Tutorial: Create a minimal API with ASP.NET Core

08/21/2024

By Rick Anderson and Tom Dykstra

Minimal APIs are architected to create HTTP APIs with minimal dependencies. They're ideal for microservices and apps that want to include only the minimum files, features, and dependencies in ASP.NET Core.

This tutorial teaches the basics of building a minimal API with ASP.NET Core. Another approach to creating APIs in ASP.NET Core is to use controllers. For help with choosing between minimal APIs and controller-based APIs, see APIs overview. For a tutorial on creating an API project based on controllers that contains more features, see Create a web API.

Overview

This tutorial creates the following API:

Expand table

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

Prerequisites

Visual Studio Code

- Visual Studio Code
- C# Dev Kit for Visual Studio Code
- .NET 9 SDK

You can follow the Visual Studio Code instructions on macOS, Linux, or Windows. Changes may be required if you use an integrated development environment (IDE) other than Visual Studio Code.

Create an API project

Visual Studio Code

- Open the integrated terminal .
- Change directories (`cd`) to the folder that will contain the project folder.
- Run the following commands:

  .NET CLI

  ```
  dotnet new web -o TodoApi
  cd TodoApi
  code -r ../TodoApi
  ```

- When a dialog box asks if you want to trust the authors, select **Yes**.
- When a dialog box asks if you want to add required assets to the project, select **Yes**.

  The preceding commands create a new web minimal API project and open it in Visual Studio Code.

Examine the code

The `Program.cs` file contains the following code:

C#

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

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

app.Run();
```

The preceding code:

- Creates a WebApplicationBuilder and a WebApplication with preconfigured defaults.
- Creates an HTTP GET endpoint `/` that returns `Hello World!`.

Run the app

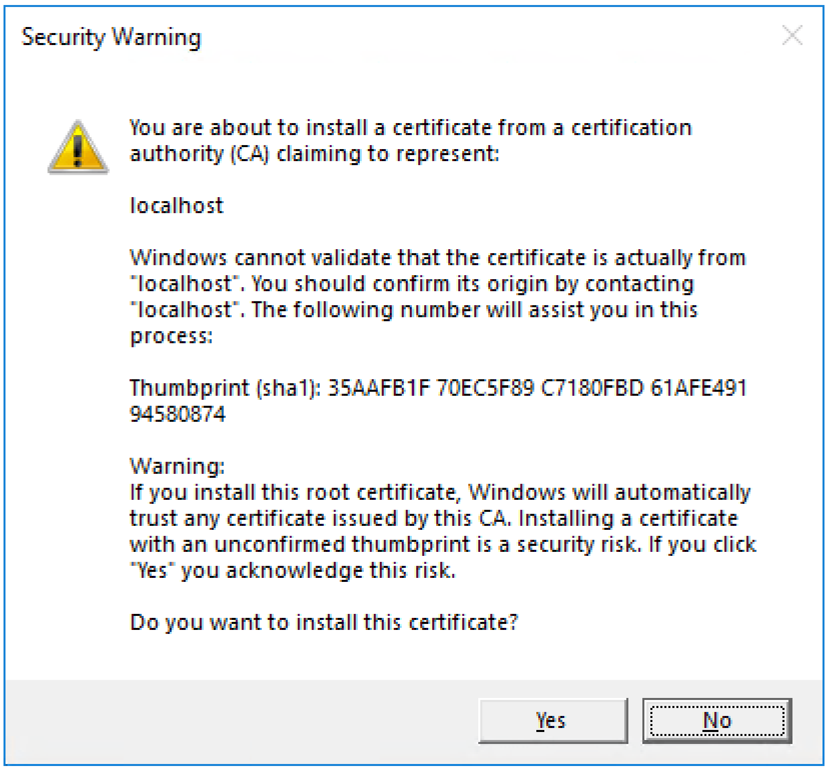
Visual Studio Code

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

.NET CLI

