The parameters in the preceding examples are all bound from request data automatically. To demonstrate the convenience that parameter binding provides, the following route handlers show how to read request data directly from the request:

C#

```
app.MapGet("/{id}", (HttpRequest request) =>
{
    var id = request.RouteValues["id"];
    var page = request.Query["page"];
    var customHeader = request.Headers["X-CUSTOM-HEADER"];

    // ...
});

app.MapPost("/", async (HttpRequest request) =>
{
    var person = await request.ReadFromJsonAsync<Person>();

    // ...
});
```

Explicit Parameter Binding

Attributes can be used to explicitly declare where parameters are bound from.

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

// Added as service
builder.Services.AddSingleton<Service>();

var app = builder.Build();


app.MapGet("/{id}", ([FromRoute] int id,
                     [FromQuery(Name = "p")] int page,
                     [FromServices] Service service,
                     [FromHeader(Name = "Content-Type")] string contentType)
                     => {});

class Service { }

record Person(string Name, int Age);
```

Expand table

Parameter	Binding Source
`id`	route value with the name `id`
`page`	query string with the name `"p"`
`service`	Provided by dependency injection
`contentType`	header with the name `"Content-Type"`

Explicit binding from form values

The [FromForm] attribute binds form values:

C#

```
app.MapPost("/todos", async ([FromForm] string name,
    [FromForm] Visibility visibility, IFormFile? attachment, TodoDb db) =>
{
    var todo = new Todo
    {
        Name = name,
        Visibility = visibility
    };

    if (attachment is not null)
    {
        var attachmentName = Path.GetRandomFileName();

        using var stream = File.Create(Path.Combine("wwwroot",
attachmentName));
        await attachment.CopyToAsync(stream);
    }

    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Ok();
});

// Remaining code removed for brevity.
```

An alternative is to use the [AsParameters] attribute with a custom type that has properties annotated with `[FromForm]`. For example, the following code binds from form values to properties of the `NewTodoRequest` record struct:

C#

```
app.MapPost("/ap/todos", async ([AsParameters] NewTodoRequest request,
TodoDb db) =>
{
    var todo = new Todo
    {
        Name = request.Name,
        Visibility = request.Visibility
    };

    if (request.Attachment is not null)
    {
        var attachmentName = Path.GetRandomFileName();

        using var stream = File.Create(Path.Combine("wwwroot",
attachmentName));
        await request.Attachment.CopyToAsync(stream);

        todo.Attachment = attachmentName;
    }

    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Ok();
});

// Remaining code removed for brevity.
```

C#

```
public record struct NewTodoRequest([FromForm] string Name,
    [FromForm] Visibility Visibility, IFormFile? Attachment);
```

For more information, see the section on AsParameters later in this article.

The complete sample code is in the AspNetCore.Docs.Samples repository.

Secure binding from IFormFile and IFormFileCollection

Complex form binding is supported using IFormFile and IFormFileCollection using the [FromForm]:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;
using Microsoft.AspNetCore.Mvc;

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

```
builder.Services.AddAntiforgery();

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

// Generate a form with an anti-forgery token and an /upload endpoint.
app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = MyUtils.GenerateHtmlForm(token.FormFieldName,
token.RequestToken!);
    return Results.Content(html, "text/html");
});

app.MapPost("/upload", async Task<Results<Ok<string>, BadRequest<string>>>
    ([FromForm] FileUploadForm fileUploadForm, HttpContext context,
                                                IAntiforgery antiforgery) =>
{
    await MyUtils.SaveFileWithName(fileUploadForm.FileDocument!,
              fileUploadForm.Name!, app.Environment.ContentRootPath);
    return TypedResults.Ok($"Your file with the description:" +
        $" {fileUploadForm.Description} has been uploaded successfully");
});

app.Run();
```

Parameters bound to the request with `[FromForm]` include an antiforgery token. The antiforgery token is validated when the request is processed. For more information, see Antiforgery with Minimal APIs.

For more information, see Form binding in minimal APIs.

The complete sample code is in the AspNetCore.Docs.Samples repository.

Parameter binding with dependency injection

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

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IDateTime, SystemDateTime>();

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

```
app.MapGet("/",   (               IDateTime dateTime) => dateTime.Now);
app.MapGet("/fs", ([FromServices] IDateTime dateTime) => dateTime.Now);
app.Run();
```

Optional parameters

Parameters declared in route handlers are treated as required:

- If a request matches the route, the route handler only runs if all required parameters are provided in the request.
- Failure to provide all required parameters results in an error.

C#

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

app.MapGet("/products", (int pageNumber) => $"Requesting page
{pageNumber}");

app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	3 returned
`/products`	`BadHttpRequestException`: Required parameter "int pageNumber" wasn't provided from query string.
`/products/1`	HTTP 404 error, no matching route

To make `pageNumber` optional, define the type as optional or provide a default value:

C#

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

app.MapGet("/products", (int? pageNumber) => $"Requesting page {pageNumber
?? 1}");

string ListProducts(int pageNumber = 1) => $"Requesting page {pageNumber}";

app.MapGet("/products2", ListProducts);
```

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

Expand table

URI	result
`/products?pageNumber=3`	3 returned
`/products`	1 returned
`/products2`	1 returned

The preceding nullable and default value applies to all sources:

C#

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

app.MapPost("/products", (Product? product) => { });

app.Run();
```

The preceding code calls the method with a null product if no request body is sent.

NOTE: If invalid data is provided and the parameter is nullable, the route handler is ***not*** run.

C#

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

app.MapGet("/products", (int? pageNumber) => $"Requesting page {pageNumber
?? 1}");

app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	`3` returned
`/products`	`1` returned

URI	result
`/products?pageNumber=two`	`BadHttpRequestException`: Failed to bind parameter `"Nullable<int> pageNumber"` from "two".
`/products/two`	HTTP 404 error, no matching route

See the Binding Failures section for more information.

Special types

The following types are bound without explicit attributes:

- HttpContext: The context which holds all the information about the current HTTP request or response:

  C#

  ```
  app.MapGet("/", (HttpContext context) =>
  context.Response.WriteAsync("Hello World"));
  ```

- HttpRequest and HttpResponse: The HTTP request and HTTP response:

  C#

  ```
  app.MapGet("/", (HttpRequest request, HttpResponse response) =>
      response.WriteAsync($"Hello World {request.Query["name"]}"));
  ```

- CancellationToken: The cancellation token associated with the current HTTP request:

  C#

  ```
  app.MapGet("/", async (CancellationToken cancellationToken) =>
      await MakeLongRunningRequestAsync(cancellationToken));
  ```

- ClaimsPrincipal: The user associated with the request, bound from HttpContext.User:

  C#

  ```
  app.MapGet("/", (ClaimsPrincipal user) => user.Identity.Name);
  ```

Bind the request body as a `Stream` or `PipeReader`

The request body can bind as a Stream or PipeReader to efficiently support scenarios where the user has to process data and:

- Store the data to blob storage or enqueue the data to a queue provider.
- Process the stored data with a worker process or cloud function.

For example, the data might be enqueued to Azure Queue storage or stored in Azure Blob storage.

The following code implements a background queue:

C#

```
using System.Text.Json;
using System.Threading.Channels;

namespace BackgroundQueueService;

class BackgroundQueue : BackgroundService
{
    private readonly Channel<ReadOnlyMemory<byte>> _queue;
    private readonly ILogger<BackgroundQueue> _logger;

    public BackgroundQueue(Channel<ReadOnlyMemory<byte>> queue,
                           ILogger<BackgroundQueue> logger)
    {
        _queue = queue;
        _logger = logger;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        await foreach (var dataStream in
_queue.Reader.ReadAllAsync(stoppingToken))
        {
            try
            {
                var person = JsonSerializer.Deserialize<Person>
(dataStream.Span)!;
                _logger.LogInformation($"{person.Name} is {person.Age} " +
                                       $"years and from {person.Country}");
            }
            catch (Exception ex)
            {
                _logger.LogError(ex.Message);
            }
        }
    }
}
```

```
class Person
{
    public string Name { get; set; } = String.Empty;
    public int Age { get; set; }
    public string Country { get; set; } = String.Empty;
}
```

The following code binds the request body to a `Stream`:

C#

```
app.MapPost("/register", async (HttpRequest req, Stream body,
                                Channel<ReadOnlyMemory<byte>> queue) =>
{
    if (req.ContentLength is not null && req.ContentLength > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // We're not above the message size and we have a content length, or
    // we're a chunked request and we're going to read up to the
maxMessageSize + 1.
    // We add one to the message size so that we can detect when a chunked
request body
    // is bigger than our configured max.
    var readSize = (int?)req.ContentLength ?? (maxMessageSize + 1);

    var buffer = new byte[readSize];

    // Read at least that many bytes from the body.
    var read = await body.ReadAtLeastAsync(buffer, readSize,
throwOnEndOfStream: false);

    // We read more than the max, so this is a bad request.
    if (read > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // Attempt to send the buffer to the background queue.
    if (queue.Writer.TryWrite(buffer.AsMemory(0..read)))
    {
        return Results.Accepted();
    }

    // We couldn't accept the message since we're overloaded.
    return Results.StatusCode(StatusCodes.Status429TooManyRequests);
});
```

The following code shows the complete `Program.cs` file:

C#

```
using System.Threading.Channels;
using BackgroundQueueService;

var builder = WebApplication.CreateBuilder(args);
// The max memory to use for the upload endpoint on this instance.
var maxMemory = 500 * 1024 * 1024;

// The max size of a single message, staying below the default LOH size of
85K.
var maxMessageSize = 80 * 1024;

// The max size of the queue based on those restrictions
var maxQueueSize = maxMemory / maxMessageSize;

// Create a channel to send data to the background queue.
builder.Services.AddSingleton<Channel<ReadOnlyMemory<byte>>>((_) =>
                    Channel.CreateBounded<ReadOnlyMemory<byte>>
(maxQueueSize));

// Create a background queue service.
builder.Services.AddHostedService<BackgroundQueue>();
var app = builder.Build();

// curl --request POST 'https://localhost:<port>/register' --header
'Content-Type: application/json' --data-raw '{ "Name":"Samson", "Age": 23,
"Country":"Nigeria" }'
// curl --request POST "https://localhost:<port>/register" --header
"Content-Type: application/json" --data-raw "{ \"Name\":\"Samson\", \"Age\":
23, \"Country\":\"Nigeria\" }"
app.MapPost("/register", async (HttpRequest req, Stream body,
                                Channel<ReadOnlyMemory<byte>> queue) =>
{
    if (req.ContentLength is not null && req.ContentLength > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // We're not above the message size and we have a content length, or
    // we're a chunked request and we're going to read up to the
maxMessageSize + 1.
    // We add one to the message size so that we can detect when a chunked
request body
    // is bigger than our configured max.
    var readSize = (int?)req.ContentLength ?? (maxMessageSize + 1);

    var buffer = new byte[readSize];

    // Read at least that many bytes from the body.
    var read = await body.ReadAtLeastAsync(buffer, readSize,
throwOnEndOfStream: false);

    // We read more than the max, so this is a bad request.
```

```
    if (read > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // Attempt to send the buffer to the background queue.
    if (queue.Writer.TryWrite(buffer.AsMemory(0..read)))
    {
        return Results.Accepted();
    }

    // We couldn't accept the message since we're overloaded.
    return Results.StatusCode(StatusCodes.Status429TooManyRequests);
});

app.Run();
```

- When reading data, the `Stream` is the same object as `HttpRequest.Body`.
- The request body isn't buffered by default. After the body is read, it's not rewindable. The stream can't be read multiple times.
- The `Stream` and `PipeReader` aren't usable outside of the minimal action handler as the underlying buffers will be disposed or reused.

File uploads using IFormFile and IFormFileCollection

The following code uses IFormFile and IFormFileCollection to upload file:

C#

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

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

app.MapPost("/upload", async (IFormFile file) =>
{
    var tempFile = Path.GetTempFileName();
    app.Logger.LogInformation(tempFile);
    using var stream = File.OpenWrite(tempFile);
    await file.CopyToAsync(stream);
});

app.MapPost("/upload_many", async (IFormFileCollection myFiles) =>
{
    foreach (var file in myFiles)
    {
        var tempFile = Path.GetTempFileName();
        app.Logger.LogInformation(tempFile);
        using var stream = File.OpenWrite(tempFile);
        await file.CopyToAsync(stream);
```

```
    }
});

app.Run();
```

Authenticated file upload requests are supported using an Authorization header, a client certificate, or a cookie header.

Binding to forms with IFormCollection, IFormFile, and IFormFileCollection

Binding from form-based parameters using IFormCollection, IFormFile, and IFormFileCollection is supported. OpenAPI metadata is inferred for form parameters to support integration with Swagger UI.

The following code uploads files using inferred binding from the `IFormFile` type:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

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

string GetOrCreateFilePath(string fileName, string filesDirectory =
"uploadFiles")
{
    var directoryPath = Path.Combine(app.Environment.ContentRootPath,
filesDirectory);
    Directory.CreateDirectory(directoryPath);
    return Path.Combine(directoryPath, fileName);
}

async Task UploadFileWithName(IFormFile file, string fileSaveName)
{
    var filePath = GetOrCreateFilePath(fileSaveName);
    await using var fileStream = new FileStream(filePath, FileMode.Create);
    await file.CopyToAsync(fileStream);
}

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
      <html>
```

```
        <body>
          <form action="/upload" method="POST" enctype="multipart/form-
data">
            <input name="{token.FormFieldName}" type="hidden" value="
{token.RequestToken}"/>
            <input type="file" name="file" placeholder="Upload an image..."
accept=".jpg,

.jpeg, .png" />
            <input type="submit" />
          </form>
        </body>
      </html>
    """;

    return Results.Content(html, "text/html");
});

app.MapPost("/upload", async Task<Results<Ok<string>,
   BadRequest<string>>> (IFormFile file, HttpContext context, IAntiforgery
antiforgery) =>
{
    var fileSaveName = Guid.NewGuid().ToString("N") +
Path.GetExtension(file.FileName);
    await UploadFileWithName(file, fileSaveName);
    return TypedResults.Ok("File uploaded successfully!");
});

app.Run();
```

Warning: When implementing forms, the app ***must prevent*** Cross-Site Request Forgery (XSRF/CSRF) attacks. In the preceding code, the IAntiforgery service is used to prevent XSRF attacks by generating and validation an antiforgery token:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

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

string GetOrCreateFilePath(string fileName, string filesDirectory =
"uploadFiles")
{
    var directoryPath = Path.Combine(app.Environment.ContentRootPath,
filesDirectory);
    Directory.CreateDirectory(directoryPath);
    return Path.Combine(directoryPath, fileName);
```

```
}

async Task UploadFileWithName(IFormFile file, string fileSaveName)
{
    var filePath = GetOrCreateFilePath(fileSaveName);
    await using var fileStream = new FileStream(filePath, FileMode.Create);
    await file.CopyToAsync(fileStream);
}

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
      <html>
        <body>
          <form action="/upload" method="POST" enctype="multipart/form-
data">
            <input name="{token.FormFieldName}" type="hidden" value="
{token.RequestToken}"/>
            <input type="file" name="file" placeholder="Upload an image..."
accept=".jpg,

.jpeg, .png" />
            <input type="submit" />
          </form>
        </body>
      </html>
    """;

    return Results.Content(html, "text/html");
});

app.MapPost("/upload", async Task<Results<Ok<string>,
   BadRequest<string>>> (IFormFile file, HttpContext context, IAntiforgery
antiforgery) =>
{
    var fileSaveName = Guid.NewGuid().ToString("N") +
Path.GetExtension(file.FileName);
    await UploadFileWithName(file, fileSaveName);
    return TypedResults.Ok("File uploaded successfully!");
});

app.Run();
```

For more information on XSRF attacks, see Antiforgery with Minimal APIs

For more information, see Form binding in minimal APIs ↗;

Bind to collections and complex types from forms

Binding is supported for:

- Collections, for example List and Dictionary
- Complex types, for example, `Todo` or `Project`

The following code shows:

- A minimal endpoint that binds a multi-part form input to a complex object.
- How to use the antiforgery services to support the generation and validation of antiforgery tokens.

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
        <html><body>
           <form action="/todo" method="POST" enctype="multipart/form-data">
               <input name="{token.FormFieldName}"
                               type="hidden" value="{token.RequestToken}"
/>
               <input type="text" name="name" />
               <input type="date" name="dueDate" />
               <input type="checkbox" name="isCompleted" value="true" />
               <input type="submit" />
               <input name="isCompleted" type="hidden" value="false" />
           </form>
        </body></html>
    """;
    return Results.Content(html, "text/html");
});

app.MapPost("/todo", async Task<Results<Ok<Todo>, BadRequest<string>>>
               ([FromForm] Todo todo, HttpContext context, IAntiforgery
antiforgery) =>
{
    try
    {
        await antiforgery.ValidateRequestAsync(context);
        return TypedResults.Ok(todo);
    }
    catch (AntiforgeryValidationException e)
```

```
    {
        return TypedResults.BadRequest("Invalid antiforgery token");
    }
});

app.Run();

class Todo
{
    public string Name { get; set; } = string.Empty;
    public bool IsCompleted { get; set; } = false;
    public DateTime DueDate { get; set; } =
DateTime.Now.Add(TimeSpan.FromDays(1));
}
```

In the preceding code:

- The target parameter ***must*** be annotated with the [FromForm] attribute to disambiguate from parameters that should be read from the JSON body.
- Binding from complex or collection types is ***not*** supported for minimal APIs that are compiled with the Request Delegate Generator.
- The markup shows an additional hidden input with a name of `isCompleted` and a value of `false`. If the `isCompleted` checkbox is checked when the form is submitted, both values `true` and `false` are submitted as values. If the checkbox is unchecked, only the hidden input value `false` is submitted. The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

An example of the form data submitted to the preceding endpoint looks as follows:

txt

```
__RequestVerificationToken:
CfDJ8Bveip67DklJm5vI2PF2VOUZ594RC8kcGWpTnVV17zCLZi1yrs-
CSz426ZRRrQnEJ0gybB0AD7hTU-0EGJXDU-OaJaktgAtWLIaaEWMOWCkoxYYm-
9U9eLV7INSUrQ6yBHqdMEE_aJpD4AI72gYiCqc
name: Walk the dog
dueDate: 2024-04-06
isCompleted: true
isCompleted: false
```

Bind arrays and string values from headers and query strings

The following code demonstrates binding query strings to an array of primitive types, string arrays, and StringValues:

C#

```
// Bind query string values to a primitive type array.
// GET  /tags?q=1&q=2&q=3
app.MapGet("/tags", (int[] q) =>
                      $"tag1: {q[0]} , tag2: {q[1]}, tag3: {q[2]}");

// Bind to a string array.
// GET /tags2?names=john&names=jack&names=jane
app.MapGet("/tags2", (string[] names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3: {names[2]}");

// Bind to StringValues.
// GET /tags3?names=john&names=jack&names=jane
app.MapGet("/tags3", (StringValues names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3: {names[2]}");
```

Binding query strings or header values to an array of complex types is supported when the type has `TryParse` implemented. The following code binds to a string array and returns all the items with the specified tags:

C#

```
// GET /todoitems/tags?tags=home&tags=work
app.MapGet("/todoitems/tags", async (Tag[] tags, TodoDb db) =>
{
    return await db.Todos
        .Where(t => tags.Select(i => i.Name).Contains(t.Tag.Name))
        .ToListAsync();
});
```

The following code shows the model and the required `TryParse` implementation:

C#

```
public class Todo
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }

    // This is an owned entity.
    public Tag Tag { get; set; } = new();
}

[Owned]
public class Tag
```

```
{
    public string? Name { get; set; } = "n/a";

    public static bool TryParse(string? name, out Tag tag)
    {
        if (name is null)
        {
            tag = default!;
            return false;
        }

        tag = new Tag { Name = name };
        return true;
    }
}
```

The following code binds to an `int` array:

C#

```
// GET /todoitems/query-string-ids?ids=1&ids=3
app.MapGet("/todoitems/query-string-ids", async (int[] ids, TodoDb db) =>
{
    return await db.Todos
        .Where(t => ids.Contains(t.Id))
        .ToListAsync();
});
```

To test the preceding code, add the following endpoint to populate the database with `Todo` items:

C#

```
// POST /todoitems/batch
app.MapPost("/todoitems/batch", async (Todo[] todos, TodoDb db) =>
{
    await db.Todos.AddRangeAsync(todos);
    await db.SaveChangesAsync();

    return Results.Ok(todos);
});
```

Use a tool like HttpRepl to pass the following data to the previous endpoint:

C#

```
[
    {
        "id": 1,
        "name": "Have Breakfast",
```

```
        "isComplete": true,
        "tag": {
            "name": "home"
        }
    },
    {
        "id": 2,
        "name": "Have Lunch",
        "isComplete": true,
        "tag": {
            "name": "work"
        }
    },
    {
        "id": 3,
        "name": "Have Supper",
        "isComplete": true,
        "tag": {
            "name": "home"
        }
    },
    {
        "id": 4,
        "name": "Have Snacks",
        "isComplete": true,
        "tag": {
            "name": "N/A"
        }
    }
]
```

The following code binds to the header key `X-Todo-Id` and returns the `Todo` items with matching `Id` values:

C#

```
// GET /todoitems/header-ids
// The keys of the headers should all be X-Todo-Id with different values
app.MapGet("/todoitems/header-ids", async ([FromHeader(Name = "X-Todo-Id")]
int[] ids, TodoDb db) =>
{
    return await db.Todos
        .Where(t => ids.Contains(t.Id))
        .ToListAsync();
});
```

ⓘ Note

When binding a `string[]` from a query string, the absence of any matching query string value will result in an empty array instead of a null value.

Parameter binding for argument lists with [AsParameters]

AsParametersAttribute enables simple parameter binding to types and not complex or recursive model binding.

Consider the following code:

C#

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

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
var app = builder.Build();

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.Select(x => new TodoItemDTO(x)).ToListAsync());

app.MapGet("/todoitems/{id}",
                            async (int Id, TodoDb Db) =>
    await Db.Todos.FindAsync(Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
// Remaining code removed for brevity.
```

Consider the following `GET` endpoint:

C#

```
app.MapGet("/todoitems/{id}",
                            async (int Id, TodoDb Db) =>
    await Db.Todos.FindAsync(Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
```

The following `struct` can be used to replace the preceding highlighted parameters:

C#

```
struct TodoItemRequest
{
    public int Id { get; set; }
    public TodoDb Db { get; set; }
}
```

The refactored `GET` endpoint uses the preceding `struct` with the AsParameters attribute:

C#

```
app.MapGet("/ap/todoitems/{id}",
                                async ([AsParameters] TodoItemRequest
request) =>
    await request.Db.Todos.FindAsync(request.Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
```

The following code shows additional endpoints in the app:

C#

```
app.MapPost("/todoitems", async (TodoItemDTO Dto, TodoDb Db) =>
{
    var todoItem = new Todo
    {
        IsComplete = Dto.IsComplete,
        Name = Dto.Name
    };

    Db.Todos.Add(todoItem);
    await Db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todoItem.Id}", new
TodoItemDTO(todoItem));
});

app.MapPut("/todoitems/{id}", async (int Id, TodoItemDTO Dto, TodoDb Db) =>
{
    var todo = await Db.Todos.FindAsync(Id);

    if (todo is null) return Results.NotFound();

    todo.Name = Dto.Name;
    todo.IsComplete = Dto.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.Ok(new TodoItemDTO(todo));
    }

    return Results.NotFound();
});
```

The following classes are used to refactor the parameter lists:

C#

```
class CreateTodoItemRequest
{
    public TodoItemDTO Dto { get; set; } = default!;
    public TodoDb Db { get; set; } = default!;
}

class EditTodoItemRequest
{
    public int Id { get; set; }
    public TodoItemDTO Dto { get; set; } = default!;
    public TodoDb Db { get; set; } = default!;
}
```

The following code shows the refactored endpoints using `AsParameters` and the preceding `struct` and classes:

C#

```
app.MapPost("/ap/todoitems", async ([AsParameters] CreateTodoItemRequest
request) =>
{
    var todoItem = new Todo
    {
        IsComplete = request.Dto.IsComplete,
        Name = request.Dto.Name
    };

    request.Db.Todos.Add(todoItem);
    await request.Db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todoItem.Id}", new
TodoItemDTO(todoItem));
});
```

```
app.MapPut("/ap/todoitems/{id}", async ([AsParameters] EditTodoItemRequest
request) =>
{
    var todo = await request.Db.Todos.FindAsync(request.Id);

    if (todo is null) return Results.NotFound();

    todo.Name = request.Dto.Name;
    todo.IsComplete = request.Dto.IsComplete;

    await request.Db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/ap/todoitems/{id}", async ([AsParameters] TodoItemRequest
request) =>
{
    if (await request.Db.Todos.FindAsync(request.Id) is Todo todo)
    {
        request.Db.Todos.Remove(todo);
        await request.Db.SaveChangesAsync();
        return Results.Ok(new TodoItemDTO(todo));
    }

    return Results.NotFound();
});
```

The following record types can be used to replace the preceding parameters:

C#

```
record TodoItemRequest(int Id, TodoDb Db);
record CreateTodoItemRequest(TodoItemDTO Dto, TodoDb Db);
record EditTodoItemRequest(int Id, TodoItemDTO Dto, TodoDb Db);
```

Using a `struct` with `AsParameters` can be more performant than using a `record` type.

The complete sample code in the AspNetCore.Docs.Samples repository.

Custom Binding

There are two ways to customize parameter binding:

1. For route, query, and header binding sources, bind custom types by adding a static `TryParse` method for the type.
2. Control the binding process by implementing a `BindAsync` method on a type.

