Tutorial: Create a web API with ASP.NET Core

Article • 02/10/2023 • 64 minutes to read

By Rick Anderson and Kirk Larkin

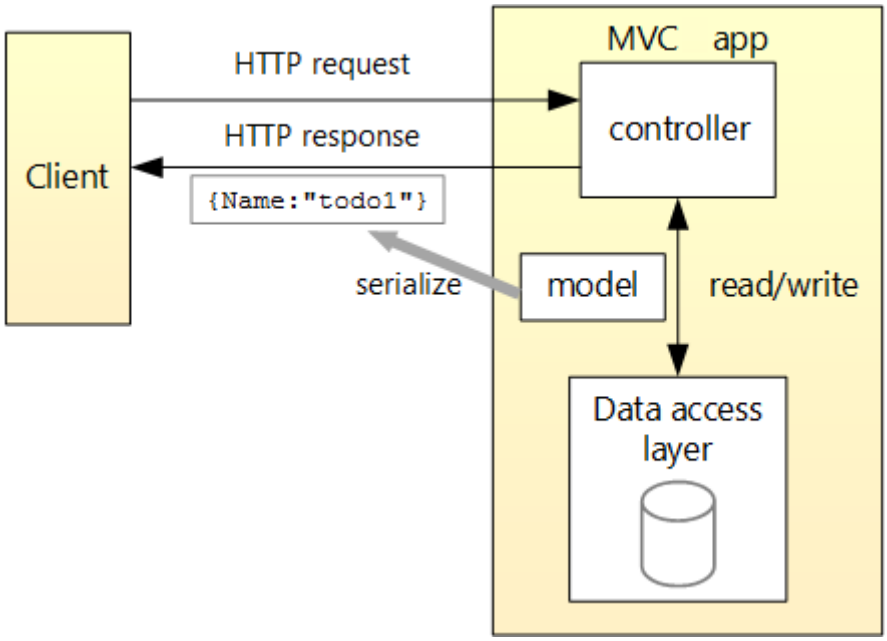
This tutorial teaches the basics of building a controller-based web API that uses a database. Another approach to creating APIs in ASP.NET Core is to create *minimal APIs*. For help in choosing between minimal APIs and controller-based APIs, see APIs overview. For a tutorial on creating a minimal API,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

Overview

This tutorial creates the following API:

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

The following diagram shows the design of the app.



Prerequisites

Visual Studio Code

- Visual Studio Code
- C# for Visual Studio Code (latest version)
- .NET 7.0 SDK

The Visual Studio Code instructions use the .NET CLI for ASP.NET Core development functions such as project creation. You can follow these instructions on macOS, Linux, or Windows and with any code editor. Minor changes may be required if you use something other than Visual Studio Code.

