Microsoft

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

Steve Smith
DevIQ

Early Draft

Author:

    Steve Smith, Founder, DevIQ

Participants and reviewers (TBD):

    Cesar de la Torre, Sr. PM, .NET product team, Microsoft

Contents

Introduction

.NET Core and ASP.NET Core offer several advantages over traditional .NET development. You should use .NET Core 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 applications can and do support these requirements, but ASP.NET Core and .NET Core 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, 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 Microsoft Azure, offers numerous competitive advantages over traditional alternatives. ASP.NET Core is optimized for modern web application development practices and cloud hosting scenarios. In this guide, you will learn how to architect your ASP.NET Core applications to best take advantage of these capabilities.

Purpose

This guide provides end-to-end guidance on building monolithic web applications using ASP.NET Core and Azure.

This guide is complementary to the "*Architecting and Developing Containerized and Microservice-based Applications with .NET*" which focuses more on Docker, Microservices, and Deployment of Containers to host enterprise applications.

Architecting and Developing Containerized Microservice Based Apps in .NET
eBook   http://aka.ms/MicroservicesEbook   **Sample Application**   http://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 and/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 Windows 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 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 will help ensure consistent application of architectural patterns and practices.

References
Choosing between .NET Core and .NET Framework for server apps   https://docs.microsoft.com/en-us/dotnet/articles/standard/choosing-core-framework-server

Characteristics of Modern Web Applications

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

Dennis Ritchie

Summary

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
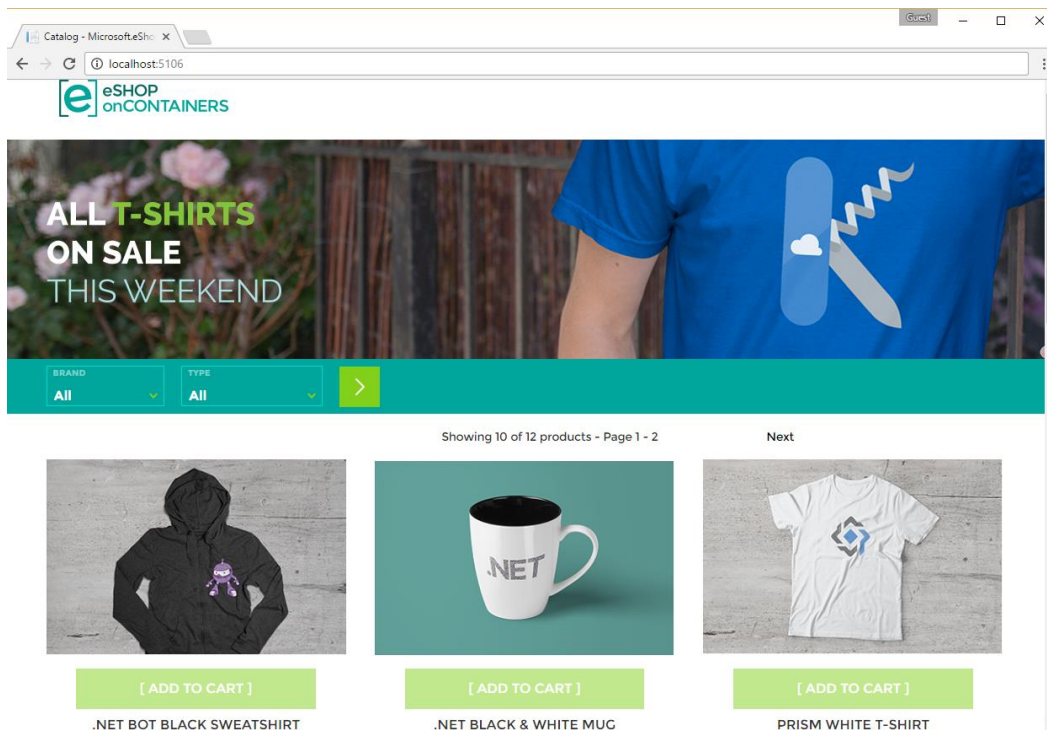
Reference Application
eShopOnWeb https://github.com/dotnet/eShopOnWeb

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 and MacOS as well as Windows. This opens up many new options for both development and deployment of apps built with ASP.NET Core. Docker containers, which typically run Linux today, can host ASP.NET Core applications, allowing them to take advantage of the benefits of containers and microservices (link to microservices and containers ebook).

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, 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 injections makes it easy to swap infrastructure concerns with fake implementations for test purposes. ASP.NET Core also ships 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.

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 which 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.

Many web applications involve a combination of traditional web application behavior (typically for content) and SPAs (for interactivity). ASP.NET Core supports both MVC 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, Windows

Azure has integrated support for git repositories and can automatically deploy updates as they are made to a specified git branch or tag.

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. Windows Azure has great longstanding support for ASP.NET 4.x applications, and many developers are familiar with this platform.

References – Modern Web Applications
Introduction to ASP.NET Core https://docs.microsoft.com/en-us/aspnet/core/ **Six Key Benefits of ASP.NET Core which make it Different and Better** http://www.trigent.com/blog/six-key-benefits-of-asp-net-core-1-0-which-make-it-different-better/ **Testing in ASP.NET Core** https://docs.microsoft.com/en-us/aspnet/core/testing/

Choosing 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

Summary

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 sub-applications within a larger traditional web application.

You should 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 team is unfamiliar with JavaScript or TypeScript development techniques.

You should use a SPA when:

- Your application must expose a rich user interface with many features.
- Your team is familiar with JavaScript and/or TypeScript 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 frameworks than traditional web applications. Configuring automated build and deployment processes and utilizing deployment options like containers are more difficult with SPA applications than traditional web apps.

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

When to choose traditional web apps

The following 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 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 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 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. You should choose a SPA style application if your app's requirements include rich functionality that goes beyond what typical HTML forms offer.

Note that frequently SPAs need to implement features that are built-in to 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
AngularJS https://angularjs.org/ **Comparison of 4 Popular JavaScript Frameworks** https://www.developereconomics.com/feature-comparison-of-4-popular-js-mv-frameworks

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.

Decision table – Traditional Web or SPA

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

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

Architectural Principles

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

Gerald Weinberg

Summary

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 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 helps ensure that the business model is easy to test and can evolve without being tightly coupled to low-level implementation details. 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 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.

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. This produces a direct dependency graph. That is, if module A calls a function in module B, which calls a function in module C, then at compile time A will depend on B which will depend on C, as shown in Figure 4-X.

Direct Dependency Graph

Compile Time

Class A

↓ References

Class B

↓ References

Class C

Run Time

Class A

↓ Control Flow

Class B

↓ Control Flow

Class C
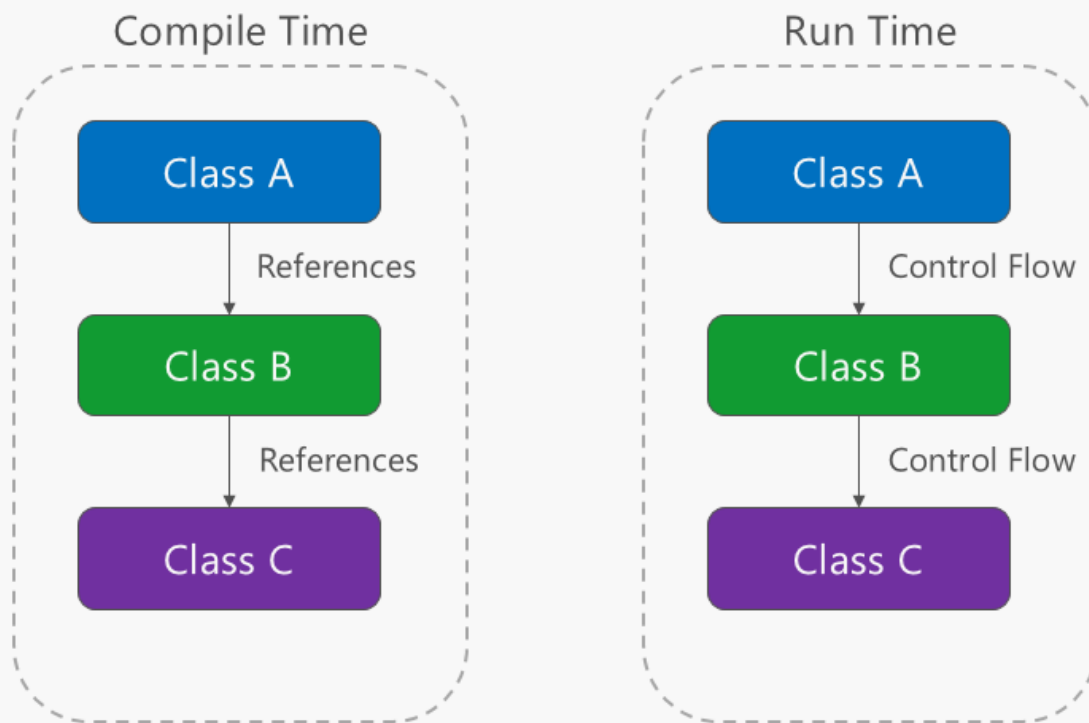
Figure 4-X. *Direct dependency graph.*

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 runtime, 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
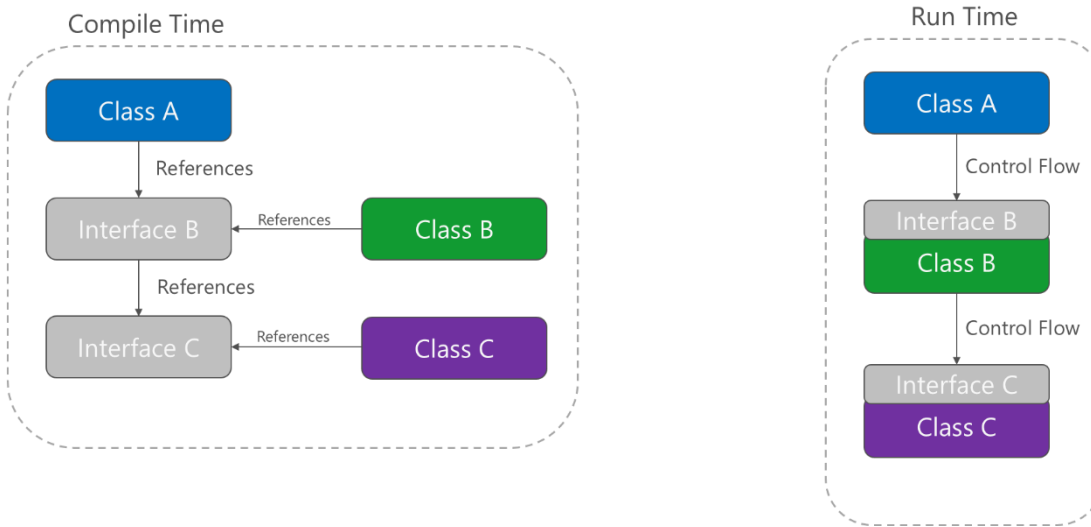
Figure 4-X. *Inverted dependency graph.*

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. 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 which 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 default 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. This 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 runtime.

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 is a frequent source of errors. At some point, a change in requirements will require changing this behavior and the likelihood that at least one instance of the behavior will fail to be updated will result in inconsistent behavior of the system.

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.

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 (e.g. using a Redis cache or Azure DocumentDb 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 default 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 which 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.

References – Modern Web Applications
Separation of Concerns   http://deviq.com/separation-of-concerns/   **Encapsulation**   http://deviq.com/encapsulation/   **Dependency Inversion Principle**   http://deviq.com/dependency-inversion-principle/   **Explicit Dependencies Principle**   http://deviq.com/explicit-dependencies-principle/   **Don't Repeat Yourself**   http://deviq.com/don-t-repeat-yourself/   **Persistence Ignorance**   http://deviq.com/persistence-ignorance/   **Bounded Context**   https://martinfowler.com/bliki/BoundedContext.html   **Domain-Driven Design Fundamentals**   http://bit.ly/PS-DDD   **SOLID Principles of Object Oriented Design**   http://bit.ly/solid-smith

Common Web Application Architectures

"If you think good architecture is expensive, try bad architecture."

Brian Foote and Joseph Yoder

Summary

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 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.

