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

Modular and loosely coupled

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

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

Easily tested with automated tests

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

Traditional and SPA behaviors supported

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

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

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

Simple development and deployment

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

Traditional ASP.NET and Web Forms

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

Blazor

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

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

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

References – Modern Web Applications

- **Introduction to ASP.NET Core**
  https://docs.microsoft.com/aspnet/core/

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

CHAPTER 2

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

"Atwood's Law: Any application that can be written in JavaScript, will eventually be written in JavaScript."
- Jeff Atwood

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

Use traditional web applications when:

- Your application's client-side requirements are simple or even read-only.
- Your application needs to function in browsers without JavaScript support.

Use a SPA when:

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

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

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

Blazor

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

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

Consider building your web application with Blazor when:

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

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

For more information about Blazor, see Get started with Blazor.

When to choose traditional web apps

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

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

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

Your application needs to function in browsers without JavaScript support

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

Your team is unfamiliar with JavaScript or TypeScript development techniques

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

When to choose SPAs

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

Your application must expose a rich user interface with many features

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

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

Your team is familiar with JavaScript and/or TypeScript development

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

References – SPA Frameworks

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

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

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

When to choose Blazor

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

Your application must expose a rich user interface

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

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

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

Decision table

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

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

CHAPTER 3

Architectural principles

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

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

Common design principles

Separation of concerns

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

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

Encapsulation

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

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

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

Dependency inversion

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

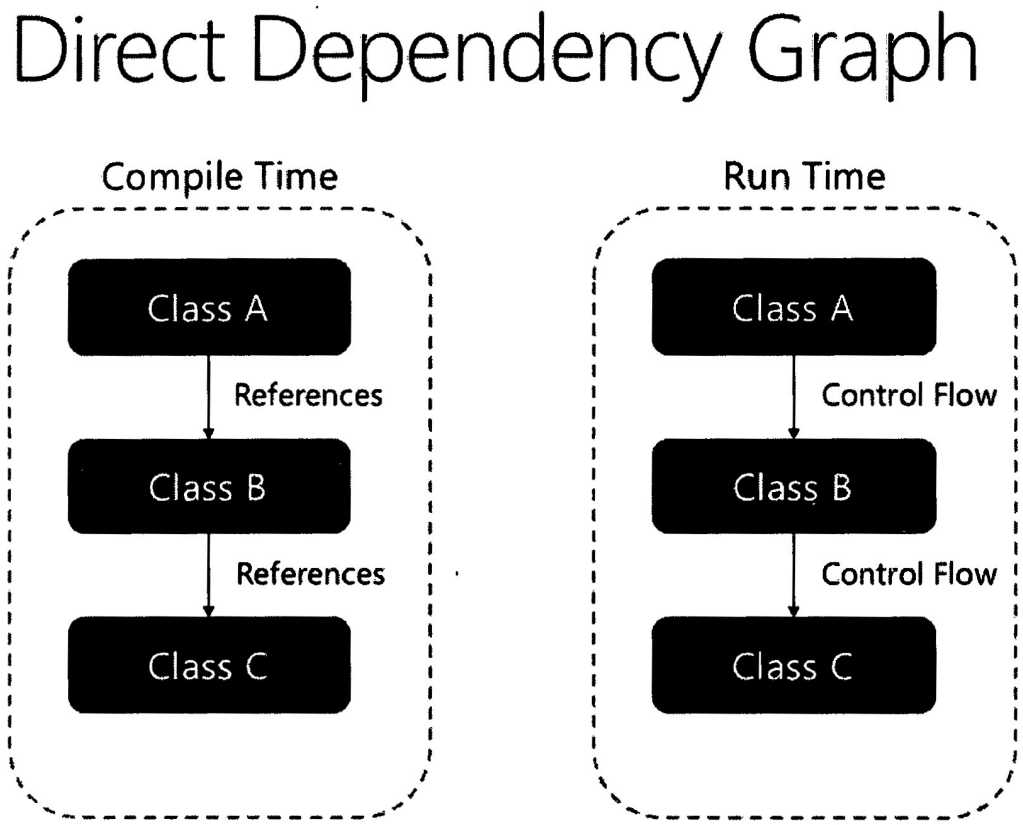

Figure 4-1. Direct dependency graph.

