Contents

[About .NET Core 4](#_Toc31796856)

[ASP.NET Core 4](#_Toc31796857)

[.NET Core vs ASP.NET Core 5](#_Toc31796858)

[Why ASP.NET Core? 5](#_Toc31796859)

[Command Line Tools 6](#_Toc31796860)

[Kestrel Web Server 7](#_Toc31796861)

[HTTP/2 support 7](#_Toc31796862)

[Kestrel in ASP.NET Core apps 7](#_Toc31796863)

[Kestrel options 8](#_Toc31796864)

[ASP.NET Core Middleware 10](#_Toc31796865)

[Middleware: Modules and handlers revisited 10](#_Toc31796866)

[From handlers and modules to middleware 11](#_Toc31796867)

[Create a middleware pipeline with IApplicationBuilder 12](#_Toc31796868)

[Middleware order 13](#_Toc31796869)

[Branch the middleware pipeline 15](#_Toc31796870)

[Built-in middleware 16](#_Toc31796871)

[Dependency Injection in ASP.NET Core 19](#_Toc31796872)

[Overview of dependency injection 19](#_Toc31796873)

[Services injected into Startup 22](#_Toc31796874)

[Register additional services with extension methods 22](#_Toc31796875)

[Service lifetimes 22](#_Toc31796876)

[Transient 22](#_Toc31796877)

[Scoped 23](#_Toc31796878)

[Singleton 23](#_Toc31796879)

[Configuration Framework 24](#_Toc31796880)

[Configuration changes in ASP.NET Core 24](#_Toc31796881)

[Default app configuration 25](#_Toc31796882)

[Remove default configuration options 26](#_Toc31796883)

[Asynchronous programming with async and await 30](#_Toc31796884)

[Don't block, await instead 31](#_Toc31796885)

[Start tasks concurrently 32](#_Toc31796886)

[Composition with tasks 33](#_Toc31796887)

[Await tasks efficiently 34](#_Toc31796888)

[Logging in .NET Core and ASP.NET Core 37](#_Toc31796889)

[Add providers 37](#_Toc31796890)

[Create logs 38](#_Toc31796891)

[Create logs in the Program class 39](#_Toc31796892)

[Create logs in the Startup class 41](#_Toc31796893)

[Configuration 42](#_Toc31796894)

[Log category 42](#_Toc31796895)

[Log level 43](#_Toc31796896)

[Serilog in .NET Core 3.1 44](#_Toc31796897)

[Logger initialization 44](#_Toc31796898)

[Plugging into ASP.NET Core 45](#_Toc31796899)

[Cleaning up remnants of the default logger 45](#_Toc31796900)

[Writing log events 46](#_Toc31796901)

[Recording structured logs 46](#_Toc31796902)

[Microservices using .NET Core and Docker 47](#_Toc31796903)

[Prerequisites 47](#_Toc31796904)

[Create a Dockerfile for an ASP.NET Core 2.2 application 47](#_Toc31796905)

[Build and run the Docker image 48](#_Toc31796906)

[View the web page running from a container 48](#_Toc31796907)

[ASP.NET Core MVC with Visual Studio Code 49](#_Toc31796908)

[Prerequisites 49](#_Toc31796909)

[Create a web app 49](#_Toc31796910)

[Run the app 49](#_Toc31796911)

[Add a Controller 50](#_Toc31796912)

[Add a View 52](#_Toc31796913)

[Change views and layout pages 52](#_Toc31796914)

[Passing Data from the Controller to the View 55](#_Toc31796915)

[Add a Model 56](#_Toc31796916)

[Add NuGet Packages 57](#_Toc31796917)

[Create a database context class 57](#_Toc31796918)

[Register the database context 58](#_Toc31796919)

[Add a database connection string 58](#_Toc31796920)

[Scaffold movie pages 58](#_Toc31796921)

[Initial migration 59](#_Toc31796922)

[The InitialCreate class 59](#_Toc31796923)

[Test the App 60](#_Toc31796924)

[Dependency injection in the controller 60](#_Toc31796925)

[Use SQLite for development, SQL Server for production 60](#_Toc31796926)

[Work with SQL in ASP.NET Core 61](#_Toc31796927)

[Seed the database 62](#_Toc31796928)

[Add the seed initializer 63](#_Toc31796929)

[Add search to an ASP.NET Core MVC app 65](#_Toc31796930)

[Add Search by genre 69](#_Toc31796931)

[Add search by genre to the Index view 70](#_Toc31796932)

[Add a new field 72](#_Toc31796933)

[Add a Rating Property to the Movie Model 72](#_Toc31796934)

[Use Code First Migrations to update the database schema 74](#_Toc31796935)

[Add validation 74](#_Toc31796936)

[Keeping things DRY 74](#_Toc31796937)

[Add validation rules to the movie model 75](#_Toc31796938)

[Validation Error UI 75](#_Toc31796939)

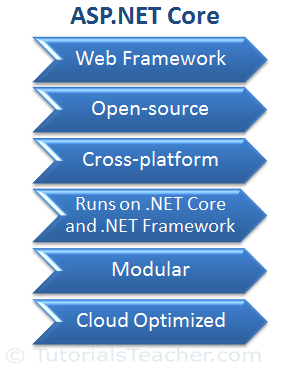
# About .NET Core

.NET Core has the following characteristics:

* **Cross-platform:** Runs on Windows, macOS, and Linux operating systems.
* **Consistent across architectures:** Runs your code with the same behavior on multiple architectures, including x64, x86, and ARM.
* **Command-line tools:** Includes easy-to-use command-line tools that can be used for local development and in continuous-integration scenarios.
* **Flexible deployment:** Can be included in your app or installed side-by-side (user-wide or system-wide installations). Can be used with Docker containers.
* **Compatible:** .NET Core is compatible with .NET Framework, Xamarin, and Mono, via .NET Standard.
* **Open source:** The .NET Core platform is open source, using MIT and Apache 2 licenses. .NET Core is a .NET Foundation project.
* **Supported by Microsoft:** .NET Core is supported by Microsoft, per .NET Core Support.

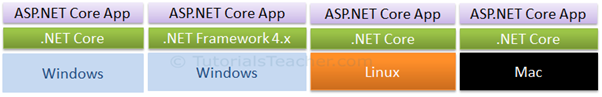
## ASP.NET Core

ASP.NET Core is a free, open-source and cloud optimized web framework which can run on Windows, Linux, or Mac. You can say that it is the new version of ASP.NET. The framework is a complete rewrite from scratch in order to make it open source, modular and cross-platform. It was initially launched as ASP.NET 5 but then it was renamed to ASP.NET Core.



ASP.NET Core is a modular framework distributed as NuGet packages. This allows us to include packages that are required in our application.

ASP.NET Core applications run on both, .NET Core and traditional .NET framework (.NET Framework 4.x).



ASP.NET Core is designed to be deployed on cloud as well as on-premises. Developers can now build cloud-based web applications, IoT (Internet of Thing) and mobile backend applications using ASP.NET Core framework which can run on Windows, Linux, and Mac operating systems.

ASP.NET Core is an open source framework supported by Microsoft and the community, so you can also contribute or download the source code from the respective repositories on Github.

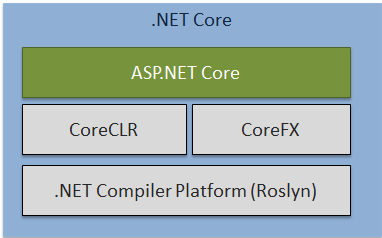
## .NET Core vs ASP.NET Core

Many people are confused between ASP.NET Core and .NET Core. Please note that ASP.NET Core and .NET Core are not the same. They are different, just like ASP.NET and .NET Framework are different.

.NET Core is a fast, lightweight, modular and open source framework for creating web applications and services that run on Windows, Linux and Mac. So, it is a platform on which ASP.NET Core application runs.

.NET Core is named "Core" because it includes core features of the .NET framework. The main objective of .NET Core is to make .NET framework open source, and cross-platform compatible so that it can be used in resource-constrained environments. It includes minimum features that are required to run a basic .NET Core app and other advanced features that can be included as a package from NuGet.

The following figure illustrates the components of .NET Core.



As you can see above, .NET Core includes .NET Compiler platform Roslyn, .NET Core runtime CoreCLR, .NET Core framework CoreFX and ASP.NET Core . ASP.NET Core is a part of .NET Core SDK so you don't need to install ASP.NET Core separately. ASP.NET Core and .NET Core is a part of .NET Foundation.

.NET Core is an open source implementation, using MIT and Apache 2 licenses.

## Why ASP.NET Core?

* **Supports Multiple Platforms:** ASP.NET Core applications can run on Windows, Linux, and Mac. So you don't need to build different apps for different platforms using different frameworks.
* **Fast:** ASP.NET Core no longer depends on System.Web.dll for browser-server communication. ASP.NET Core allows us to include packages which we need for our application. This reduces the request pipeline and improves the performance and scalability.
* **IoC Container:** It includes built-in IoC container for automatic dependency injection which makes it maintainable and testable.
* **Integration with Modern UI Frameworks:** It allows you to use and manage modern UI frameworks such as AngularJS, ReactJS, Umber, Bootstrap etc. using Bower (a package manager for the web).
* **Hosting:** ASP.NET Core web application can be hosted on multiple platforms with any web server such as IIS, Apache etc. It is not dependent only on IIS as a standard .NET Framework.
* **Code Sharing:** It allows you to build a class library which can be used with other .NET frameworks such as .NET Framework 4.x or Mono. Thus, a single code base can be shared across frameworks.
* **Side-by-Side App Versioning:** ASP.NET Core runs on .NET Core which supports simultaneous running of multiple versions of applications.
* **Smaller Deployment Footprint:** ASP.NET Core application runs on .NET Core which is smaller than full .NET Framework. So, the application which uses only a part of .NET CoreFX will have smaller deployment size. This reduces the deployment foot print.

## Command Line Tools

.NET Core applications can be created and executed from the command line using the .NET Core CLI (Command Line Interface).

So, no dependency on Visual Studio IDE. You can use the free Visual Studio Code or any other editor, like VIM.

# Kestrel Web Server

Kestrel is a cross-platform web server for ASP.NET Core. Kestrel is the web server that's included by default in ASP.NET Core project templates.

Kestrel supports the following scenarios:

* HTTPS
* Opaque upgrade used to enable WebSockets
* Unix sockets for high performance behind Nginx
* HTTP/2 (except on macOS†)

†HTTP/2 will be supported on macOS in a future release.

Kestrel is supported on all platforms and versions that .NET Core supports.

## HTTP/2 support

HTTP/2 is available for ASP.NET Core apps if the following base requirements are met:

* Operating system†
  + Windows Server 2016/Windows 10 or later‡
  + Linux with OpenSSL 1.0.2 or later (for example, Ubuntu 16.04 or later)
* Target framework: .NET Core 2.2 or later
* Application-Layer Protocol Negotiation (ALPN) connection
* TLS 1.2 or later connection

†HTTP/2 will be supported on macOS in a future release. ‡Kestrel has limited support for HTTP/2 on Windows Server 2012 R2 and Windows 8.1. Support is limited because the list of supported TLS cipher suites available on these operating systems is limited. A certificate generated using an Elliptic Curve Digital Signature Algorithm (ECDSA) may be required to secure TLS connections.

If an HTTP/2 connection is established, HttpRequest.Protocol reports HTTP/2.

HTTP/2 is disabled by default.

## Kestrel in ASP.NET Core apps

ASP.NET Core project templates use Kestrel by default. In Program.cs, the ConfigureWebHostDefaults method calls UseKestrel:

public static void Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

To provide additional configuration after calling ConfigureWebHostDefaults, use ConfigureKestrel:

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.ConfigureKestrel(serverOptions =>

{

// Set properties and call methods on options

})

.UseStartup<Startup>();

});

## Kestrel options

The Kestrel web server has constraint configuration options that are especially useful in Internet-facing deployments.

Set constraints on the Limits property of the KestrelServerOptions class. The Limits property holds an instance of the KestrelServerLimits class.