Figure 5-1. A single project ASP.NET Core app
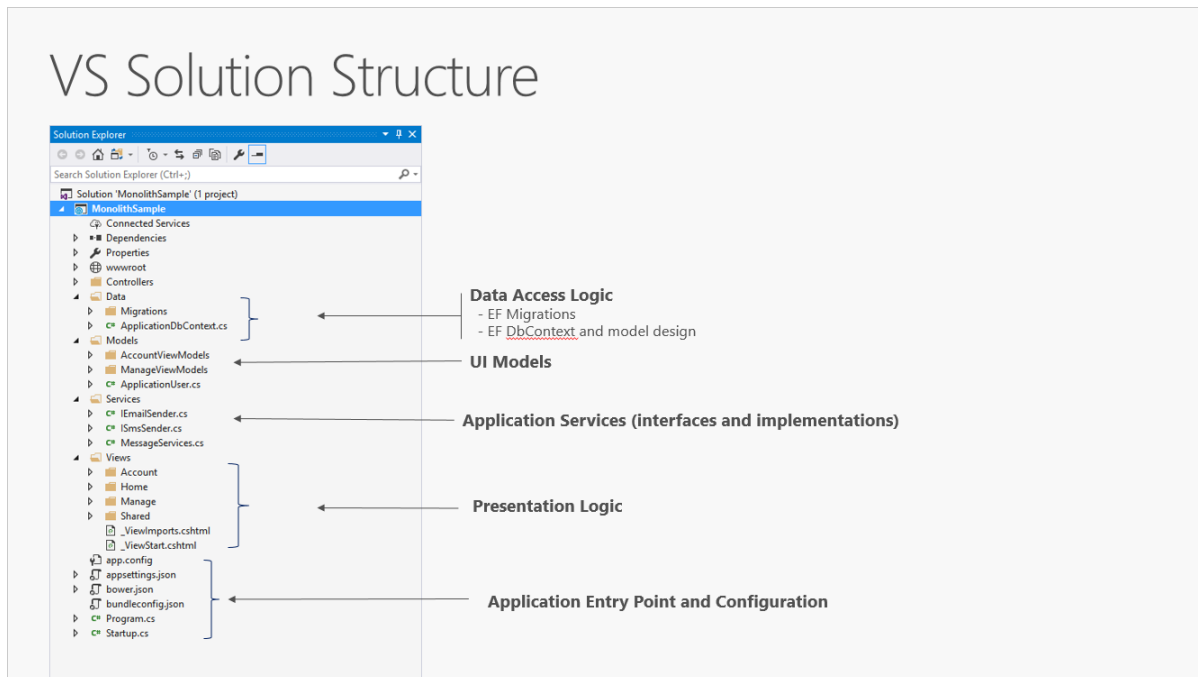
In a single project scenario, separation of concerns is achieved through the use of folders. The default template includes separate folders for MVC pattern responsibilities of Models, Views, and Controllers, as well as additional folders for Data and Services. In this arrangement, presentation details 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. UI concerns (models, views, controllers) reside in multiple folders, which are not 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 is 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.

In order 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 the application up according to its responsibilities or concerns. This 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 DRY principle.

With a layered architecture, applications can enforce restrictions on which layers can communicate with other layers. This 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 typically makes tests much easier to write and much faster to run when compared to running tests again 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 it shown in Figure 5-2.

Figure 5-2. Typical application layers.
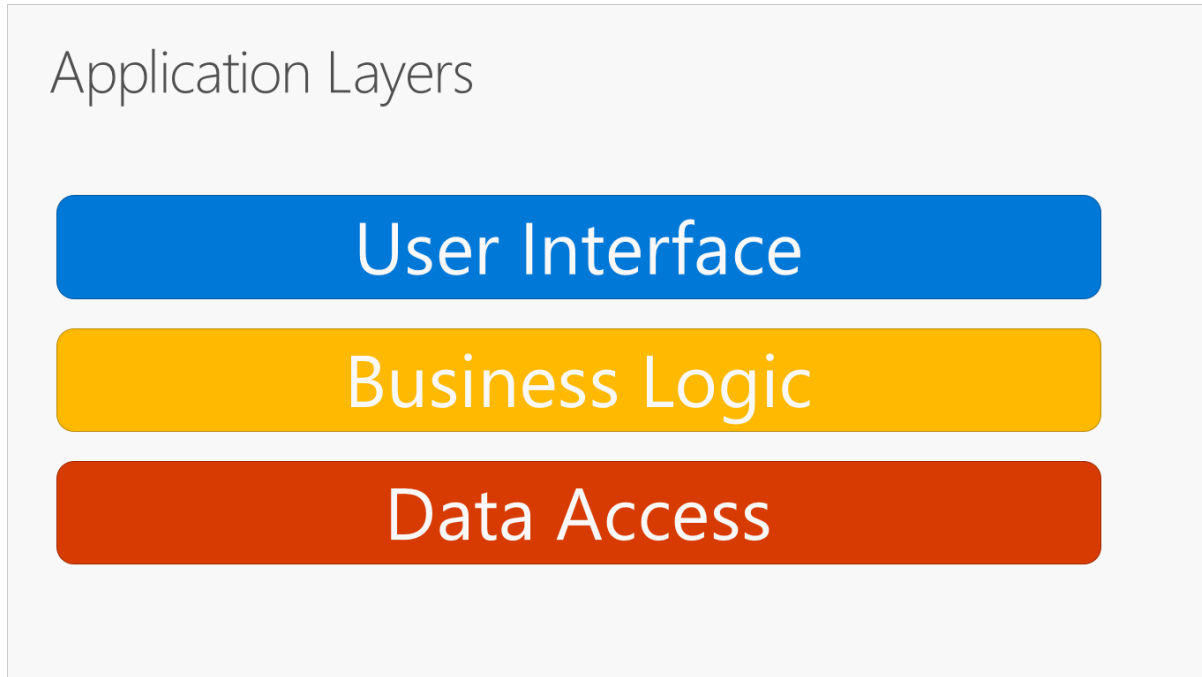
These layers are frequently abbreviated as UI, BLL (Business Logic Layer), and DAL (Data Access Layer). Using this architecture, users make requests through the UI layer, which interacts only with the BLL. The BLL, in turn, can call the DAL for data access requests. The UI layer should not make any requests 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).

Figure 5-3. A simple monolithic application with three projects.
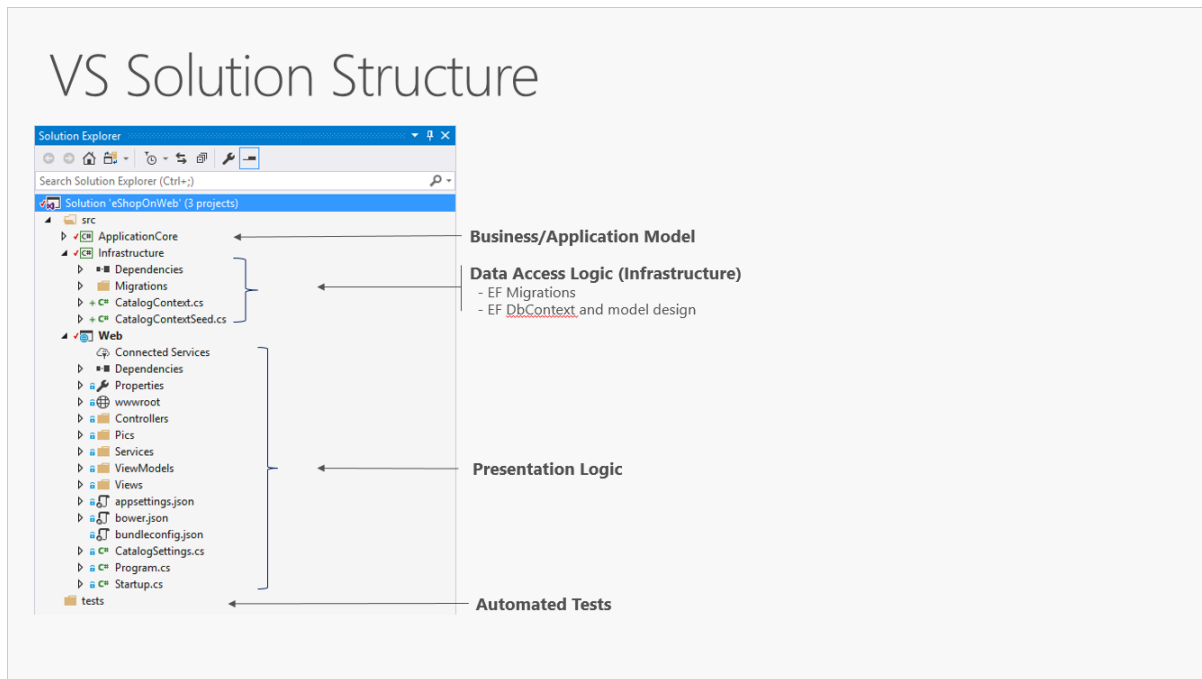
Although this application uses several projects for organizational purposes, it is still deployed as a single unit and its clients will interact with it as a single web app. This allows for very simple deployment process. Figure 5-4 shows how such an app might be hosted using Windows Azure.
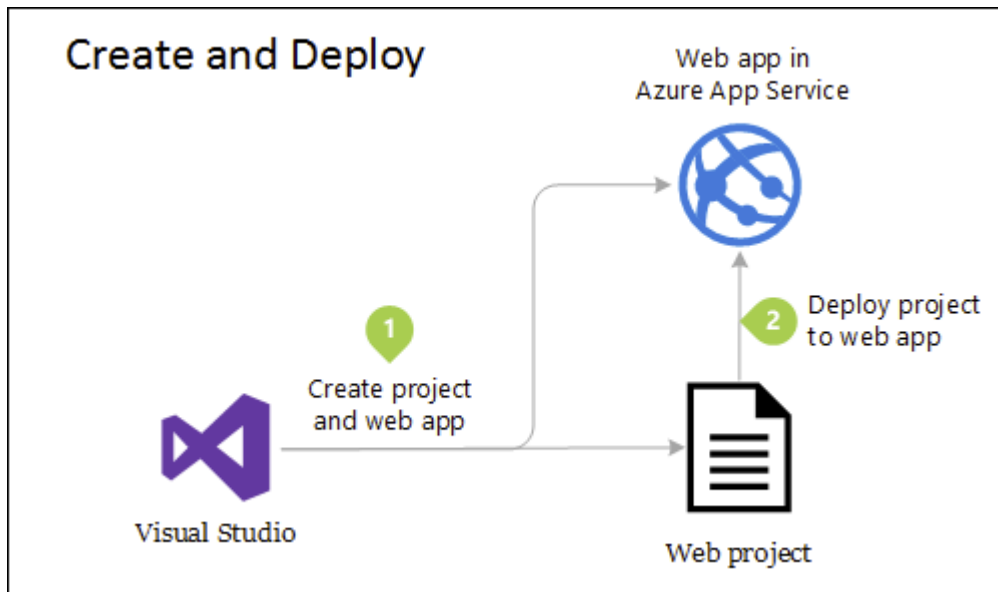
Figure 5-4. *Simple deployment of Azure Web App*

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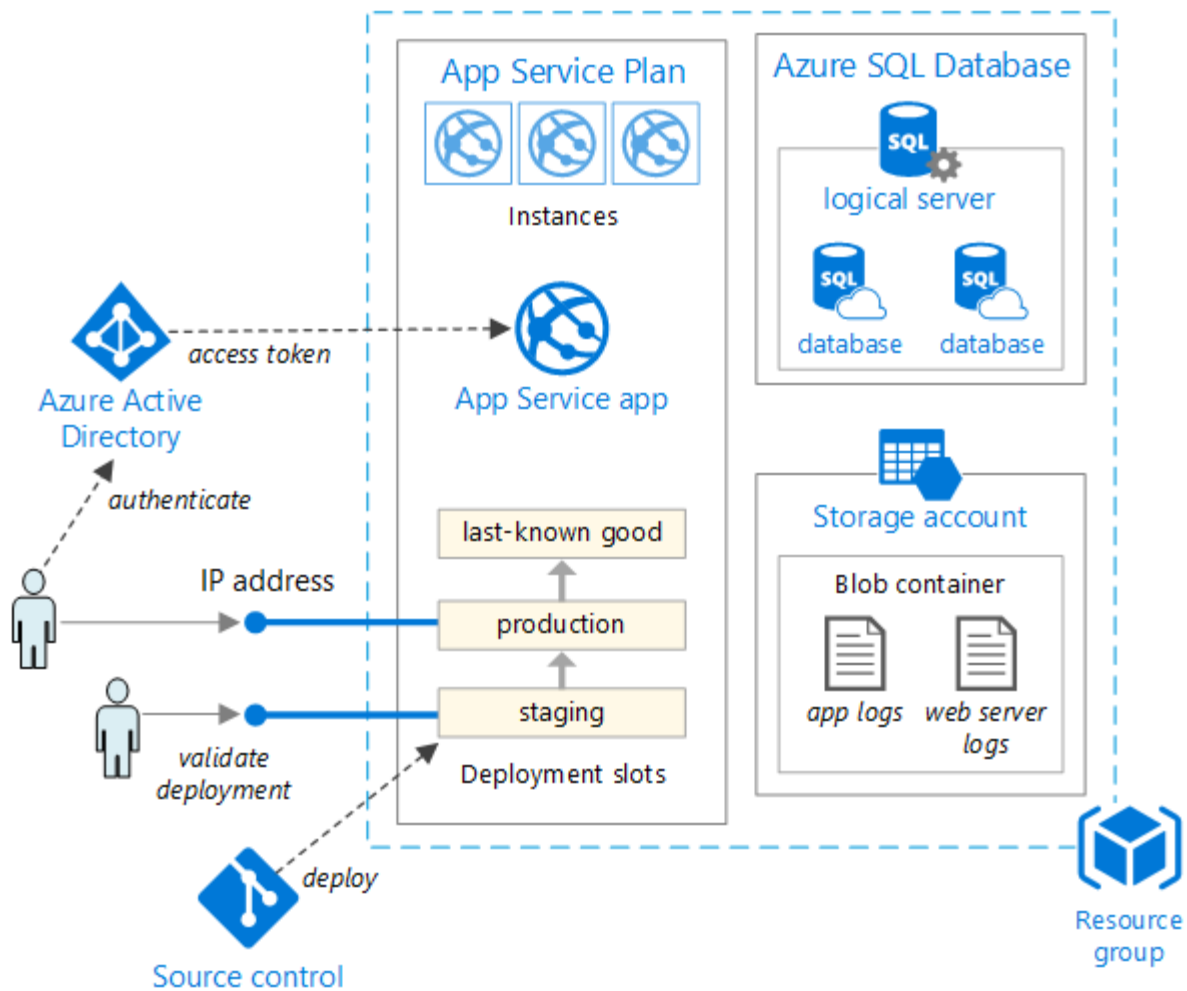
Figure 5-5. *Deploying a web app to an Azure App Service*