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

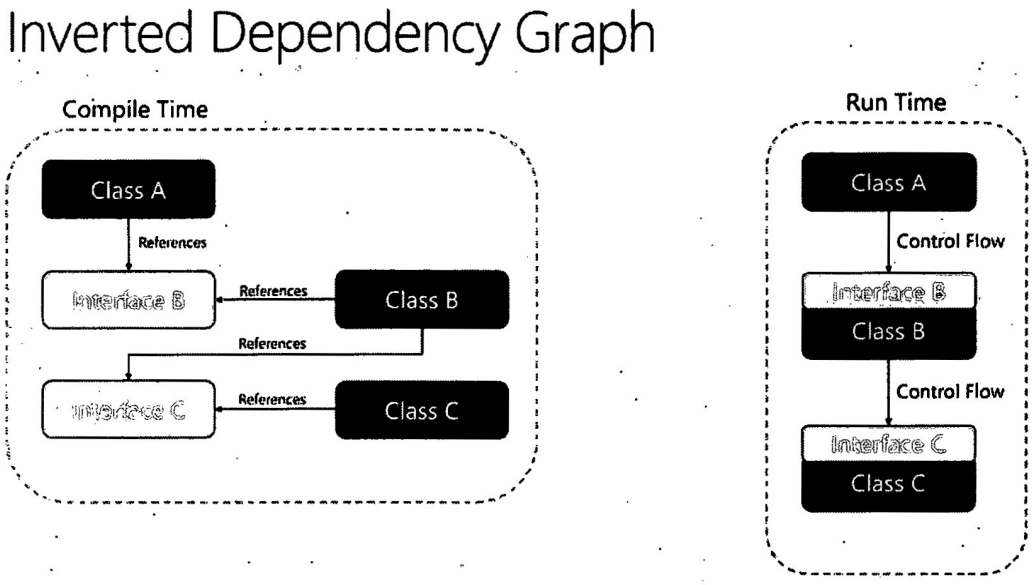

Figure 4-2. Inverted dependency graph.

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

Explicit dependencies

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

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

Single responsibility

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

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

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

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

Learn more about microservices architecture

Don't repeat yourself (DRY)

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

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

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

Persistence ignorance

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

Some examples of violations of this principle include:

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

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

Bounded contexts

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

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

Additional resources

- Principles
- Bounded Context

CHAPTER 4

Common web application architectures

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

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

What is a monolithic application?

