The following table contains examples of methods in `ControllerBase`.

Expand table

Method	Notes
BadRequest	Returns 400 status code.
NotFound	Returns 404 status code.
PhysicalFile	Returns a file.
TryUpdateModelAsync	Invokes model binding.
TryValidateModel	Invokes model validation.

For a list of all available methods and properties, see ControllerBase.

Attributes

The Microsoft.AspNetCore.Mvc namespace provides attributes that can be used to configure the behavior of web API controllers and action methods. The following example uses attributes to specify the supported HTTP action verb and any known HTTP status codes that could be returned:

C#

```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Pet> Create(Pet pet)
{
    pet.Id = _petsInMemoryStore.Any() ?
             _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
    _petsInMemoryStore.Add(pet);

    return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);
}
```

Here are some more examples of attributes that are available.

Expand table

Attribute	Notes
[Route]	Specifies URL pattern for a controller or action.

Attribute	Notes
[Bind]	Specifies prefix and properties to include for model binding.
[HttpGet]	Identifies an action that supports the HTTP GET action verb.
[Consumes]	Specifies data types that an action accepts.
[Produces]	Specifies data types that an action returns.

For a list that includes the available attributes, see the Microsoft.AspNetCore.Mvc namespace.

ApiController attribute

The [ApiController] attribute can be applied to a controller class to enable the following opinionated, API-specific behaviors:

- Attribute routing requirement
- Automatic HTTP 400 responses
- Binding source parameter inference
- Multipart/form-data request inference
- Problem details for error status codes

Attribute on specific controllers

The `[ApiController]` attribute can be applied to specific controllers, as in the following example from the project template:

C#

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Attribute on multiple controllers

One approach to using the attribute on more than one controller is to create a custom base controller class annotated with the [ApiController] attribute. The following example shows a custom base class and a controller that derives from it:

C#

```
[ApiController]
public class MyControllerBase : ControllerBase
{
}
```

C#

```
[Produces(MediaTypeNames.Application.Json)]
[Route("[controller]")]
public class PetsController : MyControllerBase
```

Attribute on an assembly

The `[ApiController]` attribute can be applied to an assembly. When the `[ApiController]` attribute is applied to an assembly, all controllers in the assembly have the `[ApiController]` attribute applied. There's no way to opt out for individual controllers. Apply the assembly-level attribute to the `Program.cs` file:

C#

```
using Microsoft.AspNetCore.Mvc;
[assembly: ApiController]
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Attribute routing requirement

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

C#

```
[ApiController]
[Route("[controller]")]
```

```
public class WeatherForecastController : ControllerBase
```

Actions are inaccessible via conventional routes defined by `UseEndpoints`, UseMvc, or 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:

C#

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

ASP.NET Core MVC uses the ModelStateInvalidFilter action filter to do the preceding check.

Default BadRequest response

The default response type for an HTTP 400 response is ValidationProblemDetails. The following response body is an example of the serialized type:

JSON

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "|7fb5e16a-4c8f23bbfc974667.",
  "errors": {
    "": [
      "A non-empty request body is required."
    ]
  }
}
```

The `ValidationProblemDetails` type:

- Provides a machine-readable format for specifying errors in web API responses.
- Complies with the RFC 7807 specification.

To make automatic and custom responses consistent, call the ValidationProblem method instead of BadRequest. `ValidationProblem` returns a ValidationProblemDetails object as well as the automatic response.

Log automatic 400 responses

To log automatic 400 responses, set the InvalidModelStateResponseFactory delegate property to perform custom processing. By default, `InvalidModelStateResponseFactory` uses ProblemDetailsFactory to create an instance of ValidationProblemDetails.

The following example shows how to retrieve an instance of ILogger<TCategoryName> to log information about an automatic 400 response:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
      // To preserve the default behavior, capture the original delegate to
call later.
        var builtInFactory = options.InvalidModelStateResponseFactory;

        options.InvalidModelStateResponseFactory = context =>
        {
            var logger = context.HttpContext.RequestServices
                                .GetRequiredService<ILogger<Program>>();

            // Perform logging here.
            // ...

            // Invoke the default behavior, which produces a
ValidationProblemDetails
            // response.
            // To produce a custom response, return a different
implementation of
            // IActionResult instead.
            return builtInFactory(context);
        };
    });

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();
```

```
app.Run();
```

Disable automatic 400 response

To disable the automatic 400 behavior, set the SuppressModelStateInvalidFilter property to `true`. Add the following highlighted code:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

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]	Request body
[FromForm]	Form data in the request body

Attribute	Binding source
[FromHeader]	Request header
[FromQuery]	Request query string parameter
[FromRoute]	Route data from the current request
[FromServices]	The request service injected as an action parameter
[AsParameters]	Method parameters

> ⚠ **Warning**
>
> Don't use `[FromRoute]` when values might contain `%2f` (that is `/`). `%2f` won't be unescaped to `/`. Use `[FromQuery]` if the value might contain `%2f`.

Without the `[ApiController]` attribute or binding source attributes like `[FromQuery]`, the ASP.NET Core runtime attempts to use the complex object model binder. The complex object model binder pulls data from value providers in a defined order.

In the following example, the `[FromQuery]` attribute indicates that the `discontinuedOnly` parameter value is provided in the request URL's query string:

C#

```
[HttpGet]
public ActionResult<List<Product>> Get(
    [FromQuery] bool discontinuedOnly = false)
{
    List<Product> products = null;

    if (discontinuedOnly)
    {
        products = _productsInMemoryStore.Where(p =>
p.IsDiscontinued).ToList();
    }
    else
    {
        products = _productsInMemoryStore;
    }

    return products;
}
```

The `[ApiController]` attribute applies inference rules for the default data sources of action parameters. These rules save you from having to identify binding sources

manually by applying attributes to the action parameters. The binding source inference rules behave as follows:

- `[FromServices]` is inferred for complex type parameters registered in the DI Container.
- `[FromBody]` is inferred for complex type parameters not registered in the DI Container. An exception to the `[FromBody]` inference rule is any complex, built-in type with a special meaning, such as IFormCollection and CancellationToken. The binding source inference code ignores those special types.
- `[FromForm]` is inferred for action parameters of type IFormFile and IFormFileCollection. It's not inferred for any simple or user-defined types.
- `[FromRoute]` is inferred for any action parameter name matching a parameter in the route template. When more than one route matches an action parameter, any route value is considered `[FromRoute]`.
- `[FromQuery]` is inferred for any other action parameters.

FromBody inference notes

`[FromBody]` isn't inferred for simple types such as `string` or `int`. Therefore, the `[FromBody]` attribute should be used for simple types when that functionality is needed.

When an action has more than one parameter bound from the request body, an exception is thrown. For example, all of the following action method signatures cause an exception:

- `[FromBody]` inferred on both because they're complex types.

  C#

  ```
  [HttpPost]
  public IActionResult Action1(Product product, Order order)
  ```

- `[FromBody]` attribute on one, inferred on the other because it's a complex type.

  C#

  ```
  [HttpPost]
  public IActionResult Action2(Product product, [FromBody] Order order)
  ```

- `[FromBody]` attribute on both.

  C#

```
[HttpPost]
public IActionResult Action3([FromBody] Product product, [FromBody]
Order order)
```

FromServices inference notes

Parameter binding binds parameters through dependency injection when the type is configured as a service. This means it's not required to explicitly apply the [FromServices] attribute to a parameter. In the following code, both actions return the time:

C#

```
[Route("[controller]")]
[ApiController]
public class MyController : ControllerBase
{
    public ActionResult GetWithAttribute([FromServices] IDateTime dateTime)
                                                        => Ok(dateTime.Now);

    [Route("noAttribute")]
    public ActionResult Get(IDateTime dateTime) => Ok(dateTime.Now);
}
```

In rare cases, automatic DI can break apps that have a type in DI that is also accepted in an API controller's action methods. It's not common to have a type in DI and as an argument in an API controller action.

To disable `[FromServices]` inference for a single action parameter, apply the desired binding source attribute to the parameter. For example, apply the `[FromBody]` attribute to an action parameter that should be bound from the body of the request.

To disable `[FromServices]` inference globally, set DisableImplicitFromServicesParameters to `true`:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddSingleton<IDateTime, SystemDateTime>();
```

```
builder.Services.Configure<ApiBehaviorOptions>(options =>
{
    options.DisableImplicitFromServicesParameters = true;
});

var app = builder.Build();

app.MapControllers();

app.Run();
```

Types are checked at app startup with IServiceProviderIsService to determine if an argument in an API controller action comes from DI or from the other sources.

The mechanism to infer binding source of API Controller action parameters uses the following rules:

- A previously specified BindingInfo.BindingSource is never overwritten.
- A complex type parameter, registered in the DI container, is assigned BindingSource.Services.
- A complex type parameter, not registered in the DI container, is assigned BindingSource.Body.
- A parameter with a name that appears as a route value in ***any*** route template is assigned BindingSource.Path.
- All other parameters are BindingSource.Query.

Disable inference rules

To disable binding source inference, set SuppressInferBindingSourcesForParameters to `true`:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
        options.DisableImplicitFromServicesParameters = true;
    });
```

```
var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Multipart/form-data request inference

The `[ApiController]` attribute applies an inference rule for action parameters of type IFormFile and IFormFileCollection. The `multipart/form-data` request content type is inferred for these types.

To disable the default behavior, set the SuppressConsumesConstraintForFormFileParameters property to `true`:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Problem details for error status codes

MVC transforms an error result (a result with status code 400 or higher) to a result with ProblemDetails. The `ProblemDetails` type is based on the RFC 7807 specification for providing machine-readable error details in an HTTP response.

Consider the following code in a controller action:

C#

```
if (pet == null)
{
    return NotFound();
}
```

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

JSON

```
{
  type: "https://tools.ietf.org/html/rfc7231#section-6.5.4",
  title: "Not Found",
  status: 404,
  traceId: "0HLHLV31KRN83:00000001"
}
```

Disable ProblemDetails response

The automatic creation of a `ProblemDetails` for error status codes is disabled when the SuppressMapClientErrors property is set to `true`. Add the following code:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
```

```
    });

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Define supported request content types with the [Consumes] attribute

By default, an action supports all available request content types. For example, if an app is configured to support both JSON and XML input formatters, an action supports multiple content types, including `application/json` and `application/xml`.

The [Consumes] attribute allows an action to limit the supported request content types. Apply the `[Consumes]` attribute to an action or controller, specifying one or more content types:

C#

```
[HttpPost]
[Consumes("application/xml")]
public IActionResult CreateProduct(Product product)
```

In the preceding code, the `CreateProduct` action specifies the content type `application/xml`. Requests routed to this action must specify a `Content-Type` header of `application/xml`. Requests that don't specify a `Content-Type` header of `application/xml` result in a 415 Unsupported Media Type response.

The `[Consumes]` attribute also allows an action to influence its selection based on an incoming request's content type by applying a type constraint. Consider the following example:

C#

```
[ApiController]
[Route("api/[controller]")]
public class ConsumesController : ControllerBase
{
    [HttpPost]
```

```
    [Consumes("application/json")]
    public IActionResult PostJson(IEnumerable<int> values) =>
        Ok(new { Consumes = "application/json", Values = values });

    [HttpPost]
    [Consumes("application/x-www-form-urlencoded")]
    public IActionResult PostForm([FromForm] IEnumerable<int> values) =>
        Ok(new { Consumes = "application/x-www-form-urlencoded", Values =
values });
}
```

In the preceding code, `ConsumesController` is configured to handle requests sent to the `https://localhost:5001/api/Consumes` URL. Both of the controller's actions, `PostJson` and `PostForm`, handle POST requests with the same URL. Without the `[Consumes]` attribute applying a type constraint, an ambiguous match exception is thrown.

The `[Consumes]` attribute is applied to both actions. The `PostJson` action handles requests sent with a `Content-Type` header of `application/json`. The `PostForm` action handles requests sent with a `Content-Type` header of `application/x-www-form-urlencoded`.

Additional resources

- View or download sample code. (How to download).
- Controller action return types in ASP.NET Core web API
- Handle errors in ASP.NET Core controller-based web APIs
- Custom formatters in ASP.NET Core Web API
- Format response data in ASP.NET Core Web API
- ASP.NET Core web API documentation with Swagger / OpenAPI
- Routing to controller actions in ASP.NET Core
- Use port tunneling Visual Studio to debug web APIs
- Create a web API with ASP.NET Core

Tutorial: Create a web API with ASP.NET Core

Article • 08/23/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson and Kirk Larkin

This tutorial teaches the basics of building a controller-based web API that uses a database. Another approach to creating APIs in ASP.NET Core is to create *minimal APIs*. For help with choosing between minimal APIs and controller-based APIs, see APIs overview. For a tutorial on creating a minimal API, see Tutorial: Create a minimal API with ASP.NET Core.

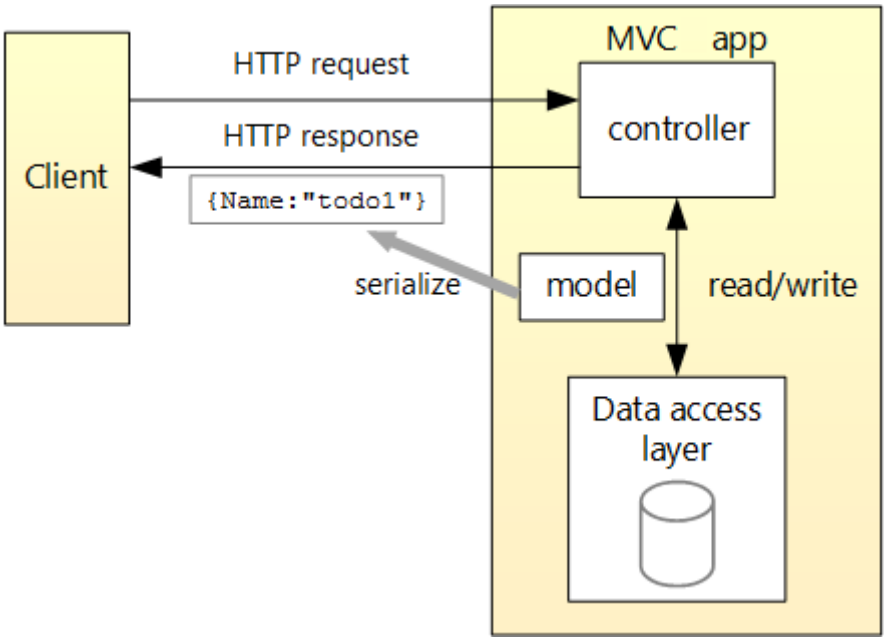
Overview

This tutorial creates the following API:

Expand table

API	Description	Request body	Response body
`GET /api/todoitems`	Get all to-do items	None	Array of to-do items
`GET /api/todoitems/{id}`	Get an item by ID	None	To-do item
`POST /api/todoitems`	Add a new item	To-do item	To-do item
`PUT /api/todoitems/{id}`	Update an existing item	To-do item	None
`DELETE /api/todoitems/{id}`	Delete an item	None	None

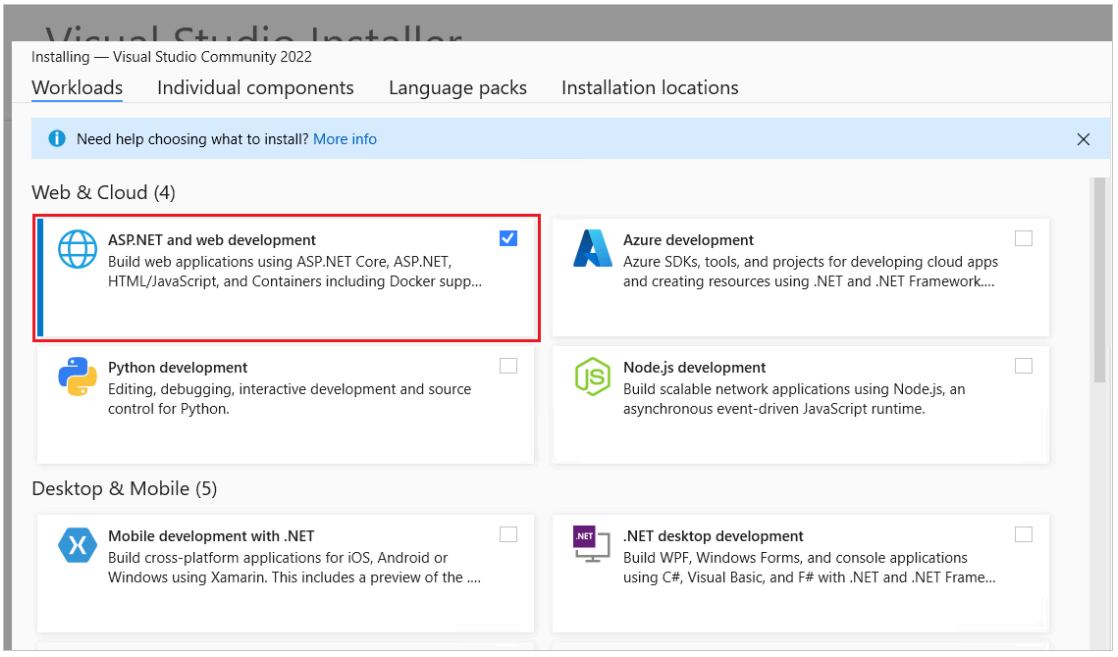
The following diagram shows the design of the app.



Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Create a web project

Visual Studio

- From the **File** menu, select **New** > **Project**.
- Enter *Web API* in the search box.
- Select the **ASP.NET Core Web API** template and select **Next**.
- In the **Configure your new project dialog**, name the project *TodoApi* and select **Next**.
- In the **Additional information** dialog:
  - Confirm the **Framework** is **.NET 8.0 (Long Term Support)**.
  - Confirm the checkbox for **Use controllers(uncheck to use minimal APIs)** is checked.
  - Confirm the checkbox for **Enable OpenAPI support** is checked.
  - Select **Create**.

Add a NuGet package

A NuGet package must be added to support the database used in this tutorial.

- From the **Tools** menu, select **NuGet Package Manager** > **Manage NuGet Packages for Solution**.
- Select the **Browse** tab.
- Enter **Microsoft.EntityFrameworkCore.InMemory** in the search box, and then select `Microsoft.EntityFrameworkCore.InMemory`.
- Select the **Project** checkbox in the right pane and then select **Install**.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

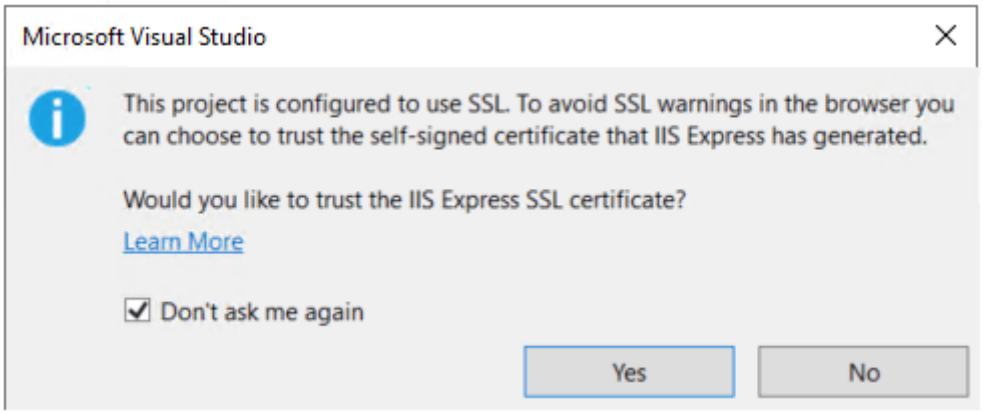
Test the project

The project template creates a `WeatherForecast` API with support for Swagger.

Visual Studio

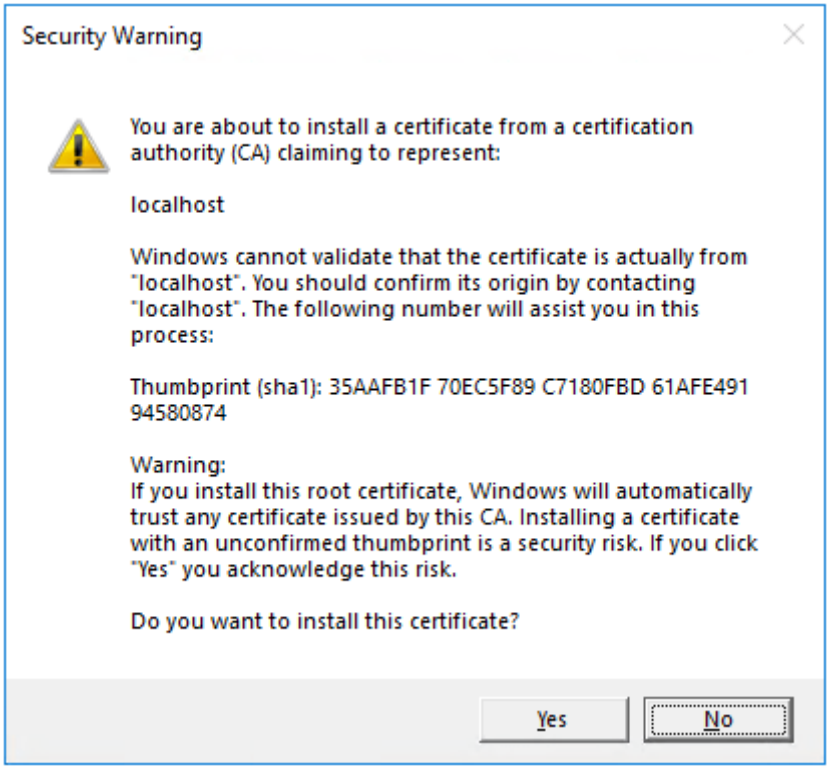
Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



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

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio launches the default browser and navigates to `https://localhost:<port>/swagger/index.html`, where `<port>` is a randomly chosen port number set at the project creation.

The Swagger page `/swagger/index.html` is displayed. Select **GET** > **Try it out** > **Execute**. The page displays:

- The Curl command to test the WeatherForecast API.
- The URL to test the WeatherForecast API.
- The response code, body, and headers.
- A drop-down list box with media types and the example value and schema.

If the Swagger page doesn't appear, see this GitHub issue.

Swagger is used to generate useful documentation and help pages for web APIs. This tutorial uses Swagger to test the app. For more information on Swagger, see ASP.NET Core web API documentation with Swagger / OpenAPI.

Copy and paste the **Request URL** in the browser: `https://localhost:<port>/weatherforecast`

JSON similar to the following example is returned:

JSON


```
[
    {
        "date": "2019-07-16T19:04:05.7257911-06:00",
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
```

```
        "summary": "Chilly"
    }
]
```

Add a model class

A *model* is a set of classes that represent the data that the app manages. The model for this app is the `TodoItem` class.

Visual Studio

- In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder `Models`.
- Right-click the `Models` folder and select **Add** > **Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following:

C#

```
namespace TodoApi.Models;

public class TodoItem
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the `Models` folder is used by convention.

Add a database context

The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the Microsoft.EntityFrameworkCore.DbContext class.

Visual Studio

- Right-click the `Models` folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.

- Enter the following code:

```
C#

using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models;

public class TodoContext : DbContext
{
    public TodoContext(DbContextOptions<TodoContext> options)
        : base(options)
    {
    }

    public DbSet<TodoItem> TodoItems { get; set; } = null!;
}
```

Register the database context

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers.

Update `Program.cs` with the following highlighted code:

```
C#

using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddDbContext<TodoContext>(opt =>
    opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
```

```
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The preceding code:

- Adds `using` directives.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

Scaffold a controller

Visual Studio

- Right-click the `Controllers` folder.
- Select **Add** > **New Scaffolded Item**.
- Select **API Controller with actions, using Entity Framework**, and then select **Add**.
- In the **Add API Controller with actions, using Entity Framework** dialog:
  - Select **TodoItem (TodoApi.Models)** in the **Model class**.
  - Select **TodoContext (TodoApi.Models)** in the **Data context class**.
  - Select **Add**.

  If the scaffolding operation fails, select **Add** to try scaffolding a second time.

The generated code:

- Marks the class with the [ApiController] attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

The ASP.NET Core templates for:

- Controllers with views include `[action]` in the route template.
- API controllers don't include `[action]` in the route template.

When the `[action]` token isn't in the route template, the action name (method name) isn't included in the endpoint. That is, the action's associated method name isn't used in the matching route.

Update the PostTodoItem create method

Update the return statement in the `PostTodoItem` to use the nameof operator:

C#

```
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    //    return CreatedAtAction("GetTodoItem", new { id = todoItem.Id },
todoItem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id },
todoItem);
}
```

The preceding code is an `HTTP POST` method, as indicated by the [HttpPost] attribute. The method gets the value of the `TodoItem` from the body of the HTTP request.