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 or virtual machines. 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 show the appropriate Azure dashboard screen to configure how many instances are serving an app.
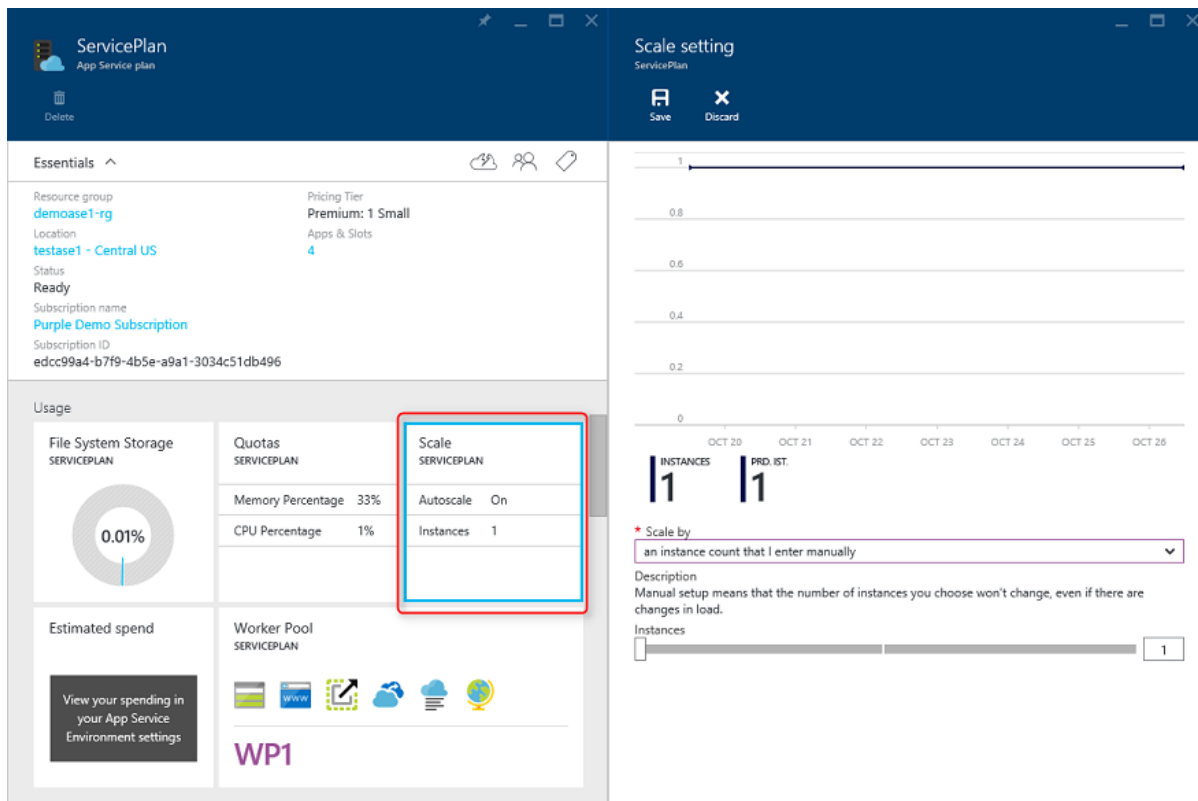
Figure 5-X. *App Service Plan scaling in Azure.*

Clean architecture

Applications that follow the Dependency Inversion Principle as well as 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. It is this last name, Clean Architecture, that is used as the basis for describing the architecture in this eBook.

Note: The term Clean Architecture can be applied to applications that are built using DDD Principles as well as to those that are not built using DDD. In the case of the former, this combination may be referred to as "Clean DDD Architecture".

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 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-X shows an example of this style of architectural representation.

Clean Architecture Layers (Onion view)
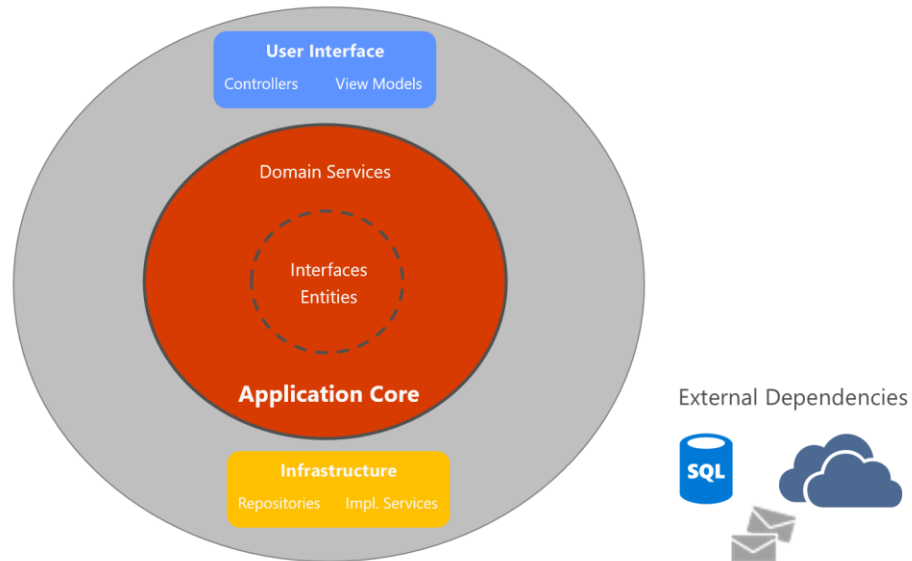
Figure 5-X. Clean Architecture; onion view

In this diagram, dependencies flow toward the innermost circle. Thus, you can see that the Application Core (which takes its name from its position at the core of this diagram) has no dependencies on other application layers. At the very center are the application's entities and interfaces. 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 User Interface and the Infrastructure layers depend on the Application Core, but not on one another (necessarily).

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

Figure 5-X. *Clean Architecture; horizontal layer view*

Note that the solid arrows represent compile-time dependencies, while the dashed arrow represents a runtime-only dependency. Using the clean architecture, the UI layer works with interfaces defined in the Application Core at compile time, and ideally should not have any knowledge of the implementation types defined in the Infrastructure layer. At runtime, however, these implementation types will be required for the app to execute, so they will need to be present and wired up to the Application Core interfaces via dependency injection.

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

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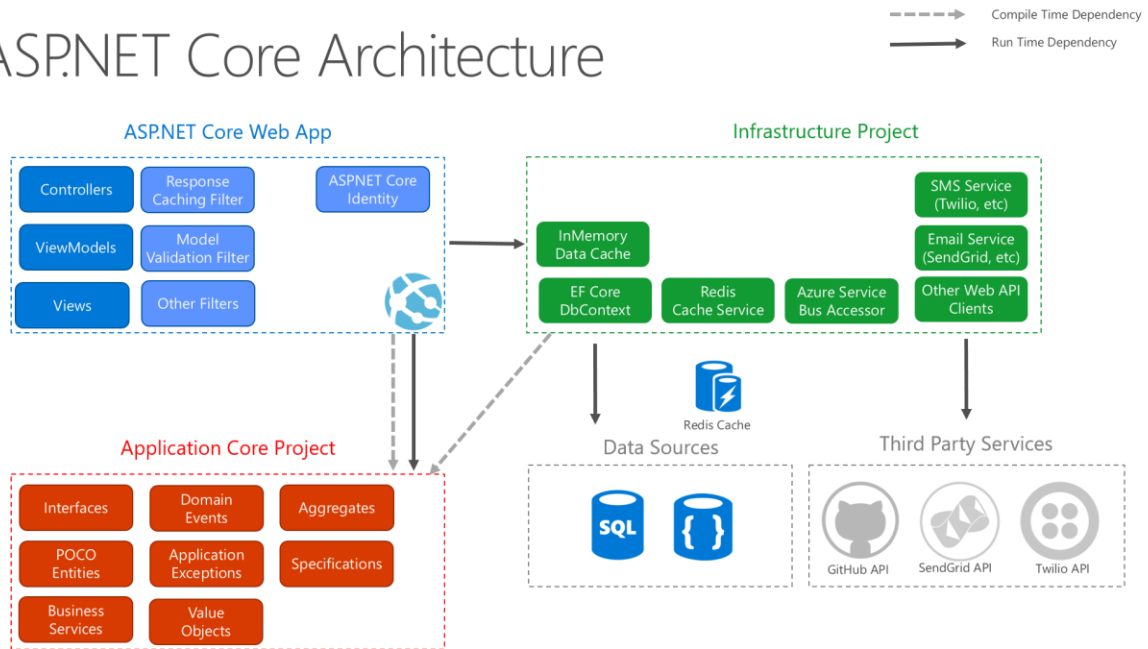
Figure 5-X. *ASP.NET Core architecture diagram following Clean Architecture.*

