ASP.NET Core is a cross-platform, high-performance, [open-source](https://github.com/dotnet/aspnetcore) framework for building modern, cloud-enabled, Internet-connected apps.

With ASP.NET Core, you can:

* Build web apps and services, [Internet of Things (IoT)](https://www.microsoft.com/internet-of-things/) apps, and mobile backends.
* Use your favorite development tools on Windows, macOS, and Linux.
* Deploy to the cloud or on-premises.
* Run on [.NET](https://learn.microsoft.com/en-us/dotnet/core/introduction).

ASP.NET Core supports two approaches to creating APIs: a controller-based approach and minimal APIs. *Controllers* in an API project are classes that derive from [ControllerBase](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase). *Minimal APIs* define endpoints with logical handlers in lambdas or methods. This article points out differences between the two approaches.

The design of minimal APIs hides the host class by default and focuses on configuration and extensibility via extension methods that take functions as lambda expressions. Controllers are classes that can take dependencies via constructor injection or property injection, and generally follow object-oriented patterns. Minimal APIs support dependency injection through other approaches such as accessing the service provider.

Minimal APIs have many of the same capabilities as controller-based APIs. They support the configuration and customization needed to scale to multiple APIs, handle complex routes, apply authorization rules, and control the content of API responses. There are a few capabilities available with controller-based APIs that are not yet supported or implemented by minimal APIs. These include:

* No built-in support for model binding ([IModelBinderProvider](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.modelbinding.imodelbinderprovider), [IModelBinder](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.modelbinding.imodelbinder)). Support can be added with a custom binding shim.
* No built-in support for validation ([IModelValidator](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.modelbinding.validation.imodelvalidator)).
* No support for [application parts](https://learn.microsoft.com/en-us/aspnet/core/mvc/advanced/app-parts?view=aspnetcore-9.0) or the [application model](https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/application-model?view=aspnetcore-9.0). There's no way to apply or build your own conventions.
* No built-in view rendering support. We recommend using [Razor Pages](https://learn.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start?view=aspnetcore-9.0) for rendering views.
* No support for [JsonPatch](https://www.nuget.org/packages/Microsoft.AspNetCore.JsonPatch/)
* No support for [OData](https://www.nuget.org/packages/Microsoft.AspNetCore.OData/)

**Program.cs**

ASP.NET Core apps created with the web templates contain the application startup code in the Program.cs file. The Program.cs file is where:

* Services required by the app are configured.
* The app's request handling pipeline is defined as a series of [middleware components](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-9.0).
* C#Copy
* using WebAll.Components;
* var builder = WebApplication.CreateBuilder(args);
* // Add services to the container.
* builder.Services.AddRazorComponents()
* .AddInteractiveServerComponents();
* builder.Services.AddRazorPages();
* builder.Services.AddControllersWithViews();
* var app = builder.Build();
* // Configure the HTTP request pipeline by adding middleware in order
* if (!app.Environment.IsDevelopment())
* {
* app.UseExceptionHandler("/Error");
* app.UseHsts();
* }
* app.UseHttpsRedirection();
* app.UseStaticFiles();
* app.UseAuthorization();
* app.MapGet("/hi", () => "Hello!");
* app.MapDefaultControllerRoute();
* app.MapRazorPages();
* app.MapRazorComponents<App>()
* .AddInteractiveServerRenderMode();
* app.UseAntiforgery();
* app.Run();

CORS: Browser security prevents a web page from making requests to a different domain than the one that served the web page. This restriction is called the *same-origin policy*. The same-origin policy prevents a malicious site from reading sensitive data from another site.

var MyAllowSpecificOrigins = "\_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options => { options.AddPolicy(name: MyAllowSpecificOrigins, policy => { policy.WithOrigins("http://example.com", "http://www.contoso.com"); }); });

// services.AddResponseCaching();

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseCors(MyAllowSpecificOrigins);

app.UseAuthorization();

app.MapControllers();

app.Run();

GDPR:

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<CookiePolicyOptions>(options => { // This lambda determines whether user consent for non-essential // cookies is needed for a given request. options.CheckConsentNeeded = context => true; options.MinimumSameSitePolicy = SameSiteMode.None; });

var app = builder.Build();

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseCookiePolicy();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();

Cross-site request forgery is an attack against web-hosted apps whereby a malicious web app can influence the interaction between a client browser and a web app that trusts that browser. These attacks are possible because web browsers send some types of authentication tokens automatically with every request to a website. This form of exploit is also known as a *one-click attack* or *session riding* because the attack takes advantage of the user's previously authenticated session. Cross-site request forgery is also known as XSRF or CSRF.

CSRF protection is built into ASP.NET Core. It ensures that requests made to a web server originate from the same domain. It is typically used in forms with the @Html.AntiForgeryToken() helper, which generates a token that must be validated on the server side.

**Antiforgery with Minimal APIs**

Call [AddAntiforgery](https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.antiforgeryservicecollectionextensions.addantiforgery) and [UseAntiforgery(IApplicationBuilder)](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.antiforgeryapplicationbuilderextensions.useantiforgery" \l "microsoft-aspnetcore-builder-antiforgeryapplicationbuilderextensions-useantiforgery(microsoft-aspnetcore-builder-iapplicationbuilder)) to register antiforgery services in DI. Antiforgery tokens are used to mitigate [cross-site request forgery attacks](https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-9.0).

C#Copy

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

app.MapGet("/", () => "Hello World!");

app.Run();

The antiforgery token is only validated if:

* The endpoint contains metadata implementing [IAntiforgeryMetadata](https://source.dot.net/" \l "Microsoft.AspNetCore.Http.Abstractions/Metadata/IAntiforgeryMetadata.cs,5f49d4d07fc58320) where RequiresValidation=true.
* The HTTP method associated with the endpoint is a relevant [HTTP method](https://developer.mozilla.org/docs/Web/HTTP/Methods). The relevant methods are all [HTTP methods](https://developer.mozilla.org/docs/Web/HTTP/Methods) except for TRACE, OPTIONS, HEAD, and GET.
* The request is associated with a valid endpoint.

***Note:*** When enabled manually, the antiforgery middleware must run after the authentication and authorization middleware to prevent reading form data when the user is unauthenticated.

By default, minimal APIs that accept form data require antiforgery token validation.

**Dependency injection (services)**

ASP.NET Core features built-in [dependency injection (DI)](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-9.0) that makes configured services available throughout an app. Services are added to the DI container with [WebApplicationBuilder.Services](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplicationbuilder.services" \l "microsoft-aspnetcore-builder-webapplicationbuilder-services), builder.Services in the preceding code. When the [WebApplicationBuilder](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplicationbuilder) is instantiated, many [framework-provided services](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-9.0#framework-provided-services) are added automatically.

**Middleware**

The request handling pipeline is composed as a series of middleware components. Each component performs operations on an [HttpContext](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/http-context?view=aspnetcore-9.0) 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{Feature} extension method.



Custom Middleware

The following code moves the middleware delegate to a class:

The middleware class must include:

* A public constructor with a parameter of type [RequestDelegate](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.requestdelegate).
* A public method named Invoke or InvokeAsync. This method must:
  + Return a Task.
  + Accept a first parameter of type [HttpContext](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.httpcontext).

**Configuration**

Application configuration in ASP.NET Core is performed using one or more [configuration providers](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-9.0#cp). Configuration providers read configuration data from key-value pairs using a variety of configuration sources:

* Settings files, such as appsettings.json
* Environment variables
* Azure Key Vault
* Azure App Configuration
* Command-line arguments
* Custom providers, installed or created
* Directory files
* In-memory .NET objects

### appsettings.json

Consider the following appsettings.json file:

JSONCopy

{

"Position": {

"Title": "Editor",

"Name": "Joe Smith"

},

"MyKey": "My appsettings.json Value",

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*"