The following examples use the Microsoft.AspNetCore.Server.Kestrel.Core namespace:

using Microsoft.AspNetCore.Server.Kestrel.Core;

Kestrel options, which are configured in C# code in the following examples, can also be set using a configuration provider. For example, the File Configuration Provider can load Kestrel configuration from an appsettings.json or appsettings.{Environment}.json file:

{

"Kestrel": {

"Limits": {

"MaxConcurrentConnections": 100,

"MaxConcurrentUpgradedConnections": 100

},

"DisableStringReuse": true

}

}

Use **one** of the following approaches:

* Configure Kestrel in Startup.ConfigureServices:

1. Inject an instance of IConfiguration into the Startup class. The following example assumes that the injected configuration is assigned to the Configuration property.
2. In Startup.ConfigureServices, load the Kestrel section of configuration into Kestrel's configuration:

using Microsoft.Extensions.Configuration

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

services.Configure<KestrelServerOptions>(

Configuration.GetSection("Kestrel"));

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

...

}

}

* Configure Kestrel when building the host:

In Program.cs, load the Kestrel section of configuration into Kestrel's configuration:

// using Microsoft.Extensions.DependencyInjection;

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureServices((context, services) =>

{

services.Configure<KestrelServerOptions>(

context.Configuration.GetSection("Kestrel"));

})

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

Both of the preceding approaches work with any configuration provider.

# ASP.NET Core Middleware

Middleware is software that's assembled into an app pipeline to handle requests and responses. Each component:

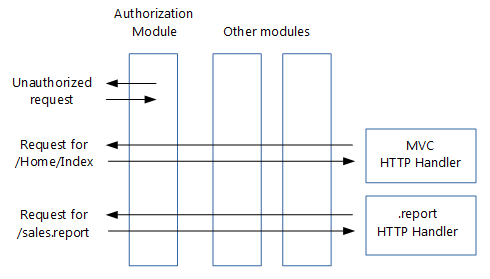
* Chooses whether to pass the request to the next component in the pipeline.
* Can perform work before and after the next component in the pipeline.

Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

Request delegates are configured using Run, Map, and Use extension methods. An individual request delegate can be specified in-line as an anonymous method (called in-line middleware), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are *middleware*, also called *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the pipeline. When a middleware short-circuits, it's called a *terminal middleware* because it prevents further middleware from processing the request.

## Middleware: Modules and handlers revisited

Before proceeding to ASP.NET Core middleware, let's first recap how HTTP modules and handlers work:



**Handlers are:**

* Classes that implement IHttpHandler
* Used to handle requests with a given file name or extension, such as *.report*
* Configured in *Web.config*

**Modules are:**

* Classes that implement IHttpModule
* Invoked for every request
* Able to short-circuit (stop further processing of a request)
* Able to add to the HTTP response, or create their own
* Configured in *Web.config*

**The order in which modules process incoming requests is determined by:**

1. The application life cycle, which is a series events fired by ASP.NET: BeginRequest, AuthenticateRequest, etc. Each module can create a handler for one or more events.
2. For the same event, the order in which they're configured in Web.config.

In addition to modules, you can add handlers for the life cycle events to your Global.asax.cs file. These handlers run after the handlers in the configured modules.

### From handlers and modules to middleware

**Middleware are simpler than HTTP modules and handlers:**

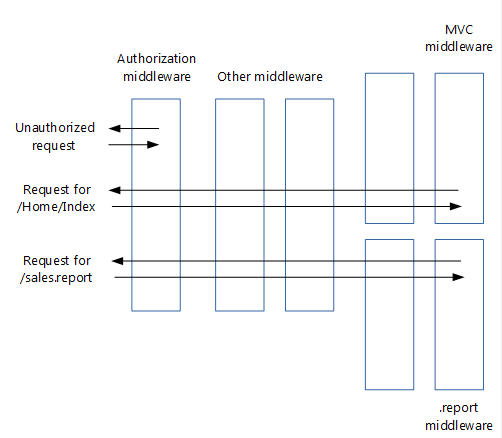
* Modules, handlers, *Global.asax.cs*, *Web.config* (except for IIS configuration) and the application life cycle are gone
* The roles of both modules and handlers have been taken over by middleware
* Middleware are configured using code rather than in *Web.config*
* Pipeline branching lets you send requests to specific middleware, based on not only the URL but also on request headers, query strings, etc.

**Middleware are very similar to modules:**

* Invoked in principle for every request
* Able to short-circuit a request, by not passing the request to the next middleware
* Able to create their own HTTP response

**Middleware and modules are processed in a different order:**

* Order of middleware is based on the order in which they're inserted into the request pipeline, while order of modules is mainly based on application life cycle events
* Order of middleware for responses is the reverse from that for requests, while order of modules is the same for requests and responses
* See Create a middleware pipeline with IApplicationBuilder



Note how in the image above, the authentication middleware short-circuited the request.

## Create a middleware pipeline with IApplicationBuilder

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other. The following diagram demonstrates the concept. The thread of execution follows the black arrows.



Each delegate can perform operations before and after the next delegate. Exception-handling delegates should be called early in the pipeline, so they can catch exceptions that occur in later stages of the pipeline.

The simplest possible ASP.NET Core app sets up a single request delegate that handles all requests. This case doesn't include an actual request pipeline. Instead, a single anonymous function is called in response to every HTTP request. (WebApp1.csproj)

public class Startup

{

public void Configure(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("Hello, World!");

});

}

}

Chain multiple request delegates together with Use. The next parameter represents the next delegate in the pipeline. You can short-circuit the pipeline by not calling the next parameter. You can typically perform actions both before and after the next delegate, as the following example demonstrates:

public class Startup

{

public void Configure(IApplicationBuilder app)

{

app.Use(async (context, next) =>

{

// Do work that doesn't write to the Response.

await next.Invoke();

// Do logging or other work that doesn't write to the Response.

});

app.Run(async context =>

{

await context.Response.WriteAsync("Hello from 2nd delegate.");

});

}

}

When a delegate doesn't pass a request to the next delegate, it's called short-circuiting the request pipeline. Short-circuiting is often desirable because it avoids unnecessary work. For example, Static File Middleware can act as a terminal middleware by processing a request for a static file and short-circuiting the rest of the pipeline. Middleware added to the pipeline before the middleware that terminates further processing still processes code after their next.Invoke statements.

Run delegates don't receive a next parameter. The first Run delegate is always terminal and terminates the pipeline. Run is a convention. Some middleware components may expose Run[Middleware] methods that run at the end of the pipeline:

public class Startup

{

public void Configure(IApplicationBuilder app)

{

app.Use(async (context, next) =>

{

// Do work that doesn't write to the Response.

await next.Invoke();

// Do logging or other work that doesn't write to the Response.

});

app.Run(async context =>

{

await context.Response.WriteAsync("Hello from 2nd delegate.");

});

}

}

## Middleware order

The order that middleware components are added in the Startup.Configure method defines the order in which the middleware components are invoked on requests and the reverse order for the response. The order is **critical** for security, performance, and functionality.

The following Startup.Configure method adds security related middleware components in the recommended order:

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

app.UseDatabaseErrorPage();

}

else

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

// app.UseCookiePolicy();

app.UseRouting();

// app.UseRequestLocalization();

// app.UseCors();

app.UseAuthentication();

app.UseAuthorization();

// app.UseSession();

app.UseEndpoints(endpoints =>

{

endpoints.MapRazorPages();

endpoints.MapControllerRoute(

name: "default",

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

});

}

In the preceding code:

* Middleware that is not added when creating a new web app with individual users accounts is commented out.
* Not every middleware needs to go in this exact order, but many do. For example, UseCors, UseAuthentication, and UseAuthorization must go in the order shown.

The preceeding Startup.Configure method adds middleware components for common app scenarios:

1. Exception/error handling
   * When the app runs in the Development environment:
     + Developer Exception Page Middleware (UseDeveloperExceptionPage) reports app runtime errors.
     + Database Error Page Middleware reports database runtime errors.
   * When the app runs in the Production environment:
     + Exception Handler Middleware (UseExceptionHandler) catches exceptions thrown in the following middlewares.
     + HTTP Strict Transport Security Protocol (HSTS) Middleware (UseHsts) adds the Strict-Transport-Security header.
2. HTTPS Redirection Middleware (UseHttpsRedirection) redirects HTTP requests to HTTPS.
3. Static File Middleware (UseStaticFiles) returns static files and short-circuits further request processing.
4. Cookie Policy Middleware (UseCookiePolicy) conforms the app to the EU General Data Protection Regulation (GDPR) regulations.
5. Routing Middleware (UseRouting) to route requests.
6. Authentication Middleware (UseAuthentication) attempts to authenticate the user before they're allowed access to secure resources.
7. Authorization Middleware (UseAuthorization) authorizes a user to access secure resources.
8. Session Middleware (UseSession) establishes and maintains session state. If the app uses session state, call Session Middleware after Cookie Policy Middleware and before MVC Middleware.
9. Endpoint Routing Middleware (UseEndpoints with MapRazorPages) to add Razor Pages endpoints to the request pipeline.

The following example demonstrates a middleware order where requests for static files are handled by Static File Middleware before Response Compression Middleware. Static files aren't compressed with this middleware order. The Razor Pages responses can be compressed.

public void Configure(IApplicationBuilder app)

{

// Static files aren't compressed by Static File Middleware.

app.UseStaticFiles();

app.UseResponseCompression();

app.UseEndpoints(endpoints =>

{

endpoints.MapRazorPages();

});

}

## Branch the middleware pipeline

Map extensions are used as a convention for branching the pipeline. Map branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

public class Startup

{

private static void HandleMapTest1(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("Map Test 1");

});

}

private static void HandleMapTest2(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("Map Test 2");

});

}

public void Configure(IApplicationBuilder app)

{

app.Map("/map1", HandleMapTest1);

app.Map("/map2", HandleMapTest2);

app.Run(async context =>

{

await context.Response.WriteAsync("Hello from non-Map delegate. <p>");

});

}

}

The following table shows the requests and responses from http://localhost:1234 using the previous code.

| **Request** | **Response** |
| --- | --- |
| localhost:1234 | Hello from non-Map delegate. |
| localhost:1234/map1 | Map Test 1 |
| localhost:1234/map2 | Map Test 2 |
| localhost:1234/map3 | Hello from non-Map delegate. |

When Map is used, the matched path segments are removed from HttpRequest.Path and appended to HttpRequest.PathBase for each request.

## Built-in middleware

ASP.NET Core ships with the following middleware components. The Order column provides notes on middleware placement in the request processing pipeline and under what conditions the middleware may terminate request processing. When a middleware short-circuits the request processing pipeline and prevents further downstream middleware from processing a request, it's called a terminal middleware.

| **Middleware** | **Description** | **Order** |
| --- | --- | --- |
| Authentication | Provides authentication support. | Before HttpContext.User is needed. Terminal for OAuth callbacks. |
| Authorization | Provides authorization support. | Immediately after the Authentication Middleware. |
| Cookie Policy | Tracks consent from users for storing personal information and enforces minimum standards for cookie fields, such as secure and SameSite. | Before middleware that issues cookies. Examples: Authentication, Session, MVC (TempData). |
| CORS | Configures Cross-Origin Resource Sharing. | Before components that use CORS. |
| Diagnostics | Several separate middlewares that provide a developer exception page, exception handling, status code pages, and the default web page for new apps. | Before components that generate errors. Terminal for exceptions or serving the default web page for new apps. |
| Forwarded Headers | Forwards proxied headers onto the current request. | Before components that consume the updated fields. Examples: scheme, host, client IP, method. |
| Health Check | Checks the health of an ASP.NET Core app and its dependencies, such as checking database availability. | Terminal if a request matches a health check endpoint. |
| HTTP Method Override | Allows an incoming POST request to override the method. | Before components that consume the updated method. |
| HTTPS Redirection | Redirect all HTTP requests to HTTPS. | Before components that consume the URL. |
| HTTP Strict Transport Security (HSTS) | Security enhancement middleware that adds a special response header. | Before responses are sent and after components that modify requests. Examples: Forwarded Headers, URL Rewriting. |
| MVC | Processes requests with MVC/Razor Pages. | Terminal if a request matches a route. |
| OWIN | Interop with OWIN-based apps, servers, and middleware. | Terminal if the OWIN Middleware fully processes the request. |
| Response Caching | Provides support for caching responses. | Before components that require caching. |
| Response Compression | Provides support for compressing responses. | Before components that require compression. |
| Request Localization | Provides localization support. | Before localization sensitive components. |
| Endpoint Routing | Defines and constrains request routes. | Terminal for matching routes. |
| Session | Provides support for managing user sessions. | Before components that require Session. |
| Static Files | Provides support for serving static files and directory browsing. | Terminal if a request matches a file. |
| URL Rewrite | Provides support for rewriting URLs and redirecting requests. | Before components that consume the URL. |
| WebSockets | Enables the WebSockets protocol. | Before components that are required to accept WebSocket requests. |

# Dependency Injection in ASP.NET Core

ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.