Because the Application Core doesn't depend on Infrastructure, it is very easy to write automated unit tests for this layer. Figures 5-X and 5-X show how tests fit into this architecture.
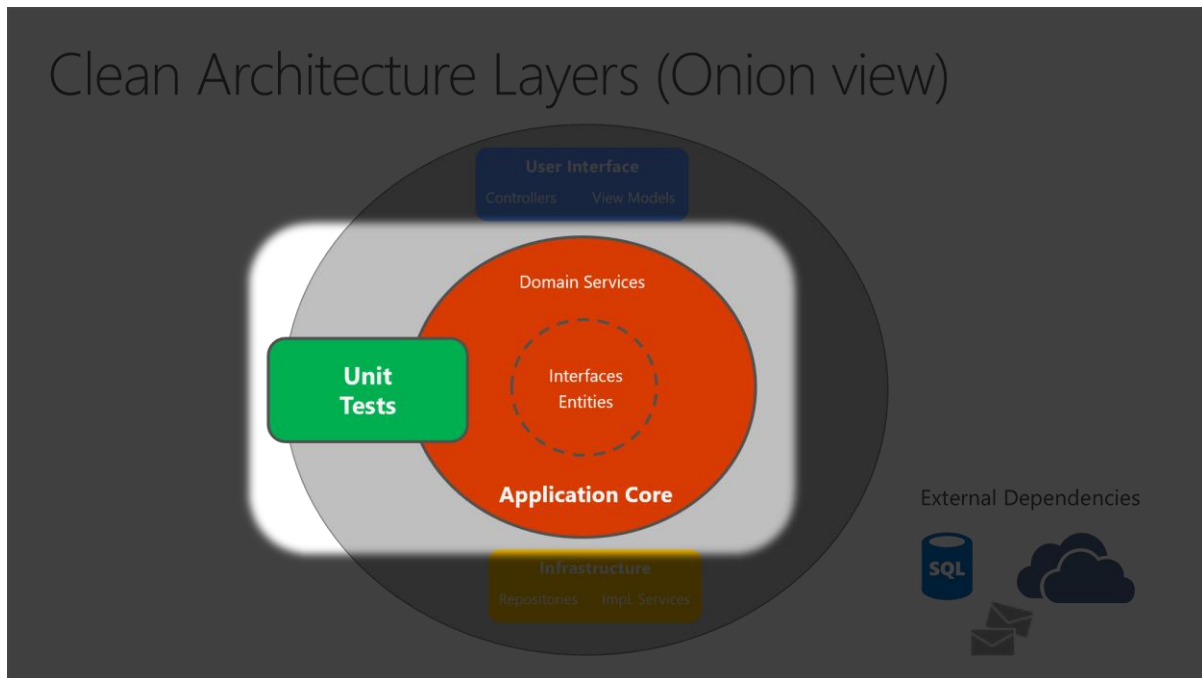
Figure 5-X. *Unit testing Application Core in isolation.*
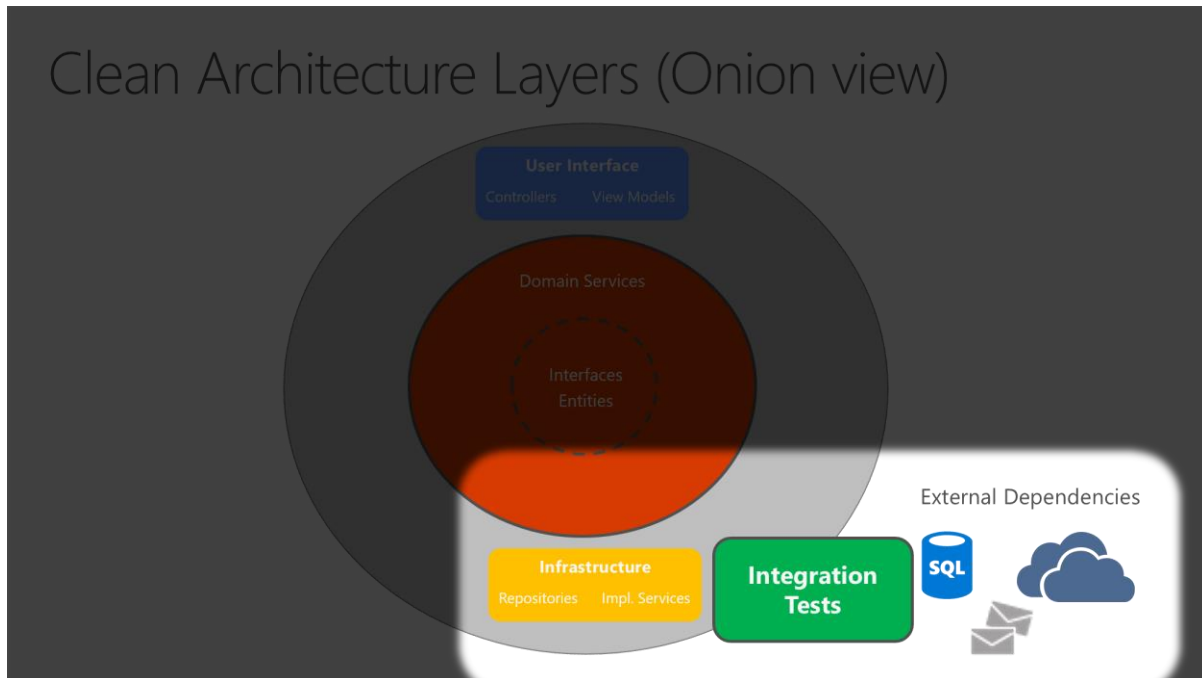
Figure 5-X. *Integration testing Infrastructure implementations with external dependencies.*

Since the UI layer doesn't have any direct dependency on types defined in the Infrastructure project, it is 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 User Interface projects are all run as a single application. The runtime application architecture might look something like Figure 5-X.

Figure 5-X. 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 will belong in each project and you'll frequently find folders corresponding to these types in the appropriate project.

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)   - Interfaces   - Services   - DTOs

The Infrastructure project will typically include data access implementations. In a typical ASP.NET Core web application, this will include the Entity Framework DbContext, any EF Core Migrations 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, Migrations)   • Data access implementation types (Repositories)   • Infrastructure-specific services (FileLogger, SmtpNotifier, etc.)

The user interface layer in an ASP.NET Core MVC application will be the entry point for the application, and will be an ASP.NET Core MVC project. 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) Infrastructure layer types should be permitted in the UI layer.

UI Layer Types
• Controllers   • Filters   • Views   • ViewModels   • Startup

The Startup class is responsible for configuring the application, and for wiring up implementation types to interfaces, allowing dependency injection to work properly at run time.

Note: In order to wire up dependency injection in ConfigureServices in the Startup.cs file of the UI project, the project may need to reference the Infrastructure project. This dependency can be eliminated, most easily by using a custom DI container. For the purposes of this sample, the simplest approach is to allow the UI project to reference the Infrastructure project.

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 is a single container like 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

App 1 = 1 Container

A monolithic application has most of its functionality within a single process/container that is componentized with internal layers or libraries.

Host 1
(Server/VM)

Host 2
(Server/VM)

Scales out by cloning the app/container on multiple servers/VMs

Host 3
(Server/VM)

Need to deploy the full application

Coarse-grained density of applications
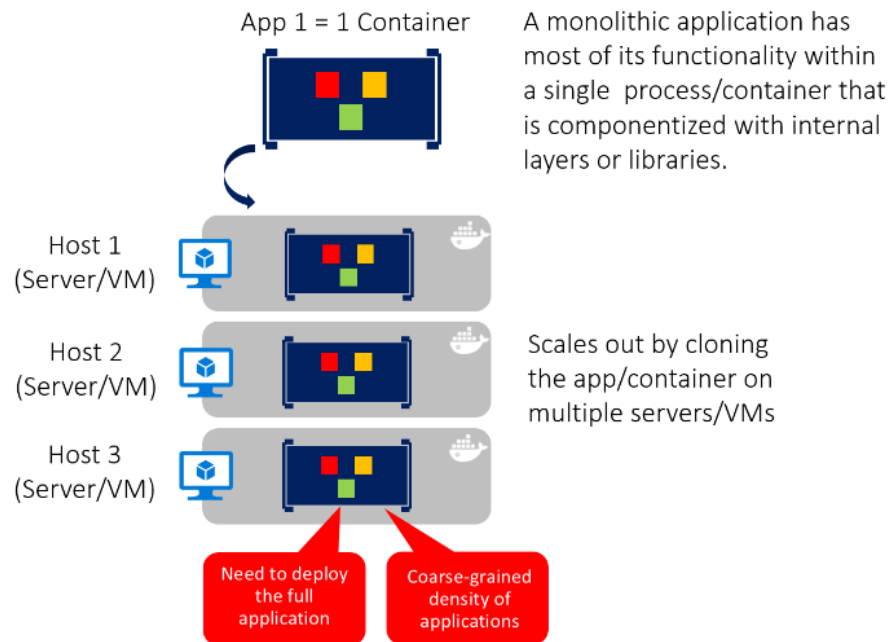
Figure <mark>5-X</mark>. Monolithic application architecture example

You can include multiple components/libraries or internal layers within each container, as illustrated in Figure <mark>5-X</mark>. But, following the container principal 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 scaled, 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 the app up to enable it to better leverage containers and microservices may be the next logical step.

Deploying monolithic applications in Microsoft Azure can be achieved using dedicated VMs for each instance. Using Azure VM 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-X, you can manage scaling.
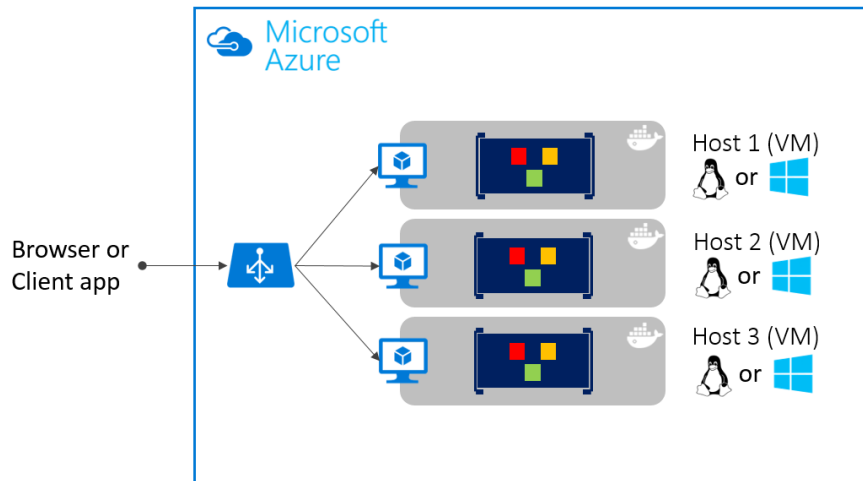
Figure 5-X. Multiple hosts scaling-out a single Docker application

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 VM Scale Sets to scale VMs, they take time to instance. 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 disk.

While monolithic apps can benefit from Docker, breaking up the monolithic application into sub systems which can be scaled, developed and deployed individually may be your entry point into the realm of microservices.

References – Common Web Architectures

Creating N-Tier Applications in C#
https://www.pluralsight.com/courses/n-tier-apps-part1
The Clean Architecture
https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html
The Onion Architecture
http://jeffreypalermo.com/blog/the-onion-architecture-part-1/
The Repository Pattern
http://deviq.com/repository-pattern/
Clean Architecture Solution Sample
https://github.com/ardalis/cleanarchitecture
Architecting Microservices eBook
http://aka.ms/MicroservicesEbook

Common Client Side Web Technologies

"Websites should look good from the inside and out."

Paul Cookson

Summary

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 JavaScript is used by web developers as part of developing applications, as provides a high-level overview of the Angular and React client side libraries.

HTML

HTML (HyperText Markup Language) 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 7-X

Figure 7-X. CSS Specificity rules, in order.

CSS Specificity

- **Element** — Least specific; applies most broadly
- **Class*** — Applies to all elements the CSS class as been applied to / Elements can have multiple classes applied to them
- **Element ID** — References a unique HTML element via its ID
- **Style Attribute** — Most specific / Added to HTML element directly

* Attribute and pseudo-class selectors also apply at this level
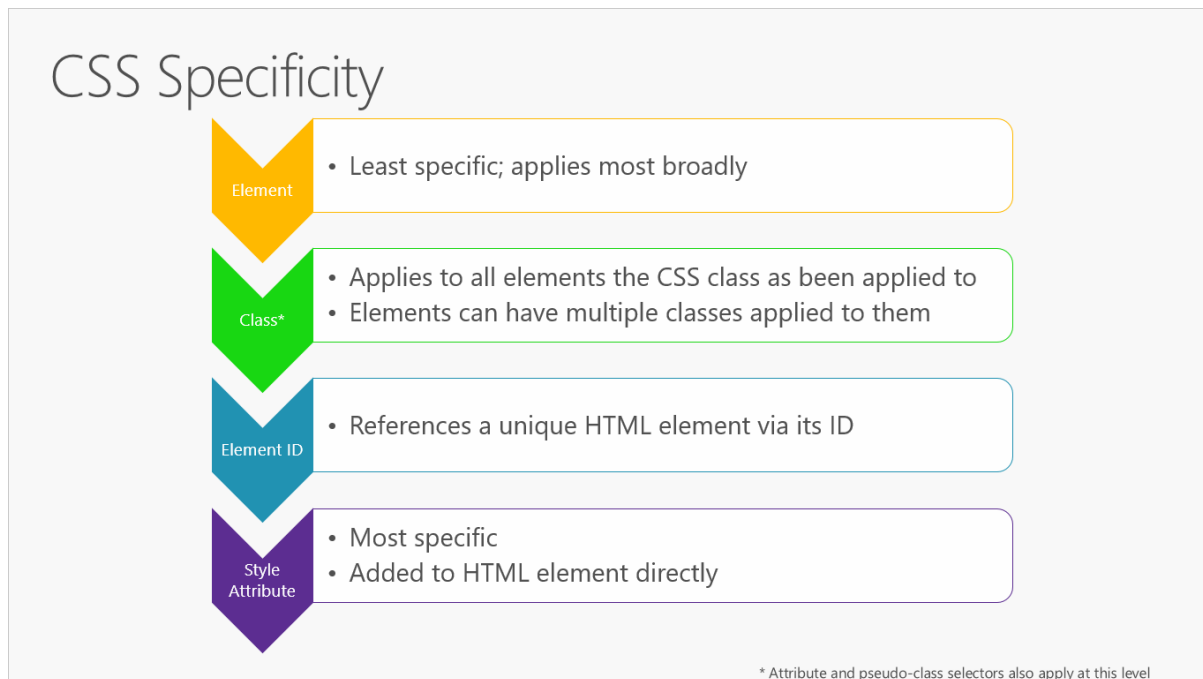
It's best to keep styles in their own separate stylesheet files, and to use selection-based cascading to implement consistent and reusable styles within the application. Placing style rules within HTML should be avoided, and applying styles to specific individual elements (rather than whole classes of 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 a lot 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 generally 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 very commonly used library for working with HTML/CSS and building applications that make AJAX calls to web APIs.

