Create a web API with ASP.NET Core MVC and Visual Studio for Windows

By Rick Anderson and Mike Wasson

In this tutorial, you'll build a web API for managing a list of "to-do" items. You won't build a UI.

There are 3 versions of this tutorial:

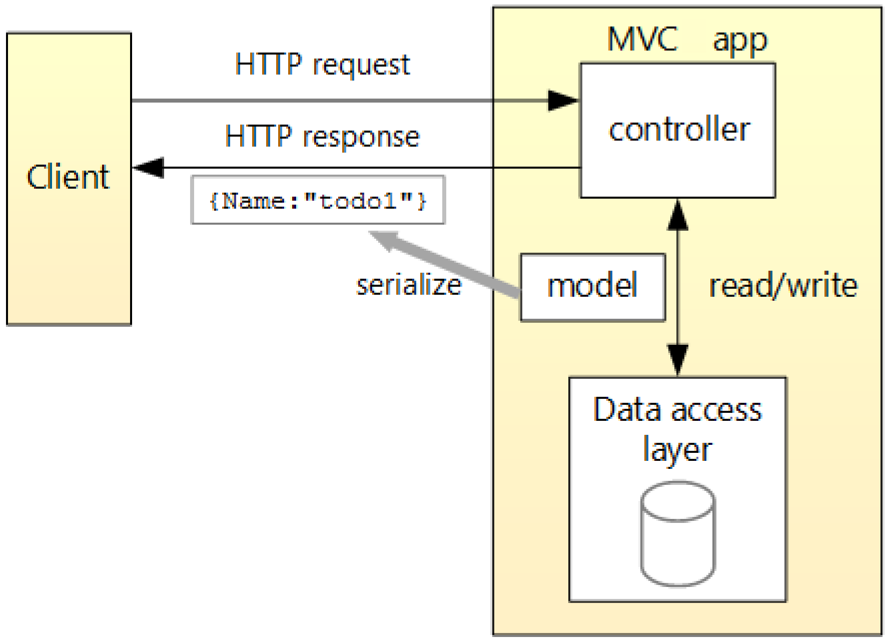
- Windows: Web API with Visual Studio for Windows (This tutorial)
- macOS: Web API with Visual Studio for Mac
- macOS, Linux, Windows: Web API with Visual Studio Code

Overview

Here is the API that you'll create:

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

The following diagram shows the basic design of the app.