}

The following code from the [sample download](https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/fundamentals/configuration/index/samples/6.x/ConfigSample) displays several of the preceding configurations settings:

public class TestModel : PageModel

{

// requires using Microsoft.Extensions.Configuration;

private readonly IConfiguration Configuration;

public TestModel(IConfiguration configuration)

{

Configuration = configuration;

}

public ContentResult OnGet()

{

var myKeyValue = Configuration["MyKey"];

var title = Configuration["Position:Title"];

var name = Configuration["Position:Name"];

var defaultLogLevel = Configuration["Logging:LogLevel:Default"];

return Content($"MyKey value: {myKeyValue} \n" +

$"Title: {title} \n" +

$"Name: {name} \n" +

$"Default Log Level: {defaultLogLevel}");

}

}

The options pattern uses classes to provide strongly typed access to groups of related settings. When [configuration settings](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-9.0) are isolated by scenario into separate classes, the app adheres to two important software engineering principles:

* [Encapsulation](https://learn.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles#encapsulation):
  + Classes that depend on configuration settings depend only on the configuration settings that they use.
* [Separation of Concerns](https://learn.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles#separation-of-concerns):
  + Settings for different parts of the app aren't dependent or coupled to one another.

Options also provide a mechanism to validate configuration data.

**Bind hierarchical configuration**

The preferred way to read related configuration values is using the [options pattern](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-9.0). For example, to read the following configuration values:

"Position": {

"Title": "Editor",

"Name": "Joe Smith"

}

Create the following PositionOptions class:

public class PositionOptions

{

public const string Position = "Position";

public string Title { get; set; } = String.Empty;

public string Name { get; set; } = String.Empty;

}

An options class:

* Must be non-abstract.
* Has public read-write properties of the type that have corresponding items in config are bound.
* Has its read-write properties bound to matching entries in configuration.
* Does ***not*** have its fields bound. In the preceding code, Position is not bound. The Position field is used so the string "Position" doesn't need to be hard coded in the app when binding the class to a configuration provider.

The following code:

* Calls [ConfigurationBinder.Bind](https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.configuration.configurationbinder.bind) to bind the PositionOptions class to the Position section.
* Displays the Position configuration data.

public class Test22Model : PageModel

{

private readonly IConfiguration Configuration;

public Test22Model(IConfiguration configuration)

{

Configuration = configuration;

}

public ContentResult OnGet()

{

var positionOptions = new PositionOptions();

Configuration.GetSection(PositionOptions.Position).Bind(positionOptions);

return Content($"Title: {positionOptions.Title} \n" +

$"Name: {positionOptions.Name}");

}

}

**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 three different hosts capable of running an ASP.NET Core app:

* ASP.NET Core WebApplication, also known as the Minimal Host
* .NET Generic Host combined with ASP.NET Core's ConfigureWebHostDefaults
* ASP.NET Core WebHost

The ASP.NET Core WebApplication and WebApplicationBuilder types are recommended and are used in all the ASP.NET Core templates. WebApplication behaves similarly to the .NET Generic Host and exposes many of the same interfaces but requires fewer callbacks to configure. The ASP.NET Core WebHost is available only for backward compatibility.

**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](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/request-features?view=aspnetcore-9.0) composed into an [HttpContext](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.httpcontext).

* [Windows](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-9.0&tabs=windows#tabpanel_1_windows)
* [macOS](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-9.0&tabs=windows#tabpanel_1_macos)
* [Linux](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-9.0&tabs=windows#tabpanel_1_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](https://www.iis.net/). In ASP.NET Core 2.0 and 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.

ASP.NET Core supports creating web APIs using controllers or using minimal APIs. *Controllers* in a web API are classes that derive from [ControllerBase](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase). Controllers are activated and disposed on a per request basis.

**ControllerBase class**

A controller-based web API consists of one or more controller classes that derive from [ControllerBase](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase).

Web API controllers should typically derive from [ControllerBase](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase) rather from [Controller](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controller). Controller derives from [ControllerBase](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase) and adds support for views, so it's for handling web pages, not web API requests. If the same controller must support views and web APIs, derive from Controller.