Show imperative code model vs. declarative with SPAs below (and support for routing, binding, etc.)

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**
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 a very 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. 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 any time 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 would 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 would simply bind the value or visibility of the element(s) in question to a *viewmodel* on the page, and changes to that viewmodel would automatically be reflected in the bound elements.

Angular SPAs

AngularJS quickly became one of the world's most popular JavaScript frameworks. With Angular 2, the team rebuilt the framework from the ground up (using [TypeScript](#)) and rebranded from AngularJS to simply Angular. Currently on version 4, 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.

Angular also makes great use of command line interface (CLI) tooling. 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 tool which can create projects, add files, and assist with testing, bundling, and deployment tasks. This CLI tooling friendliness makes Angular especially compatible with ASP.NET Core, which also features great CLI support.

Reference SPA sample in eShopOnContainers which uses Angular 2 and TypeScript.

Show a short summary of the structure of the application when using Angular 2.
https://github.com/dotnet/eShopOnContainers/tree/master/src/Web/WebSPA

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.

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.

Choosing a SPA Framework

Show considerations and a decision table with jQuery, Angular, and React as options.

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://docs.microsoft.com/en-us/aspnet/core/client-side/less-sass-fa
Client-Side Development in ASP.NET Core https://docs.microsoft.com/en-us/aspnet/core/client-side/
jQuery https://jquery.com/
jQuery vs AngularJS https://www.airpair.com/angularjs/posts/jquery-angularjs-comparison-migration-walkthrough
Angular https://angular.io/
React https://facebook.github.io/react/
React Slingshot https://github.com/coryhouse/react-slingshot
React vs Angular 2 Comparison https://www.codementor.io/codementorteam/react-vs-angular-2-comparison-beginners-guide-lvz5710ha
5 Best JavaScript Frameworks of 2017 https://hackernoon.com/5-best-javascript-frameworks-in-2017-7a63b3870282

Developing 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

Summary

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 in 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.

Mapping Requests to Responses

At its heart, ASP.NET Core apps map incoming requests to outgoing responses. At a low level, this 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.UseMvc(routes =>

{

routes.MapRoute("default","{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 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 the Configure method in the Startup class.

Attribute routes are applied to controllers and actions directly, rather than specified globally. This 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()

}
```

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 are 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 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.

References – Mapping Requests to Responses
Routing to Controller Actions https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing **Model Binding** https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding **Model Validation** https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation **Filters** https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters

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 enabled 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 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 a Startup class, which itself is configured to support dependency injection at several points. If your Startup class has a constructor, it can request dependencies through the constructor, 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 you've told it to use, 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 `WebHostBuilder` and its ConfigureServices method.

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 be comprised of 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 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 project itself 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 StructureMap, which allows you to define DI rules in Registry 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-X shows an example folder structure, using Areas.
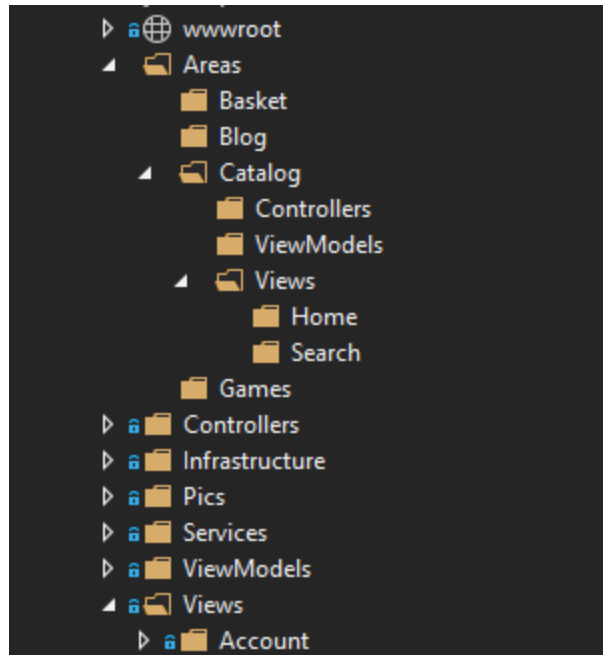
Figure 7-X Sample Area Organization

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.UseMvc(routes =>

{

  // Areas support

  routes.MapRoute(

    name: "areaRoute",

    template:
"{area:exists}/{controller=Home}/{action=Index}/{id?}");

  routes.MapRoute(
```

```
        name: "default",

        template: "{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.

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):

```
    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:

```
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 article, Feature Slices for ASP.NET Core MVC.

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-X shows how request execution flows through filters, if configured.
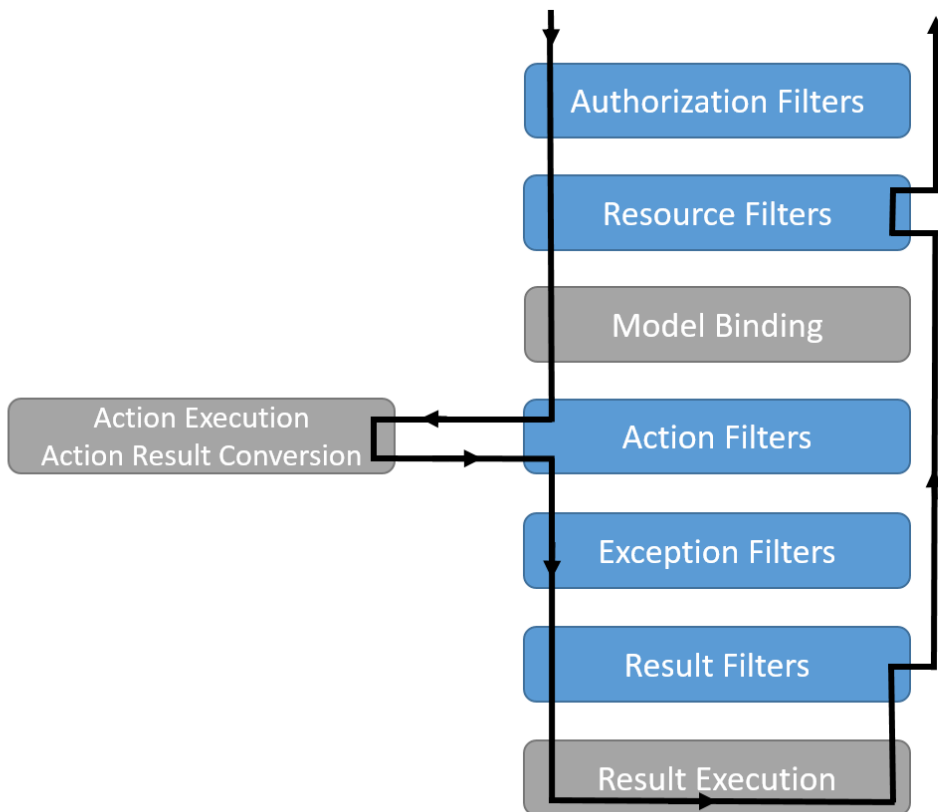
Figure 7-X Request execution through filters and request pipeline.

Filters are usually implemented as attributes, so you can apply them controllers or actions. 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]

    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 any time a command is sent to the API, can be replaced by the following attribute:

```csharp
public class ValidateModelAttribute : ActionFilterAttribute

{

    public override void OnActionExecuting(ActionExecutingContext
context)

    {

    if (!context.ModelState.IsValid)

    {

        context.Result = new
BadRequestObjectResult(context.ModelState);

    }

    }

}
```

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:

```csharp
// PUT api/authors2/5

[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 article, Real World ASP.NET Core MVC Filters.

References – Structuring Applications
Areas https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/areas **MSDN – Feature Slices for ASP.NET Core MVC** https://msdn.microsoft.com/en-us/magazine/mt763233.aspx **Filters** https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters **MSDN – Real World ASP.NET Core MVC Filters** https://msdn.microsoft.com/en-us/magazine/mt767699.aspx

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 that 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.

Authentication

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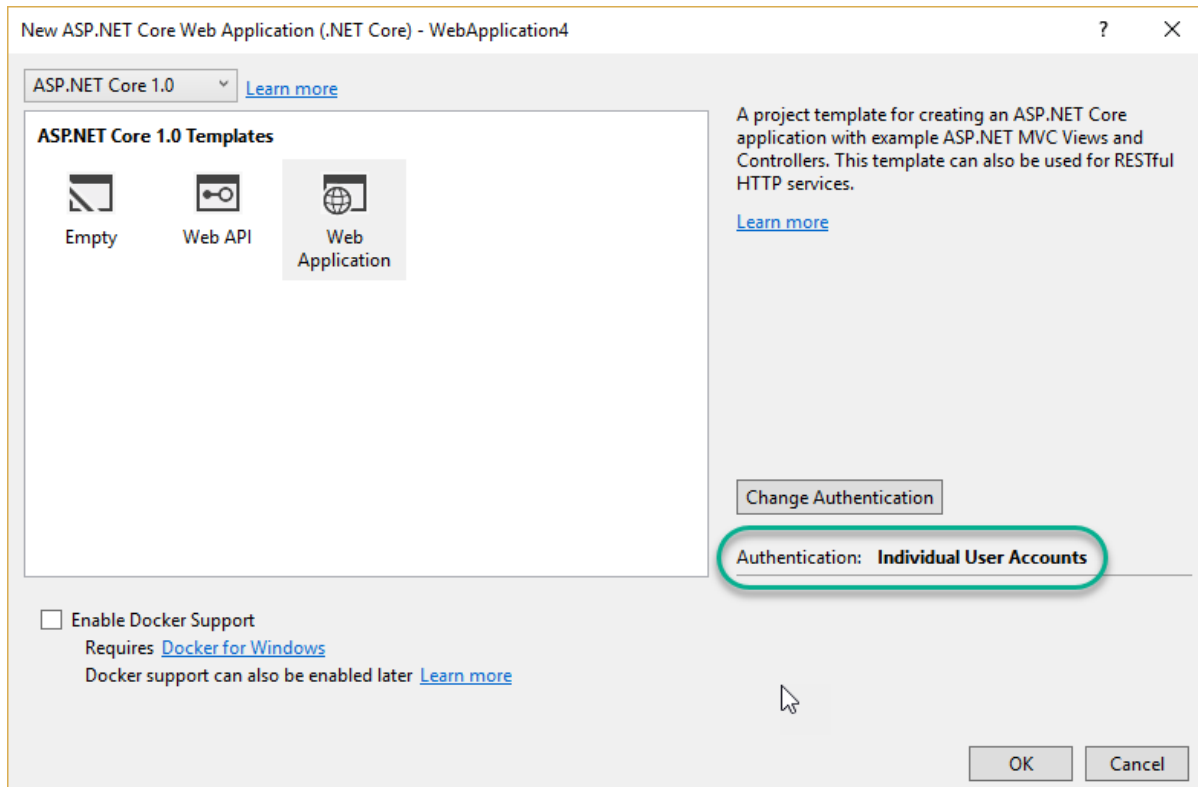
Figure 7-X Select Individual User Accounts to have Identity preconfigured.

Identity support is configured in Startup, both in ConfigureServices and Configure:

```
public void ConfigureServices(IServiceCollection services)

{

    // Add framework services.

    services.AddDbContext<ApplicationDbContext>(options =>

    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

        services.AddIdentity<ApplicationUser, IdentityRole>()

            .AddEntityFrameworkStores<ApplicationDbContext>()

            .AddDefaultTokenProviders();
```

```
        services.AddMvc();

    }
    public void Configure(IApplicationBuilder app)

    {

        app.UseStaticFiles();



        app.UseIdentity();



        app.UseMvc(routes =>

        {

            routes.MapRoute(

                name: "default",

                template: "{controller=Home}/{action=Index}/{id?}");

        });

    }
```

It's important that UseIdentity appear before UseMvc in the Configure method. When configuring Identity in ConfigureServices, 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.