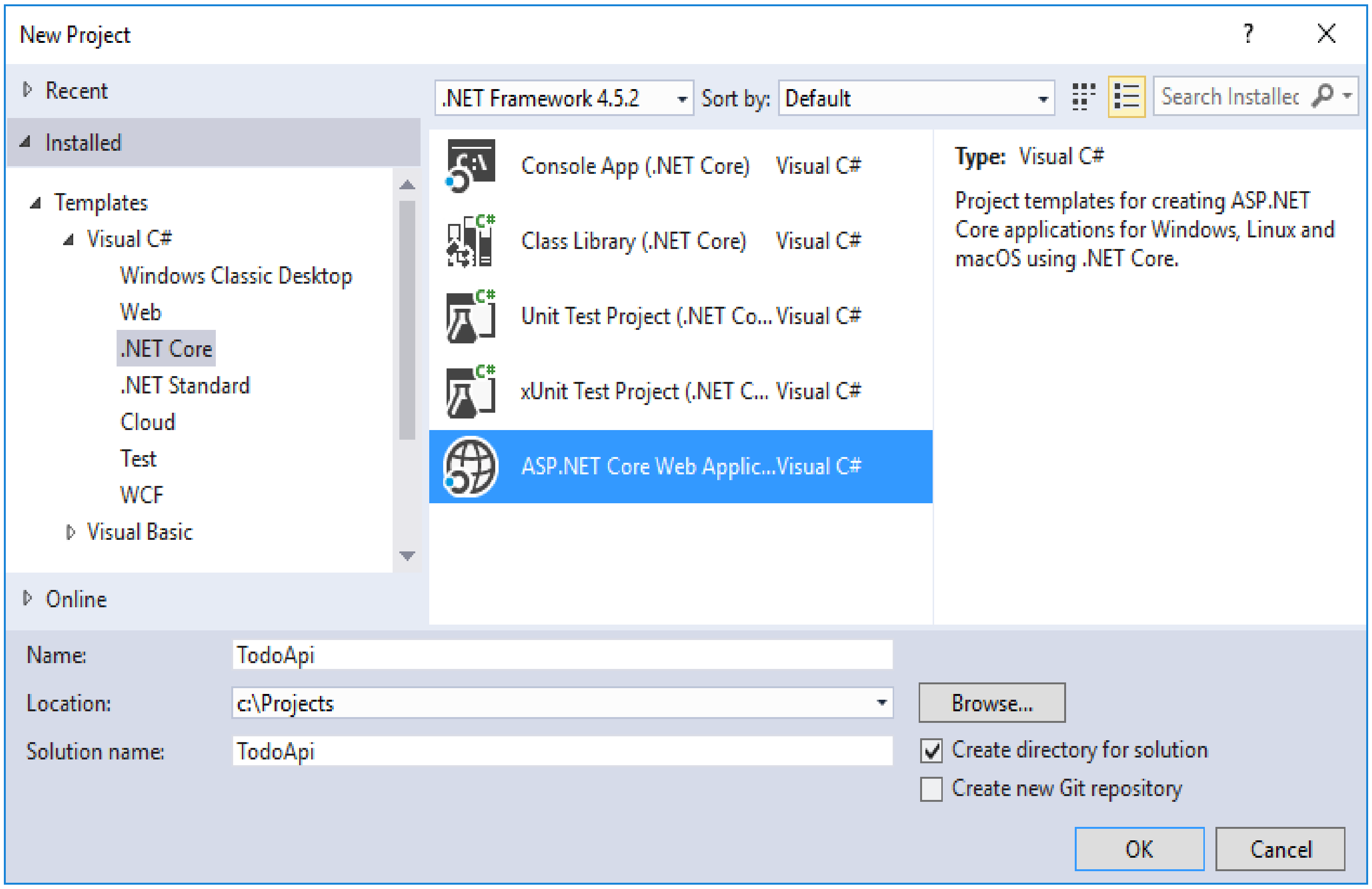


- The client is whatever consumes the web API (mobile app, browser, etc.). We aren't writing a client in this tutorial. We'll use Postman or curl to test the app.
- A *model* is an object that represents the data in your application. In this case, the only model is a to-do item. Models are represented as C# classes, also know as Plain Old C# Object (POCOs).
- A *controller* is an object that handles HTTP requests and creates the HTTP response. This app will have a single controller.
- To keep the tutorial simple, the app doesn't use a persistent database. Instead, it stores to-do items in an in-memory database.

Create the project

From Visual Studio, select File menu, > New > Project.

Select the ASP.NET Core Web Application (.NET Core) project template. Name the project `TodoApi` and select OK.



New Project

Recent
Installed
Templates
Visual C#
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
Online

.NET Framework 4.5.2 | Sort by: Default | Search Installed

Console App (.NET Core)	Visual C#
Class Library (.NET Core)	Visual C#
Unit Test Project (.NET Co...	Visual C#
xUnit Test Project (.NET C...	Visual C#
ASP.NET Core Web Applic...	Visual C#