```
dotnet dev-certs https --trust
```

The preceding command displays the following dialog, provided the certificate was not previously trusted:



Security Warning

You are about to install a certificate from a certification authority (CA) claiming to represent:

localhost

Windows cannot validate that the certificate is actually from "localhost". You should confirm its origin by contacting "localhost". The following number will assist you in this process:

Thumbprint (sha1): 35AAFB1F 70EC5F89 C7180FBD 61AFE491 94580874

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfirmed thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

Do you want to install this certificate?

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

  For more information, see the **Trust the ASP.NET Core HTTPS development certificate** section of the Enforcing SSL article.

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

In Visual Studio Code, press `Ctrl` + `F5` (Windows) or `control` + `F5` (macOS) to run the app without debugging.

The default browser launches with the following URL: `https://localhost:<port>` where `<port>` is the randomly generated port number.

Close the browser window.

In Visual Studio Code, from the *Run* menu, select *Stop Debugging* or press `Shift` + `F5` to stop the app.

Add NuGet packages

NuGet packages must be added to support the database and diagnostics used in this tutorial.

Visual Studio Code

- Run the following commands:

.NET CLI

```
dotnet add package Microsoft.EntityFrameworkCore.InMemory
dotnet add package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
```

The model and database context classes

- In the project folder, create a file named `Todo.cs` with the following code:

C#

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

The preceding code creates the model for this app. A *model* is a class that represents data that the app manages.

- Create a file named `TodoDb.cs` with the following code:

C#

```csharp
using Microsoft.EntityFrameworkCore;

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

    public DbSet<Todo> Todos => Set<Todo>();
}
```

The preceding code defines the *database context*, which is the main class that coordinates Entity Framework functionality for a data model. This class derives from the Microsoft.EntityFrameworkCore.DbContext class.

Add the API code

- Replace the contents of the `Program.cs` file with the following code:

C#

```csharp
using Microsoft.EntityFrameworkCore;

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

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

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

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

app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

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

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});

app.Run();
```

The following highlighted code adds the database context to the dependency injection (DI) container and enables displaying database-related exceptions:

C#

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

The DI container provides access to the database context and other services.

Visual Studio Code

Create API testing UI with Swagger

There are many available web API testing tools to choose from, and you can follow this tutorial's introductory API test steps with your own preferred tool.

This tutorial utilizes the .NET package NSwag.AspNetCore , which integrates Swagger tools for generating a testing UI adhering to the OpenAPI specification:

- NSwag: A .NET library that integrates Swagger directly into ASP.NET Core applications, providing middleware and configuration.
- Swagger: A set of open-source tools such as OpenAPIGenerator and SwaggerUI that generate API testing pages that follow the OpenAPI specification.
- OpenAPI specification: A document that describes the capabilities of the API, based on the XML and attribute annotations within the controllers and models.

For more information on using OpenAPI and NSwag with ASP.NET, see ASP.NET Core web API documentation with Swagger / OpenAPI.

Install Swagger tooling

- Run the following command:

.NET CLI

```
dotnet add package NSwag.AspNetCore
```

The previous command adds the NSwag.AspNetCore package, which contains tools to generate Swagger documents and UI.

Configure Swagger middleware

- In Program.cs add the following highlighted code before `app` is defined in line `var app = builder.Build();`

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddOpenApiDocument(config =>
{
    config.DocumentName = "TodoAPI";
    config.Title = "TodoAPI v1";
    config.Version = "v1";
});
var app = builder.Build();
```

In the previous code:

- `builder.Services.AddEndpointsApiExplorer();`: Enables the API Explorer, which is a service that provides metadata about the HTTP API. The API Explorer is used by Swagger to generate the Swagger document.
- `builder.Services.AddOpenApiDocument(config => {...});`: Adds the Swagger OpenAPI document generator to the application services and configures it to provide more information about the API, such as its title and version. For information on providing more robust API details, see Get started with NSwag and ASP.NET Core
- Add the following highlighted code to the next line after `app` is defined in line `var app = builder.Build();`

C#

```
var app = builder.Build();
if (app.Environment.IsDevelopment())
{
    app.UseOpenApi();
    app.UseSwaggerUi(config =>
    {
        config.DocumentTitle = "TodoAPI";
        config.Path = "/swagger";
        config.DocumentPath = "/swagger/{documentName}/swagger.json";
        config.DocExpansion = "list";
    });
}
```

The previous code enables the Swagger middleware for serving the generated JSON document and the Swagger UI. Swagger is only enabled in a development environment. Enabling Swagger in a production environment could expose potentially sensitive details about the API's structure and implementation.