## Overview of dependency injection

A dependency is any object that another object requires. Examine the following MyDependency class with a WriteMessage method that other classes in an app depend upon:

public class MyDependency

{

public MyDependency()

{

}

public Task WriteMessage(string message)

{

Console.WriteLine(

$"MyDependency.WriteMessage called. Message: {message}");

return Task.FromResult(0);

}

}

An instance of the MyDependency class can be created to make the WriteMessage method available to a class. The MyDependency class is a dependency of the IndexModel class:

public class IndexModel : PageModel

{

MyDependency \_dependency = new MyDependency();

public async Task OnGetAsync()

{

await \_dependency.WriteMessage(

"IndexModel.OnGetAsync created this message.");

}

}

The class creates and directly depends on the MyDependency instance. Code dependencies (such as the previous example) are problematic and should be avoided for the following reasons:

* To replace MyDependency with a different implementation, the class must be modified.
* If MyDependency has dependencies, they must be configured by the class. In a large project with multiple classes depending on MyDependency, the configuration code becomes scattered across the app.
* This implementation is difficult to unit test. The app should use a mock or stub MyDependency class, which isn't possible with this approach.

Dependency injection addresses these problems through:

* The use of an interface or base class to abstract the dependency implementation.
* Registration of the dependency in a service container. ASP.NET Core provides a built-in service container, IServiceProvider. Services are registered in the app's Startup.ConfigureServices method.
* *Injection* of the service into the constructor of the class where it's used. The framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed.

In the sample app, the IMyDependency interface defines a method that the service provides to the app:

public interface IMyDependency

{

Task WriteMessage(string message);

}

This interface is implemented by a concrete type, MyDependency:

public class MyDependency : IMyDependency

{

private readonly ILogger<MyDependency> \_logger;

public MyDependency(ILogger<MyDependency> logger)

{

\_logger = logger;

}

public Task WriteMessage(string message)

{

\_logger.LogInformation(

"MyDependency.WriteMessage called. Message: {MESSAGE}",

message);

return Task.FromResult(0);

}

}

MyDependency requests an ILogger<TCategoryName> in its constructor. It's not unusual to use dependency injection in a chained fashion. Each requested dependency in turn requests its own dependencies. The container resolves the dependencies in the graph and returns the fully resolved service. The collective set of dependencies that must be resolved is typically referred to as a dependency tree, dependency graph, or object graph.

IMyDependency and ILogger<TCategoryName> must be registered in the service container. IMyDependency is registered in Startup.ConfigureServices.ILogger<TCategoryName> is registered by the logging abstractions infrastructure, so it's a framework-provided service registered by default by the framework.

The container resolves ILogger<TCategoryName> by taking advantage of (generic) open types, eliminating the need to register every (generic) constructed type:

services.AddSingleton(typeof(ILogger<>), typeof(Logger<>));

In the sample app, the IMyDependency service is registered with the concrete type MyDependency. The registration scopes the service lifetime to the lifetime of a single request.

public void ConfigureServices(IServiceCollection services)

{

services.AddRazorPages();

services.AddScoped<IMyDependency, MyDependency>();

services.AddTransient<IOperationTransient, Operation>();

services.AddScoped<IOperationScoped, Operation>();

services.AddSingleton<IOperationSingleton, Operation>();

services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

// OperationService depends on each of the other Operation types.

services.AddTransient<OperationService, OperationService>();

}

If the service's constructor requires a built in type, such as a string, the type can be injected by using configuration or the options pattern:

public class MyDependency : IMyDependency

{

public MyDependency(IConfiguration config)

{

var myStringValue = config["MyStringKey"];

// Use myStringValue

}

...

}

An instance of the service is requested via the constructor of a class where the service is used and assigned to a private field. The field is used to access the service as necessary throughout the class.

In the sample app, the IMyDependency instance is requested and used to call the service's WriteMessage method:

public class IndexModel : PageModel

{

private readonly IMyDependency \_myDependency;

public IndexModel( IMyDependency myDependency)

{

\_myDependency = myDependency;

}

public async Task OnGetAsync()

{

await \_myDependency.WriteMessage( "IndexModel.OnGetAsync created this message.");

}

}

## Services injected into Startup

Only the following service types can be injected into the Startup constructor when using the Generic Host (IHostBuilder):

* IWebHostEnvironment
* IHostEnvironment
* IConfiguration

Services can be injected into Startup.Configure:

public void Configure(IApplicationBuilder app, IOptions<MyOptions> options)

{

...

}

## Register additional services with extension methods

When a service collection extension method is available to register a service (and its dependent services, if required), the convention is to use a single Add{SERVICE\_NAME} extension method to register all of the services required by that service. The following code is an example of how to add additional services to the container using the extension methods AddDbContext<TContext> and AddIdentityCore:

public void ConfigureServices(IServiceCollection services)

{

...

services.AddDbContext<ApplicationDbContext>(options =>

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

services.AddIdentity<ApplicationUser, IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

...

}

## Service lifetimes

Choose an appropriate lifetime for each registered service. ASP.NET Core services can be configured with the following lifetimes:

### Transient

Transient lifetime services (AddTransient) are created each time they're requested from the service container. This lifetime works best for lightweight, stateless services.

### Scoped

Scoped lifetime services (AddScoped) are created once per client request (connection).

### Singleton

Singleton lifetime services (AddSingleton) are created the first time they're requested (or when Startup.ConfigureServices is run and an instance is specified with the service registration). Every subsequent request uses the same instance. If the app requires singleton behavior, allowing the service container to manage the service's lifetime is recommended. Don't implement the singleton design pattern and provide user code to manage the object's lifetime in the class.

# Configuration Framework

ASP.NET Core configuration differs greatly from standard ASP.NET.

Instead of web.config or any other way to set up the configuration we use built-in Configuration framework that comes with ASP.NET Core.

It is still key-value pairs collection at the end, but we can obtain those values from various sources. One of default sources is **appsettings.json** file that comes with all templates. ASP.NET Core 2 templates have configuration already filled with values from **appsettings.json** file, along with few other sources.

However, we can use and **combine** various sources for configuration settings (values):

* Files (JSON, XML, INI)
* Command-line arguments
* Environment variables
* In-memory .NET objects
* Azure Key Vault

We can also create custom providers and plug them into the ASP.NET Core system. These providers can use different outputs.

## Configuration changes in ASP.NET Core

With the arrival of ASP.NET Core 2, the Configuration became the first-class citizen of DI. That means that we can use IConfiguration anywhere in our app to get an instance of Configuration. That is what ASP.NET Core dependency injection container does for us.

Example of getting Configuration via DI is Startup class that comes with ASP.NET Core 2 templates (Razor Pages, MVC). Since Configuration is now set up during host building process (Program.cs) default template gets Configuration instance inside of Startup class via DI.

This is the default code for Startup class that comes with default Razor Pages / MVC template:

public class Startup

    {

        public Startup(IConfiguration configuration)

        {

            Configuration = configuration;

        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.

        public void ConfigureServices(IServiceCollection services)

        {

            services.AddMvc();

        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)

        {

            if (env.IsDevelopment())

            {

                app.UseDeveloperExceptionPage();

                app.UseBrowserLink();

            }

            else

            {

                app.UseExceptionHandler("/Error");

            }

            app.UseStaticFiles();

            app.UseMvc(routes =>

            {

                routes.MapRoute(

                    name: "default",

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

            });

        }

    }

Notice IConfiguration injected to Startup constructor, which means that Configuration was already configured and ready to use before Startup kicked off. That is because Configuration is now already pre-configured in Program.cs before using Startup class.

We can, of course, inject IConfiguration anywhere in our app, including controllers or pages:

public class ConfigInject : PageModel

{

    public ConfigInject(IConfiguration config)

    {

    }

}

## Default app configuration

With ASP.NET Core 2+, configuration is baked in ASP.NET Core now and it is available to be used throughout our app via DI. Furthermore, the default application configuration construction also happens inside of ASP.NET Core.

To be precise, this happens during the process of building web host, inside of Program.cs.

Take look at default content of Program class that comes with ASP.NET Core 2:

public class Program

{

    public static void Main(string[] args)

    {

        BuildWebHost(args).Run();

    }

    public static IWebHost BuildWebHost(string[] args) =>

        WebHost.CreateDefaultBuilder(args)

            .UseStartup<Startup>()

            .Build();

}

Magic happens at line 9, this is where default builder is created. That means where default configuration is constructed. That includes app configuration, logging, default server and few other settings.

Let’s take a look at the actual content of CreateDefaultBuilder from the ASP.NET Core source (GitHub):

public static IWebHostBuilder CreateDefaultBuilder(string[] args)

{

    var builder = new WebHostBuilder()

        .UseKestrel()

        .UseContentRoot(Directory.GetCurrentDirectory())

        .ConfigureAppConfiguration((hostingContext, config) =>

        {

            var env = hostingContext.HostingEnvironment;

            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)

                  .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);

            if (env.IsDevelopment())

            {

                var appAssembly = Assembly.Load(new AssemblyName(env.ApplicationName));

                if (appAssembly != null)

                {

                    config.AddUserSecrets(appAssembly, optional: true);

                }

            }

            config.AddEnvironmentVariables();

            if (args != null)

            {

                config.AddCommandLine(args);

            }

        })

        .ConfigureLogging((hostingContext, logging) =>

        {

            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));

            logging.AddConsole();

            logging.AddDebug();

        })

        .UseIISIntegration()

        .UseDefaultServiceProvider((context, options) =>

        {

            options.ValidateScopes = context.HostingEnvironment.IsDevelopment();

        });

    return builder;

}

You can see that both appsettings.json and appsettings that is environment specific are added to configuration. Afterwards, environment variables and command line arguments are also added to the configuration.

