and store the result. The benefits of this approach, at least initially, are simplicity and performance. It's simpler to store a single serialized object with a key than to decompose the object into many tables with relationships and update rows that may have changed since the object was last retrieved from the database. Likewise, fetching and deserializing a single object from a key-based store is typically much faster and easier than complex joins or multiple database queries required to fully compose the same object from a relational database. The lack of locks or transactions or a fixed schema also makes NoSQL databases amenable to scaling across many machines, supporting very large datasets.

On the other hand, NoSQL databases (as they are typically called) have their drawbacks. Relational databases use normalization to enforce consistency and avoid duplication of data. This approach reduces the total size of the database and ensures that updates to shared data are available immediately throughout the database. In a relational database, an Address table might reference a

Country table by ID, such that if the name of a country/region were changed, the address records would benefit from the update without themselves having to be updated. However, in a NoSQL database, Address, and its associated Country might be serialized as part of many stored objects. An update to a country/region name would require all such objects to be updated, rather than a single row. Relational databases can also ensure relational integrity by enforcing rules like foreign keys. NoSQL databases typically do not offer such constraints on their data.

Another complexity NoSQL databases must deal with is versioning. When an object's properties change, it may not be able to be deserialized from past versions that were stored. Thus, all existing objects that have a serialized (previous) version of the object must be updated to conform to its new schema. This approach is not conceptually different from a relational database, where schema changes sometimes require update scripts or mapping updates. However, the number of entries that must be modified is often much greater in the NoSQL approach, because there is more duplication of data.

It's possible in NoSQL databases to store multiple versions of objects, something fixed schema relational databases typically do not support. However, in this case, your application code will need to account for the existence of previous versions of objects, adding additional complexity.

NoSQL databases typically do not enforce [ACID](#), which means they have both performance and scalability benefits over relational databases. They're well suited to extremely large datasets and objects that are not well suited to storage in normalized table structures. There is no reason why a single application cannot take advantage of both relational and NoSQL databases, using each where it is best suited.

Azure Cosmos DB

Azure Cosmos DB is a fully managed NoSQL database service that offers cloud-based schema-free data storage. Azure Cosmos DB is built for fast and predictable performance, high availability, elastic scaling, and global distribution. Despite being a NoSQL database, developers can use rich and familiar SQL query capabilities on JSON data. All resources in Azure Cosmos DB are stored as JSON documents. Resources are managed as *items*, which are documents containing metadata, and *feeds*, which are collections of items. Figure 8-2 shows the relationship between different Azure Cosmos DB resources.

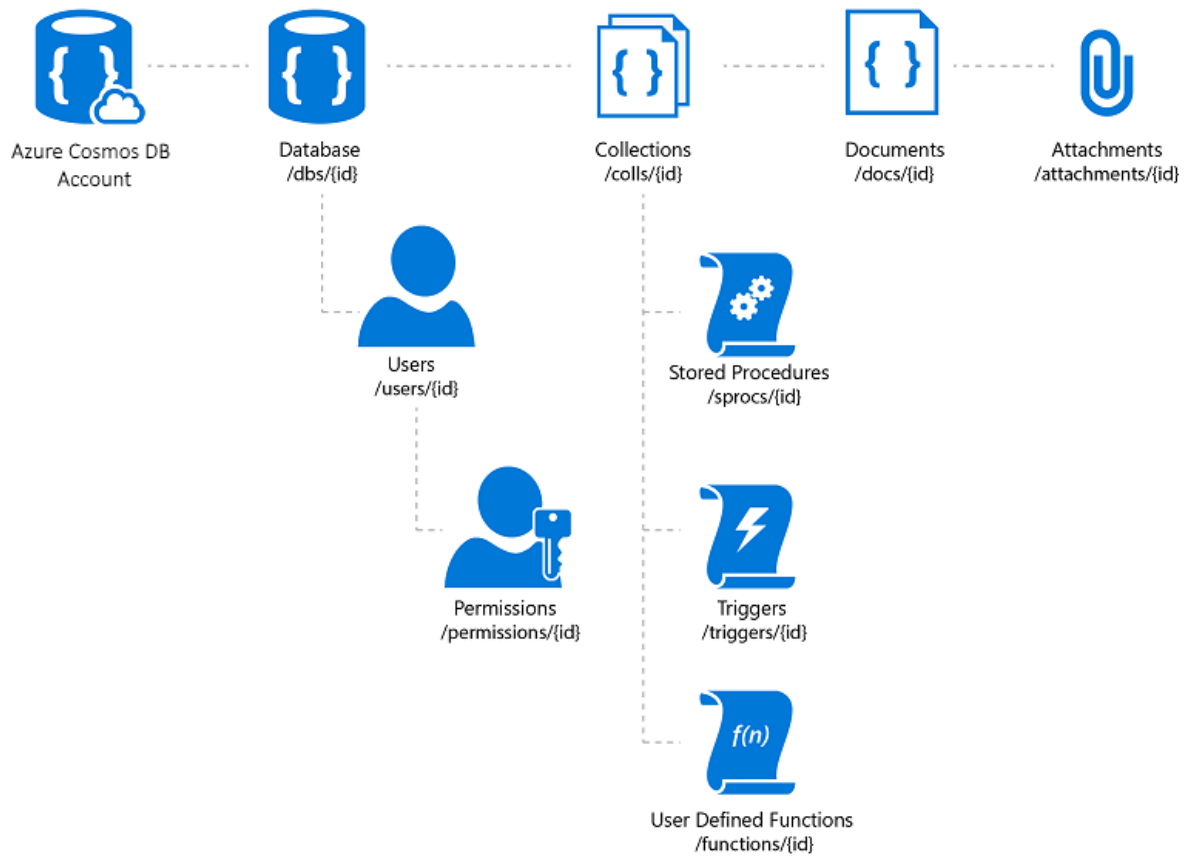


Figure 8-2. Azure Cosmos DB resource organization.

The Azure Cosmos DB query language is a simple yet powerful interface for querying JSON documents. The language supports a subset of ANSI SQL grammar and adds deep integration of JavaScript object, arrays, object construction, and function invocation.

References – Azure Cosmos DB

- Azure Cosmos DB Introduction <https://learn.microsoft.com/azure/cosmos-db/introduction>

Other persistence options

In addition to relational and NoSQL storage options, ASP.NET Core applications can use Azure Storage to store various data formats and files in a cloud-based, scalable fashion. Azure Storage is massively scalable, so you can start out storing small amounts of data and scale up to storing hundreds or terabytes if your application requires it. Azure Storage supports four kinds of data:

- Blob Storage for unstructured text or binary storage, also referred to as object storage.
- Table Storage for structured datasets, accessible via row keys.
- Queue Storage for reliable queue-based messaging.

- File Storage for shared file access between Azure virtual machines and on-premises applications.

References – Azure Storage

- Azure Storage Introduction <https://learn.microsoft.com/azure/storage/common/storage-introduction>

Caching

In web applications, each web request should be completed in the shortest time possible. One way to achieve this functionality is to limit the number of external calls the server must make to complete the request. Caching involves storing a copy of data on the server (or another data store that is more easily queried than the source of the data). Web applications, and especially non-SPA traditional web applications, need to build the entire user interface with every request. This approach frequently involves making many of the same database queries repeatedly from one user request to the next. In most cases, this data changes rarely, so there is little reason to constantly request it from the database. ASP.NET Core supports response caching, for caching entire pages, and data caching, which supports more granular caching behavior.

When implementing caching, it's important to keep in mind separation of concerns. Avoid implementing caching logic in your data access logic, or in your user interface. Instead, encapsulate caching in its own classes, and use configuration to manage its behavior. This approach follows the Open/Closed and Single Responsibility principles, and will make it easier for you to manage how you use caching in your application as it grows.

ASP.NET Core response caching

ASP.NET Core supports two levels of response caching. The first level does not cache anything on the server, but adds HTTP headers that instruct clients and proxy servers to cache responses. This functionality is implemented by adding the `ResponseCache` attribute to individual controllers or actions:

```
[ResponseCache(Duration = 60)]
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";
    return View();
}
```

The previous example will result in the following header being added to the response, instructing clients to cache the result for up to 60 seconds.

```
Cache-Control: public,max-age=60
```

In order to add server-side in-memory caching to the application, you must reference the `Microsoft.AspNetCore.ResponseCaching` NuGet package, and then add the Response Caching middleware. This middleware is configured with services and middleware during app startup:

```
builder.Services.AddResponseCaching();

// other code omitted, including building the app

app.UseResponseCaching();
```

The Response Caching Middleware will automatically cache responses based on a set of conditions, which you can customize. By default, only 200 (OK) responses requested via GET or HEAD methods are cached. In addition, requests must have a response with a Cache-Control: public header, and cannot include headers for Authorization or Set-Cookie. See a [complete list of the caching conditions used by the response caching middleware](#).

Data caching

Rather than (or in addition to) caching full web responses, you can cache the results of individual data queries. For this functionality, you can use in memory caching on the web server, or use [a distributed cache](#). This section will demonstrate how to implement in memory caching.

Add support for memory (or distributed) caching with the following code:

```
builder.Services.AddMemoryCache();
builder.Services.AddMvc();
```

Be sure to add the Microsoft.Extensions.Caching.Memory NuGet package as well.