For more information, see Attribute routing with Http[Verb] attributes.

The CreatedAtAction method:

- Returns an HTTP 201 status code if successful. `HTTP 201` is the standard response for an `HTTP POST` method that creates a new resource on the server.
- Adds a Location header to the response. The `Location` header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

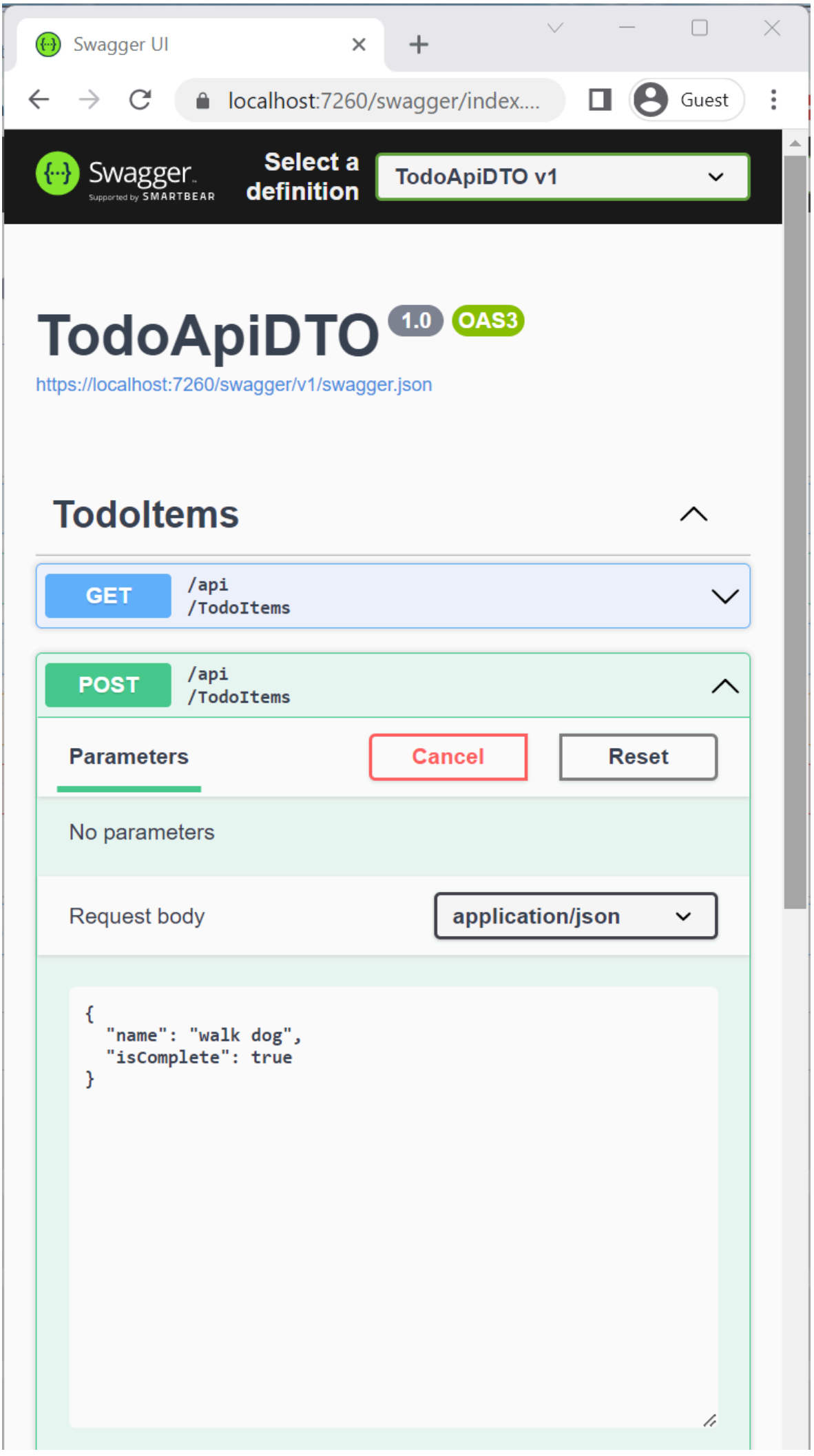
Test PostTodoItem

- Press Ctrl+F5 to run the app.

- In the Swagger browser window, select **POST /api/TodoItems**, and then select **Try it out**.
- In the **Request body** input window, update the JSON. For example,

```JSON
{
  "name": "walk dog",
  "isComplete": true
}
```

- Select **Execute**


Swagger UI
localhost:7260/swagger/index....
Guest
Swagger
Supported by SMARTBEAR
Select a definition
TodoApiDTO v1
TodoApiDTO
1.0
OAS3
https://localhost:7260/swagger/v1/swagger.json
TodoItems
GET
/api
/TodoItems
POST
/api
/TodoItems
Parameters
Cancel
Reset
No parameters
Request body
application/json
{
"name": "walk dog",
"isComplete": true
}

Test the location header URI

In the preceding POST, the Swagger UI shows the location header under **Response headers**. For example, `location: https://localhost:7260/api/TodoItems/1`. The location header shows the URI to the created resource.

To test the location header:

- In the Swagger browser window, select **GET /api/TodoItems/{id}**, and then select **Try it out**.
- Enter `1` in the `id` input box, and then select **Execute**.

GET /api/TodoItems/{id}

Parameters Cancel

Name	Description
id * required integer($int64) (path)	1

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'https://localhost:7260/api/TodoItems/1' \
  -H 'accept: text/plain'
```

Request URL

```
https://localhost:7260/api/TodoItems/1
```

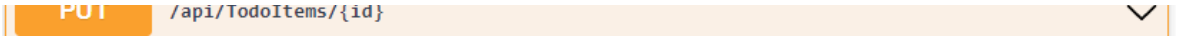
Server response

Code	Details

200	Response body
	`{ "id": 1, "name": "walk dog", "isComplete": true }` Download
	Response headers
	`content-type: application/json; charset=utf-8`
	`date: Fri,21 Oct 2022 00:32:45 GMT`
	`server: Kestrel`

Responses

Code	Description	Links
200	Success Media type: text/plain Controls Accept header. Example Value | Schema `{ "id": 0, "name": "string", "isComplete": true }`	*No links*



Examine the GET methods

Two GET endpoints are implemented:

- `GET /api/todoitems`
- `GET /api/todoitems/{id}`

The previous section showed an example of the `/api/todoitems/{id}` route.

Follow the POST instructions to add another todo item, and then test the `/api/todoitems` route using Swagger.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request doesn't return any data. If no data is returned, POST data to the app.

Routing and URL paths

The [HttpGet] attribute denotes a method that responds to an `HTTP GET` request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

```
C#

[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
```

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoItems**Controller, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.

- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.

In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

Return values

The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an `HTTP 200` response.

The PutTodoItem method

Examine the `PutTodoItem` method:

C#

```
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;
```

```
    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses `HTTP PUT`. The response is 204 (No Content). According to the HTTP specification, a `PUT` request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

Test the PutTodoItem method

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Using the Swagger UI, use the PUT button to update the `TodoItem` that has Id = 1 and set its name to `"feed fish"`. Note the response is HTTP 204 No Content.

The DeleteTodoItem method

Examine the `DeleteTodoItem` method:

C#

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
```

```
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Test the DeleteTodoItem method

Use the Swagger UI to delete the `TodoItem` that has Id = 1. Note the response is HTTP 204 No Content.

Test with other tools

There are many other tools that can be used to test web APIs, for example:

- Visual Studio Endpoints Explorer and .http files
- http-repl
- curl. Swagger uses `curl` and shows the `curl` commands it submits.
- Fiddler

For more information, see:

- Minimal API tutorial: test with .http files and Endpoints Explorer
- Install and test APIs with http-repl

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this, and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this tutorial.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `TodoItem` class to include a secret field:

C#

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
        public string? Secret { get; set; }
    }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a DTO model:

C#

```
namespace TodoApi.Models;

public class TodoItemDTO
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Update the `TodoItemsController` to use `TodoItemDTO`:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi.Controllers;

[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
{
    private readonly TodoContext _context;

    public TodoItemsController(TodoContext context)
```

```
    {
        _context = context;
    }

    // GET: api/TodoItems
    [HttpGet]
    public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
    {
        return await _context.TodoItems
            .Select(x => ItemToDTO(x))
            .ToListAsync();
    }

    // GET: api/TodoItems/5
    // <snippet_GetByID>
    [HttpGet("{id}")]
    public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            return NotFound();
        }

        return ItemToDTO(todoItem);
    }
    // </snippet_GetByID>

    // PUT: api/TodoItems/5
    // To protect from overposting attacks, see
https://go.microsoft.com/fwlink/?linkid=2123754
    // <snippet_Update>
    [HttpPut("{id}")]
    public async Task<IActionResult> PutTodoItem(long id, TodoItemDTO
todoDTO)
    {
        if (id != todoDTO.Id)
        {
            return BadRequest();
        }

        var todoItem = await _context.TodoItems.FindAsync(id);
        if (todoItem == null)
        {
            return NotFound();
        }

        todoItem.Name = todoDTO.Name;
        todoItem.IsComplete = todoDTO.IsComplete;

        try
        {
            await _context.SaveChangesAsync();
        }
```

```
        catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
        {
            return NotFound();
        }

        return NoContent();
    }
    // </snippet_Update>

    // POST: api/TodoItems
    // To protect from overposting attacks, see
https://go.microsoft.com/fwlink/?linkid=2123754
    // <snippet_Create>
    [HttpPost]
    public async Task<ActionResult<TodoItemDTO>> PostTodoItem(TodoItemDTO
todoDTO)
    {
        var todoItem = new TodoItem
        {
            IsComplete = todoDTO.IsComplete,
            Name = todoDTO.Name
        };

        _context.TodoItems.Add(todoItem);
        await _context.SaveChangesAsync();

        return CreatedAtAction(
            nameof(GetTodoItem),
            new { id = todoItem.Id },
            ItemToDTO(todoItem));
    }
    // </snippet_Create>

    // DELETE: api/TodoItems/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);
        if (todoItem == null)
        {
            return NotFound();
        }

        _context.TodoItems.Remove(todoItem);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    private bool TodoItemExists(long id)
    {
        return _context.TodoItems.Any(e => e.Id == id);
    }

    private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
```

```
        new TodoItemDTO
        {
            Id = todoItem.Id,
            Name = todoItem.Name,
            IsComplete = todoItem.IsComplete
        };
}
```

Verify you can't post or get the secret field.

Call the web API with JavaScript

See Tutorial: Call an ASP.NET Core web API with JavaScript.

Web API video series

See Video: Beginner's Series to: Web APIs.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Microsoft Entra ID
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

> **Important**
>
> Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see **Migrate from ASP.NET Core 5.0 to 6.0**.

For more information, see the Duende Identity Server documentation (Duende Software website).

Publish to Azure

For information on deploying to Azure, see Quickstart: Deploy an ASP.NET web app.

Additional resources

View or download sample code for this tutorial. See how to download.

For more information, see the following resources:

- Create web APIs with ASP.NET Core
- Tutorial: Create a minimal API with ASP.NET Core
- ASP.NET Core web API documentation with Swagger / OpenAPI
- Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8
- Routing to controller actions in ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Deploy ASP.NET Core apps to Azure App Service
- Host and deploy ASP.NET Core
- Create a web API with ASP.NET Core

Create a web API with ASP.NET Core and MongoDB

Article • 04/25/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Pratik Khandelwal and Scott Addie

This tutorial creates a web API that runs Create, Read, Update, and Delete (CRUD) operations on a MongoDB NoSQL database.

In this tutorial, you learn how to:

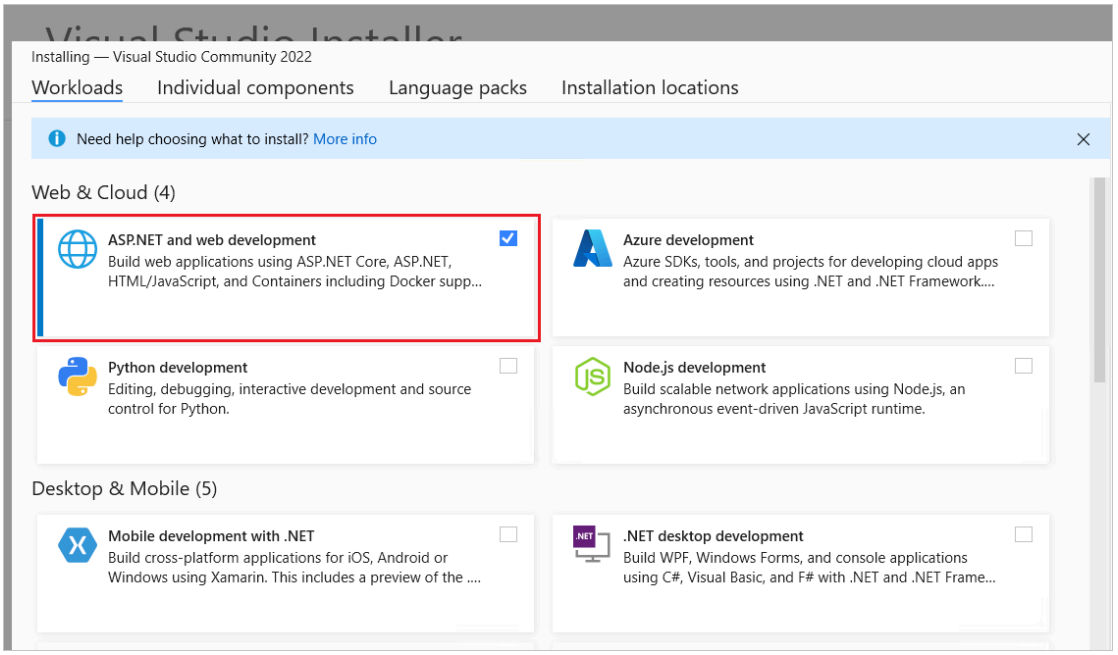
- ✓ Configure MongoDB
- ✓ Create a MongoDB database
- ✓ Define a MongoDB collection and schema
- ✓ Perform MongoDB CRUD operations from a web API
- ✓ Customize JSON serialization

Prerequisites

- MongoDB 6.0.5 or later
- MongoDB Shell

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Configure MongoDB

Enable MongoDB and MongoDB Shell access from anywhere on the development machine (Windows/Linux/macOS):

1. Download and Install MongoDB Shell:

    - macOS/Linux: Choose a directory to extract the MongoDB Shell to. Add the resulting path for `mongosh` to the `PATH` environment variable.
    - Windows: MongoDB Shell (mongosh.exe) is installed at *C:\Users<user>\AppData\Local\Programs\mongosh*. Add the resulting path for `mongosh.exe` to the `PATH` environment variable.

2. Download and Install MongoDB:

    - macOS/Linux: Verify the directory that MongoDB was installed at, usually in */usr/local/mongodb*. Add the resulting path for `mongodb` to the `PATH` environment variable.
    - Windows: MongoDB is installed at *C:\Program Files\MongoDB* by default. Add *C:\Program Files\MongoDB\Server\<version_number>\bin* to the `PATH` environment variable.

3. Choose a Data Storage Directory: Select a directory on your development machine for storing data. Create the directory if it doesn't exist. The MongoDB Shell doesn't create new directories:

    - macOS/Linux: For example, `/usr/local/var/mongodb`.

- Windows: For example, `C:\\BooksData`.

4. In the OS command shell (not the MongoDB Shell), use the following command to connect to MongoDB on default port 27017. Replace `<data_directory_path>` with the directory chosen in the previous step.

```
Console

mongod --dbpath <data_directory_path>
```

Use the previously installed MongoDB Shell in the following steps to create a database, make collections, and store documents. For more information on MongoDB Shell commands, see mongosh.

1. Open a MongoDB command shell instance by launching `mongosh.exe`.

2. In the command shell, connect to the default test database by running the following command:

```
Console

mongosh
```

3. Run the following command in the command shell:

```
Console

use BookStore
```

   A database named *BookStore* is created if it doesn't already exist. If the database does exist, its connection is opened for transactions.

4. Create a `Books` collection using following command:

```
Console

db.createCollection('Books')
```

   The following result is displayed:

```
Console

{ "ok" : 1 }
```

5. Define a schema for the `Books` collection and insert two documents using the following command:

```
Console

db.Books.insertMany([{ "Name": "Design Patterns", "Price": 54.93,
"Category": "Computers", "Author": "Ralph Johnson" }, { "Name": "Clean
Code", "Price": 43.15, "Category": "Computers","Author": "Robert C.
Martin" }])
```

A result similar to the following is displayed:

```
Console

{
    "acknowledged" : true,
    "insertedIds" : [
        ObjectId("61a6058e6c43f32854e51f51"),
        ObjectId("61a6058e6c43f32854e51f52")
    ]
}
```

> **Note**
>
> The `ObjectId`s shown in the preceding result won't match those shown in the command shell.

6. View the documents in the database using the following command:

```
Console

db.Books.find().pretty()
```

A result similar to the following is displayed:

```
Console

{
    "_id" : ObjectId("61a6058e6c43f32854e51f51"),
    "Name" : "Design Patterns",
    "Price" : 54.93,
    "Category" : "Computers",
    "Author" : "Ralph Johnson"
}
{
    "_id" : ObjectId("61a6058e6c43f32854e51f52"),
```

```
    "Name" : "Clean Code",
    "Price" : 43.15,
    "Category" : "Computers",
    "Author" : "Robert C. Martin"
}
```

The schema adds an autogenerated `_id` property of type `ObjectId` for each document.

Create the ASP.NET Core web API project

Visual Studio

1. Go to **File** > **New** > **Project**.
2. Select the **ASP.NET Core Web API** project type, and select **Next**.
3. Name the project *BookStoreApi*, and select **Next**.
4. Select the **.NET 8.0 (Long Term support)** framework and select **Create**.
5. In the **Package Manager Console** window, navigate to the project root. Run the following command to install the .NET driver for MongoDB:

PowerShell

```
Install-Package MongoDB.Driver
```

Add an entity model

1. Add a *Models* directory to the project root.
2. Add a `Book` class to the *Models* directory with the following code:

C#

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BookStoreApi.Models;

public class Book
{
```

```
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string? Id { get; set; }

    [BsonElement("Name")]
    public string BookName { get; set; } = null!;

    public decimal Price { get; set; }

    public string Category { get; set; } = null!;

    public string Author { get; set; } = null!;
}
```

In the preceding class, the `Id` property is:

- Required for mapping the Common Language Runtime (CLR) object to the MongoDB collection.
- Annotated with [BsonId] to make this property the document's primary key.
- Annotated with [BsonRepresentation(BsonType.ObjectId)] to allow passing the parameter as type `string` instead of an ObjectId structure. Mongo handles the conversion from `string` to `ObjectId`.

The `BookName` property is annotated with the [BsonElement] attribute. The attribute's value of `Name` represents the property name in the MongoDB collection.

Add a configuration model

1. Add the following database configuration values to `appsettings.json`:

JSON

```
{
    "BookStoreDatabase": {
        "ConnectionString": "mongodb://localhost:27017",
        "DatabaseName": "BookStore",
        "BooksCollectionName": "Books"
    },
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*"
}
```

2. Add a `BookStoreDatabaseSettings` class to the *Models* directory with the following code:

```C#
namespace BookStoreApi.Models;

public class BookStoreDatabaseSettings
{
    public string ConnectionString { get; set; } = null!;

    public string DatabaseName { get; set; } = null!;

    public string BooksCollectionName { get; set; } = null!;
}
```

   The preceding `BookStoreDatabaseSettings` class is used to store the `appsettings.json` file's `BookStoreDatabase` property values. The JSON and C# property names are named identically to ease the mapping process.

3. Add the following highlighted code to `Program.cs`:

```C#
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.Configure<BookStoreDatabaseSettings>(
    builder.Configuration.GetSection("BookStoreDatabase"));
```

   In the preceding code, the configuration instance to which the `appsettings.json` file's `BookStoreDatabase` section binds is registered in the Dependency Injection (DI) container. For example, the `BookStoreDatabaseSettings` object's `ConnectionString` property is populated with the `BookStoreDatabase:ConnectionString` property in `appsettings.json`.

4. Add the following code to the top of `Program.cs` to resolve the `BookStoreDatabaseSettings` reference:

```C#
using BookStoreApi.Models;
```

Add a CRUD operations service

1. Add a *Services* directory to the project root.

2. Add a `BooksService` class to the *Services* directory with the following code:

C#

```
using BookStoreApi.Models;
using Microsoft.Extensions.Options;
using MongoDB.Driver;

namespace BookStoreApi.Services;

public class BooksService
{
    private readonly IMongoCollection<Book> _booksCollection;

    public BooksService(
        IOptions<BookStoreDatabaseSettings> bookStoreDatabaseSettings)
    {
        var mongoClient = new MongoClient(
            bookStoreDatabaseSettings.Value.ConnectionString);

        var mongoDatabase = mongoClient.GetDatabase(
            bookStoreDatabaseSettings.Value.DatabaseName);

        _booksCollection = mongoDatabase.GetCollection<Book>(
            bookStoreDatabaseSettings.Value.BooksCollectionName);
    }

    public async Task<List<Book>> GetAsync() =>
        await _booksCollection.Find(_ => true).ToListAsync();

    public async Task<Book?> GetAsync(string id) =>
        await _booksCollection.Find(x => x.Id ==
id).FirstOrDefaultAsync();

    public async Task CreateAsync(Book newBook) =>
        await _booksCollection.InsertOneAsync(newBook);

    public async Task UpdateAsync(string id, Book updatedBook) =>
        await _booksCollection.ReplaceOneAsync(x => x.Id == id,
updatedBook);

    public async Task RemoveAsync(string id) =>
        await _booksCollection.DeleteOneAsync(x => x.Id == id);
}
```

In the preceding code, a `BookStoreDatabaseSettings` instance is retrieved from DI via constructor injection. This technique provides access to the `appsettings.json` configuration values that were added in the Add a configuration model section.

3. Add the following highlighted code to `Program.cs`:

```C#
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.Configure<BookStoreDatabaseSettings>(
    builder.Configuration.GetSection("BookStoreDatabase"));

builder.Services.AddSingleton<BooksService>();
```

In the preceding code, the `BooksService` class is registered with DI to support constructor injection in consuming classes. The singleton service lifetime is most appropriate because `BooksService` takes a direct dependency on `MongoClient`. Per the official Mongo Client reuse guidelines, `MongoClient` should be registered in DI with a singleton service lifetime.

4. Add the following code to the top of `Program.cs` to resolve the `BooksService` reference:

```C#
using BookStoreApi.Services;
```

The `BooksService` class uses the following `MongoDB.Driver` members to run CRUD operations against the database:

- MongoClient: Reads the server instance for running database operations. The constructor of this class is provided in the MongoDB connection string:

```C#
public BooksService(
    IOptions<BookStoreDatabaseSettings> bookStoreDatabaseSettings)
{
    var mongoClient = new MongoClient(
        bookStoreDatabaseSettings.Value.ConnectionString);

    var mongoDatabase = mongoClient.GetDatabase(
        bookStoreDatabaseSettings.Value.DatabaseName);

    _booksCollection = mongoDatabase.GetCollection<Book>(
        bookStoreDatabaseSettings.Value.BooksCollectionName);
}
```

- IMongoDatabase: Represents the Mongo database for running operations. This tutorial uses the generic GetCollection<TDocument>(collection) method on the interface to gain access to data in a specific collection. Run CRUD operations against the collection after this method is called. In the `GetCollection<TDocument>(collection)` method call:
  - `collection` represents the collection name.
  - `TDocument` represents the CLR object type stored in the collection.

`GetCollection<TDocument>(collection)` returns a MongoCollection object representing the collection. In this tutorial, the following methods are invoked on the collection:

- DeleteOneAsync: Deletes a single document matching the provided search criteria.
- Find<TDocument>: Returns all documents in the collection matching the provided search criteria.
- InsertOneAsync: Inserts the provided object as a new document in the collection.
- ReplaceOneAsync: Replaces the single document matching the provided search criteria with the provided object.

Add a controller

Add a `BooksController` class to the *Controllers* directory with the following code:

C#

```
using BookStoreApi.Models;
using BookStoreApi.Services;
using Microsoft.AspNetCore.Mvc;

namespace BookStoreApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class BooksController : ControllerBase
{
    private readonly BooksService _booksService;

    public BooksController(BooksService booksService) =>
        _booksService = booksService;