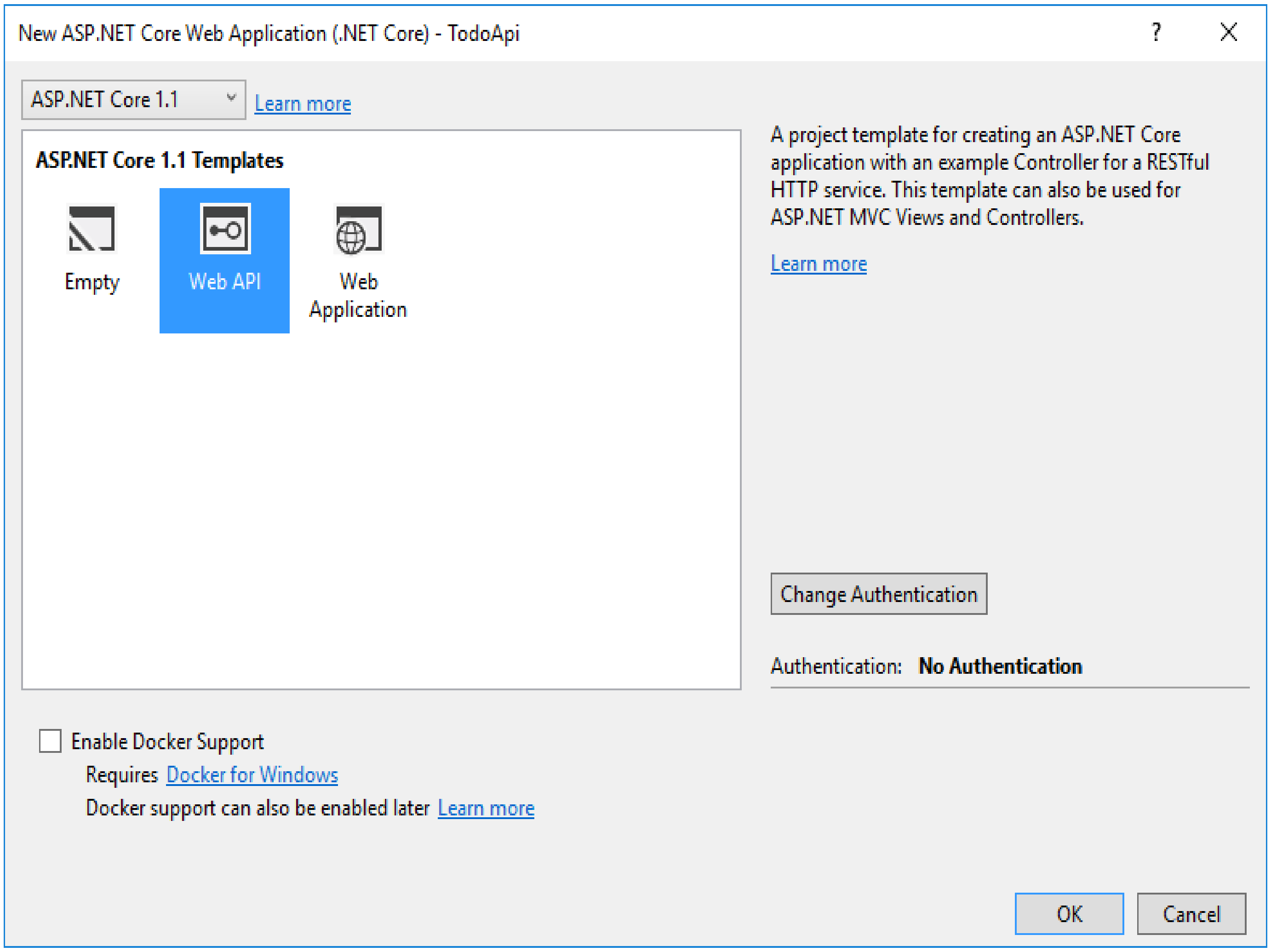
Type: Visual C#

Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core.

Name: TodoApi
Location: c:\Projects
Solution name: TodoApi

[✓] Create directory for solution
[ ] Create new Git repository

In the New ASP.NET Core Web Application (.NET Core) - TodoApi dialog, select the Web API template. Select OK. Do not select Enable Docker Support.


New ASP.NET Core Web Application (.NET Core) - TodoApi

ASP.NET Core 1.1 Learn more

ASP.NET Core 1.1 Templates

A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET MVC Views and Controllers.

Learn more

Authentication: **No Authentication**

☐ Enable Docker Support

Requires Docker for Windows

Docker support can also be enabled later Learn more

Launch the app

In Visual Studio, press CTRL+F5 to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port/api/values`, where *port* is a randomly chosen port number. If you're using Chrome, Edge or Firefox, the `ValuesController` data will be displayed:

Copy

```
["value1","value2"]
```

If you're using IE, you are prompted to open or save the *values.json* file.

Add support for Entity Framework Core

Install the Entity Framework Core InMemory database provider. This database provider allows Entity Framework Core to be used with an in-memory database.