Test posting data

The following code in `Program.cs` creates an HTTP POST endpoint `/todoitems` that adds data to the in-memory database:

C#

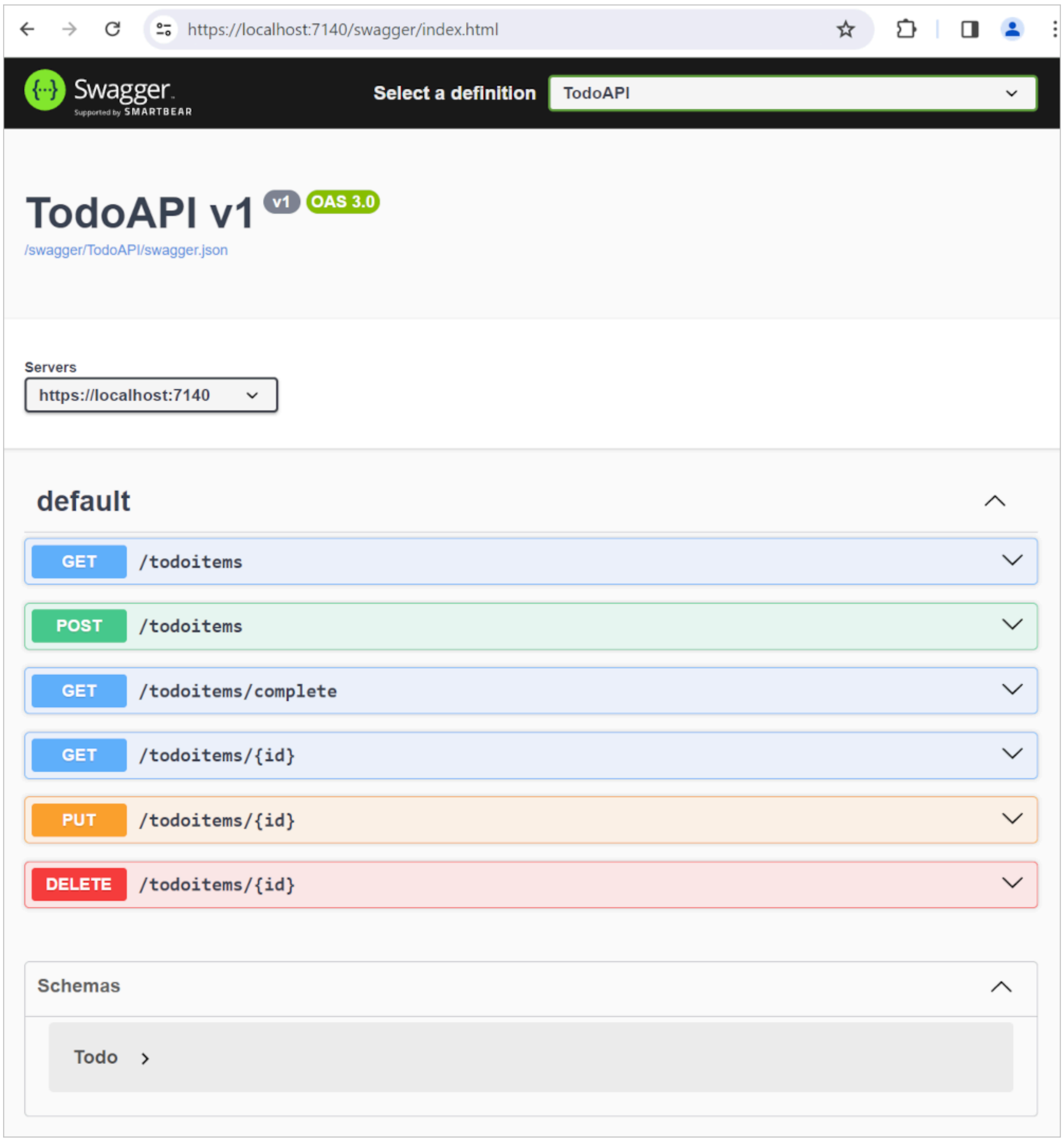
```
app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

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

Run the app. The browser displays a 404 error because there's no longer a `/` endpoint.

The POST endpoint will be used to add data to the app.

Visual Studio Code