Once you've added the service, you request IMemoryCache via dependency injection wherever you need to access the cache. In this example, the CachedCatalogService is using the Proxy (or Decorator) design pattern, by providing an alternative implementation of ICatalogService that controls access to (or adds behavior to) the underlying CatalogService implementation.

```
public class CachedCatalogService : ICatalogService
{
    private readonly IMemoryCache _cache;
    private readonly CatalogService _catalogService;
    private static readonly string _brandsKey = "brands";
    private static readonly string _typesKey = "types";
    private static readonly TimeSpan _defaultCacheDuration = TimeSpan.FromSeconds(30);

    public CachedCatalogService(
        IMemoryCache cache,
        CatalogService catalogService)
    {
        _cache = cache;
        _catalogService = catalogService;
    }

    public async Task<IEnumerable<SelectListItem>> GetBrands()
    {
        return await _cache.GetOrCreateAsync(_brandsKey, async entry =>
        {
            entry.SlidingExpiration = _defaultCacheDuration;
            return await _catalogService.GetBrands();
        });
    }

    public async Task<Catalog> GetCatalogItems(int pageIndex, int itemsPage, int? brandID,
```

```

int? typeId)
{
    string cacheKey = $"items-{pageIndex}-{itemsPage}-{brandID}-{typeId}";
    return await _cache.GetOrCreateAsync(cacheKey, async entry =>
    {
        entry.SlidingExpiration = _defaultCacheDuration;
        return await _catalogService.GetCatalogItems(pageIndex, itemsPage, brandID,
typeId);
    });
}

public async Task<IEnumerable<SelectListItem>> GetTypes()
{
    return await _cache.GetOrCreateAsync(_typesKey, async entry =>
    {
        entry.SlidingExpiration = _defaultCacheDuration;
        return await _catalogService.GetTypes();
    });
}
}

```

To configure the application to use the cached version of the service, but still allow the service to get the instance of `CatalogService` it needs in its constructor, you would add the following lines in `Program.cs`:

```

builder.Services.AddMemoryCache();
builder.Services.AddScoped<ICatalogService, CachedCatalogService>();
builder.Services.AddScoped<CatalogService>();

```

With this code in place, the database calls to fetch the catalog data will only be made once per minute, rather than on every request. Depending on the traffic to the site, this can have a significant impact on the number of queries made to the database, and the average page load time for the home page that currently depends on all three of the queries exposed by this service.

An issue that arises when caching is implemented is *stale data* – that is, data that has changed at the source but an out-of-date version remains in the cache. A simple way to mitigate this issue is to use small cache durations, since for a busy application there is a limited additional benefit to extending the length data is cached. For example, consider a page that makes a single database query, and is requested 10 times per second. If this page is cached for one minute, it will result in the number of database queries made per minute to drop from 600 to 1, a reduction of 99.8%. If instead the cache duration was made one hour, the overall reduction would be 99.997%, but now the likelihood and potential age of stale data are both increased dramatically.

Another approach is to proactively remove cache entries when the data they contain is updated. Any individual entry can be removed if its key is known:

```

_cache.Remove(cacheKey);

```

If your application exposes functionality for updating entries that it caches, you can remove the corresponding cache entries in your code that performs the updates. Sometimes there may be many different entries that depend on a particular set of data. In that case, it can be useful to create dependencies between cache entries, by using a `CancellationToken`. With a `CancellationToken`, you can expire multiple cache entries at once by canceling the token.

```
// configure CancellationToken and add entry to cache
var cts = new CancellationTokenSource();
_cache.Set("cts", cts);
_cache.Set(cacheKey, itemToCache, new CancellationChangeToken(cts.Token));

// elsewhere, expire the cache by cancelling the token\
_cache.Get<CancellationTokenSource>("cts").Cancel();
```

Caching can dramatically improve the performance of web pages that repeatedly request the same values from the database. Be sure to measure data access and page performance before applying caching, and only apply caching where you see a need for improvement. Caching consumes web server memory resources and increases the complexity of the application, so it's important you don't prematurely optimize using this technique.

Getting data to Blazor WebAssembly apps

If you're building apps that use Blazor Server, you can use Entity Framework and other direct data access technologies as they've been discussed thus far in this chapter. However, when building Blazor WebAssembly apps, like other SPA frameworks, you will need a different strategy for data access. Typically, these applications access data and interact with the server through web API endpoints.

If the data or operations being performed are sensitive, be sure to review the section on security in the [previous chapter](#) and protect your APIs against unauthorized access.

You'll find an example of a Blazor WebAssembly app in the [eShopOnWeb reference application](#), in the BlazorAdmin project. This project is hosted within the eShopOnWeb Web project, and allows users in the Administrators group to manage the items in the store. You can see a screenshot of the application in Figure 8-3.

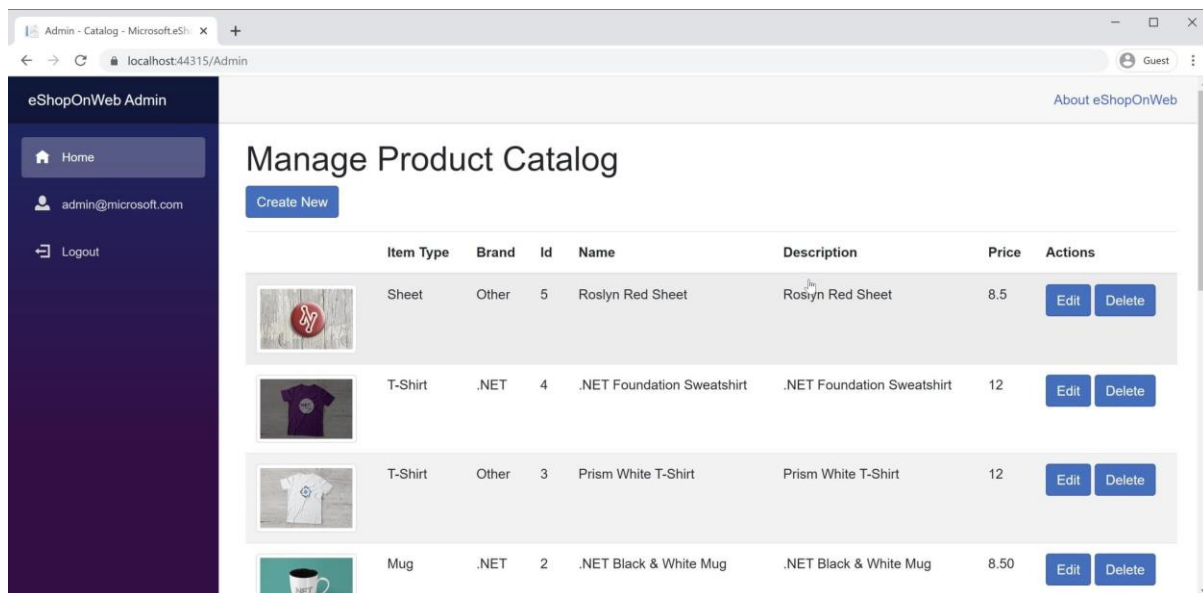


Figure 8-3. eShopOnWeb Catalog Admin Screenshot.

When fetching data from web APIs within a Blazor WebAssembly app, you just use an instance of `HttpClient` as you would in any .NET application. The basic steps involved are to create the request to send (if necessary, usually for POST or PUT requests), await the request itself, verify the status code, and deserialize the response. If you're going to make many requests to a given set of APIs, it's a good idea to encapsulate your APIs and configure the `HttpClient` base address centrally. This way, if you need to adjust any of these settings between environments, you can make the changes in just one place. You should add support for this service in your `Program.Main`:

```
builder.Services.AddScoped(sp => new HttpClient
{
    BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
});
```

If you need to access services securely, you should access a secure token and configure the `HttpClient` to pass this token as an `Authentication` header with every request:

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

This activity can be done from any component that has the `HttpClient` injected into it, provided that `HttpClient` wasn't added to the application's services with a `Transient` lifetime. Every reference to `HttpClient` in the application references the same instance, so changes to it in one component flow through the entire application. A good place to perform this authentication check (followed by specifying the token) is in a shared component like the main navigation for the site. Learn more about this approach in the `BlazorAdmin` project in the [eShopOnWeb reference application](#).

One benefit of Blazor WebAssembly over traditional JavaScript SPAs is that you don't need to keep copies of your data transfer objects (DTOs) synchronized. Your Blazor WebAssembly project and your web API project can both share the same DTOs in a common shared project. This approach eliminates some of the friction involved in developing SPAs.

To quickly get data from an API endpoint, you can use the built-in helper method, `GetFromJsonAsync`. There are similar methods for POST, PUT, etc. The following shows how to get a `CatalogItem` from an API endpoint using a configured `HttpClient` in a Blazor WebAssembly app:

```
var item = await _httpClient.GetFromJsonAsync<CatalogItem>($"catalog-items/{id}");
```

Once you have the data you need, you'll typically track changes locally. When you want to make updates to the backend data store, you'll call additional web APIs for this purpose.

References – Blazor Data

- Call a web API from ASP.NET Core Blazor <https://learn.microsoft.com/aspnet/core/blazor/call-web-api>

Test ASP.NET Core MVC apps

"If you don't like unit testing your product, most likely your customers won't like to test it, either." _- Anonymous-

Software of any complexity can fail in unexpected ways in response to changes. Thus, testing after making changes is required for all but the most trivial (or least critical) applications. Manual testing is the slowest, least reliable, most expensive way to test software. Unfortunately, if applications aren't designed to be testable, it can be the only means of testing available. Applications written to follow the architectural principles laid out in [chapter 4](#) should be largely unit testable. ASP.NET Core applications support automated integration and functional testing.

Kinds of automated tests

There are many kinds of automated tests for software applications. The simplest, lowest level test is the unit test. At a slightly higher level, there are integration tests and functional tests. Other kinds of tests, such as UI tests, load tests, stress tests, and smoke tests, are beyond the scope of this document.

Unit tests

A unit test tests a single part of your application's logic. One can further describe it by listing some of the things that it isn't. A unit test doesn't test how your code works with dependencies or infrastructure – that's what integration tests are for. A unit test doesn't test the framework your code is written on – you should assume it works or, if you find it doesn't, file a bug and code a workaround. A unit test runs completely in memory and in process. It doesn't communicate with the file system, the network, or a database. Unit tests should only test your code.

Unit tests, by virtue of the fact that they test only a single unit of your code, with no external dependencies, should execute extremely fast. Thus, you should be able to run test suites of hundreds of unit tests in a few seconds. Run them frequently, ideally before every push to a shared source control repository, and certainly with every automated build on your build server.

Integration tests

Although it's a good idea to encapsulate your code that interacts with infrastructure like databases and file systems, you will still have some of that code, and you will probably want to test it.

Additionally, you should verify that your code's layers interact as you expect when your application's dependencies are fully resolved. This functionality is the responsibility of integration tests. Integration tests tend to be slower and more difficult to set up than unit tests, because they often depend on external dependencies and infrastructure. Thus, you should avoid testing things that could be tested with unit tests in integration tests. If you can test a given scenario with a unit test, you should test it with a unit test. If you can't, then consider using an integration test.

Integration tests will often have more complex setup and teardown procedures than unit tests. For example, an integration test that goes against an actual database will need a way to return the database to a known state before each test run. As new tests are added and the production database schema evolves, these test scripts will tend to grow in size and complexity. In many large systems, it is impractical to run full suites of integration tests on developer workstations before checking in changes to shared source control. In these cases, integration tests may be run on a build server.