Authorization

The simplest form of authorization involves restricting access to anonymous users. This can be achieved by simply 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. You can 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. 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-X demonstrates this process.
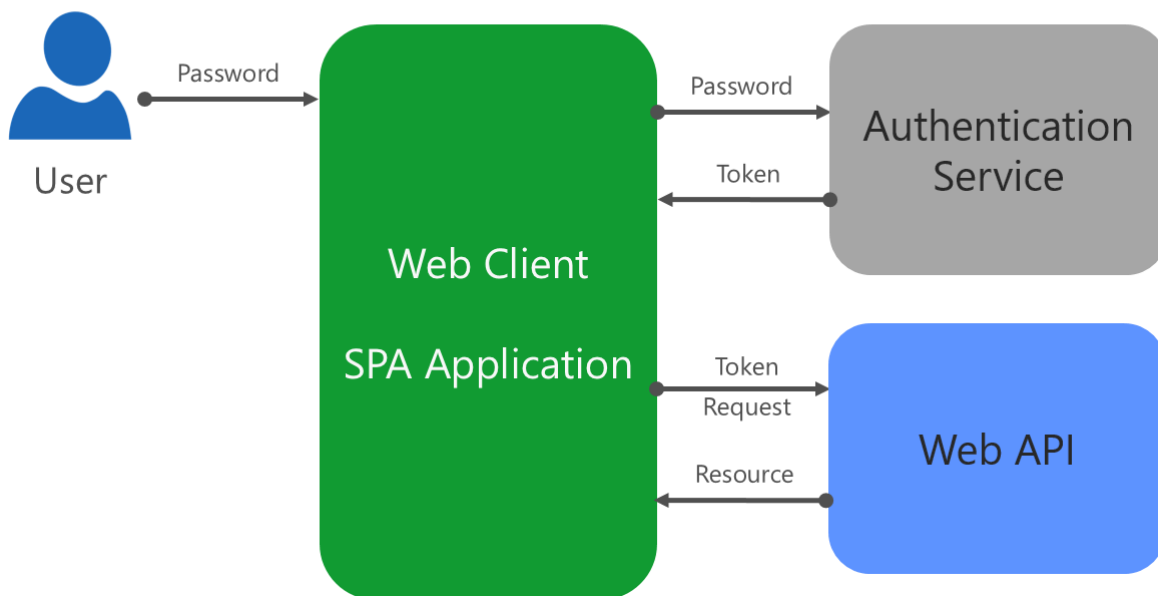
Figure 7-X. *Token-based authentication for Web APIs.*

References – Security
Security Docs Overview   https://docs.microsoft.com/en-us/aspnet/core/security/   **Enforcing SSL in an ASP.NET Core App**   https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl   **Introduction to Identity**   https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity   **Introduction to Authorization**   https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction   **Authentication and Authorization for API Apps in Azure App Service**   https://docs.microsoft.com/en-us/azure/app-service-api/app-service-api-authentication

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 kind of 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.

ASP.NET Core SignalR is currently under development, and will be available in the next release of ASP.NET Core. However, other open source WebSockets libraries are currently available.

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 are not 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/aspnet/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 (e.g. database tables). Thus, the concepts described in the ubiquitous language should form the basis for your *domain model*.

Your domain model is comprised of 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.

Note that 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.
- Domain events, for decoupling dependent behavior from triggering behavior.
- 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 needn't have 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
Domain-Driven Design Fundamentals (course) http://bit.ly/PS-DDD **Design Patterns Library (course)** http://bit.ly/DesignPatternsLibrary **DDD in Plain English (StackOverflow Answer)** http://bit.ly/2pmVgK2

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 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 hosted in the Kestrel web server must 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, and are required for edge deployments (exposed to traffic from the Internet). Kestrel is relatively new and does not yet offer defenses against certain attacks. 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.
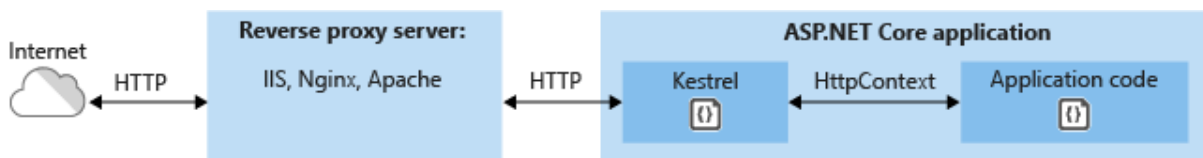

Figure 7-X ASP.NET hosted in Kestrel behind a reverse proxy server

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-X.

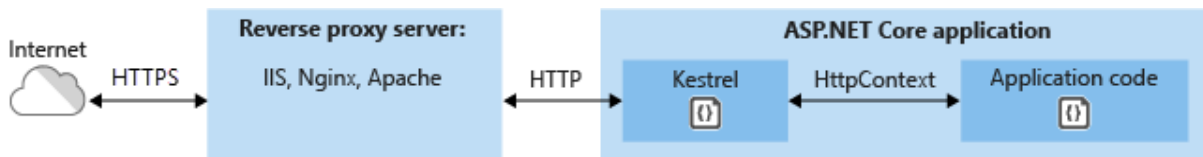
Figure 7-X ASP.NET hosted behind an HTTPS-secured reverse proxy server

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://docs.microsoft.com/en-us/aspnet/core/publishing/ **When to use Kestrel with a reverse proxy** https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel#when-to-use-kestrel-with-a-reverse-proxy **Host ASP.NET Core apps in Docker** https://docs.microsoft.com/en-us/aspnet/core/publishing/docker **Introducing Azure Application Gateway** https://docs.microsoft.com/en-us/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

Summary

Data access is an important part of almost any software application. ASP.NET Core supports a variety of 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 in 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. Note that 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 your **ConfigureServices** method. EF Core uses a **DbContextOptionsBuilder**, which supports several helpful extension methods to streamline its configuration. Tp configure **CatalogContext** to use a SQL Server database with a connection string defined in Configuration, you would add the following code to **ConfigureServices**:
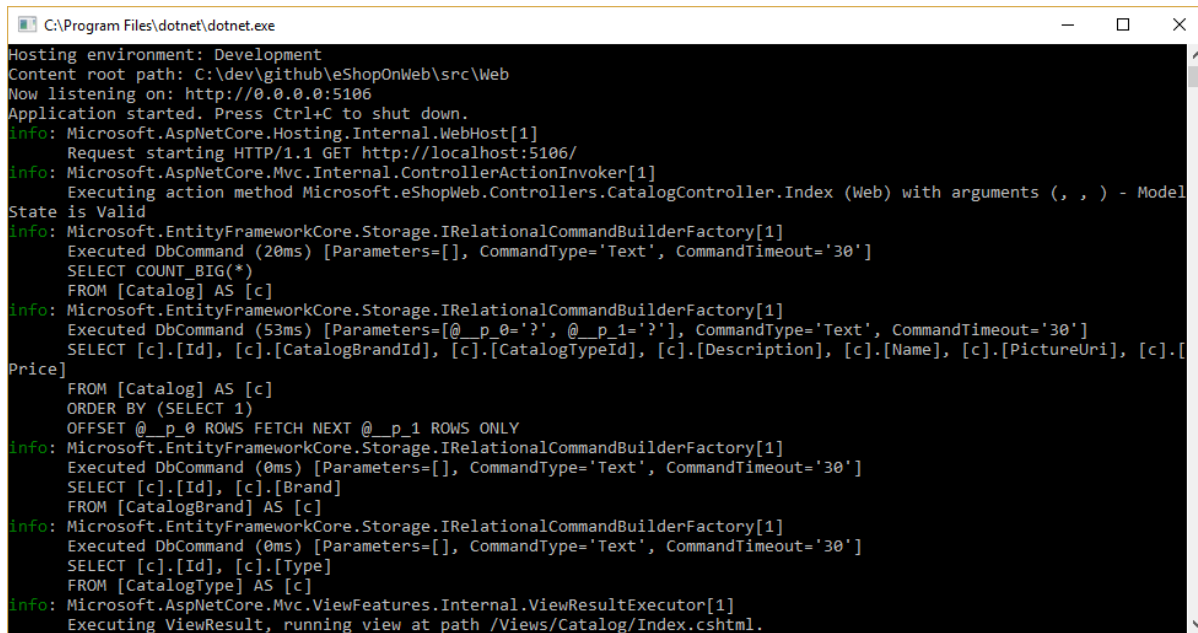
```
services.AddDbContext<CatalogContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultCon
nection")));
```

To use the in-memory database:

```
services.AddDbContext<CatalogContext>(options =>

    options.UseInMemoryDatabase());
```

Once you have installed EF Core, created a DbContext child type, and configured it in ConfigureServices, 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-X.

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 runtime, 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 (e.g. **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 SaveChanges 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 on the appropriate DbSet property, again with a call to SaveChanges 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.

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 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 sub-relationships using `ThenInclude`. EF Core will execute a single query to retrieve the resulting set of entities.

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 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. It's not currently supported by EF Core, but as with explicit loading it should typically be disabled for web applications.

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 rety 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 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.

```
// Startup.cs from any ASP.NET Core Web API
public class Startup
{
  public IServiceProvider ConfigureServices(IServiceCollection services)
  {
    //...
    services.AddDbContext<OrderingContext>(options =>
    {
      options.UseSqlServer(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 retriable operation. Each query and each call to `SaveChanges` 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 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 `SaveChanges` from multiple DbContexts 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 retriable 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://docs.microsoft.com/en-us/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
DbContexts and using an EF Execution Strategy.

References – Entity Framework Core
EF Core Docs https://docs.microsoft.com/en-us/ef/ **EF Core: Related Data** https://docs.microsoft.com/en-us/ef/core/querying/related-data **Avoid Lazy Loading Entities in ASPNET Applications** http://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 is not 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 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 `AsNotTracking` 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 slightly. Julie Lerman presents some performance data in her May 2016 MSDN article Dapper,
Entity Framework, and Hybrid Apps. Additional 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
```

```csharp
    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 is supported through a variety of 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:

```sql
    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:

```csharp
    (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:

```csharp
    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 not compile time guarantees, so they may break at runtime 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 certainly simpler to store a single serialized object with a key than to decompose the object into many tables with relationships and update and 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 very 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 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 a country's name 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 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 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 DocumentDB

Azure DocumentDB is a fully managed NoSQL database service that offers cloud-based schema-free data storage. DocumentDB 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 DocumentDB are stored as JSON documents. Resources are managed as *items*, which are documents containing metadata, and *feeds*, which are collections of items. Figure 8-X shows the relationship between different DocumentDB resources.
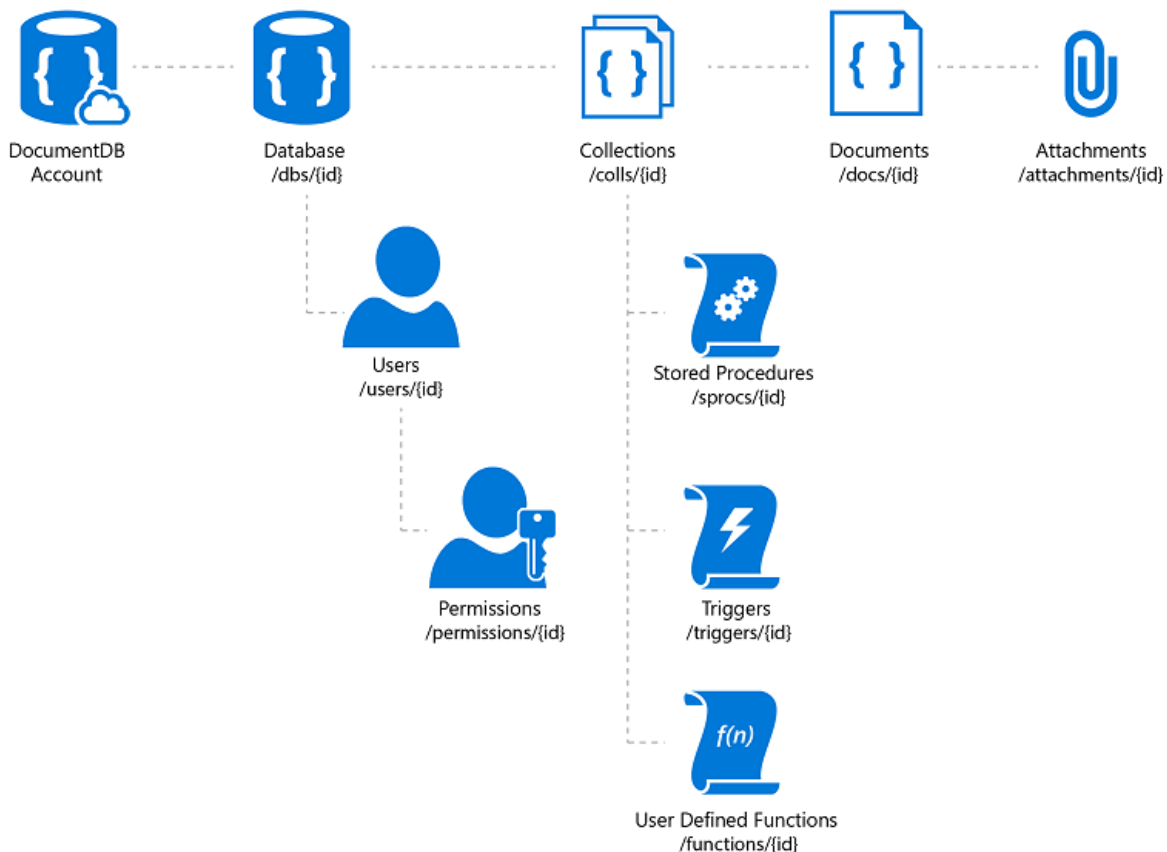
The DocumentDB query language is a simple yet powerful interface for querying JSON documents. The language supports a subset of ANSI SQL grammar and adds deep integration of JavaScript object, arrays, object construction, and function invocation.

