Microsoft

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

Steve Smith
DevIQ

Early Draft

PUBLISHED BY

DevDiv, .NET and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

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

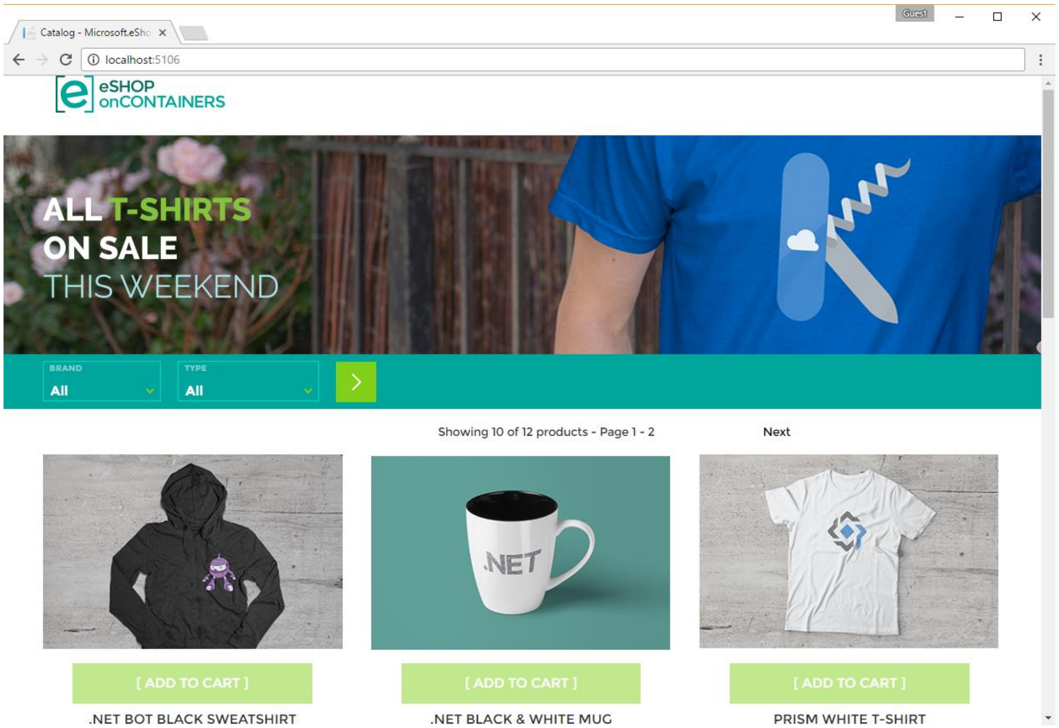
Modern web applications have higher user expectations and greater demands than ever before. Today's web apps are expected to be available 24/7 from anywhere in the world, and usable from virtually any device or screen size. Web applications must be secure, flexible, and scalable to meet spikes in demand. 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. 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. 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 A at compile time (thus, *inverting* the typical compile-time dependency).

TODO: Insert two figures demonstrating the above dependency relationships.

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).

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

SECTION 5

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?