### Remove default configuration options

We can clear out Configuration sources list and that will nullify all the default configuration that ASP.NET Core host does for us. That is the code that we saw in CreateDefaultBuilder method.

public static IWebHost BuildWebHost(string[] args)

{

    return WebHost

        .CreateDefaultBuilder(args)

        .ConfigureAppConfiguration(

            (WebHostBuilderContext context, IConfigurationBuilder builder) =>

            {

                builder.Sources.Clear();

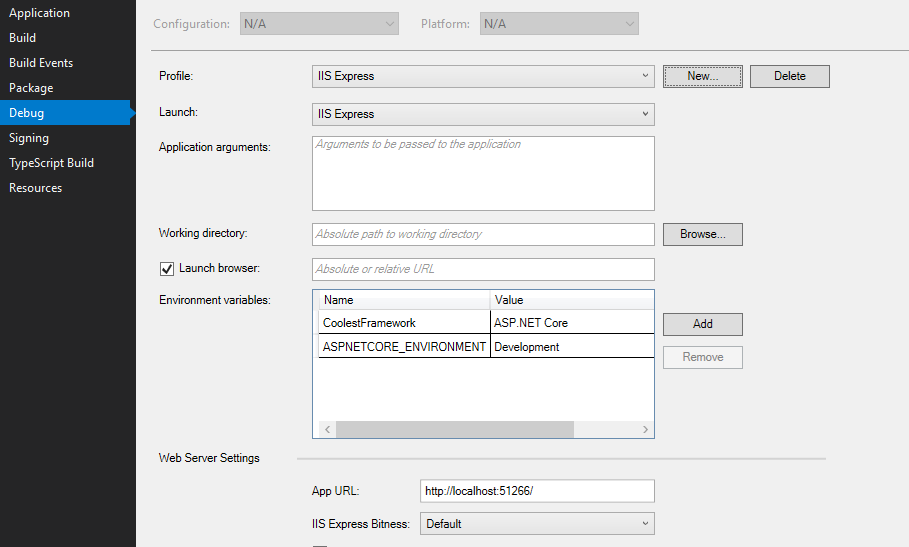
            })

        .UseStartup<Startup>()

        .Build();

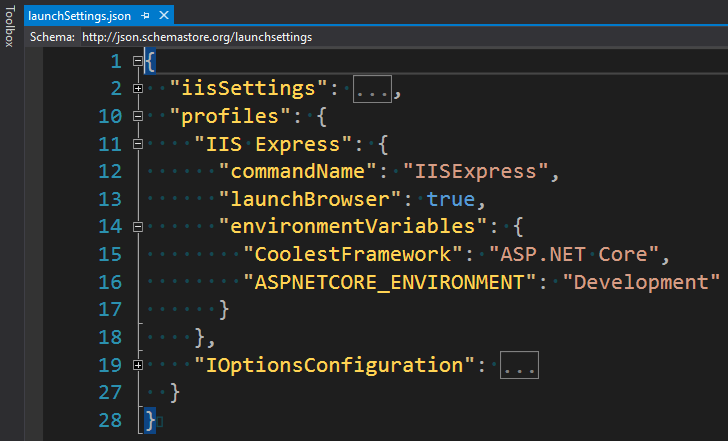
}

The line builder.Sources.Clear(); effectively removes everything that is currently set in Configuration (Env variables, appsettings.json, in memory configs etc). That means we can’t use settings from appsettings.json until we add them to the sources again.



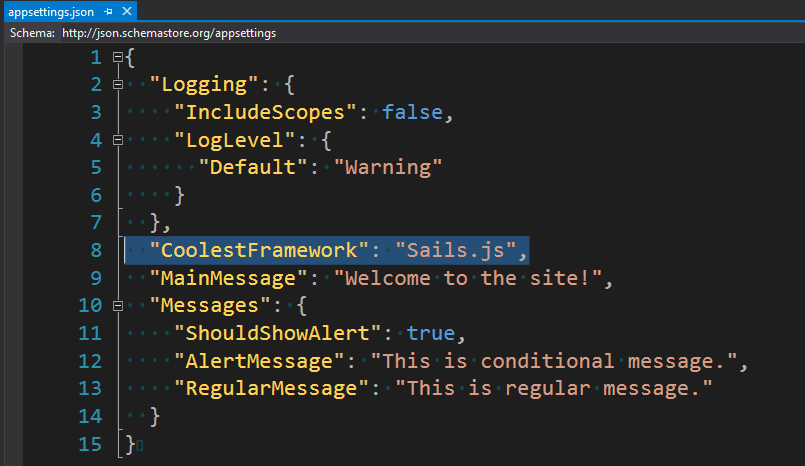
I could do this change directly via **launchSettings.json** file, by modifying profiles section. Yes, everything you change inside of Debug section withing Visual Studio will be saved to**launchSettings.json** file.

I will be starting the app via Visual Studio, which means I wanna modify my IIS Express profile:



We could of course set environment variables via command line interface. However, this is just for demonstration purposes and for debug/development session.

Now, let’s say I have following configuration in my appsettings.json file:



Do notice that now I have two same keys in my 2 separate configurations. CoolestFramework is set to “Sails.js” in **appsettings.json** source and it is set to “ASP.NET Core” via **environment variables**.

So, what happens now? Conflict? Exception?

No. **The source that is added last wins**.

Whatever comes last will override the previous configuration (key value).

Let’s see it in action:

public static IWebHost BuildWebHost(string[] args)

{

    return WebHost

        .CreateDefaultBuilder(args)

        .ConfigureAppConfiguration(

            (WebHostBuilderContext context, IConfigurationBuilder builder) =>

            {

                builder.Sources.Clear();

                builder

                    .AddEnvironmentVariables()

                    .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true);

            })

        .UseStartup<Startup>()

        .Build();

}

First, we clear existing sources that are configured by the ASP.NET Core host, by clearing out the builder.Sources list.

After that, we build the configuration and we first add environment variables and after that, we add appsettings JSON file and afterwards we add optional appsettings environment specific file.

public ConfigInject(IConfiguration config)

{

    string coolestFramework = config["CoolestFramework"];

}

This will result in coolestFramework variable having value “Sails.js”.

If we want “ASP.NET Core” (and we want that, don’t we?) to be our preferred output we would add environment variables last.

# Asynchronous programming with async and await

The Task asynchronous programming model (TAP) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs a number of transformations because some of those statements may start work and return a Task that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making a breakfast to see how the async and await keywords make it easier to reason about code that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

* Pour a cup of coffee.
* Heat up a pan, then fry two eggs.
* Fry three slices of bacon.
* Toast two pieces of bread.
* Add butter and jam to the toast.
* Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. You'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each cook (or thread) would be blocked synchronously waiting for bacon to be ready to flip, or the toast to pop.

Now, consider those same instructions written as C# statements:

static void Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("coffee is ready");

Egg eggs = FryEggs(2);

Console.WriteLine("eggs are ready");

Bacon bacon = FryBacon(3);

Console.WriteLine("bacon is ready");

Toast toast = ToastBread(2);

ApplyButter(toast);

ApplyJam(toast);

Console.WriteLine("toast is ready");

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");

}

Computers don't interpret those instructions the same way people do. The computer will block on each statement until the work is complete before moving on to the next statement. That creates an unsatisfying breakfast. The later tasks wouldn't be started until the earlier tasks had completed. It would take much longer to create the breakfast, and some items would have gotten cold before being served.

If you want the computer to execute the above instructions asynchronously, you must write asynchronous code.

These concerns are important for the programs you write today. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't make a phone appear frozen while it's downloading data from the web. When you write server programs, you don't want threads blocked. Those threads could be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for those blocked threads.

Successful modern applications require asynchronous code. Without language support, writing asynchronous code required callbacks, completion events, or other means that obscured the original intent of the code. The advantage of the synchronous code is that it's easy to understand. The step-by-step actions make it easy to scan and understand. Traditional asynchronous models forced you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

## Don't block, await instead

The preceding code demonstrates a bad practice: constructing synchronous code to perform asynchronous operations. As written, this code blocks the thread executing it from doing any other work. It won't be interrupted while any of the tasks are in progress. It would be as though you stared at the toaster after putting the bread in. You'd ignore anyone talking to you until the toast popped.

Let's start by updating this code so that the thread doesn't block while tasks are running. The await keyword provides a non-blocking way to start a task, then continue execution when that task completes. A simple asynchronous version of the make a breakfast code would look like the following snippet:

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("coffee is ready");

Egg eggs = await FryEggs(2);

Console.WriteLine("eggs are ready");

Bacon bacon = await FryBacon(3);

Console.WriteLine("bacon is ready");

Toast toast = await ToastBread(2);

ApplyButter(toast);

ApplyJam(toast);

Console.WriteLine("toast is ready");

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");

}

This code doesn't block while the eggs or the bacon are cooking. This code won't start any other tasks though. You'd still put the toast in the toaster and stare at it until it pops. But at least, you'd respond to anyone that wanted your attention. In a restaurant where multiple orders are placed, the cook could start another breakfast while the first is cooking.

Now, the thread working on the breakfast isn't blocked while awaiting any started task that hasn't yet finished. For some applications, this change is all that's needed. A GUI application still responds to the user with just this change. However, for this scenario, you want more. You don't want each of the component tasks to be executed sequentially. It's better to start each of the component tasks before awaiting the previous task's completion.

## Start tasks concurrently

In many scenarios, you want to start several independent tasks immediately. Then, as each task finishes, you can continue other work that's ready. In the breakfast analogy, that's how you get breakfast done more quickly. You also get everything done close to the same time. You'll get a hot breakfast.

The System.Threading.Tasks.Task and related types are classes you can use to reason about tasks that are in progress. That enables you to write code that more closely resembles the way you'd actually create breakfast. You'd start cooking the eggs, bacon, and toast at the same time. As each requires action, you'd turn your attention to that task, take care of the next action, then await for something else that requires your attention.

You start a task and hold on to the Task object that represents the work. You'll await each task before working with its result.

Let's make these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than awaiting them:

Coffee cup = PourCoffee();

Console.WriteLine("coffee is ready");

Task<Egg> eggsTask = FryEggs(2);

Egg eggs = await eggsTask;

Console.WriteLine("eggs are ready");

Task<Bacon> baconTask = FryBacon(3);

Bacon bacon = await baconTask;

Console.WriteLine("bacon is ready");

Task<Toast> toastTask = ToastBread(2);

Toast toast = await toastTask;

ApplyButter(toast);

ApplyJam(toast);

Console.WriteLine("toast is ready");

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");

Next, you can move the await statements for the bacon and eggs to the end of the method, before serving breakfast:

Coffee cup = PourCoffee();

Console.WriteLine("coffee is ready");

Task<Egg> eggsTask = FryEggs(2);

Task<Bacon> baconTask = FryBacon(3);

Task<Toast> toastTask = ToastBread(2);

Toast toast = await toastTask;

ApplyButter(toast);

ApplyJam(toast);

Console.WriteLine("toast is ready");

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Egg eggs = await eggsTask;

Console.WriteLine("eggs are ready");

Bacon bacon = await baconTask;

Console.WriteLine("bacon is ready");

Console.WriteLine("Breakfast is ready!");

The preceding code works better. You start all the asynchronous tasks at once. You await each task only when you need the results. The preceding code may be similar to code in a web application that makes requests of different microservices, then combines the results into a single page. You'll make all the requests immediately, then await all those tasks and compose the web page.

## Composition with tasks

You have everything ready for breakfast at the same time except the toast. Making the toast is the composition of an asynchronous operation (toasting the bread), and synchronous operations (adding the butter and the jam). Updating this code illustrates an important concept:

The preceding code showed you that you can use Task or Task<TResult> objects to hold running tasks. You await each task before using its result. The next step is to create methods that represent the combination of other work. Before serving breakfast, you want to await the task that represents toasting the bread before adding butter and jam. You can represent that work with the following code:

async Task<Toast> MakeToastWithButterAndJamAsync(int number)

{

var toast = await ToastBreadAsync(number);

ApplyButter(toast);

ApplyJam(toast);

return toast;

}

The preceding method has the async modifier in its signature. That signals to the compiler that this method contains an await statement; it contains asynchronous operations. This method represents the task that toasts the bread, then adds butter and jam. This method returns a Task<TResult> that represents the composition of those three operations. The main block of code now becomes:

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("coffee is ready");

var eggsTask = FryEggsAsync(2);

var baconTask = FryBaconAsync(3);

var toastTask = MakeToastWithButterAndJamAsync(2);

var eggs = await eggsTask;

Console.WriteLine("eggs are ready");

var bacon = await baconTask;

Console.WriteLine("bacon is ready");

var toast = await toastTask;

Console.WriteLine("toast is ready");

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");

async Task<Toast> MakeToastWithButterAndJamAsync(int number)

{

var toast = await ToastBreadAsync(number);

ApplyButter(toast);

ApplyJam(toast);

return toast;

}

}

The previous change illustrated an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to await that task. You can start other tasks concurrently.