Functional tests

Integration tests are written from the perspective of the developer, to verify that some components of the system work correctly together. Functional tests are written from the perspective of the user, and verify the correctness of the system based on its requirements. The following excerpt offers a useful analogy for how to think about functional tests, compared to unit tests:

"Many times the development of a system is likened to the building of a house. While this analogy isn't quite correct, we can extend it for the purposes of understanding the difference between unit and functional tests. Unit testing is analogous to a building inspector visiting a house's construction site. He is focused on the various internal systems of the house, the foundation, framing, electrical, plumbing, and so on. He ensures (tests) that the parts of the house will work correctly and safely, that is, meet the building code. Functional tests in this scenario are analogous to the homeowner visiting this same construction site. He assumes that the internal systems will behave appropriately, that the building inspector is performing his task. The homeowner is focused on what it will be like to live in this house. He is concerned with how the house looks, are the various rooms a comfortable size, does the house fit the family's needs, are the windows in a good spot to catch the morning sun. The homeowner is performing functional tests on the house. He has the user's perspective. The building inspector is performing unit tests on the house. He has the builder's perspective."

Source: [Unit Testing versus Functional Tests](#)

I'm fond of saying "As developers, we fail in two ways: we build the thing wrong, or we build the wrong thing." Unit tests ensure you are building the thing right; functional tests ensure you are building the right thing.

Since functional tests operate at the system level, they may require some degree of UI automation. Like integration tests, they usually work with some kind of test infrastructure as well. This activity makes them slower and more brittle than unit and integration tests. You should have only as many functional tests as you need to be confident the system is behaving as users expect.

Testing Pyramid

Martin Fowler wrote about the testing pyramid, an example of which is shown in Figure 9-1.

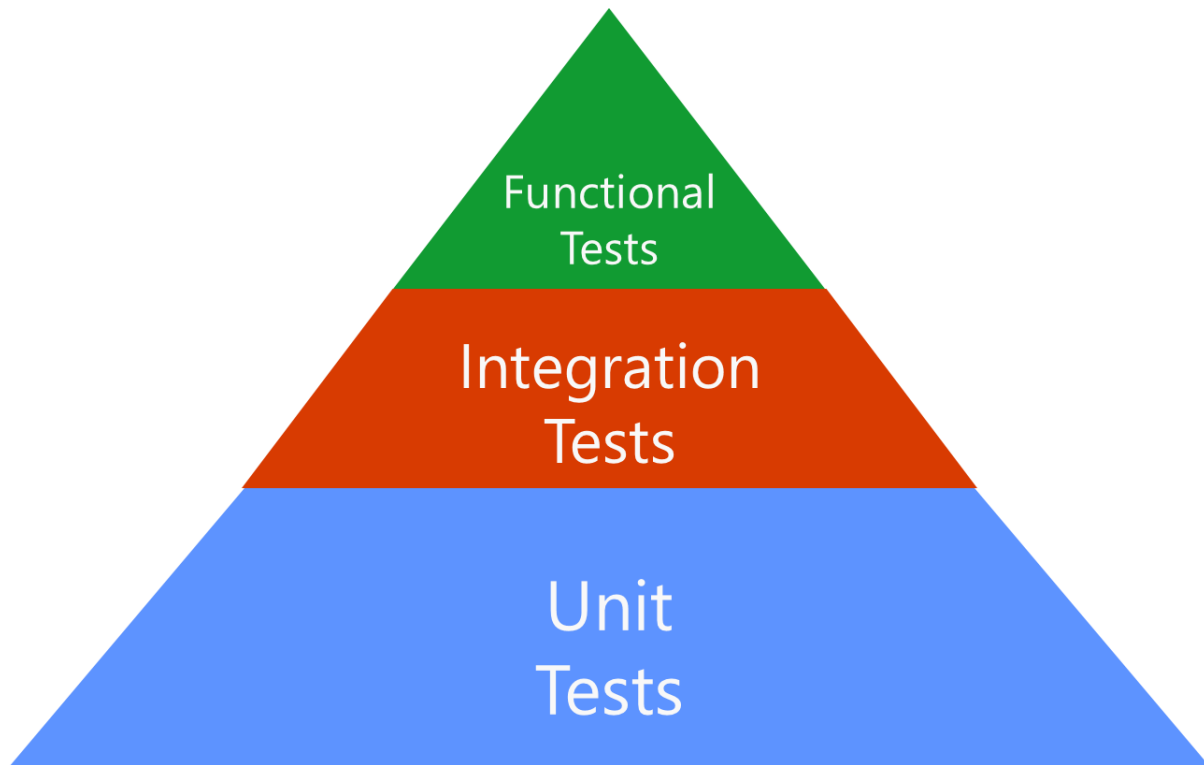


Figure 9-1. Testing Pyramid

The different layers of the pyramid, and their relative sizes, represent different kinds of tests and how many you should write for your application. As you can see, the recommendation is to have a large base of unit tests, supported by a smaller layer of integration tests, with an even smaller layer of functional tests. Each layer should ideally only have tests in it that cannot be performed adequately at a lower layer. Keep the testing pyramid in mind when you are trying to decide which kind of test you need for a particular scenario.

What to test

A common problem for developers who are inexperienced with writing automated tests is coming up with what to test. A good starting point is to test conditional logic. Anywhere you have a method with behavior that changes based on a conditional statement (if-else, switch, and so on), you should be able to come up with at least a couple of tests that confirm the correct behavior for certain conditions. If your code has error conditions, it's good to write at least one test for the "happy path" through the code (with no errors), and at least one test for the "sad path" (with errors or atypical results) to confirm your application behaves as expected in the face of errors. Finally, try to focus on testing things that can fail, rather than focusing on metrics like code coverage. More code coverage is better than less, generally. However, writing a few more tests of a complex and business-critical method is usually a better use of time than writing tests for auto-properties just to improve test code coverage metrics.

Organizing test projects

Test projects can be organized however works best for you. It's a good idea to separate tests by type (unit test, integration test) and by what they are testing (by project, by namespace). Whether this separation consists of folders within a single test project, or multiple test projects, is a design decision. One project is simplest, but for large projects with many tests, or in order to more easily run different sets of tests, you might want to have several different test projects. Many teams organize test projects based on the project they are testing, which for applications with more than a few projects can result in a large number of test projects, especially if you still break these down according to what kind of tests are in each project. A compromise approach is to have one project per kind of test, per application, with folders inside the test projects to indicate the project (and class) being tested.

A common approach is to organize the application projects under a 'src' folder, and the application's test projects under a parallel 'tests' folder. You can create matching solution folders in Visual Studio, if you find this organization useful.

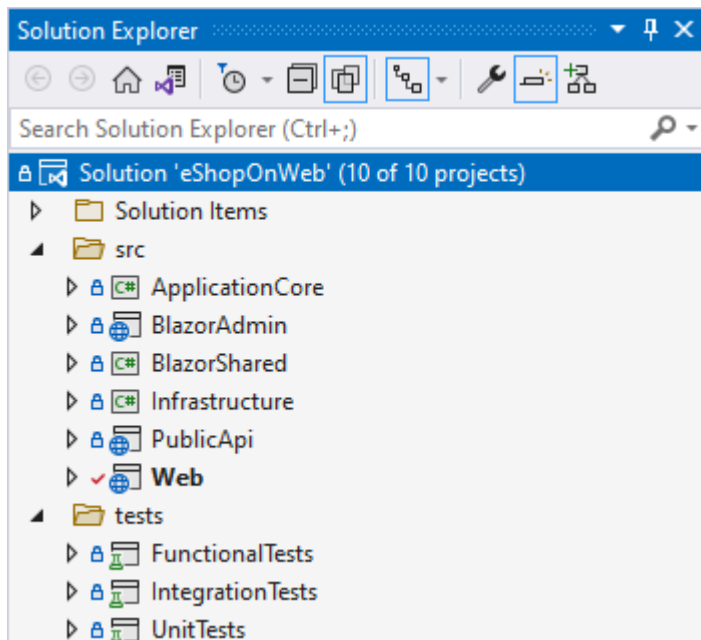


Figure 9-2. Test organization in your solution

You can use whichever test framework you prefer. The xUnit framework works well and is what all of the ASP.NET Core and EF Core tests are written in. You can add an xUnit test project in Visual Studio using the template shown in Figure 9-3, or from the CLI using `dotnet new xunit`.

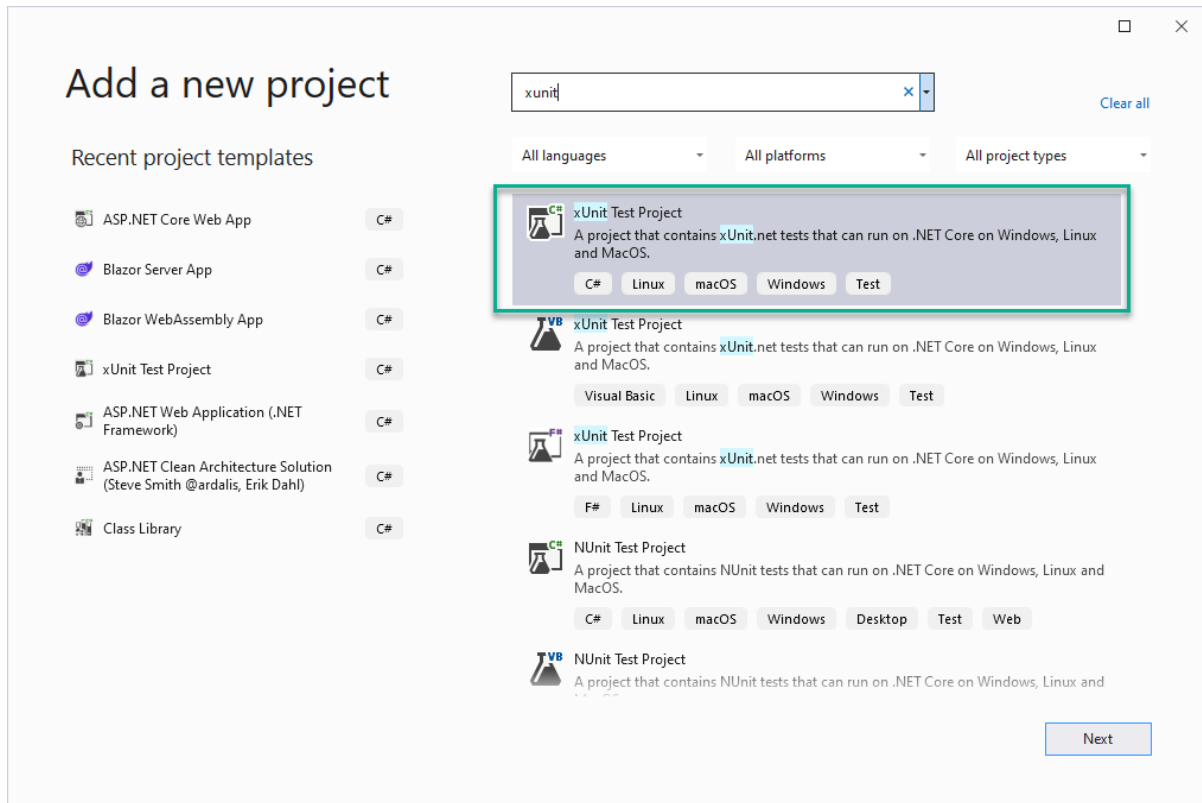


Figure 9-3. Add an xUnit Test Project in Visual Studio

Test naming

Name your tests in a consistent fashion, with names that indicate what each test does. One approach I've had great success with is to name test classes according to the class and method they are testing. This approach results in many small test classes, but it makes it extremely clear what each test is responsible for. With the test class name set up, to identify the class and method to be tested, the test method name can be used to specify the behavior being tested. This name should include the expected behavior and any inputs or assumptions that should yield this behavior. Some example test names:

- `CatalogControllerGetImage.CallsImageServiceWithId`
- `CatalogControllerGetImage.LogsWarningGivenImageMissingException`
- `CatalogControllerGetImage.ReturnsFileResultWithBytesGivenSuccess`
- `CatalogControllerGetImage.ReturnsNotFoundResultGivenImageMissingException`

A variation of this approach ends each test class name with "Should" and modifies the tense slightly:

- `CatalogControllerGetImageShould.CallImageServiceWithId`
- `CatalogControllerGetImageShould.LogWarningGivenImageMissingException`

Some teams find the second naming approach clearer, though slightly more verbose. In any case, try to use a naming convention that provides insight into test behavior, so that when one or more tests fail, it's obvious from their names what cases have failed. Avoid naming your tests vaguely, such as `ControllerTests.Test1`, as these names offer no value when you see them in test results.

If you follow a naming convention like the one above that produces many small test classes, it's a good idea to further organize your tests using folders and namespaces. Figure 9-4 shows one approach to organizing tests by folder within several test projects.

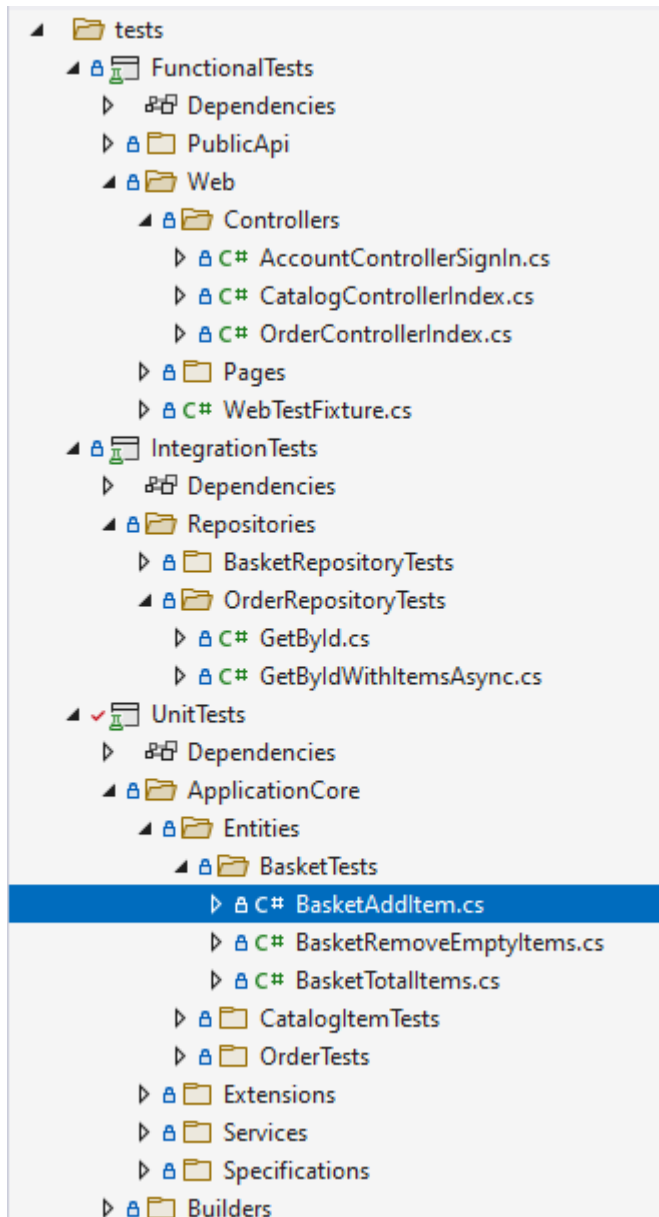


Figure 9-4. Organizing test classes by folder based on class being tested.

If a particular application class has many methods being tested (and thus many test classes), it may make sense to place these classes in a folder corresponding to the application class. This organization is no different than how you might organize files into folders elsewhere. If you have more than three

or four related files in a folder containing many other files, it's often helpful to move them into their own subfolder.

Unit testing ASP.NET Core apps

In a well-designed ASP.NET Core application, most of the complexity and business logic will be encapsulated in business entities and a variety of services. The ASP.NET Core MVC app itself, with its controllers, filters, viewmodels, and views, should require few unit tests. Much of the functionality of a given action lies outside the action method itself. Testing whether routing or global error handling work correctly cannot be done effectively with a unit test. Likewise, any filters, including model validation and authentication and authorization filters, cannot be unit tested with a test targeting a controller's action method. Without these sources of behavior, most action methods should be trivially small, delegating the bulk of their work to services that can be tested independent of the controller that uses them.

Sometimes you'll need to refactor your code in order to unit test it. Frequently this activity involves identifying abstractions and using dependency injection to access the abstraction in the code you'd like to test, rather than coding directly against infrastructure. For example, consider this easy action method for displaying images:

```
[HttpGet("[controller]/pic/{id}")]
public IActionResult GetImage(int id)
{
    var contentRoot = _env.ContentRootPath + "//Pics";
    var path = Path.Combine(contentRoot, id + ".png");
    Byte[] b = System.IO.File.ReadAllBytes(path);
    return File(b, "image/png");
}
```

Unit testing this method is made difficult by its direct dependency on `System.IO.File`, which it uses to read from the file system. You can test this behavior to ensure it works as expected, but doing so with real files is an integration test. It's worth noting you can't unit test this method's route—you'll see how to do this testing with a functional test shortly.

If you can't unit test the file system behavior directly, and you can't test the route, what is there to test? Well, after refactoring to make unit testing possible, you may discover some test cases and missing behavior, such as error handling. What does the method do when a file isn't found? What should it do? In this example, the refactored method looks like this:

```
[HttpGet("[controller]/pic/{id}")]
public IActionResult GetImage(int id)
{
    byte[] imageBytes;
    try
    {
        imageBytes = _imageService.GetImageBytesById(id);
    }
    catch (CatalogImageMissingException ex)
    {
        _logger.LogWarning($"No image found for id: {id}");
        return NotFound();
    }
}
```

```
return File(imageBytes, "image/png");  
}
```

_logger and _imageService are both injected as dependencies. Now you can test that the same ID that is passed to the action method is passed to _imageService, and that the resulting bytes are returned as part of the FileResult. You can also test that error logging is happening as expected, and that a NotFound result is returned if the image is missing, assuming this behavior is important application behavior (that is, not just temporary code the developer added to diagnose an issue). The actual file logic has moved into a separate implementation service, and has been augmented to return an application-specific exception for the case of a missing file. You can test this implementation independently, using an integration test.

In most cases, you'll want to use global exception handlers in your controllers, so the amount of logic in them should be minimal and probably not worth unit testing. Do most of your testing of controller actions using functional tests and the TestServer class described below.

Integration testing ASP.NET Core apps

Most of the integration tests in your ASP.NET Core apps should be testing services and other implementation types defined in your Infrastructure project. For example, you could [test that EF Core was successfully updating and retrieving the data that you expect](#) from your data access classes residing in the Infrastructure project. The best way to test that your ASP.NET Core MVC project is behaving correctly is with functional tests that run against your app running in a test host.

Functional testing ASP.NET Core apps

For ASP.NET Core applications, the TestServer class makes functional tests fairly easy to write. You configure a TestServer using a WebHostBuilder (or HostBuilder) directly (as you normally do for your application), or with the WebApplicationFactory type (available since version 2.1). Try to match your test host to your production host as closely as possible, so your tests exercise behavior similar to what the app will do in production. The WebApplicationFactory class is helpful for configuring the TestServer's ContentRoot, which is used by ASP.NET Core to locate static resource like Views.

You can create simple functional tests by creating a test class that implements `IClassFixture<WebApplicationFactory<TEntryPoint>>`, where `TEntryPoint` is your web application's Startup class. With this interface in place, your test fixture can create a client using the factory's `CreateClient` method:

```
public class BasicWebTests : IClassFixture<WebApplicationFactory<Program>>  
{  
    protected readonly HttpClient _client;  
  
    public BasicWebTests(WebApplicationFactory<Program> factory)  
    {  
        _client = factory.CreateClient();  
    }  
}
```

```
// write tests that use _client
}
```

Tip

If you're using minimal API configuration in your *Program.cs* file, by default the class will be declared internal and won't be accessible from the test project. You can choose any other instance class in your web project instead, or add this to your *Program.cs* file:

```
[{custom-style=Code}`csharp // Make the implicit Program class public so test projects can access it
public partial class Program { } ]`
```

Frequently, you'll want to perform some additional configuration of your site before each test runs, such as configuring the application to use an in-memory data store and then seeding the application with test data. To achieve this functionality, create your own subclass of `WebApplicationFactory<TEnterpoint>` and override its `ConfigureWebHost` method. The example below is from the `eShopOnWeb FunctionalTests` project and is used as part of the tests on the main web application.