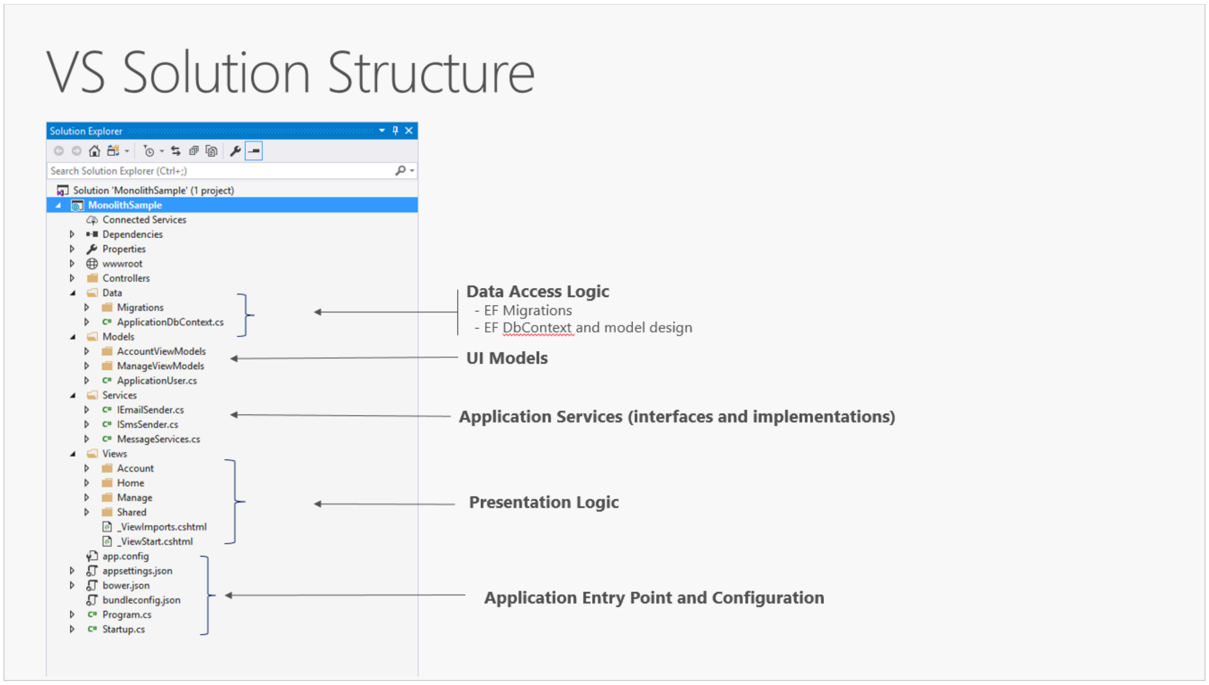
A monolithic application is one that is entirely self-contained, in terms of its behavior. It may interact with other services or data stores in the course of performing its operations, but the core of its behavior runs within its own process and the entire application is typically deployed as a single 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](https://en.wikipedia.org/wiki/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.

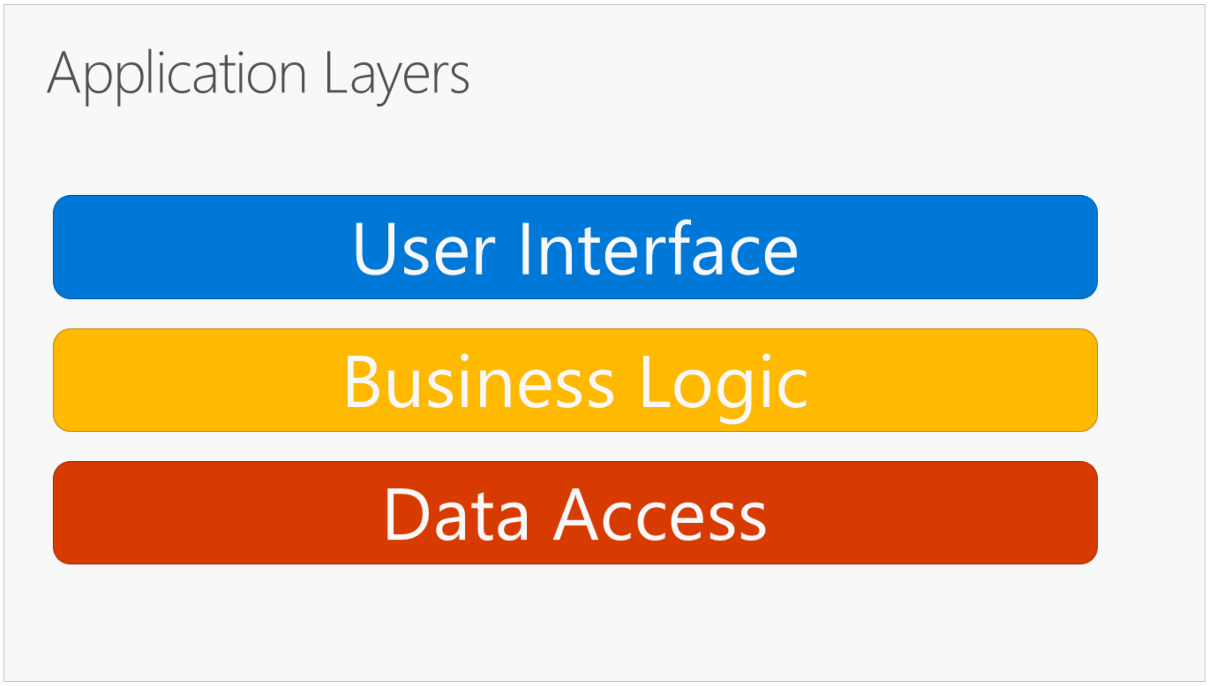
Logical layering is a common technique for improving the organization of code in enterprise software applications, and there are several ways in which code can be organized into layers.