    [HttpGet]
    public async Task<List<Book>> Get() =>
        await _booksService.GetAsync();

    [HttpGet("{id:length(24)}")]
```

```
    public async Task<ActionResult<Book>> Get(string id)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        return book;
    }

    [HttpPost]
    public async Task<IActionResult> Post(Book newBook)
    {
        await _booksService.CreateAsync(newBook);

        return CreatedAtAction(nameof(Get), new { id = newBook.Id },
newBook);
    }

    [HttpPut("{id:length(24)}")]
    public async Task<IActionResult> Update(string id, Book updatedBook)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        updatedBook.Id = book.Id;

        await _booksService.UpdateAsync(id, updatedBook);

        return NoContent();
    }

    [HttpDelete("{id:length(24)}")]
    public async Task<IActionResult> Delete(string id)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        await _booksService.RemoveAsync(id);

        return NoContent();
    }
}
```

The preceding web API controller:

- Uses the `BooksService` class to run CRUD operations.
- Contains action methods to support GET, POST, PUT, and DELETE HTTP requests.
- Calls CreatedAtAction in the `Create` action method to return an HTTP 201 response. Status code 201 is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtAction` also adds a `Location` header to the response. The `Location` header specifies the URI of the newly created book.

Test the web API

1. Build and run the app.

2. Navigate to `https://localhost:<port>/api/books`, where `<port>` is the automatically assigned port number for the app, to test the controller's parameterless `Get` action method, select **Try it out** > **Execute**. A JSON response similar to the following is displayed:

   JSON

   ```
   [
     {
       "id": "61a6058e6c43f32854e51f51",
       "bookName": "Design Patterns",
       "price": 54.93,
       "category": "Computers",
       "author": "Ralph Johnson"
     },
     {
       "id": "61a6058e6c43f32854e51f52",
       "bookName": "Clean Code",
       "price": 43.15,
       "category": "Computers",
       "author": "Robert C. Martin"
     }
   ]
   ```

3. Navigate to `https://localhost:<port>/api/books/{id here}` to test the controller's overloaded `Get` action method. A JSON response similar to the following is displayed:

   JSON

   ```
   {
     "id": "61a6058e6c43f32854e51f52",
   ```

```
  "bookName": "Clean Code",
  "price": 43.15,
  "category": "Computers",
  "author": "Robert C. Martin"
}
```

Configure JSON serialization options

There are two details to change about the JSON responses returned in the Test the web API section:

- The property names' default camel casing should be changed to match the Pascal casing of the CLR object's property names.
- The `bookName` property should be returned as `Name`.

To satisfy the preceding requirements, make the following changes:

1. In `Program.cs`, chain the following highlighted code on to the `AddControllers` method call:

   C#

   ```
   var builder = WebApplication.CreateBuilder(args);

   // Add services to the container.
   builder.Services.Configure<BookStoreDatabaseSettings>(
       builder.Configuration.GetSection("BookStoreDatabase"));

   builder.Services.AddSingleton<BooksService>();

   builder.Services.AddControllers()
       .AddJsonOptions(
           options => options.JsonSerializerOptions.PropertyNamingPolicy =
   null);
   ```

   With the preceding change, property names in the web API's serialized JSON response match their corresponding property names in the CLR object type. For example, the `Book` class's `Author` property serializes as `Author` instead of `author`.

2. In `Models/Book.cs`, annotate the `BookName` property with the [JsonPropertyName] attribute:

   C#

   ```
   [BsonElement("Name")]
   [JsonPropertyName("Name")]
   ```

```
public string BookName { get; set; } = null!;
```

The `[JsonPropertyName]` attribute's value of `Name` represents the property name in the web API's serialized JSON response.

3. Add the following code to the top of `Models/Book.cs` to resolve the `[JsonProperty]` attribute reference:

```
C#

using System.Text.Json.Serialization;
```

4. Repeat the steps defined in the Test the web API section. Notice the difference in JSON property names.

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Microsoft Entra ID
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

> **Important**
>
> Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see Migrate from ASP.NET Core 5.0 to 6.0.

For more information, see the Duende Identity Server documentation (Duende Software website).

Additional resources

- View or download sample code (how to download)
- Create web APIs with ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Create a web API with ASP.NET Core

Controller action return types in ASP.NET Core web API

Article • 01/22/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

View or download sample code (how to download)

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

- Specific type
- IActionResult
- ActionResult<T>
- HttpResults

This article explains when it's most appropriate to use each return type.

Specific type

The most basic action returns a primitive or complex data type, for example, `string` or a custom object. Consider the following action, which returns a collection of custom `Product` objects:

C#

```
[HttpGet]
public Task<List<Product>> Get() =>
    _productContext.Products.OrderBy(p => p.Name).ToListAsync();
```

Without known conditions to safeguard against, returning a specific type could suffice. The preceding action accepts no parameters, so parameter constraints validation isn't needed.

When multiple return types are possible, it's common to mix an ActionResult return type with the primitive or complex return type. Either IActionResult or ActionResult<T> are

necessary to accommodate this type of action. Several samples of multiple return types are provided in this article.

Return IEnumerable<T> or IAsyncEnumerable<T>

See Return IEnumerable<T> or IAsyncEnumerable<T> for performance considerations.

ASP.NET Core buffers the result of actions that return IEnumerable<T> before writing them to the response. Consider declaring the action signature's return type as IAsyncEnumerable<T> to guarantee asynchronous iteration. Ultimately, the iteration mode is based on the underlying concrete type being returned and the selected formatter affects how the result is processed:

- When using `System.Text.Json` formatter, MVC relies on the support that `System.Text.Json` added to **stream** the result.
- When using `Newtonsoft.Json` or with `XML-based` formatters the result is buffered.

Consider the following action, which returns sale-priced product records as `IEnumerable<Product>`:

C#

```
[HttpGet("syncsale")]
public IEnumerable<Product> GetOnSaleProducts()
{
    var products = _productContext.Products.OrderBy(p => p.Name).ToList();

    foreach (var product in products)
    {
        if (product.IsOnSale)
        {
            yield return product;
        }
    }
}
```

The `IAsyncEnumerable<Product>` equivalent of the preceding action is:

C#

```
[HttpGet("asyncsale")]
public async IAsyncEnumerable<Product> GetOnSaleProductsAsync()
{
    var products = _productContext.Products.OrderBy(p =>
p.Name).AsAsyncEnumerable();

    await foreach (var product in products)
```

```
    {
        if (product.IsOnSale)
        {
            yield return product;
        }
    }
}
```

IActionResult type

The 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 (400), NotFoundResult (404), and OkObjectResult (200). Alternatively, convenience methods in the ControllerBase class can be used to return `ActionResult` types from an action. For example, `return BadRequest();` is a shorthand form of `return new BadRequestResult();`.

Because there are multiple return types and paths in this type of action, liberal use of the [ProducesResponseType] attribute is necessary. This attribute produces more descriptive response details for web API help pages generated by tools like Swagger. `[ProducesResponseType]` indicates the known types and HTTP status codes to be returned by the action.

Synchronous action

Consider the following synchronous action in which there are two possible return types:

C#

```
[HttpGet("{id}")]
[ProducesResponseType<Product>(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public IActionResult GetById_IActionResult(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? NotFound() : Ok(product);
}
```

In the preceding action:

- A 404 status code is returned when the product represented by `id` doesn't exist in the underlying data store. The NotFound convenience method is invoked as shorthand for `return new NotFoundResult();`.

- A 200 status code is returned with the `Product` object when the product does exist. The Ok convenience method is invoked as shorthand for `return new OkObjectResult(product);`.

Asynchronous action

Consider the following asynchronous action in which there are two possible return types:

C#

```
[HttpPost()]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> CreateAsync_IActionResult(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();

    return CreatedAtAction(nameof(CreateAsync_IActionResult), new { id =
product.Id }, product);
}
```

In the preceding action:

- A 400 status code is returned when the product description contains "XYZ Widget". The BadRequest convenience method is invoked as shorthand for `return new BadRequestResult();`.

- A 201 status code is generated by the CreatedAtAction convenience method when a product is created. The following code is an alternative to calling `CreatedAtAction`:

  C#

  ```
  return new CreatedAtActionResult(nameof(CreateAsync),
                                   "Products",
                                   new { id = product.Id },
                                   product);
  ```

In the preceding code path, the `Product` object is provided in the response body. A `Location` response header containing the newly created product's URL is provided.

For example, the following model indicates that requests must include the `Name` and `Description` properties. Failure to provide `Name` and `Description` in the request causes model validation to fail.

C#

```
public class Product
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; } = string.Empty;

    [Required]
    public string Description { get; set; } = string.Empty;

    public bool IsOnSale { get; set; }
}
```

If the [ApiController] attribute is applied, model validation errors result in a 400 status code. For more information, see Automatic HTTP 400 responses.

ActionResult vs IActionResult

The following section compares `ActionResult` to `IActionResult`

ActionResult<T> type

ASP.NET Core includes the ActionResult<T> return type for web API controller actions. It enables returning a type deriving from ActionResult or return a specific type. `ActionResult<T>` offers the following benefits over the IActionResult type:

- The [ProducesResponseType] attribute's `Type` property can be excluded. For example, `[ProducesResponseType(200, Type = typeof(Product))]` is simplified to `[ProducesResponseType(200)]`. The action's expected return type is inferred from the `T` in `ActionResult<T>`.
- Implicit cast operators support the conversion of both `T` and `ActionResult` to `ActionResult<T>`. `T` converts to ObjectResult, which means `return new ObjectResult(T);` is simplified to `return T;`.

C# doesn't support implicit cast operators on interfaces. Consequently, conversion of the interface to a concrete type is necessary to use `ActionResult<T>`. For example, use of `IEnumerable` in the following example doesn't work:

```
C#
```

```
[HttpGet]
public ActionResult<IEnumerable<Product>> Get() =>
    _repository.GetProducts();
```

One option to fix the preceding code is to return `_repository.GetProducts().ToList();`.

Most actions have a specific return type. Unexpected conditions can occur during action execution, in which case the specific type isn't returned. For example, an action's input parameter may fail model validation. In such a case, it's common to return the appropriate `ActionResult` type instead of the specific type.

Synchronous action

Consider a synchronous action in which there are two possible return types:

```
C#
```

```
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public ActionResult<Product> GetById_ActionResultOfT(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? NotFound() : product;
}
```

In the preceding action:

- A 404 status code is returned when the product doesn't exist in the database.
- A 200 status code is returned with the corresponding `Product` object when the product does exist.

Asynchronous action

Consider an asynchronous action in which there are two possible return types:

```
C#
```

```
[HttpPost()]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<ActionResult<Product>> CreateAsync_ActionResultOfT(Product
product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();

    return CreatedAtAction(nameof(CreateAsync_ActionResultOfT), new { id =
product.Id }, product);
}
```

In the preceding action:

- A 400 status code (BadRequest) is returned by the ASP.NET Core runtime when:
  - The [ApiController] attribute has been applied and model validation fails.
  - The product description contains "XYZ Widget".
- A 201 status code is generated by the CreatedAtAction method when a product is created. In this code path, the `Product` object is provided in the response body. A `Location` response header containing the newly created product's URL is provided.

HttpResults type

In addition to the MVC-specific built-in result types (IActionResult and ActionResult<T>), ASP.NET Core includes the HttpResults types that can be used in both Minimal APIs and Web API.

Different than the MVC-specific result types, the `HttpResults`:

- Are a results implementation that is processed by a call to IResult.ExecuteAsync.
- Does ***not*** leverage the configured Formatters. Not leveraging the configured formatters means:
  - Some features like `Content negotiation` aren't available.
  - The produced `Content-Type` is decided by the `HttpResults` implementation.

The `HttpResults` can be useful when sharing code between Minimal APIs and Web API.

IResult type

The Microsoft.AspNetCore.Http.HttpResults namespace contains classes that implement the IResult interface. The `IResult` interface defines a contract that represents the result of an HTTP endpoint. The static Results class is used to create varying `IResult` objects that represent different types of responses.

The Built-in results table shows the common result helpers.

Consider the following code:

C#

```
[HttpGet("{id}")]
[ProducesResponseType<Product>(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public IResult GetById(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? Results.NotFound() : Results.Ok(product);
}
```

In the preceding action:

- A 404 status code is returned when the product doesn't exist in the database.
- A 200 status code is returned with the corresponding `Product` object when the product does exist, generated by the Results.Ok<T>().

Consider the following code:

C#

```
[HttpPost]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType<Product>(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IResult> CreateAsync(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return Results.BadRequest();
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();

    var location = Url.Action(nameof(CreateAsync), new { id = product.Id })
?? $"/{product.Id}";
```

```
    return Results.Created(location, product);
}
```

In the preceding action:

- A 400 status code is returned when:
  - The [ApiController] attribute has been applied and model validation fails.
  - The product description contains "XYZ Widget".
- A 201 status code is generated by the Results.Create method when a product is created. In this code path, the `Product` object is provided in the response body. A `Location` response header containing the newly created product's URL is provided.

Results<TResult1, TResultN> type

The static TypedResults class returns the concrete `IResult` implementation that allows using `IResult` as return type. The usage of the concrete `IResult` implementation offers the following benefit over the IResult type:

- All the [ProducesResponseType] attributes can be excluded, since the `HttpResult` implementation contributes automatically to the endpoint metadata.

When multiple `IResult` return types are needed, returning Results<TResult1, TResultN> is preferred over returning `IResult`. Returning `Results<TResult1, TResultN>` is preferred because generic union types automatically retain the endpoint metadata.

The `Results<TResult1, TResultN>` union types implement implicit cast operators so that the compiler can automatically convert the types specified in the generic arguments to an instance of the union type. This has the added benefit of providing compile-time checking that a route handler actually only returns the results that it declares it does. Attempting to return a type that isn't declared as one of the generic arguments to `Results<>` results in a compilation error.

Consider the following code:

C#

```
[HttpGet("{id}")]
public Results<NotFound, Ok<Product>> GetById(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? TypedResults.NotFound() :
TypedResults.Ok(product);
}
```

In the preceding action:

- A 404 status code is returned when the product doesn't exist in the database.
- A 200 status code is returned with the corresponding `Product` object when the product does exist, generated by the TypedResults.Ok<T>.

```
C#

[HttpPost]
public async Task<Results<BadRequest, Created<Product>>> CreateAsync(Product
product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return TypedResults.BadRequest();
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();

    var location = Url.Action(nameof(CreateAsync), new { id = product.Id })
?? $"/{product.Id}";
    return TypedResults.Created(location, product);
}
```

In the preceding action:

- A 400 status code is returned when:
  - The [ApiController] attribute was applied and model validation fails.
  - The product description contains "XYZ Widget".
- A 201 status code is generated by the TypedResults.Created method when a product is created. In this code path, the `Product` object is provided in the response body. A `Location` response header containing the newly created product's URL is provided.

Additional resources

- Handle requests with controllers in ASP.NET Core MVC
- Model validation in ASP.NET Core MVC
- ASP.NET Core web API documentation with Swagger / OpenAPI

JsonPatch in ASP.NET Core web API

Article • 03/08/2023

This article explains how to handle JSON Patch requests in an ASP.NET Core web API.

Package installation

JSON Patch support in ASP.NET Core web API is based on `Newtonsoft.Json` and requires the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package. To enable JSON Patch support:

- Install the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package.
- Call AddNewtonsoftJson. For example:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .AddNewtonsoftJson();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

`AddNewtonsoftJson` replaces the default `System.Text.Json`-based input and output formatters used for formatting ***all*** JSON content. This extension method is compatible with the following MVC service registration methods:

- AddRazorPages
- AddControllersWithViews
- AddControllers

JsonPatch requires setting the `Content-Type` header to `application/json-patch+json`.

Add support for JSON Patch when using System.Text.Json

The `System.Text.Json`-based input formatter doesn't support JSON Patch. To add support for JSON Patch using `Newtonsoft.Json`, while leaving the other input and output formatters unchanged:

- Install the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package.
- Update `Program.cs`:

C#

```
using JsonPatchSample;
using Microsoft.AspNetCore.Mvc.Formatters;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.InputFormatters.Insert(0,
MyJPIF.GetJsonPatchInputFormatter());
});

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Formatters;
using Microsoft.Extensions.Options;

namespace JsonPatchSample;

public static class MyJPIF
{
    public static NewtonsoftJsonPatchInputFormatter
GetJsonPatchInputFormatter()
    {
        var builder = new ServiceCollection()
            .AddLogging()
```

```
                .AddMvc()
                .AddNewtonsoftJson()
                .Services.BuildServiceProvider();

            return builder
                .GetRequiredService<IOptions<MvcOptions>>()
                .Value
                .InputFormatters
                .OfType<NewtonsoftJsonPatchInputFormatter>()
                .First();
        }
    }
```

The preceding code creates an instance of NewtonsoftJsonPatchInputFormatter and inserts it as the first entry in the MvcOptions.InputFormatters collection. This order of registration ensures that:

- `NewtonsoftJsonPatchInputFormatter` processes JSON Patch requests.
- The existing `System.Text.Json`-based input and formatters process all other JSON requests and responses.

Use the `Newtonsoft.Json.JsonConvert.SerializeObject` method to serialize a JsonPatchDocument.

PATCH HTTP request method

The PUT and PATCH 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 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

```
JSON
```

```
{
  "customerName": "John",
  "orders": [
    {
      "orderName": "Order0",
      "orderType": null
    },
    {
      "orderName": "Order1",
      "orderType": null
    }
  ]
}
```

JSON patch example

JSON

```
[
  {
    "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:

JSON

```
{
  "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.

Path syntax

The path property of an operation object has slashes between levels. For example, `"/address/zipCode"`.

Zero-based indexes are used to specify array elements. The first element of the `addresses` array would be at `/addresses/0`. To `add` to the end of an array, use a hyphen (`-`) rather than an index number: `/addresses/-`.

Operations

The following table shows supported operations as defined in the JSON Patch specification:

Expand table

Operation	Notes
`add`	Add a property or array element. For existing property: set value.
`remove`	Remove a property or array element.
`replace`	Same as `remove` followed by `add` at same location.

Operation	Notes
`move`	Same as `remove` from source followed by `add` to destination using value from source.
`copy`	Same as `add` to destination using value from source.
`test`	Return success status code if value at `path` = provided `value`.

JSON Patch in ASP.NET Core

The ASP.NET Core implementation of JSON Patch is provided in the 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>, typically with [FromBody].
- Calls ApplyTo(Object) on the patch document to apply the changes.

Here's an example:

C#

```
[HttpPatch]
public IActionResult JsonPatchWithModelState(
    [FromBody] JsonPatchDocument<Customer> patchDoc)
{
    if (patchDoc != null)
    {
        var customer = CreateCustomer();

        patchDoc.ApplyTo(customer, ModelState);

        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        return new ObjectResult(customer);
    }
    else
    {
        return BadRequest(ModelState);
```

```
    }
}
```

This code from the sample app works with the following `Customer` model:

C#

```
namespace JsonPatchSample.Models;

public class Customer
{
    public string? CustomerName { get; set; }
    public List<Order>? Orders { get; set; }
}
```

C#

```
namespace JsonPatchSample.Models;

public class Order
{
    public string OrderName { get; set; }
    public string OrderType { get; set; }
}
```

The sample action method:

- Constructs a `Customer`.
- Applies the patch.
- Returns the result in the body of the response.

In a real app, the code would retrieve the data from a store such as a database and update the database after applying the patch.

Model state

The preceding action method example calls an overload of `ApplyTo` that takes model state as one of its parameters. With this option, you can get error messages in responses. The following example shows the body of a 400 Bad Request response for a `test` operation:

JSON

```
{
  "Customer": [
    "The current value 'John' at path 'customerName' != test value 'Nancy'."
```

```
  ]
}
```

Dynamic objects

The following action method example shows how to apply a patch to a dynamic object:

C#

```
[HttpPatch]
public IActionResult JsonPatchForDynamic([FromBody]JsonPatchDocument patch)
{
    dynamic obj = new ExpandoObject();
    patch.ApplyTo(obj);

    return Ok(obj);
}
```

The add operation

- If `path` points to an array element: inserts new element before the one specified by `path`.
- If `path` points to a property: sets the property value.
- If `path` points to a nonexistent location:
  - If the resource to patch is a dynamic object: adds a property.
  - If the resource to patch is a static object: the request fails.

The following sample patch document sets the value of `CustomerName` and adds an `Order` object to the end of the `Orders` array.

JSON

```
[
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "add",
    "path": "/orders/-",
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
```

```
  }
]
```

The remove operation

- If `path` points to an array element: removes the element.
- If `path` points to a property:
  - If resource to patch is a dynamic object: removes the property.
  - If resource to patch is a static object:
    - If the property is nullable: sets it to null.
    - If the property is non-nullable, sets it to `default<T>`.

The following sample patch document sets `CustomerName` to null and deletes `Orders[0]`:

JSON

```
[
  {
    "op": "remove",
    "path": "/customerName"
  },
  {
    "op": "remove",
    "path": "/orders/0"
  }
]
```

The replace operation

This operation is functionally the same as a `remove` followed by an `add`.

The following sample patch document sets the value of `CustomerName` and replaces `Orders[0]` with a new `Order` object:

JSON

```
[
  {
    "op": "replace",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "replace",
    "path": "/orders/0",
```

```
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
  }
]
```

The move operation

- If `path` points to an array element: copies `from` element to location of `path` element, then runs a `remove` operation on the `from` element.
- If `path` points to a property: copies value of `from` property to `path` property, then runs a `remove` operation on the `from` property.
- If `path` points to a nonexistent property:
  - If the resource to patch is a static object: the request fails.
  - If the resource to patch is a dynamic object: copies `from` property to location indicated by `path`, then runs a `remove` operation on the `from` property.

The following sample patch document:

- Copies the value of `Orders[0].OrderName` to `CustomerName`.
- Sets `Orders[0].OrderName` to null.
- Moves `Orders[1]` to before `Orders[0]`.

JSON

```
[
  {
    "op": "move",
    "from": "/orders/0/orderName",
    "path": "/customerName"
  },
  {
    "op": "move",
    "from": "/orders/1",
    "path": "/orders/0"
  }
]
```

The copy operation

This operation is functionally the same as a `move` operation without the final `remove` step.

The following sample patch document:

- Copies the value of `Orders[0].OrderName` to `CustomerName`.
- Inserts a copy of `Orders[1]` before `Orders[0]`.

JSON

```
[
  {
    "op": "copy",
    "from": "/orders/0/orderName",
    "path": "/customerName"
  },
  {
    "op": "copy",
    "from": "/orders/1",
    "path": "/orders/0"
  }
]
```

The test operation

If the value at the location indicated by `path` is different from the value provided in `value`, the request fails. In that case, the whole PATCH request fails even if all other operations in the patch document would otherwise succeed.

The `test` operation is commonly used to prevent an update when there's a concurrency conflict.

The following sample patch document has no effect if the initial value of `CustomerName` is "John", because the test fails:

JSON

```
[
  {
    "op": "test",
    "path": "/customerName",
    "value": "Nancy"
  },
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  }
]
```

Get the code

View or download sample code. (How to download).

To test the sample, run the app and send HTTP requests with the following settings:

- URL: `http://localhost:{port}/jsonpatch/jsonpatchwithmodelstate`
- HTTP method: `PATCH`
- Header: `Content-Type: application/json-patch+json`
- Body: Copy and paste one of the JSON patch document samples from the *JSON* project folder.

Additional resources

- IETF RFC 5789 PATCH method specification
- IETF RFC 6902 JSON Patch specification
- IETF RFC 6901 JSON Pointer
- JSON Patch documentation. Includes links to resources for creating JSON Patch documents.
- ASP.NET Core JSON Patch source code

Format response data in ASP.NET Core Web API

Article • 07/16/2024

ASP.NET Core MVC supports formatting response data, using specified formats or in response to a client's request.

Format-specific Action Results

Some action result types are specific to a particular format, such as JsonResult and 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.

An action isn't required to return any specific type. ASP.NET Core supports any object return value. Results from actions that return objects that aren't IActionResult types are serialized using the appropriate IOutputFormatter implementation. For more information, see Controller action return types in ASP.NET Core web API.

By default, the built-in helper method ControllerBase.Ok returns JSON-formatted data:

C#

```
[HttpGet]
public IActionResult Get()
    => Ok(_todoItemStore.GetList());
```

The sample code returns a list of todo items. Using the F12 browser developer tools or http-repl with the previous code displays:

- The response header containing **content-type:** `application/json; charset=utf-8`.
- The request headers. For example, the `Accept` header. The `Accept` header is ignored by the preceding code.

To return plain text formatted data, use ContentResult and the Content helper:

C#

```
[HttpGet("Version")]
public ContentResult GetVersion()
    => Content("v1.0.0");
```

In the preceding code, the `Content-Type` returned is `text/plain`.

For actions with multiple return types, return `IActionResult`. For example, when returning different HTTP status codes based on the result of the operation.

Content negotiation

Content negotiation occurs when the client specifies an Accept header. The default format used by ASP.NET Core is JSON. Content negotiation is:

- Implemented by 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`.

The following action method uses the `Ok` and `NotFound` helper methods:

C#

```
[HttpGet("{id:long}")]
public IActionResult GetById(long id)
{
    var todo = _todoItemStore.GetById(id);

    if (todo is null)
    {
        return NotFound();
    }

    return Ok(todo);
}
```

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

- `application/json`
- `text/json`
- `text/plain`

Tools such as Fiddler or http-repl can set the `Accept` request header to specify the return format. When the `Accept` header contains a type the server supports, that type is returned. The next section shows how to add additional formatters.

Controller actions can return POCOs (Plain Old CLR Objects). When a POCO is returned, the runtime automatically creates an `ObjectResult` that wraps the object. The client gets

the formatted serialized object. If the object being returned is `null`, a `204 No Content` response is returned.

The following example returns an object type:

```
C#

[HttpGet("{id:long}")]
public TodoItem? GetById(long id)
    => _todoItemStore.GetById(id);
```

In the preceding code, a request for a valid todo item returns a `200 OK` response. A request for an invalid todo item returns a `204 No Content` response.

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 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.

If the Accept header contains `*/*`, the Header is ignored unless `RespectBrowserAcceptHeader` is set to true on MvcOptions.

Browsers and content negotiation

Unlike typical API clients, web browsers supply `Accept` headers. Web browsers specify many formats, including wildcards. By default, when the framework detects that the

request is coming from a browser:

- The `Accept` header is ignored.
- The content is returned in JSON, unless otherwise configured.

This approach provides a more consistent experience across browsers when consuming APIs.

To configure an app to respect browser accept headers, set the RespectBrowserAcceptHeader property to `true`:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.RespectBrowserAcceptHeader = true;
});
```

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. Output formatters are used to format responses. For information on creating a custom formatter, see Custom Formatters.

