ASP.NET Core fundamentals

03/30/2020 • 17 minutes to read • +10

In this article

This article provides an overview of key topics for understanding how to develop ASP.NET Core apps.

The Startup class

The `Startup` class is where:

- Services required by the app are configured.
- The app's request handling pipeline is defined, as a series of middleware components.

Here's a sample `Startup` class:

C# Copy

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<RazorPagesMovieContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));

        services.AddControllersWithViews();
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapDefaultControllerRoute();
            endpoints.MapRazorPages();
        });
    }
}
```

For more information, see App startup in ASP.NET Core.

Dependency injection (services)

ASP.NET Core includes a built-in dependency injection (DI) framework that makes configured services available throughout an app. For example, a logging component is a service.

Code to configure (or *register*) services is added to the `Startup.ConfigureServices` method. For example:

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<RazorPagesMovieContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));

    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

Services are typically resolved from DI using constructor injection. With constructor injection, a class declares a constructor parameter of either the required type or an interface. The DI framework provides an instance of this service at runtime.

The following example uses constructor injection to resolve a `RazorPagesMovieContext` from DI:

C# Copy

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovieContext context)
    {
        _context = context;
    }

    // ...

    public async Task OnGetAsync()
    {
        Movies = await _context.Movies.ToListAsync();
    }
}
```

If the built-in Inversion of Control (IoC) container doesn't meet all of an app's needs, a third-party IoC container can be used instead.

For more information, see Dependency injection in ASP.NET Core.

Middleware

The request handling pipeline is composed as a series of middleware components. Each component performs operations on an `HttpContext` and either invokes the next middleware in the pipeline or terminates the request.

By convention, a middleware component is added to the pipeline by invoking a `Use...` extension method in the `Startup.Configure` method. For example, to enable rendering of static files, call `UseStaticFiles`.

The following example configures a request handling pipeline:

C# Copy

```csharp
public void Configure(IApplicationBuilder app)
{
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
    });
}
```

ASP.NET Core includes a rich set of built-in middleware. Custom middleware components can also be written.

For more information, see ASP.NET Core Middleware.

Host

On startup, an ASP.NET Core app builds a *host*. The host encapsulates all of the app's resources, such as:

- An HTTP server implementation
- Middleware components
- Logging
- Dependency injection (DI) services
- Configuration

There are two different hosts:

- .NET Generic Host
- ASP.NET Core Web Host

The .NET Generic Host is recommended. The ASP.NET Core Web Host is available only for backwards compatibility.

The following example creates a .NET Generic Host:

C# Copy

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The `CreateDefaultBuilder` and `ConfigureWebHostDefaults` methods configure a host with a set of default options, such as:

- Use Kestrel as the web server and enable IIS integration.
- Load configuration from *appsettings.json*, *appsettings.{Environment Name}.json*, environment variables, command line arguments, and other configuration sources.
- Send logging output to the console and debug providers.

For more information, see .NET Generic Host.

Non-web scenarios

The Generic Host allows other types of apps to use cross-cutting framework extensions, such as logging, dependency injection (DI), configuration, and app lifetime management. For more information, see .NET Generic Host and Background tasks with hosted services in ASP.NET Core.

Servers

An ASP.NET Core app uses an HTTP server implementation to listen for HTTP requests. The server surfaces requests to the app as a set of request features composed into an `HttpContext`.

Windows macOS Linux

ASP.NET Core provides the following server implementations:

- *Kestrel* is a cross-platform web server. Kestrel is often run in a reverse proxy configuration using IIS. In ASP.NET Core 2.0 or later, Kestrel can be run as a public-facing edge server exposed directly to the Internet.
- *IIS HTTP Server* is a server for Windows that uses IIS. With this server, the ASP.NET Core app and IIS run in the same process.
- *HTTP.sys* is a server for Windows that isn't used with IIS.

For more information, see Web server implementations in ASP.NET Core.

Configuration

ASP.NET Core provides a configuration framework that gets settings as name-value pairs from an ordered set of configuration providers. Built-in configuration providers are available for a variety of sources, such as *.json* files, *.xml* files, environment variables, and command-line arguments. Write custom configuration providers to support other sources.

By default, ASP.NET Core apps are configured to read from *appsettings.json*, environment variables, the command line, and more. When the app's configuration is loaded, values from environment variables override values from *appsettings.json*.

The preferred way to read related configuration values is using the options pattern. For more information, see Bind hierarchical configuration data using the options pattern.