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.

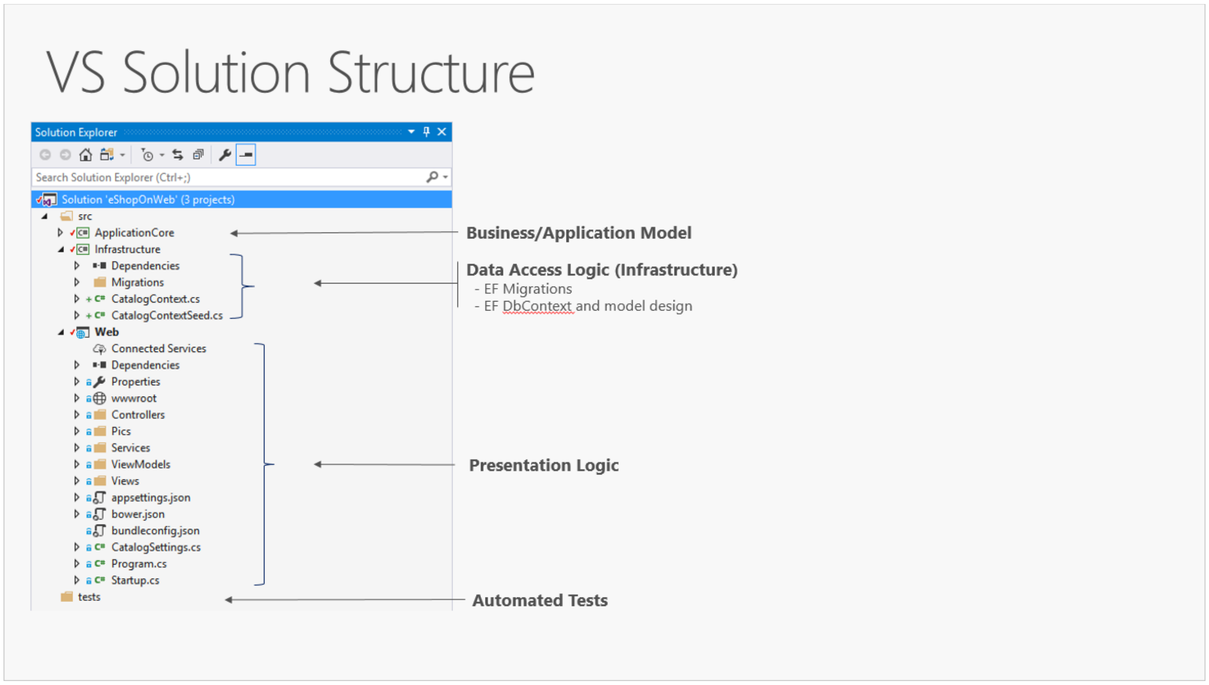


These layers are frequently abbreviated as UI, BLL (Business Logic Layer), and DAL (Data Access Layer). Using this architecture, users make requests through the UI layer, which interacts only with the BLL. The BLL, in turn, can call the DAL for data access requests. The UI layer should not make any requests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

Azure Web App

TODO: Show Azure Web App Deployment

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.