Create a web 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 webapi -o TodoApi
cd TodoApi
dotnet add package Microsoft.EntityFrameworkCore.InMemory
code -r ../TodoApi
```

These commands:

- Create a new web API project and open it in Visual Studio Code.
- Add a NuGet package that is needed for the next section.

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

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

Test the project

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

Visual Studio Code

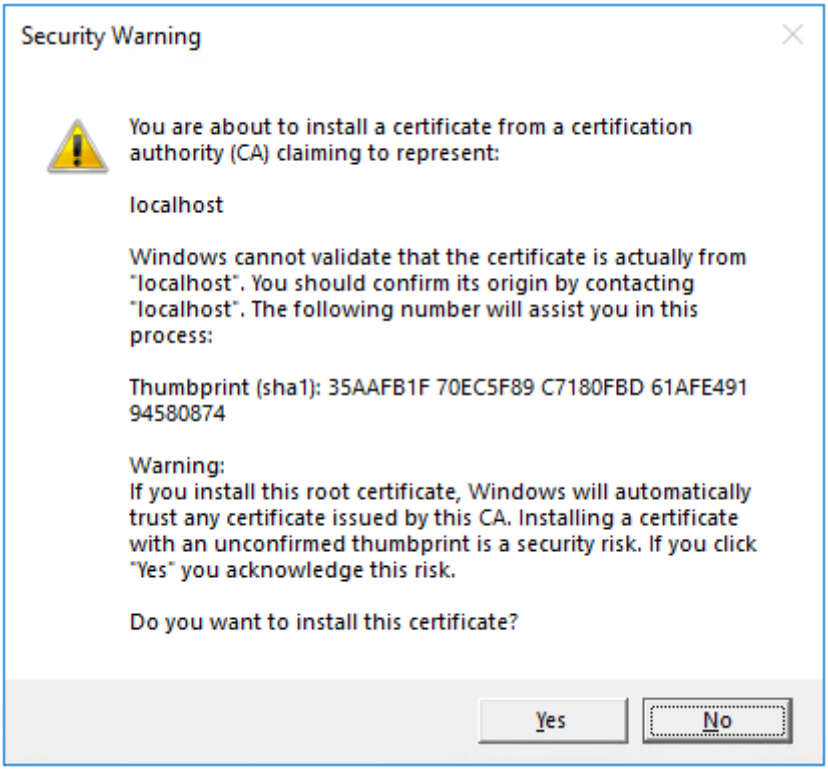
- Trust the HTTPS development certificate by running the following command:

.NET CLI

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

The preceding command doesn't work on Linux. See your Linux distribution's documentation for trusting a certificate.

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



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

  See Trust the ASP.NET Core HTTPS development certificate for more information.

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

Run the app:

- Press Ctrl+F5 to run the app.
- Visual Studio Code launches the default browser to `https://localhost:<port>`, where `<port>` is the randomly chosen port number displayed in the output. There is no endpoint at `https://localhost:<port>` so the browser returns HTTP 404 Not Found . Append `/swagger` to the URL, `https://localhost:<port>/swagger`.

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

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

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

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

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

JSON similar to the following example is returned:

JSON


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


Add a model class

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

Visual Studio Code

- Add a folder named `Models`.
- Add a `TodoItem.cs` file to the `Models` folder with the following code:

C#

```
namespace TodoApi.Models;

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

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

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

Add a database context

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

Visual Studio Code / Visual Studio for Mac

- Add a `TodoContext.cs` file to the `Models` folder.

- Enter the following code:

C#

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models;
```

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

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

Register the database context

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

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

C#

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

var builder = WebApplication.CreateBuilder(args);

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

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The preceding code:

- Adds `using` directives.

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

Scaffold a controller

Visual Studio Code / Visual Studio for Mac

Make sure that all of your changes so far are saved.

- Control-click the **TodoAPI** project and select **Open in Terminal**. The terminal opens at the `TodoAPI` project folder. Run the following commands:

.NET CLI

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design -v
7.0.0
dotnet add package Microsoft.EntityFrameworkCore.Design -v 7.0.0
dotnet add package Microsoft.EntityFrameworkCore.SqlServer -v 7.0.0
dotnet tool uninstall -g dotnet-aspnet-codegenerator
dotnet tool install -g dotnet-aspnet-codegenerator
```

The preceding commands:

- Add NuGet packages required for scaffolding.
- Install the scaffolding engine (`dotnet-aspnet-codegenerator`) after uninstalling any possible previous version.

Build the project.

Run the following command:

.NET CLI

```
dotnet-aspnet-codegenerator controller -name TodoItemsController -
async -api -m TodoItem -dc TodoContext -outDir Controllers
```

The preceding command scaffolds the `TodoItemsController`.

The generated code:

- Marks the class with the [ApiController] attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.

- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

The ASP.NET Core templates for:

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

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

Update the PostTodoItem create method

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

C#

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

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

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

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

The CreatedAtAction method:

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

`CreatedAtAction` call.