**Attributes**

The [Microsoft.AspNetCore.Mvc](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc) namespace provides attributes that can be used to configure the behavior of web API controllers and action methods

| **Attribute** | **Notes** |
| --- | --- |
| [[Route]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.routeattribute) | Specifies URL pattern for a controller or action. |
| [[Bind]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.bindattribute) | Specifies prefix and properties to include for model binding. |
| [[HttpGet]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.httpgetattribute) | Identifies an action that supports the HTTP GET action verb. |
| [[Consumes]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.consumesattribute) | Specifies data types that an action accepts. |
| [[Produces]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.producesattribute) | Specifies data types that an action returns. |

**ApiController attribute**

The [[ApiController]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.apicontrollerattribute) attribute can be applied to a controller class to enable the following opinionated, API-specific behaviors:

* [Attribute routing requirement](https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-9.0#attribute-routing-requirement)
* [Automatic HTTP 400 responses](https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-9.0#automatic-http-400-responses)
* [Binding source parameter inference](https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-9.0#binding-source-parameter-inference)
* [Multipart/form-data request inference](https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-9.0#multipartform-data-request-inference)
* [Problem details for error status codes](https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-9.0#problem-details-for-error-status-codes)

**Attribute routing requirement**

The [ApiController] attribute makes attribute routing a requirement. For example:

[ApiController]

[Route("[controller]")]

public class WeatherForecastController : ControllerBase

Actions are inaccessible via [conventional routes](https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-9.0#conventional-routing) defined by UseEndpoints, [UseMvc](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.mvcapplicationbuilderextensions.usemvc), or [UseMvcWithDefaultRoute](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.mvcapplicationbuilderextensions.usemvcwithdefaultroute).

**Automatic HTTP 400 responses**

The [ApiController] attribute makes model validation errors automatically trigger an HTTP 400 response. Consequently, the following code is unnecessary in an action method:

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

**Binding source parameter inference**

A binding source attribute defines the location at which an action parameter's value is found. The following binding source attributes exist:

Expand table

| **Attribute** | **Binding source** |
| --- | --- |
| [[FromBody]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.frombodyattribute) | Request body |
| [[FromForm]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.fromformattribute) | Form data in the request body |
| [[FromHeader]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.fromheaderattribute) | Request header |
| [[FromQuery]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.fromqueryattribute) | Request query string parameter |
| [[FromRoute]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.fromrouteattribute) | Route data from the current request |
| [[FromServices]](https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-9.0#action-injection-with-fromservices) | The request service injected as an action parameter |
| [[AsParameters]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.asparametersattribute) | [Method parameters](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis?view=aspnetcore-9.0#asparam7) |

**Multipart/form-data request inference**

The [ApiController] attribute applies an inference rule for action parameters of type [IFormFile](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.iformfile) and [IFormFileCollection](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.iformfilecollection). The multipart/form-data request content type is inferred for these types.

**Problem details for error status codes**

MVC transforms an error result (a result with status code 400 or higher) to a result with [ProblemDetails](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.problemdetails). The ProblemDetails type is based on the [RFC 7807 specification](https://tools.ietf.org/html/rfc7807) for providing machine-readable error details in an HTTP response.

Consider the following code in a controller action:

if (pet == null)

{

return NotFound();

}

The NotFound method produces an HTTP 404 status code with a ProblemDetails body. For example:

JSONCopy

{

type: "https://tools.ietf.org/html/rfc7231#section-6.5.4",

title: "Not Found",

status: 404,

traceId: "0HLHLV31KRN83:00000001"

}

ASP.NET Core provides the following options for web API controller action return types:

* [Specific type](https://learn.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-9.0#specific-type) – List<Product>
* [IActionResult](https://learn.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-9.0#iactionresult-type) - The [IActionResult](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.iactionresult) return type is appropriate when multiple ActionResult return types are possible in an action. The ActionResult types represent various HTTP status codes. Any non-abstract class deriving from ActionResult qualifies as a valid return type. Some common return types in this category are [BadRequestResult](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.badrequestresult) (400), [NotFoundResult](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.notfoundresult) (404), and [OkObjectResult](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.okobjectresult) (200).
* [ActionResult<T>](https://learn.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-9.0#actionresultt-type)
* [HttpResults](https://learn.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-9.0#httpresults-type) - In addition to the MVC-specific built-in result types ([IActionResult](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.iactionresult) and [ActionResult<T>](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.actionresult-1)), ASP.NET Core includes the [HttpResults](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.httpresults) types that can be used in both [Minimal APIs](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis) and Web API.
  1. Different than the MVC-specific result types, the HttpResults:
  2. Are a results implementation that is processed by a call to [IResult.ExecuteAsync](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.iresult.executeasync).
  3. Does ***not*** leverage the configured [Formatters](https://learn.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting#format-specific-action-results).
  4. Not leveraging the configured formatters means:
     1. Some features like Content negotiation aren't available.
     2. The produced Content-Type is decided by the HttpResults implementation.
* The HttpResults can be useful when sharing code between Minimal APIs and Web API.

**PATCH HTTP request method**

The PUT and [PATCH](https://tools.ietf.org/html/rfc5789) methods are used to update an existing resource. The difference between them is that PUT replaces the entire resource, while PATCH specifies only the changes.

**JSON Patch**

[JSON Patch](https://tools.ietf.org/html/rfc6902) is a format for specifying updates to be applied to a resource. A JSON Patch document has an array of *operations*. Each operation identifies a particular type of change. Examples of such changes include adding an array element or replacing a property value.

For example, the following JSON documents represent a resource, a JSON Patch document for the resource, and the result of applying the Patch operations.

**Resource example**

JSONCopy

{

"customerName": "John",

"orders": [

{

"orderName": "Order0",

"orderType": null

},

{

"orderName": "Order1",

"orderType": null

}

]

}

**JSON patch example**

JSONCopy

[

{

"op": "add",

"path": "/customerName",

"value": "Barry"

},

{

"op": "add",

"path": "/orders/-",

"value": {

"orderName": "Order2",

"orderType": null

}

}

]

In the preceding JSON:

* The op property indicates the type of operation.
* The path property indicates the element to update.
* The value property provides the new value.

**Resource after patch**

Here's the resource after applying the preceding JSON Patch document:

JSONCopy

{

"customerName": "Barry",

"orders": [

{

"orderName": "Order0",

"orderType": null

},

{

"orderName": "Order1",

"orderType": null

},

{

"orderName": "Order2",

"orderType": null

}

]

}

The changes made by applying a JSON Patch document to a resource are atomic. If any operation in the list fails, no operation in the list is applied.

**JSON Patch in ASP.NET Core**

The ASP.NET Core implementation of JSON Patch is provided in the [Microsoft.AspNetCore.JsonPatch](https://www.nuget.org/packages/microsoft.aspnetcore.jsonpatch/) NuGet package.

**Action method code**

In an API controller, an action method for JSON Patch:

* Is annotated with the HttpPatch attribute.
* Accepts a [JsonPatchDocument<TModel>](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.jsonpatch.jsonpatchdocument-1), typically with [[FromBody]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.frombodyattribute).
* Calls [ApplyTo(Object)](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.jsonpatch.jsonpatchdocument.applyto" \l "microsoft-aspnetcore-jsonpatch-jsonpatchdocument-applyto(system-object)) on the patch document to apply the changes.

**Format-specific Action Results**

Some action result types are specific to a particular format, such as [JsonResult](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.jsonresult) and [ContentResult](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.contentresult). Actions can return results that always use a specified format, ignoring a client's request for a different format. For example, returning JsonResult returns JSON-formatted data and returning ContentResult returns plain-text-formatted string data.