TODO: Show an example of how to use Azure to scale an application deployed as a Web App.

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 we are using 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".

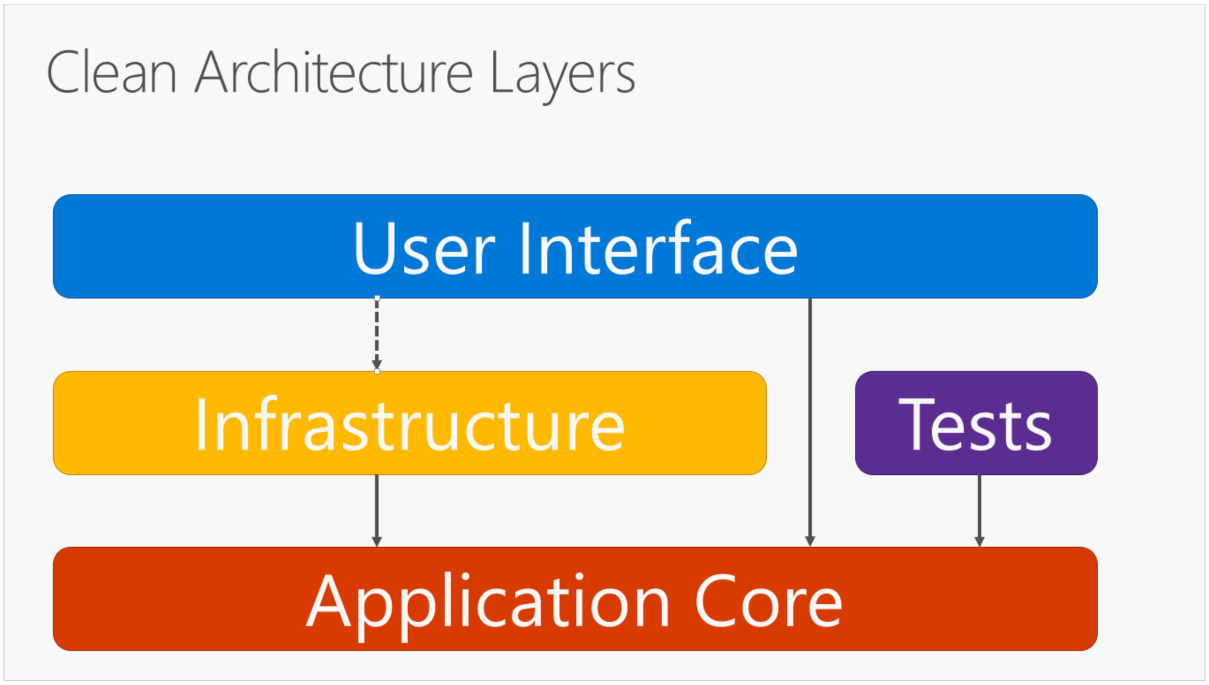
Clean architecture puts the business logic and application model at the center of the application. Instead of having business logic depend on data access or other infrastructure concerns, this dependency is inverted: infrastructure and implementation details depend on the Application Core. This is achieved 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.

Figure 5-X. Clean Architecture; onion view

TODO: Insert circular "onion" architecture diagram.

In this diagram, dependencies flow toward the center. 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. This view still doesn't offer a perfect visualization, though, since it implies that the UI layer must call through the Infrastructure (or Tests) layer in order to reach the Core layer. 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.

Because the Application Core doesn't depend on Infrastructure, it is very easy to write automated unit tests for this layer. 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.

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.