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.Testing;
using Microsoft.EntityFrameworkCore;
using Microsoft.eShopWeb.Infrastructure.Data;
using Microsoft.eShopWeb.Infrastructure.Identity;
using Microsoft.eShopWeb.Web;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using System;

namespace Microsoft.eShopWeb.FunctionalTests.Web;
public class WebTestFixture : WebApplicationFactory<Startup>
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.UseEnvironment("Testing");

        builder.ConfigureServices(services =>
        {
            services.AddEntityFrameworkInMemoryDatabase();

            // Create a new service provider.
            var provider = services
                .AddEntityFrameworkInMemoryDatabase()
                .BuildServiceProvider();

            // Add a database context (ApplicationDbContext) using an in-memory
            // database for testing.
            services.AddDbContext<CatalogContext>(options =>
            {
                options.UseInMemoryDatabase("InMemoryDbForTesting");
                options.UseInternalServiceProvider(provider);
            });

            services.AddDbContext<AppIdentityDbContext>(options =>
            {
```

```

        options.UseInMemoryDatabase("Identity");
        options.UseInternalServiceProvider(provider);
    });

    // Build the service provider.
    var sp = services.BuildServiceProvider();

    // Create a scope to obtain a reference to the database
    // context (ApplicationDbContext).
    using (var scope = sp.CreateScope())
    {
        var scopedServices = scope.ServiceProvider;
        var db = scopedServices.GetRequiredService<CatalogContext>();
        var loggerFactory = scopedServices.GetRequiredService<ILoggerFactory>();

        var logger = scopedServices
            .GetRequiredService<ILogger<WebTestFixture>>();

        // Ensure the database is created.
        db.Database.EnsureCreated();

        try
        {
            // Seed the database with test data.
            CatalogContextSeed.SeedAsync(db, loggerFactory).Wait();

            // seed sample user data
            var userManager =
scopedServices.GetRequiredService<UserManager<ApplicationUser>>();
            var roleManager = scopedServices.GetRequiredService<RoleManager<IdentityRole>>();
            AppIdentityDbContextSeed.SeedAsync(userManager, roleManager).Wait();
        }
        catch (Exception ex)
        {
            logger.LogError(ex, $"An error occurred seeding the " +
                "database with test messages. Error: {ex.Message}");
        }
    }
});
}
}

```

Tests can make use of this custom `WebApplicationFactory` by using it to create a client and then making requests to the application using this client instance. The application will have data seeded that can be used as part of the test's assertions. The following test verifies that the home page of the `eShopOnWeb` application loads correctly and includes a product listing that was added to the application as part of the seed data.

```

using Microsoft.eShopWeb.FunctionalTests.Web;
using System.Net.Http;
using System.Threading.Tasks;
using Xunit;

namespace Microsoft.eShopWeb.FunctionalTests.WebRazorPages;
[Collection("Sequential")]
public class HomePageOnGet : IClassFixture<WebTestFixture>
{
    public HomePageOnGet(WebTestFixture factory)
    {
    }
}

```

```

    Client = factory.CreateClient();
}

public HttpClient Client { get; }

[Fact]
public async Task ReturnsHomePageWithProductListing()
{
    // Arrange & Act
    var response = await Client.GetAsync("/");
    response.EnsureSuccessStatusCode();
    var stringResponse = await response.Content.ReadAsStringAsync();