Add XML format support

To configure XML formatters implemented using XmlSerializer, call AddXmlSerializerFormatters:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .AddXmlSerializerFormatters();
```

When using the preceding code, controller methods return the appropriate format based on the request's `Accept` header.

Configure `System.Text.Json`-based formatters

To configure features for the `System.Text.Json`-based formatters, use Microsoft.AspNetCore.Mvc.JsonOptions.JsonSerializerOptions. The following highlighted code configures PascalCase formatting instead of the default camelCase formatting:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.PropertyNamingPolicy = null;
    });
```

To configure output serialization options for specific actions, use JsonResult. For example:

C#

```
[HttpGet]
public IActionResult Get()
    => new JsonResult(
        _todoItemStore.GetList(),
        new JsonSerializerOptions { PropertyNamingPolicy = null });
```

Add `Newtonsoft.Json`-based JSON format support

The default JSON formatters use `System.Text.Json`. To use the `Newtonsoft.Json`-based formatters, install the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package and configure it in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .AddNewtonsoftJson();
```

In the preceding code, the call to `AddNewtonsoftJson` configures the following Web API, MVC, and Razor Pages features to use `Newtonsoft.Json`:

- Input and output formatters that read and write JSON
- JsonResult
- JSON Patch
- IJsonHelper

- TempData

Some features may not work well with `System.Text.Json`-based formatters and require a reference to the `Newtonsoft.Json`-based formatters. Continue using the `Newtonsoft.Json`-based formatters when the app:

- Uses `Newtonsoft.Json` attributes. For example, `[JsonProperty]` or `[JsonIgnore]`.
- Customizes the serialization settings.
- Relies on features that `Newtonsoft.Json` provides.

To configure features for the `Newtonsoft.Json`-based formatters, use SerializerSettings:

C#

```
builder.Services.AddControllers()
    .AddNewtonsoftJson(options =>
    {
        options.SerializerSettings.ContractResolver = new
DefaultContractResolver();
    });
```

To configure output serialization options for specific actions, use `JsonResult`. For example:

C#

```
[HttpGet]
public IActionResult GetNewtonsoftJson()
    => new JsonResult(
        _todoItemStore.GetList(),
        new JsonSerializerSettings { ContractResolver = new
DefaultContractResolver() });
```

Format `ProblemDetails` and `ValidationProblemDetails` responses

The following action method calls ControllerBase.Problem to create a ProblemDetails response:

C#

```
[HttpGet("Error")]
public IActionResult GetError()
    => Problem("Something went wrong.");
```

A `ProblemDetails` response is always camelCase, even when the app sets the format to PascalCase. `ProblemDetails` follows RFC 7807, which specifies lowercase.

When the [ApiController] attribute is applied to a controller class, the controller creates a ValidationProblemDetails response when Model Validation fails. This response includes a dictionary that uses the model's property names as error keys, unchanged. For example, the following model includes a single property that requires validation:

C#

```
public class SampleModel
{
    [Range(1, 10)]
    public int Value { get; set; }
}
```

By default, the `ValidationProblemDetails` response returned when the `Value` property is invalid uses an error key of `Value`, as shown in the following example:

C#

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "00-00000000000000000000000000000000-0000000000000000-00",
  "errors": {
    "Value": [
      "The field Value must be between 1 and 10."
    ]
  }
}
```

To format the property names used as error keys, add an implementation of IMetadataDetailsProvider to the MvcOptions.ModelMetadataDetailsProviders collection. The following example adds a `System.Text.Json`-based implementation, `SystemTextJsonValidationMetadataProvider`, which formats property names as camelCase by default:

C#

```
builder.Services.AddControllers();

builder.Services.Configure<MvcOptions>(options =>
{
    options.ModelMetadataDetailsProviders.Add(
```

```
        new SystemTextJsonValidationMetadataProvider());
});
```

`SystemTextJsonValidationMetadataProvider` also accepts an implementation of JsonNamingPolicy in its constructor, which specifies a custom naming policy for formatting property names.

To set a custom name for a property within a model, use the [JsonPropertyName] attribute on the property:

C#

```
public class SampleModel
{
    [Range(1, 10)]
    [JsonPropertyName("sampleValue")]
    public int Value { get; set; }
}
```

The `ValidationProblemDetails` response returned for the preceding model when the `Value` property is invalid uses an error key of `sampleValue`, as shown in the following example:

C#

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "00-00000000000000000000000000000000-0000000000000000-00",
  "errors": {
    "sampleValue": [
      "The field Value must be between 1 and 10."
    ]
  }
}
```

To format the `ValidationProblemDetails` response using `Newtonsoft.Json`, use `NewtonsoftJsonValidationMetadataProvider`:

C#

```
builder.Services.AddControllers()
    .AddNewtonsoftJson();

builder.Services.Configure<MvcOptions>(options =>
{
```

```
    options.ModelMetadataDetailsProviders.Add(
        new NewtonsoftJsonValidationMetadataProvider());
});
```

By default, `NewtonsoftJsonValidationMetadataProvider` formats property names as camelCase. `NewtonsoftJsonValidationMetadataProvider` also accepts an implementation of `NamingPolicy` in its constructor, which specifies a custom naming policy for formatting property names. To set a custom name for a property within a model, use the `[JsonProperty]` attribute.

Specify a format

To restrict the response formats, apply the [Produces] filter. Like most Filters, `[Produces]` can be applied at the action, controller, or global scope:

C#

```
[ApiController]
[Route("api/[controller]")]
[Produces("application/json")]
public class TodoItemsController : ControllerBase
```

The preceding [Produces] filter:

- Forces all actions within the controller to return JSON-formatted responses for POCOs (Plain Old CLR Objects) or ObjectResult and its derived types.
- Return JSON-formatted responses even if other formatters are configured and the client specifies a different format.

For more information, see Filters.

Special case formatters

Some special cases are implemented using built-in formatters. By default, `string` return types are formatted as *text/plain* (*text/html* if requested via the `Accept` header). This behavior can be deleted by removing the StringOutputFormatter. Formatters are removed in `Program.cs`. Actions that have a model object return type return `204 No Content` when returning `null`. This behavior can be deleted by removing the HttpNoContentOutputFormatter. The following code removes the `StringOutputFormatter` and `HttpNoContentOutputFormatter`.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    // using Microsoft.AspNetCore.Mvc.Formatters;
    options.OutputFormatters.RemoveType<StringOutputFormatter>();
    options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();
});
```

Without the `StringOutputFormatter`, the built-in JSON formatter formats `string` return types. If the built-in JSON formatter is removed and an XML formatter is available, the XML formatter formats `string` return types. Otherwise, `string` return types return `406 Not Acceptable`.

Without the `HttpNoContentOutputFormatter`, null objects are formatted using the configured formatter. For example:

- The JSON formatter returns a response with a body of `null`.
- The XML formatter returns an empty XML element with the attribute `xsi:nil="true"` set.

Response format URL mappings

Clients can request a particular format as part of the URL, for example:

- In the query string or part of the path.
- By using a format-specific file extension such as .xml or .json.

The mapping from request path should be specified in the route the API is using. For example:

C#

```
[ApiController]
[Route("api/[controller]")]
[FormatFilter]
public class TodoItemsController : ControllerBase
{
    private readonly TodoItemStore _todoItemStore;

    public TodoItemsController(TodoItemStore todoItemStore)
        => _todoItemStore = todoItemStore;

    [HttpGet("{id:long}.{format?}")]
```

```
    public TodoItem? GetById(long id)
        => _todoItemStore.GetById(id);
```

The preceding route allows the requested format to be specified using an optional file extension. The [FormatFilter] attribute checks for the existence of the format value in the `RouteData` and maps the response format to the appropriate formatter when the response is created.

Expand table

Route	Formatter
`/api/todoitems/5`	The default output formatter
`/api/todoitems/5.json`	The JSON formatter (if configured)
`/api/todoitems/5.xml`	The XML formatter (if configured)

Polymorphic deserialization

Built-in features provide a limited range of polymorphic serialization but no support for deserialization at all. Deserialization requires a custom converter. See Polymorphic deserialization for a complete sample of polymorphic deserialization.

Additional resources

- View or download sample code (how to download)

Custom formatters in ASP.NET Core Web API

Article • 07/16/2024

ASP.NET Core MVC supports data exchange in Web APIs using input and output formatters. Input formatters are used by Model Binding. Output formatters are used to format responses.

The framework provides built-in input and output formatters for JSON and XML. It provides a built-in output formatter for plain text, but doesn't provide an input formatter for plain text.

This article shows how to add support for additional formats by creating custom formatters. For an example of a custom plain text input formatter, see TextPlainInputFormatter on GitHub.

View or download sample code (how to download)

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 and OutputFormatters collections in MvcOptions.

Create a custom formatter

To create a formatter:

- Derive the class from the appropriate base class. The sample app derives from TextOutputFormatter and TextInputFormatter.
- Specify supported media types and encodings in the constructor.
- Override the CanReadType and CanWriteType methods.

- Override the ReadRequestBodyAsync and WriteResponseBodyAsync methods.

The following code shows the `VcardOutputFormatter` class from the sample:

C#

```
public class VcardOutputFormatter : TextOutputFormatter
{
    public VcardOutputFormatter()
    {
        SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

        SupportedEncodings.Add(Encoding.UTF8);
        SupportedEncodings.Add(Encoding.Unicode);
    }

    protected override bool CanWriteType(Type? type)
        => typeof(Contact).IsAssignableFrom(type)
            || typeof(IEnumerable<Contact>).IsAssignableFrom(type);

    public override async Task WriteResponseBodyAsync(
        OutputFormatterWriteContext context, Encoding selectedEncoding)
    {
        var httpContext = context.HttpContext;
        var serviceProvider = httpContext.RequestServices;

        var logger =
serviceProvider.GetRequiredService<ILogger<VcardOutputFormatter>>();
        var buffer = new StringBuilder();

        if (context.Object is IEnumerable<Contact> contacts)
        {
            foreach (var contact in contacts)
            {
                FormatVcard(buffer, contact, logger);
            }
        }
        else
        {
            FormatVcard(buffer, (Contact)context.Object!, logger);
        }

        await httpContext.Response.WriteAsync(buffer.ToString(),
selectedEncoding);
    }

    private static void FormatVcard(
        StringBuilder buffer, Contact contact, ILogger logger)
    {
        buffer.AppendLine("BEGIN:VCARD");
        buffer.AppendLine("VERSION:2.1");
        buffer.AppendLine($"N:{contact.LastName};{contact.FirstName}");
        buffer.AppendLine($"FN:{contact.FirstName} {contact.LastName}");
        buffer.AppendLine($"UID:{contact.Id}");
```

```
            buffer.AppendLine("END:VCARD");

            logger.LogInformation("Writing {FirstName} {LastName}",
                contact.FirstName, contact.LastName);
        }
    }
```

Derive from the appropriate base class

For text media types (for example, vCard), derive from the TextInputFormatter or TextOutputFormatter base class:

C#

```
public class VcardOutputFormatter : TextOutputFormatter
```

For binary types, derive from the InputFormatter or OutputFormatter base class.

Specify supported media types and encodings

In the constructor, specify supported media types and encodings by adding to the SupportedMediaTypes and SupportedEncodings collections:

C#

```
public VcardOutputFormatter()
{
    SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

    SupportedEncodings.Add(Encoding.UTF8);
    SupportedEncodings.Add(Encoding.Unicode);
}
```

A formatter class can **not** use constructor injection for its dependencies. For example, `ILogger<VcardOutputFormatter>` can't be added as a parameter to the constructor. To access services, use the context object that gets passed in to the methods. A code example in this article and the sample show how to do this.

Override CanReadType and CanWriteType

Specify the type to deserialize into or serialize from by overriding the CanReadType or CanWriteType methods. For example, to create vCard text from a `Contact` type and vice versa:

C#

```
protected override bool CanWriteType(Type? type)
    => typeof(Contact).IsAssignableFrom(type)
        || typeof(IEnumerable<Contact>).IsAssignableFrom(type);
```

The CanWriteResult method

In some scenarios, CanWriteResult must be overridden rather than CanWriteType. Use `CanWriteResult` if the following conditions are true:

- The action method returns a model class.
- There are derived classes that might be returned at runtime.
- The derived class returned by the action must be known at runtime.

For example, suppose the action method:

- Signature returns a `Person` type.
- Can return a `Student` or `Instructor` type that derives from `Person`.

For the formatter to handle only `Student` objects, check the type of Object in the context object provided to the `CanWriteResult` method. When the action method returns IActionResult:

- It's not necessary to use `CanWriteResult`.
- The `CanWriteType` method receives the runtime type.

Override ReadRequestBodyAsync and WriteResponseBodyAsync

Deserialization or serialization is performed in ReadRequestBodyAsync or WriteResponseBodyAsync. The following example shows how to get services from the dependency injection container. Services can't be obtained from constructor parameters:

C#

```
public override async Task WriteResponseBodyAsync(
    OutputFormatterWriteContext context, Encoding selectedEncoding)
{
    var httpContext = context.HttpContext;
    var serviceProvider = httpContext.RequestServices;

    var logger =
```

```
serviceProvider.GetRequiredService<ILogger<VcardOutputFormatter>>();
    var buffer = new StringBuilder();

    if (context.Object is IEnumerable<Contact> contacts)
    {
        foreach (var contact in contacts)
        {
            FormatVcard(buffer, contact, logger);
        }
    }
    else
    {
        FormatVcard(buffer, (Contact)context.Object!, logger);
    }

    await httpContext.Response.WriteAsync(buffer.ToString(),
selectedEncoding);
}

private static void FormatVcard(
    StringBuilder buffer, Contact contact, ILogger logger)
{
    buffer.AppendLine("BEGIN:VCARD");
    buffer.AppendLine("VERSION:2.1");
    buffer.AppendLine($"N:{contact.LastName};{contact.FirstName}");
    buffer.AppendLine($"FN:{contact.FirstName} {contact.LastName}");
    buffer.AppendLine($"UID:{contact.Id}");
    buffer.AppendLine("END:VCARD");

    logger.LogInformation("Writing {FirstName} {LastName}",
        contact.FirstName, contact.LastName);
}
```

Configure MVC to use a custom formatter

To use a custom formatter, add an instance of the formatter class to the MvcOptions.InputFormatters or MvcOptions.OutputFormatters collection:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.InputFormatters.Insert(0, new VcardInputFormatter());
    options.OutputFormatters.Insert(0, new VcardOutputFormatter());
});
```

Formatters are evaluated in the order they're inserted, where the first one takes precedence.

The complete `VcardInputFormatter` class

The following code shows the `VcardInputFormatter` class from the sample:

C#

```
public class VcardInputFormatter : TextInputFormatter
{
    public VcardInputFormatter()
    {
        SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

        SupportedEncodings.Add(Encoding.UTF8);
        SupportedEncodings.Add(Encoding.Unicode);
    }

    protected override bool CanReadType(Type type)
        => type == typeof(Contact);

    public override async Task<InputFormatterResult> ReadRequestBodyAsync(
        InputFormatterContext context, Encoding effectiveEncoding)
    {
        var httpContext = context.HttpContext;
        var serviceProvider = httpContext.RequestServices;

        var logger =
serviceProvider.GetRequiredService<ILogger<VcardInputFormatter>>();

        using var reader = new StreamReader(httpContext.Request.Body,
effectiveEncoding);
        string? nameLine = null;

        try
        {
            await ReadLineAsync("BEGIN:VCARD", reader, context, logger);
            await ReadLineAsync("VERSION:", reader, context, logger);

            nameLine = await ReadLineAsync("N:", reader, context, logger);

            var split = nameLine.Split(";".ToCharArray());
            var contact = new Contact(FirstName: split[1], LastName:
split[0].Substring(2));

            await ReadLineAsync("FN:", reader, context, logger);
            await ReadLineAsync("END:VCARD", reader, context, logger);

            logger.LogInformation("nameLine = {nameLine}", nameLine);

            return await InputFormatterResult.SuccessAsync(contact);
        }
        catch
        {
            logger.LogError("Read failed: nameLine = {nameLine}", nameLine);
```

```
            return await InputFormatterResult.FailureAsync();
        }
    }

    private static async Task<string> ReadLineAsync(
        string expectedText, StreamReader reader, InputFormatterContext
context,
        ILogger logger)
    {
        var line = await reader.ReadLineAsync();

        if (line is null || !line.StartsWith(expectedText))
        {
            var errorMessage = $"Looked for '{expectedText}' and got
'{line}'";

            context.ModelState.TryAddModelError(context.ModelName,
errorMessage);
            logger.LogError(errorMessage);

            throw new Exception(errorMessage);
        }

        return line;
    }
}
```

Test the app

Run the sample app for this article, which implements basic vCard input and output formatters. The app reads and writes vCards similar to the following format:

```
BEGIN:VCARD
VERSION:2.1
N:Davolio;Nancy
FN:Nancy Davolio
END:VCARD
```

To see vCard output, run the app and send a Get request with Accept header `text/vcard` to `https://localhost:<port>/api/contacts`.

To add a vCard to the in-memory collection of contacts:

- Send a `Post` request to `/api/contacts` with a tool like http-repl.
- Set the `Content-Type` header to `text/vcard`.
- Set `vCard` text in the body, formatted like the preceding example.

Additional resources

- Format response data in ASP.NET Core Web API
- Manage Protobuf references with dotnet-grpc

Use web API analyzers

Article • 04/10/2024

ASP.NET Core provides an MVC analyzers package intended for use with web API projects. The analyzers work with controllers annotated with ApiControllerAttribute, while building on web API conventions.

The analyzers package notifies you of any controller action that:

- Returns an undeclared status code.
- Returns an undeclared success result.
- Documents a status code that isn't returned.
- Includes an explicit model validation check.

Reference the analyzer package

The analyzers are included in the .NET Core SDK. To enable the analyzer in your project, include the `IncludeOpenAPIAnalyzers` property in the project file:

XML

```xml
<PropertyGroup>
 <IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>
</PropertyGroup>
```

Analyzers for web API conventions

OpenAPI documents contain status codes and response types that an action may return. In ASP.NET Core MVC, attributes such as ProducesResponseTypeAttribute and ProducesAttribute are used to document an action. ASP.NET Core web API documentation with Swagger / OpenAPI goes into further detail on documenting your web API.

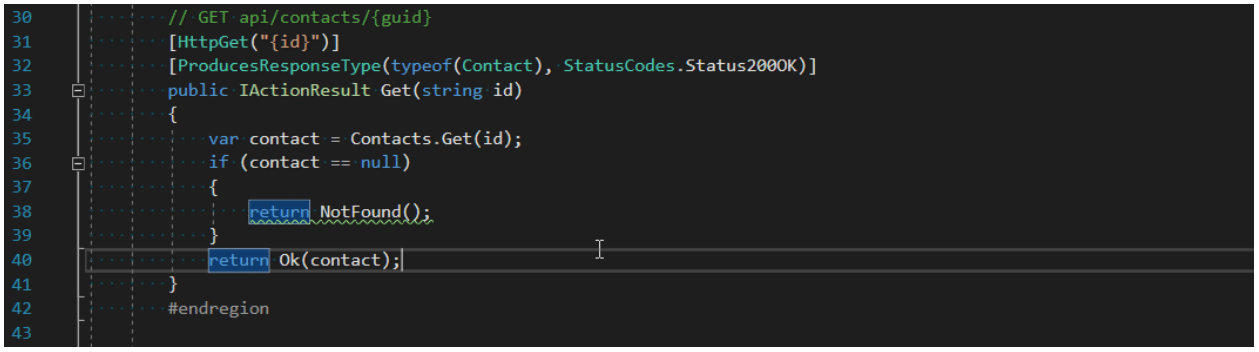
One of the analyzers in the package inspects controllers annotated with ApiControllerAttribute and identifies actions that don't entirely document their responses. Consider the following example:

C#

```csharp
// GET api/contacts/{guid}
[HttpGet("{id}", Name = "GetById")]
[ProducesResponseType(typeof(Contact), StatusCodes.Status200OK)]
```

```
public IActionResult Get(string id)
{
    var contact = _contacts.Get(id);

    if (contact == null)
    {
        return NotFound();
    }

    return Ok(contact);
}
```

The preceding action documents the HTTP 200 success return type but doesn't document the HTTP 404 failure status code. The analyzer reports the missing documentation for the HTTP 404 status code as a warning. An option to fix the problem is provided.



Analyzers require Microsoft.NET.Sdk.Web

Analyzers don't work with library projects or projects referencing `Sdk="Microsoft.NET.Sdk"`.

Additional resources

- Use web API conventions
- ASP.NET Core web API documentation with Swagger / OpenAPI
- Create web APIs with ASP.NET Core

Use web API conventions

Article • 04/10/2024

Common API documentation can be extracted and applied to multiple actions, controllers, or all controllers within an assembly. Web API conventions are a substitute for decorating individual actions with [ProducesResponseType].

A convention allows you to:

- Define the most common return types and status codes returned from a specific type of action.
- Identify actions that deviate from the defined standard.

Default conventions are available from Microsoft.AspNetCore.Mvc.DefaultApiConventions. The conventions are demonstrated with the `ValuesController.cs` added to an **API** project template:

C#

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace WebApp1.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ValuesController : ControllerBase
    {
        // GET api/values
        [HttpGet]
        public ActionResult<IEnumerable<string>> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/values/5
        [HttpGet("{id}")]
        public ActionResult<string> Get(int id)
        {
            return "value";
        }

        // POST api/values
        [HttpPost]
        public void Post([FromBody] string value)
        {
        }

        // PUT api/values/5
```

```
        [HttpPut("{id}")]
        public void Put(int id, [FromBody] string value)
        {
        }

        // DELETE api/values/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
}
```

Actions that follow the patterns in the `ValuesController.cs` work with the default conventions. If the default conventions don't meet your needs, see Create web API conventions.

At runtime, Microsoft.AspNetCore.Mvc.ApiExplorer understands conventions. `ApiExplorer` is MVC's abstraction to communicate with OpenAPI (also known as Swagger) document generators. Attributes from the applied convention are associated with an action and are included in the action's OpenAPI documentation. API analyzers also understand conventions. If your action is unconventional (for example, it returns a status code that isn't documented by the applied convention), a warning encourages you to document the status code.

View or download sample code (how to download)

Apply web API conventions

Conventions don't compose; each action may be associated with exactly one convention. More specific conventions take precedence over less specific conventions. The selection is non-deterministic when two or more conventions of the same priority apply to an action. The following options exist to apply a convention to an action, from the most specific to the least specific:

1. `Microsoft.AspNetCore.Mvc.ApiConventionMethodAttribute` — Applies to individual actions and specifies the convention type and the convention method that applies.