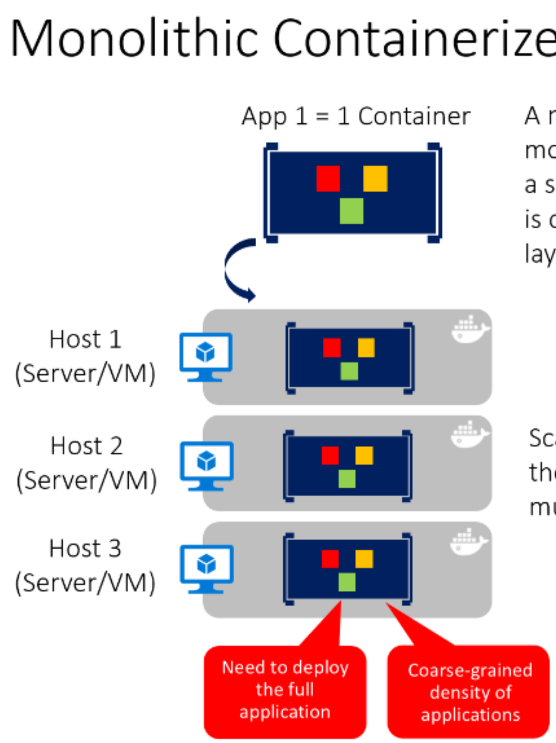


Figure 5-X. Monolithic application architecture example

You can include multiple components/libraries or internal layers within each container, as illustrated in Figure 5-X. 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.

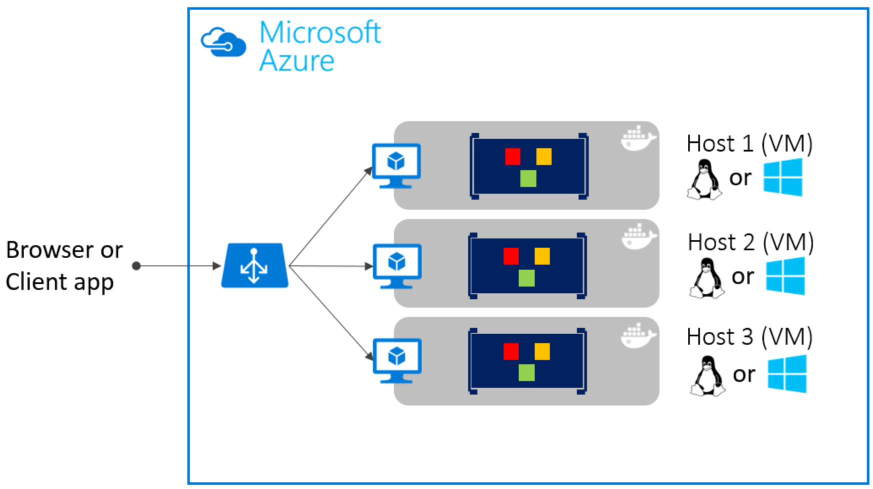


Figure 5-X. Multiple hosts scaling-out a single Docker application

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/ **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.