- With the app still running, in the browser, navigate to `https://localhost:<port>/swagger` to display the API testing page generated by Swagger.

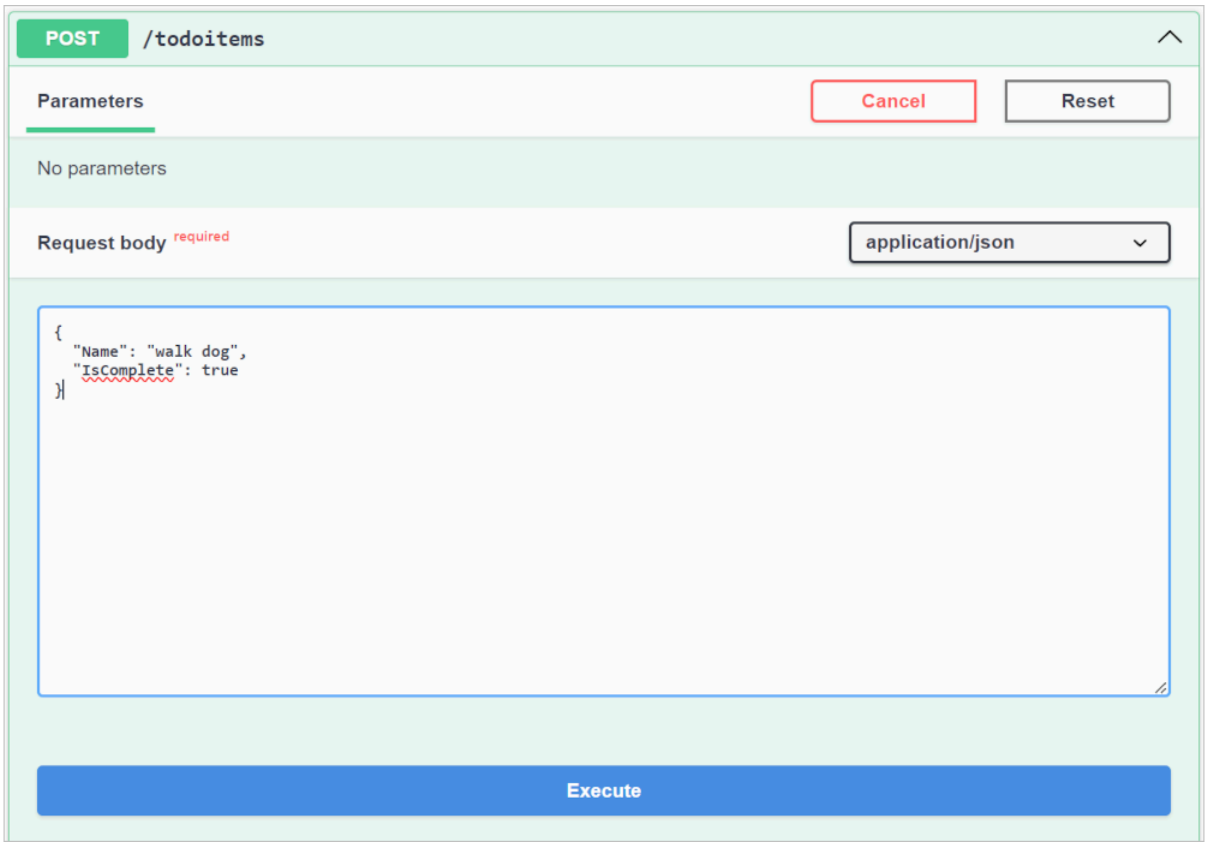


- On the Swagger API testing page, select **Post /todoitems** > **Try it out**.
- Note that the **Request body** field contains a generated example format reflecting the parameters for the API.
- In the request body enter JSON for a to-do item, without specifying the optional `id`:

JSON

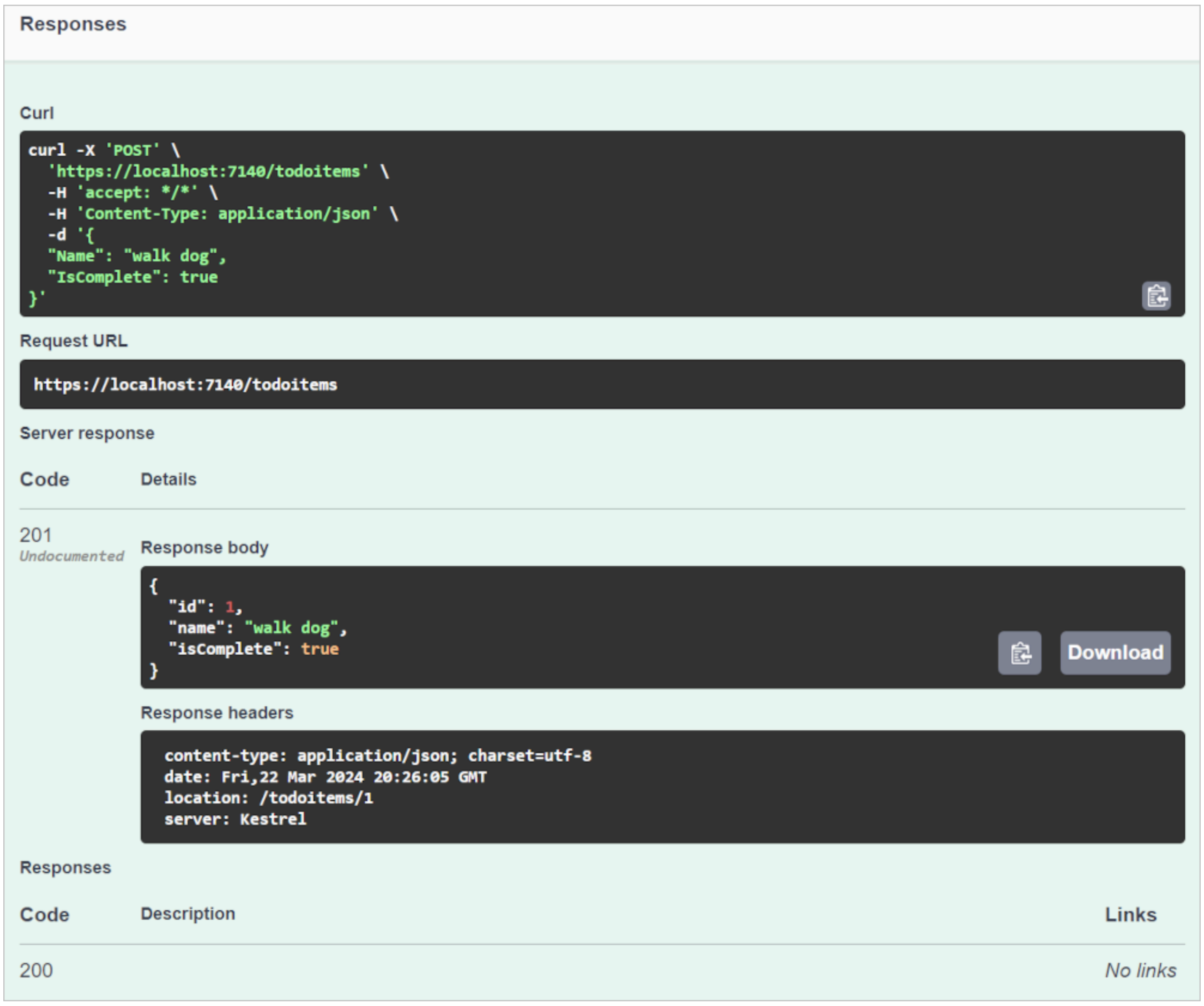
```json
{
  "name":"walk dog",
  "isComplete":true
}
```

- Select **Execute**.