## Await tasks efficiently

The series of await statements at the end of the preceding code can be improved by using methods of the Task class. One of those APIs is WhenAll, which returns a Task that completes when all the tasks in its argument list have completed, as shown in the following code:

await Task.WhenAll(eggsTask, baconTask, toastTask);

Console.WriteLine("eggs are ready");

Console.WriteLine("bacon is ready");

Console.WriteLine("toast is ready");

Console.WriteLine("Breakfast is ready!");

Another option is to use WhenAny, which returns a Task<Task> that completes when any of its arguments completes. You can await the returned task, knowing that it has already finished. The following code shows how you could use WhenAny to await the first task to finish and then process its result. After processing the result from the completed task, you remove that completed task from the list of tasks passed to WhenAny.

var allTasks = new List<Task>{eggsTask, baconTask, toastTask};

while (allTasks.Any())

{

Task finished = await Task.WhenAny(allTasks);

if (finished == eggsTask)

{

Console.WriteLine("eggs are ready");

}

else if (finished == baconTask)

{

Console.WriteLine("bacon is ready");

}

else if (finished == toastTask)

{

Console.WriteLine("toast is ready");

}

allTasks.Remove(finished);

}

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");

After all those changes, the final version of Main looks like the following code:

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("coffee is ready");

var eggsTask = FryEggsAsync(2);

var baconTask = FryBaconAsync(3);

var toastTask = MakeToastWithButterAndJamAsync(2);

var allTasks = new List<Task>{eggsTask, baconTask, toastTask};

while (allTasks.Any())

{

Task finished = await Task.WhenAny(allTasks);

if (finished == eggsTask)

{

Console.WriteLine("eggs are ready");

}

else if (finished == baconTask)

{

Console.WriteLine("bacon is ready");

}

else if (finished == toastTask)

{

Console.WriteLine("toast is ready");

}

allTasks.Remove(finished);

}

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");

async Task<Toast> MakeToastWithButterAndJamAsync(int number)

{

var toast = await ToastBreadAsync(number);

ApplyButter(toast);

ApplyJam(toast);

return toast;

}

}

This final code is asynchronous. It more accurately reflects how a person would cook a breakfast. Compare the preceding code with the first code sample in this article. The core actions are still clear from reading the code. You can read this code the same way you'd read those instructions for making a breakfast at the beginning of this article. The language features for async and await provide the translation every person makes to follow those written instructions: start tasks as you can and don't block waiting for tasks to complete.

# Logging in .NET Core and ASP.NET Core

.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. This article shows how to use the logging API with built-in providers.

## Add providers

A logging provider displays or stores logs. For example, the Console provider displays logs on the console, and the Azure Application Insights provider stores them in Azure Application Insights. Logs can be sent to multiple destinations by adding multiple providers.

To add a provider in an app that uses Generic Host, call the provider's Add{provider name} extension method in Program.cs:

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureLogging(logging =>

{

logging.ClearProviders();

logging.AddConsole();

})

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

In a non-host console app, call the provider's Add{provider name} extension method while creating a LoggerFactory:

var loggerFactory = LoggerFactory.Create(builder =>

{

builder

.AddFilter("Microsoft", LogLevel.Warning)

.AddFilter("System", LogLevel.Warning)

.AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)

.AddConsole()

.AddEventLog();

});

ILogger logger = loggerFactory.CreateLogger<Program>();

logger.LogInformation("Example log message");

LoggerFactory and AddConsole require a using statement for Microsoft.Extensions.Logging.

The default ASP.NET Core project templates call CreateDefaultBuilder, which adds the following logging providers:

* Console
* Debug
* EventSource
* EventLog (only when running on Windows)

You can replace the default providers with your own choices. Call ClearProviders, and add the providers you want.

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureLogging(logging =>

{

logging.ClearProviders();

logging.AddConsole();

})

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

## Create logs

To create logs, use an ILogger<TCategoryName> object. In a web app or hosted service, get an ILogger from dependency injection (DI). In non-host console apps, use the LoggerFactory to create an ILogger.

The following ASP.NET Core example creates a logger with TodoApiSample.Pages.AboutModel as the category. The log category is a string that is associated with each log. The ILogger<T> instance provided by DI creates logs that have the fully qualified name of type T as the category.

public class AboutModel : PageModel

{

private readonly ILogger \_logger;

public AboutModel(ILogger<AboutModel> logger)

{

\_logger = logger;

}

The following non-host console app example creates a logger with LoggingConsoleApp.Program as the category.

var loggerFactory = LoggerFactory.Create(builder =>

{

builder

.AddFilter("Microsoft", LogLevel.Warning)

.AddFilter("System", LogLevel.Warning)

.AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)

.AddConsole()

.AddEventLog();

});

ILogger logger = loggerFactory.CreateLogger<Program>();

logger.LogInformation("Example log message");

In the following ASP.NET Core and console app examples, the logger is used to create logs with Information as the level. The Log level indicates the severity of the logged event.

public void OnGet()

{

Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";

\_logger.LogInformation("Message displayed: {Message}", Message);

}

var loggerFactory = LoggerFactory.Create(builder =>

{

builder

.AddFilter("Microsoft", LogLevel.Warning)

.AddFilter("System", LogLevel.Warning)

.AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)

.AddConsole()

.AddEventLog();

});

ILogger logger = loggerFactory.CreateLogger<Program>();

logger.LogInformation("Example log message");

### Create logs in the Program class

To write logs in the Program class of an ASP.NET Core app, get an ILogger instance from DI after building the host:

public static void Main(string[] args)

{

var host = CreateHostBuilder(args).Build();

var todoRepository = host.Services.GetRequiredService<ITodoRepository>();

todoRepository.Add(new Core.Model.TodoItem() { Name = "Feed the dog" });

todoRepository.Add(new Core.Model.TodoItem() { Name = "Walk the dog" });

var logger = host.Services.GetRequiredService<ILogger<Program>>();

logger.LogInformation("Seeded the database.");

IMyService myService = host.Services.GetRequiredService<IMyService>();

myService.WriteLog("Logged from MyService.");

host.Run();

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

Logging during host construction isn't directly supported. However, a separate logger can be used. In the following example, a Serilog logger is used to log in CreateHostBuilder. AddSerilog uses the static configuration specified in Log.Logger:

using System;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.Hosting;

using Microsoft.Extensions.Logging;

public class Program

{

public static void Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

public static IHostBuilder CreateHostBuilder(string[] args)

{

var builtConfig = new ConfigurationBuilder()

.AddJsonFile("appsettings.json")

.AddCommandLine(args)

.Build();

Log.Logger = new LoggerConfiguration()

.WriteTo.Console()

.WriteTo.File(builtConfig["Logging:FilePath"])

.CreateLogger();

try

{

return Host.CreateDefaultBuilder(args)

.ConfigureServices((context, services) =>

{

services.AddRazorPages();

})

.ConfigureAppConfiguration((hostingContext, config) =>

{

config.AddConfiguration(builtConfig);

})

.ConfigureLogging(logging =>

{

logging.AddSerilog();

})

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

}

catch (Exception ex)

{

Log.Fatal(ex, "Host builder error");

throw;

}

finally

{

Log.CloseAndFlush();

}

}

}

### Create logs in the Startup class

To write logs in the Startup.Configure method of an ASP.NET Core app, include an ILogger parameter in the method signature:

public void Configure(IApplicationBuilder app, IHostEnvironment env, ILogger<Startup> logger)

{

if (env.IsDevelopment())

{

logger.LogInformation("In Development environment");

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

endpoints.MapRazorPages();

});

}

Writing logs before completion of the DI container setup in the Startup.ConfigureServices method is not supported:

* Logger injection into the Startup constructor is not supported.
* Logger injection into the Startup.ConfigureServices method signature is not supported

The reason for this restriction is that logging depends on DI and on configuration, which in turns depends on DI. The DI container isn't set up until ConfigureServices finishes.

## Configuration

Logging provider configuration is provided by one or more configuration providers:

* File formats (INI, JSON, and XML).
* Command-line arguments.
* Environment variables.
* In-memory .NET objects.
* The unencrypted Secret Manager storage.
* An encrypted user store, such as Azure Key Vault.
* Custom providers (installed or created).

For example, logging configuration is commonly provided by the Logging section of app settings files. The following example shows the contents of a typical appsettings.Development.json file:

{

"Logging": {

"LogLevel": {

"Default": "Debug",

"System": "Information",

"Microsoft": "Information"

},

"Console":

{

"IncludeScopes": true

}

}

}

The Logging property can have LogLevel and log provider properties (Console is shown).

The LogLevel property under Logging specifies the minimum level to log for selected categories. In the example, System and Microsoft categories log at Information level, and all others log at Debug level.

## Log category

When an ILogger object is created, a category is specified for it. That category is included with each log message created by that instance of ILogger. The category may be any string, but the convention is to use the class name, such as "TodoApi.Controllers.TodoController".

Use ILogger<T> to get an ILogger instance that uses the fully qualified type name of T as the category:

public class TodoController : Controller

{

private readonly ITodoRepository \_todoRepository;

private readonly ILogger \_logger;

public TodoController(ITodoRepository todoRepository,

ILogger<TodoController> logger)

{

\_todoRepository = todoRepository;

\_logger = logger;

}

To explicitly specify the category, call ILoggerFactory.CreateLogger:

public class TodoController : Controller

{

private readonly ITodoRepository \_todoRepository;

private readonly ILogger \_logger;

public TodoController(ITodoRepository todoRepository,

ILoggerFactory logger)

{

\_todoRepository = todoRepository;

\_logger = logger.CreateLogger("TodoApiSample.Controllers.TodoController");

}

ILogger<T> is equivalent to calling CreateLogger with the fully qualified type name of T.

## Log level

Every log specifies a LogLevel value. The log level indicates the severity or importance. For example, you might write an Information log when a method ends normally and a Warning log when a method returns a 404 Not Found status code.

The following code creates Information and Warning logs:

public IActionResult GetById(string id)

{

\_logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);

var item = \_todoRepository.Find(id);

if (item == null)

{

\_logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);

return NotFound();

}

return new ObjectResult(item);

}

In the preceding code, the first parameter is the Log event ID. The second parameter is a message template with placeholders for argument values provided by the remaining method parameters. The method parameters are explained in the message template section later in this article.

Log methods that include the level in the method name (for example, LogInformation and LogWarning) are extension methods for ILogger. These methods call a Log method that takes a LogLevel parameter. You can call the Log method directly rather than one of these extension methods, but the syntax is relatively complicated.

ASP.NET Core defines the following log levels, ordered here from lowest to highest severity.

* Trace = 0

For information that's typically valuable only for debugging. These messages may contain sensitive application data and so shouldn't be enabled in a production environment. *Disabled by default.*

* Debug = 1

For information that may be useful in development and debugging. Example: Entering method Configure with flag set to true. Enable Debug level logs in production only when troubleshooting, due to the high volume of logs.

* Information = 2

For tracking the general flow of the app. These logs typically have some long-term value. Example: Request received for path /api/todo

* Warning = 3

For abnormal or unexpected events in the app flow. These may include errors or other conditions that don't cause the app to stop but might need to be investigated. Handled exceptions are a common place to use the Warning log level. Example: FileNotFoundException for file quotes.txt.

* Error = 4

For errors and exceptions that cannot be handled. These messages indicate a failure in the current activity or operation (such as the current HTTP request), not an app-wide failure. Example log message: Cannot insert record due to duplicate key violation.

* Critical = 5

For failures that require immediate attention. Examples: data loss scenarios, out of disk space.

## Serilog in .NET Core 3.1

Install-package Serilog.AspNetCore

## Logger initialization

Logger initialization in **Program.cs** is done as follows:

**using** Serilog;

**public** **static** **void** **Main**(**string**[] args)

{

Log.Logger = **new** **LoggerConfiguration**()

.Enrich.**FromLogContext**()

.WriteTo.**Console**()

.**CreateLogger**();

**try**

{

Log.**Information**("Starting up");

**CreateHostBuilder**(args).**Build**().**Run**();

}

**catch** (Exception ex)

{

Log.**Fatal**(ex, "Application start-up failed");

}

**finally**

{

Log.**CloseAndFlush**();

}

}

## Plugging into ASP.NET Core

If you run the application now, you’ll see Serilog’s and ASP.NET Core’s log output side-by-side. Our goal is to have all log events processed through the same (Serilog) logging pipeline, and to do that we add UsingSerilog() into Program.cs’s CreateHostBuilder() method:

**public** **static** IHostBuilder **CreateHostBuilder**(**string**[] args) =>

Host.**CreateDefaultBuilder**(args)

.**UseSerilog**() *// <- Add this line*

.**ConfigureWebHostDefaults**(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

**Important:** Serilog in ASP.NET Core 3 plugs into the generic host and not webBuilder.

That’s it! If you dotnet run you’ll now see clean, consistent, Serilog output.