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

All-in-one applications

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

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

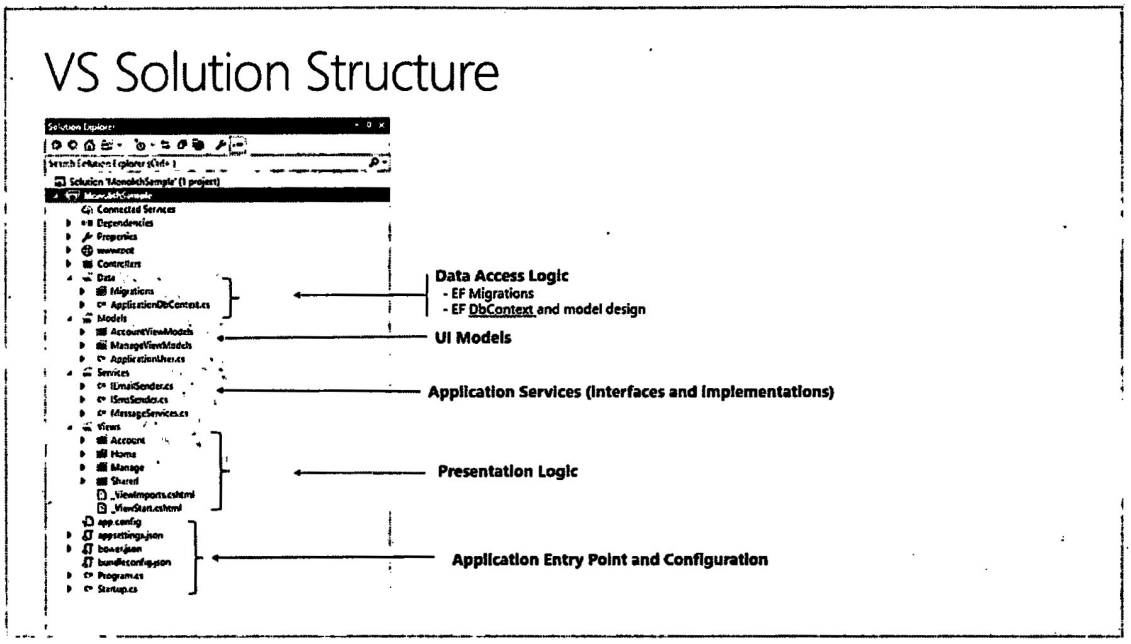

Figure 5-1. A single project ASP.NET Core app.

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

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

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

What are layers?

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

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

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

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

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

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

Note

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

Traditional "N-Layer" architecture applications

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

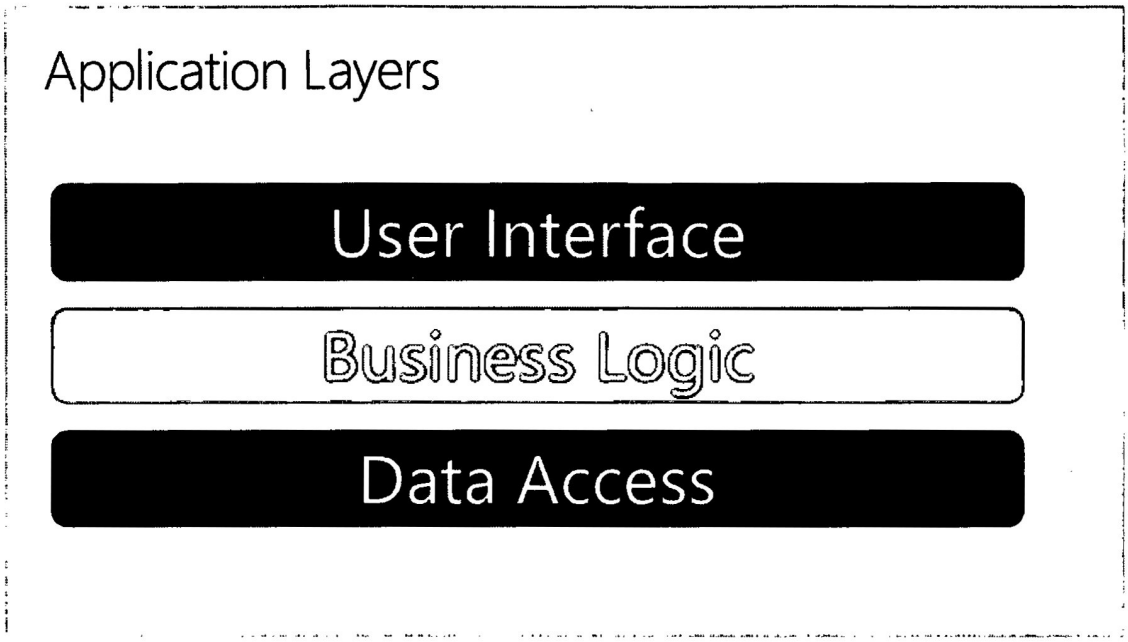

Figure 5-2. Typical application layers.