   In the following example, the default convention type's `Microsoft.AspNetCore.Mvc.DefaultApiConventions.Put` convention method is applied to the `Update` action:

   ```
   C#
   ```

```
// PUT api/contactsconvention/{guid}
[HttpPut("{id}")]
[ApiConventionMethod(typeof(DefaultApiConventions),
                     nameof(DefaultApiConventions.Put))]
public IActionResult Update(string id, Contact contact)
{
    var contactToUpdate = _contacts.Get(id);

    if (contactToUpdate == null)
    {
        return NotFound();
    }

    _contacts.Update(contact);

    return NoContent();
}
```

The `Microsoft.AspNetCore.Mvc.DefaultApiConventions.Put` convention method applies the following attributes to the action:

C#

```
[ProducesDefaultResponseType]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
```

For more information on `[ProducesDefaultResponseType]`, see Default Response.

2. `Microsoft.AspNetCore.Mvc.ApiConventionTypeAttribute` applied to a controller — Applies the specified convention type to all actions on the controller. A convention method is marked with hints that determine the actions to which the convention method applies. For more information on hints, see Create web API conventions).

   In the following example, the default set of conventions is applied to all actions in *ContactsConventionController*:

   C#

   ```
   [ApiController]
   [ApiConventionType(typeof(DefaultApiConventions))]
   [Route("api/[controller]")]
   public class ContactsConventionController : ControllerBase
   {
   ```

3. `Microsoft.AspNetCore.Mvc.ApiConventionTypeAttribute` applied to an assembly — Applies the specified convention type to all controllers in the current assembly. As a recommendation, apply assembly-level attributes in the `Startup.cs` file.

   In the following example, the default set of conventions is applied to all controllers in the assembly:

   ```
   C#

   [assembly: ApiConventionType(typeof(DefaultApiConventions))]
   namespace ApiConventions
   {
       public class Startup
       {
   ```

Create web API conventions

If the default API conventions don't meet your needs, create your own conventions. A convention is:

- A static type with methods.
- Capable of defining response types and naming requirements on actions.

Response types

These methods are annotated with `[ProducesResponseType]` or `[ProducesDefaultResponseType]` attributes. For example:

```
C#

public static class MyAppConventions
{
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status404NotFound)]
    public static void Find(int id)
    {
    }
}
```

If more specific metadata attributes are absent, applying this convention to an assembly enforces that:

- The convention method applies to any action named `Find`.
- A parameter named `id` is present on the `Find` action.

Naming requirements

The `[ApiConventionNameMatch]` and `[ApiConventionTypeMatch]` attributes can be applied to the convention method that determines the actions to which they apply. For example:

C#

```
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ApiConventionNameMatch(ApiConventionNameMatchBehavior.Prefix)]
public static void Find(
    [ApiConventionNameMatch(ApiConventionNameMatchBehavior.Suffix)]
    int id)
{ }
```

In the preceding example:

- The `Microsoft.AspNetCore.Mvc.ApiExplorer.ApiConventionNameMatchBehavior.Prefix` option applied to the method indicates that the convention matches any action prefixed with "Find". Examples of matching actions include `Find`, `FindPet`, and `FindById`.
- The `Microsoft.AspNetCore.Mvc.ApiExplorer.ApiConventionNameMatchBehavior.Suffix` applied to the parameter indicates that the convention matches methods with exactly one parameter ending in the suffix identifier. Examples include parameters such as `id` or `petId`. `ApiConventionTypeMatch` can be similarly applied to types to constrain the parameter type. A `params[]` argument indicates remaining parameters that don't need to be explicitly matched.

Additional resources

- Video: Create metadata for NSwagClient
- Video: Beginner's Series to: Web APIs
- Use web API analyzers
- ASP.NET Core web API documentation with Swagger / OpenAPI

Handle errors in ASP.NET Core controller-based web APIs

Article • 08/02/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This article describes how to handle errors and customize error handling in controller-based ASP.NET Core web APIs. For information about error handling in minimal APIs, see Handle errors in ASP.NET Core and Handle errors in minimal APIs.

Developer Exception Page

The *Developer Exception Page* displays detailed information about unhandled request exceptions. It uses 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.
- The app was created with the current templates, that is, by using WebApplication.CreateBuilder.

Apps created using earlier templates, that is, by using WebHost.CreateDefaultBuilder, can enable the developer exception page by calling app.UseDeveloperExceptionPage.

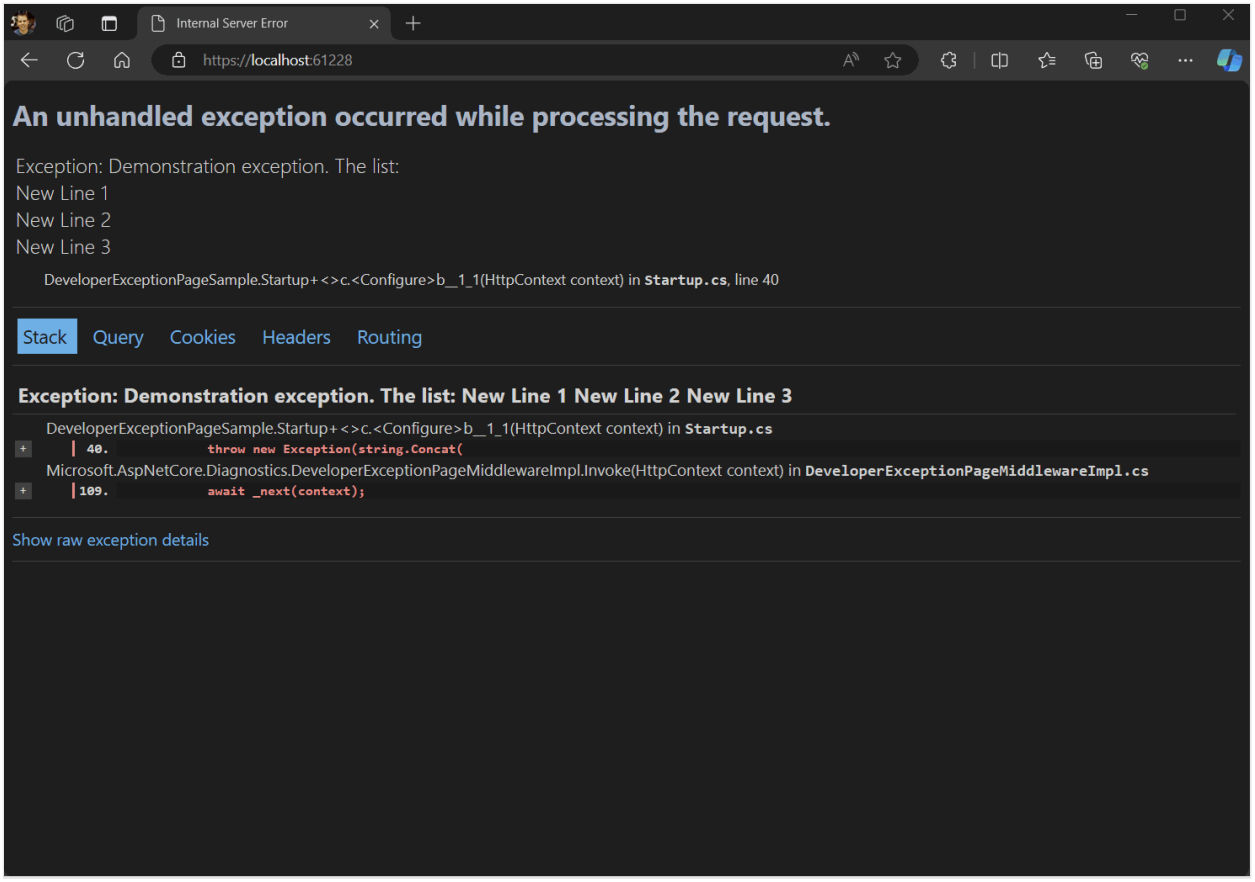
> **⚠ Warning**
>
> Don't enable the Developer Exception Page **unless the app is running in the Development environment**. Don't share detailed exception information publicly when the app runs in production. For more information on configuring environments, see **Use multiple environments in ASP.NET Core**.

The Developer Exception Page can include the following information about the exception and the request:

- Stack trace
- Query string parameters, if any
- Cookies, if any
- Headers
- Endpoint metadata, if any

The Developer Exception Page isn't guaranteed to provide any information. Use Logging for complete error information.

The following image shows a sample developer exception page with animation to show the tabs and the information displayed:



In response to a request with an `Accept: text/plain` header, the Developer Exception Page returns plain text instead of HTML. For example:

text

```
Status: 500 Internal Server Error
Time: 9.39 msSize: 480 bytes
FormattedRawHeadersRequest
Body
text/plain; charset=utf-8, 480 bytes
System.InvalidOperationException: Sample Exception
```

```
   at WebApplicationMinimal.Program.<>c.<Main>b__0_0() in
C:\Source\WebApplicationMinimal\Program.cs:line 12
   at lambda_method1(Closure, Object, HttpContext)
   at
Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddlewareImpl.Invoke
(HttpContext context)

HEADERS
=======
Accept: text/plain
Host: localhost:7267
traceparent: 00-0eab195ea19d07b90a46cd7d6bf2f
```

To see the Developer Exception Page:

- Add the following controller action to a controller-based API. The action throws an exception when the endpoint is requested.

  C#

  ```
  [HttpGet("Throw")]
  public IActionResult Throw() =>
      throw new Exception("Sample exception.");
  ```

- Run the app in the development environment.
- Go to the endpoint defined by the controller action.

Exception handler

In non-development environments, use Exception Handling Middleware to produce an error payload:

1. In `Program.cs`, call UseExceptionHandler to add the Exception Handling Middleware:

   C#

   ```
   var app = builder.Build();

   app.UseHttpsRedirection();

   if (!app.Environment.IsDevelopment())
   {
       app.UseExceptionHandler("/error");
   }

   app.UseAuthorization();
   ```

```
app.MapControllers();

app.Run();
```

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

```
C#

[Route("/error")]
public IActionResult HandleError() =>
    Problem();
```

The preceding `HandleError` action sends an RFC 7807 -compliant payload to the client.

> **⚠ Warning**
>
> Don't mark the error handler action method with HTTP method attributes, such as `HttpGet`. Explicit verbs prevent some requests from reaching the action method.
>
> For web APIs that use Swagger / OpenAPI, mark the error handler action with the [ApiExplorerSettings] attribute and set its IgnoreApi property to `true`. This attribute configuration excludes the error handler action from the app's OpenAPI specification:
>
> ```
> C#
>
> [ApiExplorerSettings(IgnoreApi = true)]
> ```
>
> Allow anonymous access to the method if unauthenticated users should see the error.

Exception Handling Middleware can also be used in the Development environment to produce a consistent payload format across all environments:

1. In `Program.cs`, register environment-specific Exception Handling Middleware instances:

```
C#

if (app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/error-development");
}
```

```
else
{
    app.UseExceptionHandler("/error");
}
```

In the preceding code, the middleware is registered with:

- A route of `/error-development` in the Development environment.
- A route of `/error` in non-Development environments.

2. Add controller actions for both the Development and non-Development routes:

```
C#

[Route("/error-development")]
public IActionResult HandleErrorDevelopment(
    [FromServices] IHostEnvironment hostEnvironment)
{
    if (!hostEnvironment.IsDevelopment())
    {
        return NotFound();
    }

    var exceptionHandlerFeature =
        HttpContext.Features.Get<IExceptionHandlerFeature>()!;

    return Problem(
        detail: exceptionHandlerFeature.Error.StackTrace,
        title: exceptionHandlerFeature.Error.Message);
}

[Route("/error")]
public IActionResult HandleError() =>
    Problem();
```

Use exceptions to modify the response

The contents of the response can be modified from outside of the controller using a custom exception and an action filter:

1. Create a well-known exception type named `HttpResponseException`:

```
C#

public class HttpResponseException : Exception
{
    public HttpResponseException(int statusCode, object? value = null)
=>
```

```
        (StatusCode, Value) = (statusCode, value);

    public int StatusCode { get; }

    public object? Value { get; }
}
```

2. Create an action filter named `HttpResponseExceptionFilter`:

```C#
public class HttpResponseExceptionFilter : IActionFilter,
IOrderedFilter
{
    public int Order => int.MaxValue - 10;

    public void OnActionExecuting(ActionExecutingContext context) { }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        if (context.Exception is HttpResponseException
httpResponseException)
        {
            context.Result = new
ObjectResult(httpResponseException.Value)
            {
                StatusCode = httpResponseException.StatusCode
            };

            context.ExceptionHandled = true;
        }
    }
}
```

   The preceding filter specifies an `Order` of the maximum integer value minus 10. This `Order` allows other filters to run at the end of the pipeline.

3. In `Program.cs`, add the action filter to the filters collection:

```C#
builder.Services.AddControllers(options =>
{
    options.Filters.Add<HttpResponseExceptionFilter>();
});
```

Validation failure error response

For web API controllers, MVC responds with a ValidationProblemDetails response type when model validation fails. MVC uses the results of InvalidModelStateResponseFactory to construct the error response for a validation failure. The following example replaces the default factory with an implementation that also supports formatting responses as XML, in `Program.cs`:

C#

```
builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
            new BadRequestObjectResult(context.ModelState)
            {
                ContentTypes =
                {
                    // using static System.Net.Mime.MediaTypeNames;
                    Application.Json,
                    Application.Xml
                }
            };
    })
    .AddXmlSerializerFormatters();
```

Client error response

An *error result* is defined as a result with an HTTP status code of 400 or higher. For web API controllers, MVC transforms an error result to produce a ProblemDetails.

The automatic creation of a `ProblemDetails` for error status codes is enabled by default, but error responses can be configured in one of the following ways:

1. Use the problem details service
2. Implement ProblemDetailsFactory
3. Use ApiBehaviorOptions.ClientErrorMapping

Default problem details response

The following `Program.cs` file was generated by the web application templates for API controllers:

C#

```
var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Consider the following controller, which returns BadRequest when the input is invalid:

C#

```
[Route("api/[controller]/[action]")]
[ApiController]
public class Values2Controller : ControllerBase
{
    // /api/values2/divide/1/2
    [HttpGet("{Numerator}/{Denominator}")]
    public IActionResult Divide(double Numerator, double Denominator)
    {
        if (Denominator == 0)
        {
            return BadRequest();
        }

        return Ok(Numerator / Denominator);
    }

    // /api/values2 /squareroot/4
    [HttpGet("{radicand}")]
    public IActionResult Squareroot(double radicand)
    {
        if (radicand < 0)
        {
            return BadRequest();
        }

        return Ok(Math.Sqrt(radicand));
    }
}
```

A problem details response is generated with the preceding code when any of the following conditions apply:

- The `/api/values2/divide` endpoint is called with a zero denominator.
- The `/api/values2/squareroot` endpoint is called with a radicand less than zero.

The default problem details response body has the following `type`, `title`, and `status` values:

JSON

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "Bad Request",
  "status": 400,
  "traceId": "00-84c1fd4063c38d9f3900d06e56542d48-85d1d4-00"
}
```

Problem details service

ASP.NET Core supports creating Problem Details for HTTP APIs using the IProblemDetailsService. For more information, see the Problem details service.

The following code configures the app to generate a problem details response for all HTTP client and server error responses that ***don't have body content yet***:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseExceptionHandler();
app.UseStatusCodePages();

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.MapControllers();
app.Run();
```

Consider the API controller from the preceding section, which returns BadRequest when the input is invalid:

C#

```
[Route("api/[controller]/[action]")]
[ApiController]
public class Values2Controller : ControllerBase
```

```
{
    // /api/values2/divide/1/2
    [HttpGet("{Numerator}/{Denominator}")]
    public IActionResult Divide(double Numerator, double Denominator)
    {
        if (Denominator == 0)
        {
            return BadRequest();
        }

        return Ok(Numerator / Denominator);
    }

    // /api/values2 /squareroot/4
    [HttpGet("{radicand}")]
    public IActionResult Squareroot(double radicand)
    {
        if (radicand < 0)
        {
            return BadRequest();
        }

        return Ok(Math.Sqrt(radicand));
    }
}
```

A problem details response is generated with the preceding code when any of the following conditions apply:

- An invalid input is supplied.
- The URI has no matching endpoint.
- An unhandled exception occurs.

The automatic creation of a `ProblemDetails` for error status codes is disabled when the SuppressMapClientErrors property is set to `true`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressMapClientErrors = true;
    });

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();
```

```
app.MapControllers();

app.Run();
```

Using the preceding code, when an API controller returns `BadRequest`, an HTTP 400 response status is returned with no response body. `SuppressMapClientErrors` prevents a `ProblemDetails` response from being created, even when calling `WriteAsync` for an API Controller endpoint. `WriteAsync` is explained later in this article.

The next section shows how to customize the problem details response body, using CustomizeProblemDetails, to return a more helpful response. For more customization options, see Customizing problem details.

Customize problem details with `CustomizeProblemDetails`

The following code uses ProblemDetailsOptions to set CustomizeProblemDetails:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

builder.Services.AddProblemDetails(options =>
        options.CustomizeProblemDetails = (context) =>
        {

            var mathErrorFeature = context.HttpContext.Features

.Get<MathErrorFeature>();
            if (mathErrorFeature is not null)
            {
                (string Detail, string Type) details =
mathErrorFeature.MathError switch
                {
                    MathErrorType.DivisionByZeroError =>
                    ("Divison by zero is not defined.",

"https://wikipedia.org/wiki/Division_by_zero"),
                    _ => ("Negative or complex numbers are not valid
input.",

"https://wikipedia.org/wiki/Square_root")
                };

                context.ProblemDetails.Type = details.Type;
                context.ProblemDetails.Title = "Bad Input";
                context.ProblemDetails.Detail = details.Detail;
```

```
            }
        }
    );

var app = builder.Build();

app.UseHttpsRedirection();

app.UseStatusCodePages();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The updated API controller:

C#

```
[Route("api/[controller]/[action]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // /api/values/divide/1/2
    [HttpGet("{Numerator}/{Denominator}")]
    public IActionResult Divide(double Numerator, double Denominator)
    {
        if (Denominator == 0)
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.DivisionByZeroError
            };
            HttpContext.Features.Set(errorType);
            return BadRequest();
        }

        return Ok(Numerator / Denominator);
    }

    // /api/values/squareroot/4
    [HttpGet("{radicand}")]
    public IActionResult Squareroot(double radicand)
    {
        if (radicand < 0)
```

```
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.NegativeRadicandError
            };
            HttpContext.Features.Set(errorType);
            return BadRequest();
        }
        return Ok(Math.Sqrt(radicand));
    }

}
```

The following code contains the `MathErrorFeature` and `MathErrorType`, which are used with the preceding sample:

C#

```
// Custom Http Request Feature
class MathErrorFeature
{
    public MathErrorType MathError { get; set; }
}

// Custom math errors
enum MathErrorType
{
    DivisionByZeroError,
    NegativeRadicandError
}
```

A problem details response is generated with the preceding code when any of the following conditions apply:

- The `/divide` endpoint is called with a zero denominator.
- The `/squareroot` endpoint is called with a radicand less than zero.
- The URI has no matching endpoint.

The problem details response body contains the following when either `squareroot` endpoint is called with a radicand less than zero:

JSON

```
{
  "type": "https://en.wikipedia.org/wiki/Square_root",
  "title": "Bad Input",
  "status": 400,
```

```
  "detail": "Negative or complex numbers are not allowed."
}
```

View or download sample code

Implement `ProblemDetailsFactory`

MVC uses Microsoft.AspNetCore.Mvc.Infrastructure.ProblemDetailsFactory to produce all instances of ProblemDetails and ValidationProblemDetails. This factory is used for:

- Client error responses
- Validation failure error responses
- ControllerBase.Problem and ControllerBase.ValidationProblem

To customize the problem details response, register a custom implementation of ProblemDetailsFactory in `Program.cs`:

C#

```
builder.Services.AddControllers();
builder.Services.AddTransient<ProblemDetailsFactory,
SampleProblemDetailsFactory>();
```

Use `ApiBehaviorOptions.ClientErrorMapping`

Use the ClientErrorMapping property to configure the contents of the `ProblemDetails` response. For example, the following code in `Program.cs` updates the Link property for 404 responses:

C#

```
builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });
```

Additional resources

- How to Use ModelState Validation in ASP.NET Core Web API
- View or download sample code
- Hellang.Middleware.ProblemDetails

Test web APIs with the HttpRepl

Article • 07/28/2023

The HTTP Read-Eval-Print Loop (REPL) is:

- A lightweight, cross-platform command-line tool that's supported everywhere .NET Core is supported.
- Used for making HTTP requests to test ASP.NET Core web APIs (and non-ASP.NET Core web APIs) and view their results.
- Capable of testing web APIs hosted in any environment, including localhost and Azure App Service.

The following HTTP verbs are supported:

- DELETE
- GET
- HEAD
- OPTIONS
- PATCH
- POST
- PUT

To follow along, view or download the sample ASP.NET Core web API (how to download).

Prerequisites

- .NET Core 3.1 SDK

Installation

To install the HttpRepl, run the following command:

.NET CLI

```
dotnet tool install -g Microsoft.dotnet-httprepl
```

A .NET Core Global Tool is installed from the Microsoft.dotnet-httprepl NuGet package.

> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

On macOS, update the path:

```
Bash

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

Usage

After successful installation of the tool, run the following command to start the HttpRepl:

```
Console

httprepl
```

To view the available HttpRepl commands, run one of the following commands:

```
Console

httprepl -h
```

```
Console

httprepl --help
```

The following output is displayed:

```
Console

Usage:
  httprepl [<BASE_ADDRESS>] [options]

Arguments:
  <BASE_ADDRESS> - The initial base address for the REPL.

Options:
```

```
  -h|--help - Show help information.

Once the REPL starts, these commands are valid:

Setup Commands:
Use these commands to configure the tool for your API server

connect        Configures the directory structure and base address of the
api server
set header     Sets or clears a header for all requests. e.g. `set header
content-type application/json`

HTTP Commands:
Use these commands to execute requests against your application.

GET            get - Issues a GET request
POST           post - Issues a POST request
PUT            put - Issues a PUT request
DELETE         delete - Issues a DELETE request
PATCH          patch - Issues a PATCH request
HEAD           head - Issues a HEAD request
OPTIONS        options - Issues a OPTIONS request

Navigation Commands:
The REPL allows you to navigate your URL space and focus on specific APIs
that you are working on.

ls             Show all endpoints for the current path
cd             Append the given directory to the currently selected path, or
move up a path when using `cd ..`

Shell Commands:
Use these commands to interact with the REPL shell.

clear          Removes all text from the shell
echo [on/off]  Turns request echoing on or off, show the request that was
made when using request commands
exit           Exit the shell

REPL Customization Commands:
Use these commands to customize the REPL behavior.

pref [get/set] Allows viewing or changing preferences, e.g. 'pref set
editor.command.default 'C:\\Program Files\\Microsoft VS Code\\Code.exe'`
run            Runs the script at the given path. A script is a set of
commands that can be typed with one command per line
ui             Displays the Swagger UI page, if available, in the default
browser

Use `help <COMMAND>` for more detail on an individual command. e.g. `help
get`.
For detailed tool info, see https://aka.ms/http-repl-doc.
```

The HttpRepl offers command completion. Pressing the `Tab` key iterates through the list of commands that complete the characters or API endpoint that you typed. The following sections outline the available CLI commands.

Connect to the web API

Connect to a web API by running the following command:

```
Console

httprepl <ROOT URI>
```

`<ROOT URI>` is the base URI for the web API. For example:

```
Console

httprepl https://localhost:5001
```

Alternatively, run the following command at any time while the HttpRepl is running:

```
Console

connect <ROOT URI>
```

For example:

```
Console

(Disconnected)> connect https://localhost:5001
```

Manually point to the OpenAPI description for the web API

The connect command above will attempt to find the OpenAPI description automatically. If for some reason it's unable to do so, you can specify the URI of the OpenAPI description for the web API by using the `--openapi` option:

```
Console

connect <ROOT URI> --openapi <OPENAPI DESCRIPTION ADDRESS>
```

For example:

```
Console
(Disconnected)> connect https://localhost:5001 --openapi
/swagger/v1/swagger.json
```

Enable verbose output for details on OpenAPI description searching, parsing, and validation

Specifying the `--verbose` option with the `connect` command will produce more details when the tool searches for the OpenAPI description, parses, and validates it.