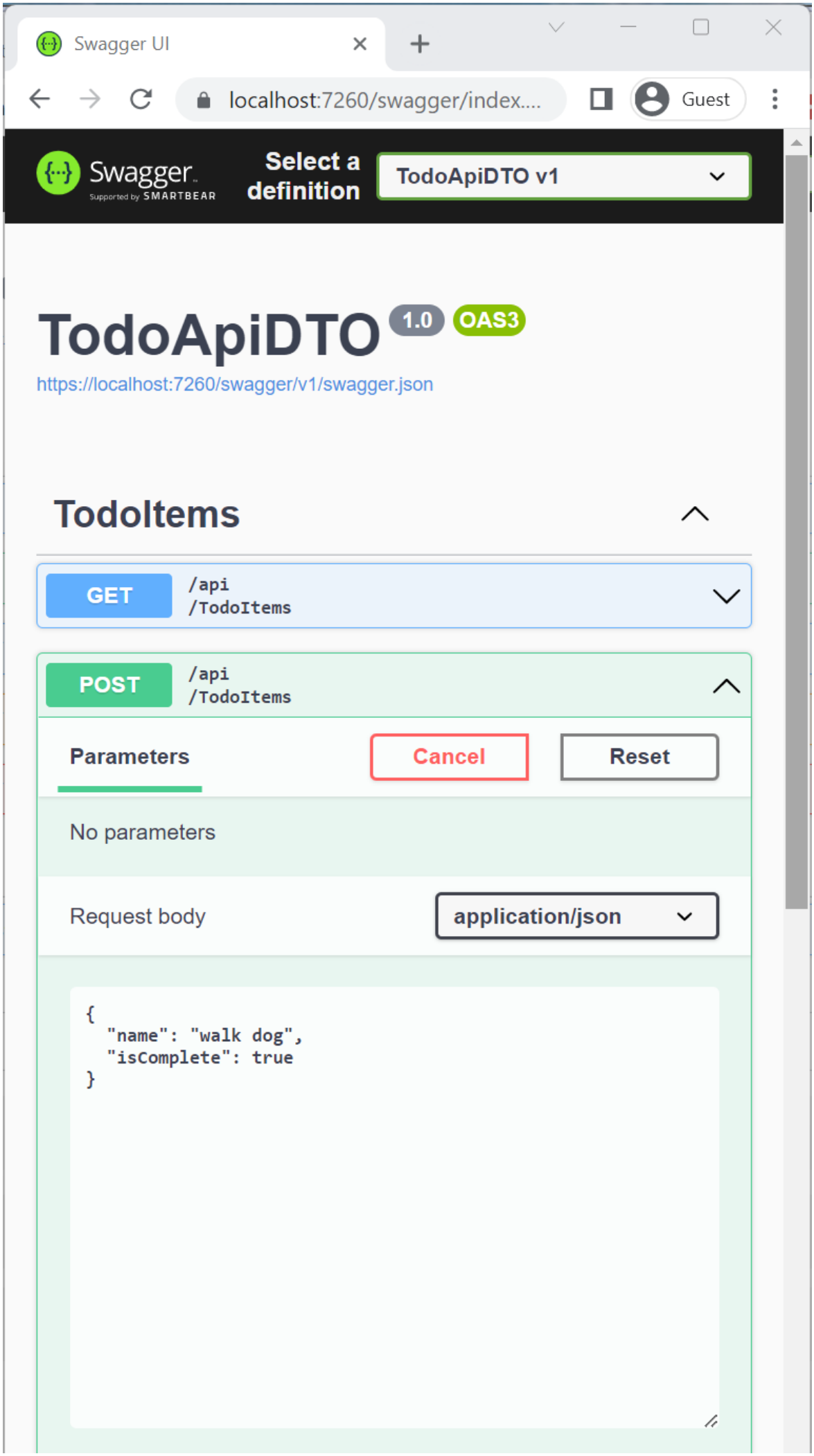
Test PostTodoItem

- Press Ctrl+F5 to run the app.
- In the Swagger browser window, select **POST /api/TodoItems**, and then select **Try it out**.
- In the **Request body** input window, update the JSON. For example,