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

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

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

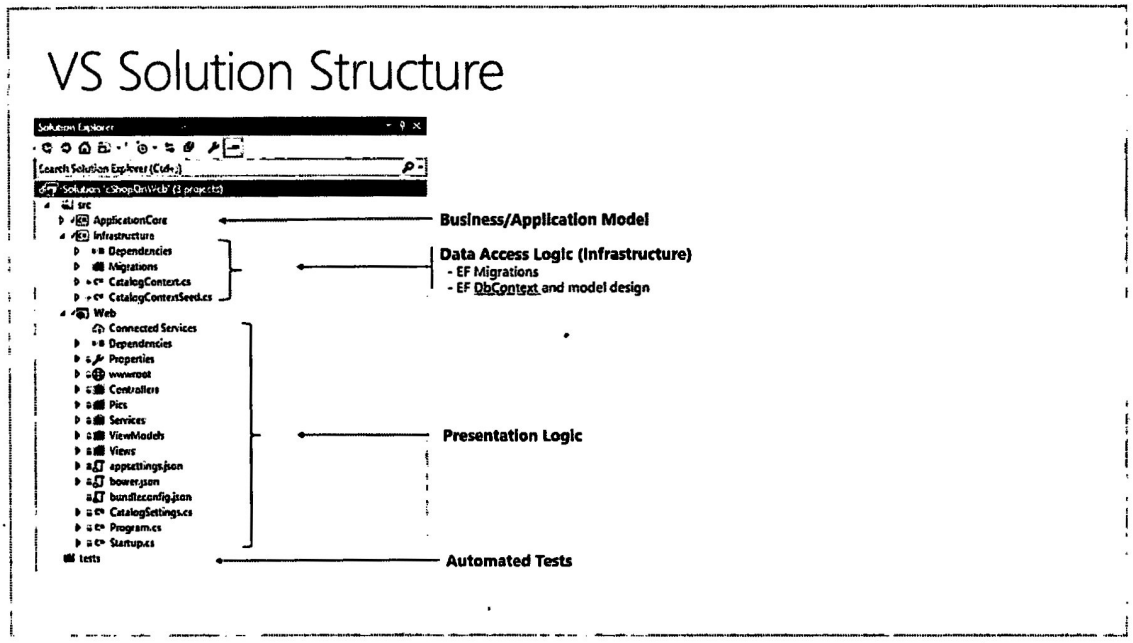

Figure 5-3. A simple monolithic application with three projects.