    // Assert
    Assert.Contains(".NET Bot Black Sweatshirt", stringResponse);
}
}

```

This functional test exercises the full ASP.NET Core MVC / Razor Pages application stack, including all middleware, filters, and binders that may be in place. It verifies that a given route ("/") returns the expected success status code and HTML output. It does so without setting up a real web server, and avoids much of the brittleness that using a real web server for testing can experience (for example, problems with firewall settings). Functional tests that run against TestServer are usually slower than integration and unit tests, but are much faster than tests that would run over the network to a test web server. Use functional tests to ensure your application's front-end stack is working as expected. These tests are especially useful when you find duplication in your controllers or pages and you address the duplication by adding filters. Ideally, this refactoring won't change the behavior of the application, and a suite of functional tests will verify this is the case.

References – Test ASP.NET Core MVC apps

- **Testing in ASP.NET Core**
<https://learn.microsoft.com/aspnet/core/testing/>
- **Unit Test Naming Convention**
<https://ardalis.com/unit-test-naming-convention>
- **Testing EF Core**
<https://learn.microsoft.com/ef/core/miscellaneous/testing/>
- **Integration tests in ASP.NET Core**
<https://learn.microsoft.com/aspnet/core/test/integration-tests>

Development process for Azure

"With the cloud, individuals and small businesses can snap their fingers and instantly set up enterprise-class services."

- Roy Stephan

Vision

Develop well-designed ASP .NET Core applications the way you like, using Visual Studio or the dotnet CLI and Visual Studio Code or your editor of choice.

Development environment for ASP.NET Core apps

Development tools choices: IDE or editor

Whether you prefer a full and powerful IDE or a lightweight and agile editor, Microsoft has you covered when developing ASP.NET Core applications.

Visual Studio 2022. Visual Studio 2022 is the best-in-class IDE for developing applications for ASP.NET Core. It offers a host of features that increase developer productivity. You can use it to develop the application, then analyze its performance and other characteristics. The integrated debugger lets you pause code execution and step back and forth through code on the fly as it's running. Its support for hot reloads allows you to continue working with your app where you left off, even after making code changes, without having to restart the app. The built-in test runner lets you organize your tests and their results and can even perform live unit testing while you're coding. Using Live Share, you can collaborate in real-time with other developers, sharing your code session seamlessly over the network. And when you're ready, Visual Studio includes everything you need to publish your application to Azure or wherever you might host it.

[Download Visual Studio 2022](#)

Visual Studio Code and dotnet CLI (Cross-Platform Tools for Mac, Linux, and Windows). If you prefer a lightweight and cross-platform editor supporting any development language, you can use Microsoft Visual Studio Code and the dotnet CLI. These products provide a simple yet robust experience that

streamlines the developer workflow. Additionally, Visual Studio Code supports extensions for C# and web development, providing intellisense and shortcut-tasks within the editor.

[Download the .NET SDK](#)

[Download Visual Studio Code](#)

Development workflow for Azure-hosted ASP.NET Core apps

The application development lifecycle starts from each developer's machine, coding the app using their preferred language and testing it locally. Developers may choose their preferred source control system and can configure Continuous Integration (CI) and/or Continuous Delivery/Deployment (CD) using a build server or based on built-in Azure features.

To get started with developing an ASP.NET Core application using CI/CD, you can use Azure DevOps Services or your organization's own Team Foundation Server (TFS). GitHub Actions provide another option for easily building and deploying apps to Azure, for apps whose code is hosted on GitHub.

Initial setup

To create a release pipeline for your app, you need to have your application code in source control. Set up a local repository and connect it to a remote repository in a team project. Follow these instructions:

- [Share your code with Git and Visual Studio](#) or
- [Share your code with TFVC and Visual Studio](#)

Create an Azure App Service where you'll deploy your application. Create a Web App by going to the App Services blade on the Azure portal. Click +Add, select the Web App template, click Create, and provide a name and other details. The web app will be accessible from {name}.azurewebsites.net.

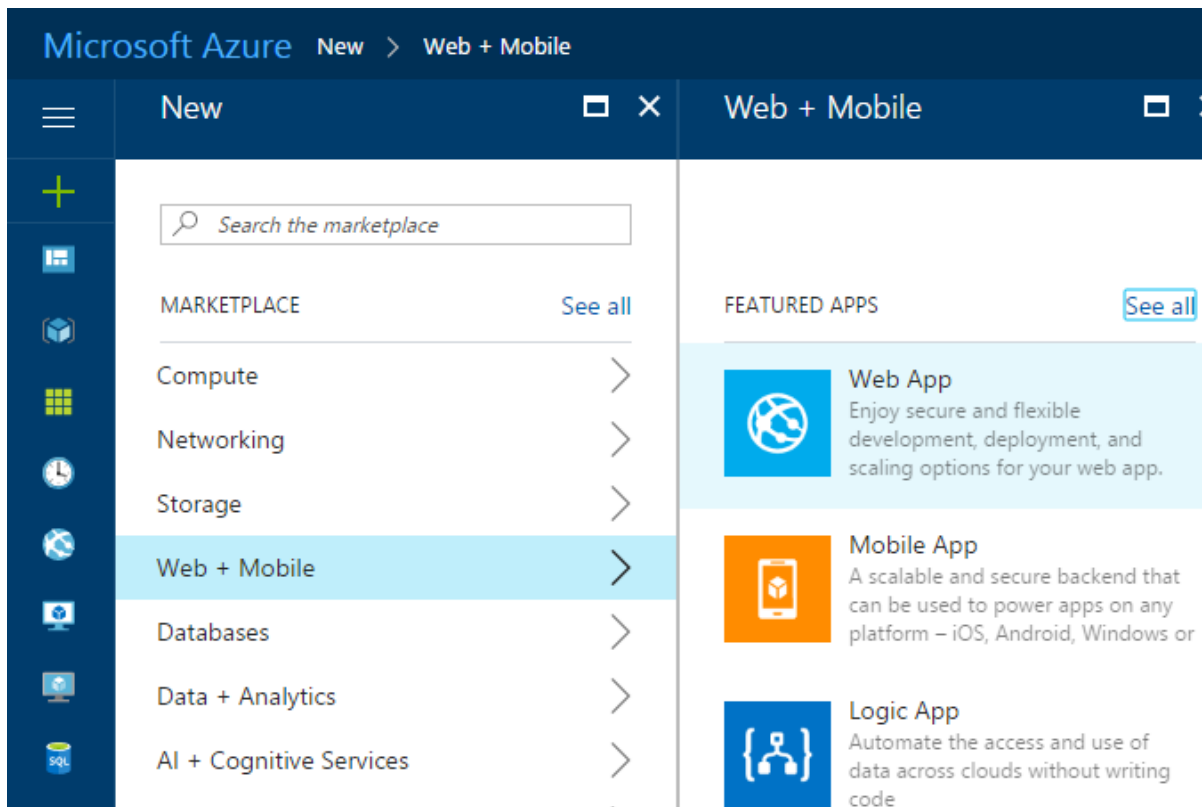


Figure 10-1. Creating a new Azure App Service Web App in the Azure Portal.

Your CI build process will perform an automated build whenever new code is committed to the project's source control repository. This process gives you immediate feedback that the code builds (and, ideally, passes automated tests) and can potentially be deployed. This CI build will produce a web deploy package artifact and publish it for consumption by your CD process.

[Define your CI build process](#)

Be sure to enable continuous integration so the system will queue a build whenever someone on your team commits new code. Test the build and verify that it is producing a web deploy package as one of its artifacts.

When a build succeeds, your CD process will deploy the results of your CI build to your Azure web app. To configure this step, you create and configure a *Release*, which will deploy to your Azure App Service.

[Deploy an Azure web app](#)

Once your CI/CD pipeline is configured, you can easily make updates to your web app and commit them to source control to have them deployed.

Workflow for developing Azure-hosted ASP.NET Core applications

Once you have configured your Azure account and your CI/CD process, developing Azure-hosted ASP.NET Core applications is simple. The following are the basic steps you usually take when building an ASP.NET Core app, hosted in Azure App Service as a Web App, as illustrated in Figure 10-2.

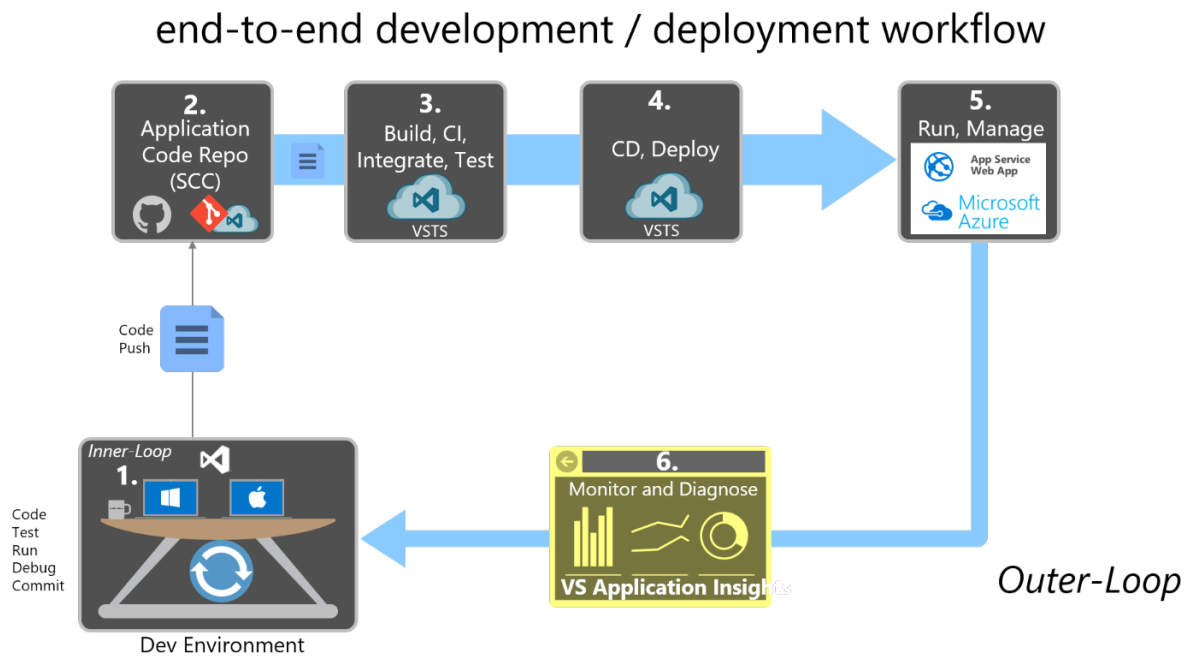


Figure 10-2. Step-by-step workflow for building ASP.NET Core apps and hosting them in Azure

Step 1. Local dev environment inner loop

Developing your ASP.NET Core application for deployment to Azure is no different from developing your application otherwise. Use the local development environment you're comfortable with, whether that's Visual Studio 2019 or the dotnet CLI and Visual Studio Code or your preferred editor. You can write code, run and debug your changes, run automated tests, and make local commits to source control until you're ready to push your changes to your shared source control repository.

Step 2. Application code repository

Whenever you're ready to share your code with your team, you should push your changes from your local source repository to your team's shared source repository. If you've been working in a custom branch, this step usually involves merging your code into a shared branch (perhaps by means of a [pull request](#)).