TryParse

`TryParse` has two APIs:

```C#
public static bool TryParse(string value, out T result);
public static bool TryParse(string value, IFormatProvider provider, out T
result);
```

The following code displays `Point: 12.3, 10.1` with the URI `/map?Point=12.3,10.1`:

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

// GET /map?Point=12.3,10.1
app.MapGet("/map", (Point point) => $"Point: {point.X}, {point.Y}");

app.Run();

public class Point
{
    public double X { get; set; }
    public double Y { get; set; }

    public static bool TryParse(string? value, IFormatProvider? provider,
                                out Point? point)
    {
        // Format is "(12.3,10.1)"
        var trimmedValue = value?.TrimStart('(').TrimEnd(')');
        var segments = trimmedValue?.Split(',',
                StringSplitOptions.RemoveEmptyEntries |
StringSplitOptions.TrimEntries);
        if (segments?.Length == 2
            && double.TryParse(segments[0], out var x)
            && double.TryParse(segments[1], out var y))
        {
            point = new Point { X = x, Y = y };
            return true;
        }

        point = null;
        return false;
    }
}
```

BindAsync

`BindAsync` has the following APIs:

```
C#

public static ValueTask<T?> BindAsync(HttpContext context, ParameterInfo
parameter);
public static ValueTask<T?> BindAsync(HttpContext context);
```

The following code displays `SortBy:xyz, SortDirection:Desc, CurrentPage:99` with the URI `/products?SortBy=xyz&SortDir=Desc&Page=99`:

```
C#

using System.Reflection;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// GET /products?SortBy=xyz&SortDir=Desc&Page=99
app.MapGet("/products", (PagingData pageData) => $"SortBy:{pageData.SortBy},
" +
       $"SortDirection:{pageData.SortDirection}, CurrentPage:
{pageData.CurrentPage}");

app.Run();

public class PagingData
{
    public string? SortBy { get; init; }
    public SortDirection SortDirection { get; init; }
    public int CurrentPage { get; init; } = 1;

    public static ValueTask<PagingData?> BindAsync(HttpContext context,
                                                   ParameterInfo parameter)
    {
        const string sortByKey = "sortBy";
        const string sortDirectionKey = "sortDir";
        const string currentPageKey = "page";

        Enum.TryParse<SortDirection>
(context.Request.Query[sortDirectionKey],
                                     ignoreCase: true, out var
sortDirection);
        int.TryParse(context.Request.Query[currentPageKey], out var page);
        page = page == 0 ? 1 : page;

        var result = new PagingData
        {
            SortBy = context.Request.Query[sortByKey],
            SortDirection = sortDirection,
            CurrentPage = page
        };
```

```
        return ValueTask.FromResult<PagingData?>(result);
    }
}

public enum SortDirection
{
    Default,
    Asc,
    Desc
}
```

Binding failures

When binding fails, the framework logs a debug message and returns various status codes to the client depending on the failure mode.

Expand table

Failure mode	Nullable Parameter Type	Binding Source	Status code
`{ParameterType}.TryParse` returns `false`	yes	route/query/header	400
`{ParameterType}.BindAsync` returns `null`	yes	custom	400
`{ParameterType}.BindAsync` throws	doesn't matter	custom	500
Failure to deserialize JSON body	doesn't matter	body	400
Wrong content type (not `application/json`)	doesn't matter	body	415

Binding Precedence

The rules for determining a binding source from a parameter:

1. Explicit attribute defined on parameter (From* attributes) in the following order:
    a. Route values: [FromRoute]
    b. Query string: [FromQuery]
    c. Header: [FromHeader]
    d. Body: [FromBody]
    e. Form: [FromForm]
    f. Service: [FromServices]

g. Parameter values: [AsParameters]

2. Special types
   a. HttpContext
   b. HttpRequest (HttpContext.Request)
   c. HttpResponse (HttpContext.Response)
   d. ClaimsPrincipal (HttpContext.User)
   e. CancellationToken (HttpContext.RequestAborted)
   f. IFormCollection (HttpContext.Request.Form)
   g. IFormFileCollection (HttpContext.Request.Form.Files)
   h. IFormFile (HttpContext.Request.Form.Files[paramName])
   i. Stream (HttpContext.Request.Body)
   j. PipeReader (HttpContext.Request.BodyReader)
3. Parameter type has a valid static BindAsync method.
4. Parameter type is a string or has a valid static TryParse method.
   a. If the parameter name exists in the route template for example, `app.Map("/todo/{id}", (int id) => {});`, then it's bound from the route.
   b. Bound from the query string.
5. If the parameter type is a service provided by dependency injection, it uses that service as the source.
6. The parameter is from the body.

Configure JSON deserialization options for body binding

The body binding source uses System.Text.Json for deserialization. It is ***not*** possible to change this default, but JSON serialization and deserialization options can be configured.

Configure JSON deserialization options globally

Options that apply globally for an app can be configured by invoking ConfigureHttpJsonOptions. The following example includes public fields and formats JSON output.

C#

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

builder.Services.ConfigureHttpJsonOptions(options => {
    options.SerializerOptions.WriteIndented = true;
    options.SerializerOptions.IncludeFields = true;
});

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

```
app.MapPost("/", (Todo todo) => {
    if (todo is not null) {
        todo.Name = todo.NameField;
    }
    return todo;
});

app.Run();

class Todo {
    public string? Name { get; set; }
    public string? NameField;
    public bool IsComplete { get; set; }
}
// If the request body contains the following JSON:
//
// {"nameField":"Walk dog", "isComplete":false}
//
// The endpoint returns the following JSON:
//
// {
//    "name":"Walk dog",
//    "nameField":"Walk dog",
//    "isComplete":false
// }
```

Since the sample code configures both serialization and deserialization, it can read `NameField` and include `NameField` in the output JSON.

Configure JSON deserialization options for an endpoint

ReadFromJsonAsync has overloads that accept a JsonSerializerOptions object. The following example includes public fields and formats JSON output.

C#

```
using System.Text.Json;

var app = WebApplication.Create();

var options = new JsonSerializerOptions(JsonSerializerDefaults.Web) {
    IncludeFields = true,
    WriteIndented = true
};

app.MapPost("/", async (HttpContext context) => {
    if (context.Request.HasJsonContentType()) {
        var todo = await context.Request.ReadFromJsonAsync<Todo>(options);
        if (todo is not null) {
            todo.Name = todo.NameField;
```

```
        }
        return Results.Ok(todo);
    }
    else {
        return Results.BadRequest();
    }
});

app.Run();

class Todo
{
    public string? Name { get; set; }
    public string? NameField;
    public bool IsComplete { get; set; }
}
// If the request body contains the following JSON:
//
// {"nameField":"Walk dog", "isComplete":false}
//
// The endpoint returns the following JSON:
//
// {
//    "name":"Walk dog",
//    "isComplete":false
// }
```

Since the preceding code applies the customized options only to deserialization, the output JSON excludes `NameField`.

Read the request body

Read the request body directly using a HttpContext or HttpRequest parameter:

C#

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

app.MapPost("/uploadstream", async (IConfiguration config, HttpRequest
request) =>
{
    var filePath = Path.Combine(config["StoredFilesPath"],
Path.GetRandomFileName());

    await using var writeStream = File.Create(filePath);
    await request.BodyReader.CopyToAsync(writeStream);
});

app.Run();
```

The preceding code:

- Accesses the request body using HttpRequest.BodyReader.
- Copies the request body to a local file.

Responses

Route handlers support the following types of return values:

1. `IResult` based - This includes `Task<IResult>` and `ValueTask<IResult>`
2. `string` - This includes `Task<string>` and `ValueTask<string>`
3. `T` (Any other type) - This includes `Task<T>` and `ValueTask<T>`

Expand table

Return value	Behavior	Content-Type
`IResult`	The framework calls IResult.ExecuteAsync	Decided by the `IResult` implementation
`string`	The framework writes the string directly to the response	`text/plain`
`T` (Any other type)	The framework JSON-serializes the response	`application/json`

For a more in-depth guide to route handler return values see Create responses in Minimal API applications

Example return values

string return values

C#

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

JSON return values

C#

```
app.MapGet("/hello", () => new { Message = "Hello World" });
```

Return TypedResults

The following code returns a TypedResults:

```
C#
```

```
app.MapGet("/hello", () => TypedResults.Ok(new Message() {  Text = "Hello
World!" }));
```

Returning `TypedResults` is preferred to returning Results. For more information, see TypedResults vs Results.

IResult return values

```
C#
```

```
app.MapGet("/hello", () => Results.Ok(new { Message = "Hello World" }));
```

The following example uses the built-in result types to customize the response:

```
C#
```

```
app.MapGet("/api/todoitems/{id}", async (int id, TodoDb db) =>
         await db.Todos.FindAsync(id)
         is Todo todo
         ? Results.Ok(todo)
         : Results.NotFound())
   .Produces<Todo>(StatusCodes.Status200OK)
   .Produces(StatusCodes.Status404NotFound);
```

JSON

```
C#
```

```
app.MapGet("/hello", () => Results.Json(new { Message = "Hello World" }));
```

Custom Status Code

```
C#
```

```
app.MapGet("/405", () => Results.StatusCode(405));
```

Text

```
C#

app.MapGet("/text", () => Results.Text("This is some text"));
```

Stream

```
C#

var proxyClient = new HttpClient();
app.MapGet("/pokemon", async () =>
{
    var stream = await
proxyClient.GetStreamAsync("http://contoso/pokedex.json");
    // Proxy the response as JSON
    return Results.Stream(stream, "application/json");
});
```

See Create responses in Minimal API applications for more examples.

Redirect

```
C#

app.MapGet("/old-path", () => Results.Redirect("/new-path"));
```

File

```
C#

app.MapGet("/download", () => Results.File("myfile.text"));
```

Built-in results

Common result helpers exist in the Results and TypedResults static classes. Returning `TypedResults` is preferred to returning `Results`. For more information, see TypedResults vs Results.

Customizing results

Applications can control responses by implementing a custom IResult type. The following code is an example of an HTML result type:

C#

```
using System.Net.Mime;
using System.Text;
static class ResultsExtensions
{
    public static IResult Html(this IResultExtensions resultExtensions,
string html)
    {
        ArgumentNullException.ThrowIfNull(resultExtensions);

        return new HtmlResult(html);
    }
}

class HtmlResult : IResult
{
    private readonly string _html;

    public HtmlResult(string html)
    {
        _html = html;
    }

    public Task ExecuteAsync(HttpContext httpContext)
    {
        httpContext.Response.ContentType = MediaTypeNames.Text.Html;
        httpContext.Response.ContentLength =
Encoding.UTF8.GetByteCount(_html);
        return httpContext.Response.WriteAsync(_html);
    }
}
```

We recommend adding an extension method to Microsoft.AspNetCore.Http.IResultExtensions to make these custom results more discoverable.

C#

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

app.MapGet("/html", () => Results.Extensions.Html(@$"<!doctype html>
<html>
    <head><title>miniHTML</title></head>
    <body>
        <h1>Hello World</h1>
        <p>The time on the server is {DateTime.Now:O}</p>
```

```
    </body>
</html>"));

app.Run();
```

Typed results

The IResult interface can represent values returned from minimal APIs that don't utilize the implicit support for JSON serializing the returned object to the HTTP response. The static Results class is used to create varying `IResult` objects that represent different types of responses. For example, setting the response status code or redirecting to another URL.

The types implementing `IResult` are public, allowing for type assertions when testing. For example:

C#

```
[TestClass()]
public class WeatherApiTests
{
    [TestMethod()]
    public void MapWeatherApiTest()
    {
        var result = WeatherApi.GetAllWeathers();
        Assert.IsInstanceOfType(result, typeof(Ok<WeatherForecast[]>));
    }
}
```

You can look at the return types of the corresponding methods on the static TypedResults class to find the correct public `IResult` type to cast to.

See Create responses in Minimal API applications for more examples.

Filters

For more information, see Filters in Minimal API apps.

Authorization

Routes can be protected using authorization policies. These can be declared via the [Authorize] attribute or by using the RequireAuthorization method:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebRPauth.Data;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthorization(o => o.AddPolicy("AdminsOnly",
                                    b => b.RequireClaim("admin", "true")));

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

var app = builder.Build();

app.UseAuthorization();

app.MapGet("/auth", [Authorize] () => "This endpoint requires
authorization.");
app.MapGet("/", () => "This endpoint doesn't require authorization.");
app.MapGet("/Identity/Account/Login", () => "Sign in page at this
endpoint.");

app.Run();
```

The preceding code can be written with RequireAuthorization:

C#

```
app.MapGet("/auth", () => "This endpoint requires authorization")
   .RequireAuthorization();
```

The following sample uses policy-based authorization:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebRPauth.Data;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthorization(o => o.AddPolicy("AdminsOnly",
                                    b => b.RequireClaim("admin", "true")));
```

```
var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

var app = builder.Build();

app.UseAuthorization();

app.MapGet("/admin", [Authorize("AdminsOnly")] () =>
                             "The /admin endpoint is for admins only.");

app.MapGet("/admin2", () => "The /admin2 endpoint is for admins only.")
   .RequireAuthorization("AdminsOnly");

app.MapGet("/", () => "This endpoint doesn't require authorization.");
app.MapGet("/Identity/Account/Login", () => "Sign in page at this
endpoint.");

app.Run();
```

Allow unauthenticated users to access an endpoint

The [AllowAnonymous] allows unauthenticated users to access endpoints:

C#

```
app.MapGet("/login", [AllowAnonymous] () => "This endpoint is for all
roles.");


app.MapGet("/login2", () => "This endpoint also for all roles.")
   .AllowAnonymous();
```

CORS

Routes can be CORS enabled using CORS policies. CORS can be declared via the [EnableCors] attribute or by using the RequireCors method. The following samples enable CORS:

C#

```
const string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

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

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

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

app.Run();
```

C#

```
using Microsoft.AspNetCore.Cors;

const string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

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

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

app.MapGet("/cors", [EnableCors(MyAllowSpecificOrigins)] () =>
                         "This endpoint allows cross origin requests!");
app.MapGet("/cors2", () => "This endpoint allows cross origin requests!")
                     .RequireCors(MyAllowSpecificOrigins);

app.Run();
```

For more information, see Enable Cross-Origin Requests (CORS) in ASP.NET Core

ValidateScopes and ValidateOnBuild

ValidateScopes and ValidateOnBuild are enabled by default in the Development environment but disabled in other environments.

When `ValidateOnBuild` is `true`, the DI container validates the service configuration at build time. If the service configuration is invalid, the build fails at app startup, rather than at runtime when the service is requested.

When `ValidateScopes` is `true`, the DI container validates that a scoped service isn't resolved from the root scope. Resolving a scoped service from the root scope can result in a memory leak because the service is retained in memory longer than the scope of the request.

`ValidateScopes` and `ValidateOnBuild` are false by default in non-Development modes for performance reasons.

The following code shows `ValidateScopes` is enabled by default in development mode but disabled in release mode:

C#

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

builder.Services.AddScoped<MyScopedService>();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    Console.WriteLine("Development environment");
}
else
{
    Console.WriteLine("Release environment");
}

app.MapGet("/", context =>
{
    // Intentionally getting service provider from app, not from the request
    // This causes an exception from attempting to resolve a scoped service
    // outside of a scope.
    // Throws System.InvalidOperationException:
    // 'Cannot resolve scoped service 'MyScopedService' from root provider.'
    var service = app.Services.GetRequiredService<MyScopedService>();
    return context.Response.WriteAsync("Service resolved");
});

app.Run();
```

```
public class MyScopedService { }
```

The following code shows `ValidateOnBuild` is enabled by default in development mode but disabled in release mode:

C#

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

builder.Services.AddScoped<MyScopedService>();
builder.Services.AddScoped<AnotherService>();

// System.AggregateException: 'Some services are not able to be constructed
(Error
// while validating the service descriptor 'ServiceType: AnotherService
Lifetime:
// Scoped ImplementationType: AnotherService': Unable to resolve service for
type
// 'BrokenService' while attempting to activate 'AnotherService'.)'
var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    Console.WriteLine("Development environment");
}
else
{
    Console.WriteLine("Release environment");
}

app.MapGet("/", context =>
{
    var service =
context.RequestServices.GetRequiredService<MyScopedService>();
    return context.Response.WriteAsync("Service resolved correctly!");
});

app.Run();

public class MyScopedService { }

public class AnotherService
{
    public AnotherService(BrokenService brokenService) { }
}

public class BrokenService { }
```

The following code disables `ValidateScopes` and `ValidateOnBuild` in `Development`:

C#

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

if (builder.Environment.IsDevelopment())
{
    Console.WriteLine("Development environment");
    // Doesn't detect the validation problems because ValidateScopes is
false.
    builder.Host.UseDefaultServiceProvider(options =>
    {
        options.ValidateScopes = false;
        options.ValidateOnBuild = false;
    });
}
```

See also

- Minimal APIs quick reference
- Generate OpenAPI documents
- Create responses in Minimal API applications
- Filters in Minimal API apps
- Handle errors in minimal APIs
- Authentication and authorization in minimal APIs
- Test Minimal API apps
- Short-circuit routing
- Identity API endpoints
- Keyed service dependency injection container support
- A look behind the scenes of minimal API endpoints
- Organizing ASP.NET Core Minimal APIs
- Fluent validation discussion on GitHub

ⓘ **Note:** The author created this article with assistance from AI. Learn more

WebApplication and WebApplicationBuilder in Minimal API apps

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.

`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();
```

Route Handlers in Minimal API apps

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.

A configured `WebApplication` supports `Map{Verb}` and MapMethods where `{Verb}` is a Pascal-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 in Minimal API applications describes the rules in detail for how route handler parameters are populated.

Responses

Create responses in Minimal API applications describes in detail how values returned from route handlers are converted into responses.

Parameter Binding in Minimal API apps

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.

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);
```

The parameters in the preceding examples are all bound from request data automatically. To demonstrate the convenience that parameter binding provides, the following route handlers show how to read request data directly from the request:

C#

```
app.MapGet("/{id}", (HttpRequest request) =>
{
    var id = request.RouteValues["id"];
    var page = request.Query["page"];
    var customHeader = request.Headers["X-CUSTOM-HEADER"];

    // ...
});

app.MapPost("/", async (HttpRequest request) =>
```

```
{
    var person = await request.ReadFromJsonAsync<Person>();

    // ...
});
```

Explicit Parameter Binding

Attributes can be used to explicitly declare where parameters are bound from.

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

// Added as service
builder.Services.AddSingleton<Service>();

var app = builder.Build();


app.MapGet("/{id}", ([FromRoute] int id,
                     [FromQuery(Name = "p")] int page,
                     [FromServices] Service service,
                     [FromHeader(Name = "Content-Type")] string contentType)
                     => {});

class Service { }

record Person(string Name, int Age);
```

Expand table

Parameter	Binding Source
`id`	route value with the name `id`
`page`	query string with the name `"p"`
`service`	Provided by dependency injection
`contentType`	header with the name `"Content-Type"`

Explicit binding from form values

The [FromForm] attribute binds form values:

C#

```
app.MapPost("/todos", async ([FromForm] string name,
    [FromForm] Visibility visibility, IFormFile? attachment, TodoDb db) =>
{
    var todo = new Todo
    {
        Name = name,
        Visibility = visibility
    };

    if (attachment is not null)
    {
        var attachmentName = Path.GetRandomFileName();

        using var stream = File.Create(Path.Combine("wwwroot",
attachmentName));
        await attachment.CopyToAsync(stream);
    }

    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Ok();
});

// Remaining code removed for brevity.
```

An alternative is to use the [AsParameters] attribute with a custom type that has properties annotated with `[FromForm]`. For example, the following code binds from form values to properties of the `NewTodoRequest` record struct:

C#

```
app.MapPost("/ap/todos", async ([AsParameters] NewTodoRequest request,
TodoDb db) =>
{
    var todo = new Todo
    {
        Name = request.Name,
        Visibility = request.Visibility
    };

    if (request.Attachment is not null)
    {
        var attachmentName = Path.GetRandomFileName();

        using var stream = File.Create(Path.Combine("wwwroot",
attachmentName));
        await request.Attachment.CopyToAsync(stream);

        todo.Attachment = attachmentName;
```

```
    }

    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Ok();
});

// Remaining code removed for brevity.
```

C#

```
public record struct NewTodoRequest([FromForm] string Name,
    [FromForm] Visibility Visibility, IFormFile? Attachment);
```

For more information, see the section on AsParameters later in this article.

The complete sample code is in the AspNetCore.Docs.Samples repository.

Secure binding from IFormFile and IFormFileCollection

Complex form binding is supported using IFormFile and IFormFileCollection using the [FromForm]:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

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

// Generate a form with an anti-forgery token and an /upload endpoint.
app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = MyUtils.GenerateHtmlForm(token.FormFieldName,
token.RequestToken!);
    return Results.Content(html, "text/html");
});
```

```
app.MapPost("/upload", async Task<Results<Ok<string>, BadRequest<string>>>
    ([FromForm] FileUploadForm fileUploadForm, HttpContext context,
                                                IAntiforgery antiforgery) =>
{
    await MyUtils.SaveFileWithName(fileUploadForm.FileDocument!,
              fileUploadForm.Name!, app.Environment.ContentRootPath);
    return TypedResults.Ok($"Your file with the description:" +
        $" {fileUploadForm.Description} has been uploaded successfully");
});

app.Run();
```

Parameters bound to the request with `[FromForm]` include an antiforgery token. The antiforgery token is validated when the request is processed. For more information, see Antiforgery with Minimal APIs.

For more information, see Form binding in minimal APIs.

The complete sample code is in the AspNetCore.Docs.Samples repository.

Parameter binding with dependency injection

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

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IDateTime, SystemDateTime>();

var app = builder.Build();

app.MapGet("/",   (               IDateTime dateTime) => dateTime.Now);
app.MapGet("/fs", ([FromServices] IDateTime dateTime) => dateTime.Now);
app.Run();
```

Optional parameters

Parameters declared in route handlers are treated as required:

- If a request matches the route, the route handler only runs if all required parameters are provided in the request.

- Failure to provide all required parameters results in an error.

C#

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

app.MapGet("/products", (int pageNumber) => $"Requesting page
{pageNumber}");

app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	3 returned
`/products`	`BadHttpRequestException`: Required parameter "int pageNumber" wasn't provided from query string.
`/products/1`	HTTP 404 error, no matching route

To make `pageNumber` optional, define the type as optional or provide a default value:

C#

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

app.MapGet("/products", (int? pageNumber) => $"Requesting page {pageNumber
?? 1}");

string ListProducts(int pageNumber = 1) => $"Requesting page {pageNumber}";

app.MapGet("/products2", ListProducts);

app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	3 returned
`/products`	1 returned

URI	result
`/products2`	1 returned

The preceding nullable and default value applies to all sources:

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapPost("/products", (Product? product) => { });

app.Run();
```

The preceding code calls the method with a null product if no request body is sent.

NOTE: If invalid data is provided and the parameter is nullable, the route handler is ***not*** run.

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/products", (int? pageNumber) => $"Requesting page {pageNumber
?? 1}");

app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	`3` returned
`/products`	`1` returned
`/products?pageNumber=two`	`BadHttpRequestException`: Failed to bind parameter `"Nullable<int> pageNumber"` from "two".
`/products/two`	HTTP 404 error, no matching route

See the Binding Failures section for more information.

Special types

The following types are bound without explicit attributes:

- HttpContext: The context which holds all the information about the current HTTP request or response:

  C#

  ```
  app.MapGet("/", (HttpContext context) =>
  context.Response.WriteAsync("Hello World"));
  ```

- HttpRequest and HttpResponse: The HTTP request and HTTP response:

  C#

  ```
  app.MapGet("/", (HttpRequest request, HttpResponse response) =>
      response.WriteAsync($"Hello World {request.Query["name"]}"));
  ```

- CancellationToken: The cancellation token associated with the current HTTP request:

  C#

  ```
  app.MapGet("/", async (CancellationToken cancellationToken) =>
      await MakeLongRunningRequestAsync(cancellationToken));
  ```

- ClaimsPrincipal: The user associated with the request, bound from HttpContext.User:

  C#

  ```
  app.MapGet("/", (ClaimsPrincipal user) => user.Identity.Name);
  ```

Bind the request body as a `Stream` or `PipeReader`

The request body can bind as a Stream or PipeReader to efficiently support scenarios where the user has to process data and:

- Store the data to blob storage or enqueue the data to a queue provider.
- Process the stored data with a worker process or cloud function.

For example, the data might be enqueued to Azure Queue storage or stored in Azure Blob storage.