```
{
  "Name": "walk dog",
  "IsComplete": true
}
```

Swagger provides a **Responses** pane below the **Execute** button.



```
curl -X 'POST' \
  'https://localhost:7140/todoitems' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "Name": "walk dog",
  "IsComplete": true
}'
```

```
https://localhost:7140/todoitems
```

```
{
  "id": 1,
  "name": "walk dog",
  "isComplete": true
}
```

```
content-type: application/json; charset=utf-8
date: Fri,22 Mar 2024 20:26:05 GMT
location: /todoitems/1
server: Kestrel
```

Note a few of the useful details:

- cURL: Swagger provides an example cURL command in Unix/Linux syntax, which can be run at the command line with any bash shell that uses Unix/Linux syntax, including Git Bash from Git for Windows .
- Request URL: A simplified representation of the HTTP request made by Swagger UI's JavaScript code for the API call. Actual requests can include details such as headers and query parameters and a request body.
- Server response: Includes the response body and headers. The response body shows the `id` was set to `1`.
- Response Code: A 201 `HTTP` status code was returned, indicating that the request was successfully processed and resulted in the creation of a new resource.

Examine the GET endpoints

The sample app implements several GET endpoints by calling `MapGet`:

Expand table

API	Description	Request body	Response body
`GET /todoitems`	Get all to-do items	None	Array of to-do items
`GET /todoitems/complete`	Get all completed to-do items	None	Array of to-do items
`GET /todoitems/{id}`	Get an item by ID	None	To-do item

C#

```csharp
app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

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

Test the GET endpoints

Visual Studio Code

Test the app by calling the endpoints from a browser or Swagger.