References – DocumentDB

- DocumentDB Introduction
  https://docs.microsoft.com/en-us/azure/documentdb/documentdb-introduction

Other Persistence Options

In addition to relational and NoSQL storage options, ASP.NET Core applications can use Azure Storage to store a variety of 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://docs.microsoft.com/en-us/azure/storage/storage-introduction

Caching

In web applications, each web request should be completed in the shortest time possible. One way to achieve this 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 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 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 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 above 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 in both `ConfigureServices` and `Configure` in Startup:

```
public void ConfigureServices(IServiceCollection services)

{

    services.AddResponseCaching();

}

public void Configure(IApplicationBuilder 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, 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.

You add support for memory (or distributed) caching in ConfigureServices:

```
public void ConfigureServices(IServiceCollection services)

{

    services.AddMemoryCache();

    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 string _itemsKeyTemplate = "items-{0}-
{1}-{2}-{3}";

    private static readonly TimeSpan _defaultCacheDuration =
TimeSpan.FromSeconds(30);


    public CachedCatalogService(IMemoryCache cache,
```

```csharp
        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 = String.Format(_itemsKeyTemplate,
pageIndex, itemsPage, brandID, typeId);

        return await _cache.GetOrCreateAsync(cacheKey, async entry
=>

        {
```

```csharp
            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 in `ConfigureServices`:

```csharp
services.AddMemoryCache();

services.AddScoped<ICatalogService, CachedCatalogService>();

services.AddScoped<CatalogService>();
```

With this 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 very 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 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 were 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 `CancellationChangeToken`. With a `CancellationChangeToken`, you can expire multiple cache entries at once by cancelling 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();
```

Testing 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

Summary

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 are not designed to be testable, it can be the only means available. Applications written following the architectural principles laid out in chapter X should be unit testable, and ASP.NET Core applications support automated integration and functional testing as well.

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, like 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 quickly. 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 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 tests 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.

The `LocalFileImageService` implementation class implements the logic for fetching and returning the bytes of an image file from a particular folder given an id:

```
public class LocalFileImageService : IImageService

{

    private readonly IHostingEnvironment _env;


    public LocalFileImageService(IHostingEnvironment env)

    {

        _env = env;

    }

    public byte[] GetImageBytesById(int id)

    {

        try

        {

            var contentRoot = _env.ContentRootPath + "//Pics";

            var path = Path.Combine(contentRoot, id + ".png");
```

```
                    return File.ReadAllBytes(path);
```

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 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-X.
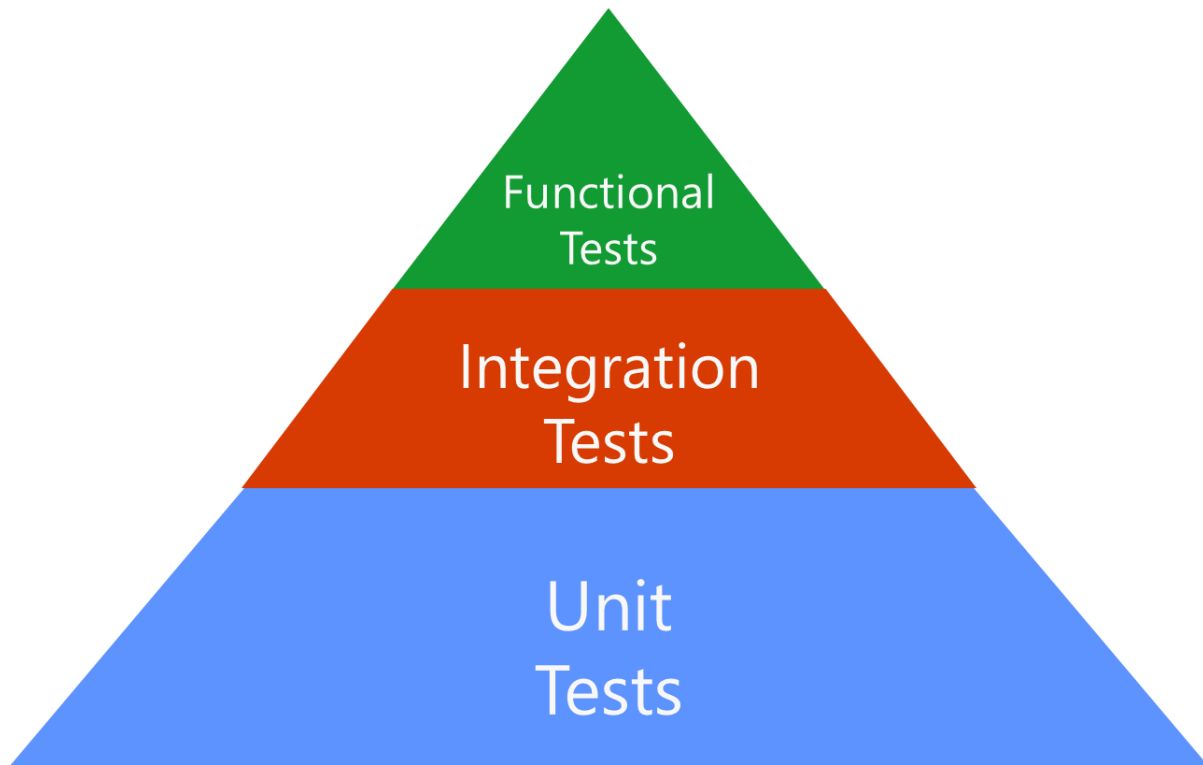
Figure 9-X Testing Pyramid

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, etc.), you should be able to come up 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 very 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.
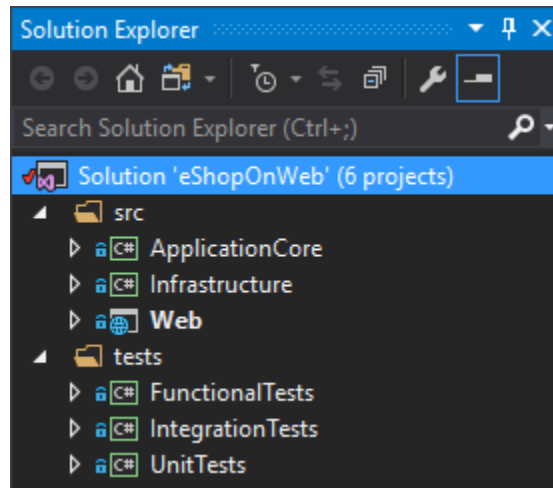

Figure 9-X Test organization in your solution

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-X, or from the CLI using `dotnet new xunit`.
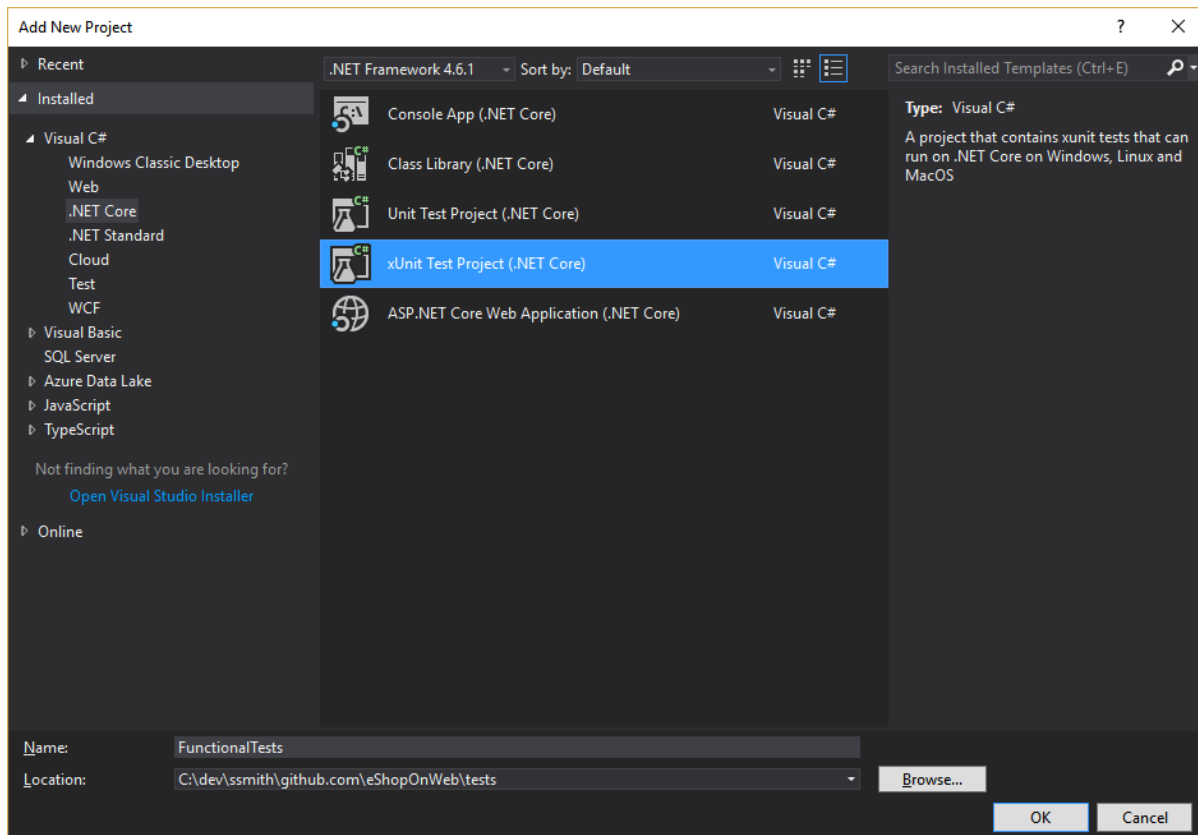
Figure 9-X Add an xUnit Test Project in Visual Studio

Test Naming

You should 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 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 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:

- CatalogControllerGetImage**Should**.**Call**ImageServiceWithId
- CatalogControllerGetImage**Should**.**Log**WarningGivenImageMissingException

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 you tests vaguely, such as `ControllerTests.Test1`, as these 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-X shows one approach to organizing tests by folder within several test projects.
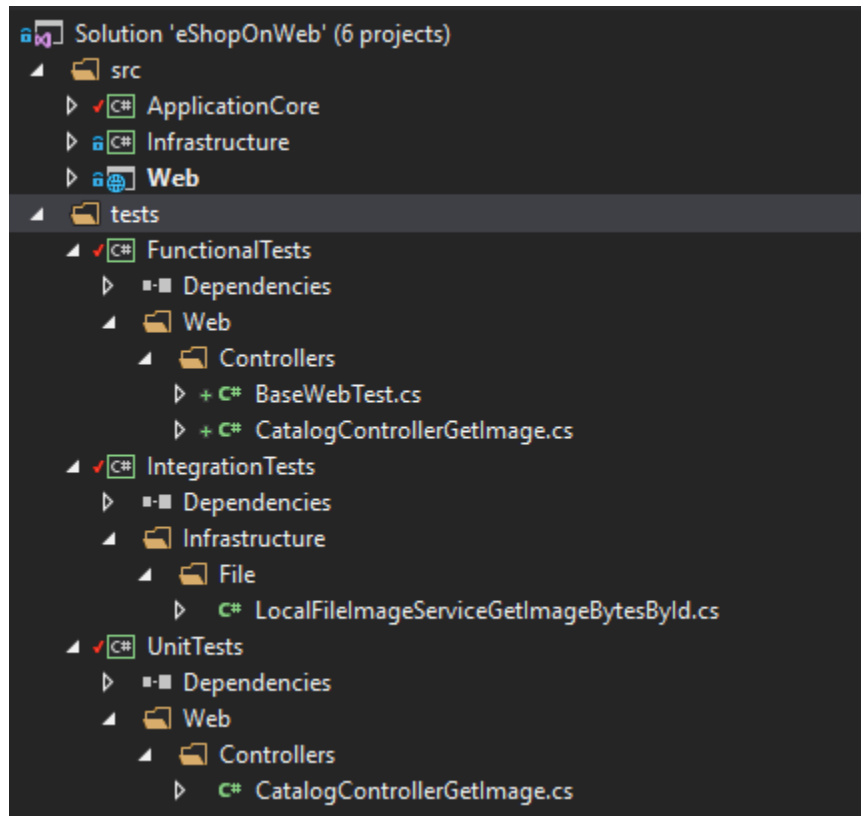

Figure 9-X. *Organizing test classes by folder based on class being tested.*

Of course, if a particular application class has many methods being tested (and thus many test classes), it may make sense to place these 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 very few unit tests. Much of the functionality of a given action lies outside the action method itself. Testing whether routing works correctly, or global error handling, cannot be done effectively with a unit test. Likewise, any filters, including model validation and authentication and authorization filters, cannot be unit tested. 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 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 simple 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 test this method's route – you'll see how to do this 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");

    }
```

The _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 the _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 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.