Edit the *TodoApi.csproj* file. In Solution Explorer, right-click the project. Select Edit TodoApi.csproj. In the `ItemGroup` element, add "Microsoft.EntityFrameworkCore.InMemory":

XMLCopy

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore" Version="2.0.0"
/>
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
    <PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.InMemory" Version="1.1.1" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
Version="1.0.0" />
  </ItemGroup>

</Project>
```

Add a model class

A model is an object that represents the data in your application. In this case, the only model is a to-do item.

Add a folder named "Models". In Solution Explorer, right-click the project. Select Add > New Folder. Name the folder *Models*.

Note: You can put model classes anywhere in your project, but the *Models* folder is used by convention.

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

Replace the generated code with:
C#Copy

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

The database generates the `Id` when a `TodoItem` is created.

Create the database context

The *database context* is the main class that coordinates Entity Framework functionality for a given data model. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

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

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }
        public DbSet<TodoItem> TodoItems { get; set; }

    }
}
```

Register the database context

In order to inject the database context into the controller, we need to register it with the dependency injection container. Register the database context with the service container using the built-in support for dependency injection. Replace the contents of the *Startup.cs* file with the following:6

C#Copy

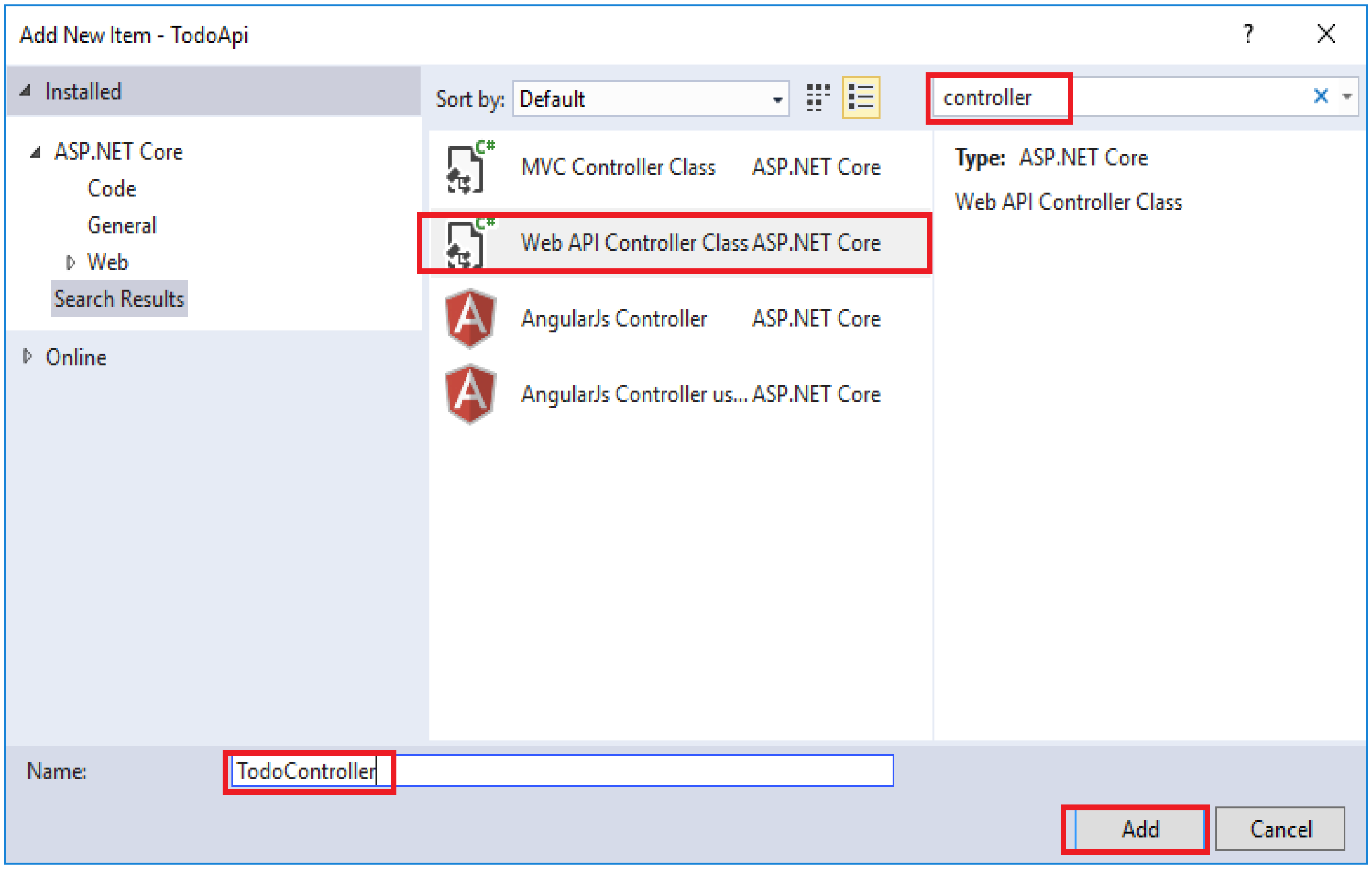
```
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase());
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseMvc();
        }
    }
}
```