The following code implements a background queue:

C#

```
using System.Text.Json;
using System.Threading.Channels;

namespace BackgroundQueueService;

class BackgroundQueue : BackgroundService
{
    private readonly Channel<ReadOnlyMemory<byte>> _queue;
    private readonly ILogger<BackgroundQueue> _logger;

    public BackgroundQueue(Channel<ReadOnlyMemory<byte>> queue,
                           ILogger<BackgroundQueue> logger)
    {
        _queue = queue;
        _logger = logger;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        await foreach (var dataStream in
_queue.Reader.ReadAllAsync(stoppingToken))
        {
            try
            {
                var person = JsonSerializer.Deserialize<Person>
(dataStream.Span)!;
                _logger.LogInformation($"{person.Name} is {person.Age} " +
                                       $"years and from {person.Country}");
            }
            catch (Exception ex)
            {
                _logger.LogError(ex.Message);
            }
        }
    }
}

class Person
{
    public string Name { get; set; } = String.Empty;
    public int Age { get; set; }
    public string Country { get; set; } = String.Empty;
}
```

The following code binds the request body to a `Stream`:

C#

```
app.MapPost("/register", async (HttpRequest req, Stream body,
                                Channel<ReadOnlyMemory<byte>> queue) =>
```

```
{
    if (req.ContentLength is not null && req.ContentLength > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // We're not above the message size and we have a content length, or
    // we're a chunked request and we're going to read up to the
maxMessageSize + 1.
    // We add one to the message size so that we can detect when a chunked
request body
    // is bigger than our configured max.
    var readSize = (int?)req.ContentLength ?? (maxMessageSize + 1);

    var buffer = new byte[readSize];

    // Read at least that many bytes from the body.
    var read = await body.ReadAtLeastAsync(buffer, readSize,
throwOnEndOfStream: false);

    // We read more than the max, so this is a bad request.
    if (read > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // Attempt to send the buffer to the background queue.
    if (queue.Writer.TryWrite(buffer.AsMemory(0..read)))
    {
        return Results.Accepted();
    }

    // We couldn't accept the message since we're overloaded.
    return Results.StatusCode(StatusCodes.Status429TooManyRequests);
});
```

The following code shows the complete `Program.cs` file:

C#

```
using System.Threading.Channels;
using BackgroundQueueService;

var builder = WebApplication.CreateBuilder(args);
// The max memory to use for the upload endpoint on this instance.
var maxMemory = 500 * 1024 * 1024;

// The max size of a single message, staying below the default LOH size of
85K.
var maxMessageSize = 80 * 1024;

// The max size of the queue based on those restrictions
var maxQueueSize = maxMemory / maxMessageSize;
```

```
// Create a channel to send data to the background queue.
builder.Services.AddSingleton<Channel<ReadOnlyMemory<byte>>>((_) =>
                    Channel.CreateBounded<ReadOnlyMemory<byte>>
(maxQueueSize));

// Create a background queue service.
builder.Services.AddHostedService<BackgroundQueue>();
var app = builder.Build();

// curl --request POST 'https://localhost:<port>/register' --header
'Content-Type: application/json' --data-raw '{ "Name":"Samson", "Age": 23,
"Country":"Nigeria" }'
// curl --request POST "https://localhost:<port>/register" --header
"Content-Type: application/json" --data-raw "{ \"Name\":\"Samson\", \"Age\":
23, \"Country\":\"Nigeria\" }"
app.MapPost("/register", async (HttpRequest req, Stream body,
                                Channel<ReadOnlyMemory<byte>> queue) =>
{
    if (req.ContentLength is not null && req.ContentLength > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // We're not above the message size and we have a content length, or
    // we're a chunked request and we're going to read up to the
maxMessageSize + 1.
    // We add one to the message size so that we can detect when a chunked
request body
    // is bigger than our configured max.
    var readSize = (int?)req.ContentLength ?? (maxMessageSize + 1);

    var buffer = new byte[readSize];

    // Read at least that many bytes from the body.
    var read = await body.ReadAtLeastAsync(buffer, readSize,
throwOnEndOfStream: false);

    // We read more than the max, so this is a bad request.
    if (read > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // Attempt to send the buffer to the background queue.
    if (queue.Writer.TryWrite(buffer.AsMemory(0..read)))
    {
        return Results.Accepted();
    }

    // We couldn't accept the message since we're overloaded.
    return Results.StatusCode(StatusCodes.Status429TooManyRequests);
});

app.Run();
```

- When reading data, the `Stream` is the same object as `HttpRequest.Body`.
- The request body isn't buffered by default. After the body is read, it's not rewindable. The stream can't be read multiple times.
- The `Stream` and `PipeReader` aren't usable outside of the minimal action handler as the underlying buffers will be disposed or reused.

File uploads using IFormFile and IFormFileCollection

The following code uses IFormFile and IFormFileCollection to upload file:

C#

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

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

app.MapPost("/upload", async (IFormFile file) =>
{
    var tempFile = Path.GetTempFileName();
    app.Logger.LogInformation(tempFile);
    using var stream = File.OpenWrite(tempFile);
    await file.CopyToAsync(stream);
});

app.MapPost("/upload_many", async (IFormFileCollection myFiles) =>
{
    foreach (var file in myFiles)
    {
        var tempFile = Path.GetTempFileName();
        app.Logger.LogInformation(tempFile);
        using var stream = File.OpenWrite(tempFile);
        await file.CopyToAsync(stream);
    }
});

app.Run();
```

Authenticated file upload requests are supported using an Authorization header, a client certificate, or a cookie header.

Binding to forms with IFormCollection, IFormFile, and IFormFileCollection

Binding from form-based parameters using IFormCollection, IFormFile, and IFormFileCollection is supported. OpenAPI metadata is inferred for form parameters to support integration with Swagger UI.

The following code uploads files using inferred binding from the `IFormFile` type:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

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

string GetOrCreateFilePath(string fileName, string filesDirectory =
"uploadFiles")
{
    var directoryPath = Path.Combine(app.Environment.ContentRootPath,
filesDirectory);
    Directory.CreateDirectory(directoryPath);
    return Path.Combine(directoryPath, fileName);
}

async Task UploadFileWithName(IFormFile file, string fileSaveName)
{
    var filePath = GetOrCreateFilePath(fileSaveName);
    await using var fileStream = new FileStream(filePath, FileMode.Create);
    await file.CopyToAsync(fileStream);
}

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
      <html>
        <body>
          <form action="/upload" method="POST" enctype="multipart/form-
data">
            <input name="{token.FormFieldName}" type="hidden" value="
{token.RequestToken}"/>
            <input type="file" name="file" placeholder="Upload an image..."
accept=".jpg,

.jpeg, .png" />
            <input type="submit" />
          </form>
        </body>
      </html>
    """;
```

```
    return Results.Content(html, "text/html");
});

app.MapPost("/upload", async Task<Results<Ok<string>,
   BadRequest<string>>> (IFormFile file, HttpContext context, IAntiforgery
antiforgery) =>
{
    var fileSaveName = Guid.NewGuid().ToString("N") +
Path.GetExtension(file.FileName);
    await UploadFileWithName(file, fileSaveName);
    return TypedResults.Ok("File uploaded successfully!");
});

app.Run();
```

Warning: When implementing forms, the app ***must prevent*** Cross-Site Request Forgery (XSRF/CSRF) attacks. In the preceding code, the IAntiforgery service is used to prevent XSRF attacks by generating and validation an antiforgery token:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

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

string GetOrCreateFilePath(string fileName, string filesDirectory =
"uploadFiles")
{
    var directoryPath = Path.Combine(app.Environment.ContentRootPath,
filesDirectory);
    Directory.CreateDirectory(directoryPath);
    return Path.Combine(directoryPath, fileName);
}

async Task UploadFileWithName(IFormFile file, string fileSaveName)
{
    var filePath = GetOrCreateFilePath(fileSaveName);
    await using var fileStream = new FileStream(filePath, FileMode.Create);
    await file.CopyToAsync(fileStream);
}

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
      <html>
```

```
        <body>
          <form action="/upload" method="POST" enctype="multipart/form-
data">
            <input name="{token.FormFieldName}" type="hidden" value="
{token.RequestToken}"/>
            <input type="file" name="file" placeholder="Upload an image..."
accept=".jpg,

.jpeg, .png" />
            <input type="submit" />
          </form>
        </body>
      </html>
    """;

    return Results.Content(html, "text/html");
});

app.MapPost("/upload", async Task<Results<Ok<string>,
   BadRequest<string>>> (IFormFile file, HttpContext context, IAntiforgery
antiforgery) =>
{
    var fileSaveName = Guid.NewGuid().ToString("N") +
Path.GetExtension(file.FileName);
    await UploadFileWithName(file, fileSaveName);
    return TypedResults.Ok("File uploaded successfully!");
});

app.Run();
```

For more information on XSRF attacks, see Antiforgery with Minimal APIs

For more information, see Form binding in minimal APIs ↗;

Bind to collections and complex types from forms

Binding is supported for:

- Collections, for example List and Dictionary
- Complex types, for example, `Todo` or `Project`

The following code shows:

- A minimal endpoint that binds a multi-part form input to a complex object.
- How to use the antiforgery services to support the generation and validation of antiforgery tokens.

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
        <html><body>
           <form action="/todo" method="POST" enctype="multipart/form-data">
               <input name="{token.FormFieldName}"
                                type="hidden" value="{token.RequestToken}"
/>
               <input type="text" name="name" />
               <input type="date" name="dueDate" />
               <input type="checkbox" name="isCompleted" value="true" />
               <input type="submit" />
               <input name="isCompleted" type="hidden" value="false" />
           </form>
        </body></html>
    """;
    return Results.Content(html, "text/html");
});

app.MapPost("/todo", async Task<Results<Ok<Todo>, BadRequest<string>>>
               ([FromForm] Todo todo, HttpContext context, IAntiforgery
antiforgery) =>
{
    try
    {
        await antiforgery.ValidateRequestAsync(context);
        return TypedResults.Ok(todo);
    }
    catch (AntiforgeryValidationException e)
    {
        return TypedResults.BadRequest("Invalid antiforgery token");
    }
});

app.Run();

class Todo
{
    public string Name { get; set; } = string.Empty;
    public bool IsCompleted { get; set; } = false;
    public DateTime DueDate { get; set; } =
```

```
DateTime.Now.Add(TimeSpan.FromDays(1));
}
```

In the preceding code:

- The target parameter ***must*** be annotated with the [FromForm] attribute to disambiguate from parameters that should be read from the JSON body.
- Binding from complex or collection types is ***not*** supported for minimal APIs that are compiled with the Request Delegate Generator.
- The markup shows an additional hidden input with a name of `isCompleted` and a value of `false`. If the `isCompleted` checkbox is checked when the form is submitted, both values `true` and `false` are submitted as values. If the checkbox is unchecked, only the hidden input value `false` is submitted. The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

An example of the form data submitted to the preceding endpoint looks as follows:

txt

```
__RequestVerificationToken:
CfDJ8Bveip67DklJm5vI2PF2VOUZ594RC8kcGWpTnVV17zCLZi1yrs-
CSz426ZRRrQnEJ0gybB0AD7hTU-0EGJXDU-OaJaktgAtWLIaaEWMOWCkoxYYm-
9U9eLV7INSUrQ6yBHqdMEE_aJpD4AI72gYiCqc
name: Walk the dog
dueDate: 2024-04-06
isCompleted: true
isCompleted: false
```

Bind arrays and string values from headers and query strings

The following code demonstrates binding query strings to an array of primitive types, string arrays, and StringValues:

C#

```
// Bind query string values to a primitive type array.
// GET  /tags?q=1&q=2&q=3
app.MapGet("/tags", (int[] q) =>
                      $"tag1: {q[0]} , tag2: {q[1]}, tag3: {q[2]}");

// Bind to a string array.
// GET /tags2?names=john&names=jack&names=jane
```

```
app.MapGet("/tags2", (string[] names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3: {names[2]}");

// Bind to StringValues.
// GET /tags3?names=john&names=jack&names=jane
app.MapGet("/tags3", (StringValues names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3: {names[2]}");
```

Binding query strings or header values to an array of complex types is supported when the type has `TryParse` implemented. The following code binds to a string array and returns all the items with the specified tags:

C#

```
// GET /todoitems/tags?tags=home&tags=work
app.MapGet("/todoitems/tags", async (Tag[] tags, TodoDb db) =>
{
    return await db.Todos
        .Where(t => tags.Select(i => i.Name).Contains(t.Tag.Name))
        .ToListAsync();
});
```

The following code shows the model and the required `TryParse` implementation:

C#

```
public class Todo
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }

    // This is an owned entity.
    public Tag Tag { get; set; } = new();
}

[Owned]
public class Tag
{
    public string? Name { get; set; } = "n/a";

    public static bool TryParse(string? name, out Tag tag)
    {
        if (name is null)
        {
            tag = default!;
            return false;
        }

        tag = new Tag { Name = name };
        return true;
```

```
    }
}
```

The following code binds to an `int` array:

C#

```
// GET /todoitems/query-string-ids?ids=1&ids=3
app.MapGet("/todoitems/query-string-ids", async (int[] ids, TodoDb db) =>
{
    return await db.Todos
        .Where(t => ids.Contains(t.Id))
        .ToListAsync();
});
```

To test the preceding code, add the following endpoint to populate the database with `Todo` items:

C#

```
// POST /todoitems/batch
app.MapPost("/todoitems/batch", async (Todo[] todos, TodoDb db) =>
{
    await db.Todos.AddRangeAsync(todos);
    await db.SaveChangesAsync();

    return Results.Ok(todos);
});
```

Use a tool like HttpRepl to pass the following data to the previous endpoint:

C#

```
[
    {
        "id": 1,
        "name": "Have Breakfast",
        "isComplete": true,
        "tag": {
            "name": "home"
        }
    },
    {
        "id": 2,
        "name": "Have Lunch",
        "isComplete": true,
        "tag": {
            "name": "work"
        }
    },
```

```
    {
        "id": 3,
        "name": "Have Supper",
        "isComplete": true,
        "tag": {
            "name": "home"
        }
    },
    {
        "id": 4,
        "name": "Have Snacks",
        "isComplete": true,
        "tag": {
            "name": "N/A"
        }
    }
]
```

The following code binds to the header key `X-Todo-Id` and returns the `Todo` items with matching `Id` values:

C#

```
// GET /todoitems/header-ids
// The keys of the headers should all be X-Todo-Id with different values
app.MapGet("/todoitems/header-ids", async ([FromHeader(Name = "X-Todo-Id")]
int[] ids, TodoDb db) =>
{
    return await db.Todos
        .Where(t => ids.Contains(t.Id))
        .ToListAsync();
});
```

> ⓘ Note
>
> When binding a `string[]` from a query string, the absence of any matching query string value will result in an empty array instead of a null value.

Parameter binding for argument lists with [AsParameters]

AsParametersAttribute enables simple parameter binding to types and not complex or recursive model binding.

Consider the following code:

C#

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

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
var app = builder.Build();

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.Select(x => new TodoItemDTO(x)).ToListAsync());

app.MapGet("/todoitems/{id}",
                            async (int Id, TodoDb Db) =>
    await Db.Todos.FindAsync(Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
// Remaining code removed for brevity.
```

Consider the following `GET` endpoint:

C#

```
app.MapGet("/todoitems/{id}",
                            async (int Id, TodoDb Db) =>
    await Db.Todos.FindAsync(Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
```

The following `struct` can be used to replace the preceding highlighted parameters:

C#

```
struct TodoItemRequest
{
    public int Id { get; set; }
    public TodoDb Db { get; set; }
}
```

The refactored `GET` endpoint uses the preceding `struct` with the AsParameters attribute:

C#

```
app.MapGet("/ap/todoitems/{id}",
                                async ([AsParameters] TodoItemRequest
request) =>
```

```
    await request.Db.Todos.FindAsync(request.Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
```

The following code shows additional endpoints in the app:

C#

```
app.MapPost("/todoitems", async (TodoItemDTO Dto, TodoDb Db) =>
{
    var todoItem = new Todo
    {
        IsComplete = Dto.IsComplete,
        Name = Dto.Name
    };

    Db.Todos.Add(todoItem);
    await Db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todoItem.Id}", new
TodoItemDTO(todoItem));
});

app.MapPut("/todoitems/{id}", async (int Id, TodoItemDTO Dto, TodoDb Db) =>
{
    var todo = await Db.Todos.FindAsync(Id);

    if (todo is null) return Results.NotFound();

    todo.Name = Dto.Name;
    todo.IsComplete = Dto.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.Ok(new TodoItemDTO(todo));
    }

    return Results.NotFound();
});
```

The following classes are used to refactor the parameter lists:

C#

```
class CreateTodoItemRequest
{
    public TodoItemDTO Dto { get; set; } = default!;
    public TodoDb Db { get; set; } = default!;
}

class EditTodoItemRequest
{
    public int Id { get; set; }
    public TodoItemDTO Dto { get; set; } = default!;
    public TodoDb Db { get; set; } = default!;
}
```

The following code shows the refactored endpoints using `AsParameters` and the preceding `struct` and classes:

C#

```
app.MapPost("/ap/todoitems", async ([AsParameters] CreateTodoItemRequest
request) =>
{
    var todoItem = new Todo
    {
        IsComplete = request.Dto.IsComplete,
        Name = request.Dto.Name
    };

    request.Db.Todos.Add(todoItem);
    await request.Db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todoItem.Id}", new
TodoItemDTO(todoItem));
});

app.MapPut("/ap/todoitems/{id}", async ([AsParameters] EditTodoItemRequest
request) =>
{
    var todo = await request.Db.Todos.FindAsync(request.Id);

    if (todo is null) return Results.NotFound();

    todo.Name = request.Dto.Name;
    todo.IsComplete = request.Dto.IsComplete;

    await request.Db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/ap/todoitems/{id}", async ([AsParameters] TodoItemRequest
```

```
request) =>
{
    if (await request.Db.Todos.FindAsync(request.Id) is Todo todo)
    {
        request.Db.Todos.Remove(todo);
        await request.Db.SaveChangesAsync();
        return Results.Ok(new TodoItemDTO(todo));
    }

    return Results.NotFound();
});
```

The following record types can be used to replace the preceding parameters:

C#

```
record TodoItemRequest(int Id, TodoDb Db);
record CreateTodoItemRequest(TodoItemDTO Dto, TodoDb Db);
record EditTodoItemRequest(int Id, TodoItemDTO Dto, TodoDb Db);
```

Using a `struct` with `AsParameters` can be more performant than using a `record` type.

The complete sample code in the AspNetCore.Docs.Samples repository.

Custom Binding

There are two ways to customize parameter binding:

1. For route, query, and header binding sources, bind custom types by adding a static `TryParse` method for the type.
2. Control the binding process by implementing a `BindAsync` method on a type.

TryParse

`TryParse` has two APIs:

C#

```
public static bool TryParse(string value, out T result);
public static bool TryParse(string value, IFormatProvider provider, out T
result);
```

The following code displays `Point: 12.3, 10.1` with the URI `/map?Point=12.3,10.1`:

C#

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

// GET /map?Point=12.3,10.1
app.MapGet("/map", (Point point) => $"Point: {point.X}, {point.Y}");

app.Run();

public class Point
{
    public double X { get; set; }
    public double Y { get; set; }

    public static bool TryParse(string? value, IFormatProvider? provider,
                                out Point? point)
    {
        // Format is "(12.3,10.1)"
        var trimmedValue = value?.TrimStart('(').TrimEnd(')');
        var segments = trimmedValue?.Split(',',
                StringSplitOptions.RemoveEmptyEntries |
StringSplitOptions.TrimEntries);
        if (segments?.Length == 2
            && double.TryParse(segments[0], out var x)
            && double.TryParse(segments[1], out var y))
        {
            point = new Point { X = x, Y = y };
            return true;
        }

        point = null;
        return false;
    }
}
```

BindAsync

`BindAsync` has the following APIs:

C#

```
public static ValueTask<T?> BindAsync(HttpContext context, ParameterInfo
parameter);
public static ValueTask<T?> BindAsync(HttpContext context);
```

The following code displays `SortBy:xyz, SortDirection:Desc, CurrentPage:99` with the URI `/products?SortBy=xyz&SortDir=Desc&Page=99`:

C#

```
using System.Reflection;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// GET /products?SortBy=xyz&SortDir=Desc&Page=99
app.MapGet("/products", (PagingData pageData) => $"SortBy:{pageData.SortBy},
" +
       $"SortDirection:{pageData.SortDirection}, CurrentPage:
{pageData.CurrentPage}");

app.Run();

public class PagingData
{
    public string? SortBy { get; init; }
    public SortDirection SortDirection { get; init; }
    public int CurrentPage { get; init; } = 1;

    public static ValueTask<PagingData?> BindAsync(HttpContext context,
                                                   ParameterInfo parameter)
    {
        const string sortByKey = "sortBy";
        const string sortDirectionKey = "sortDir";
        const string currentPageKey = "page";

        Enum.TryParse<SortDirection>
(context.Request.Query[sortDirectionKey],
                                     ignoreCase: true, out var
sortDirection);
        int.TryParse(context.Request.Query[currentPageKey], out var page);
        page = page == 0 ? 1 : page;

        var result = new PagingData
        {
            SortBy = context.Request.Query[sortByKey],
            SortDirection = sortDirection,
            CurrentPage = page
        };

        return ValueTask.FromResult<PagingData?>(result);
    }
}

public enum SortDirection
{
    Default,
    Asc,
    Desc
}
```

Binding failures

When binding fails, the framework logs a debug message and returns various status codes to the client depending on the failure mode.

Expand table

Failure mode	Nullable Parameter Type	Binding Source	Status code
`{ParameterType}.TryParse` returns `false`	yes	route/query/header	400
`{ParameterType}.BindAsync` returns `null`	yes	custom	400
`{ParameterType}.BindAsync` throws	doesn't matter	custom	500
Failure to deserialize JSON body	doesn't matter	body	400
Wrong content type (not `application/json`)	doesn't matter	body	415

Binding Precedence

The rules for determining a binding source from a parameter:

1. Explicit attribute defined on parameter (From* attributes) in the following order:
   a. Route values: [FromRoute]
   b. Query string: [FromQuery]
   c. Header: [FromHeader]
   d. Body: [FromBody]
   e. Form: [FromForm]
   f. Service: [FromServices]
   g. Parameter values: [AsParameters]
2. Special types
   a. HttpContext
   b. HttpRequest (HttpContext.Request)
   c. HttpResponse (HttpContext.Response)
   d. ClaimsPrincipal (HttpContext.User)
   e. CancellationToken (HttpContext.RequestAborted)
   f. IFormCollection (HttpContext.Request.Form)
   g. IFormFileCollection (HttpContext.Request.Form.Files)
   h. IFormFile (HttpContext.Request.Form.Files[paramName])
   i. Stream (HttpContext.Request.Body)
   j. PipeReader (HttpContext.Request.BodyReader)

3. Parameter type has a valid static BindAsync method.
4. Parameter type is a string or has a valid static TryParse method.
   a. If the parameter name exists in the route template for example, `app.Map("/todo/{id}", (int id) => {});`, then it's bound from the route.
   b. Bound from the query string.
5. If the parameter type is a service provided by dependency injection, it uses that service as the source.
6. The parameter is from the body.

Configure JSON deserialization options for body binding

The body binding source uses System.Text.Json for deserialization. It is ***not*** possible to change this default, but JSON serialization and deserialization options can be configured.

Configure JSON deserialization options globally

Options that apply globally for an app can be configured by invoking ConfigureHttpJsonOptions. The following example includes public fields and formats JSON output.

C#

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

builder.Services.ConfigureHttpJsonOptions(options => {
    options.SerializerOptions.WriteIndented = true;
    options.SerializerOptions.IncludeFields = true;
});

var app = builder.Build();

app.MapPost("/", (Todo todo) => {
    if (todo is not null) {
        todo.Name = todo.NameField;
    }
    return todo;
});

app.Run();

class Todo {
    public string? Name { get; set; }
    public string? NameField;
    public bool IsComplete { get; set; }
}
// If the request body contains the following JSON:
```

```
//
// {"nameField":"Walk dog", "isComplete":false}
//
// The endpoint returns the following JSON:
//
// {
//    "name":"Walk dog",
//    "nameField":"Walk dog",
//    "isComplete":false
// }
```

Since the sample code configures both serialization and deserialization, it can read `NameField` and include `NameField` in the output JSON.

Configure JSON deserialization options for an endpoint

ReadFromJsonAsync has overloads that accept a JsonSerializerOptions object. The following example includes public fields and formats JSON output.

C#

```
using System.Text.Json;

var app = WebApplication.Create();

var options = new JsonSerializerOptions(JsonSerializerDefaults.Web) {
    IncludeFields = true,
    WriteIndented = true
};

app.MapPost("/", async (HttpContext context) => {
    if (context.Request.HasJsonContentType()) {
        var todo = await context.Request.ReadFromJsonAsync<Todo>(options);
        if (todo is not null) {
            todo.Name = todo.NameField;
        }
        return Results.Ok(todo);
    }
    else {
        return Results.BadRequest();
    }
});

app.Run();

class Todo
{
    public string? Name { get; set; }
    public string? NameField;
    public bool IsComplete { get; set; }
}
```

```
// If the request body contains the following JSON:
//
// {"nameField":"Walk dog", "isComplete":false}
//
// The endpoint returns the following JSON:
//
// {
//    "name":"Walk dog",
//    "isComplete":false
// }
```

Since the preceding code applies the customized options only to deserialization, the output JSON excludes `NameField`.

Read the request body

Read the request body directly using a HttpContext or HttpRequest parameter:

C#

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

app.MapPost("/uploadstream", async (IConfiguration config, HttpRequest
request) =>
{
    var filePath = Path.Combine(config["StoredFilesPath"],
Path.GetRandomFileName());

    await using var writeStream = File.Create(filePath);
    await request.BodyReader.CopyToAsync(writeStream);
});

app.Run();
```

The preceding code:

- Accesses the request body using HttpRequest.BodyReader.
- Copies the request body to a local file.

How to create responses in Minimal API apps

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.

Minimal endpoints support the following types of return values:

1. `string` - This includes `Task<string>` and `ValueTask<string>`.
2. `T` (Any other type) - This includes `Task<T>` and `ValueTask<T>`.
3. `IResult` based - This includes `Task<IResult>` and `ValueTask<IResult>`.

`string` return values

Expand table

Behavior	Content-Type
The framework writes the string directly to the response.	`text/plain`

Consider the following route handler, which returns a `Hello world` text.

C#

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

The `200` status code is returned with `text/plain` Content-Type header and the following content.

text

```
Hello World
```

`T` (Any other type) return values

Expand table

Behavior	Content-Type
The framework JSON-serializes the response.	`application/json`

Consider the following route handler, which returns an anonymous type containing a `Message` string property.

C#

```
app.MapGet("/hello", () => new { Message = "Hello World" });
```

The `200` status code is returned with `application/json` Content-Type header and the following content.

JSON

```
{"message":"Hello World"}
```

`IResult` return values

Expand table

Behavior	Content-Type
The framework calls IResult.ExecuteAsync.	Decided by the `IResult` implementation.

The `IResult` interface defines a contract that represents the result of an HTTP endpoint. The static Results class and the static TypedResults are used to create various `IResult` objects that represent different types of responses.

TypedResults vs Results

The Results and TypedResults static classes provide similar sets of results helpers. The `TypedResults` class is the *typed* equivalent of the `Results` class. However, the `Results` helpers' return type is IResult, while each `TypedResults` helper's return type is one of the `IResult` implementation types. The difference means that for `Results` helpers a conversion is needed when the concrete type is needed, for example, for unit testing.

The implementation types are defined in the Microsoft.AspNetCore.Http.HttpResults namespace.

Returning `TypedResults` rather than `Results` has the following advantages:

- `TypedResults` helpers return strongly typed objects, which can improve code readability, unit testing, and reduce the chance of runtime errors.
- The implementation type automatically provides the response type metadata for OpenAPI to describe the endpoint.

Consider the following endpoint, for which a `200 OK` status code with the expected JSON response is produced.