Integration Testing ASP.NET Core Apps

```
        }

        catch (FileNotFoundException ex)

        {

            throw new CatalogImageMissingException(ex);

        }

    }

}
```

This service uses the IHostingEnvironment, just as the CatalogController code did before it was refactored into a separate service. Since this was the only code in the controller that used IHostingEnvironment, that dependency was removed from CatalogController's constructor.

To test that this service works correctly, you need to create a known test image file and verify that the service returns it given a specific input. You should take care not to use mock objects on the behavior you actually want to test (in this case, reading from the file system). However, mock objects may still be useful to set up integration tests. In this case, you can mock IHostingEnvironment so that its

`ContentRootPath` points to the folder you're going to use for your test image. The complete working integration test class is shown here:

```
public class LocalFileImageServiceGetImageBytesById

{

    private byte[] _testBytes = new byte[] { 0x01, 0x02, 0x03 };

    private readonly Mock<IHostingEnvironment> _mockEnvironment =
new Mock<IHostingEnvironment>();

    private int _testImageId = 123;

    private string _testFileName = "123.png";


    public LocalFileImageServiceGetImageBytesById()

    {

        // create folder if necessary

        Directory.CreateDirectory(Path.Combine(GetFileDirectory(),
"Pics"));


        string filePath = GetFilePath(_testFileName);

        System.IO.File.WriteAllBytes(filePath, _testBytes);

        _mockEnvironment.SetupGet<string>(m =>
m.ContentRootPath).Returns(GetFileDirectory());

    }


    private string GetFilePath(string fileName)

    {

        return Path.Combine(GetFileDirectory(), "Pics", fileName);
```

```csharp
        }


        private string GetFileDirectory()

        {

            var location =
    System.Reflection.Assembly.GetEntryAssembly().Location;

            return Path.GetDirectoryName(location);

        }



        [Fact]

        public void ReturnsFileContentResultGivenValidId()

        {

            var fileService = new
    LocalFileImageService(_mockEnvironment.Object);


            var result = fileService.GetImageBytesById(_testImageId);


            Assert.Equal(_testBytes, result);

        }

    }
```

Note that the test itself is very simple – the bulk of the code is necessary to configure the system and create the testing infrastructure (in this case, an actual file to be read from disk). This is typical for integration tests, which often require more complex setup work than unit tests.

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`, just as you normally do for your application. This `WebHostBuilder` should be configured just like your application's real host, but you can modify any aspects of it that make testing easier. Most of the time, you'll reuse the same `TestServer` for many test cases, so you can encapsulate it in a reusable method (perhaps in a base class):

```
public abstract class BaseWebTest

{

    protected readonly HttpClient _client;

    protected string _contentRoot;


    public BaseWebTest()

    {

        _client = GetClient();

    }


    protected HttpClient GetClient()

    {

        var startupAssembly =
typeof(Startup).GetTypeInfo().Assembly;

        _contentRoot = GetProjectPath("src", startupAssembly);

        var builder = new WebHostBuilder()

            .UseContentRoot(_contentRoot)

            .UseStartup<Startup>();
```

```
        var server = new TestServer(builder);

        var client = server.CreateClient();


        return client;

    }

}
```

The `GetProjectPath` method simply returns the physical path to the web project (download sample solution). The `WebHostBuilder` in this case simply specifies where the content root for the web application is, and references the same Startup class the real web application uses. To work with the `TestServer`, you use the standard `System.Net.HttpClient` type to make requests to it. TestServer exposes a helpful CreateClient method that provides a pre-configured client that is ready to make requests to the application running on the TestServer. You use this client (set to the protected _client member on the base test above) when writing functional tests for your ASP.NET Core application:

```
 public class CatalogControllerGetImage : BaseWebTest

{

    [Fact]

    public async Task ReturnsFileContentResultGivenValidId()

    {

        var testFilePath = Path.Combine(_contentRoot,
"pics//1.png");

        var expectedFileBytes = File.ReadAllBytes(testFilePath);


        var response = await _client.GetAsync("/catalog/pic/1");

        response.EnsureSuccessStatusCode();

        var streamResponse = await
response.Content.ReadAsStreamAsync();

        byte[] byteResult;
```

```
        using (var ms = new MemoryStream())

        {

            streamResponse.CopyTo(ms);

            byteResult = ms.ToArray();

        }



        Assert.Equal(expectedFileBytes, byteResult);

    }

}
```

This functional test exercises the full ASP.NET Core MVC application stack, including all middleware, filters, binders, etc. that may be in place. It verifies that a given route ("/catalog/pic/1") returns the expected byte array for a file in a known location. It does so without setting up a real web server, and so 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.

Development process for Azure-hosted ASP.NET Core applications

"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 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

References

https://www.visualstudio.com/en-us/docs/build/apps/aspnet/aspnetcore-to-azure

app service

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 Docker applications.

Visual Studio with Docker Tools. If you're using *Visual Studio 2015* you can install the add-in tools "Docker Tools for Visual Studio". If you're using *Visual Studio 2017*, Docker Tools are already installed. In either case you can develop, run and validate your applications directly in the target Docker

environment. F5 your application (single container or multiple containers) directly into a Docker host with debugging, or CTRL + F5 to edit & refresh your app without having to rebuild the container. This is the simplest and most powerful choice for Windows developers targeting Docker containers for Linux or Windows.
[Download Docker Tools for Visual Studio](#)
[Download Docker for Mac and Windows](#)

Visual Studio Code and Docker 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 Docker CLI. These products provide a simple yet robust experience that streamlines the developer workflow. By installing either the "Docker for Mac" or "Docker for Windows" development environment, Docker developers can use a single Docker CLI to build apps for both Windows and Linux. Additionally, Visual Studio Code supports extensions for Docker such as intellisense for Dockerfiles and shortcut-tasks to run Docker commands from the editor.
[Download Visual Studio Code](#)
[Download Docker for Mac and Windows](#)

.NET languages and frameworks for ASP.NET Core

As introduced in initial sections, you can use **.NET Framework**, **.NET Core,** or the OSS project **Mono** when developing ASP.NET Core applications. You can develop in **C#**, **F#** or **Visual Basic** on Windows, Mac, or **Linux** systems, depending on the chosen framework.

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.

The inner-loop development workflow that utilizes Azure can use the following process. Note that the initial steps to set up the environment are not included, as that has to be done only once.

Workflow for developing Azure-hosted ASP.NET Core applications

TBD

The following are the basic steps you usually take when building an ASP.NET Core app, hosted in Azure, as illustrated in Figure 10-X.

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

In this guide, this whole process is detailed and every critical step is explained.

For further details on how to debug containers, read the following articles.

Build, Debug, Update and Refresh apps in a local Docker container:
https://azure.microsoft.com/en-us/documentation/articles/vs-azure-tools-docker-edit-and-refresh/

Simplified workflow when developing containers with Visual Studio

Effectively, the workflow when using Visual Studio is a lot simpler than a regular Docker container development process because most of the steps required by Docker related to dockerfile and docker-compose.yml are hidden or simplified by Visual Studio, as shown in the image X-XX.
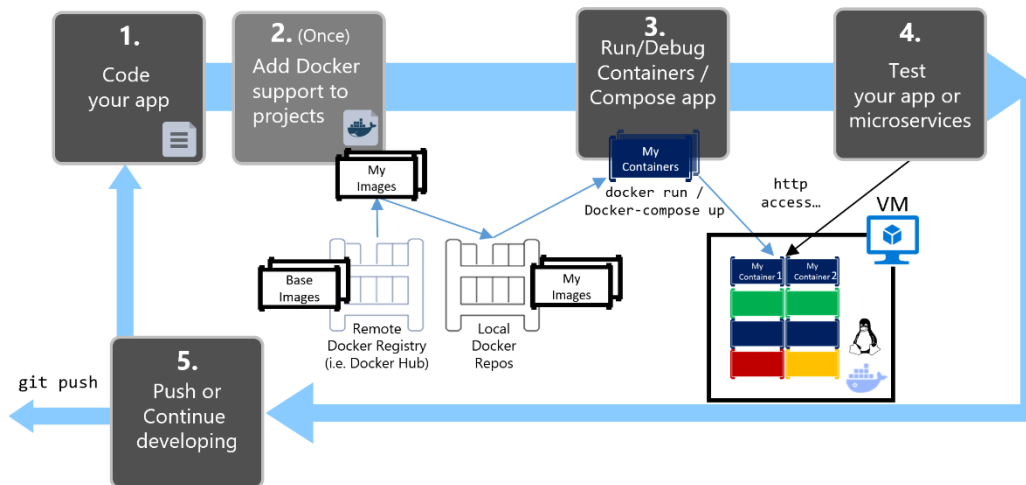

Figure X-XX. Simplified workflow when developing with Visual Studio

Even further, the step number 2, "Add Docker support to your projects" needs to be done just once. So usually that process or workflow remains pretty similar to your usual development tasks when using plain .NET. However, you still need to know what's going on under the covers (images build process, what base images you are using, deployment of containers, etc.) and sometimes you will also need to edit the *dockerfile* or *docker-compose.yml* when customizing the behaviors. However, for the most part of the work, it'll be greatly simplified by Visual Studio, making you a lot more productive.

Azure Hosting Recommendations for ASP.NET Core Web Apps

"Line-of-business leaders everywhere are bypassing IT departments to get applications from the cloud (aka 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

Summary

Whatever your application's needs and architecture, Windows Azure can support it. Your hosting needs can be...

https://azure.microsoft.com/en-us/campaigns/developer-guide/

https://docs.microsoft.com/en-us/azure/app-service/app-service-value-prop-what-is

Web Applications

TBD – Azure Web App (App Service), container, VM. Preferred way for a single web application.

Azure Service Fabric, ACS, are also options. Mainly worth it to use these if you have many microservices, orchestrator, etc. Describe orchestrators, what is ACS (deploy kubernetes, swarm, etc.) or Service Fabric.

Cesar's eBook has a table about orchestrators that I could reuse.

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 your 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 DocumentDB. Of these, DocumentDB 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 Storage Bus offers more flexibility and is the recommended service for non-trivial messaging within and between applications.

References – Azure Hosting Recommendations

Architecture

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 bind a reference architecture that maps closely to your application's requirements, or at least offers a starting point.

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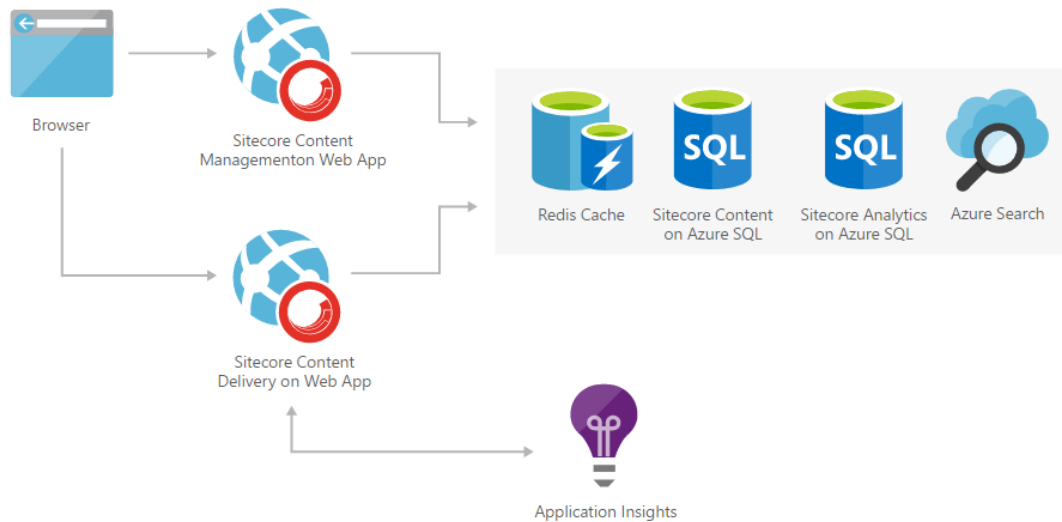
Figure 11-X. *Sitecore marketing website reference architecture.*

References – Azure Hosting Recommendations

- Azure Solution Architectures
  https://azure.microsoft.com/en-us/solutions/architecture/