```
Console
connect <ROOT URI> --verbose
```

For example:

```
Console
(Disconnected)> connect https://localhost:5001 --verbose
Checking https://localhost:5001/swagger.json... 404 NotFound
Checking https://localhost:5001/swagger/v1/swagger.json... 404 NotFound
Checking https://localhost:5001/openapi.json... Found
Parsing... Successful (with warnings)
The field 'info' in 'document' object is REQUIRED [#/info]
The field 'paths' in 'document' object is REQUIRED [#/paths]
```

Navigate the web API

View available endpoints

To list the different endpoints (controllers) at the current path of the web API address, run the `ls` or `dir` command:

```
Console
https://localhost:5001/> ls
```

The following output format is displayed:

Console

```
.        []
Fruits   [get|post]
People   [get|post]

https://localhost:5001/>
```

The preceding output indicates that there are two controllers available: `Fruits` and `People`. Both controllers support parameterless HTTP GET and POST operations.

Navigating into a specific controller reveals more detail. For example, the following command's output shows the `Fruits` controller also supports HTTP GET, PUT, and DELETE operations. Each of these operations expects an `id` parameter in the route:

Console

```
https://localhost:5001/fruits> ls
.      [get|post]
..     []
{id}   [get|put|delete]

https://localhost:5001/fruits>
```

Alternatively, run the `ui` command to open the web API's Swagger UI page in a browser. For example:

Console

```
https://localhost:5001/> ui
```

Navigate to an endpoint

To navigate to a different endpoint on the web API, run the `cd` command:

Console

```
https://localhost:5001/> cd people
```

The path following the `cd` command is case insensitive. The following output format is displayed:

Console

```
/people    [get|post]

https://localhost:5001/people>
```

Customize the HttpRepl

The HttpRepl's default colors can be customized. Additionally, a default text editor can be defined. The HttpRepl preferences are persisted across the current session and are honored in future sessions. Once modified, the preferences are stored in the following file:

Windows

%USERPROFILE%\.httpreplprefs

The *.httpreplprefs* file is loaded on startup and not monitored for changes at runtime. Manual modifications to the file take effect only after restarting the tool.

View the settings

To view the available settings, run the `pref get` command. For example:

Console

```
https://localhost:5001/> pref get
```

The preceding command displays the available key-value pairs:

Console

```
colors.json=Green
colors.json.arrayBrace=BoldCyan
colors.json.comma=BoldYellow
colors.json.name=BoldMagenta
colors.json.nameSeparator=BoldWhite
colors.json.objectBrace=Cyan
colors.protocol=BoldGreen
colors.status=BoldYellow
```

Set color preferences

Response colorization is currently supported for JSON only. To customize the default HttpRepl tool coloring, locate the key corresponding to the color to be changed. For instructions on how to find the keys, see the View the settings section. For example, change the `colors.json` key value from `Green` to `White` as follows:

Console

```
https://localhost:5001/people> pref set colors.json White
```

Only the allowed colors may be used. Subsequent HTTP requests display output with the new coloring.

When specific color keys aren't set, more generic keys are considered. To demonstrate this fallback behavior, consider the following example:

- If `colors.json.name` doesn't have a value, `colors.json.string` is used.
- If `colors.json.string` doesn't have a value, `colors.json.literal` is used.
- If `colors.json.literal` doesn't have a value, `colors.json` is used.
- If `colors.json` doesn't have a value, the command shell's default text color (`AllowedColors.None`) is used.

Set indentation size

Response indentation size customization is currently supported for JSON only. The default size is two spaces. For example:

JSON

```
[
  {
    "id": 1,
    "name": "Apple"
  },
  {
    "id": 2,
    "name": "Orange"
  },
  {
    "id": 3,
    "name": "Strawberry"
  }
]
```

To change the default size, set the `formatting.json.indentSize` key. For example, to always use four spaces:

Console

```
pref set formatting.json.indentSize 4
```

Subsequent responses honor the setting of four spaces:

JSON

```
[
    {
        "id": 1,
        "name": "Apple"
    },
    {
        "id": 2,
        "name": "Orange"
    },
    {
        "id": 3,
        "name": "Strawberry"
    }
]
```

Set the default text editor

By default, the HttpRepl has no text editor configured for use. To test web API methods requiring an HTTP request body, a default text editor must be set. The HttpRepl tool launches the configured text editor for the sole purpose of composing the request body. Run the following command to set your preferred text editor as the default:

Console

```
pref set editor.command.default "<EXECUTABLE>"
```

In the preceding command, `<EXECUTABLE>` is the full path to the text editor's executable file. For example, run the following command to set Visual Studio Code as the default text editor:

Windows

Console

```
pref set editor.command.default "C:\Program Files\Microsoft VS
Code\Code.exe"
```

To launch the default text editor with specific CLI arguments, set the `editor.command.default.arguments` key. For example, assume Visual Studio Code is the default text editor and that you always want the HttpRepl to open Visual Studio Code in a new session with extensions disabled. Run the following command:

Console

```
pref set editor.command.default.arguments "--disable-extensions --new-
window"
```

> **Tip**
>
> If your default editor is Visual Studio Code, you'll usually want to pass the `-w` or `--wait` argument to force Visual Studio Code to wait for you to close the file before returning.

Set the OpenAPI Description search paths

By default, the HttpRepl has a set of relative paths that it uses to find the OpenAPI description when executing the `connect` command without the `--openapi` option. These relative paths are combined with the root and base paths specified in the `connect` command. The default relative paths are:

- `swagger.json`
- `swagger/v1/swagger.json`
- `/swagger.json`
- `/swagger/v1/swagger.json`
- `openapi.json`
- `/openapi.json`

To use a different set of search paths in your environment, set the `swagger.searchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

Console

```
pref set swagger.searchPaths
"swagger/v2/swagger.json|swagger/v3/swagger.json"
```

Instead of replacing the default list altogether, the list can also be modified by adding or removing paths.

To add one or more search paths to the default list, set the `swagger.addToSearchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

```
Console

pref set swagger.addToSearchPaths
"openapi/v2/openapi.json|openapi/v3/openapi.json"
```

To remove one or more search paths from the default list, set the `swagger.addToSearchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

```
Console

pref set swagger.removeFromSearchPaths "swagger.json|/swagger.json"
```

Test HTTP GET requests

Synopsis

```
Console

get <PARAMETER> [-F|--no-formatting] [-h|--header] [--response:body] [--
response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

The following options are available for the `get` command:

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:

- `{header}={value}`
- `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Example

To issue an HTTP GET request:

1. Run the `get` command on an endpoint that supports it:

   Console

   ```
   https://localhost:5001/people> get
   ```

   The preceding command displays the following output format:

   Console

   ```
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Fri, 21 Jun 2019 03:38:45 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "name": "Scott Hunter"
     },
     {
       "id": 2,
       "name": "Scott Hanselman"
     },
   ```

```
  {
    "id": 3,
    "name": "Scott Guthrie"
  }
]


https://localhost:5001/people>
```

2. Retrieve a specific record by passing a parameter to the `get` command:

```
Console

https://localhost:5001/people> get 2
```

The preceding command displays the following output format:

```
Console

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 21 Jun 2019 06:17:57 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 2,
    "name": "Scott Hanselman"
  }
]


https://localhost:5001/people>
```

Test HTTP POST requests

Synopsis

```
Console

post <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--
no-formatting] [--response] [--response:body] [--response:headers] [-s|--
streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

Indicates that no HTTP request body is needed.

Example

To issue an HTTP POST request:

1. Run the `post` command on an endpoint that supports it:

   ```
   Console

   https://localhost:5001/people> post -h Content-Type=application/json
   ```

   In the preceding command, the `Content-Type` HTTP request header is set to indicate a request body media type of JSON. The default text editor opens a *.tmp* file with a JSON template representing the HTTP request body. For example:

   ```
   JSON

   {
     "id": 0,
     "name": ""
   }
   ```

   > **Tip**
   >
   > To set the default text editor, see the **Set the default text editor** section.

2. Modify the JSON template to satisfy model validation requirements:

   ```
   JSON

   {
     "id": 0,
     "name": "Scott Addie"
   }
   ```

3. Save the *.tmp* file, and close the text editor. The following output appears in the command shell:

   ```
   Console

   HTTP/1.1 201 Created
   Content-Type: application/json; charset=utf-8
   Date: Thu, 27 Jun 2019 21:24:18 GMT
   ```

```
Location: https://localhost:5001/people/4
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 4,
  "name": "Scott Addie"
}


https://localhost:5001/people>
```

Test HTTP PUT requests

Synopsis

Console

```
put <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Example

To issue an HTTP PUT request:

1. *Optional*: Run the `get` command to view the data before modifying it:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:07:32 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "data": "Apple"
     },
   ```

```
  {
    "id": 2,
    "data": "Orange"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]
```

2. Run the `put` command on an endpoint that supports it:

```
Console

https://localhost:5001/fruits> put 2 -h Content-Type=application/json
```

In the preceding command, the `Content-Type` HTTP request header is set to indicate a request body media type of JSON. The default text editor opens a *.tmp* file with a JSON template representing the HTTP request body. For example:

```
JSON

{
  "id": 0,
  "name": ""
}
```

> **Tip**
>
> To set the default text editor, see the **Set the default text editor** section.

3. Modify the JSON template to satisfy model validation requirements:

```
JSON

{
  "id": 2,
  "name": "Cherry"
}
```

4. Save the *.tmp* file, and close the text editor. The following output appears in the command shell:

```
Console
```

```
[main 2019-06-28T17:27:01.805Z] update#setState idle
HTTP/1.1 204 No Content
Date: Fri, 28 Jun 2019 17:28:21 GMT
Server: Kestrel
```

5. *Optional*: Issue a `get` command to see the modifications. For example, if you typed "Cherry" in the text editor, a `get` returns the following output:

Console

```
https://localhost:5001/fruits> get
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Sat, 22 Jun 2019 00:08:20 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 1,
    "data": "Apple"
  },
  {
    "id": 2,
    "data": "Cherry"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]


https://localhost:5001/fruits>
```

Test HTTP DELETE requests

Synopsis

Console

```
delete <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Example

To issue an HTTP DELETE request:

1. *Optional*: Run the `get` command to view the data before modifying it:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:07:32 GMT
   Server: Kestrel
   Transfer-Encoding: chunked
   ```

```
[
  {
    "id": 1,
    "data": "Apple"
  },
  {
    "id": 2,
    "data": "Orange"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]
```

2. Run the `delete` command on an endpoint that supports it:

   Console

   ```
   https://localhost:5001/fruits> delete 2
   ```

   The preceding command displays the following output format:

   Console

   ```
   HTTP/1.1 204 No Content
   Date: Fri, 28 Jun 2019 17:36:42 GMT
   Server: Kestrel
   ```

3. *Optional*: Issue a `get` command to see the modifications. In this example, a `get` returns the following output:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:16:30 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "data": "Apple"
     },
     {
       "id": 3,
       "data": "Strawberry"
     }
   ```

```
]


https://localhost:5001/fruits>
```

Test HTTP PATCH requests

Synopsis

Console

```
patch <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't

exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Test HTTP HEAD requests

Synopsis

Console

```
head <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

Sets an HTTP request header. The following two value formats are supported:
    - `{header}={value}`
    - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Test HTTP OPTIONS requests

Synopsis

Console

```
options <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Set HTTP request headers

To set an HTTP request header, use one of the following approaches:

- Set inline with the HTTP request. For example:

  Console

  ```
  https://localhost:5001/people> post -h Content-Type=application/json
  ```

  With the preceding approach, each distinct HTTP request header requires its own `-h` option.

- Set before sending the HTTP request. For example:

  Console

  ```
  https://localhost:5001/people> set header Content-Type application/json
  ```

  When setting the header before sending a request, the header remains set for the duration of the command shell session. To clear the header, provide an empty value. For example:

  Console

```
https://localhost:5001/people> set header Content-Type
```

Test secured endpoints

The HttpRepl supports the testing of secured endpoints in the following ways:

- Via the default credentials of the logged in user.
- Through the use of HTTP request headers.

Default credentials

Consider a web API you're testing that's hosted in IIS and secured with Windows authentication. You want the credentials of the user running the tool to flow across to the HTTP endpoints being tested. To pass the default credentials of the logged in user:

1. Set the `httpClient.useDefaultCredentials` preference to `true`:

   Console

   ```
   pref set httpClient.useDefaultCredentials true
   ```

2. Exit and restart the tool before sending another request to the web API.

Default proxy credentials

Consider a scenario in which the web API you're testing is behind a proxy secured with Windows authentication. You want the credentials of the user running the tool to flow to the proxy. To pass the default credentials of the logged in user:

1. Set the `httpClient.proxy.useDefaultCredentials` preference to `true`:

   Console

   ```
   pref set httpClient.proxy.useDefaultCredentials true
   ```

2. Exit and restart the tool before sending another request to the web API.

HTTP request headers

Examples of supported authentication and authorization schemes include:

- basic authentication
- JWT bearer tokens
- digest authentication

For example, you can send a bearer token to an endpoint with the following command:

Console

```
set header Authorization "bearer <TOKEN VALUE>"
```

To access an Azure-hosted endpoint or to use the Azure REST API, you need a bearer token. Use the following steps to obtain a bearer token for your Azure subscription via the Azure CLI. The HttpRepl sets the bearer token in an HTTP request header. A list of Azure App Service Web Apps is retrieved.

1. Sign in to Azure:

   Azure CLI

   ```
   az login
   ```

2. Get your subscription ID with the following command:

   Azure CLI

   ```
   az account show --query id
   ```

3. Copy your subscription ID and run the following command:

   Azure CLI

   ```
   az account set --subscription "<SUBSCRIPTION ID>"
   ```

4. Get your bearer token with the following command:

   Azure CLI

   ```
   az account get-access-token --query accessToken
   ```

5. Connect to the Azure REST API via the HttpRepl:

   Console

```
httprepl https://management.azure.com
```

6. Set the `Authorization` HTTP request header:

```
Console

https://management.azure.com/> set header Authorization "bearer <ACCESS
TOKEN>"
```

7. Navigate to the subscription:

```
Console

https://management.azure.com/> cd subscriptions/<SUBSCRIPTION ID>
```

8. Get a list of your subscription's Azure App Service Web Apps:

```
Console

https://management.azure.com/subscriptions/{SUBSCRIPTION ID}> get
providers/Microsoft.Web/sites?api-version=2016-08-01
```

The following response is displayed:

```
Console

HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Length: 35948
Content-Type: application/json; charset=utf-8
Date: Thu, 19 Sep 2019 23:04:03 GMT
Expires: -1
Pragma: no-cache
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
x-ms-correlation-request-id: <em>xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx</em>
x-ms-original-request-ids: <em>xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx;xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx</em>
x-ms-ratelimit-remaining-subscription-reads: 11999
x-ms-request-id: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
x-ms-routing-request-id: WESTUS:xxxxxxxxxxxxxxxx:xxxxxxxx-xxxx-xxxx-
xxxx-xxxxxxxxxx
{
  "value": [
    <AZURE RESOURCES LIST>
```

```
  ]
}
```

Toggle HTTP request display

By default, display of the HTTP request being sent is suppressed. It's possible to change the corresponding setting for the duration of the command shell session.

Enable request display

View the HTTP request being sent by running the `echo on` command. For example:

Console

```
https://localhost:5001/people> echo on
Request echoing is on
```

Subsequent HTTP requests in the current session display the request headers. For example:

Console

```
https://localhost:5001/people> post

[main 2019-06-28T18:50:11.930Z] update#setState idle
Request to https://localhost:5001...

POST /people HTTP/1.1
Content-Length: 41
Content-Type: application/json
User-Agent: HTTP-REPL

{
  "id": 0,
  "name": "Scott Addie"
}

Response from https://localhost:5001...

HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Fri, 28 Jun 2019 18:50:21 GMT
Location: https://localhost:5001/people/4
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 4,
```

```
  "name": "Scott Addie"
}


https://localhost:5001/people>
```

Disable request display

Suppress display of the HTTP request being sent by running the `echo off` command. For example:

Console

```
https://localhost:5001/people> echo off
Request echoing is off
```

Run a script

If you frequently execute the same set of HttpRepl commands, consider storing them in a text file. Commands in the file take the same form as commands executed manually on the command line. The commands can be executed in a batched fashion using the `run` command. For example:

1. Create a text file containing a set of newline-delimited commands. To illustrate, consider a *people-script.txt* file containing the following commands:

   text

   ```
   set base https://localhost:5001
   ls
   cd People
   ls
   get 1
   ```

2. Execute the `run` command, passing in the text file's path. For example:

   Console

   ```
   https://localhost:5001/> run C:\http-repl-scripts\people-script.txt
   ```

   The following output appears:

   Console

```
https://localhost:5001/> set base https://localhost:5001
Using OpenAPI description at
https://localhost:5001/swagger/v1/swagger.json

https://localhost:5001/> ls
.        []
Fruits   [get|post]
People   [get|post]

https://localhost:5001/> cd People
/People    [get|post]

https://localhost:5001/People> ls
.      [get|post]
..     []
{id}   [get|put|delete]

https://localhost:5001/People> get 1
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 12 Jul 2019 19:20:10 GMT
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 1,
  "name": "Scott Hunter"
}


https://localhost:5001/People>
```

Clear the output

To remove all output written to the command shell by the HttpRepl tool, run the `clear` or `cls` command. To illustrate, imagine the command shell contains the following output:

Console

```
httprepl https://localhost:5001
(Disconnected)> set base "https://localhost:5001"
Using OpenAPI description at https://localhost:5001/swagger/v1/swagger.json

https://localhost:5001/> ls
.        []
Fruits   [get|post]
People   [get|post]
```

```
https://localhost:5001/>
```

Run the following command to clear the output:

Console

```
https://localhost:5001/> clear
```

After running the preceding command, the command shell contains only the following output:

Console

```
https://localhost:5001/>
```

Additional resources

- REST API requests
- HttpRepl GitHub repository
- Configure Visual Studio to launch HttpRepl
- Configure Visual Studio Code to launch HttpRepl
- Configure Visual Studio for Mac to launch HttpRepl

HttpRepl telemetry

Article • 06/17/2024

The HttpRepl includes a telemetry feature that collects usage data. It's important that the HttpRepl team understands how the tool is used so it can be improved.

How to opt out

The HttpRepl telemetry feature is enabled by default. To opt out of the telemetry feature, set the `DOTNET_HTTPREPL_TELEMETRY_OPTOUT` environment variable to `1` or `true`.

Disclosure

The HttpRepl displays text similar to the following when you first run the tool. Text may vary slightly depending on the version of the tool you're running. This "first run" experience is how Microsoft notifies you about data collection.

Console

```
Telemetry
---------
The .NET tools collect usage data in order to help us improve your
experience. It is collected by Microsoft and shared with the community. You
can opt-out of telemetry by setting the DOTNET_HTTPREPL_TELEMETRY_OPTOUT
environment variable to '1' or 'true' using your favorite shell.
```

To suppress the "first run" experience text, set the `DOTNET_HTTPREPL_SKIP_FIRST_TIME_EXPERIENCE` environment variable to `1` or `true`.

Data points

The telemetry feature doesn't:

- Collect personal data, such as usernames, email addresses, or URLs.
- Scan your HTTP requests or responses.

The data is sent securely to Microsoft servers and held under restricted access.

Protecting your privacy is important to us. If you suspect the telemetry feature is collecting sensitive data or the data is being insecurely or inappropriately handled, take one of the following actions:

- File an issue in the dotnet/httprepl repository.
- Send an email to dotnet@microsoft.com for investigation.

The telemetry feature collects the following data.

Expand table

.NET SDK versions	Data
>=5.0	Timestamp of invocation.
>=5.0	Three-octet IP address used to determine the geographical location.
>=5.0	Operating system and version.
>=5.0	Runtime ID (RID) the tool is running on.
>=5.0	Whether the tool is running in a container.
>=5.0	Hashed Media Access Control (MAC) address: a cryptographically (SHA256) hashed and unique ID for a machine.
>=5.0	Kernel version.
>=5.0	HttpRepl version.
>=5.0	Whether the tool was started with `help`, `run`, or `connect` arguments. Actual argument values aren't collected.
>=5.0	Command invoked (for example, `get`) and whether it succeeded.
>=5.0	For the `connect` command, whether the `root`, `base`, or `openapi` arguments were supplied. Actual argument values aren't collected.
>=5.0	For the `pref` command, whether a `get` or `set` was issued and which preference was accessed. If not a well-known preference, the name is hashed. The value isn't collected.
>=5.0	For the `set header` command, the header name being set. If not a well-known header, the name is hashed. The value isn't collected.
>=5.0	For the `connect` command, whether a special case for `dotnet new webapi` was used and, whether it was bypassed via preference.
>=5.0	For all HTTP commands (for example, GET, POST, PUT), whether each of the options was specified. The values of the options aren't collected.

Additional resources

- .NET Core SDK telemetry
- .NET CLI telemetry data

Minimal APIs overview

Article • 07/26/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

Minimal APIs are a simplified approach for building fast HTTP APIs with ASP.NET Core. You can build fully functioning REST endpoints with minimal code and configuration. Skip traditional scaffolding and avoid unnecessary controllers by fluently declaring API routes and actions. For example, the following code creates an API at the root of the web app that returns the text, `"Hello World!"`.

C#

```
var app = WebApplication.Create(args);

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

app.Run();
```

Most APIs accept parameters as part of the route.

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/users/{userId}/books/{bookId}",
    (int userId, int bookId) => $"The user id is {userId} and book id is
{bookId}");

app.Run();
```

That's all it takes to get started, but it's not all that's available. Minimal APIs support the configuration and customization needed to scale to multiple APIs, handle complex routes, apply authorization rules, and control the content of API responses. A good place to get started is Tutorial: Create a minimal API with ASP.NET Core.

Want to see some code examples?

For a full list of common scenarios with code examples, see Minimal APIs quick reference.

Want to jump straight into your first project?

Build a minimal API app with our tutorial: Tutorial: Create a minimal API with ASP.NET Core.

Tutorial: Create a minimal API with ASP.NET Core

Article • 08/21/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson and Tom Dykstra

Minimal APIs are architected to create HTTP APIs with minimal dependencies. They're ideal for microservices and apps that want to include only the minimum files, features, and dependencies in ASP.NET Core.

This tutorial teaches the basics of building a minimal API with ASP.NET Core. Another approach to creating APIs in ASP.NET Core is to use controllers. For help with choosing between minimal APIs and controller-based APIs, see APIs overview. For a tutorial on creating an API project based on controllers that contains more features, see Create a web API.