C#

```
app.MapGet("/hello", () => Results.Ok(new Message() { Text = "Hello World!"
}))
    .Produces<Message>();
```

In order to document this endpoint correctly the extensions method `Produces` is called. However, it's not necessary to call `Produces` if `TypedResults` is used instead of `Results`, as shown in the following code. `TypedResults` automatically provides the metadata for the endpoint.

C#

```
app.MapGet("/hello2", () => TypedResults.Ok(new Message() { Text = "Hello
World!" }));
```

For more information about describing a response type, see OpenAPI support in minimal APIs.

As mentioned previously, when using `TypedResults`, a conversion is not needed. Consider the following minimal API which returns a `TypedResults` class

C#

```
public static async Task<Ok<Todo[]>> GetAllTodos(TodoGroupDbContext
database)
{
    var todos = await database.Todos.ToArrayAsync();
    return TypedResults.Ok(todos);
}
```

The following test checks for the full concrete type:

C#

```
[Fact]
public async Task GetAllReturnsTodosFromDatabase()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    context.Todos.Add(new Todo
    {
        Id = 1,
        Title = "Test title 1",
        Description = "Test description 1",
        IsDone = false
    });

    context.Todos.Add(new Todo
    {
        Id = 2,
        Title = "Test title 2",
        Description = "Test description 2",
        IsDone = true
    });

    await context.SaveChangesAsync();

    // Act
    var result = await TodoEndpointsV1.GetAllTodos(context);

    //Assert
    Assert.IsType<Ok<Todo[]>>(result);

    Assert.NotNull(result.Value);
    Assert.NotEmpty(result.Value);
    Assert.Collection(result.Value, todo1 =>
    {
        Assert.Equal(1, todo1.Id);
        Assert.Equal("Test title 1", todo1.Title);
        Assert.False(todo1.IsDone);
    }, todo2 =>
    {
        Assert.Equal(2, todo2.Id);
        Assert.Equal("Test title 2", todo2.Title);
        Assert.True(todo2.IsDone);
    });
}
```

Because all methods on `Results` return `IResult` in their signature, the compiler automatically infers that as the request delegate return type when returning different results from a single endpoint. `TypedResults` requires the use of `Results<T1, TN>` from such delegates.

The following method compiles because both Results.Ok and Results.NotFound are declared as returning `IResult`, even though the actual concrete types of the objects returned are different:

```
C#

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

The following method does not compile, because `TypedResults.Ok` and `TypedResults.NotFound` are declared as returning different types and the compiler won't attempt to infer the best matching type:

```
C#

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
     await db.Todos.FindAsync(id)
     is Todo todo
        ? TypedResults.Ok(todo)
        : TypedResults.NotFound());
```

To use `TypedResults`, the return type must be fully declared, which when asynchronous requires the `Task<>` wrapper. Using `TypedResults` is more verbose, but that's the trade-off for having the type information be statically available and thus capable of self-describing to OpenAPI:

```
C#

app.MapGet("/todoitems/{id}", async Task<Results<Ok<Todo>, NotFound>> (int
id, TodoDb db) =>
   await db.Todos.FindAsync(id)
    is Todo todo
       ? TypedResults.Ok(todo)
       : TypedResults.NotFound());
```

Results<TResult1, TResultN>

Use Results<TResult1, TResultN> as the endpoint handler return type instead of `IResult` when:

- Multiple `IResult` implementation types are returned from the endpoint handler.

- The static `TypedResult` class is used to create the `IResult` objects.

This alternative is better than returning `IResult` because the generic union types automatically retain the endpoint metadata. And since the `Results<TResult1, TResultN>` union types implement implicit cast operators, 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 endpoint, for which a `400 BadRequest` status code is returned when the `orderId` is greater than `999`. Otherwise, it produces a `200 OK` with the expected content.

C#

```
app.MapGet("/orders/{orderId}", IResult (int orderId)
    => orderId > 999 ? TypedResults.BadRequest() : TypedResults.Ok(new
Order(orderId)))
    .Produces(400)
    .Produces<Order>();
```

In order to document this endpoint correctly the extension method `Produces` is called. However, since the `TypedResults` helper automatically includes the metadata for the endpoint, you can return the `Results<T1, Tn>` union type instead, as shown in the following code.

C#

```
app.MapGet("/orders/{orderId}", Results<BadRequest, Ok<Order>> (int orderId)
    => orderId > 999 ? TypedResults.BadRequest() : TypedResults.Ok(new
Order(orderId)));
```

Built-in results

Common result helpers exist in the Results and TypedResults static classes. Returning `TypedResults` is preferred to returning `Results`. For more information, see TypedResults vs Results.

The following sections demonstrate the usage of the common result helpers.

JSON

```C#
app.MapGet("/hello", () => Results.Json(new { Message = "Hello World" }));
```

WriteAsJsonAsync is an alternative way to return JSON:

```C#
app.MapGet("/", (HttpContext context) => context.Response.WriteAsJsonAsync
    (new { Message = "Hello World" }));
```

Custom Status Code

```C#
app.MapGet("/405", () => Results.StatusCode(405));
```

Internal Server Error

```C#
app.MapGet("/500", () => Results.InternalServerError("Something went
wrong!"));
```

The preceding example returns a 500 status code.

Text

```C#
app.MapGet("/text", () => Results.Text("This is some text"));
```

Stream

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

var proxyClient = new HttpClient();
```

```
app.MapGet("/pokemon", async () =>
{
    var stream = await
proxyClient.GetStreamAsync("http://contoso/pokedex.json");
    // Proxy the response as JSON
    return Results.Stream(stream, "application/json");
});

app.Run();
```

Results.Stream overloads allow access to the underlying HTTP response stream without buffering. The following example uses ImageSharp to return a reduced size of the specified image:

C#

```
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Formats.Jpeg;
using SixLabors.ImageSharp.Processing;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/process-image/{strImage}", (string strImage, HttpContext http,
CancellationToken token) =>
{
    http.Response.Headers.CacheControl = $"public,max-age=
{TimeSpan.FromHours(24).TotalSeconds}";
    return Results.Stream(stream => ResizeImageAsync(strImage, stream,
token), "image/jpeg");
});

async Task ResizeImageAsync(string strImage, Stream stream,
CancellationToken token)
{
    var strPath = $"wwwroot/img/{strImage}";
    using var image = await Image.LoadAsync(strPath, token);
    int width = image.Width / 2;
    int height = image.Height / 2;
    image.Mutate(x =>x.Resize(width, height));
    await image.SaveAsync(stream, JpegFormat.Instance, cancellationToken:
token);
}
```

The following example streams an image from Azure Blob storage:

C#

```
app.MapGet("/stream-image/{containerName}/{blobName}",
    async (string blobName, string containerName, CancellationToken token)
```

```
=>
{
    var conStr = builder.Configuration["blogConStr"];
    BlobContainerClient blobContainerClient = new
BlobContainerClient(conStr, containerName);
    BlobClient blobClient = blobContainerClient.GetBlobClient(blobName);
    return Results.Stream(await blobClient.OpenReadAsync(cancellationToken:
token), "image/jpeg");
});
```

The following example streams a video from an Azure Blob:

C#

```
// GET /stream-video/videos/earth.mp4
app.MapGet("/stream-video/{containerName}/{blobName}",
     async (HttpContext http, CancellationToken token, string blobName,
string containerName) =>
{
    var conStr = builder.Configuration["blogConStr"];
    BlobContainerClient blobContainerClient = new
BlobContainerClient(conStr, containerName);
    BlobClient blobClient = blobContainerClient.GetBlobClient(blobName);

    var properties = await blobClient.GetPropertiesAsync(cancellationToken:
token);

    DateTimeOffset lastModified = properties.Value.LastModified;
    long length = properties.Value.ContentLength;

    long etagHash = lastModified.ToFileTime() ^ length;
    var entityTag = new EntityTagHeaderValue('\"' +
Convert.ToString(etagHash, 16) + '\"');

    http.Response.Headers.CacheControl = $"public,max-age=
{TimeSpan.FromHours(24).TotalSeconds}";

    return Results.Stream(await blobClient.OpenReadAsync(cancellationToken:
token),
        contentType: "video/mp4",
        lastModified: lastModified,
        entityTag: entityTag,
        enableRangeProcessing: true);
});
```

Redirect

C#

```
app.MapGet("/old-path", () => Results.Redirect("/new-path"));
```

File

```
C#
```

```
app.MapGet("/download", () => Results.File("myfile.text"));
```

HttpResult interfaces

The following interfaces in the Microsoft.AspNetCore.Http namespace provide a way to detect the `IResult` type at runtime, which is a common pattern in filter implementations:

- IContentTypeHttpResult
- IFileHttpResult
- INestedHttpResult
- IStatusCodeHttpResult
- IValueHttpResult
- IValueHttpResult<TValue>

Here's an example of a filter that uses one of these interfaces:

```
C#
```

```
app.MapGet("/weatherforecast", (int days) =>
{
    if (days <= 0)
    {
        return Results.BadRequest();
    }

    var forecast = Enumerable.Range(1, days).Select(index =>
       new WeatherForecast(DateTime.Now.AddDays(index),
Random.Shared.Next(-20, 55), "Cool"))
        .ToArray();
    return Results.Ok(forecast);
}).
AddEndpointFilter(async (context, next) =>
{
    var result = await next(context);

    return result switch
    {
        IValueHttpResult<WeatherForecast[]> weatherForecastResult => new
WeatherHttpResult(weatherForecastResult.Value),
        _ => result
    };
});
```

For more information, see Filters in Minimal API apps and IResult implementation types.

Customizing responses

Applications can control responses by implementing a custom IResult type. The following code is an example of an HTML result type:

C#

```
using System.Net.Mime;
using System.Text;
static class ResultsExtensions
{
    public static IResult Html(this IResultExtensions resultExtensions,
string html)
    {
        ArgumentNullException.ThrowIfNull(resultExtensions);

        return new HtmlResult(html);
    }
}

class HtmlResult : IResult
{
    private readonly string _html;

    public HtmlResult(string html)
    {
        _html = html;
    }

    public Task ExecuteAsync(HttpContext httpContext)
    {
        httpContext.Response.ContentType = MediaTypeNames.Text.Html;
        httpContext.Response.ContentLength =
Encoding.UTF8.GetByteCount(_html);
        return httpContext.Response.WriteAsync(_html);
    }
}
```

We recommend adding an extension method to Microsoft.AspNetCore.Http.IResultExtensions to make these custom results more discoverable.

C#

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

app.MapGet("/html", () => Results.Extensions.Html(@$"<!doctype html>
```

```
<html>
    <head><title>miniHTML</title></head>
    <body>
        <h1>Hello World</h1>
        <p>The time on the server is {DateTime.Now:O}</p>
    </body>
</html>"));

app.Run();
```

Also, a custom `IResult` type can provide its own annotation by implementing the IEndpointMetadataProvider interface. For example, the following code adds an annotation to the preceding `HtmlResult` type that describes the response produced by the endpoint.

C#

```
class HtmlResult : IResult, IEndpointMetadataProvider
{
    private readonly string _html;

    public HtmlResult(string html)
    {
        _html = html;
    }

    public Task ExecuteAsync(HttpContext httpContext)
    {
        httpContext.Response.ContentType = MediaTypeNames.Text.Html;
        httpContext.Response.ContentLength =
Encoding.UTF8.GetByteCount(_html);
        return httpContext.Response.WriteAsync(_html);
    }

    public static void PopulateMetadata(MethodInfo method, EndpointBuilder
builder)
    {
        builder.Metadata.Add(new ProducesHtmlMetadata());
    }
}
```

The `ProducesHtmlMetadata` is an implementation of IProducesResponseTypeMetadata that defines the produced response content type `text/html` and the status code `200 OK`.

C#

```
internal sealed class ProducesHtmlMetadata : IProducesResponseTypeMetadata
{
    public Type? Type => null;
```

```
    public int StatusCode => 200;

    public IEnumerable<string> ContentTypes { get; } = new[] {
MediaTypeNames.Text.Html };
}
```

An alternative approach is using the Microsoft.AspNetCore.Mvc.ProducesAttribute to describe the produced response. The following code changes the `PopulateMetadata` method to use `ProducesAttribute`.

C#

```
public static void PopulateMetadata(MethodInfo method, EndpointBuilder
builder)
{
    builder.Metadata.Add(new ProducesAttribute(MediaTypeNames.Text.Html));
}
```

Configure JSON serialization options

By default, minimal API apps use Web defaults options during JSON serialization and deserialization.

Configure JSON serialization options globally

Options can be configured globally for an app by invoking ConfigureHttpJsonOptions. The following example includes public fields and formats JSON output.

C#

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

builder.Services.ConfigureHttpJsonOptions(options => {
    options.SerializerOptions.WriteIndented = true;
    options.SerializerOptions.IncludeFields = true;
});

var app = builder.Build();

app.MapPost("/", (Todo todo) => {
    if (todo is not null) {
        todo.Name = todo.NameField;
    }
    return todo;
});

app.Run();
```

```
class Todo {
    public string? Name { get; set; }
    public string? NameField;
    public bool IsComplete { get; set; }
}
// If the request body contains the following JSON:
//
// {"nameField":"Walk dog", "isComplete":false}
//
// The endpoint returns the following JSON:
//
// {
//    "name":"Walk dog",
//    "nameField":"Walk dog",
//    "isComplete":false
// }
```

Since fields are included, the preceding code reads `NameField` and includes it in the output JSON.

Configure JSON serialization options for an endpoint

To configure serialization options for an endpoint, invoke Results.Json and pass to it a JsonSerializerOptions object, as shown in the following example:

C#

```
using System.Text.Json;

var app = WebApplication.Create();

var options = new JsonSerializerOptions(JsonSerializerDefaults.Web)
    { WriteIndented = true };

app.MapGet("/", () =>
    Results.Json(new Todo { Name = "Walk dog", IsComplete = false },
options));

app.Run();

class Todo
{
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
// The endpoint returns the following JSON:
//
// {
//   "name":"Walk dog",
```

```
//   "isComplete":false
// }
```

As an alternative, use an overload of WriteAsJsonAsync that accepts a JsonSerializerOptions object. The following example uses this overload to format the output JSON:

C#

```
using System.Text.Json;

var app = WebApplication.Create();

var options = new JsonSerializerOptions(JsonSerializerDefaults.Web) {
    WriteIndented = true };

app.MapGet("/", (HttpContext context) =>
    context.Response.WriteAsJsonAsync<Todo>(
        new Todo { Name = "Walk dog", IsComplete = false }, options));

app.Run();

class Todo
{
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
// The endpoint returns the following JSON:
//
// {
//   "name":"Walk dog",
//   "isComplete":false
// }
```

Additional Resources

- Authentication and authorization in minimal APIs

Filters in Minimal API apps

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.

By Fiyaz Bin Hasan, Martin Costello, and Rick Anderson

Minimal API filters allow developers to implement business logic that supports:

- Running code before and after the endpoint handler.
- Inspecting and modifying parameters provided during an endpoint handler invocation.
- Intercepting the response behavior of an endpoint handler.

Filters can be helpful in the following scenarios:

- Validating the request parameters and body that are sent to an endpoint.
- Logging information about the request and response.
- Validating that a request is targeting a supported API version.

Filters can be registered by providing a Delegate that takes a EndpointFilterInvocationContext and returns a EndpointFilterDelegate. The `EndpointFilterInvocationContext` provides access to the `HttpContext` of the request and an `Arguments` list indicating the arguments passed to the handler in the order in which they appear in the declaration of the handler.

C#

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

var app = builder.Build();

string ColorName(string color) => $"Color specified: {color}!";

app.MapGet("/colorSelector/{color}", ColorName)
    .AddEndpointFilter(async (invocationContext, next) =>
    {
        var color = invocationContext.GetArgument<string>(0);
```

```
        if (color == "Red")
        {
            return Results.Problem("Red not allowed!");
        }
        return await next(invocationContext);
    });

app.Run();
```

The preceding code:

- Calls the `AddEndpointFilter` extension method to add a filter to the `/colorSelector/{color}` endpoint.
- Returns the color specified except for the value `"Red"`.
- Returns Results.Problem when the `/colorSelector/Red` is requested.
- Uses `next` as the `EndpointFilterDelegate` and `invocationContext` as the `EndpointFilterInvocationContext` to invoke the next filter in the pipeline or the request delegate if the last filter has been invoked.

The filter is run before the endpoint handler. When multiple `AddEndpointFilter` invocations are made on a handler:

- Filter code called before the `EndpointFilterDelegate` (`next`) is called are executed in order of First In, First Out (FIFO) order.
- Filter code called after the `EndpointFilterDelegate` (`next`) is called are executed in order of First In, Last Out (FILO) order.

C#

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

var app = builder.Build();

app.MapGet("/", () =>
    {
        app.Logger.LogInformation("             Endpoint");
        return "Test of multiple filters";
    })
    .AddEndpointFilter(async (efiContext, next) =>
    {
        app.Logger.LogInformation("Before first filter");
        var result = await next(efiContext);
        app.Logger.LogInformation("After first filter");
        return result;
    })
    .AddEndpointFilter(async (efiContext, next) =>
    {
```

```
        app.Logger.LogInformation(" Before 2nd filter");
        var result = await next(efiContext);
        app.Logger.LogInformation(" After 2nd filter");
        return result;
    })
    .AddEndpointFilter(async (efiContext, next) =>
    {
        app.Logger.LogInformation("     Before 3rd filter");
        var result = await next(efiContext);
        app.Logger.LogInformation("     After 3rd filter");
        return result;
    });

app.Run();
```

In the preceding code, the filters and endpoint log the following output:

.NET CLI

```
Before first filter
    Before 2nd filter
        Before 3rd filter
            Endpoint
        After 3rd filter
    After 2nd filter
After first filter
```

The following code uses filters that implement the `IEndpointFilter` interface:

C#

```
using Filters.EndpointFilters;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () =>
    {
        app.Logger.LogInformation("Endpoint");
        return "Test of multiple filters";
    })
    .AddEndpointFilter<AEndpointFilter>()
    .AddEndpointFilter<BEndpointFilter>()
    .AddEndpointFilter<CEndpointFilter>();

app.Run();
```

In the preceding code, the filters and handlers logs show the order they are run:

.NET CLI

```
AEndpointFilter Before next
BEndpointFilter Before next
CEndpointFilter Before next
      Endpoint
CEndpointFilter After next
BEndpointFilter After next
AEndpointFilter After next
```

Filters implementing the `IEndpointFilter` interface are shown in the following example:

C#

```
namespace Filters.EndpointFilters;

public abstract class ABCEndpointFilters : IEndpointFilter
{
    protected readonly ILogger Logger;
    private readonly string _methodName;

    protected ABCEndpointFilters(ILoggerFactory loggerFactory)
    {
        Logger = loggerFactory.CreateLogger<ABCEndpointFilters>();
        _methodName = GetType().Name;
    }

    public virtual async ValueTask<object?>
InvokeAsync(EndpointFilterInvocationContext context,
        EndpointFilterDelegate next)
    {
        Logger.LogInformation("{MethodName} Before next", _methodName);
        var result = await next(context);
        Logger.LogInformation("{MethodName} After next", _methodName);
        return result;
    }
}

class AEndpointFilter : ABCEndpointFilters
{
    public AEndpointFilter(ILoggerFactory loggerFactory) :
base(loggerFactory) { }
}

class BEndpointFilter : ABCEndpointFilters
{
    public BEndpointFilter(ILoggerFactory loggerFactory) :
base(loggerFactory) { }
}

class CEndpointFilter : ABCEndpointFilters
{
```

```
    public CEndpointFilter(ILoggerFactory loggerFactory) :
base(loggerFactory) { }
}
```

Validate an object with a filter

Consider a filter that validates a `Todo` object:

C#

```
app.MapPut("/todoitems/{id}", async (Todo inputTodo, int id, 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();
}).AddEndpointFilter(async (efiContext, next) =>
{
    var tdparam = efiContext.GetArgument<Todo>(0);

    var validationError = Utilities.IsValid(tdparam);

    if (!string.IsNullOrEmpty(validationError))
    {
        return Results.Problem(validationError);
    }
    return await next(efiContext);
});
```

In the preceding code:

- The `EndpointFilterInvocationContext` object provides access to the parameters associated with a particular request issued to the endpoint via the `GetArguments` method.
- The filter is registered using a `delegate` that takes a `EndpointFilterInvocationContext` and returns a `EndpointFilterDelegate`.

In addition to being passed as delegates, filters can be registered by implementing the `IEndpointFilter` interface. The following code shows the preceding filter encapsulated in a class which implements `IEndpointFilter`:

C#

```
public class TodoIsValidFilter : IEndpointFilter
{
    private ILogger _logger;

    public TodoIsValidFilter(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<TodoIsValidFilter>();
    }

    public async ValueTask<object?>
InvokeAsync(EndpointFilterInvocationContext efiContext,
        EndpointFilterDelegate next)
    {
        var todo = efiContext.GetArgument<Todo>(0);

        var validationError = Utilities.IsValid(todo!);

        if (!string.IsNullOrEmpty(validationError))
        {
            _logger.LogWarning(validationError);
            return Results.Problem(validationError);
        }
        return await next(efiContext);
    }
}
```

Filters that implement the `IEndpointFilter` interface can resolve dependencies from Dependency Injection(DI), as shown in the previous code. Although filters can resolve dependencies from DI, filters themselves can ***not*** be resolved from DI.

The `ToDoIsValidFilter` is applied to the following endpoints:

C#

```
app.MapPut("/todoitems2/{id}", async (Todo inputTodo, int id, 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();
}).AddEndpointFilter<TodoIsValidFilter>();

app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
```

```
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
}).AddEndpointFilter<TodoIsValidFilter>();
```

The following filter validates the `Todo` object and modifies the `Name` property:

C#

```
public class TodoIsValidUcFilter : IEndpointFilter
{
    public async ValueTask<object?>
InvokeAsync(EndpointFilterInvocationContext efiContext,
        EndpointFilterDelegate next)
    {
        var todo = efiContext.GetArgument<Todo>(0);
        todo.Name = todo.Name!.ToUpper();

        var validationError = Utilities.IsValid(todo!);

        if (!string.IsNullOrEmpty(validationError))
        {
            return Results.Problem(validationError);
        }
        return await next(efiContext);
    }
}
```

Register a filter using an endpoint filter factory

In some scenarios, it might be necessary to cache some of the information provided in the MethodInfo in a filter. For example, let's assume that we wanted to verify that the handler an endpoint filter is attached to has a first parameter that evaluates to a `Todo` type.

C#

```
app.MapPut("/todoitems/{id}", async (Todo inputTodo, int id, 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();
}).AddEndpointFilterFactory((filterFactoryContext, next) =>
{
    var parameters = filterFactoryContext.MethodInfo.GetParameters();
    if (parameters.Length >= 1 && parameters[0].ParameterType ==
typeof(Todo))
    {
        return async invocationContext =>
        {
            var todoParam = invocationContext.GetArgument<Todo>(0);

            var validationError = Utilities.IsValid(todoParam);

            if (!string.IsNullOrEmpty(validationError))
            {
                return Results.Problem(validationError);
            }
            return await next(invocationContext);
        };
    }
    return invocationContext => next(invocationContext);
});
```

In the preceding code:

- The `EndpointFilterFactoryContext` object provides access to the MethodInfo associated with the endpoint's handler.
- The signature of the handler is examined by inspecting `MethodInfo` for the expected type signature. If the expected signature is found, the validation filter is registered onto the endpoint. This factory pattern is useful to register a filter that depends on the signature of the target endpoint handler.
- If a matching signature isn't found, then a pass-through filter is registered.

Register a filter on controller actions

In some scenarios, it might be necessary to apply the same filter logic for both route-handler based endpoints and controller actions. For this scenario, it is possible to invoke `AddEndpointFilter` on `ControllerActionEndpointConventionBuilder` to support executing the same filter logic on actions and endpoints.