[HttpGet("Version")]

public ContentResult GetVersion()

=> Content("v1.0.0");

**Content negotiation**

Content negotiation occurs when the client specifies an [Accept header](https://www.rfc-editor.org/rfc/rfc9110#field.accept). The default format used by ASP.NET Core is [JSON](https://json.org/). Content negotiation is:

* Implemented by [ObjectResult](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.objectresult).
* Built into the status code-specific action results returned from the helper methods. The action results helper methods are based on ObjectResult.

When a model type is returned, the return type is ObjectResult.

By default, ASP.NET Core supports the following media types:

* application/json
* text/json
* text/plain

**The Accept header**

Content *negotiation* takes place when an Accept header appears in the request. When a request contains an accept header, ASP.NET Core:

* Enumerates the media types in the accept header in preference order.
* Tries to find a formatter that can produce a response in one of the formats specified.

If no formatter is found that can satisfy the client's request, ASP.NET Core:

* Returns 406 Not Acceptable if [MvcOptions.ReturnHttpNotAcceptable](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions.returnhttpnotacceptable" \l "microsoft-aspnetcore-mvc-mvcoptions-returnhttpnotacceptable) is set to true, or -
* Tries to find the first formatter that can produce a response.

If no formatter is configured for the requested format, the first formatter that can format the object is used. If no Accept header appears in the request:

* The first formatter that can handle the object is used to serialize the response.
* There isn't any negotiation taking place. The server is determining what format to return.

## Configure formatters

Apps that need to support extra formats can add the appropriate NuGet packages and configure support. There are separate formatters for input and output. Input formatters are used by [Model Binding](https://learn.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-9.0). Output formatters are used to format responses. For information on creating a custom formatter, see [Custom Formatters](https://learn.microsoft.com/en-us/aspnet/core/web-api/advanced/custom-formatters?view=aspnetcore-9.0).

### Add XML format support

To configure XML formatters implemented using [XmlSerializer](https://learn.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer), call [AddXmlSerializerFormatters](https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.mvcxmlmvcbuilderextensions.addxmlserializerformatters):

C#Copy

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()

.AddXmlSerializerFormatters();

**Specify a format**

To restrict the response formats, apply the [[Produces]](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.producesattribute) filter. Like most [Filters](https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-9.0), [Produces] can be applied at the action, controller, or global scope:

[ApiController]

[Route("api/[controller]")]

[Produces("application/json")]

public class TodoItemsController : ControllerBase

**When to use a custom formatter**

Use a custom formatter to add support for a content type that isn't handled by the built-in formatters.

**Overview of how to create a custom formatter**

To create a custom formatter:

* For serializing data sent to the client, create an output formatter class.
* For deserializing data received from the client, create an input formatter class.
* Add instances of formatter classes to the [InputFormatters](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions.inputformatters) and [OutputFormatters](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions.outputformatters) collections in [MvcOptions](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions).

**Create a custom formatter**

To create a formatter:

* Derive the class from the appropriate base class. The sample app derives from [TextOutputFormatter](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.textoutputformatter) and [TextInputFormatter](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.textinputformatter).
* Specify supported media types and encodings in the constructor.
* Override the [CanReadType](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.inputformatter.canreadtype) and [CanWriteType](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.outputformatter.canwritetype) methods.
* Override the [ReadRequestBodyAsync](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.inputformatter.readrequestbodyasync) and [WriteResponseBodyAsync](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.outputformatter.writeresponsebodyasync) methods.

**Developer Exception Page**

The *Developer Exception Page* displays detailed information about unhandled request exceptions. It uses [DeveloperExceptionPageMiddleware](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.diagnostics.developerexceptionpagemiddleware) to capture synchronous and asynchronous exceptions from the HTTP pipeline and to generate error responses. The developer exception page runs early in the middleware pipeline, so that it can catch unhandled exceptions thrown in middleware that follows.

ASP.NET Core apps enable the developer exception page by default when both:

* Running in the [Development environment](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-9.0).
* The app was created with the current templates, that is, by using [WebApplication.CreateBuilder](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplication.createbuilder).