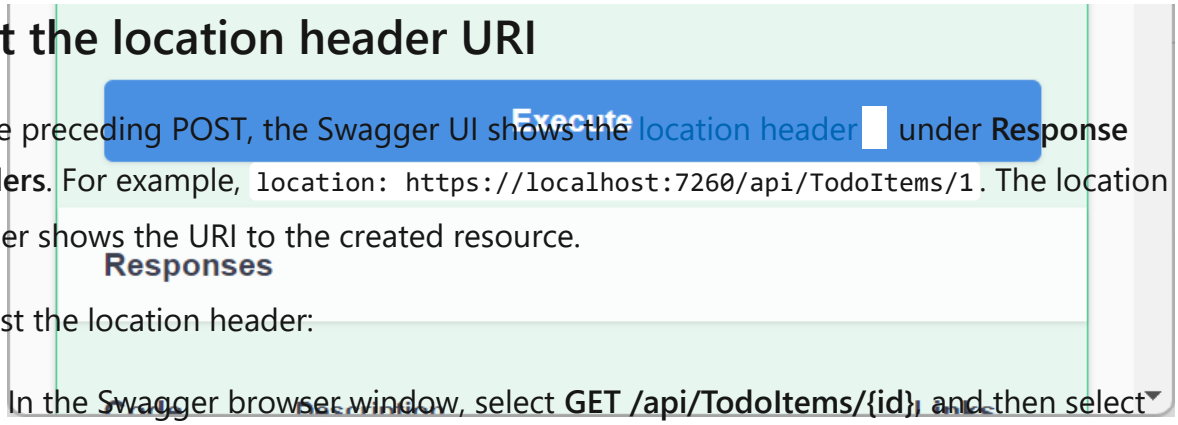
```
JSON
```

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

- Select **Execute**


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

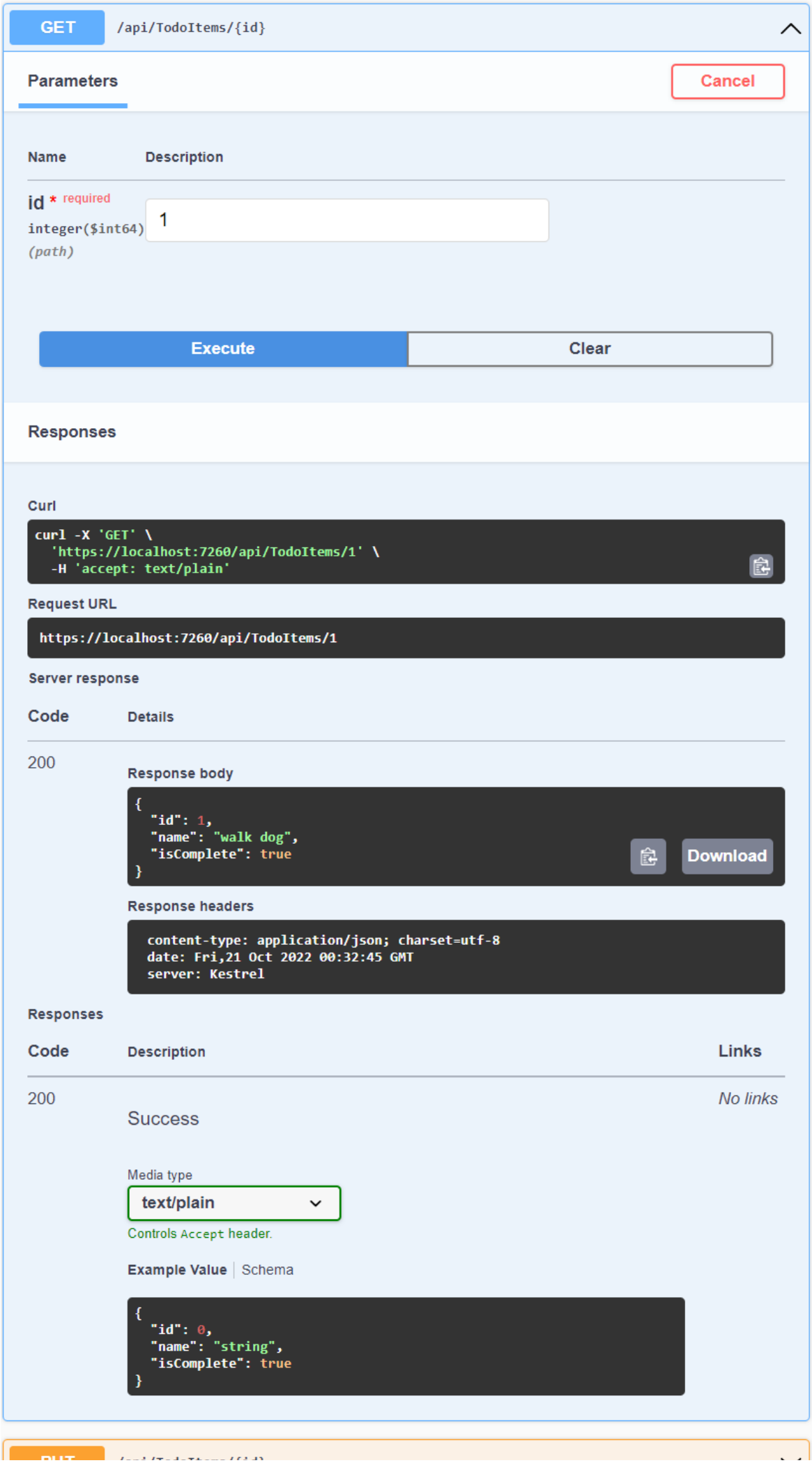
Test the location header URI



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

To test the location header:

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


GET
/api/TodoItems/{id}
Parameters
Cancel
Name
Description
id * required
integer($int64)
(path)
1
Execute
Clear
Responses
Curl
curl -X 'GET' \
'https://localhost:7260/api/TodoItems/1' \
-H 'accept: text/plain'
Request URL
https://localhost:7260/api/TodoItems/1
Server response
Code
Details
200
Response body
{
"id": 1,
"name": "walk dog",
"isComplete": true
}
Download
Response headers
content-type: application/json; charset=utf-8
date: Fri,21 Oct 2022 00:32:45 GMT
server: Kestrel
Responses
Code
Description
Links
200
Success
No links
Media type
text/plain
Controls Accept header.
Example Value
Schema
{
"id": 0,
"name": "string",
"isComplete": true
}

Examine the GET methods

Two GET endpoints are implemented:

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

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

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

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

Routing and URL paths

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

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

  C#

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

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

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

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

C#

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

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

    return todoItem;
}
```

Return values

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

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

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

The PutTodoItem method

Examine the `PutTodoItem` method:

C#

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

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

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

    return NoContent();
}
```

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

Test the PutTodoItem method

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

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

The DeleteTodoItem method

Examine the `DeleteTodoItem` method:

C#

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

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

    return NoContent();
}
```

Test the DeleteTodoItem method

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

Test with http-repl, Postman, or curl

http-repl, Postman , and curl are often used to test API's. Swagger uses `curl` and shows the `curl` command it submitted.

For instructions on these tools, see the following links:

- Test APIs with Postman
- Install and test APIs with http-repl

For more information on `http-repl`, see Test web APIs with the HttpRepl.

Prevent over-posting

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

A DTO may be used to:

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

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

C#

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

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

Verify you can post and get the secret field.

Create a DTO model:

C#

```
namespace TodoApi.Models;

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

Update the `TodoItemsController` to use `TodoItemDTO`:

C#

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

namespace TodoApi.Controllers;

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

    public TodoItemsController(TodoContext context)
    {
        _context = context;
```

```
    }

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

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

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

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

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

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

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

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
```

```
        {
            return NotFound();
        }

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

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

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

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

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

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

        return NoContent();
    }

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

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

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

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

Call the web API with JavaScript

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

Web API video series

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

Add authentication support to a web API

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

- Azure Active Directory
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

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

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

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

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

Publish to Azure

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

Additional resources

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

For more information, see the following resources:

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