### Cleaning up remnants of the default logger

There are a few spots in the application that traces of the default logger might remain. Serilog completely replaces the logging implementation on .NET Core: it’s not just a provider that works side-by-side with the built-in logging, but rather, an alternative implementation of the .NET Core logging APIs. The benefit of this is that you’re not running two different logging frameworks with tricky edge cases where they overlap in functionality.

The "Logging" section that you’ll find in appSettings.json isn’t used by Serilog, and can be removed:

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*"

}

After cleaning up here (and in appSettings.Development.json), the configuration looks like this:

{

"AllowedHosts": "\*"

}

## Writing log events

Here’s Hello, world! for 2019: the HomeController class generated by dotnet new mvc already includes an ILogger<T> field pre-configured, so we can call that in action methods such as Index(), and have events written through Serilog:

**public** **class** **HomeController** : Controller

{

**private** **readonly** ILogger<HomeController> \_logger;

**public** **HomeController**(ILogger<HomeController> logger)

{

\_logger = logger;

}

**public** IActionResult **Index**([FromQuery] **string** name)

{

\_logger.**LogInformation**("Hello, {Name}!", name);

**return** **View**();

}

## Recording structured logs

The logs now produced by the app leave a lot of room for improvement, which we’ll tackle in the next section. Before we go on, though, it’s useful to see the underlying key/value properties associated with each log event, since these play a big part in real-world diagnostics.

Structured logs can be captured as easily as specifying a formatter for console output:

*// Back in Program.Main():*

.WriteTo.**Console**(**new** **RenderedCompactJsonFormatter**())

Or by adding a file sink with similar configuration:

.WriteTo.**File**(**new** **RenderedCompactJsonFormatter**(), "/logs/log.ndjson")

# Microservices using .NET Core and Docker

## Prerequisites

This example assumes you already have an ASP.NET Core app on your machine. If you are new to ASP.NET you can follow a [simple tutorial](https://www.asp.net/get-started) to initialize a project or clone our [ASP.NET Docker Sample](https://github.com/dotnet/dotnet-docker/tree/master/samples/aspnetapp).

**Demo Projects**:

* ASP.NET Core 3.1:
  + D:\Users\AjayS\Tryouts\dotNET\_Core\dotnet-docker-samples\samples\aspnetapp
* ASP.NET Core 2.2:
  + D:\Users\AjayS\Tryouts\dotNET\_Core\WebAppForDocker

## Create a Dockerfile for an ASP.NET Core 2.2 application

1. Create a Dockerfile in your project folder.
2. Add the text below to your Dockerfile for either Linux or Windows Containers. The tags below are multi-arch meaning they pull either Windows or Linux containers depending on what mode is set in [Docker Desktop for Windows](https://docs.docker.com/docker-for-windows/). Read more on switching containers.
3. The Dockerfile assumes that your application is called aspnetapp. Change the Dockerfile to use the DLL file of your project.

FROM microsoft/dotnet:2.2-sdk as build

ARG BUILDCONFIG=RELEASE

ARG VERSION=1.0.0

COPY WebAppDocker2\_2/\*.csproj /build/WebAppDocker2\_2/

RUN dotnet restore ./build/WebAppDocker2\_2/WebAppDocker2\_2.csproj

COPY . ./build/

WORKDIR /build/WebAppDocker2\_2/

RUN dotnet publish ./WebAppDocker2\_2.csproj -c $BUILDCONFIG -o out /p:Version=$VERSION

FROM microsoft/dotnet:2.2-aspnetcore-runtime

WORKDIR /app

COPY --from=build /build/WebAppDocker2\_2/out .

ENTRYPOINT ["dotnet", "WebAppDocker2\_2.dll"]

To make your build context as small as possible add a [.dockerignore file](https://docs.docker.com/engine/reference/builder/#dockerignore-file) to your project folder and copy the following into it.

bin\

obj\

## Build and run the Docker image

1. Open a command prompt and navigate to your project folder.
2. Use the following commands to build and run your Docker image:

$ docker build -t aspnetapp .

$ docker run -d -p 8080:80 --name myapp aspnetapp

## View the web page running from a container

* Go to [localhost:8080](http://localhost:8080/) to access your app in a web browser.
* You can get the IP address of your container with the following steps:

1. Run  docker inspect -f "{{ .NetworkSettings.Networks.nat.IPAddress }}" myapp
2. Copy the container IP address and paste into your browser. (For example, 172.16.240.197)

# ASP.NET Core MVC with Visual Studio Code

## Prerequisites

* Visual Studio
* Visual Studio Code
* .NET Core 3.1 SDK or later

The Visual Studio Code instructions use the .NET Core CLI for ASP.NET Core development functions such as project creation. You can follow these instructions on any platform (macOS, Linux, or Windows) and with any code editor. Minor changes may be required if you use something other than Visual Studio Code.

## Create a web app

* Open the integrated terminal.
* Change directories (cd) to a folder which will contain the project.
* Run the following command:

dotnet new mvc -o MvcMovie

code -r MvcMovie

* A dialog box appears with **Required assets to build and debug are missing from 'MvcMovie'. Add them?** Select **Yes**
* dotnet new mvc -o MvcMovie: creates a new ASP.NET Core MVC project in the MvcMovie folder.
* code -r MvcMovie: Loads the MvcMovie.csproj project file in Visual Studio Code.

### Run the app

Press Ctrl+F5 to run without the debugger.

* Trust the HTTPS development certificate by running the following command:

dotnet dev-certs https --trust

The preceding command doesn't work on Linux. See your Linux distribution's documentation for trusting a certificate.

The preceding command displays the following dialog:



* Select **Yes** if you agree to trust the development certificate.

Visual Studio Code starts Kestrel, launches a browser, and navigates to https://localhost:5001. The address bar shows localhost:port:5001 and not something like example.com. That's because localhost is the standard hostname for local computer. Localhost only serves web requests from the local computer.

Launching the app with Ctrl+F5 (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to refresh the page and view changes.

## Add a Controller

Select the **EXPLORER** icon and then control-click (right-click) **Controllers > New File** and name the new file HelloWorldController.cs.

Replace the contents of Controllers/HelloWorldController.cs with the following:

using Microsoft.AspNetCore.Mvc;

using System.Text.Encodings.Web;

namespace MvcMovie.Controllers

{

public class HelloWorldController : Controller

{

//

// GET: /HelloWorld/

public string Index()

{

return "This is my default action...";

}

//

// GET: /HelloWorld/Welcome/

public string Welcome()

{

return "This is the Welcome action method...";

}

}

}

Run the app in non-debug mode and append "HelloWorld" to the path in the address bar. The Index method returns a string.

Browse to https://localhost:{PORT}/HelloWorld/Welcome. The Welcome method runs and returns the string This is the Welcome action method.... For this URL, the controller is HelloWorld and Welcome is the action method. You haven't used the [Parameters] part of the URL yet.

Modify the code to pass some parameter information from the URL to the controller. For example, /HelloWorld/Welcome?name=Rick&numtimes=4. Change the Welcome method to include two parameters as shown in the following code.

// GET: /HelloWorld/Welcome/

// Requires using System.Text.Encodings.Web;

public string Welcome(string name, int numTimes = 1)

{

return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");

}

The preceding code:

* Uses the C# optional-parameter feature to indicate that the numTimes parameter defaults to 1 if no value is passed for that parameter.
* Uses HtmlEncoder.Default.Encode to protect the app from malicious input (namely JavaScript).
* Uses Interpolated Strings in $"Hello {name}, NumTimes is: {numTimes}".

Run the app and browse to:

https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4

In the image above, the URL segment (Parameters) isn't used, the name and numTimes parameters are passed in the query string. The ? (question mark) in the above URL is a separator, and the query string follows. The & character separates field-value pairs.

Replace the Welcome method with the following code:

public string Welcome(string name, int ID = 1)

{

return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");

}

Run the app and enter the following URL: https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick

This time the third URL segment matched the route parameter id. The Welcome method contains a parameter id that matched the URL template in the MapControllerRoute method. The trailing ? (in id?) indicates the id parameter is optional.

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

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

});

## Add a View

Currently the Index method returns a string with a message that's hard-coded in the controller class. In the HelloWorldController class, replace the Index method with the following code:

public IActionResult Index()

{

return View();

}

Add an Index view for the HelloWorldController.

* Add a new folder named *Views/HelloWorld*.
* Add a new file to the *Views/HelloWorld* folder name *Index.cshtml*.

Replace the contents of the *Views/HelloWorld/Index.cshtml* Razor view file with the following:

@{

ViewData["Title"] = "Index";

}

<h2>Index</h2>

<p>Hello from our View Template!</p>

Navigate to https://localhost:{PORT}/HelloWorld.

### Change views and layout pages

Select the menu links (**MvcMovie**, **Home**, and **Privacy**). Each page shows the same menu layout. The menu layout is implemented in the Views/Shared/\_Layout.cshtml file. Open the Views/Shared/\_Layout.cshtml file.

Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the @RenderBody() line. RenderBody is a placeholder where all the view-specific pages you create show up, wrapped in the layout page. For example, if you select the **Privacy** link, the **Views/Home/Privacy.cshtml** view is rendered inside the RenderBody method.

Change the title, footer, and menu link in the layout file

Replace the content of the Views/Shared/\_Layout.cshtml file with the following markup. The changes are highlighted:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<title>@ViewData["Title"] - Movie App</title>

<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />

<link rel="stylesheet" href="~/css/site.css" />

</head>

<body>

<header>

<nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">

<div class="container">

<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>

<button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-collapse" aria-controls="navbarSupportedContent"

aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">

<ul class="navbar-nav flex-grow-1">

<li class="nav-item">

<a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>

</li>

<li class="nav-item">

<a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>

</li>

</ul>

</div>

</div>

</nav>

</header>

<div class="container">

<main role="main" class="pb-3">

@RenderBody()

</main>

</div>

<footer class="border-top footer text-muted">

<div class="container">

&copy; 2020 - Movie App - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>

</div>

</footer>

<script src="~/lib/jquery/dist/jquery.min.js"></script>

<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>

<script src="~/js/site.js" asp-append-version="true"></script>

@RenderSection("Scripts", required: false)

</body>

</html>

Save your changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy - Movie App** instead of **Privacy Policy - Mvc Movie**.

Select the **Home** link and notice that the title and anchor text also display **Movie App**. We were able to make the change once in the layout template and have all pages on the site reflect the new link text and new title.

Examine the Views/\_ViewStart.cshtml file:

@{

Layout = "\_Layout";

}

The Views/\_ViewStart.cshtml file brings in the Views/Shared/\_Layout.cshtml file to each view. The Layout property can be used to set a different layout view, or set it to null so no layout file will be used.

Change the title and <h2> element of the Views/HelloWorld/Index.cshtml view file:

@{

ViewData["Title"] = "Movie List";

}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>

The title and <h2> element are slightly different so you can see which bit of code changes the display.

ViewData["Title"] = "Movie List"; in the code above sets the Title property of the ViewData dictionary to "Movie List". The Title property is used in the <title> HTML element in the layout page:

<title>@ViewData["Title"] - Movie App</title>

Save the change and navigate to https://localhost:{PORT}/HelloWorld. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.) The browser title is created with ViewData["Title"] we set in the Index.cshtml view template and the additional "- Movie App" added in the layout file.

The content in the Index.cshtml view template is merged with the Views/Shared/\_Layout.cshtml view template. A single HTML response is sent to the browser. Layout templates make it easy to make changes that apply across all of the pages in an app.

### Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response. A best practice: View templates should **not** perform business logic or interact with a database directly. Rather, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code clean, testable, and maintainable.

Currently, the Welcome method in the HelloWorldController class takes a name and a ID parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate bits of data must be passed from the controller to the view in order to generate the response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a ViewData dictionary that the view template can then access.