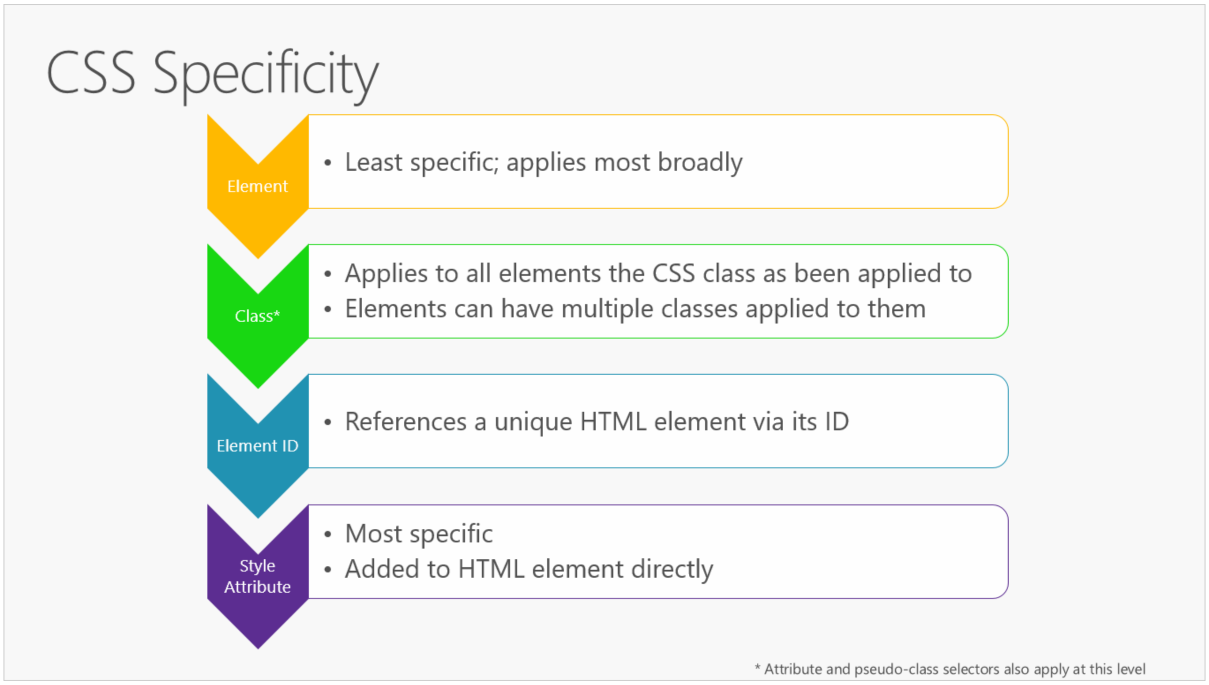
HTML

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.



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 easily be achieved using a third-party DI container like StructureMap:

TODO: Show how to use Registries to avoid need to reference Infrastructure project

Security

TBD

Client Communication

TBD – Discuss SignalR.

Domain-Driven Design – Should You Apply It?

TBD

Deployment

TBD

References – Client Web Technologies
Title https://azure.microsoft.com/en-us/blog/microservices-an-application-revolution-powered-by-the-cloud/

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

TBD

SQL or NoSQL

TBD

Entity Framework Core (for relational databases)

TBD

EF or micro-ORM?

TBD

Azure DocDB

TBD

Other Persistence Options

TBD

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

TBD

Kinds of Automated Tests

TBD

Unit Tests

TBD

Integration Tests

TBD

Functional Tests

TBD

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.

Unit Testing ASP.NET Core Apps

TBD

Integration and Functional Tests

TBD

.

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

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 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 X-XX.

Figure X-XX. 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.

When using a CLI+Editor development approach like using just **Visual Studio Code** plus **Docker CLI**, you need to know every step. If using Visual Studio Code and Docker CLI, check the eBook Containerized Docker Application lifecycle with Microsoft Platforms and Tools for explicit non-Visual Studio details.

When using **Visual Studio 2015** or **2017**, many of those steps are transparent so it dramatically improves your productivity. This is especially true when using **Visual Studio 2017** (because…).

However, making those steps transparent doesn't mean that you don't need to know what's going on underneath with dotnet and Azure. Therefore, every step is detailed in the following step-by-step guidance.

Visual Studio simplifies that workflow to "the minimum" as explained in the next sections.

Step 1. Start coding and create your initial app/service baseline

The way you develop your application is similar to the way you would do it without Docker. The difference is that while developing for Docker, you are deploying and testing your application or services running within Docker containers placed in your local environment (either a Linux VM or Windows).

Update to show installation of Azure development SDK and tools.

Setup of your local environment

With the latest version of **Docker for Windows**, it is easier than ever to develop Docker applications. The setup is straightforward, as explained in the following reference.

Installing Docker for Windows: https://docs.docker.com/docker-for-windows/

In addition, you'll need Visual Studio 2015 with the tools for Docker, or Visual Studio 2017 which includes the tooling for Docker if you selected the ".NET Core and Docker" workload during installation, as shown in Figure x-x.