Overview

This tutorial creates the following API:

Expand table

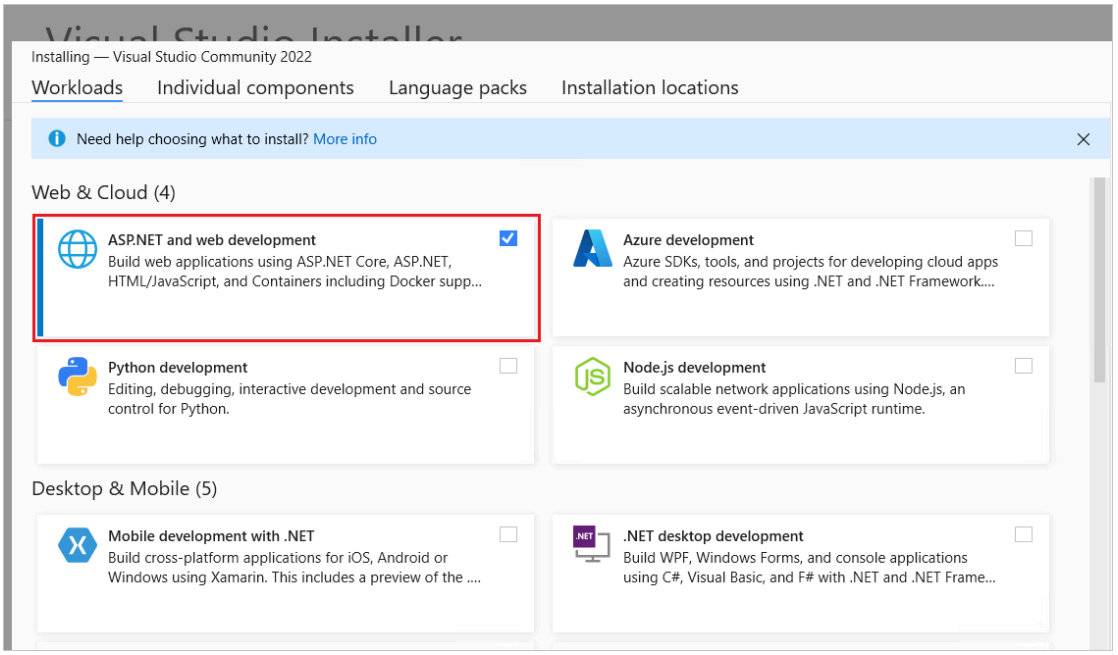
API	Description	Request body	Response body
`GET /todoitems`	Get all to-do items	None	Array of to-do items
`GET /todoitems/complete`	Get completed to-do items	None	Array of to-do items
`GET /todoitems/{id}`	Get an item by ID	None	To-do item
`POST /todoitems`	Add a new item	To-do item	To-do item
`PUT /todoitems/{id}`	Update an existing item	To-do item	None

API	Description	Request body	Response body
`DELETE /todoitems/{id}`	Delete an item	None	None

Prerequisites

Visual Studio

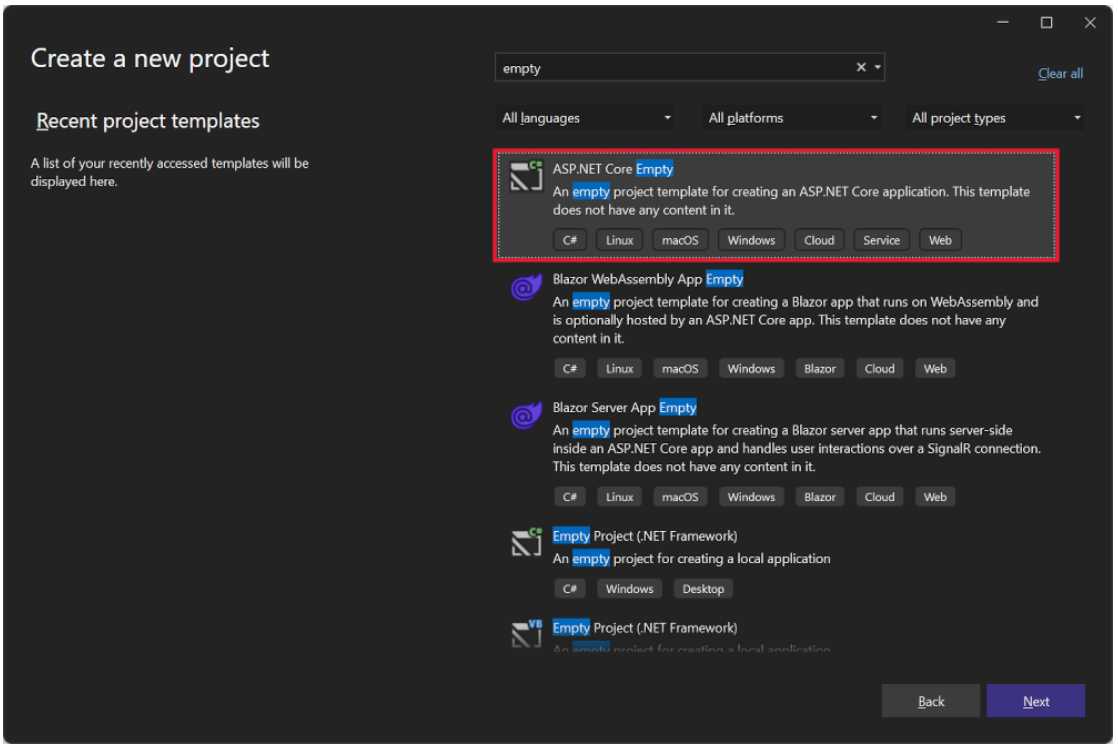
- Visual Studio 2022 Preview with the **ASP.NET and web development** workload.



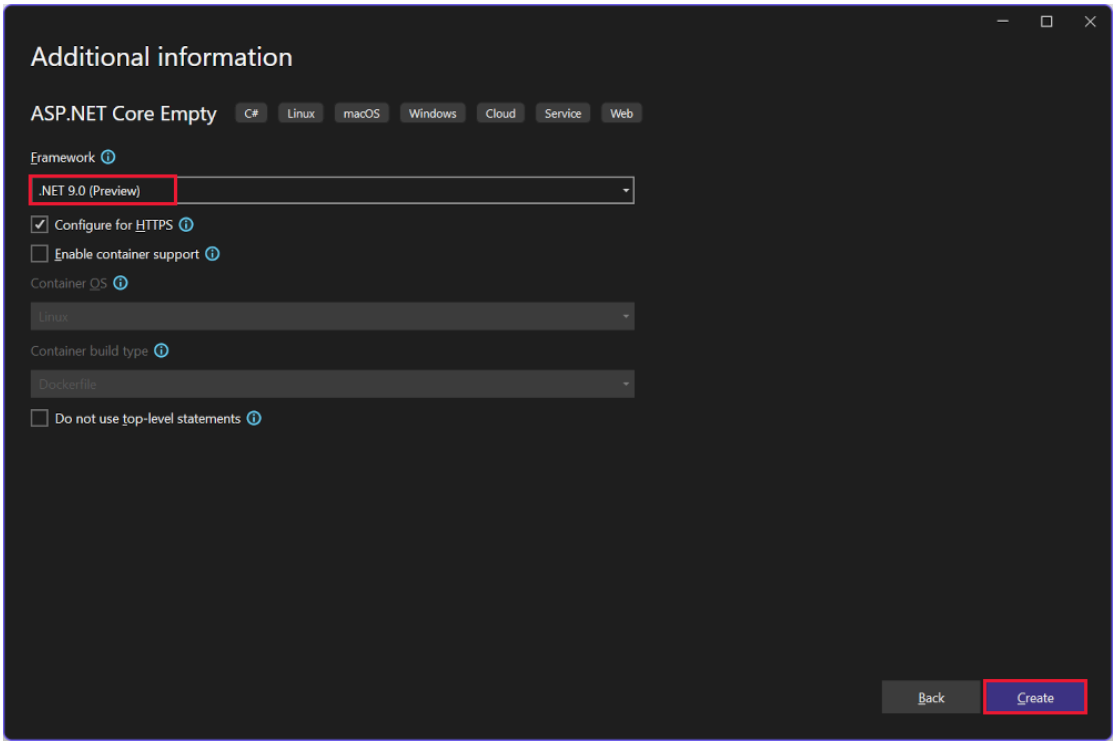
Create an API project

Visual Studio

- Start Visual Studio 2022 and select **Create a new project**.
- In the **Create a new project** dialog:
  - Enter `Empty` in the **Search for templates** search box.
  - Select the **ASP.NET Core Empty** template and select **Next**.



- Name the project *TodoApi* and select **Next**.
- In the **Additional information** dialog:
  - Select **.NET 9.0 (Preview)**
  - Uncheck **Do not use top-level statements**
  - Select **Create**



Examine the code

The `Program.cs` file contains the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

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

app.Run();
```

The preceding code:

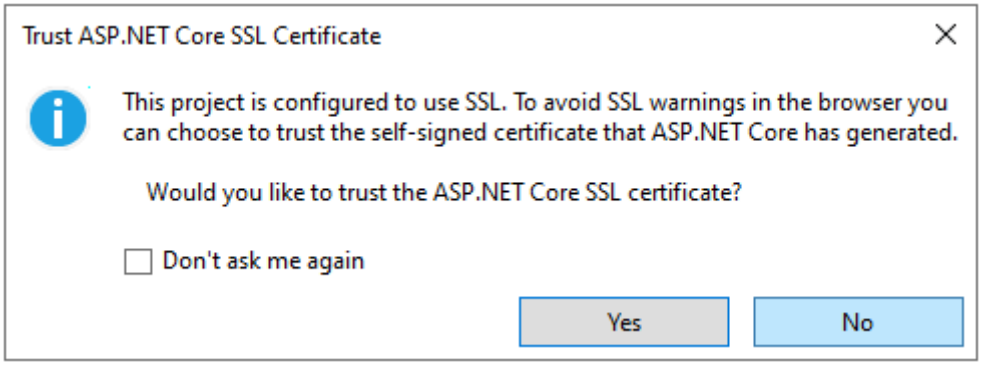
- Creates a WebApplicationBuilder and a WebApplication with preconfigured defaults.
- Creates an HTTP GET endpoint `/` that returns `Hello World!`:

Run the app

Visual Studio

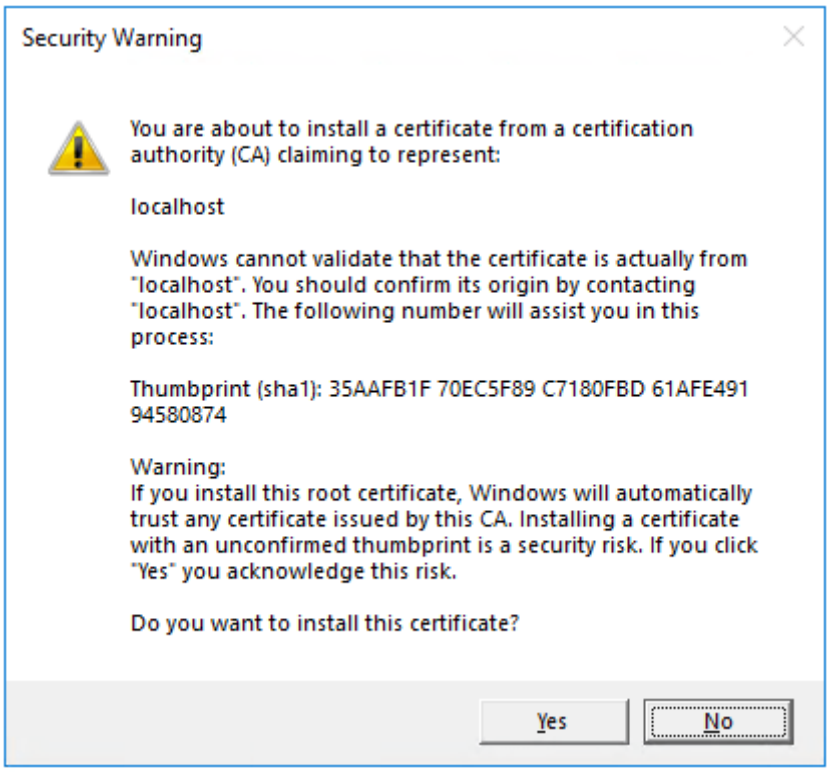
Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



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

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio launches the Kestrel web server and opens a browser window.

`Hello World!` is displayed in the browser. The `Program.cs` file contains a minimal but complete app.

Close the browser window.

Add NuGet packages

NuGet packages must be added to support the database and diagnostics used in this tutorial.

Visual Studio

- From the **Tools** menu, select **NuGet Package Manager** > **Manage NuGet Packages for Solution**.
- Select the **Browse** tab.
- Select **Include Prelease**.
- Enter **Microsoft.EntityFrameworkCore.InMemory** in the search box, and then select `Microsoft.EntityFrameworkCore.InMemory`.

- Select the **Project** checkbox in the right pane and then select **Install**.
- Follow the preceding instructions to add the `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` package.

The model and database context classes

- In the project folder, create a file named `Todo.cs` with the following code:

C#

```
public class Todo
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The preceding code creates the model for this app. A *model* is a class that represents data that the app manages.

- Create a file named `TodoDb.cs` with the following code:

C#

```
using Microsoft.EntityFrameworkCore;

class TodoDb : DbContext
{
    public TodoDb(DbContextOptions<TodoDb> options)
        : base(options) { }

    public DbSet<Todo> Todos => Set<Todo>();
}
```

The preceding code defines the *database context*, which is the main class that coordinates Entity Framework functionality for a data model. This class derives from the Microsoft.EntityFrameworkCore.DbContext class.

Add the API code

- Replace the contents of the `Program.cs` file with the following code:

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());

app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});
```

```
app.Run();
```

The following highlighted code adds the database context to the dependency injection (DI) container and enables displaying database-related exceptions:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();
```

The DI container provides access to the database context and other services.

Visual Studio

This tutorial uses Endpoints Explorer and .http files to test the API.

Test posting data

The following code in `Program.cs` creates an HTTP POST endpoint `/todoitems` that adds data to the in-memory database:

C#

```
app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});
```

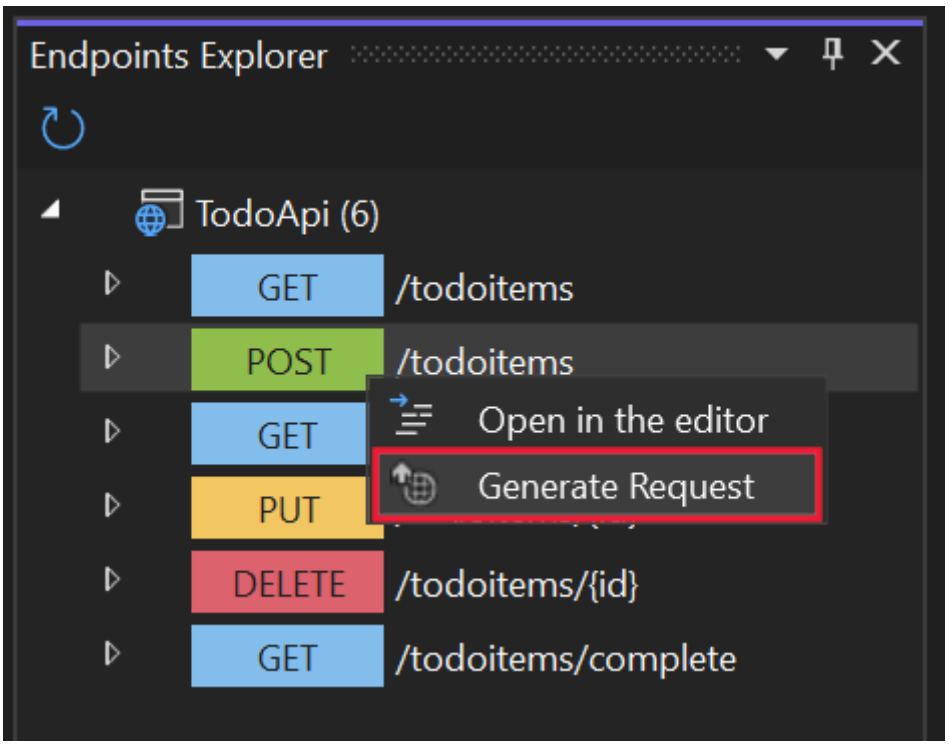
Run the app. The browser displays a 404 error because there's no longer a `/` endpoint.

The POST endpoint will be used to add data to the app.

Visual Studio

- Select **View** > **Other Windows** > **Endpoints Explorer**.

- Right-click the **POST** endpoint and select **Generate request**.



A new file is created in the project folder named `TodoApi.http`, with contents similar to the following example:

```
@TodoApi_HostAddress = https://localhost:7031

Post {{TodoApi_HostAddress}}/todoitems

###
```

  - The first line creates a variable that is used for all of the endpoints.
  - The next line defines a POST request.
  - The triple hashtag (`###`) line is a request delimiter: what comes after it is for a different request.

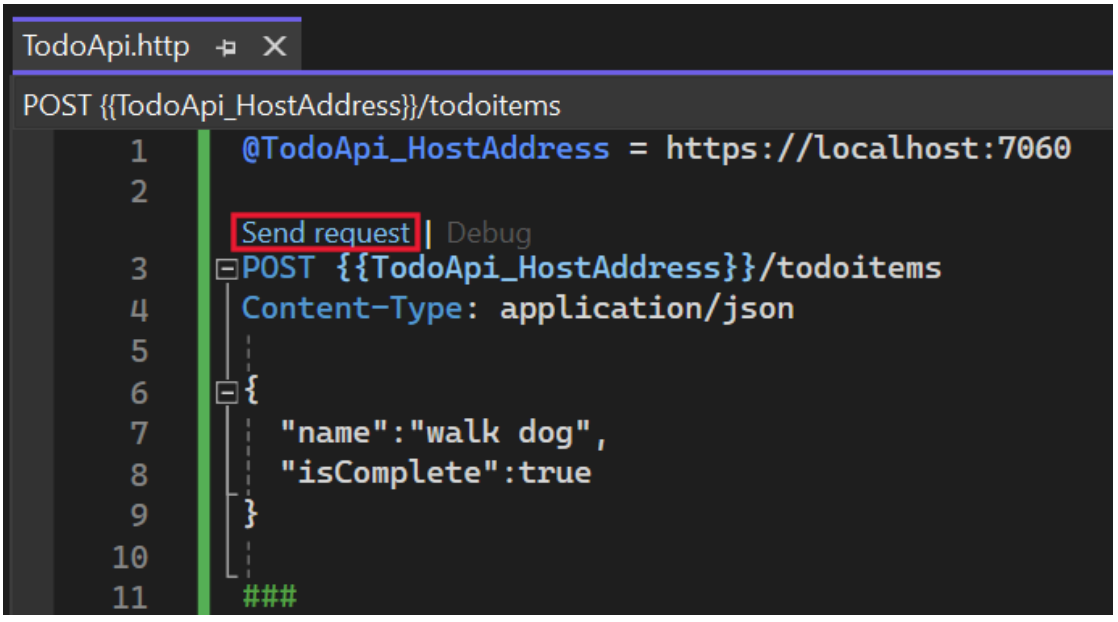
- The POST request needs headers and a body. To define those parts of the request, add the following lines immediately after the POST request line:

```
Content-Type: application/json

{
  "name":"walk dog",
  "isComplete":true
}
```

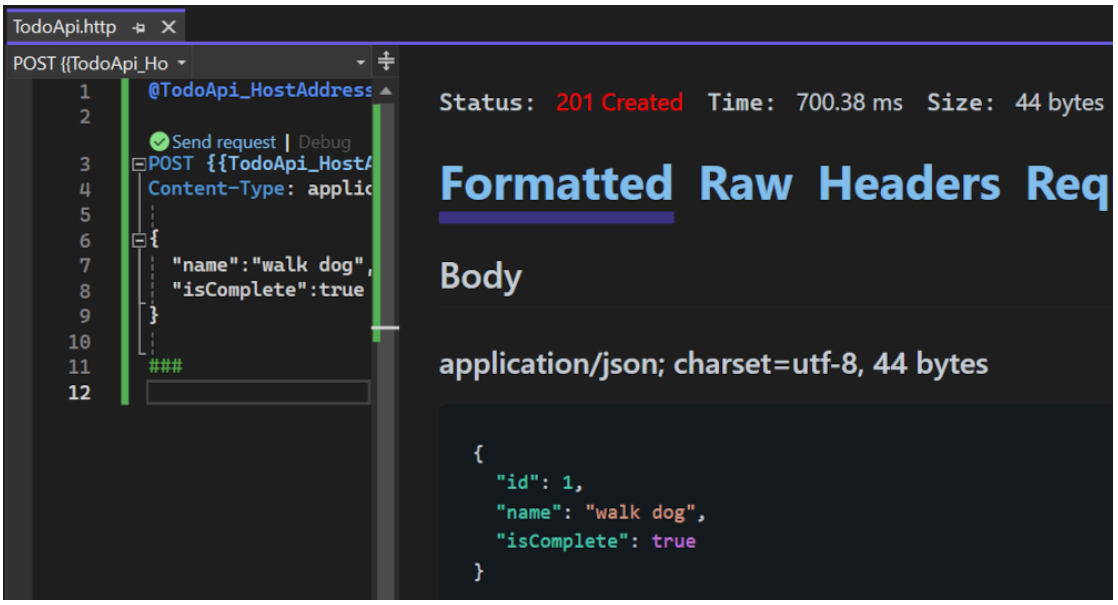
The preceding code adds a Content-Type header and a JSON request body. The TodoApi.http file should now look like the following example, but with your port number:

```
@TodoApi_HostAddress = https://localhost:7057

Post {{TodoApi_HostAddress}}/todoitems
Content-Type: application/json

{
  "name":"walk dog",
  "isComplete":true
}

###
```

- Run the app.
- Select the **Send request** link that is above the `POST` request line.



The POST request is sent to the app and the response is displayed in the **Response** pane.



Examine the GET endpoints

The sample app implements several GET endpoints by calling `MapGet`:

Expand table

API	Description	Request body	Response body
`GET /todoitems`	Get all to-do items	None	Array of to-do items
`GET /todoitems/complete`	Get all completed to-do items	None	Array of to-do items
`GET /todoitems/{id}`	Get an item by ID	None	To-do item

C#

```
app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

Test the GET endpoints

Visual Studio

Test the app by calling the `GET` endpoints from a browser or by using **Endpoints Explorer**. The following steps are for **Endpoints Explorer**.

- In **Endpoints Explorer**, right-click the first **GET** endpoint, and select **Generate request**.

  The following content is added to the `TodoApi.http` file:

  ```
  Get {{TodoApi_HostAddress}}/todoitems

  ###
  ```

- Select the **Send request** link that is above the new `GET` request line.

  The GET request is sent to the app and the response is displayed in the **Response** pane.

- The response body is similar to the following JSON:

  JSON

  ```
  [
    {
      "id": 1,
      "name": "walk dog",
      "isComplete": true
    }
  ]
  ```

- In **Endpoints Explorer**, right-click the `/todoitems/{id}` **GET** endpoint and select **Generate request**. The following content is added to the `TodoApi.http` file:

  ```
  GET {{TodoApi_HostAddress}}/todoitems/{id}

  ###
  ```

- Replace `{id}` with `1`.

- Select the **Send request** link that is above the new GET request line.

  The GET request is sent to the app and the response is displayed in the **Response** pane.

- The response body is similar to the following JSON:

  ```
  JSON

  {
    "id": 1,
    "name": "walk dog",
    "isComplete": true
  }
  ```

This app uses an in-memory database. If the app is restarted, the GET request doesn't return any data. If no data is returned, POST data to the app and try the GET request again.

Return values

ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

The return types can represent a wide range of HTTP status codes. For example, `GET /todoitems/{id}` can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

Examine the PUT endpoint

The sample app implements a single PUT endpoint using `MapPut`:

```
C#

app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
```

```
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});
```

This method is similar to the `MapPost` method, except it uses HTTP PUT. A successful response returns 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

Test the PUT endpoint

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has `Id = 1` and set its name to `"feed fish"`.

Visual Studio

- In **Endpoints Explorer**, right-click the **PUT** endpoint, and select **Generate request**.

  The following content is added to the `TodoApi.http` file:

  ```
  Put {{TodoApi_HostAddress}}/todoitems/{id}

  ###
  ```

- In the PUT request line, replace `{id}` with `1`.
- Add the following lines immediately after the PUT request line:

  ```
  ```

```
Content-Type: application/json

{
  "name": "feed fish",
  "isComplete": false
}
```

The preceding code adds a Content-Type header and a JSON request body.

- Select the **Send request** link that is above the new PUT request line.

  The PUT request is sent to the app and the response is displayed in the **Response** pane. The response body is empty, and the status code is 204.

Examine and test the DELETE endpoint

The sample app implements a single DELETE endpoint using `MapDelete`:

C#

```
app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});
```

Visual Studio

- In **Endpoints Explorer**, right-click the **DELETE** endpoint and select **Generate request**.

  A DELETE request is added to `TodoApi.http`.

- Replace `{id}` in the DELETE request line with `1`. The DELETE request should look like the following example:

  ```
  
  ```

```
DELETE {{TodoApi_HostAddress}}/todoitems/1

###
```

- Select the **Send request** link for the DELETE request.

  The DELETE request is sent to the app and the response is displayed in the **Response** pane. The response body is empty, and the status code is 204.

Use the MapGroup API

The sample app code repeats the `todoitems` URL prefix each time it sets up an endpoint. APIs often have groups of endpoints with a common URL prefix, and the MapGroup method is available to help organize such groups. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata.

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

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", async (TodoDb db) =>
    await db.Todos.ToListAsync());

todoItems.MapGet("/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

todoItems.MapGet("/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

```
todoItems.MapPost("/", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

todoItems.MapPut("/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

todoItems.MapDelete("/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});

app.Run();
```

The preceding code has the following changes:

- Adds `var todoItems = app.MapGroup("/todoitems");` to set up the group using the URL prefix `/todoitems`.
- Changes all the `app.Map<HttpVerb>` methods to `todoItems.Map<HttpVerb>`.
- Removes the URL prefix `/todoitems` from the `Map<HttpVerb>` method calls.

Test the endpoints to verify that they work the same.

Use the TypedResults API

Returning TypedResults rather than Results has several advantages, including testability and automatically returning the response type metadata for OpenAPI to describe the

endpoint. For more information, see TypedResults vs Results.

The `Map<HttpVerb>` methods can call route handler methods instead of using lambdas. To see an example, update *Program.cs* with the following code:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);

app.Run();

static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(todo)
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(Todo todo, TodoDb db)
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();
```

```
    return TypedResults.Created($"/todoitems/{todo.Id}", todo);
}

static async Task<IResult> UpdateTodo(int id, Todo inputTodo, TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

The `Map<HttpVerb>` code now calls methods instead of lambdas:

C#

```
var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);
```

These methods return objects that implement IResult and are defined by TypedResults:

C#

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}
```

```
static async Task<IResult> GetCompleteTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(todo)
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(Todo todo, TodoDb db)
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return TypedResults.Created($"/todoitems/{todo.Id}", todo);
}

static async Task<IResult> UpdateTodo(int id, Todo inputTodo, TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

Unit tests can call these methods and test that they return the correct type. For example, if the method is `GetAllTodos`:

```
C#
```

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}
```

Unit test code can verify that an object of type Ok<Todo[]> is returned from the handler method. For example:

C#

```
public async Task GetAllTodos_ReturnsOkOfTodosResult()
{
    // Arrange
    var db = CreateDbContext();

    // Act
    var result = await TodosApi.GetAllTodos(db);

    // Assert: Check for the correct returned type
    Assert.IsType<Ok<Todo[]>>(result);
}
```

Prevent over-posting

Currently the sample app exposes the entire `Todo` object. Production apps In production applications, a subset of the model is often used to restrict the data that can be input and returned. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO can be used to:

- Prevent over-posting.
- Hide properties that clients aren't supposed to view.
- Omit some properties to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `Todo` class to include a secret field:

C#

```
public class Todo
{
    public int Id { get; set; }
```

```
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
    public string? Secret { get; set; }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a file named `TodoItemDTO.cs` with the following code:

C#

```
public class TodoItemDTO
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }

    public TodoItemDTO() { }
    public TodoItemDTO(Todo todoItem) =>
    (Id, Name, IsComplete) = (todoItem.Id, todoItem.Name,
todoItem.IsComplete);
}
```

Replace the contents of the `Program.cs` file with the following code to use this DTO model:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

RouteGroupBuilder todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);
```

```
app.Run();

static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Select(x => new
TodoItemDTO(x)).ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db) {
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).Select(x => new TodoItemDTO(x)).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(new TodoItemDTO(todo))
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(TodoItemDTO todoItemDTO, TodoDb
db)
{
    var todoItem = new Todo
    {
        IsComplete = todoItemDTO.IsComplete,
        Name = todoItemDTO.Name
    };

    db.Todos.Add(todoItem);
    await db.SaveChangesAsync();

    todoItemDTO = new TodoItemDTO(todoItem);

    return TypedResults.Created($"/todoitems/{todoItem.Id}",
todoItemDTO);
}

static async Task<IResult> UpdateTodo(int id, TodoItemDTO todoItemDTO,
TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = todoItemDTO.Name;
    todo.IsComplete = todoItemDTO.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}
```

```
static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

Verify you can post and get all fields except the secret field.

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

- Configure JSON serialization options.
- Handle errors and exceptions: The developer exception page is enabled by default in the development environment for minimal API apps. For information about how to handle errors and exceptions, see Handle errors in ASP.NET Core APIs.
- For an example of testing a minimal API app, see this GitHub sample.
- OpenAPI support in minimal APIs.
- Quickstart: Publish to Azure.
- Organizing ASP.NET Core Minimal APIs.

Learn more

See Minimal APIs quick reference

Minimal APIs quick reference

Article • 08/07/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This document:

- Provides a quick reference for minimal APIs.
- Is intended for experienced developers. For an introduction, see Tutorial: Create a minimal API with ASP.NET Core.

The minimal APIs consist of:

- WebApplication and WebApplicationBuilder
- Route Handlers

`WebApplication`

The following code is generated by an ASP.NET Core template:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

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

app.Run();
```

The preceding code can be created via `dotnet new web` on the command line or selecting the Empty Web template in Visual Studio.