- In Swagger select **GET /todoitems** > **Try it out** > **Execute**.
- Alternatively, call **GET /todoitems** from a browser by entering the URI `http://localhost:<port>/todoitems`. For example, `http://localhost:5001/todoitems`

The call to `GET /todoitems` produces a response similar to the following:

JSON

```
[
  {
    "id": 1,
    "name": "walk dog",
    "isComplete": true
  }
]
```

- Call **GET /todoitems/{id}** in Swagger to return data from a specific id:
  - Select **GET /todoitems** > **Try it out**.
  - Set the **id** field to `1` and select **Execute**.
- Alternatively, call **GET /todoitems** from a browser by entering the URI `https://localhost:<port>/todoitems/1`. For example, `https://localhost:5001/todoitems/1`
- The response is similar to the following:

  JSON

  ```
  {
    "id": 1,
    "name": "walk dog",
    "isComplete": true
  }
  ```

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

Return values

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

The return types can represent a wide range of HTTP status codes. For example, `GET /todoitems/{id}` can return two different status values:

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

Examine the PUT endpoint

The sample app implements a single PUT endpoint using `MapPut`:

C#

```
app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

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

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

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

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

Test the PUT endpoint

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

Update the to-do item that has `Id = 1` and set its name to `"feed fish"`.

Visual Studio Code

Use Swagger to send a PUT request:

- Select **Put /todoitems/{id}** > **Try it out**.
- Set the **id** field to `1`.
- Set the request body to the following JSON:

JSON

```json
{
  "id": 1,
  "name": "feed fish",
  "isComplete": false
}
```

- Select **Execute**.

Examine and test the DELETE endpoint

The sample app implements a single DELETE endpoint using `MapDelete`:

C#

```csharp
app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

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

Visual Studio Code

Use Swagger to send a DELETE request:

- Select **DELETE /todoitems/{id}** > **Try it out**.

- Set the **ID** field to `1` and select **Execute**.

  The DELETE request is sent to the app and the response is displayed in the **Responses** pane. The response body is empty, and the **Server response** status code is 204.

Use the MapGroup API

The sample app code repeats the `todoitems` URL prefix each time it sets up an endpoint. APIs often have groups of endpoints with a common URL prefix, and the MapGroup method is available to help organize such groups. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata.

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

Visual Studio Code

C#

```csharp
using Microsoft.EntityFrameworkCore;

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

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddOpenApiDocument(config =>
{
    config.DocumentName = "TodoAPI";
    config.Title = "TodoAPI v1";
    config.Version = "v1";
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseOpenApi();
    app.UseSwaggerUi(config =>
    {
        config.DocumentTitle = "TodoAPI";
        config.Path = "/swagger";
        config.DocumentPath = "/swagger/{documentName}/swagger.json";
        config.DocExpansion = "list";
    });
}

var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", async (TodoDb db) =>
    await db.Todos.ToListAsync());

todoItems.MapGet("/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

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

todoItems.MapPost("/", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

todoItems.MapPut("/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

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

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

todoItems.MapDelete("/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});

app.Run();
```

The preceding code has the following changes:

- Adds `var todoItems = app.MapGroup("/todoitems");` to set up the group using the URL prefix `/todoitems`.
- Changes all the `app.Map<HttpVerb>` methods to `todoItems.Map<HttpVerb>`.
- Removes the URL prefix `/todoitems` from the `Map<HttpVerb>` method calls.

Test the endpoints to verify that they work the same.

Use the TypedResults API

Returning TypedResults rather than Results has several advantages, including testability and automatically returning the response type metadata for OpenAPI to describe the endpoint. For more information, see TypedResults vs Results.