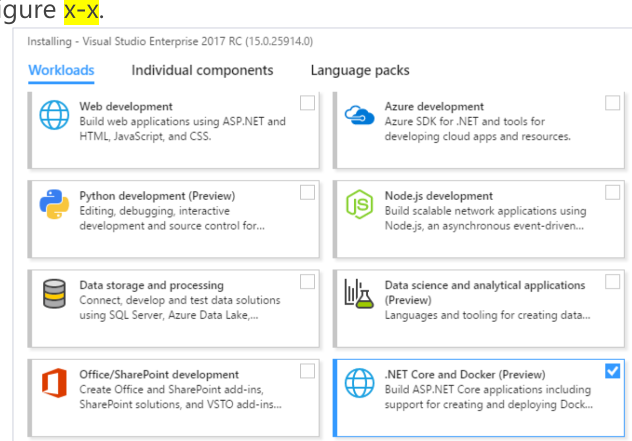


Figure X-X. Selecting the Docker and .NET Core workload

Visual Studio 2017 https://www.visualstudio.com/vs/visual-studio-2017-rc/ **Visual Studio Tools for Docker:** http://aka.ms/vstoolsfordocker

https://docs.microsoft.com/en-us/dotnet/articles/core/docker/visual-studio-tools-for-docker

Working with .NET and Visual Studio

You can start coding your app in .NET (usually in .NET Core if you are planning to use containers) even before enabling Docker in your app and deploying/testing in Docker. However, it's recommended that you start working on Docker as soon as possible, as that will be the real environment and any issues can be discovered as soon as possible. This is very much encouraged because Visual Studio makes it so easy to work with Docker that it almost feels transparent, even with debugging support with multi-container applications.

Step 2. Create a dockerfile related to an existing .NET base image

Replace with package/publish options for Azure.

References - Base Docker images
Building Docker Images for .NET Core Applications https://docs.microsoft.com/en-us/dotnet/articles/core/docker/building-net-docker-images **Build your own images** https://docs.docker.com/engine/tutorials/dockerimages/

Step 3. Create your custom Docker images embedding your service in it

Step 4. Define your services in docker-compose.yml when building a multi-container Docker app with multiple services

TBD.

Step 5. Build and run your Docker app

Run the app.

Running and debugging a multi-container application with Visual Studio

Again, when using Visual Studio 2017 it cannot get simpler and you are not only running the multi-container application but being able to debug all of its containers at once.

As mentioned before, each time you add “Docker Solution Support” to a specific project within a solution, you will get that project configured in the global/solution docker-compose.yml, so you will be able to run or debug the whole solution at once because Visual Studio will spin up a container per project that has Docker solution Support enabled while creating all the internal steps for you (dotnet publish, docker build to build the Docker images, etc.).

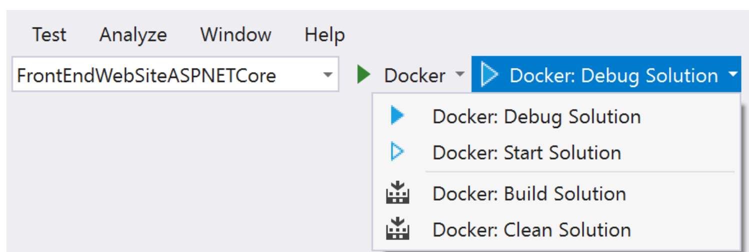


Figure X-XX. Running multi-container apps in Visual Studio 2017

The important point here is that, as shown in figure 5-26, in **Visual Studio 2017** you have an additional F5 button that we have added so you can run or debug a whole multiple container application by running all the containers that are defined in the docker-compose.yml file at the solution level that was modified by Visual Studio while adding “Docker Solution Support” to each of your projects. This means that you could set several breakpoints up, each breakpoint in a different project/container and while debugging from Visual Studio you will be stopping in breakpoints defined in different projects and running on different containers.