C#

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

var app = builder.Build();

app.MapController()
    .AddEndpointFilter(async (efiContext, next) =>
```

```
    {
        efiContext.HttpContext.Items["endpointFilterCalled"] = true;
        var result = await next(efiContext);
        return result;
    });

app.Run();
```

Additional Resources

- View or download sample code (how to download)
- ValidationFilterRouteHandlerBuilderExtensions Validation extension methods.
- Tutorial: Create a minimal API with ASP.NET Core
- Authentication and authorization in minimal APIs

Unit and integration tests in Minimal API apps

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.

By Fiyaz Bin Hasan, and Rick Anderson

Introduction to integration tests

Integration tests evaluate an app's components on a broader level than unit tests. Unit tests are used to test isolated software components, such as individual class methods. Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:

- Database
- File system
- Network appliances
- Request-response pipeline

Unit tests use fabricated components, known as *fakes* or *mock objects*, in place of infrastructure components.

In contrast to unit tests, integration tests:

- Use the actual components that the app uses in production.
- Require more code and data processing.
- Take longer to run.

Therefore, limit the use of integration tests to the most important infrastructure scenarios. If a behavior can be tested using either a unit test or an integration test,

choose the unit test.

In discussions of integration tests, the tested project is frequently called the ***System Under Test***, or "**SUT**" for short. "SUT" is used throughout this article to refer to the ASP.NET Core app being tested.

Don't write integration tests for every permutation of data and file access with databases and file systems. Regardless of how many places across an app interact with databases and file systems, a focused set of read, write, update, and delete integration tests are usually capable of adequately testing database and file system components. Use unit tests for routine tests of method logic that interact with these components. In unit tests, the use of infrastructure fakes or mocks result in faster test execution.

ASP.NET Core integration tests

Integration tests in ASP.NET Core require the following:

- A test project is used to contain and execute the tests. The test project has a reference to the SUT.
- The test project creates a test web host for the SUT and uses a test server client to handle requests and responses with the SUT.
- A test runner is used to execute the tests and report the test results.

Integration tests follow a sequence of events that include the usual *Arrange*, *Act*, and *Assert* test steps:

1. The SUT's web host is configured.
2. A test server client is created to submit requests to the app.
3. The *Arrange* test step is executed: The test app prepares a request.
4. The *Act* test step is executed: The client submits the request and receives the response.
5. The *Assert* test step is executed: The *actual* response is validated as a *pass* or *fail* based on an *expected* response.
6. The process continues until all of the tests are executed.
7. The test results are reported.

Usually, the test web host is configured differently than the app's normal web host for the test runs. For example, a different database or different app settings might be used for the tests.

Infrastructure components, such as the test web host and in-memory test server (TestServer), are provided or managed by the Microsoft.AspNetCore.Mvc.Testing package. Use of this package streamlines test creation and execution.

The `Microsoft.AspNetCore.Mvc.Testing` package handles the following tasks:

- Copies the dependencies file (`.deps`) from the SUT into the test project's `bin` directory.
- Sets the content root to the SUT's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory class to streamline bootstrapping the SUT with `TestServer`.

The unit tests documentation describes how to set up a test project and test runner, along with detailed instructions on how to run tests and recommendations for how to name tests and test classes.

Separate unit tests from integration tests into different projects. Separating the tests:

- Helps ensure that infrastructure testing components aren't accidentally included in the unit tests.
- Allows control over which set of tests are run.

The sample code on GitHub provides an example of unit and integration tests on a Minimal API app.

IResult implementation types

Public IResult implementation types in the Microsoft.AspNetCore.Http.HttpResults namespace can be used to unit test minimal route handlers when using named methods instead of lambdas.

The following code uses the NotFound<TValue> class:

C#

```
[Fact]
public async Task GetTodoReturnsNotFoundIfNotExists()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    // Act
    var result = await TodoEndpointsV1.GetTodo(1, context);

    //Assert
    Assert.IsType<Results<Ok<Todo>, NotFound>>(result);

    var notFoundResult = (NotFound) result.Result;
```

```
    Assert.NotNull(notFoundResult);
}
```

The following code uses the Ok<TValue> class:

C#

```
[Fact]
public async Task GetTodoReturnsTodoFromDatabase()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    context.Todos.Add(new Todo
    {
        Id = 1,
        Title = "Test title",
        Description = "Test description",
        IsDone = false
    });

    await context.SaveChangesAsync();

    // Act
    var result = await TodoEndpointsV1.GetTodo(1, context);

    //Assert
    Assert.IsType<Results<Ok<Todo>, NotFound>>(result);

    var okResult = (Ok<Todo>)result.Result;

    Assert.NotNull(okResult.Value);
    Assert.Equal(1, okResult.Value.Id);
}
```

Additional Resources

- Basic authentication tests is not a .NET repository but was written by a member of the .NET team. It provides examples of basic authentication testing.
- View or download sample code
- Authentication and authorization in minimal APIs
- Use port tunneling Visual Studio to debug web APIs
- Test controller logic in ASP.NET Core
- Razor Pages unit tests in ASP.NET Core

Middleware in Minimal API apps

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**.

WebApplication automatically adds the following middleware in Minimal API applications depending on certain conditions:

- UseDeveloperExceptionPage is added first when the HostingEnvironment is `"Development"`.
- UseRouting is added second if user code didn't already call `UseRouting` and if there are endpoints configured, for example `app.MapGet`.
- UseEndpoints is added at the end of the middleware pipeline if any endpoints are configured.
- UseAuthentication is added immediately after `UseRouting` if user code didn't already call `UseAuthentication` and if IAuthenticationSchemeProvider can be detected in the service provider. `IAuthenticationSchemeProvider` is added by default when using AddAuthentication, and services are detected using IServiceProviderIsService.
- UseAuthorization is added next if user code didn't already call `UseAuthorization` and if IAuthorizationHandlerProvider can be detected in the service provider. `IAuthorizationHandlerProvider` is added by default when using AddAuthorization, and services are detected using `IServiceProviderIsService`.
- User configured middleware and endpoints are added between `UseRouting` and `UseEndpoints`.

The following code is effectively what the automatic middleware being added to the app produces:

C#

```
if (isDevelopment)
{
    app.UseDeveloperExceptionPage();
}
```

```
app.UseRouting();

if (isAuthenticationConfigured)
{
    app.UseAuthentication();
}

if (isAuthorizationConfigured)
{
    app.UseAuthorization();
}

// user middleware/endpoints
app.CustomMiddleware(...);
app.MapGet("/", () => "hello world");
// end user middleware/endpoints

app.UseEndpoints(e => {});
```

In some cases, the default middleware configuration isn't correct for the app and requires modification. For example, UseCors should be called before UseAuthentication and UseAuthorization. The app needs to call `UseAuthentication` and `UseAuthorization` if `UseCors` is called:

C#

```
app.UseCors();
app.UseAuthentication();
app.UseAuthorization();
```

If middleware should be run before route matching occurs, UseRouting should be called and the middleware should be placed before the call to `UseRouting`. UseEndpoints isn't required in this case as it is automatically added as described previously:

C#

```
app.Use((context, next) =>
{
    return next(context);
});

app.UseRouting();

// other middleware and endpoints
```

When adding a terminal middleware:

- The middleware must be added after `UseEndpoints`.
- The app needs to call `UseRouting` and `UseEndpoints` so that the terminal middleware can be placed at the correct location.

C#

```
app.UseRouting();

app.MapGet("/", () => "hello world");

app.UseEndpoints(e => {});

app.Run(context =>
{
    context.Response.StatusCode = 404;
    return Task.CompletedTask;
});
```

Terminal middleware is middleware that runs if no endpoint handles the request.

For information on antiforgery middleware in Minimal APIs, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core

For more information about middleware see ASP.NET Core Middleware, and the list of built-in middleware that can be added to applications.

For more information about Minimal APIs see Minimal APIs overview.

How to handle errors in Minimal API apps

Article • 06/18/2024

> **ⓘ 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**.

With contributions by David Acker

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

Exceptions

In a Minimal API app, there are two different built-in centralized mechanisms to handle unhandled exceptions:

- Developer Exception Page middleware (For use in the **Development environment only**.)
- Exception handler middleware

This section refers to the following sample app to demonstrate ways to handle exceptions in a Minimal API. It throws an exception when the endpoint `/exception` is requested:

C#

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

app.MapGet("/exception", () =>
{
    throw new InvalidOperationException("Sample Exception");
});

app.MapGet("/", () => "Test by calling /exception");
```

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

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.

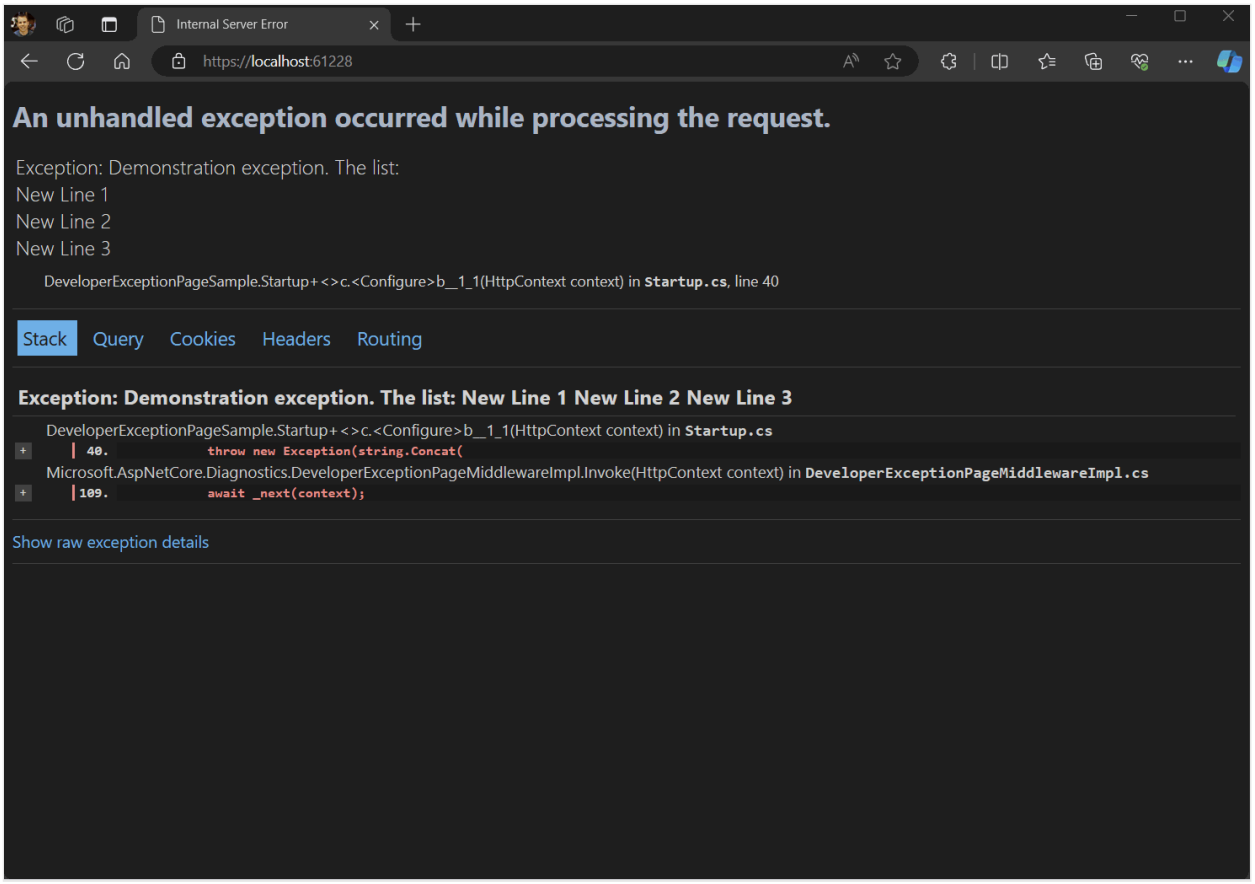
> ⚠ **Warning**
>
> Don't enable the Developer Exception Page **unless the app is running in the Development environment**. Don't share detailed exception information publicly when the app runs in production. For more information on configuring environments, see **Use multiple environments in ASP.NET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

- Run the sample app in the Development environment.
- Go to the `/exception` endpoint.

Exception handler

In non-development environments, use the Exception Handler Middleware to produce an error payload. To configure the `Exception Handler Middleware`, call UseExceptionHandler.

For example, the following code changes the app to respond with an RFC 7807 - compliant payload to the client. For more information, see the Problem Details section later in this article.

C#

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

app.UseExceptionHandler(exceptionHandlerApp
    => exceptionHandlerApp.Run(async context
        => await Results.Problem()
                      .ExecuteAsync(context)));

app.MapGet("/exception", () =>
{
    throw new InvalidOperationException("Sample Exception");
});

app.MapGet("/", () => "Test by calling /exception");

app.Run();
```

Client and Server error responses

Consider the following Minimal API app.

C#

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

app.MapGet("/users/{id:int}", (int id)
    => id <= 0 ? Results.BadRequest() : Results.Ok(new User(id)));

app.MapGet("/", () => "Test by calling /users/{id:int}");

app.Run();

public record User(int Id);
```

The `/users` endpoint produces `200 OK` with a `json` representation of `User` when `id` is greater than `0`, otherwise a `400 BAD REQUEST` status code without a response body. For more information about creating a response, see Create responses in Minimal API apps.

The Status Code Pages middleware can be configured to produce a common body content, **when empty**, for all HTTP client (`400`-`499`) or server (`500` -`599`) responses. The middleware is configured by calling the UseStatusCodePages extension method.

For example, the following example changes the app to respond with an RFC 7807 - compliant payload to the client for all client and server responses, including routing errors (for example, `404 NOT FOUND`). For more information, see the Problem Details section.

C#

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

app.UseStatusCodePages(async statusCodeContext
    => await Results.Problem(statusCode:
statusCodeContext.HttpContext.Response.StatusCode)
                  .ExecuteAsync(statusCodeContext.HttpContext));
app.MapGet("/users/{id:int}", (int id)
    => id <= 0 ? Results.BadRequest() : Results.Ok(new User(id)) );

app.MapGet("/", () => "Test by calling /users/{id:int}");

app.Run();

public record User(int Id);
```

Problem details

Problem Details are not the only response format to describe an HTTP API error, however, they are commonly used to report errors for HTTP APIs.

The problem details service implements the IProblemDetailsService interface, which supports creating problem details in ASP.NET Core. The AddProblemDetails(IServiceCollection) extension method on IServiceCollection registers the default `IProblemDetailsService` implementation.

In ASP.NET Core apps, the following middleware generates problem details HTTP responses when `AddProblemDetails` is called, except when the Accept request HTTP header doesn't include one of the content types supported by the registered IProblemDetailsWriter (default: `application/json`):

- ExceptionHandlerMiddleware: Generates a problem details response when a custom handler is not defined.
- StatusCodePagesMiddleware: Generates a problem details response by default.
- DeveloperExceptionPageMiddleware: Generates a problem details response in development when the `Accept` request HTTP header doesn't include `text/html`.

Minimal API apps can be configured to generate problem details response for all HTTP client and server error responses that ***don't have body content yet*** by using the `AddProblemDetails` extension method.

The following code configures the app to generate problem details:

C#

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

var app = builder.Build();

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

app.MapGet("/users/{id:int}", (int id)
    => id <= 0 ? Results.BadRequest() : Results.Ok(new User(id)));

app.MapGet("/", () => "Test by calling /users/{id:int}");

app.Run();

public record User(int Id);
```

For more information on using `AddProblemDetails`, see Problem Details

IProblemDetailsService fallback

In the following code, `httpContext.Response.WriteAsync("Fallback: An error occurred.")` returns an error if the IProblemDetailsService implementation isn't able to generate a ProblemDetails:

C#

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

var app = builder.Build();

app.UseExceptionHandler(exceptionHandlerApp =>
```

```
{
    exceptionHandlerApp.Run(async httpContext =>
    {
        var pds =
httpContext.RequestServices.GetService<IProblemDetailsService>();
        if (pds == null
            || !await pds.TryWriteAsync(new() { HttpContext = httpContext
}))
        {
            // Fallback behavior
            await httpContext.Response.WriteAsync("Fallback: An error
occurred.");
        }
    });
});

app.MapGet("/exception", () =>
{
    throw new InvalidOperationException("Sample Exception");
});

app.MapGet("/", () => "Test by calling /exception");

app.Run();
```

The preceding code:

- Writes an error message with the fallback code if the `problemDetailsService` is unable to write a `ProblemDetails`. For example, an endpoint where the Accept request header specifies a media type that the `DefaulProblemDetailsWriter` does not support.
- Uses the Exception Handler Middleware.

The following sample is similar to the preceding except that it calls the Status Code Pages middleware.

C#

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

var app = builder.Build();

app.UseStatusCodePages(statusCodeHandlerApp =>
{
    statusCodeHandlerApp.Run(async httpContext =>
    {
        var pds =
httpContext.RequestServices.GetService<IProblemDetailsService>();
        if (pds == null
            || !await pds.TryWriteAsync(new() { HttpContext = httpContext
```

```
}))
        {
            // Fallback behavior
            await httpContext.Response.WriteAsync("Fallback: An error
occurred.");
        }
    });
});

app.MapGet("/users/{id:int}", (int id) =>
{
    return id <= 0 ? Results.BadRequest() : Results.Ok(new User(id));
});

app.MapGet("/", () => "Test by calling /users/{id:int}");

app.Run();

public record User(int Id);
```

Authentication and authorization in minimal APIs

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 support all the authentication and authorization options available in ASP.NET Core and provide some additional functionality to improve the experience working with authentication.

Key concepts in authentication and authorization

Authentication is the process of determining a user's identity. Authorization is the process of determining whether a user has access to a resource. Both authentication and authorization scenarios share similar implementation semantics in ASP.NET Core. Authentication is handled by the authentication service, IAuthenticationService, which is used by authentication middleware. Authorization is handled by the authorization service, IAuthorizationService, which is used by the authorization middleware.

The authentication service uses registered authentication handlers to complete authentication-related actions. For example, an authentication-related action is authenticating a user or signing out a user. Authentication schemes are names that are used to uniquely identify an authentication handler and its configuration options. Authentication handlers are responsible for implementing the strategies for authentication and generating a user's claims given a particular authentication strategy, such as OAuth or OIDC. The configuration options are unique to the strategy as well and provide the handler with configuration that affects authentication behavior, such as redirect URIs.

There are two strategies for determining user access to resources in the authorization layer:

- Role-based strategies determine a user's access based on the role they are assigned, such as `Administrator` or `User`. For more information on role-based authorization, see role-based authorization documentation.
- Claim-based strategies determine a user's access based on claims that are issued by a central authority. For more information on claim-based authorization, see claim-based authorization documentation.

In ASP.NET Core, both strategies are captured into an authorization requirement. The authorization service leverages authorization handlers to determine whether or not a particular user meets the authorization requirements applied onto a resource.

Enabling authentication in minimal apps

To enable authentication, call AddAuthentication to register the required authentication services on the app's service provider.

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthentication();
var app = builder.Build();

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

Typically, a specific authentication strategy is used. In the following sample, the app is configured with support for JWT bearer-based authentication. This example makes use of the APIs available in the Microsoft.AspNetCore.Authentication.JwtBearer NuGet package.

C#

```
var builder = WebApplication.CreateBuilder(args);
// Requires Microsoft.AspNetCore.Authentication.JwtBearer
builder.Services.AddAuthentication().AddJwtBearer();
var app = builder.Build();

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

By default, the WebApplication automatically registers the authentication and authorization middlewares if certain authentication and authorization services are enabled. In the following sample, it's not necessary to invoke UseAuthentication or

UseAuthorization to register the middlewares because WebApplication does this automatically after `AddAuthentication` or `AddAuthorization` are called.

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization();
var app = builder.Build();

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

In some cases, such as controlling middleware order, it's necessary to explicitly register authentication and authorization. In the following sample, the authentication middleware runs *after* the CORS middleware has run. For more information on middlewares and this automatic behavior, see Middleware in Minimal API apps.

C#

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

builder.Services.AddCors();
builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization();

var app = builder.Build();

app.UseCors();
app.UseAuthentication();
app.UseAuthorization();

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

Configuring authentication strategy

Authentication strategies typically support a variety of configurations that are loaded via options. Minimal apps support loading options from configuration for the following authentication strategies:

- JWT bearer-based
- OpenID Connection-based

The ASP.NET Core framework expects to find these options under the `Authentication:Schemes:{SchemeName}` section in configuration. In the following sample,

two different schemes, `Bearer` and `LocalAuthIssuer`, are defined with their respective options. The `Authentication:DefaultScheme` option can be used to configure the default authentication strategy that's used.

JSON

```
{
  "Authentication": {
    "DefaultScheme":  "LocalAuthIssuer",
    "Schemes": {
      "Bearer": {
        "ValidAudiences": [
          "https://localhost:7259",
          "http://localhost:5259"
        ],
        "ValidIssuer": "dotnet-user-jwts"
      },
      "LocalAuthIssuer": {
        "ValidAudiences": [
          "https://localhost:7259",
          "http://localhost:5259"
        ],
        "ValidIssuer": "local-auth"
      }
    }
  }
}
```

In `Program.cs`, two JWT bearer-based authentication strategies are registered, with the:

- "Bearer" scheme name.
- "LocalAuthIssuer" scheme name.

"Bearer" is the typical default scheme in JWT-bearer based enabled apps, but the default scheme can be overridden by setting the `DefaultScheme` property as in the preceding example.

The scheme name is used to uniquely identify an authentication strategy and is used as the lookup key when resolving authentication options from config, as shown in the following example:

C#

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

builder.Services.AddAuthentication()
  .AddJwtBearer()
  .AddJwtBearer("LocalAuthIssuer");
```

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

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

Configuring authorization policies in minimal apps

Authentication is used to identify and validate the identity of users against an API. Authorization is used to validate and verify access to resources in an API and is facilitated by the `IAuthorizationService` registered by the `AddAuthorization` extension method. In the following scenario, a `/hello` resource is added that requires a user to present an `admin` role claim with a `greetings_api` scope claim.

Configuring authorization requirements on a resource is a two-step process that requires:

1. Configuring the authorization requirements in a policy globally.
2. Applying individual policies to resources.

In the following code, AddAuthorizationBuilder is invoked which:

- Adds authorization-related services to the DI container.
- Returns an AuthorizationBuilder that can be used to directly register authorization policies.

The code creates a new authorization policy, named `admin_greetings`, that encapsulates two authorization requirements:

- A role-based requirement via RequireRole for users with an `admin` role.
- A claim-based requirement via RequireClaim that the user must provide a `greetings_api` scope claim.

The `admin_greetings` policy is provided as a required policy to the `/hello` endpoint.

C#

```
using Microsoft.Identity.Web;

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

```
builder.Services.AddAuthorizationBuilder()
  .AddPolicy("admin_greetings", policy =>
        policy
            .RequireRole("admin")
            .RequireClaim("scope", "greetings_api"));

var app = builder.Build();

app.MapGet("/hello", () => "Hello world!")
  .RequireAuthorization("admin_greetings");

app.Run();
```

Use `dotnet user-jwts` for development testing

Throughout this article, an app configured with JWT-bearer based authentication is used. JWT bearer-based authentication requires that clients present a token in the request header to validate their identity and claims. Typically, these tokens are issued by a central authority, such as an identity server.

When developing on the local machine, the dotnet user-jwts tool can be used to create bearer tokens.

.NET CLI

```
dotnet user-jwts create
```

> **Note**
>
> When invoked on a project, the tool automatically adds the authentication options matching the generated token to `appsettings.json`.

Tokens can be configured with a variety of customizations. For example, to create a token for the `admin` role and `greetings_api` scope expected by the authorization policy in the preceding code:

.NET CLI

```
dotnet user-jwts create --scope "greetings_api" --role "admin"
```

The generated token can then be sent as part of the header in the testing tool of choice. For example, with curl:

```
```

.NET CLI

```
curl -i -H "Authorization: Bearer {token}" https://localhost:{port}/hello
```

For more information on the `dotnet user-jwts` tool, read the complete documentation.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

OpenAPI support in ASP.NET Core API apps

Article • 10/28/2024

> **ⓘ 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.

ASP.NET Core supports the generation of OpenAPI documents in controller-based and minimal APIs apps. The OpenAPI specification is a programming language-agnostic standard for documenting HTTP APIs. This standard is supported in ASP.NET Core apps through a combination of built-in APIs and open-source libraries. There are three key aspects to OpenAPI integration in an application:

- Generating information about the endpoints in the app.
- Gathering the information into a format that matches the OpenAPI schema.
- Exposing the generated OpenAPI document via a visual UI or a serialized file.

ASP.NET Core apps provide built-in support for generating information about endpoints in an app via the `Microsoft.AspNetCore.OpenApi` package.

The following code is generated by the ASP.NET Core minimal web API template and uses OpenAPI:

C#

```
using Microsoft.AspNetCore.OpenApi;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddOpenApi();

var app = builder.Build();

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

app.UseHttpsRedirection();
```

```
var summaries = new[]
{
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot",
"Sweltering", "Scorching"
};

app.MapGet("/weatherforecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateTime.Now.AddDays(index),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast");

app.Run();

internal record WeatherForecast(DateTime Date, int TemperatureC, string?
Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

In the preceding highlighted code:

- `AddOpenApi` registers services required for OpenAPI document generation into the application's DI container.
- `MapOpenApi` adds an endpoint into the application for viewing the OpenAPI document serialized into JSON. The OpenAPI endpoint is restricted to the development environment to minimize the risk of exposing sensitive information and reduce the vulnerabilities in production.

`Microsoft.AspNetCore.OpenApi` NuGet package

The Microsoft.AspNetCore.OpenApi package provides the following features:

- Support for generating OpenAPI documents at run time and accessing them via an endpoint on the application.
- Support for "transformer" APIs that allow modifying the generated document.

To use the `Microsoft.AspNetCore.OpenApi` package, add it as a PackageReference to a project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <PropertyGroup>
    <OpenApiGenerateDocuments>true</OpenApiGenerateDocuments>
    <OpenApiDocumentsDirectory>$(MSBuildProjectDirectory)
</OpenApiDocumentsDirectory>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="9.0.*-
*" />
    <PackageReference Include="Microsoft.Extensions.ApiDescription.Server"
Version="9.0.*-*">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>

</Project>
```

To learn more about the `Microsoft.AspNetCore.OpenApi` package, see Generate OpenAPI documents.

`Microsoft.Extensions.ApiDescription.Server` NuGet package

The Microsoft.Extensions.ApiDescription.Server package provides support for generating OpenAPI documents at build time and serializing them.

To use `Microsoft.Extensions.ApiDescription.Server`, add it as a PackageReference to a project file. Document generation at build time is enabled by setting the `OpenApiGenerateDocuments` property. By default, the generated OpenAPI document is saved to the `obj` directory, but you can customize the output directory by setting the `OpenApiDocumentsDirectory` property.

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <PropertyGroup>
    <OpenApiGenerateDocuments>true</OpenApiGenerateDocuments>
    <OpenApiDocumentsDirectory>$(MSBuildProjectDirectory)
</OpenApiDocumentsDirectory>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="9.0.*-
*" />
    <PackageReference Include="Microsoft.Extensions.ApiDescription.Server"
Version="9.0.*-*">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>

</Project>
```

ASP.NET Core OpenAPI source code on GitHub

- AddOpenApi
- OpenApiDocumentService
- OpenApiOptions

Additional Resources

- Authentication and authorization in minimal APIs

Generate OpenAPI documents

Article • 12/11/2024

The Microsoft.AspNetCore.OpenApi package provides built-in support for OpenAPI document generation in ASP.NET Core. The package provides the following features:

- Support for generating OpenAPI documents at run time and accessing them via an endpoint on the application.
- Support for "transformer" APIs that allow modifying the generated document.
- Support for generating multiple OpenAPI documents from a single app.
- Takes advantage of JSON schema support provided by System.Text.Json.
- Is compatible with native AoT.

Package installation

Install the `Microsoft.AspNetCore.OpenApi` package:

Visual Studio

Run the following command from the **Package Manager Console**:

PowerShell

```
Install-Package Microsoft.AspNetCore.OpenApi
```

Configure OpenAPI document generation

The following code:

- Adds OpenAPI services.
- Enables the endpoint for viewing the OpenAPI document in JSON format.