The preceding code:

- Removes the code we're not using.
- Specifies an in-memory database is injected into the service container.

Add a controller

In Solution Explorer, right-click the *Controllers* folder. Select Add > New Item. In the Add New Item dialog, select the Web API Controller Class template. Name the class `TodoController`.



2

Replace the generated code with the following:

C#Copy

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using TodoApi.Models;
using System.Linq;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

The preceding code:

- Defines an empty controller class. In the next sections, we'll add methods to implement the API.
- The constructor uses Dependency Injection to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.
- The constructor adds an item to the in-memory database if one doesn't exist.

Getting to-do items

To get to-do items, add the following methods to the `TodoController` class.5

C#Copy

```
[HttpGet]
public IEnumerable<TodoItem> GetAll()
{
    return _context.TodoItems.ToList();
}

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
```

These methods implement the two GET methods:

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

Here is an example HTTP response for the `GetAll` method:

Copy

```
HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Server: Microsoft-IIS/10.0
   Date: Thu, 18 Jun 2015 20:51:10 GMT
   Content-Length: 82

   [{"Key":"1", "Name":"Item1","IsComplete":false}]
```

Later in the tutorial I'll show how you can view the HTTP response using [Postman]() or [curl]().

Routing and URL paths

The `[HttpGet]` attribute specifies an HTTP GET method. The URL path for each method is constructed as follows:

- Take the template string in the controller's route attribute:

C#Copy

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;
```

- Replace "[Controller]" with the name of the controller, which is the controller class name minus the "Controller" suffix. For this sample, the controller class name is TodoController and the root name is "todo". ASP.NET Core routing is not case sensitive.
- If the `[HttpGet]` attribute has a route template (such as `[HttpGet("/products")]`, append that to the path. This sample doesn't use a template. See Attribute routing with Http[Verb] attributes for more information.

In the `GetById` method:

C#Copy

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
```

`"{id}"` is a placeholder variable for the ID of the `todo` item. When `GetById` is invoked, it assigns the value of "{id}" in the URL to the method's `id` parameter.

`Name = "GetTodo"` creates a named route and allows you to link to this route in an HTTP Response. I'll explain it with an example later. See Routing to Controller Actions for detailed information.

Return values

The `GetAll` method returns an `IEnumerable`. MVC automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this method is 200, assuming there are no unhandled exceptions. (Unhandled exceptions are translated into 5xx errors.)

In contrast, the `GetById` method returns the more general `IActionResult` type, which represents a wide range of return types. `GetById` has two different return types:

- If no item matches the requested ID, the method returns a 404 error. This is done by returning `NotFound`.
- Otherwise, the method returns 200 with a JSON response body. This is done by returning an `ObjectResult`

Launch the app

In Visual Studio, press CTRL+F5 to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port/api/values`, where *port* is a randomly chosen port number. If you're using Chrome, Edge or Firefox, the data will be displayed. If you're using IE, IE will prompt to you open or save the *values.json* file. Navigate to the `Todo` controller we just created `http://localhost:port/api/todo`.

Implement the other CRUD operations

We'll add `Create`, `Update`, and `Delete` methods to the controller. These are variations on a theme, so I'll just show the code and highlight the main differences. Build the project after adding or changing code.