For further details on the services implementation and deployment to a Docker host, read the following articles.

Deploy an ASP.NET container to a remote Docker host: https://azure.microsoft.com/en-us/documentation/articles/vs-azure-tools-docker-hosting-web-apps-in-docker/

IMPORTANT NOTE: “***docker-compose up***” and “***docker run***” (or running/debugging the containers from Visual Studio which is under the covers using the same techniques) might be enough for testing your containers in your development environment, but might not be used at all if you are targeting Docker clusters and orchestrators like **Docker Swarm**, **Mesosphere DC/OS** or **Kubernetes**, in order to be able to scale-up. If using a cluster, like **[Docker Swarm mode]** (available in *Docker for Windows and Mac* since version 1.12), you need to deploy and test with additional commands like "***docker service create***" for single services or when deploying an app composed by several containers, using "***docker compose bundle***" and "***docker deploy myBundleFile***", by deploying the composed app as a "stack" as explained in the article Distributed Application Bundles, from Docker.

For DC/OS and Kubernetes you would use different deployment commands and scripts, as well.

6.
Test
your app or
microservices

Step 6. Test your Docker application (locally, in your local CD VM)

This step will vary depending on what is your app doing.

In a very simple .NET Core Web API hello world deployed as a single container/service, you'd just need to access the service by providing the TCP port specified in the dockerfile, as in the following simple example.

Testing and Debugging containers with Visual Studio

As mentioned, when running/debugging the containers with Visual Studio you'll be able to debug the .NET application running on containers in a similar way than you could do when running on the plain OS.

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.

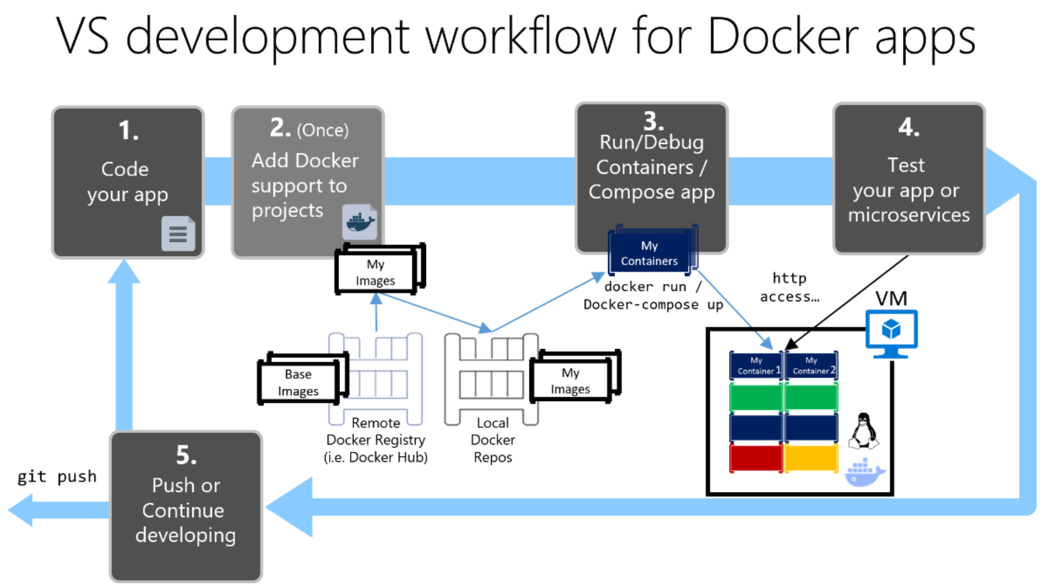


Figure X-XX. Simplified workflow when developing with Visual Studio

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 me

Web Applications

TBD – Azure Web App, container, VM

APIs

TBD – Azure Functions, microservices/container, Azure Web App

Logical Processes

TBD – Azure Functions

Data

TBD – Windows Azure SQL Database, Azure DocumentDB, Azure Storage, Redis, ,

References

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

Key takeaways

- TBD