C#

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

builder.Services.AddOpenApi();

var app = builder.Build();

app.MapOpenApi();
```

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

app.Run();
```

Launch the app and navigate to `https://localhost:<port>/openapi/v1.json` to view the generated OpenAPI document.

Options to Customize OpenAPI document generation

The following sections demonstrate how to customize OpenAPI document generation.

Customize the OpenAPI document name

Each OpenAPI document in an app has a unique name. The default document name that is registered is `v1`.

C#

```
builder.Services.AddOpenApi(); // Document name is v1
```

The document name can be modified by passing the name as a parameter to the `AddOpenApi` call.

C#

```
builder.Services.AddOpenApi("internal"); // Document name is internal
```

The document name surfaces in several places in the OpenAPI implementation.

When fetching the generated OpenAPI document, the document name is provided as the `documentName` parameter argument in the request. The following requests resolve the `v1` and `internal` documents.

Bash

```
GET http://localhost:5000/openapi/v1.json
GET http://localhost:5000/openapi/internal.json
```

Customize the OpenAPI version of a generated document

By default, OpenAPI document generation creates a document that is compliant with v3.0 of the OpenAPI specification. The following code demonstrates how to modify the default version of the OpenAPI document:

```
C#

builder.Services.AddOpenApi(options =>
{
    options.OpenApiVersion = OpenApiSpecVersion.OpenApi2_0;
});
```

Customize the OpenAPI endpoint route

By default, the OpenAPI endpoint registered via a call to MapOpenApi exposes the document at the `/openapi/{documentName}.json` endpoint. The following code demonstrates how to customize the route at which the OpenAPI document is registered:

```
C#

app.MapOpenApi("/openapi/{documentName}/openapi.json");
```

It's possible, but not recommended, to remove the `documentName` route parameter from the endpoint route. When the `documentName` route parameter is removed from the endpoint route, the framework attempts to resolve the document name from the query parameter. Not providing the `documentName` in either the route or query can result in unexpected behavior.

Customize the OpenAPI endpoint

Because the OpenAPI document is served via a route handler endpoint, any customization that is available to standard minimal endpoints is available to the OpenAPI endpoint.

Limit OpenAPI document access to authorized users

The OpenAPI endpoint doesn't enable any authorization checks by default. However, authorization checks can be applied to the OpenAPI document. In the following code, access to the OpenAPI document is limited to those with the `tester` role:

```
C#
```

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.OpenApi;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization(o =>
{
    o.AddPolicy("ApiTesterPolicy", b => b.RequireRole("tester"));
});
builder.Services.AddOpenApi();

var app = builder.Build();

app.MapOpenApi()
    .RequireAuthorization("ApiTesterPolicy");

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

app.Run();
```

Cache generated OpenAPI document

The OpenAPI document is regenerated every time a request to the OpenAPI endpoint is sent. Regeneration enables transformers to incorporate dynamic application state into their operation. For example, regenerating a request with details of the HTTP context. When applicable, the OpenAPI document can be cached to avoid executing the document generation pipeline on each HTTP request.

C#

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.OpenApi;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder();

builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(policy =>
policy.Expire(TimeSpan.FromMinutes(10)));
});
builder.Services.AddOpenApi();

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

```
app.UseOutputCache();

app.MapOpenApi()
    .CacheOutput();

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

app.Run();
```

Generate OpenAPI documents at build-time

In typical web applications, OpenAPI documents are generated at run-time and served via an HTTP request to the application server.

In some scenarios, it's helpful to generate the OpenAPI document during the application's build step. These scenarios include:

- Generating OpenAPI documentation that is committed into source control.
- Generating OpenAPI documentation that is used for spec-based integration testing.
- Generating OpenAPI documentation that is served statically from the web server.

To add support for generating OpenAPI documents at build time, install the `Microsoft.Extensions.ApiDescription.Server` package:

Visual Studio

Run the following command from the **Package Manager Console**:

PowerShell

```
Install-Package Microsoft.Extensions.ApiDescription.Server
```

Upon installation, this package will automatically generate the Open API document(s) associated with the application during build and populate them into the application's output directory.

cli

```
$ dotnet build
$ cat bin/Debug/net9.0/{ProjectName}.json
```

Customizing build-time document generation

Modifying the output directory of the generated Open API file

By default, the generated OpenAPI document will be emitted to the application's output directory. To modify the location of the emitted file, set the target path in the `OpenApiDocumentsDirectory` property.

XML

```
<PropertyGroup>
  <OpenApiDocumentsDirectory>./</OpenApiDocumentsDirectory>
</PropertyGroup>
```

The value of `OpenApiDocumentsDirectory` is resolved relative to the project file. Using the `./` value above will emit the OpenAPI document in the same directory as the project file.

Modifying the output file name

By default, the generated OpenAPI document will have the same name as the application's project file. To modify the name of the emitted file, set the `--file-name` argument in the `OpenApiGenerateDocumentsOptions` property.

XML

```
<PropertyGroup>
  <OpenApiGenerateDocumentsOptions>--file-name my-open-
api</OpenApiGenerateDocumentsOptions>
</PropertyGroup>
```

Selecting the OpenAPI document to generate

Some applications may be configured to emit multiple OpenAPI documents, for various versions of an API or to distinguish between public and internal APIs. By default, the build-time document generator will emit files for all documents that are configured in an application. To only emit for a single document name, set the `--document-name` argument in the `OpenApiGenerateDocumentsOptions` property.

XML

```
<PropertyGroup>
  <OpenApiGenerateDocumentsOptions>--document-name
v2</OpenApiGenerateDocumentsOptions>
</PropertyGroup>
```

Customizing run-time behavior during build-time document generation

Build-time OpenAPI document generation functions by launching the apps entrypoint with a mock server implementation. A mock server is required to produce accurate OpenAPI documents because all information in the OpenAPI document can't be statically analyzed. Because the apps entrypoint is invoked, any logic in the apps startup is invoked. This includes code that injects services into the DI container or reads from configuration. In some scenarios, it's necessary to restrict the code paths that will run when the apps entry point is being invoked from build-time document generation. These scenarios include:

- Not reading from certain configuration strings.
- Not registering database-related services.

In order to restrict these code paths from being invoked by the build-time generation pipeline, they can be conditioned behind a check of the entry assembly:

C#

```
using System.Reflection;

var builder = WebApplication.CreateBuilder(args);

if (Assembly.GetEntryAssembly()?.GetName().Name != "GetDocument.Insider")
{
    builder.AddServiceDefaults();
}

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    app.UseHsts();
}

var myKeyValue = app.Configuration["MyKey"];

app.MapGet("/", () => {
    return Results.Ok($"The value of MyKey is: {myKeyValue}");
})
```

```
    .WithName("TestKey");

app.Run();
```

AddServiceDefaults Adds common .NET Aspire services such as service discovery, resilience, health checks, and OpenTelemetry.

Trimming and Native AOT

OpenAPI in ASP.NET Core supports trimming and native AOT. The following steps create and publish an OpenAPI app with trimming and native AOT:

Create a new ASP.NET Core Web API (Native AOT) project.

Console

```
dotnet new webapiaot
```

Add the Microsoft.AspNetCore.OpenAPI package.

Console

```
dotnet add package Microsoft.AspNetCore.OpenApi --prerelease
```

Update `Program.cs` to enable generating OpenAPI documents.

diff

```
+ builder.Services.AddOpenApi();

var app = builder.Build();

+ app.MapOpenApi();
```

Publish the app.

Console

```
dotnet publish
```

Include OpenAPI metadata in an ASP.NET Core app

Article • 11/06/2024

Include OpenAPI metadata for endpoints

ASP.NET collects metadata from the web app's endpoints and uses it to generate an OpenAPI document. In controller-based apps, metadata is collected from attributes like [EndpointDescription], [HttpPost], and [Produces]. In minimal APIs, metadata can be collected from attributes, but may also be set by using extension methods and other strategies, such as returning TypedResults from route handlers. The following table provides an overview of the metadata collected and the strategies for setting it.

Expand table

Metadata	Attribute	Extension method	Other strategies
summary	[EndpointSummary]	WithSummary	
description	[EndpointDescription]	WithDescription	
tags	[Tags]	WithTags	
operationId	[EndpointName]	WithName	
parameters	[FromQuery], [FromRoute], [FromHeader], [FromForm]		
parameter description	[EndpointDescription]		
requestBody	[FromBody]	Accepts	
responses	[Produces]	Produces, ProducesProblem	TypedResults
Excluding endpoints	[ExcludeFromDescription], [ApiExplorerSettings]	ExcludeFromDescription	

ASP.NET Core does not collect metadata from XML doc comments.

The following sections demonstrate how to include metadata in an app to customize the generated OpenAPI document.

Summary and description

The endpoint summary and description can be set using the [EndpointSummary] and [EndpointDescription] attributes, or in minimal APIs, using the WithSummary and WithDescription extension methods.

Minimal APIs

The following sample demonstrates the different strategies for setting summaries and descriptions.

Note that the attributes are placed on the delegate method and not on the app.MapGet method.

C#

```
app.MapGet("/extension-methods", () => "Hello world!")
  .WithSummary("This is a summary.")
  .WithDescription("This is a description.");

app.MapGet("/attributes",
  [EndpointSummary("This is a summary.")]
  [EndpointDescription("This is a description.")]
  () => "Hello world!");
```

tags

OpenAPI supports specifying tags on each endpoint as a form of categorization.

Minimal APIs

In minimal APIs, tags can be set using either the [Tags] attribute or the WithTags extension method.

The following sample demonstrates the different strategies for setting tags.

C#

```
app.MapGet("/extension-methods", () => "Hello world!")
  .WithTags("todos", "projects");

app.MapGet("/attributes",
```

```
  [Tags("todos", "projects")]
  () => "Hello world!");
```

operationId

OpenAPI supports an operationId on each endpoint as a unique identifier or name for the operation.

Minimal APIs

In minimal APIs, the operationId can be set using either the [EndpointName] attribute or the WithName extension method.

The following sample demonstrates the different strategies for setting the operationId.

C#

```
app.MapGet("/extension-methods", () => "Hello world!")
  .WithName("FromExtensionMethods");

app.MapGet("/attributes",
  [EndpointName("FromAttributes")]
  () => "Hello world!");
```

parameters

OpenAPI supports annotating path, query string, header, and cookie parameters that are consumed by an API.

The framework infers the types for request parameters automatically based on the signature of the route handler.

The [EndpointDescription] attribute can be used to provide a description for a parameter.

Minimal APIs

The follow sample demonstrates how to set a description for a parameter.

C#

```
app.MapGet("/attributes",
  ([Description("This is a description.")] string name) => "Hello
world!");
```

Describe the request body

The `requestBody` field in OpenAPI describes the body of a request that an API client can send to the server, including the content type(s) supported and the schema for the body content.

When the endpoint handler method accepts parameters that are bound from the request body, ASP.NET Core generates a corresponding `requestBody` for the operation in the OpenAPI document. Metadata for the request body can also be specified using attributes or extension methods. Additional metadata can be set with a document transformer or operation transformer.

If the endpoint doesn't define any parameters bound to the request body, but instead consumes the request body from the HttpContext directly, ASP.NET Core provides mechanisms to specify request body metadata. This is a common scenario for endpoints that process the request body as a stream.

Some request body metadata can be determined from the FromBody or FromForm parameters of the route handler method.

A description for the request body can be set with a [Description] attribute on the parameter with FromBody or FromForm.

If the FromBody parameter is non-nullable and EmptyBodyBehavior is not set to Allow in the FromBody attribute, the request body is required and the `required` field of the `requestBody` is set to `true` in the generated OpenAPI document. Form bodies are always required and have `required` set to `true`.

Use a document transformer or an operation transformer to set the `example`, `examples`, or `encoding` fields, or to add specification extensions for the request body in the generated OpenAPI document.

Other mechanisms for setting request body metadata depend on the type of app being developed and are described in the following sections.

Minimal APIs

The content types for the request body in the generated OpenAPI document are determined from the type of the parameter that is bound to the request body or specified with the Accepts extension method. By default, the content type of a FromBody parameter will be `application/json` and the content type for FromForm parameter(s) will be `multipart/form-data` or `application/x-www-form-urlencoded`.

Support for these default content types is built in to Minimal APIs, and other content types can be handled by using custom binding. See the Custom binding topic of the Minimal APIs documentation for more information.

There are several ways to specify a different content type for the request body. If the type of the FromBody parameter implements IEndpointParameterMetadataProvider, ASP.NET Core uses this interface to determine the content type(s) in the request body. The framework uses the PopulateMetadata method of this interface to set the content type(s) and type of the body content of the request body. For example, a `Todo` class that accepts either `application/xml` or `text/xml` content-type can use IEndpointParameterMetadataProvider to provide this information to the framework.

C#

```
public class Todo : IEndpointParameterMetadataProvider
{
    public static void PopulateMetadata(ParameterInfo parameter,
EndpointBuilder builder)
    {
        builder.Metadata.Add(new AcceptsMetadata(["application/xml",
"text/xml"], typeof(Todo)));
    }
}
```

The Accepts extension method can also be used to specify the content type of the request body. In the following example, the endpoint accepts a `Todo` object in the request body with an expected content-type of `application/xml`.

C#

```
app.MapPut("/todos/{id}", (int id, Todo todo) => ...)
  .Accepts<Todo>("application/xml");
```

Since `application/xml` is not a built-in content type, the `Todo` class must implement the IBindableFromHttpContext<TSelf> interface to provide a custom binding for the request body. For example:

C#

```
public class Todo : IBindableFromHttpContext<Todo>
{
    public static async ValueTask<Todo?> BindAsync(HttpContext context,
ParameterInfo parameter)
    {
        var xmlDoc = await XDocument.LoadAsync(context.Request.Body,
LoadOptions.None, context.RequestAborted);
        var serializer = new XmlSerializer(typeof(Todo));
        return (Todo?)serializer.Deserialize(xmlDoc.CreateReader());
    }
```

If the endpoint doesn't define any parameters bound to the request body, use the Accepts extension method to specify the content type that the endpoint accepts.

If you specify <AspNetCore.Http.OpenApiRouteHandlerBuilderExtensions.Accepts%2A> multiple times, only metadata from the last one is used -- they aren't combined.

Describe response types

OpenAPI supports providing a description of the responses returned from an API. ASP.NET Core provides several strategies for setting the response metadata of an endpoint. Response metadata that can be set includes the status code, the type of the response body, and content type(s) of a response. Responses in OpenAPI may have additional metadata, such as description, headers, links, and examples. This additional metadata can be set with a document transformer or operation transformer.

The specific mechanisms for setting response metadata depend on the type of app being developed.

Minimal APIs

In Minimal API apps, ASP.NET Core can extract the response metadata added by extension methods on the endpoint, attributes on the route handler, and the return type of the route handler.

- The Produces extension method can be used on the endpoint to specify the status code, the type of the response body, and content type(s) of a response from an endpoint.
- The [ProducesResponseType] or ProducesResponseTypeAttribute<T> attribute can be used to specify the type of the response body.

- A route handler can be used to return a type that implements IEndpointMetadataProvider to specify the type and content-type(s) of the response body.
- The ProducesProblem extension method on the endpoint can be used to specify the status code and content-type(s) of an error response.

Note that the Produces and ProducesProblem extension methods are supported on both RouteHandlerBuilder and on RouteGroupBuilder. This allows, for example, a common set of error responses to be defined for all operations in a group.

When not specified by one of the preceding strategies, the:

- Status code for the response defaults to 200.
- Schema for the response body can be inferred from the implicit or explicit return type of the endpoint method, for example, from `T` in Task<TResult>; otherwise, it's considered to be unspecified.
- Content-type for the specified or inferred response body is "application/json".

In Minimal APIs, the Produces extension method and the [ProducesResponseType] attribute only set the response metadata for the endpoint. They do not modify or constrain the behavior of the endpoint, which may return a different status code or response body type than specified by the metadata, and the content-type is determined by the return type of the route handler method, irrespective of any content-type specified in attributes or extension methods.

The Produces extension method can specify an endpoint's response type, with a default status code of 200 and a default content type of `application/json`. The following example illustrates this:

C#

```
app.MapGet("/todos", async (TodoDb db) => await db.Todos.ToListAsync())
  .Produces<IList<Todo>>();
```

The [ProducesResponseType] can be used to add response metadata to an endpoint. Note that the attribute is applied to the route handler method, not the method invocation to create the route, as shown in the following example:

C#

```
app.MapGet("/todos",
    [ProducesResponseType<List<Todo>>(200)]
    async (TodoDb db) => await db.Todos.ToListAsync());
```

Using TypedResults in the implementation of an endpoint's route handler automatically includes the response type metadata for the endpoint. For example, the following code automatically annotates the endpoint with a response under the `200` status code with an `application/json` content type.

C#

```
app.MapGet("/todos", async (TodoDb db) =>
{
    var todos = await db.Todos.ToListAsync();
    return TypedResults.Ok(todos);
});
```

Only return types that implement IEndpointMetadataProvider create a `responses` entry in the OpenAPI document. The following is a partial list of some of the TypedResults helper methods that produce a `responses` entry:

Expand table

TypedResults helper method	status code
Ok()	200
Created()	201
CreatedAtRoute()	201
Accepted()	202
AcceptedAtRoute()	202
NoContent()	204
BadRequest()	400
ValidationProblem()	400
NotFound()	404
Conflict()	409
UnprocessableEntity()	422

All of these methods except `NoContent` have a generic overload that specifies the type of the response body.

A class can be implemented to set the endpoint metadata and return it from the route handler.

Set responses for `ProblemDetails`

When setting the response type for endpoints that may return a ProblemDetails response, the following can be used to add the appropriate response metadata for the endpoint:

- ProducesProblem
- ProducesValidationProblem extension method.
- TypedResults with a status code in the (400-499) range.

For more information on how to configure a Minimal API app to return ProblemDetails responses, see Handle errors in minimal APIs.

Multiple response types

If an endpoint can return different response types in different scenarios, you can provide metadata in the following ways:

- Call the Produces extension method multiple times, as shown in the following example:

```
C#
```

```
app.MapGet("/api/todoitems/{id}", async (int id, TodoDb db) =>
        await db.Todos.FindAsync(id)
        is Todo todo
        ? Results.Ok(todo)
        : Results.NotFound())
   .Produces<Todo>(StatusCodes.Status200OK)
   .Produces(StatusCodes.Status404NotFound);
```

- Use Results<TResult1,TResult2,TResult3,TResult4,TResult5,TResult6> in the signature and TypedResults in the body of the handler, as shown in the following example:

```
C#
```

```
app.MapGet("/book/{id}", Results<Ok<Book>, NotFound> (int id,
List<Book> bookList) =>
{
    return bookList.FirstOrDefault((i) => i.Id == id) is Book book
     ? TypedResults.Ok(book)
     : TypedResults.NotFound();
});
```

The `Results<TResult1,TResult2,TResultN>` union types declare that a route handler returns multiple `IResult`-implementing concrete types, and any of those types that implement IEndpointMetadataProvider will contribute to the endpoint's metadata.

The union types implement implicit cast operators. These operators enable the compiler to automatically convert the types specified in the generic arguments to an instance of the union type. This capability has the added benefit of providing compile-time checking that a route handler 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<TResult1,TResult2,TResultN>` results in a compilation error.

Exclude endpoints from the generated document

By default, all endpoints that are defined in an app are documented in the generated OpenAPI file, but endpoints can be excluded from the document using attributes or extension methods.

The mechanism for specifying an endpoint that should be excluded depends on the type of app being developed.

Minimal APIs

Minimal APIs support two strategies for excluding a given endpoint from the OpenAPI document:

- ExcludeFromDescription extension method
- [ExcludeFromDescription] attribute

The following sample demonstrates the different strategies for excluding a given endpoint from the generated OpenAPI document.

C#

```
app.MapGet("/extension-method", () => "Hello world!")
  .ExcludeFromDescription();

app.MapGet("/attributes",
  [ExcludeFromDescription]
  () => "Hello world!");
```

Include OpenAPI metadata for data types

C# classes or records used in request or response bodies are represented as schemas in the generated OpenAPI document. By default, only public properties are represented in the schema, but there are JsonSerializerOptions to also create schema properties for fields.

When the PropertyNamingPolicy is set to camel-case (this is the default in ASP.NET web applications), property names in a schema are the camel-case form of the class or record property name. The [JsonPropertyName] can be used on an individual property to specify the name of the property in the schema.

type and format

The JSON Schema library maps standard C# types to OpenAPI `type` and `format` as follows:

Expand table

C# Type	OpenAPI `type`	OpenAPI `format`
int	integer	int32
long	integer	int64
short	integer	int16
byte	integer	uint8
float	number	float
double	number	double
decimal	number	double
bool	boolean	
string	string	
char	string	char
byte[]	string	byte
DateTimeOffset	string	date-time
DateOnly	string	date
TimeOnly	string	time

C# Type	OpenAPI `type`	OpenAPI `format`
Uri	string	uri
Guid	string	uuid
object	*omitted*	
dynamic	*omitted*	

Note that object and dynamic types have *no* type defined in the OpenAPI because these can contain data of any type, including primitive types like int or string.

The `type` and `format` can also be set with a Schema Transformer. For example, you may want the `format` of decimal types to be `decimal` instead of `double`.

Use attributes to add metadata

ASP.NET uses metadata from attributes on class or record properties to set metadata on the corresponding properties of the generated schema.

The following table summarizes attributes from the `System.ComponentModel` namespace that provide metadata for the generated schema:

Expand table

Attribute	Description
[Description]	Sets the `description` of a property in the schema.
[Required]	Marks a property as `required` in the schema.
[DefaultValue]	Sets the `default` value of a property in the schema.
[Range]	Sets the `minimum` and `maximum` value of an integer or number.
[MinLength]	Sets the `minLength` of a string.
[MaxLength]	Sets the `maxLength` of a string.
[RegularExpression]	Sets the `pattern` of a string.

Note that in controller-based apps, these attributes add filters to the operation to validate that any incoming data satisfies the constraints. In Minimal APIs, these attributes set the metadata in the generated schema but validation must be performed explicitly via an endpoint filter, in the route handler's logic, or via a third-party package.

Attributes can also be placed on parameters in the parameter list of a record definition but must include the `property` modifier. For example:

C#

```
public record Todo(
    [property: Required]
    [property: Description("The unique identifier for the todo")]
    int Id,
    [property: Description("The title of the todo")]
    [property: MaxLength(120)]
    string Title,
    [property: Description("Whether the todo has been completed")]
    bool Completed
) {}
```

Other sources of metadata for generated schemas

required

In a class, struct, or record, properties with the [Required] attribute or required modifier are always `required` in the corresponding schema.

Other properties may also be required based on the constructors (implicit and explicit) for the class, struct, or record.

- For a class or record class with a single public constructor, any property with the same type and name (case-insensitive match) as a parameter to the constructor is required in the corresponding schema.
- For a class or record class with multiple public constructors, no other properties are required.
- For a struct or record struct, no other properties are required since C# always defines an implicit parameterless constructor for a struct.

enum

Enum types in C# are integer-based, but can be represented as strings in JSON with a [JsonConverter] and a JsonStringEnumConverter. When an enum type is represented as a string in JSON, the generated schema will have an `enum` property with the string values of the enum.

The following example demonstrates how to use a `JsonStringEnumConverter` to represent an enum as a string in JSON:

C#

```
[JsonConverter(typeof(JsonStringEnumConverter<DayOfTheWeekAsString>))]
public enum DayOfTheWeekAsString
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}
```

A special case is when an enum type has the [Flags] attribute, which indicates that the enum can be treated as a bit field; that is, a set of flags. A flags enum with a [JsonConverterAttribute] will be defined as `type: string` in the generated schema with no `enum` property, since the value could be any combination of the enum values. For example, the following enum:

C#

```
[Flags, JsonConverter(typeof(JsonStringEnumConverter<PizzaToppings>))]
public enum PizzaToppings { Pepperoni = 1, Sausage = 2, Mushrooms = 4,
Anchovies = 8 }
```

could have values such as `"Pepperoni, Sausage"` or `"Sausage, Mushrooms, Anchovies"`.

An enum type without a [JsonConverter] will be defined as `type: integer` in the generated schema.

Note: The [AllowedValues] attribute does not set the `enum` values of a property.

nullable

Properties defined as a nullable value or reference type have `nullable: true` in the generated schema. This is consistent with the default behavior of the System.Text.Json deserializer, which accepts `null` as a valid value for a nullable property.

additionalProperties

Schemas are generated without an `additionalProperties` assertion by default, which implies the default of `true`. This is consistent with the default behavior of the

System.Text.Json deserializer, which silently ignores additional properties in a JSON object.

If the additional properties of a schema should only have values of a specific type, define the property or class as a `Dictionary<string, type>`. The key type for the dictionary must be `string`. This generates a schema with `additionalProperties` specifying the schema for "type" as the required value types.

Polymorphic types

Use the [JsonPolymorphic] and [JsonDerivedType] attributes on a parent class to to specify the discriminator field and subtypes for a polymorphic type.

The [JsonDerivedType] adds the discriminator field to the schema for each subclass, with an enum specifying the specific discriminator value for the subclass. This attribute also modifies the constructor of each derived class to set the discriminator value.

An abstract class with a [JsonPolymorphic] attribute has a `discriminator` field in the schema, but a concrete class with a [JsonPolymorphic] attribute doesn't have a `discriminator` field. OpenAPI requires that the discriminator property be a required property in the schema, but since the discriminator property isn't defined in the concrete base class, the schema cannot include a `discriminator` field.

Add metadata with a schema transformer

A schema transformer can be used to override any default metadata or add additional metadata, such as `example` values, to the generated schema. See Use schema transformers for more information.

Additional resources

- Use the generated OpenAPI documents
- OpenAPI specification

Customize OpenAPI documents

Article • 11/04/2024

OpenAPI document transformers

This section demonstrates how to customize OpenAPI documents with transformers.

Customize OpenAPI documents with transformers

Transformers provide an API for modifying the OpenAPI document with user-defined customizations. Transformers are useful for scenarios like:

- Adding parameters to all operations in a document.
- Modifying descriptions for parameters or operations.
- Adding top-level information to the OpenAPI document.

Transformers fall into three categories:

- Document transformers have access to the entire OpenAPI document. These can be used to make global modifications to the document.
- Operation transformers apply to each individual operation. Each individual operation is a combination of path and HTTP method. These can be used to modify parameters or responses on endpoints.
- Schema transformers apply to each schema in the document. These can be used to modify the schema of request or response bodies, or any nested schemas.

Transformers can be registered onto the document by calling the AddDocumentTransformer method on the OpenApiOptions object. The following snippet shows different ways to register transformers onto the document:

- Register a document transformer using a delegate.
- Register a document transformer using an instance of IOpenApiDocumentTransformer.
- Register a document transformer using a DI-activated IOpenApiDocumentTransformer.
- Register an operation transformer using a delegate.
- Register an operation transformer using an instance of IOpenApiOperationTransformer.
- Register an operation transformer using a DI-activated IOpenApiOperationTransformer.
- Register a schema transformer using a delegate.

- Register a schema transformer using an instance of IOpenApiSchemaTransformer.
- Register a schema transformer using a DI-activated IOpenApiSchemaTransformer.

C#