Step 3. Build Server: Continuous integration. build, test, package

A new build is triggered on the build server whenever a new commit is made to the shared application code repository. As part of the CI process, this build should fully compile the application and run automated tests to confirm everything is working as expected. The end result of the CI process should be a packaged version of the web app, ready for deployment.

Step 4. Build Server: Continuous delivery

Once a build has succeeded, the CD process will pick up the build artifacts produced. This process will include a web deploy package. The build server will deploy this package to Azure App Service,

replacing any existing service with the newly created one. Typically this step targets a staging environment, but some applications deploy directly to production through a CD process.

Step 5. Azure App Service Web App

Once deployed, the ASP.NET Core application runs within the context of an Azure App Service Web App. This Web App can be monitored and further configured using the Azure Portal.

Step 6. Production monitoring and diagnostics

While the Web App is running, you can monitor the health of the application and collect diagnostics and user behavior data. Application Insights is included in Visual Studio, and offers automatic instrumentation for ASP.NET apps. It can provide you with information on usage, exceptions, requests, performance, and logs.

References

Build and Deploy Your ASP.NET Core App to Azure

<https://learn.microsoft.com/azure/devops/build-release/apps/aspnet/build-aspnet-core>

Azure hosting recommendations for ASP.NET Core web apps

“Line-of-business leaders everywhere are bypassing IT departments to get applications from the cloud (also known as SaaS) and paying for them like they would a magazine subscription. And when the service is no longer required, they can cancel the subscription with no equipment left unused in the corner.”

- Daryl Plummer, Gartner analyst

Whatever your application’s needs and architecture, Microsoft Azure can support it. Your hosting needs can be as simple as a static website or a sophisticated application made up of dozens of services. For ASP.NET Core monolithic web applications and supporting services, there are several well-known configurations that are recommended. The recommendations on this article are grouped based on the kind of resource to be hosted, whether full applications, individual processes, or data.

Web applications

Web applications can be hosted with:

- App Service Web Apps
- Containers (several options)
- Virtual Machines (VMs)

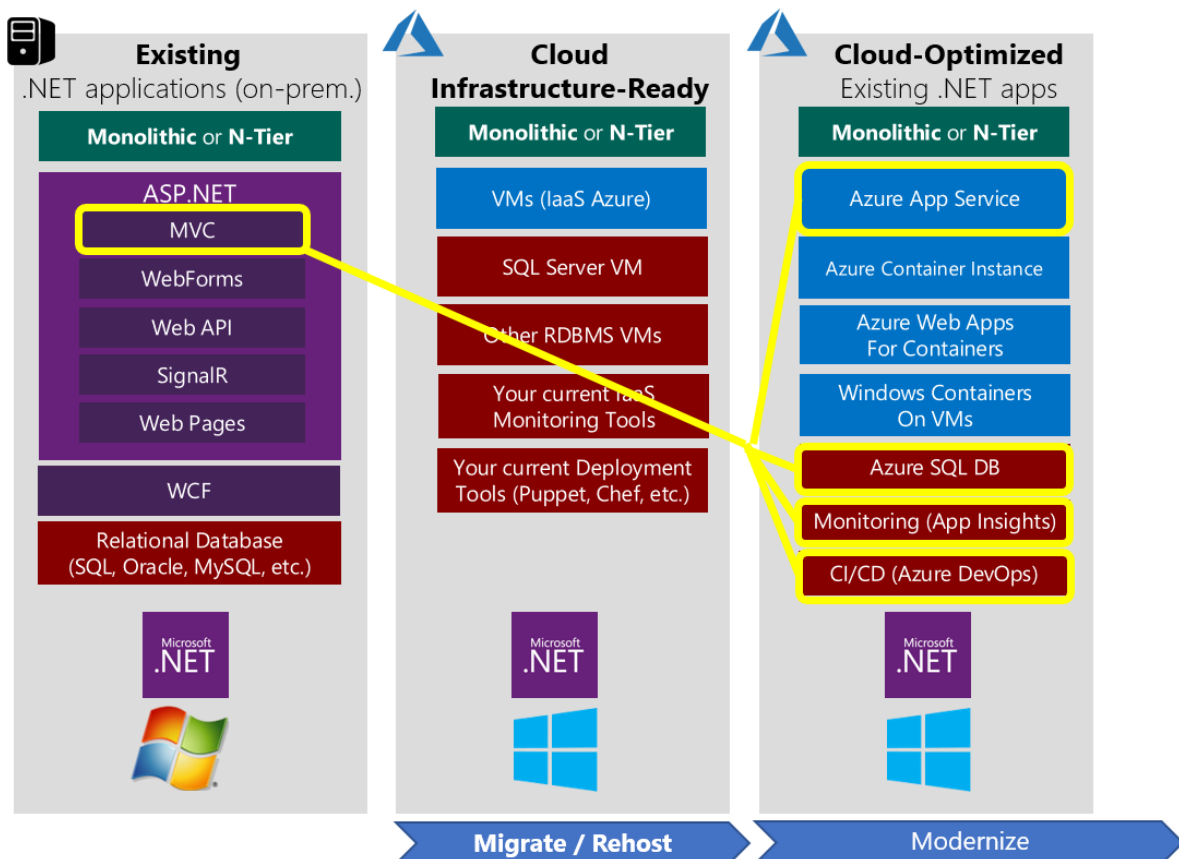
Of these, App Service Web Apps is the recommended approach for most scenarios, including simple container-based apps. For microservice architectures, consider a container-based approach. If you need more control over the machines running your application, consider Azure Virtual Machines.

App Service Web Apps

App Service Web Apps offers a fully managed platform optimized for hosting web applications. It’s a platform as a service (PaaS) offering that lets you focus on your business logic, while Azure takes care of the infrastructure needed to run and scale the app. Some key features of App Service Web Apps:

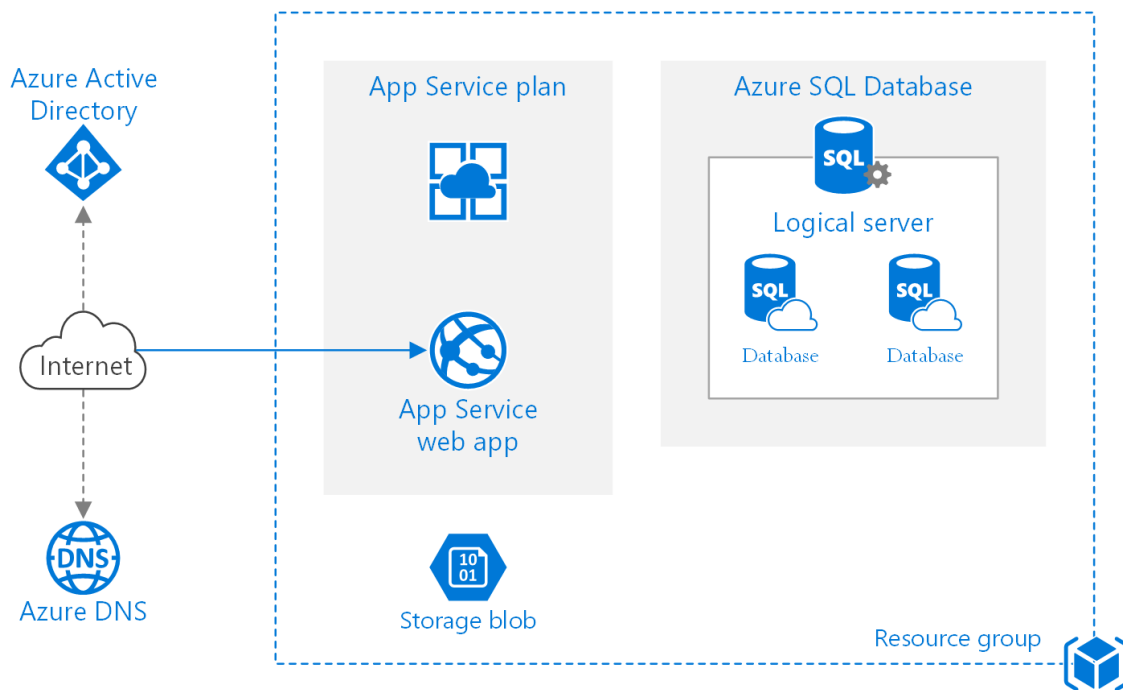
- DevOps optimization (continuous integration and delivery, multiple environments, A/B testing, scripting support).
- Global scale and high availability.
- Connections to SaaS platforms and your on-premises data.
- Security and compliance.
- Visual Studio integration.

Azure App Service is the best choice for most web apps. Deployment and management are integrated into the platform, sites can scale quickly to handle high traffic loads, and the built-in load balancing and traffic manager provide high availability. You can move existing sites to Azure App Service easily with an online migration tool. You can use an open-source app from the Web Application Gallery, or create a new site using the framework and tools of your choice. The WebJobs feature makes it easy to add background job processing to your App Service web app. If you have an existing ASP.NET application hosted on-premises using a local database, there's a clear path to migrate. You can use App Service Web App with an Azure SQL Database (or secure access to your on-premises database server, if preferred).



In most cases, moving from a locally hosted ASP.NET app to an App Service Web App is a straightforward process. Little or no modification should be required of the app itself, and it can quickly start to take advantage of the many features that Azure App Service Web Apps offer.

In addition to apps that are not optimized for the cloud, Azure App Service Web Apps are an excellent solution for many simple monolithic (non-distributed) applications, such as many ASP.NET Core apps. In this approach, the architecture is basic and simple to understand and manage:

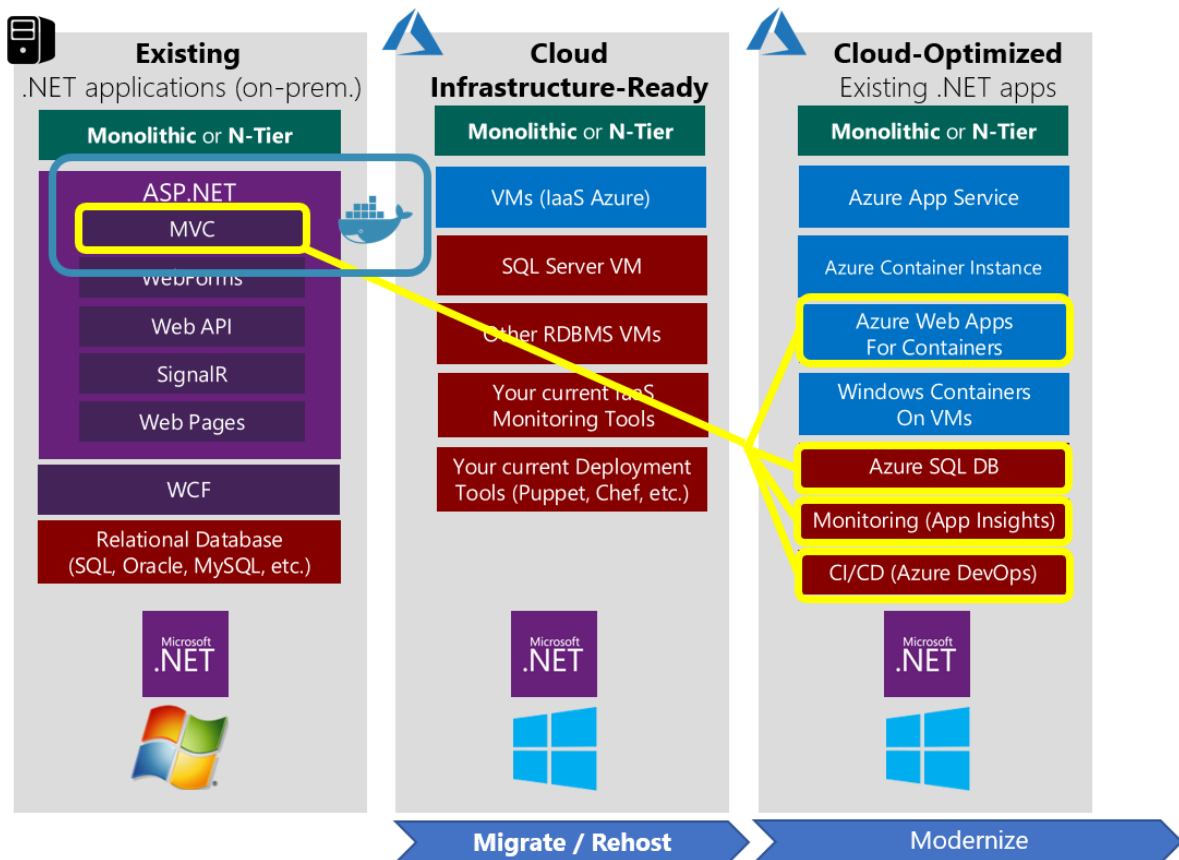


A small number of resources in a single resource group is typically sufficient to manage such an app. Apps that are typically deployed as a single unit, rather than those apps that are made up of many separate processes, are good candidates for this [basic architectural approach](#). Though architecturally simple, this approach still allows the hosted app to scale both up (more resources per node) and out (more hosted nodes) to meet any increase in demand. With autoscale, the app can be configured to automatically adjust the number of nodes hosting the app based on demand and average load across nodes.

App Service Web Apps for Containers

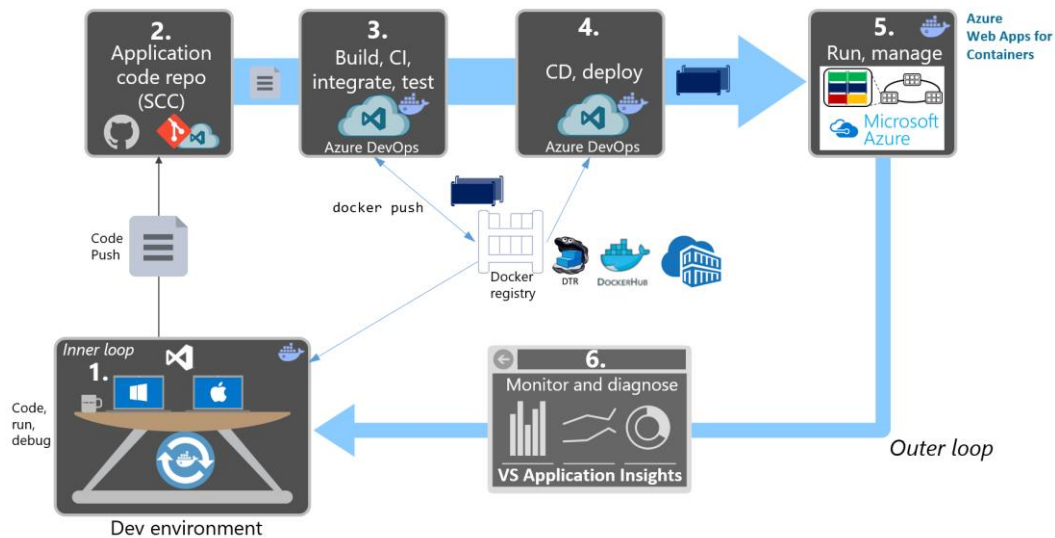
In addition to support for hosting web apps directly, [App Service Web Apps for Containers](#) can be used to run containerized applications on Windows and Linux. Using this service, you can easily deploy and run containerized applications that can scale with your business. The apps have all of the features of App Service Web Apps listed above. In addition, Web Apps for Containers supports streamlined CI/CD with Docker Hub, Azure Container Registry, and GitHub. You can use Azure DevOps to define build and deployment pipelines that publish changes to a registry. These changes can then be tested in a staging environment and automatically deployed to production using deployment slots, allowing zero-downtime upgrades. Rolling back to previous versions can be done just as easily.

There are a few scenarios where Web Apps for Containers makes the most sense. If you have existing apps that you can containerize, whether in Windows or Linux containers, you can host these easily using this toolset. Just publish your container and then configure Web Apps for Containers to pull the latest version of that image from your registry of choice. This is a “lift and shift” approach to migrating from classic app hosting models to a cloud-optimized model.



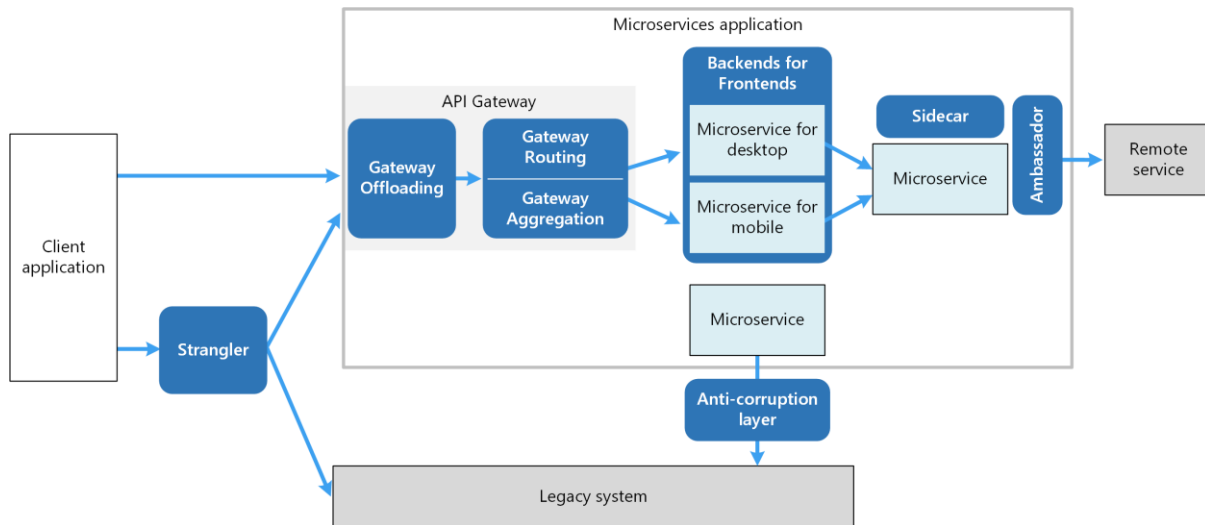
This approach also works well if your development team is able to move to a container-based development process. The “inner loop” of developing apps with containers includes building the app with containers. Changes made to the code as well as to container configuration are pushed to source control, and an automated build is responsible for publishing new container images to a registry like Docker Hub or Azure Container Registry. These images are then used as the basis for additional development, as well as for deployments to production, as shown in the following diagram:

End to End Docker DevOps Lifecycle Workflow



Developing with containers offers many advantages, especially when containers are used in production. The same container configuration is used to host the app in each environment in which it runs, from the local development machine to build and test systems to production. This approach greatly reduces the likelihood of defects resulting from differences in machine configuration or software versions. Developers can also use whatever tools they're most productive with, including the operating system, since containers can run on any OS. In some cases, distributed applications involving many containers may be very resource-intensive to run on a single development machine. In this scenario, it may make sense to upgrade to using Kubernetes and Azure Dev Spaces, covered in the next section.

As portions of larger applications are broken up into their own smaller, independent *microservices*, additional design patterns can be used to improve app behavior. Instead of working directly with individual services, an *API gateway* can simplify access and decouple the client from its back end. Having separate service back ends for different front ends also allows services to evolve in concert with their consumers. Common services can be accessed via a separate *sidecar* container, which might include common client connectivity libraries using the *ambassador* pattern.



[Learn more about design patterns to consider when building microservice-based systems.](#)

Azure Kubernetes Service

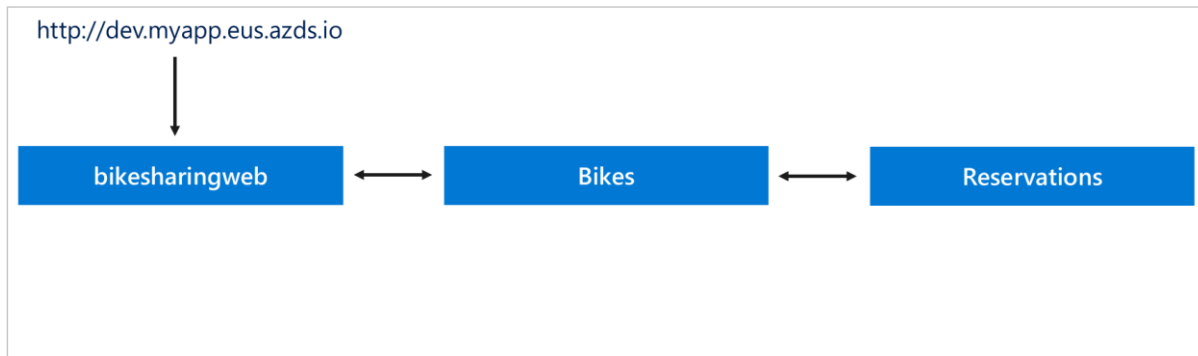
Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on-demand, without taking your applications offline.

AKS reduces the complexity and operational overhead of managing a Kubernetes cluster by offloading much of that responsibility to Azure. As a hosted Kubernetes service, Azure handles critical tasks like health monitoring and maintenance for you. Also, you pay only for the agent nodes within your clusters, not for the masters. As a managed Kubernetes service, AKS provides:

- Automated Kubernetes version upgrades and patching.
- Easy cluster scaling.
- Self-healing hosted control plane (masters).
- Cost savings - pay only for running agent pool nodes.

With Azure handling the management of the nodes in your AKS cluster, you no longer need to perform many tasks manually, like cluster upgrades. Because Azure handles these critical maintenance tasks for you, AKS doesn't provide direct access (such as with SSH) to the cluster.

Teams who are leveraging AKS can also take advantage of Azure Dev Spaces. Azure Dev Spaces helps teams to focus on the development and rapid iteration of their microservice application by allowing teams to work directly with their entire microservices architecture or application running in AKS. Azure Dev Spaces also provides a way to independently update portions of your microservices architecture in isolation without affecting the rest of the AKS cluster or other developers.



Azure Dev Spaces:

- Minimize local machine setup time and resource requirements
- Allow teams to iterate more rapidly
- Reduce the number of integration environments required by a team
- Remove the need to mock certain services in a distributed system when developing/testing

[Learn more about Azure Dev Spaces](#)

Azure Virtual Machines

If you have an existing application that would require substantial modifications to run in App Service, you could choose Virtual Machines in order to simplify migrating to the cloud. However, correctly configuring, securing, and maintaining VMs requires much more time and IT expertise compared to Azure App Service. If you're considering Azure Virtual Machines, make sure you take into account the ongoing maintenance effort required to patch, update, and manage your VM environment. Azure Virtual Machines is infrastructure as a service (IaaS), while App Service is PaaS. You should also consider whether deploying your app as a Windows Container to Web App for Containers might be a viable option for your scenario.

Logical processes

Individual logical processes that can be decoupled from the rest of the application may be deployed independently to Azure Functions in a "serverless" manner. Azure Functions lets you just write the code you need for a given problem, without worrying about the application or infrastructure to run it. You can choose from a variety of programming languages, including C#, F#, Node.js, Python, and PHP, allowing you to pick the most productive language for the task at hand. Like most cloud-based solutions, you pay only for the amount of time you use, and you can trust Azure Functions to scale up as needed.

Data

Azure offers a wide variety of data storage options, so that your application can use the appropriate data provider for the data in question.

For transactional, relational data, Azure SQL Databases are the best option. For high performance read-mostly data, a Redis cache backed by an Azure SQL Database is a good solution.

Unstructured JSON data can be stored in a variety of ways, from SQL Database columns to Blobs or Tables in Azure Storage, to Azure Cosmos DB. Of these, Azure Cosmos DB offers the best querying functionality, and is the recommended option for large numbers of JSON-based documents that must support querying.

Transient command- or event-based data used to orchestrate application behavior can use Azure Service Bus or Azure Storage Queues. Azure Service Bus offers more flexibility and is the recommended service for non-trivial messaging within and between applications.

Architecture recommendations

Your application's requirements should dictate its architecture. There are many different Azure services available. Choosing the right one is an important decision. Microsoft offers a gallery of reference architectures to help identify typical architectures optimized for common scenarios. You may find a reference architecture that maps closely to your application's requirements, or at least offers a starting point.

Figure 11-1 shows an example reference architecture. This diagram describes a recommended architecture approach for a Sitecore content management system website optimized for marketing.

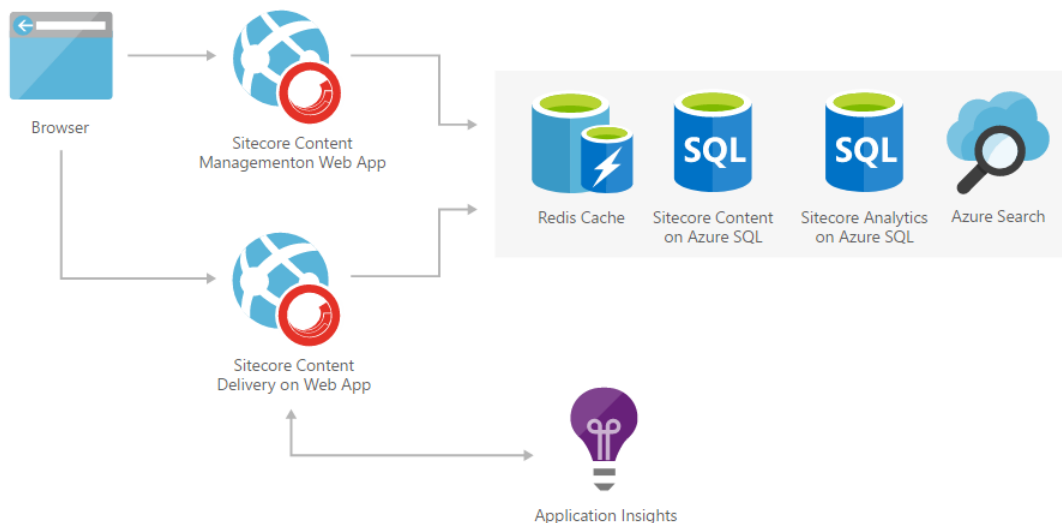


Figure 11-1. Sitecore marketing website reference architecture.

References – Azure hosting recommendations

- Azure Solution Architectures
<https://azure.microsoft.com/solutions/architecture/>

- Azure Basic Web Application Architecture
<https://learn.microsoft.com/azure/architecture/reference-architectures/app-service-web-app/basic-web-app>
- Design Patterns for Microservices
<https://learn.microsoft.com/azure/architecture/microservices/design/patterns>
- Azure Developer Guide
<https://azure.microsoft.com/campaigns/developer-guide/>
- Web Apps overview
<https://learn.microsoft.com/azure/app-service/app-service-web-overview>
- Web App for Containers
<https://azure.microsoft.com/services/app-service/containers/>
- Introduction to Azure Kubernetes Service (AKS)
<https://learn.microsoft.com/azure/aks/intro-kubernetes>