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

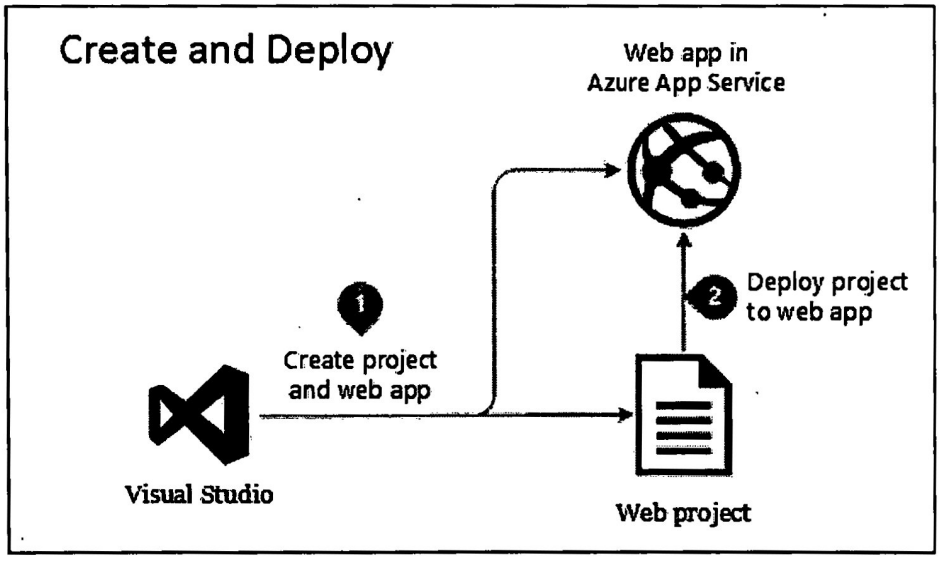

Figure 5-4. Simple deployment of Azure Web App

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

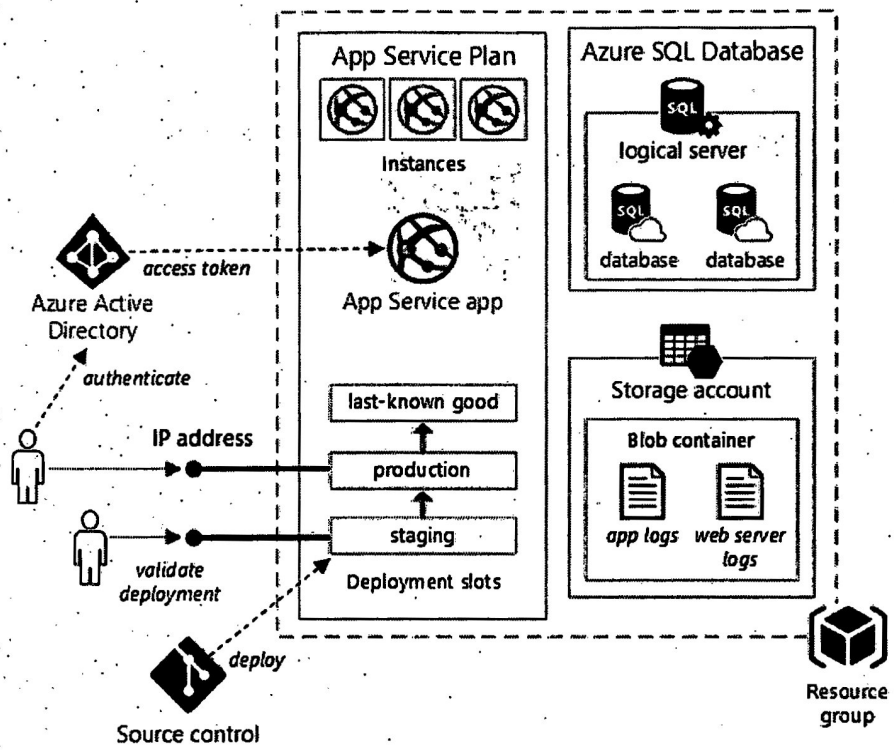

Figure 5-5. Deploying a web app to an Azure App Service

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

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

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