```
using Microsoft.AspNetCore.OpenApi;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder();

builder.Services.AddOpenApi(options =>
{
    options.AddDocumentTransformer((document, context, cancellationToken)
                            => Task.CompletedTask);
    options.AddDocumentTransformer(new MyDocumentTransformer());
    options.AddDocumentTransformer<MyDocumentTransformer>();
    options.AddOperationTransformer((operation, context, cancellationToken)
                           => Task.CompletedTask);
    options.AddOperationTransformer(new MyOperationTransformer());
    options.AddOperationTransformer<MyOperationTransformer>();
    options.AddSchemaTransformer((schema, context, cancellationToken)
                           => Task.CompletedTask);
    options.AddSchemaTransformer(new MySchemaTransformer());
    options.AddSchemaTransformer<MySchemaTransformer>();
});

var app = builder.Build();

app.MapOpenApi();

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

app.Run();
```

Execution order for transformers

Transformers execute in first-in first-out order based on registration. In the following snippet, the document transformer has access to the modifications made by the operation transformer:

C#

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

```
builder.Services.AddOpenApi(options =>
{
    options.AddOperationTransformer((operation, context, cancellationToken)
                                     => Task.CompletedTask);
    options.AddDocumentTransformer((document, context, cancellationToken)
                                     => Task.CompletedTask);
});

var app = builder.Build();

app.MapOpenApi();

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

app.Run();
```

Use document transformers

Document transformers have access to a context object that includes:

- The name of the document being modified.
- The ApiDescriptionGroups associated with that document.
- The IServiceProvider used in document generation.

Document transformers can also mutate the OpenAPI document that is generated. The following example demonstrates a document transformer that adds some information about the API to the OpenAPI document.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Builder;

var builder = WebApplication.CreateBuilder();

builder.Services.AddOpenApi(options =>
{
    options.AddDocumentTransformer((document, context, cancellationToken) =>
    {
        document.Info = new()
        {
            Title = "Checkout API",
            Version = "v1",
            Description = "API for processing checkouts from cart."
        };
        return Task.CompletedTask;
    });
});

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

```
app.MapOpenApi();

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

app.Run();
```

Service-activated document transformers can utilize instances from DI to modify the app. The following sample demonstrates a document transformer that uses the IAuthenticationSchemeProvider service from the authentication layer. It checks if any JWT bearer-related schemes are registered in the app and adds them to the OpenAPI document's top level:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.OpenApi;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAuthentication().AddJwtBearer();

builder.Services.AddOpenApi(options =>
{
    options.AddDocumentTransformer<BearerSecuritySchemeTransformer>();
});

var app = builder.Build();

app.MapOpenApi();

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

app.Run();

internal sealed class
BearerSecuritySchemeTransformer(IAuthenticationSchemeProvider
authenticationSchemeProvider) : IOpenApiDocumentTransformer
{
    public async Task TransformAsync(OpenApiDocument document,
OpenApiDocumentTransformerContext context, CancellationToken
cancellationToken)
    {
        var authenticationSchemes = await
authenticationSchemeProvider.GetAllSchemesAsync();
        if (authenticationSchemes.Any(authScheme => authScheme.Name ==
"Bearer"))
        {
            var requirements = new Dictionary<string, OpenApiSecurityScheme>
```

```
            {
                ["Bearer"] = new OpenApiSecurityScheme
                {
                    Type = SecuritySchemeType.Http,
                    Scheme = "bearer", // "bearer" refers to the header name
here
                    In = ParameterLocation.Header,
                    BearerFormat = "Json Web Token"
                }
            };
            document.Components ??= new OpenApiComponents();
            document.Components.SecuritySchemes = requirements;
        }
    }
}
```

Document transformers are unique to the document instance they're associated with. In the following example, a transformer:

- Registers authentication-related requirements to the `internal` document.
- Leaves the `public` document unmodified.

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.OpenApi;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAuthentication().AddJwtBearer();

builder.Services.AddOpenApi("internal", options =>
{
    options.AddDocumentTransformer<BearerSecuritySchemeTransformer>();
});
builder.Services.AddOpenApi("public");

var app = builder.Build();

app.MapOpenApi();

app.MapGet("/world", () => "Hello world!")
    .WithGroupName("internal");
app.MapGet("/", () => "Hello universe!")
    .WithGroupName("public");

app.Run();

internal sealed class
```

```
BearerSecuritySchemeTransformer(IAuthenticationSchemeProvider
authenticationSchemeProvider) : IOpenApiDocumentTransformer
{
    public async Task TransformAsync(OpenApiDocument document,
OpenApiDocumentTransformerContext context, CancellationToken
cancellationToken)
    {
        var authenticationSchemes = await
authenticationSchemeProvider.GetAllSchemesAsync();
        if (authenticationSchemes.Any(authScheme => authScheme.Name ==
"Bearer"))
        {
            // Add the security scheme at the document level
            var requirements = new Dictionary<string, OpenApiSecurityScheme>
            {
                ["Bearer"] = new OpenApiSecurityScheme
                {
                    Type = SecuritySchemeType.Http,
                    Scheme = "bearer", // "bearer" refers to the header name
here
                    In = ParameterLocation.Header,
                    BearerFormat = "Json Web Token"
                }
            };
            document.Components ??= new OpenApiComponents();
            document.Components.SecuritySchemes = requirements;

            // Apply it as a requirement for all operations
            foreach (var operation in document.Paths.Values.SelectMany(path
=> path.Operations))
            {
                operation.Value.Security.Add(new OpenApiSecurityRequirement
                {
                    [new OpenApiSecurityScheme { Reference = new
OpenApiReference { Id = "Bearer", Type = ReferenceType.SecurityScheme } }] =
Array.Empty<string>()
                });
            }
        }
    }
}
```

Use operation transformers

Operations are unique combinations of HTTP paths and methods in an OpenAPI document. Operation transformers are helpful when a modification:

- Should be made to each endpoint in an app, or
- Conditionally applied to certain routes.

Operation transformers have access to a context object which contains:

- The name of the document the operation belongs to.
- The ApiDescription associated with the operation.
- The IServiceProvider used in document generation.

For example, the following operation transformer adds `500` as a response status code supported by all operations in the document.

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.OpenApi;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAuthentication().AddJwtBearer();

builder.Services.AddOpenApi(options =>
{
    options.AddOperationTransformer((operation, context, cancellationToken)
=>
    {
        operation.Responses.Add("500", new OpenApiResponse { Description =
"Internal server error" });
        return Task.CompletedTask;
    });
});

var app = builder.Build();

app.MapOpenApi();

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

app.Run();
```

Use schema transformers

Schemas are the data models that are used in request and response bodies in an OpenAPI document. Schema transformers are useful when a modification:

- Should be made to each schema in the document, or
- Conditionally applied to certain schemas.

Schema transformers have access to a context object which contains:

- The name of the document the schema belongs to.

- The JSON type information associated with the target schema.
- The IServiceProvider used in document generation.

For example, the following schema transformer sets the `format` of decimal types to `decimal` instead of `double`:

C#

```
using Microsoft.AspNetCore.OpenApi;

var builder = WebApplication.CreateBuilder();

builder.Services.AddOpenApi(options => {
    // Schema transformer to set the format of decimal to 'decimal'
    options.AddSchemaTransformer((schema, context, cancellationToken) =>
    {
        if (context.JsonTypeInfo.Type == typeof(decimal))
        {
            schema.Format = "decimal";
        }
        return Task.CompletedTask;
    });
});

var app = builder.Build();

app.MapOpenApi();

app.MapGet("/", () => new Body { Amount = 1.1m });

app.Run();

public class Body {
    public decimal Amount { get; set; }
}
```

Additional resources

- Use the generated OpenAPI documents
- OpenAPI specification

Use openAPI documents

Article • 12/11/2024

Use Swagger UI for local ad-hoc testing

By default, the `Microsoft.AspNetCore.OpenApi` package doesn't ship with built-in support for visualizing or interacting with the OpenAPI document. Popular tools for visualizing or interacting with the OpenAPI document include Swagger UI and ReDoc. Swagger UI and ReDoc can be integrated in an app in several ways. Editors such as Visual Studio and VS Code offer extensions and built-in experiences for testing against an OpenAPI document.

The `Swashbuckle.AspNetCore.SwaggerUi` package provides a bundle of Swagger UI's web assets for use in apps. This package can be used to render a UI for the generated document. To configure this:

- Install the `Swashbuckle.AspNetCore.SwaggerUi` package.
- Enable the swagger-ui middleware with a reference to the OpenAPI route registered earlier.
- To limit information disclosure and security vulnerability, ***only enable Swagger UI in development environments.***

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.OpenApi;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder();

builder.Services.AddOpenApi();

var app = builder.Build();

app.MapOpenApi();
if (app.Environment.IsDevelopment())
{
    app.UseSwaggerUI(options =>
    {
        options.SwaggerEndpoint("/openapi/v1.json", "v1");
    });

}
```

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

app.Run();
```

Use Scalar for interactive API documentation

Scalar is an open-source interactive document UI for OpenAPI. Scalar can integrate with the OpenAPI endpoint provided by ASP.NET Core. To configure Scalar, install the `Scalar.AspNetCore` package.

C#

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.OpenApi;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.OpenApi.Models;
using Scalar.AspNetCore;

var builder = WebApplication.CreateBuilder();

builder.Services.AddOpenApi();

var app = builder.Build();

app.MapOpenApi();

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

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

app.Run();
```

Launch the app and navigate to `https://localhost:<port>/scalar/v1` to view the Scalar UI.

Lint generated OpenAPI documents with Spectral

Spectral is an open-source OpenAPI document linter. Spectral can be incorporated into an app build to verify the quality of generated OpenAPI documents. Install Spectral according to the package installation directions.

To take advantage of Spectral, install the `Microsoft.Extensions.ApiDescription.Server` package to enable build-time OpenAPI document generation.

Enable document generation at build time by setting the following properties in the app's `.csproj` file":

XML

```xml
<PropertyGroup>
    <OpenApiDocumentsDirectory>$(MSBuildProjectDirectory)
</OpenApiDocumentsDirectory>
    <OpenApiGenerateDocuments>true</OpenApiGenerateDocuments>
</PropertyGroup>
```

Run `dotnet build` to generate the document.

.NET CLI

```
dotnet build
```

Create a `.spectral.yml` file with the following contents.

text

```
extends: ["spectral:oas"]
```

Run `spectral lint` on the generated file.

.NET CLI

```
spectral lint WebMinOpenApi.json
...

The output shows any issues with the OpenAPI document. For example:

```output
1:1  warning  oas3-api-servers       OpenAPI "servers" must be present and
non-empty array.
3:10  warning  info-contact           Info object must have "contact"
object.                         info
3:10  warning  info-description       Info "description" must be present and
non-empty string.       info
9:13  warning  operation-description  Operation "description" must be
present and non-empty string.  paths./.get
9:13  warning  operation-operationId  Operation must have "operationId".
paths./.get
```
```

```
✕ 5 problems (0 errors, 5 warnings, 0 infos, 0 hints)
```

.NET OpenAPI tool command reference and installation

Article • 08/06/2024

Microsoft.dotnet-openapi is a .NET Core Global Tool for managing OpenAPI references within a project.

Installation

To install `Microsoft.dotnet-openapi`, run the following command:

.NET CLI

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

> **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.

Add

Adding an OpenAPI reference using any of the commands on this page adds an `<OpenApiReference />` element similar to the following to the `.csproj` file:

XML

```
<OpenApiReference Include="openapi.json" />
```

The preceding reference is required for the app to call the generated client code.

Add File

Options

Expand table

Short option	Long option	Description	Example
-p	--updateProject	The project to operate on.	dotnet openapi add file *--updateProject .\Ref.csproj* .\OpenAPI.json
-c	--code-generator	The code generator to apply to the reference. Options are `NSwagCSharp` and `NSwagTypeScript`. If `--code-generator` is not specified the tooling defaults to `NSwagCSharp`.	dotnet openapi add file .\OpenApi.json --code-generator
-h	--help	Show help information	dotnet openapi add file --help

Arguments

Expand table

Argument	Description	Example
source-file	The source to create a reference from. Must be an OpenAPI file.	dotnet openapi add file *.\OpenAPI.json*

Add URL

Options

Expand table

Short option	Long option	Description	Example
-p	--updateProject	The project to operate on.	dotnet openapi add url *--updateProject .\Ref.csproj* `https://contoso.com/openapi.json`
-o	--output-file	Where to place the local copy of the OpenAPI file.	dotnet openapi add url `https://contoso.com/openapi.json` *--output-file myclient.json*

Short option	Long option	Description	Example
-c	--code-generator	The code generator to apply to the reference. Options are `NSwagCSharp` and `NSwagTypeScript`.	dotnet openapi add url `https://contoso.com/openapi.json` --code-generator
-h	--help	Show help information	dotnet openapi add url --help

Arguments

Expand table

Argument	Description	Example
source-URL	The source to create a reference from. Must be a URL.	dotnet openapi add url `https://contoso.com/openapi.json`

Remove

Removes the OpenAPI reference matching the given filename from the `.csproj` file. When the OpenAPI reference is removed, clients won't be generated. Local `.json` and `.yaml` files are deleted.

Options

Expand table

Short option	Long option	Description	Example
-p	--updateProject	The project to operate on.	dotnet openapi remove *--updateProject .\Ref.csproj* .\OpenAPI.json
-h	--help	Show help information	dotnet openapi remove --help

Arguments

Expand table

Argument	Description	Example
source-file	The source to remove the reference to.	dotnet openapi remove .*OpenAPI.json*

Refresh

Refreshes the local version of a file that was downloaded using the latest content from the download URL.

Options

Expand table

Short option	Long option	Description	Example
-p	--updateProject	The project to operate on.	dotnet openapi refresh *--updateProject .\Ref.csproj* `https://contoso.com/openapi.json`
-h	--help	Show help information	dotnet openapi refresh --help

Arguments

Expand table

Argument	Description	Example
source-URL	The URL to refresh the reference from.	dotnet openapi refresh `https://contoso.com/openapi.json`

Overview of ASP.NET Core SignalR

Article • 12/02/2024

What is SignalR?

ASP.NET Core SignalR is an open-source library that simplifies adding real-time web functionality to apps. Real-time web functionality enables server-side code to push content to clients instantly.

Good candidates for SignalR:

- Apps that require high frequency updates from the server. Examples are gaming, social networks, voting, auction, maps, and GPS apps.
- Dashboards and monitoring apps. Examples include company dashboards, instant sales updates, or travel alerts.
- Collaborative apps. Whiteboard apps and team meeting software are examples of collaborative apps.
- Apps that require notifications. Social networks, email, chat, games, travel alerts, and many other apps use notifications.

SignalR provides an API for creating server-to-client remote procedure calls (RPC). The RPCs invoke functions on clients from server-side .NET Core code. There are several supported platforms, each with their respective client SDK. Because of this, the programming language being invoked by the RPC call varies.

Here are some features of SignalR for ASP.NET Core:

- Handles connection management automatically.
- Sends messages to all connected clients simultaneously. For example, a chat room.
- Sends messages to specific clients or groups of clients.
- Scales to handle increasing traffic.
- SignalR Hub Protocol

The source is hosted in a SignalR repository on GitHub.

Transports

SignalR supports the following techniques for handling real-time communication (in order of graceful fallback):

- WebSockets

- Server-Sent Events
- Long Polling

SignalR automatically chooses the best transport method that is within the capabilities of the server and client.

Hubs

SignalR uses *hubs* to communicate between clients and servers.

A hub is a high-level pipeline that allows a client and server to call methods on each other. SignalR handles the dispatching across machine boundaries automatically, allowing clients to call methods on the server and vice versa. You can pass strongly-typed parameters to methods, which enables model binding. SignalR provides two built-in hub protocols: a text protocol based on JSON and a binary protocol based on MessagePack. MessagePack generally creates smaller messages compared to JSON. Older browsers must support XHR level 2 to provide MessagePack protocol support.

Hubs call client-side code by sending messages that contain the name and parameters of the client-side method. Objects sent as method parameters are deserialized using the configured protocol. The client tries to match the name to a method in the client-side code. When the client finds a match, it calls the method and passes to it the deserialized parameter data.

Browsers that don't support ECMAScript 6 (ES6)

SignalR targets ES6. For browsers that don't support ES6, transpile the library to ES5. For more information, see Getting Started with ES6 – Transpiling ES6 to ES5 with Traceur and Babel.

Additional resources

- Introduction to ASP.NET Core SignalR
- Get started with SignalR for ASP.NET Core
- Supported Platforms
- Hubs
- JavaScript client
- Browsers that don't support ECMAScript 6 (ES6)
- ASP.NET Core Blazor SignalR guidance

ASP.NET Core SignalR supported platforms

Article • 06/18/2024

Server system requirements

SignalR for ASP.NET Core supports any server platform that ASP.NET Core supports.

JavaScript client

The JavaScript client runs on the current Node.js long-term support (LTS) release and the following browsers:

Expand table

Browser	Version
Apple Safari, including iOS	Current†
Google Chrome, including Android	Current†
Microsoft Edge	Current†
Mozilla Firefox	Current†

†*Current* refers to the latest version of the browser.

The JavaScript client doesn't support Internet Explorer and other older browsers. The client might have unexpected behavior and errors on unsupported browsers.

.NET client

The .NET client runs on any platform supported by ASP.NET Core. For example, Xamarin developers can use SignalR for building Android apps using Xamarin.Android 8.4.0.1 and later and iOS apps using Xamarin.iOS 11.14.0.4 and later.

If the server runs IIS, the WebSockets transport requires IIS 8.0 or later on Windows Server 2012 or later. Other transports are supported on all platforms.

Java client

The Java client supports Java 8 and later versions.

Unsupported clients

The following clients are available but are experimental or unofficial. The following clients aren't currently supported and may never be supported:

- C++ client
- Swift client

Browsers that don't support ECMAScript 6 (ES6)

SignalR targets ES6. For browsers that don't support ES6, transpile the library to ES5. For more information, see Getting Started with ES6 – Transpiling ES6 to ES5 with Traceur and Babel.

Tutorial: Get started with ASP.NET Core SignalR

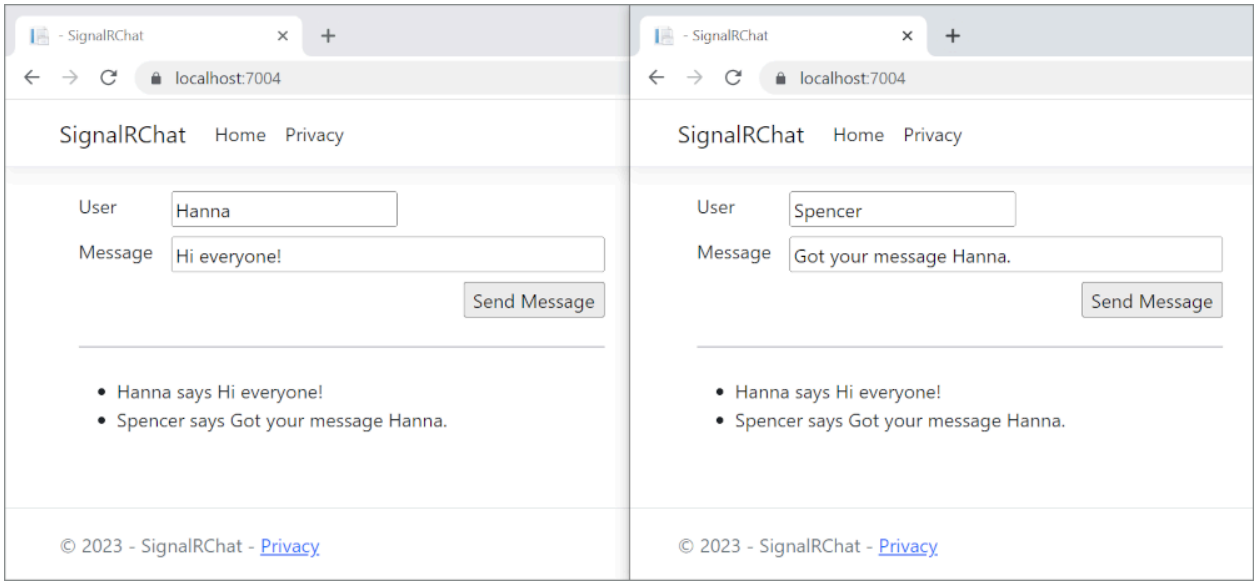
Article • 11/16/2023

> **ⓘ 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 tutorial teaches the basics of building a real-time app using SignalR. You learn how to:

- ✓ Create a web project.
- ✓ Add the SignalR client library.
- ✓ Create a SignalR hub.
- ✓ Configure the project to use SignalR.
- ✓ Add code that sends messages from any client to all connected clients.

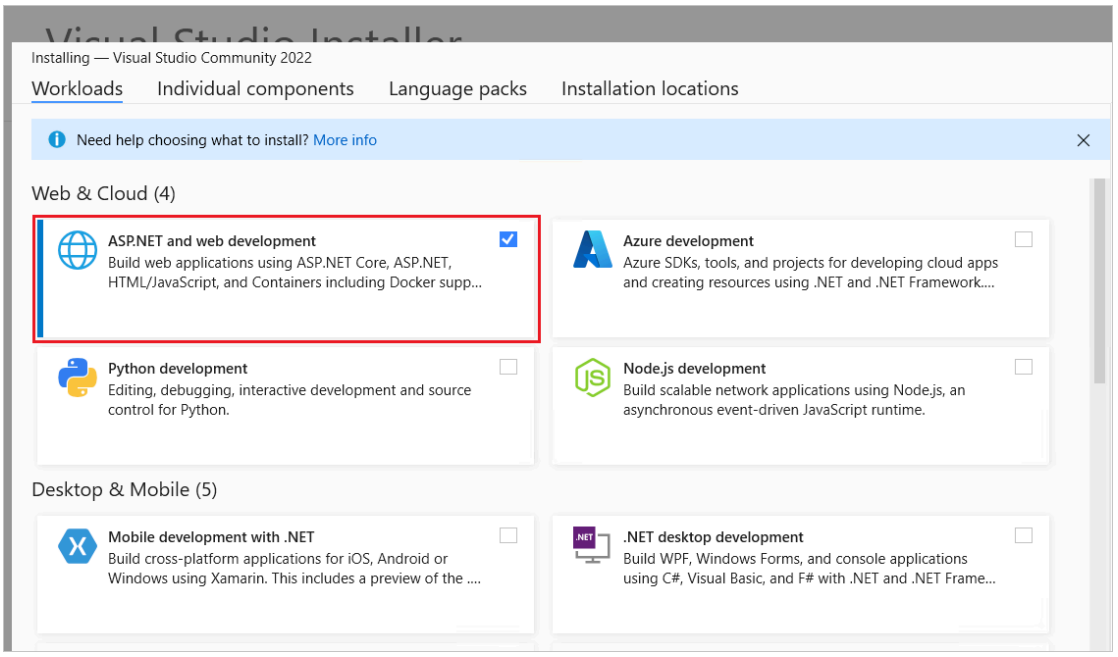
At the end, you'll have a working chat app:



Prerequisites

Visual Studio

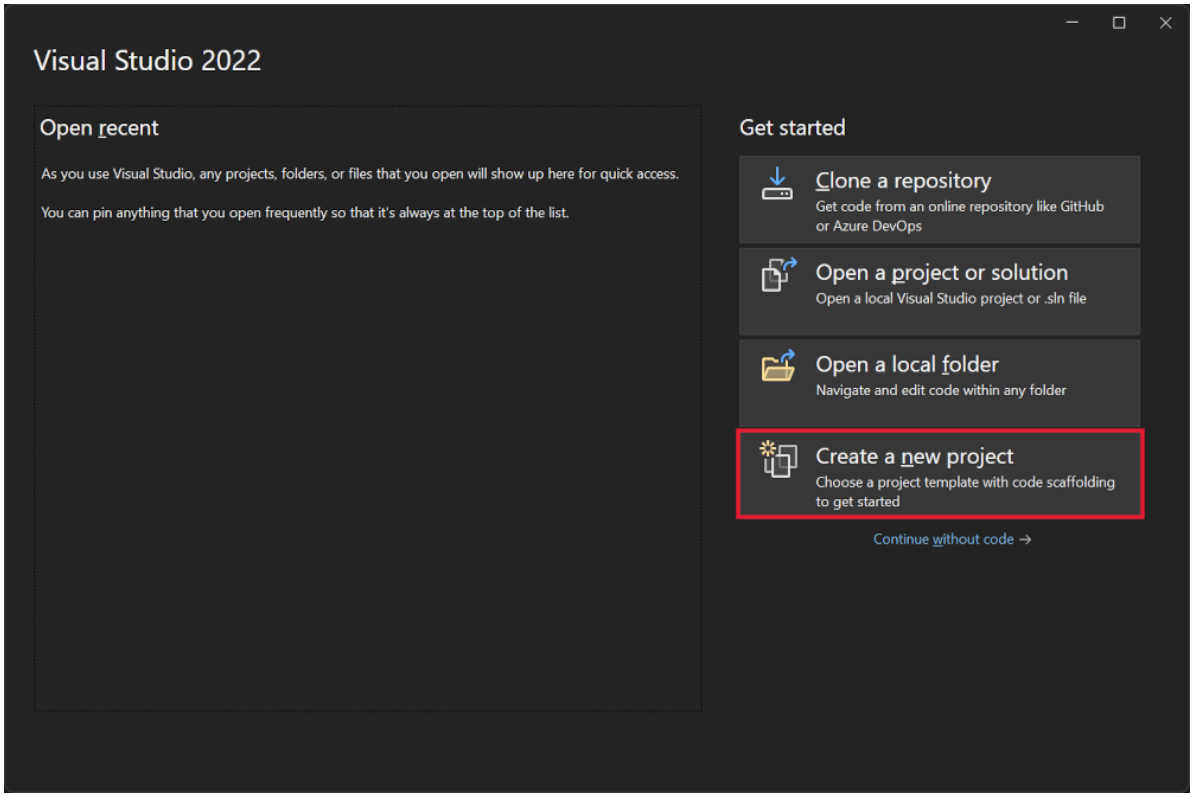
- Visual Studio 2022 with the **ASP.NET and web development** workload.



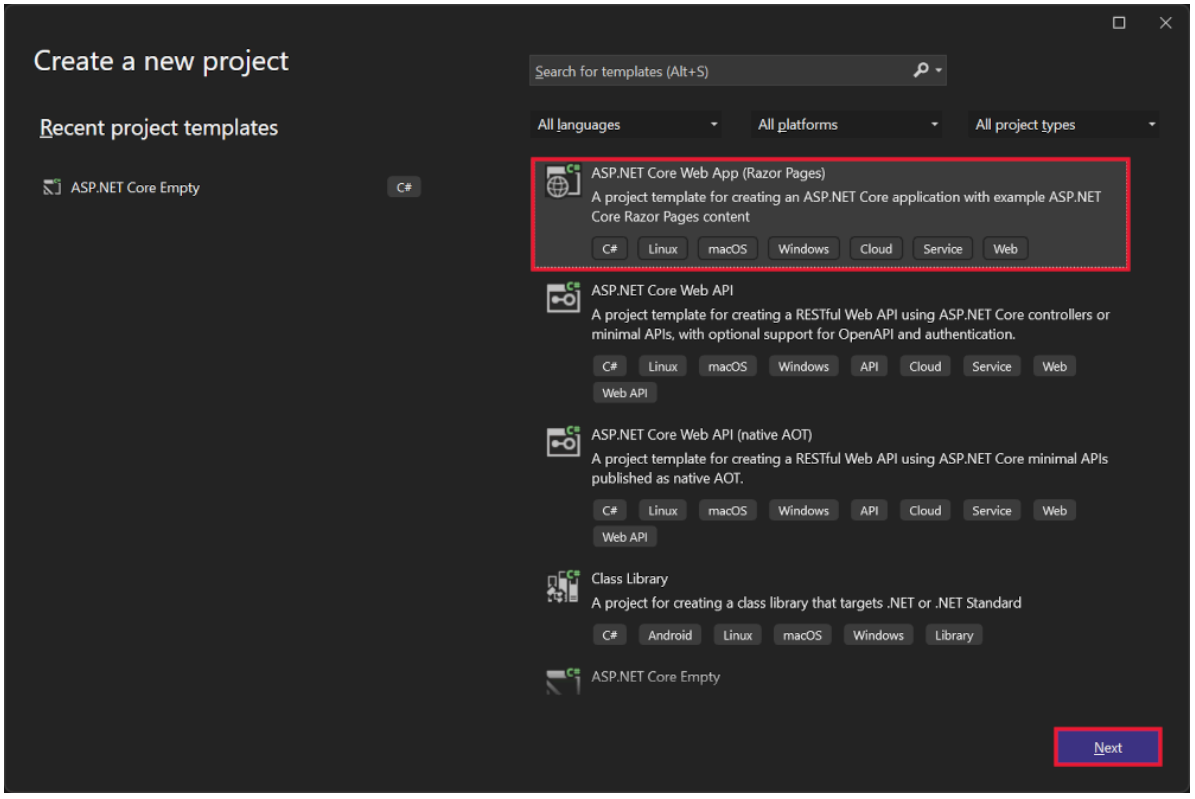
Create a web app project

Visual Studio

Start Visual Studio 2022 and select **Create a new project**.



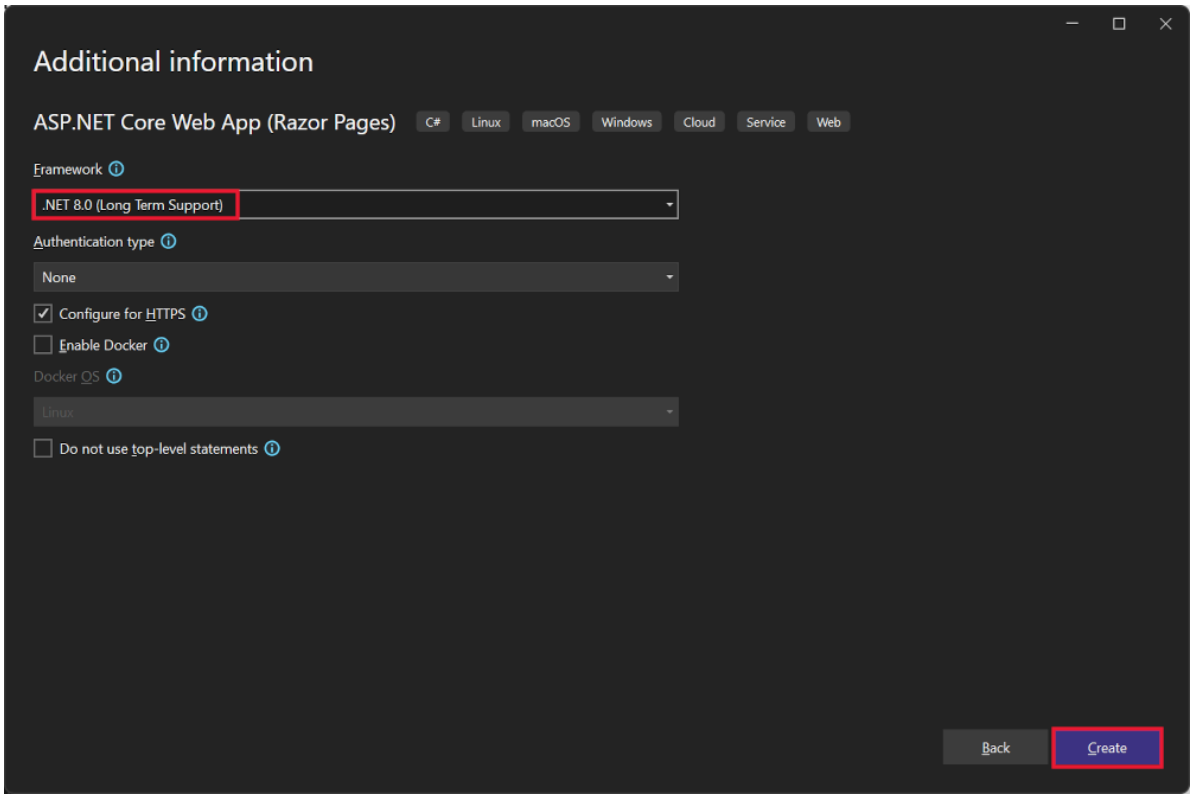
In the **Create a new project** dialog, select **ASP.NET Core Web App (Razor Pages)**, and then select **Next**.



In the **Configure your new project** dialog, enter `SignalRChat` for **Project name**. It's important to name the project `SignalRChat`, including matching the capitalization, so the namespaces match the code in the tutorial.

Select **Next**.

In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.



Add the SignalR client library

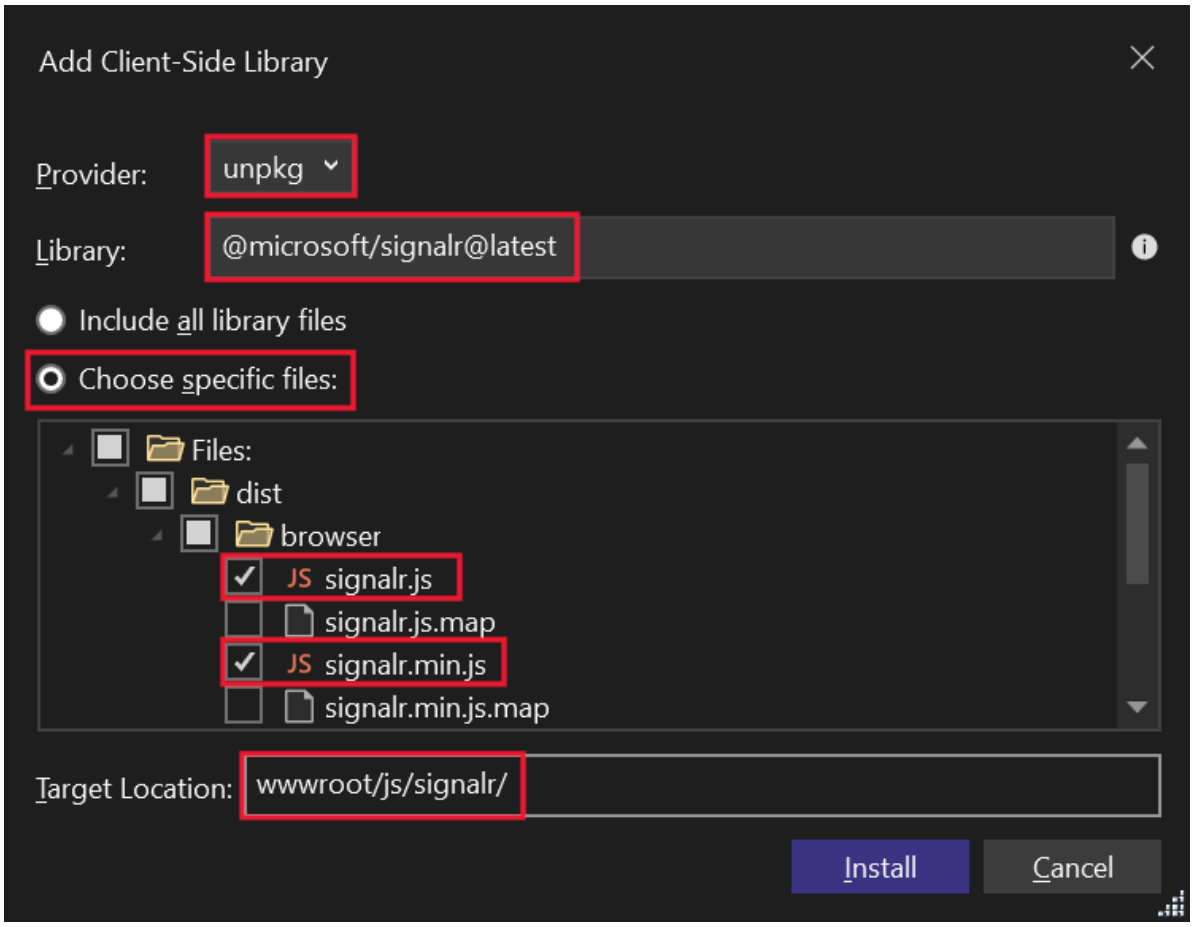
The SignalR server library is included in the ASP.NET Core shared framework. The JavaScript client library isn't automatically included in the project. For this tutorial, use Library Manager (LibMan) to get the client library from unpkg. `unpkg` is a fast, global content delivery network for everything on npm.

Visual Studio

In **Solution Explorer**, right-click the project, and select **Add** > **Client-Side Library**.

In the **Add Client-Side Library** dialog:

- Select **unpkg** for **Provider**
- Enter `@microsoft/signalr@latest` for **Library**.
- Select **Choose specific files**, expand the *dist/browser* folder, and select `signalr.js` and `signalr.min.js`.
- Set **Target Location** to `wwwroot/js/signalr/`.
- Select **Install**.



LibMan creates a `wwwroot/js/signalr` folder and copies the selected files to it.

Create a SignalR hub

A *hub* is a class that serves as a high-level pipeline that handles client-server communication.

In the SignalRChat project folder, create a `Hubs` folder.

In the `Hubs` folder, create the `ChatHub` class with the following code:

C#