In HelloWorldController.cs, change the Welcome method to add a Message and NumTimes value to the ViewData dictionary. The ViewData dictionary is a dynamic object, which means any type can be used; the ViewData object has no defined properties until you put something inside it. The MVC model binding system automatically maps the named parameters (name and numTimes) from the query string in the address bar to parameters in your method. The complete HelloWorldController.cs file looks like this:

using Microsoft.AspNetCore.Mvc;

using System.Text.Encodings.Web;

namespace MvcMovie.Controllers

{

public class HelloWorldController : Controller

{

public IActionResult Index()

{

return View();

}

public IActionResult Welcome(string name, int numTimes = 1)

{

ViewData["Message"] = "Hello " + name;

ViewData["NumTimes"] = numTimes;

return View();

}

}

}

The ViewData dictionary object contains data that will be passed to the view.

Create a Welcome view template named Views/HelloWorld/Welcome.cshtml.

You'll create a loop in the Welcome.cshtml view template that displays "Hello" NumTimes. Replace the contents of Views/HelloWorld/Welcome.cshtml with the following:

@{

ViewData["Title"] = "Welcome";

}

<h2>Welcome</h2>

<ul>

@for (int i = 0; i < (int)ViewData["NumTimes"]; i++)

{

<li>@ViewData["Message"]</li>

}

</ul>

Save your changes and browse to the following URL:

https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4

Data is taken from the URL and passed to the controller using the MVC model binder . The controller packages the data into a ViewData dictionary and passes that object to the view. The view then renders the data as HTML to the browser.

## Add a Model

Add a file named Movie.cs to the Models folder.

Update the *Movie.cs* file with the following code:

using System;

using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models

{

public class Movie

{

public int Id { get; set; }

public string Title { get; set; }

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

public string Genre { get; set; }

public decimal Price { get; set; }

}

}

### Add NuGet Packages

Run the following .NET Core CLI commands:

dotnet tool install --global dotnet-ef

dotnet tool install --global dotnet-aspnet-codegenerator

dotnet add package Microsoft.EntityFrameworkCore.SQLite

dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design

dotnet add package Microsoft.EntityFrameworkCore.Design

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

The preceding commands add:

* The aspnet-codegenerator scaffolding tool.
* The Entity Framework Core Tools for the .NET Core CLI.
* The EF Core SQLite provider, which installs the EF Core package as a dependency.
* Packages needed for scaffolding: Microsoft.VisualStudio.Web.CodeGeneration.Design and Microsoft.EntityFrameworkCore.SqlServer.

### Create a database context class

A database context class is needed to coordinate EF Core functionality (Create, Read, Update, Delete) for the Movie model. The database context is derived from Microsoft.EntityFrameworkCore.DbContext and specifies the entities to include in the data model.

Create a Data folder.

Add a Data/MvcMovieContext.cs file with the following code:

using Microsoft.EntityFrameworkCore;

using MvcMovie.Models;

namespace MvcMovie.Data

{

public class MvcMovieContext : DbContext

{

public MvcMovieContext (DbContextOptions<MvcMovieContext> options)

: base(options)

{

}

public DbSet<Movie> Movie { get; set; }

}

}

### Register the database context

ASP.NET Core is built with dependency injection (DI). Services (such as the EF Core DB context) must be registered with DI during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial. In this section, you register the database context with the DI container.

Add the following using statements at the top of Startup.cs:

using MvcMovie.Data;

using Microsoft.EntityFrameworkCore;

Add the following highlighted code in Startup.ConfigureServices:

public void ConfigureServices(IServiceCollection services)

{

services.AddControllersWithViews();

services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlite(Configuration.GetConnectionString("MvcMovieContext")));

}

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the appsettings.json file.

### Add a database connection string

Add a connection string to the appsettings.json file:

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": { "MvcMovieContext": "Data Source=MvcMovie.db" }

}

Build the project as a check for compiler errors.

### Scaffold movie pages

Use the scaffolding tool to produce Create, Read, Update, and Delete (CRUD) pages for the movie model.

* Open a command window in the project directory (The directory that contains the Program.cs, Startup.cs, and .csproj files).
* On Linux, export the scaffold tool path:

export PATH=$HOME/.dotnet/tools:$PATH

* Run the following command:

dotnet aspnet-codegenerator controller -name MoviesController -m Movie -dc MvcMovieContext --relativeFolderPath Controllers --useDefaultLayout --referenceScriptLibraries

The following table details the ASP.NET Core code generator parameters:

| **Parameter** | **Description** |
| --- | --- |
| -m | The name of the model. |
| -dc | The data context. |
| -udl | Use the default layout. |
| --relativeFolderPath | The relative output folder path to create the files. |
| --useDefaultLayout | The default layout should be used for the views. |
| --referenceScriptLibraries | Adds \_ValidationScriptsPartial to Edit and Create pages |

Use the h switch to get help on the aspnet-codegenerator controller command:

dotnet aspnet-codegenerator controller -h

You can't use the scaffolded pages yet because the database doesn't exist. If you run the app and click on the **Movie App** link, you get a Cannot open database or no such table: Movie error message.

### Initial migration

Use the EF Core Migrations feature to create the database. Migrations is a set of tools that let you create and update a database to match your data model.

Run the following .NET Core CLI commands:

dotnet ef migrations add InitialCreate

dotnet ef database update

* ef migrations add InitialCreate: Generates an Migrations/{timestamp}\_InitialCreate.cs migration file. The InitialCreate argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration. Because this is the first migration, the generated class contains code to create the database schema. The database schema is based on the model specified in the MvcMovieContext class (in the Data/MvcMovieContext.cs file).
* ef database update: Updates the database to the latest migration, which the previous command created. This command runs the Up method in the Migrations/{time-stamp}\_InitialCreate.cs file, which creates the database.

### The InitialCreate class

Examine the Migrations/{timestamp}\_InitialCreate.cs migration file:

The Up method creates the Movie table and configures Id as the primary key. The Down method reverts the schema changes made by the Up migration.

### Test the App

* Run the app and click the **Movie App** link.
* Test the **Create** page. Enter and submit data.
* Test the **Edit**, **Details**, and **Delete** pages.

## Dependency injection in the controller

public class MoviesController : Controller

{

private readonly MvcMovieContext \_context;

public MoviesController(MvcMovieContext context)

{

\_context = context;

}

The constructor uses Dependency Injection to inject the database context (MvcMovieContext) into the controller. The database context is used in each of the CRUD methods in the controller.

### Use SQLite for development, SQL Server for production

When SQLite is selected, the template generated code is ready for development. The following code shows how to inject IWebHostEnvironment into Startup. IWebHostEnvironment is injected so ConfigureServices can use SQLite in development and SQL Server in production.

public class Startup

{

public Startup(IConfiguration configuration, IWebHostEnvironment env)

{

Environment = env;

Configuration = configuration;

}

public IConfiguration Configuration { get; }

public IWebHostEnvironment Environment { get; }

public void ConfigureServices(IServiceCollection services)

{

services.AddControllersWithViews();

services.AddDbContext<MvcMovieContext>(options =>

{

var connectionString = Configuration.GetConnectionString("MvcMovieContext");

if (Environment.IsDevelopment())

{

options.UseSqlite(connectionString);

}

else

{

options.UseSqlServer(connectionString);

}

});

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Home/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

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

});

}

}

Examine the contents of the Views/Movies/Details.cshtml file:

## Work with SQL in ASP.NET Core

The MvcMovieContext object handles the task of connecting to the database and mapping Movie objects to database records. The database context is registered with the Dependency Injection container in the ConfigureServices method in the Startup.cs file:

public void ConfigureServices(IServiceCollection services)

{

services.AddControllersWithViews();

services.AddDbContext<MvcMovieContext>(options => options.UseSqlite(Configuration.GetConnectionString("MvcMovieContext")));

}

The ASP.NET Core Configuration system reads the ConnectionString. For local development, it gets the connection string from the appsettings.json file:

"ConnectionStrings": {

"MovieContext": "Data Source=MvcMovie.db"

}

When the app is deployed to a test or production server, an environment variable can be used to set the connection string to a production SQL Server.

### Seed the database

Create a new class named SeedData in the Models folder. Replace the generated code with the following:

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.DependencyInjection;

using MvcMovie.Data;

using System;

using System.Linq;

namespace MvcMovie.Models

{

public static class SeedData

{

public static void Initialize(IServiceProvider serviceProvider)

{

using (var context = new MvcMovieContext(

serviceProvider.GetRequiredService<

DbContextOptions<MvcMovieContext>>()))

{

// Look for any movies.

if (context.Movie.Any())

{

return; // DB has been seeded

}

context.Movie.AddRange(

new Movie

{

Title = "When Harry Met Sally",

ReleaseDate = DateTime.Parse("1989-2-12"),

Genre = "Romantic Comedy",

Price = 7.99M

},

new Movie

{

Title = "Ghostbusters ",

ReleaseDate = DateTime.Parse("1984-3-13"),

Genre = "Comedy",

Price = 8.99M

},

new Movie

{

Title = "Ghostbusters 2",

ReleaseDate = DateTime.Parse("1986-2-23"),

Genre = "Comedy",

Price = 9.99M

},

new Movie

{

Title = "Rio Bravo",

ReleaseDate = DateTime.Parse("1959-4-15"),

Genre = "Western",

Price = 3.99M

}

);

context.SaveChanges();

}

}

}

}

If there are any movies in the DB, the seed initializer returns and no movies are added.

if (context.Movie.Any())

{

return; // DB has been seeded.

}

### Add the seed initializer

Replace the contents of Program.cs with the following code:

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using Microsoft.Extensions.Logging;

using MvcMovie.Data;

using MvcMovie.Models;

using System;

namespace MvcMovie

{

public class Program

{

public static void Main(string[] args)

{

var host = CreateHostBuilder(args).Build();

using (var scope = host.Services.CreateScope())

{

var services = scope.ServiceProvider;

try

{

SeedData.Initialize(services);

}

catch (Exception ex)

{

var logger = services.GetRequiredService<ILogger<Program>>();

logger.LogError(ex, "An error occurred seeding the DB.");

}

}

host.Run();

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

}

}

Test the app

*Delete all the records in the DB (So the seed method will run). Stop and start the app to seed the database.*

## Add search to an ASP.NET Core MVC app

Update the Index method found inside *Controllers/MoviesController.cs* with the following code:

public async Task<IActionResult> Index(string searchString)

{

var movies = from m in \_context.Movie

select m;

if (!String.IsNullOrEmpty(searchString))

{

movies = movies.Where(s => s.Title.Contains(searchString));

}

return View(await movies.ToListAsync());

}

The query is *only* defined at this point, it has **not** been run against the database.

If the searchString parameter contains a string, the movies query is modified to filter on the value of the search string:

if (!String.IsNullOrEmpty(searchString))

{

movies = movies.Where(s => s.Title.Contains(searchString));

}

**Note**: *The Contains method is run on the database, not in the c# code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, Contains maps to SQL LIKE, which is case insensitive. In SQLite, with the default collation, it's case sensitive.*

Navigate to /Movies/Index. Append a query string such as ?searchString=Ghost to the URL. The filtered movies are displayed.

If you change the signature of the Index method to have a parameter named id, the id parameter will match the optional {id} placeholder for the default routes set in *Startup.cs*.

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

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

});

Change the parameter to id and all occurrences of searchString change to id.

The previous Index method:

public async Task<IActionResult> Index(string searchString)

{

var movies = from m in \_context.Movie

select m;

if (!String.IsNullOrEmpty(searchString))

{

movies = movies.Where(s => s.Title.Contains(searchString));

}

return View(await movies.ToListAsync());

}

The updated Index method with id parameter:

public async Task<IActionResult> Index(string id)

{

var movies = from m in \_context.Movie

select m;

if (!String.IsNullOrEmpty(id))

{

movies = movies.Where(s => s.Title.Contains(id));

}

return View(await movies.ToListAsync());

}

You can now pass the search title as route data (a URL segment) instead of as a query string value as *http://localhost:<PORT>/Movies/Index/ghost*.

However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI elements to help them filter movies. If you changed the signature of the Index method to test how to pass the route-bound ID parameter, change it back so that it takes a parameter named searchString.