For managing confidential configuration data such as passwords, ASP.NET Core provides the Secret Manager. For production secrets, we recommend Azure Key Vault.

For more information, see Configuration in ASP.NET Core.

Environments

Execution environments, such as `Development`, `Staging`, and `Production`, are a first-class notion in ASP.NET Core. Specify the environment an app is running in by setting the `ASPNETCORE_ENVIRONMENT` environment variable. ASP.NET Core reads that environment variable at app startup and stores the value in an `IWebHostEnvironment` implementation. This implementation is available anywhere in an app via dependency injection (DI).

The following example configures the app to provide detailed error information when running in the `Development` environment:

C#

```csharp
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

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

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
    });
}
```

For more information, see Use multiple environments in ASP.NET Core.

Logging

ASP.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. Available providers include:

- Console
- Debug
- Event Tracing on Windows
- Windows Event Log
- TraceSource
- Azure App Service
- Azure Application Insights

To create logs, resolve an ILogger<TCategoryName> service from dependency injection (DI) and call logging methods such as LogInformation. For example:

C# Copy

```csharp
public class TodoController : ControllerBase
{
    private readonly ILogger _logger;

    public TodoController(ILogger<TodoController> logger)
    {
        _logger = logger;
    }

    [HttpGet("{id}", Name = "GetTodo")]
    public ActionResult<TodoItem> GetById(string id)
    {
        _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);

        // Item lookup code removed.
        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
            return NotFound();
        }

        return item;
    }
}
```

Logging methods such as `LogInformation` support any number of fields. These fields are commonly used to construct a message `string`, but some logging providers send these to a data store as separate fields. This feature makes it possible for logging providers to implement semantic logging, also known as structured logging.

For more information, see Logging in .NET Core and ASP.NET Core.

Routing

A *route* is a URL pattern that is mapped to a handler. The handler is typically a Razor page, an action method in an MVC controller, or a middleware. ASP.NET Core routing gives you control over the URLs used by your app.

For more information, see Routing in ASP.NET Core.

Error handling

ASP.NET Core has built-in features for handling errors, such as:

- A developer exception page
- Custom error pages
- Static status code pages
- Startup exception handling

For more information, see Handle errors in ASP.NET Core.

Make HTTP requests

An implementation of `IHttpClientFactory` is available for creating `HttpClient` instances. The factory:

- Provides a central location for naming and configuring logical `HttpClient` instances. For example, register and configure a *github* client for accessing GitHub. Register and configure a default client for other purposes.
- Supports registration and chaining of multiple delegating handlers to build an outgoing request middleware pipeline. This pattern is similar to ASP.NET Core's inbound middleware pipeline. The pattern provides a mechanism to manage cross-cutting concerns for HTTP requests, including caching, error handling, serialization, and logging.
- Integrates with *Polly*, a popular third-party library for transient fault handling.
- Manages the pooling and lifetime of underlying `HttpClientHandler` instances to avoid common DNS problems that occur when managing `HttpClient` lifetimes manually.
- Adds a configurable logging experience via ILogger for all requests sent through clients created by the factory.

For more information, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

Content root

The content root is the base path for:

- The executable hosting the app (*.exe*).
- Compiled assemblies that make up the app (*.dll*).
- Content files used by the app, such as:
  - Razor files (*.cshtml*, *.razor*)
  - Configuration files (*.json*, *.xml*)
  - Data files (*.db*)
- The Web root, typically the *wwwroot* folder.

During development, the content root defaults to the project's root directory. This directory is also the base path for both the app's content files and the Web root. Specify a different content root by setting its path when building the host. For more information, see Content root.

Web root

The web root is the base path for public, static resource files, such as:

- Stylesheets (*.css*)
- JavaScript (*.js*)
- Images (*.png*, *.jpg*)

By default, static files are served only from the web root directory and its sub-directories. The web root path defaults to *{content root}/wwwroot*. Specify a different web root by setting its path when building the host. For more information, see Web root.

Prevent publishing files in *wwwroot* with the <Content> project item in the project file. The following example prevents publishing content in *wwwroot/local* and its sub-directories:

XML Copy

```
<ItemGroup>
  <Content Update="wwwroot\local\**\*.*" CopyToPublishDirectory="Never" />
</ItemGroup>
```

In Razor *.cshtml* files, tilde-slash (`~/`) points to the web root. A path beginning with `~/` is referred to as a *virtual path*.

For more information, see Static files in ASP.NET Core.

Is this page helpful?

👍 Yes 👎 No