```
using Microsoft.AspNetCore.SignalR;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
```

```
    }
}
```

The `ChatHub` class inherits from the SignalR Hub class. The `Hub` class manages connections, groups, and messaging.

The `SendMessage` method can be called by a connected client to send a message to all clients. JavaScript client code that calls the method is shown later in the tutorial. SignalR code is asynchronous to provide maximum scalability.

Configure SignalR

The SignalR server must be configured to pass SignalR requests to SignalR. Add the following highlighted code to the `Program.cs` file.

C#

```
using SignalRChat.Hubs;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

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

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();
app.MapHub<ChatHub>("/chatHub");

app.Run();
```

The preceding highlighted code adds SignalR to the ASP.NET Core dependency injection and routing systems.

Add SignalR client code

Replace the content in `Pages/Index.cshtml` with the following code:

CSHTML

```
@page
<div class="container">
    <div class="row p-1">
        <div class="col-1">User</div>
        <div class="col-5"><input type="text" id="userInput" /></div>
    </div>
    <div class="row p-1">
        <div class="col-1">Message</div>
        <div class="col-5"><input type="text" class="w-100"
id="messageInput" /></div>
    </div>
    <div class="row p-1">
        <div class="col-6 text-end">
            <input type="button" id="sendButton" value="Send Message" />
        </div>
    </div>
    <div class="row p-1">
        <div class="col-6">
            <hr />
        </div>
    </div>
    <div class="row p-1">
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
</div>
<script src="~/js/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

The preceding markup:

- Creates text boxes and a submit button.
- Creates a list with `id="messagesList"` for displaying messages that are received from the SignalR hub.
- Includes script references to SignalR and the `chat.js` app code is created in the next step.

In the `wwwroot/js` folder, create a `chat.js` file with the following code:

JavaScript

```
"use strict";

var connection = new
signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable the send button until connection is established.
document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var li = document.createElement("li");
    document.getElementById("messagesList").appendChild(li);
    // We can assign user-supplied strings to an element's textContent
because it
    // is not interpreted as markup. If you're assigning in any other way,
you
    // should be aware of possible script injection concerns.
    li.textContent = `${user} says ${message}`;
});

connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function
(event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

The preceding JavaScript:

- Creates and starts a connection.
- Adds to the submit button a handler that sends messages to the hub.
- Adds to the connection object a handler that receives messages from the hub and adds them to the list.

Run the app

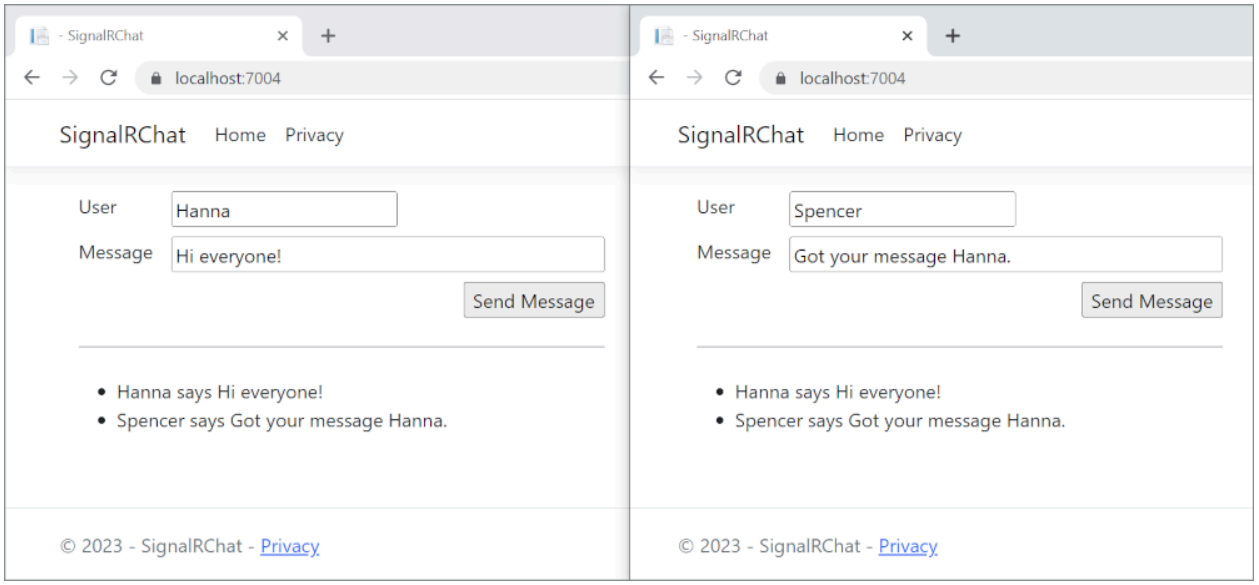
Visual Studio

Select `Ctrl`+`F5` to run the app without debugging.

Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

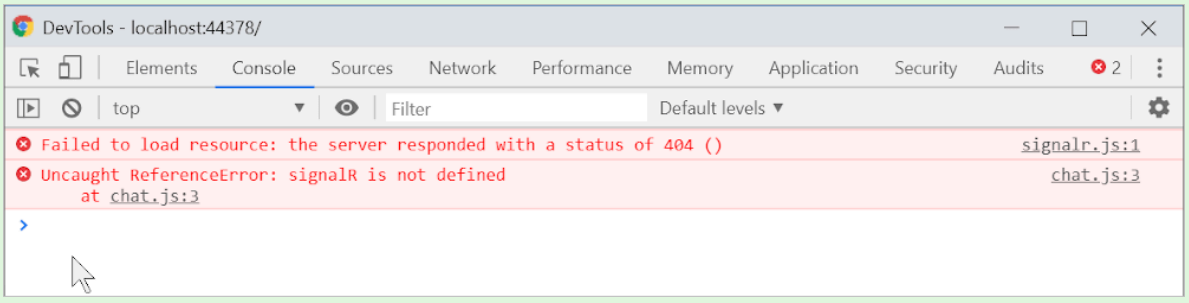
Choose either browser, enter a name and message, and select the **Send Message** button.

The name and message are displayed on both pages instantly.



Tip

If the app doesn't work, open the browser developer tools (F12) and go to the console. Look for possible errors related to HTML and JavaScript code. For example, if `signalr.js` was put in a different folder than directed, the reference to that file won't work resulting in a 404 error in the console.



If an `ERR_SPDY_INADEQUATE_TRANSPORT_SECURITY` error has occurred in Chrome, run the following commands to update the development certificate:

.NET CLI

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Publish to Azure

For information on deploying to Azure, see Quickstart: Deploy an ASP.NET web app. For more information on Azure SignalR Service, see What is Azure SignalR Service?.

Next steps

- Use hubs
- Strongly typed hubs
- Authentication and authorization in ASP.NET Core SignalR
- View or download sample code (how to download)

Tutorial: Get started with ASP.NET Core SignalR using TypeScript and Webpack

Article • 11/16/2023

> **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 Sébastien Sougnez

This tutorial demonstrates using Webpack in an ASP.NET Core SignalR web app to bundle and build a client written in TypeScript. Webpack enables developers to bundle and build the client-side resources of a web app.

In this tutorial, you learn how to:

- ✓ Create an ASP.NET Core SignalR app
- ✓ Configure the SignalR server
- ✓ Configure a build pipeline using Webpack
- ✓ Configure the SignalR TypeScript client
- ✓ Enable communication between the client and the server

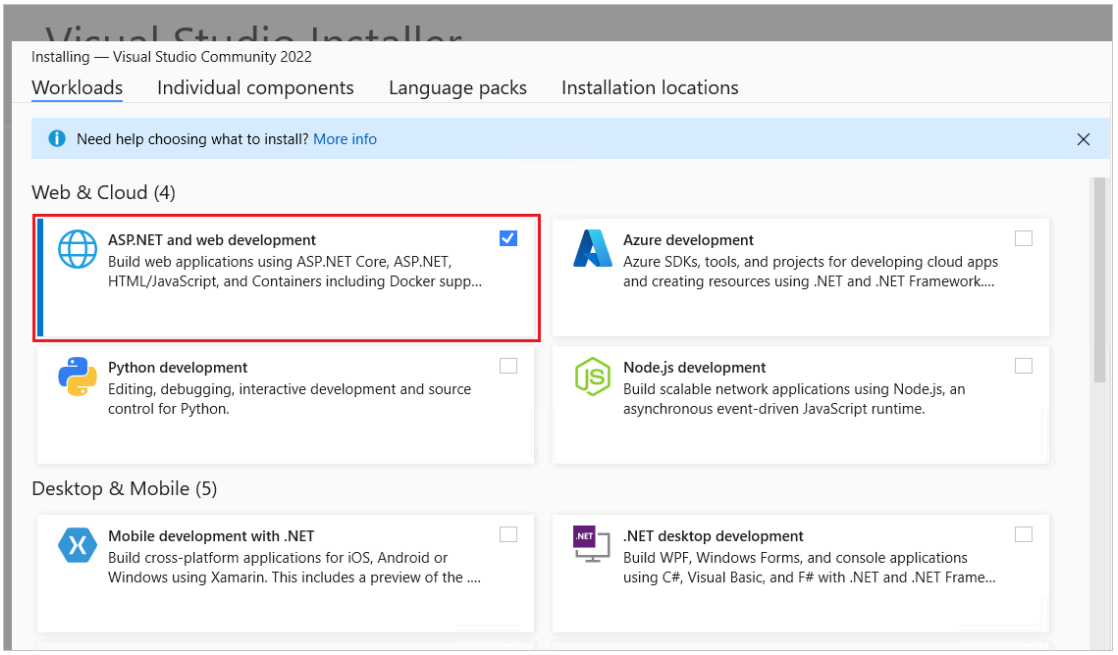
View or download sample code (how to download)

Prerequisites

- Node.js with npm

Visual Studio

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



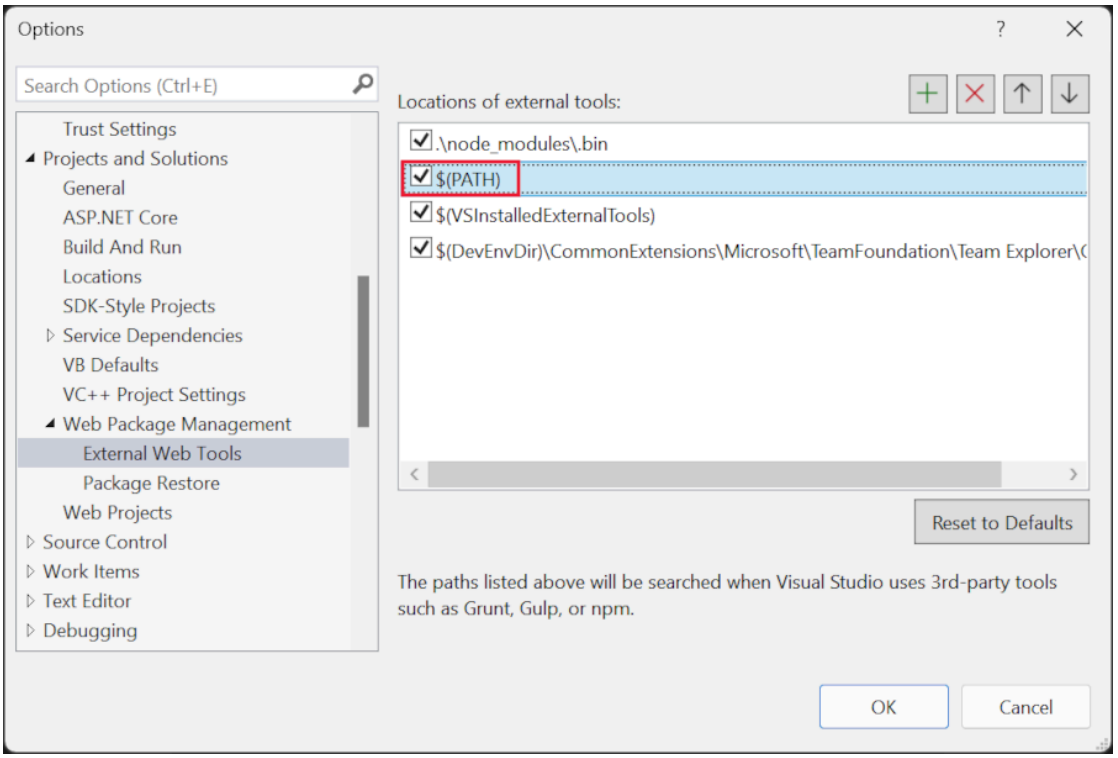
Create the ASP.NET Core web app

Visual Studio

By default, Visual Studio uses the version of npm found in its installation directory. To configure Visual Studio to look for npm in the `PATH` environment variable:

Launch Visual Studio. At the start window, select **Continue without code**.

1. Navigate to **Tools** > **Options** > **Projects and Solutions** > **Web Package Management** > **External Web Tools**.
2. Select the `$(PATH)` entry from the list. Select the up arrow to move the entry to the second position in the list, and select **OK**:



.

To create a new ASP.NET Core web app:

1. Use the **File** > **New** > **Project** menu option and choose the **ASP.NET Core Empty** template. Select **Next**.
2. Name the project `SignalRWebpack`, and select **Create**.
3. Select **.NET 8.0 (Long Term Support)** from the **Framework** drop-down. Select **Create**.

Add the Microsoft.TypeScript.MSBuild NuGet package to the project:

1. In **Solution Explorer**, right-click the project node and select **Manage NuGet Packages**. In the **Browse** tab, search for `Microsoft.TypeScript.MSBuild` and then select **Install** on the right to install the package.

Visual Studio adds the NuGet package under the **Dependencies** node in **Solution Explorer**, enabling TypeScript compilation in the project.

Configure the server

In this section, you configure the ASP.NET Core web app to send and receive SignalR messages.

1. In `Program.cs`, call AddSignalR:

```
C#
```