**Exception handler**

In non-development environments, use [Exception Handling Middleware](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/error-handling?view=aspnetcore-9.0) to produce an error payload:

1. In Program.cs, call [UseExceptionHandler](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.exceptionhandlerextensions.useexceptionhandler) to add the Exception Handling Middleware:

var app = builder.Build();

app.UseHttpsRedirection();

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/error");

}

app.UseAuthorization();

app.MapControllers();

app.Run();

1. Configure a controller action to respond to the /error route:

[Route("/error")]

public IActionResult HandleError() =>

Problem();

**IActionResult:**

* IActionResult is an **interface** that represents the result of an action method in ASP.NET Core.
* It is a base interface for various action result types, which can return different kinds of responses (such as JSON, HTML, redirects, etc.).
* This allows you to return different types of results, like OkResult, NotFoundResult, RedirectResult, etc., all of which implement the IActionResult interface.

**ActionResult<T>:**

* ActionResult<T> is a **generic class** that is an extension of ActionResult and specifically designed to return a strongly-typed response.
* This allows you to return both a specific type (e.g., a model or data object) and an HTTP status code in one return value.
* ActionResult<T> is a more recent improvement in ASP.NET Core, intended to offer the best of both worlds:
  + It allows you to return a strongly-typed result (T).
  + It still supports HTTP status codes (ActionResult) for more standard response handling.

**AddSingleton:**

* **Lifetime**: Singleton services are created **once for the entire application** and the same instance is **shared across all requests and consumers**. This means the service is instantiated once when the application starts and persists for the lifetime of the application.
* **Use Case**: Use AddSingleton when the service holds global state or when you want to optimize for performance and reuse, as only one instance will exist for the whole application.
* **Example**: Services like logging, caching, or configuration management that do not depend on specific request data and can be reused across multiple requests.

**AddScoped:**

* **Lifetime**: Scoped services are created **once per HTTP request or per scope** (such as within a unit of work or a transaction). This means a new instance is created for each **HTTP request**, but the same instance is used throughout the lifecycle of that request.
* **Use Case**: Use AddScoped when the service needs to maintain state for the duration of a request or a unit of work (e.g., interacting with a database, where the service should not change between method calls within the same request).
* **Example**: Shopping Cart Service in Ecom.

**AddTransient:**

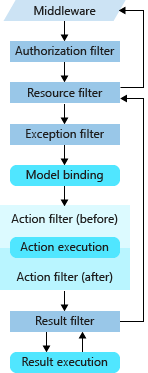
* **Lifetime**: Transient services are **created each time they are requested**.
* **Use Case**: Use AddTransient for lightweight, stateless services that are not expensive to create and are short-lived
* **Example**: Payment Processor in Ecom

In ASP.NET Core, background tasks can be implemented as *hosted services*. A hosted service is a class with background task logic that implements the [IHostedService](https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.hosting.ihostedservice) interface. This article provides three hosted service examples:

* Background task that runs on a timer.
* Hosted service that activates a [scoped service](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-9.0#service-lifetimes). The scoped service can use [dependency injection (DI)](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-9.0).
* Queued background tasks that run sequentially.

IWebHostEnvironment provides information about the web hosting environment (e.g., development, staging, production). It allows you to access environment-specific details like configuration and application root path. Supported in (.Net Core 5 <)

Filters run within the *ASP.NET Core action invocation pipeline*, sometimes referred to as the *filter pipeline*. The filter pipeline runs after ASP.NET Core selects the action to execute:



**Difference between UseRouting and UseEndpoints**

1. **UseRouting**: Maps the request to a route.
2. **UseEndpoints**: Maps the route to an actual handler (such as a controller action, Razor page, or other endpoints).

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddControllersWithViews();

// Razor Pages or MVC can be added here

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

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

app.UseHsts();

}

// Step 1: UseRouting to match the route

app.UseRouting();

// Step 2: UseEndpoints to map the request to the specific handler

app.UseEndpoints(endpoints =>

{

// Configure MVC, API, or Razor Pages routing

endpoints.MapControllerRoute(

name: "default",

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

});

}

}