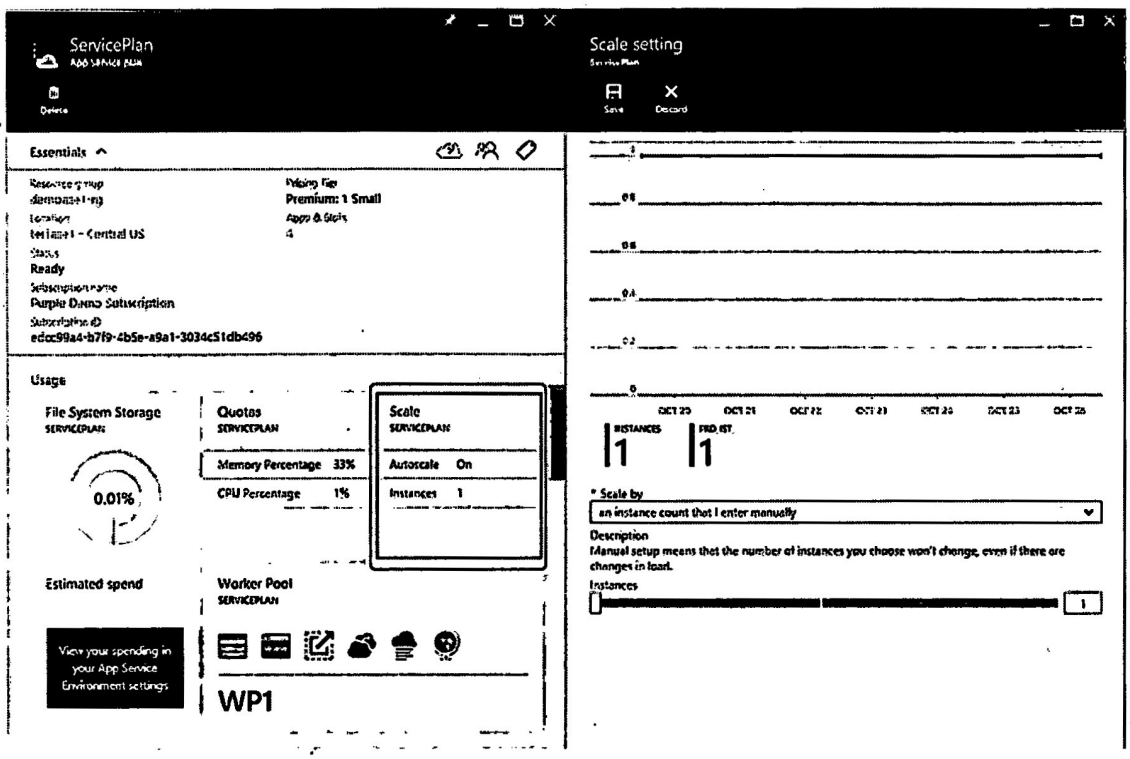

Figure 5-6. App Service Plan scaling in Azure.

Clean architecture

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

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

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

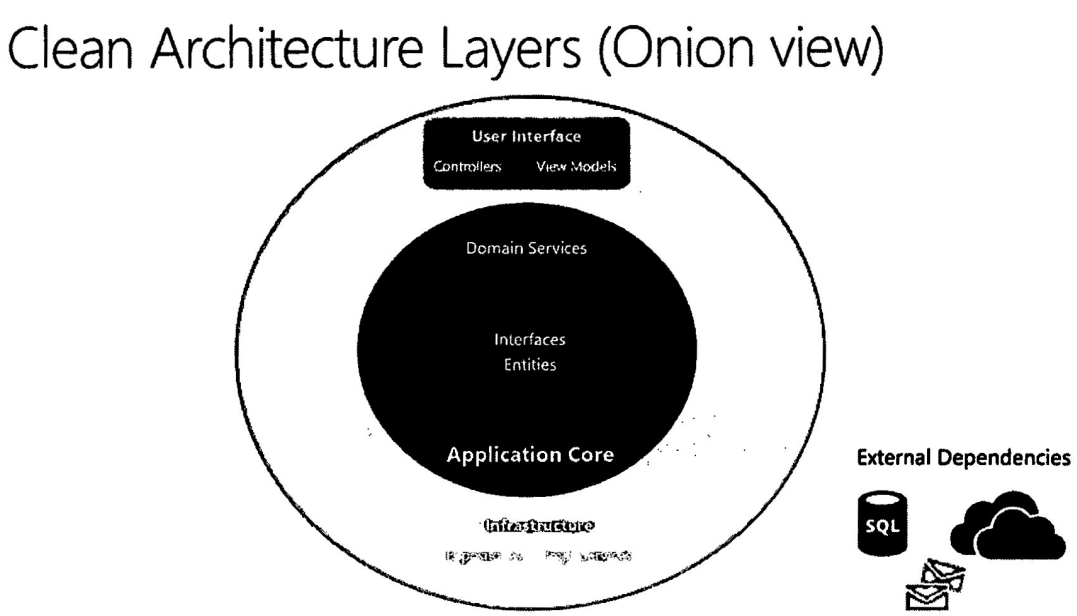

Figure 5-7. Clean Architecture; onion view

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

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