Open the *Views/Movies/Index.cshtml* file, and add the <form> markup highlighted below:

ViewData["Title"] = "Index";

}

<h2>Index</h2>

<p>

<a asp-action="Create">Create New</a>

</p>

<form asp-controller="Movies" asp-action="Index">

<p>

Title: <input type="text" name="SearchString">

<input type="submit" value="Filter" />

</p>

</form>

<table class="table">

<thead>

The HTML <form> tag uses the Form Tag Helper, so when you submit the form, the filter string is posted to the Index action of the movies controller. Save your changes and then test the filter.

There's no [HttpPost] overload of the Index method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data.

You could add the following [HttpPost] Index method.

[HttpPost]

public string Index(string searchString, bool notUsed)

{

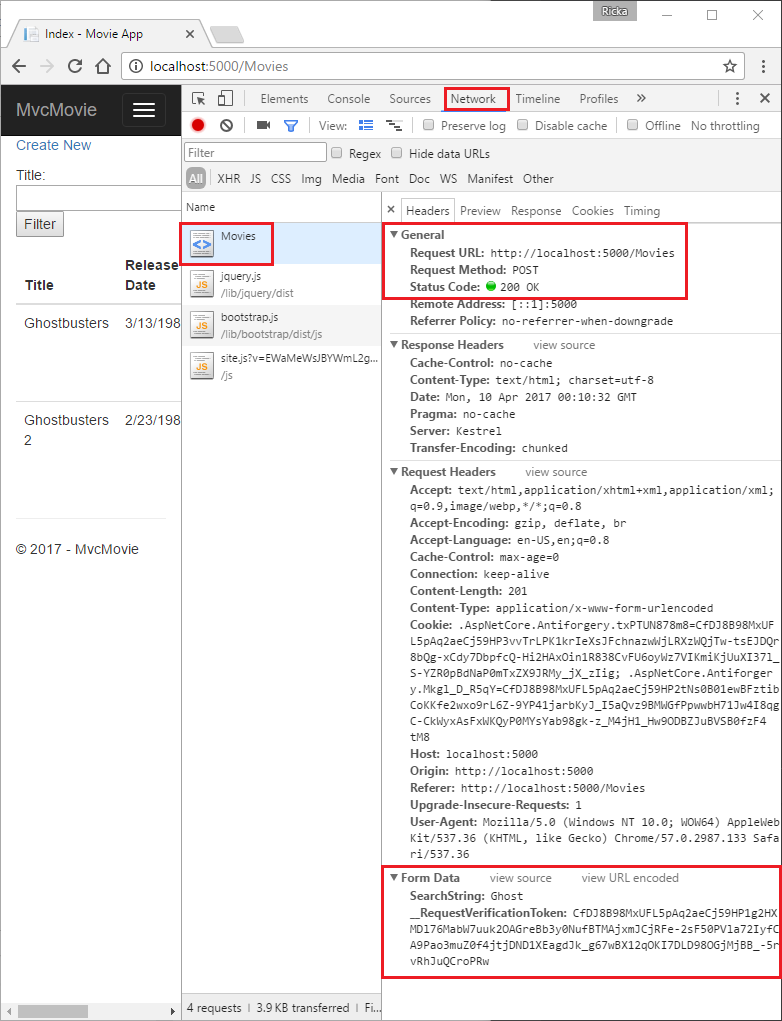
return "From [HttpPost]Index: filter on " + searchString;

}

The notUsed parameter is used to create an overload for the Index method. We'll talk about that later in the tutorial.

If you add this method, the action invoker would match the [HttpPost] Index method, and the [HttpPost] Index method would run.

However, even if you add this [HttpPost] version of the Index method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:{PORT}/Movies/Index) -- there's no search information in the URL. The search string information is sent to the server as a form field value. You can verify that with the browser Developer tools or the excellent Fiddler tool. The image below shows the Chrome browser Developer tools:



You can see the search parameter and XSRF token in the request body. Note, as mentioned in the previous tutorial, the Form Tag Helper generates an XSRF anti-forgery token. We're not modifying data, so we don't need to validate the token in the controller method.

Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. Fix this by specifying the request should be HTTP GET found in the *Views/Movies/Index.cshtml* file.

@model IEnumerable<MvcMovie.Models.Movie>

@{

ViewData["Title"] = "Index";

}

<h1>Index</h1>

<p>

<a asp-action="Create">Create New</a>

</p>

<form asp-controller="Movies" asp-action="Index" method="get">

<p>

Title: <input type="text" name="SearchString">

<input type="submit" value="Filter" />

</p>

</form>

<table class="table">

<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.Title)

Now when you submit a search, the URL contains the search query string. Searching will also go to the HttpGet Index action method, even if you have a HttpPost Index method.

### Add Search by genre

Add the following MovieGenreViewModel class to the Models folder:

using Microsoft.AspNetCore.Mvc.Rendering;

using System.Collections.Generic;

namespace MvcMovie.Models

{

public class MovieGenreViewModel

{

public List<Movie> Movies { get; set; }

public SelectList Genres { get; set; }

public string MovieGenre { get; set; }

public string SearchString { get; set; }

}

}

Replace the Index method in MoviesController.cs with the following code:

// GET: Movies

public async Task<IActionResult> Index(string movieGenre, string searchString)

{

// Use LINQ to get list of genres.

IQueryable<string> genreQuery = from m in \_context.Movie

orderby m.Genre

select m.Genre;

var movies = from m in \_context.Movie

select m;

if (!string.IsNullOrEmpty(searchString))

{

movies = movies.Where(s => s.Title.Contains(searchString));

}

if (!string.IsNullOrEmpty(movieGenre))

{

movies = movies.Where(x => x.Genre == movieGenre);

}

var movieGenreVM = new MovieGenreViewModel

{

Genres = new SelectList(await genreQuery.Distinct().ToListAsync()),

Movies = await movies.ToListAsync()

};

return View(movieGenreVM);

}

When the user searches for the item, the search value is retained in the search box.

### Add search by genre to the Index view

Update Index.cshtml found in Views/Movies/ as follows:

@model MvcMovie.Models.MovieGenreViewModel

@{

ViewData["Title"] = "Index";

}

<h1>Index</h1>

<p>

<a asp-action="Create">Create New</a>

</p>

<form asp-controller="Movies" asp-action="Index" method="get">

<p>

<select asp-for="MovieGenre" asp-items="Model.Genres">

<option value="">All</option>

</select>

Title: <input type="text" asp-for="SearchString" />

<input type="submit" value="Filter" />

</p>

</form>

<table class="table">

<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.Movies[0].Title)

</th>

<th>

@Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)

</th>

<th>

@Html.DisplayNameFor(model => model.Movies[0].Genre)

</th>

<th>

@Html.DisplayNameFor(model => model.Movies[0].Price)

</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model.Movies)

{

<tr>

<td>

@Html.DisplayFor(modelItem => item.Title)

</td>

<td>

@Html.DisplayFor(modelItem => item.ReleaseDate)

</td>

<td>

@Html.DisplayFor(modelItem => item.Genre)

</td>

<td>

@Html.DisplayFor(modelItem => item.Price)

</td>

<td>

<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |

<a asp-action="Details" asp-route-id="@item.Id">Details</a> |

<a asp-action="Delete" asp-route-id="@item.Id">Delete</a>

</td>

</tr>

}

</tbody>

</table>

Test the app by searching by genre, by movie title, and by both.

## Add a new field

When EF Code First is used to automatically create a database, Code First:

* Adds a table to the database to track the schema of the database.
* Verifies the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

### Add a Rating Property to the Movie Model

Add a Rating property to Models/Movie.cs:

public class Movie

{

public int Id { get; set; }

public string Title { get; set; }

[Display(Name = "Release Date")]

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

public string Genre { get; set; }

[Column(TypeName = "decimal(18, 2)")]

public decimal Price { get; set; }

public string Rating { get; set; }

}

Build the app.

dotnet build

Because you've added a new field to the Movie class, you need to update the binding white list so this new property will be included. In *MoviesController.cs*, update the [Bind] attribute for both the Create and Edit action methods to include the Rating property:

[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")]

Update the view templates in order to display, create, and edit the new Rating property in the browser view.

Edit the */Views/Movies/Index.cshtml* file and add a Rating field:

<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.Movies[0].Title)

</th>

<th>

@Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)

</th>

<th>

@Html.DisplayNameFor(model => model.Movies[0].Genre)

</th>

<th>

@Html.DisplayNameFor(model => model.Movies[0].Price)

</th>

<th>

@Html.DisplayNameFor(model => model.Movies[0].Rating)

</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model.Movies)

{

<tr>

<td>

@Html.DisplayFor(modelItem => item.Title)

</td>

<td>

@Html.DisplayFor(modelItem => item.ReleaseDate)

</td>

<td>

@Html.DisplayFor(modelItem => item.Genre)

</td>

<td>

@Html.DisplayFor(modelItem => item.Price)

</td>

<td>

@Html.DisplayFor(modelItem => item.Rating)

</td>

<td>

<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |

Update the /Views/Movies/Create.cshtml with a Rating field.

Update the remaining templates.

Update the SeedData class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each new Movie.

new Movie

{

Title = "When Harry Met Sally",

ReleaseDate = DateTime.Parse("1989-1-11"),

Genre = "Romantic Comedy",

Rating = "R",

Price = 7.99M

},

The app won't work until the DB is updated to include the new field. If it's run now, the following SqlException is thrown:

SqlException: Invalid column name 'Rating'.

This error occurs because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no Rating column in the database table.)

### Use Code First Migrations to update the database schema

**Note***: For this tutorial you use the Entity Framework Core migrations feature where possible. Migrations updates the database schema to match changes in the data model. However, migrations can only do the kinds of changes that the EF Core provider supports, and the SQLite provider's capabilities are limited. For example, adding a column is supported, but removing or changing a column is not supported. If a migration is created to remove or change a column, the ef migrations add command succeeds but the ef database update command fails. Due to these limitations, this tutorial doesn't use migrations for SQLite schema changes. Instead, when the schema changes, you drop and re-create the database.*

*The workaround for the SQLite limitations is to manually write migrations code to perform a table rebuild when something in the table changes. A table rebuild involves:*

* *Creating a new table.*
* *Copying data from the old table to the new table.*
* *Dropping the old table.*
* *Renaming the new table.*

Delete the database and use migrations to re-create the database. To delete the database, delete the database file (*MvcMovie.db*). Then run the ef database update command:

dotnet ef database update

Run the app and verify you can create/edit/display movies with a Rating field. You should add the Rating field to the Edit, Details, and Delete view templates.

## Add validation

### Keeping things DRY

One of the design tenets of MVC is DRY ("Don't Repeat Yourself"). ASP.NET Core MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an app. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by MVC and Entity Framework Core Code First is a good example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

### Add validation rules to the movie model

The DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like DataType that help with formatting and don't provide any validation.

Update the Movie class to take advantage of the built-in Required, StringLength, RegularExpression, and Range validation attributes.

public class Movie

{

public int Id { get; set; }

[StringLength(60, MinimumLength = 3)]

[Required]

public string Title { get; set; }

[Display(Name = "Release Date")]

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

[Range(1, 100)]

[DataType(DataType.Currency)]

[Column(TypeName = "decimal(18, 2)")]

public decimal Price { get; set; }

[RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]\*$")]

[Required]

[StringLength(30)]

public string Genre { get; set; }

[RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]\*$")]

[StringLength(5)]

[Required]

public string Rating { get; set; }

}

### Validation Error UI

Run the app and navigate to the Movies controller.

Tap the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.

The DisplayFormat attribute is used to explicitly specify the date format:

[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]

public DateTime ReleaseDate { get; set; }

The ApplyFormatInEditMode setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably don't want the currency symbol in the text box for editing.)

You can use the DisplayFormat attribute by itself, but it's generally a good idea to use the DataType attribute. The DataType attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

* The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
* By default, the browser will render data using the correct format based on your locale.
* The DataType attribute can enable MVC to choose the right field template to render the data (the DisplayFormat if used by itself uses the string template).

**Note**

*jQuery validation doesn't work with the Range attribute and DateTime. For example, the following code will always display a client side validation error, even when the date is in the specified range:*

*[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]*

You will need to disable jQuery date validation to use the Range attribute with DateTime. It's generally not a good practice to compile hard dates in your models, so using the Range attribute and DateTime is discouraged.