The following code creates a WebApplication (`app`) without explicitly creating a WebApplicationBuilder:

C#

```
var app = WebApplication.Create(args);

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

app.Run();
```

WebApplication.Create initializes a new instance of the WebApplication class with preconfigured defaults.

Working with ports

When a web app is created with Visual Studio or `dotnet new`, a `Properties/launchSettings.json` file is created that specifies the ports the app responds to. In the port setting samples that follow, running the app from Visual Studio returns an error dialog `Unable to connect to web server 'AppName'`. Visual Studio returns an error because it's expecting the port specified in `Properties/launchSettings.json`, but the app is using the port specified by `app.Run("http://localhost:3000")`. Run the following port changing samples from the command line.

The following sections set the port the app responds to.

C#

```
var app = WebApplication.Create(args);

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

app.Run("http://localhost:3000");
```

In the preceding code, the app responds to port `3000`.

Multiple ports

In the following code, the app responds to port `3000` and `4000`.

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://localhost:3000");
app.Urls.Add("http://localhost:4000");

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

```
app.Run();
```

Set the port from the command line

The following command makes the app respond to port `7777`:

.NET CLI

```
dotnet run --urls="https://localhost:7777"
```

If the Kestrel endpoint is also configured in the `appsettings.json` file, the `appsettings.json` file specified URL is used. For more information, see Kestrel endpoint configuration

Read the port from environment

The following code reads the port from the environment:

C#

```
var app = WebApplication.Create(args);

var port = Environment.GetEnvironmentVariable("PORT") ?? "3000";

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

app.Run($"http://localhost:{port}");
```

The preferred way to set the port from the environment is to use the `ASPNETCORE_URLS` environment variable, which is shown in the following section.

Set the ports via the ASPNETCORE_URLS environment variable

The `ASPNETCORE_URLS` environment variable is available to set the port:

```
ASPNETCORE_URLS=http://localhost:3000
```

`ASPNETCORE_URLS` supports multiple URLs:

```
ASPNETCORE_URLS=http://localhost:3000;https://localhost:5000
```

Listen on all interfaces

The following samples demonstrate listening on all interfaces

http://*:3000

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://*:3000");

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

app.Run();
```

http://+:3000

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://+:3000");

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

app.Run();
```

`http://0.0.0.0:3000`

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://0.0.0.0:3000");

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

app.Run();
```

Listen on all interfaces using ASPNETCORE_URLS

The preceding samples can use `ASPNETCORE_URLS`

```
ASPNETCORE_URLS=http://*:3000;https://+:5000;http://0.0.0.0:5005
```

Listen on all interfaces using ASPNETCORE_HTTPS_PORTS

The preceding samples can use `ASPNETCORE_HTTPS_PORTS` and `ASPNETCORE_HTTP_PORTS`.

```
ASPNETCORE_HTTP_PORTS=3000;5005
ASPNETCORE_HTTPS_PORTS=5000
```

For more information, see Configure endpoints for the ASP.NET Core Kestrel web server

Specify HTTPS with development certificate

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("https://localhost:3000");

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

app.Run();
```

For more information on the development certificate, see Trust the ASP.NET Core HTTPS development certificate on Windows and macOS.

Specify HTTPS using a custom certificate

The following sections show how to specify the custom certificate using the `appsettings.json` file and via configuration.

Specify the custom certificate with `appsettings.json`

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Kestrel": {
    "Certificates": {
      "Default": {
        "Path": "cert.pem",
        "KeyPath": "key.pem"
      }
    }
  }
}
```

Specify the custom certificate via configuration

C#

```
var builder = WebApplication.CreateBuilder(args);

// Configure the cert and the key
builder.Configuration["Kestrel:Certificates:Default:Path"] = "cert.pem";
builder.Configuration["Kestrel:Certificates:Default:KeyPath"] = "key.pem";

var app = builder.Build();

app.Urls.Add("https://localhost:3000");

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

app.Run();
```

Use the certificate APIs

C#

```
using System.Security.Cryptography.X509Certificates;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(options =>
{
    options.ConfigureHttpsDefaults(httpsOptions =>
    {
        var certPath = Path.Combine(builder.Environment.ContentRootPath,
```

```
"cert.pem");
        var keyPath = Path.Combine(builder.Environment.ContentRootPath,
"key.pem");

        httpsOptions.ServerCertificate =
X509Certificate2.CreateFromPemFile(certPath,
                                           keyPath);
    });
});

var app = builder.Build();

app.Urls.Add("https://localhost:3000");

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

app.Run();
```

Read the environment

C#

```
var app = WebApplication.Create(args);

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/oops");
}

app.MapGet("/", () => "Hello World");
app.MapGet("/oops", () => "Oops! An error happened.");

app.Run();
```

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

Configuration

The following code reads from the configuration system:

C#

```
var app = WebApplication.Create(args);

var message = app.Configuration["HelloKey"] ?? "Config failed!";

app.MapGet("/", () => message);
```

```
app.Run();
```

For more information, see Configuration in ASP.NET Core

Logging

The following code writes a message to the log on application startup:

C#

```
var app = WebApplication.Create(args);

app.Logger.LogInformation("The app started");

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

app.Run();
```

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

Access the Dependency Injection (DI) container

The following code shows how to get services from the DI container during application startup:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddScoped<SampleService>();

var app = builder.Build();

app.MapControllers();

using (var scope = app.Services.CreateScope())
{
    var sampleService =
scope.ServiceProvider.GetRequiredService<SampleService>();
    sampleService.DoSomething();
}

app.Run();
```

The following code shows how to access keys from the DI container using the [FromKeyedServices] attribute:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");

var app = builder.Build();

app.MapGet("/big", ([FromKeyedServices("big")] ICache bigCache) =>
bigCache.Get("date"));

app.MapGet("/small", ([FromKeyedServices("small")] ICache smallCache) =>
smallCache.Get("date"));

app.Run();

public interface ICache
{
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}
```

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

WebApplicationBuilder

This section contains sample code using WebApplicationBuilder.

Change the content root, application name, and environment

The following code sets the content root, application name, and environment:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    ApplicationName = typeof(Program).Assembly.FullName,
    ContentRootPath = Directory.GetCurrentDirectory(),
    EnvironmentName = Environments.Staging,
    WebRootPath = "customwwwroot"
});

Console.WriteLine($"Application Name:
{builder.Environment.ApplicationName}");
Console.WriteLine($"Environment Name:
{builder.Environment.EnvironmentName}");
Console.WriteLine($"ContentRoot Path:
{builder.Environment.ContentRootPath}");
Console.WriteLine($"WebRootPath: {builder.Environment.WebRootPath}");

var app = builder.Build();
```

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults.

For more information, see ASP.NET Core fundamentals overview

Change the content root, app name, and environment by using environment variables or command line

The following table shows the environment variable and command-line argument used to change the content root, app name, and environment:

Expand table

feature	Environment variable	Command-line argument
Application name	ASPNETCORE_APPLICATIONNAME	--applicationName
Environment name	ASPNETCORE_ENVIRONMENT	--environment
Content root	ASPNETCORE_CONTENTROOT	--contentRoot

Add configuration providers

The following sample adds the INI configuration provider:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddIniFile("appsettings.ini");

var app = builder.Build();
```

For detailed information, see File configuration providers in Configuration in ASP.NET Core.

Read configuration

By default the WebApplicationBuilder reads configuration from multiple sources, including:

- `appSettings.json` and `appSettings.{environment}.json`
- Environment variables
- The command line

For a complete list of configuration sources read, see Default configuration in Configuration in ASP.NET Core.

The following code reads `HelloKey` from configuration and displays the value at the `/` endpoint. If the configuration value is null, "Hello" is assigned to `message`:

C#

```
var builder = WebApplication.CreateBuilder(args);

var message = builder.Configuration["HelloKey"] ?? "Hello";

var app = builder.Build();

app.MapGet("/", () => message);

app.Run();
```

Read the environment

C#

```
var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsDevelopment())
{
    Console.WriteLine($"Running in development.");
```

```
}

var app = builder.Build();

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

app.Run();
```

Add logging providers

C#

```
var builder = WebApplication.CreateBuilder(args);

// Configure JSON logging to the console.
builder.Logging.AddJsonConsole();

var app = builder.Build();

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

app.Run();
```

Add services

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add the memory cache services.
builder.Services.AddMemoryCache();

// Add a custom scoped service.
builder.Services.AddScoped<ITodoRepository, TodoRepository>();
var app = builder.Build();
```

Customize the IHostBuilder

Existing extension methods on IHostBuilder can be accessed using the Host property:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Wait 30 seconds for graceful shutdown.
builder.Host.ConfigureHostOptions(o => o.ShutdownTimeout =
TimeSpan.FromSeconds(30));
```

```
var app = builder.Build();

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

app.Run();
```

Customize the IWebHostBuilder

Extension methods on IWebHostBuilder can be accessed using the WebApplicationBuilder.WebHost property.

C#

```
var builder = WebApplication.CreateBuilder(args);

// Change the HTTP server implemenation to be HTTP.sys based
builder.WebHost.UseHttpSys();

var app = builder.Build();

app.MapGet("/", () => "Hello HTTP.sys");

app.Run();
```

Change the web root

By default, the web root is relative to the content root in the `wwwroot` folder. Web root is where the static files middleware looks for static files. Web root can be changed with `WebHostOptions`, the command line, or with the UseWebRoot method:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    // Look for static files in webroot
    WebRootPath = "webroot"
});

var app = builder.Build();

app.Run();
```

Custom dependency injection (DI) container

The following example uses Autofac:

```C#
var builder = WebApplication.CreateBuilder(args);

builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());

// Register services directly with Autofac here. Don't
// call builder.Populate(), that happens in AutofacServiceProviderFactory.
builder.Host.ConfigureContainer<ContainerBuilder>(builder =>
builder.RegisterModule(new MyApplicationModule()));

var app = builder.Build();
```

Add Middleware

Any existing ASP.NET Core middleware can be configured on the `WebApplication`:

```C#
var app = WebApplication.Create(args);

// Setup the file server to serve static files.
app.UseFileServer();

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

app.Run();
```

For more information, see ASP.NET Core Middleware

Developer exception page

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults. The developer exception page is enabled in the preconfigured defaults. When the following code is run in the development environment, navigating to `/` renders a friendly page that shows the exception.

```C#
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () =>
{
```

```
    throw new InvalidOperationException("Oops, the '/' route has thrown an
exception.");
});

app.Run();
```

ASP.NET Core Middleware

The following table lists some of the middleware frequently used with minimal APIs.

Expand table

Middleware	Description	API
Authentication	Provides authentication support.	UseAuthentication
Authorization	Provides authorization support.	UseAuthorization
CORS	Configures Cross-Origin Resource Sharing.	UseCors
Exception Handler	Globally handles exceptions thrown by the middleware pipeline.	UseExceptionHandler
Forwarded Headers	Forwards proxied headers onto the current request.	UseForwardedHeaders
HTTPS Redirection	Redirects all HTTP requests to HTTPS.	UseHttpsRedirection
HTTP Strict Transport Security (HSTS)	Security enhancement middleware that adds a special response header.	UseHsts
Request Logging	Provides support for logging HTTP requests and responses.	UseHttpLogging
Request Timeouts	Provides support for configuring request timeouts, global default and per endpoint.	`UseRequestTimeouts`
W3C Request Logging	Provides support for logging HTTP requests and responses in the W3C format.	UseW3CLogging
Response Caching	Provides support for caching responses.	UseResponseCaching
Response Compression	Provides support for compressing responses.	UseResponseCompression
Session	Provides support for managing user sessions.	UseSession

Middleware	Description	API
Static Files	Provides support for serving static files and directory browsing.	UseStaticFiles, UseFileServer
WebSockets	Enables the WebSockets protocol.	UseWebSockets

The following sections cover request handling: routing, parameter binding, and responses.

Routing

A configured `WebApplication` supports `Map{Verb}` and MapMethods where `{Verb}` is a camel-cased HTTP method like `Get`, `Post`, `Put` or `Delete`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "This is a GET");
app.MapPost("/", () => "This is a POST");
app.MapPut("/", () => "This is a PUT");
app.MapDelete("/", () => "This is a DELETE");

app.MapMethods("/options-or-head", new[] { "OPTIONS", "HEAD" },
                           () => "This is an options or head request ");

app.Run();
```

The Delegate arguments passed to these methods are called "route handlers".

Route Handlers

Route handlers are methods that execute when the route matches. Route handlers can be a lambda expression, a local function, an instance method or a static method. Route handlers can be synchronous or asynchronous.

Lambda expression

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
```

```
app.MapGet("/inline", () => "This is an inline lambda");

var handler = () => "This is a lambda variable";

app.MapGet("/", handler);

app.Run();
```

Local function

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

string LocalFunction() => "This is local function";

app.MapGet("/", LocalFunction);

app.Run();
```

Instance method

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

var handler = new HelloHandler();

app.MapGet("/", handler.Hello);

app.Run();

class HelloHandler
{
    public string Hello()
    {
        return "Hello Instance method";
    }
}
```

Static method

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", HelloHandler.Hello);

app.Run();

class HelloHandler
{
    public static string Hello()
    {
        return "Hello static method";
    }
}
```

Endpoint defined outside of `Program.cs`

Minimal APIs don't have to be located in `Program.cs`.

`Program.cs`

C#

```
using MinAPISeparateFile;

var builder = WebApplication.CreateSlimBuilder(args);

var app = builder.Build();

TodoEndpoints.Map(app);

app.Run();
```

`TodoEndpoints.cs`

C#

```
namespace MinAPISeparateFile;

public static class TodoEndpoints
{
    public static void Map(WebApplication app)
    {
        app.MapGet("/", async context =>
        {
            // Get all todo items
            await context.Response.WriteAsJsonAsync(new { Message = "All
todo items" });
        });
```

```
        app.MapGet("/{id}", async context =>
        {
            // Get one todo item
            await context.Response.WriteAsJsonAsync(new { Message = "One
todo item" });
        });
    }
}
```

See also Route groups later in this article.

Named endpoints and link generation

Endpoints can be given names in order to generate URLs to the endpoint. Using a named endpoint avoids having to hard code paths in an app:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/hello", () => "Hello named route")
   .WithName("hi");

app.MapGet("/", (LinkGenerator linker) =>
        $"The link to the hello route is {linker.GetPathByName("hi", values:
null)}");

app.Run();
```

The preceding code displays `The link to the hello route is /hello` from the `/` endpoint.

NOTE: Endpoint names are case sensitive.

Endpoint names:

- Must be globally unique.
- Are used as the OpenAPI operation id when OpenAPI support is enabled. For more information, see OpenAPI.

Route Parameters

Route parameters can be captured as part of the route pattern definition:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/users/{userId}/books/{bookId}",
    (int userId, int bookId) => $"The user id is {userId} and book id is
{bookId}");

app.Run();
```

The preceding code returns `The user id is 3 and book id is 7` from the URI `/users/3/books/7`.

The route handler can declare the parameters to capture. When a request is made to a route with parameters declared to capture, the parameters are parsed and passed to the handler. This makes it easy to capture the values in a type safe way. In the preceding code, `userId` and `bookId` are both `int`.

In the preceding code, if either route value cannot be converted to an `int`, an exception is thrown. The GET request `/users/hello/books/3` throws the following exception:

```
BadHttpRequestException: Failed to bind parameter "int userId" from "hello".
```

Wildcard and catch all routes

The following catch all route returns `Routing to hello` from the `/posts/hello' endpoint:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/posts/{*rest}", (string rest) => $"Routing to {rest}");

app.Run();
```

Route constraints

Route constraints constrain the matching behavior of a route.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/todos/{id:int}", (int id) => db.Todos.Find(id));
```

```
app.MapGet("/todos/{text}", (string text) => db.Todos.Where(t =>
t.Text.Contains(text));
app.MapGet("/posts/{slug:regex(^[a-z0-9_-]+$)}", (string slug) => $"Post
{slug}");

app.Run();
```

The following table demonstrates the preceding route templates and their behavior:

Expand table

Route Template	Example Matching URI
`/todos/{id:int}`	`/todos/1`
`/todos/{text}`	`/todos/something`
`/posts/{slug:regex(^[a-z0-9_-]+$)}`	`/posts/mypost`

For more information, see Route constraint reference in Routing in ASP.NET Core.

Route groups

The MapGroup extension method helps organize groups of endpoints with a common prefix. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata which add endpoint metadata.

For example, the following code creates two similar groups of endpoints:

C#

```
app.MapGroup("/public/todos")
    .MapTodosApi()
    .WithTags("Public");

app.MapGroup("/private/todos")
    .MapTodosApi()
    .WithTags("Private")
    .AddEndpointFilterFactory(QueryPrivateTodos)
    .RequireAuthorization();


EndpointFilterDelegate QueryPrivateTodos(EndpointFilterFactoryContext
factoryContext, EndpointFilterDelegate next)
{
    var dbContextIndex = -1;

    foreach (var argument in factoryContext.MethodInfo.GetParameters())
```

```
    {
        if (argument.ParameterType == typeof(TodoDb))
        {
            dbContextIndex = argument.Position;
            break;
        }
    }

    // Skip filter if the method doesn't have a TodoDb parameter.
    if (dbContextIndex < 0)
    {
        return next;
    }

    return async invocationContext =>
    {
        var dbContext = invocationContext.GetArgument<TodoDb>
(dbContextIndex);
        dbContext.IsPrivate = true;

        try
        {
            return await next(invocationContext);
        }
        finally
        {
            // This should only be relevant if you're pooling or otherwise
reusing the DbContext instance.
            dbContext.IsPrivate = false;
        }
    };
}
```

C#

```
public static RouteGroupBuilder MapTodosApi(this RouteGroupBuilder group)
{
    group.MapGet("/", GetAllTodos);
    group.MapGet("/{id}", GetTodo);
    group.MapPost("/", CreateTodo);
    group.MapPut("/{id}", UpdateTodo);
    group.MapDelete("/{id}", DeleteTodo);

    return group;
}
```

In this scenario, you can use a relative address for the `Location` header in the `201 Created` result:

C#

```
public static async Task<Created<Todo>> CreateTodo(Todo todo, TodoDb
database)
{
    await database.AddAsync(todo);
    await database.SaveChangesAsync();

    return TypedResults.Created($"{todo.Id}", todo);
}
```

The first group of endpoints will only match requests prefixed with `/public/todos` and are accessible without any authentication. The second group of endpoints will only match requests prefixed with `/private/todos` and require authentication.

The `QueryPrivateTodos` endpoint filter factory is a local function that modifies the route handler's `TodoDb` parameters to allow to access and store private todo data.

Route groups also support nested groups and complex prefix patterns with route parameters and constraints. In the following example, and route handler mapped to the `user` group can capture the `{org}` and `{group}` route parameters defined in the outer group prefixes.

The prefix can also be empty. This can be useful for adding endpoint metadata or filters to a group of endpoints without changing the route pattern.

C#

```
var all = app.MapGroup("").WithOpenApi();
var org = all.MapGroup("{org}");
var user = org.MapGroup("{user}");
user.MapGet("", (string org, string user) => $"{org}/{user}");
```

Adding filters or metadata to a group behaves the same way as adding them individually to each endpoint before adding any extra filters or metadata that may have been added to an inner group or specific endpoint.

C#

```
var outer = app.MapGroup("/outer");
var inner = outer.MapGroup("/inner");

inner.AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("/inner group filter");
    return next(context);
});

outer.AddEndpointFilter((context, next) =>
```

```
{
    app.Logger.LogInformation("/outer group filter");
    return next(context);
});

inner.MapGet("/", () => "Hi!").AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("MapGet filter");
    return next(context);
});
```

In the above example, the outer filter will log the incoming request before the inner filter even though it was added second. Because the filters were applied to different groups, the order they were added relative to each other does not matter. The order filters are added does matter if applied to the same group or specific endpoint.

A request to `/outer/inner/` will log the following:

.NET CLI

```
/outer group filter
/inner group filter
MapGet filter
```

Parameter binding

Parameter binding is the process of converting request data into strongly typed parameters that are expressed by route handlers. A binding source determines where parameters are bound from. Binding sources can be explicit or inferred based on HTTP method and parameter type.

Supported binding sources:

- Route values
- Query string
- Header
- Body (as JSON)
- Form values
- Services provided by dependency injection
- Custom

The following `GET` route handler uses some of these parameter binding sources:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Added as service
builder.Services.AddSingleton<Service>();

var app = builder.Build();

app.MapGet("/{id}", (int id,
                     int page,
                     [FromHeader(Name = "X-CUSTOM-HEADER")] string
customHeader,
                     Service service) => { });

class Service { }
```

The following table shows the relationship between the parameters used in the preceding example and the associated binding sources.

Expand table

Parameter	Binding Source
`id`	route value
`page`	query string
`customHeader`	header
`service`	Provided by dependency injection

The HTTP methods `GET`, `HEAD`, `OPTIONS`, and `DELETE` don't implicitly bind from body. To bind from body (as JSON) for these HTTP methods, bind explicitly with [FromBody] or read from the HttpRequest.

The following example POST route handler uses a binding source of body (as JSON) for the `person` parameter:

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapPost("/", (Person person) => { });

record Person(string Name, int Age);
```