The `Map<HttpVerb>` methods can call route handler methods instead of using lambdas. To see an example, update *Program.cs* with the following code:

Visual Studio Code

C#

```csharp
using Microsoft.EntityFrameworkCore;

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

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddOpenApiDocument(config =>
{
    config.DocumentName = "TodoAPI";
    config.Title = "TodoAPI v1";
    config.Version = "v1";
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseOpenApi();
    app.UseSwaggerUi(config =>
    {
        config.DocumentTitle = "TodoAPI";
        config.Path = "/swagger";
        config.DocumentPath = "/swagger/{documentName}/swagger.json";
        config.DocExpansion = "list";
    });
}
```

```
var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);

app.Run();

static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(todo)
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(Todo todo, TodoDb db)
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return TypedResults.Created($"/todoitems/{todo.Id}", todo);
}

static async Task<IResult> UpdateTodo(int id, Todo inputTodo, TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

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

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

The `Map<HttpVerb>` code now calls methods instead of lambdas:

C#

```
var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);
```

These methods return objects that implement IResult and are defined by TypedResults:

C#

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Where(t => t.IsComplete).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(todo)
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(Todo todo, TodoDb db)
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return TypedResults.Created($"/todoitems/{todo.Id}", todo);
}

static async Task<IResult> UpdateTodo(int id, Todo inputTodo, TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

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

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

Unit tests can call these methods and test that they return the correct type. For example, if the method is `GetAllTodos`:

C#

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}
```

Unit test code can verify that an object of type Ok<Todo[]> is returned from the handler method. For example:

C#

```
public async Task GetAllTodos_ReturnsOkOfTodosResult()
{
    // Arrange
    var db = CreateDbContext();

    // Act
    var result = await TodosApi.GetAllTodos(db);
    // Assert: Check for the correct returned type
    Assert.IsType<Ok<Todo[]>>(result);
}
```

Prevent over-posting

Currently the sample app exposes the entire `Todo` object. In production applications, a subset of the model is often used to restrict the data that can be input and returned. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO can be used to:

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

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

C#

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

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

Verify you can post and get the secret field.

Create a file named `TodoItemDTO.cs` with the following code:

C#

```
public class TodoItemDTO
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
    public TodoItemDTO() { }
    public TodoItemDTO(Todo todoItem) =>
    (Id, Name, IsComplete) = (todoItem.Id, todoItem.Name, todoItem.IsComplete);
}
```

Replace the contents of the `Program.cs` file with the following code to use this DTO model:

Visual Studio Code

C#

```
using Microsoft.EntityFrameworkCore;

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

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddOpenApiDocument(config =>
{
    config.DocumentName = "TodoAPI";
    config.Title = "TodoAPI v1";
    config.Version = "v1";
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseOpenApi();
    app.UseSwaggerUi(config =>
    {
        config.DocumentTitle = "TodoAPI";
        config.Path = "/swagger";
        config.DocumentPath = "/swagger/{documentName}/swagger.json";
        config.DocExpansion = "list";
    });
}

RouteGroupBuilder todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);

app.Run();
```

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Select(x => new
TodoItemDTO(x)).ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db) {
    return TypedResults.Ok(await db.Todos.Where(t => t.IsComplete).Select(x =>
new TodoItemDTO(x)).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(new TodoItemDTO(todo))
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(TodoItemDTO todoItemDTO, TodoDb db)
{
    var todoItem = new Todo
    {
        IsComplete = todoItemDTO.IsComplete,
        Name = todoItemDTO.Name
    };

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

    todoItemDTO = new TodoItemDTO(todoItem);

    return TypedResults.Created($"/todoitems/{todoItem.Id}", todoItemDTO);
}

static async Task<IResult> UpdateTodo(int id, TodoItemDTO todoItemDTO, TodoDb
db)
{
    var todo = await db.Todos.FindAsync(id);

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

    todo.Name = todoItemDTO.Name;
    todo.IsComplete = todoItemDTO.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
```

```
            await db.SaveChangesAsync();
            return TypedResults.NoContent();
        }

        return TypedResults.NotFound();
    }
```

Verify you can post and get all fields except the secret field.

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

- Configure JSON serialization options.
- Handle errors and exceptions: The developer exception page is enabled by default in the development environment for minimal API apps. For information about how to handle errors and exceptions, see Handle errors in ASP.NET Core APIs.
- For an example of testing a minimal API app, see this GitHub sample .
- OpenAPI support in minimal APIs.
- Quickstart: Publish to Azure.
- Organizing ASP.NET Core Minimal APIs .

Learn more

See Minimal APIs quick reference