Create

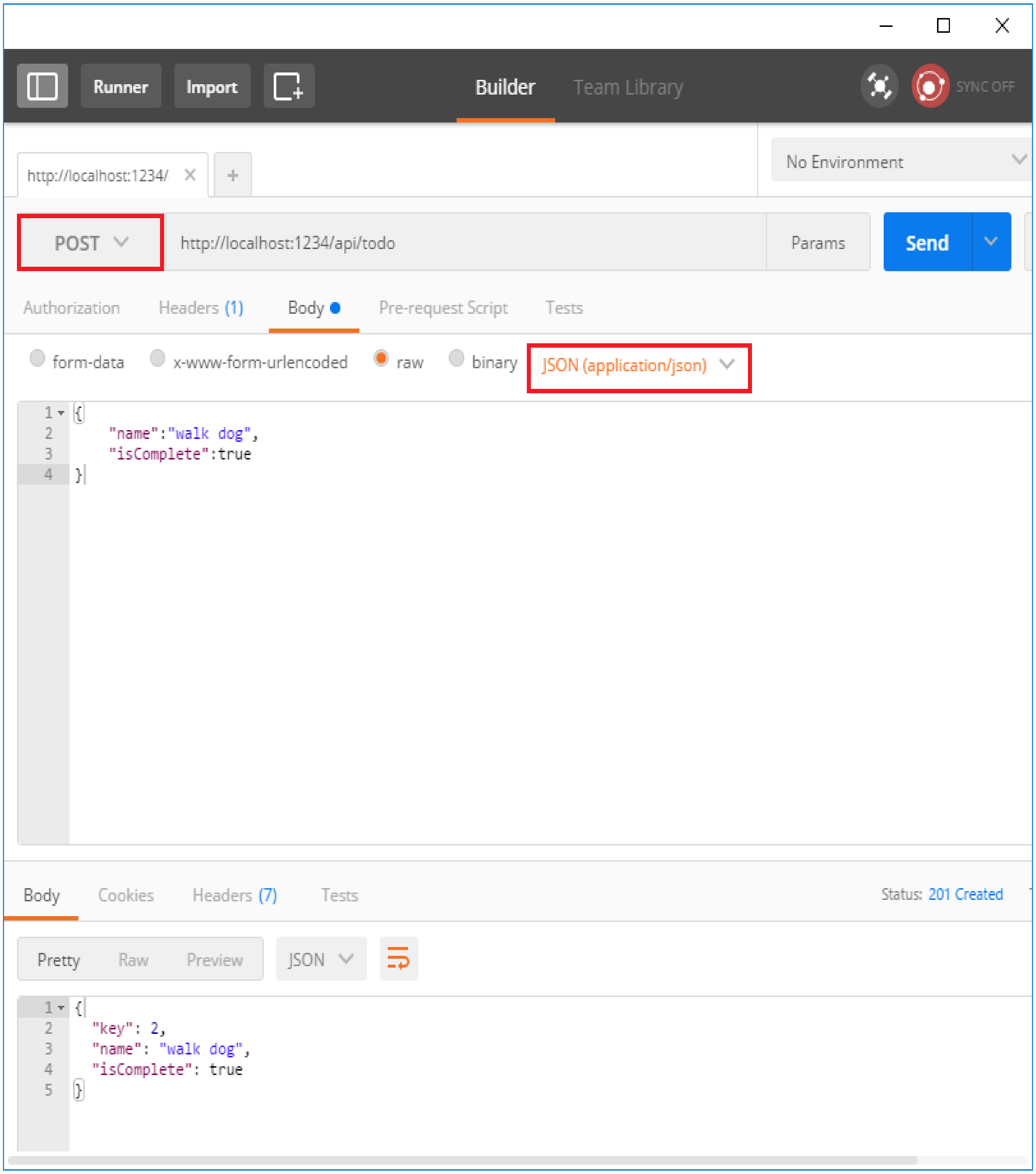
C#Copy

```
[HttpPost]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

This is an HTTP POST method, indicated by the `[HttpPost]` attribute. The `[FromBody]` attribute tells MVC to get the value of the to-do item from the body of the HTTP request.

The `CreatedAtRoute` method returns a 201 response, which is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. See 10.2.2 201 Created.

Use Postman to send a Create request



POST http://localhost:1234/api/todo

Body: raw, JSON (application/json)

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

Status: 201 Created

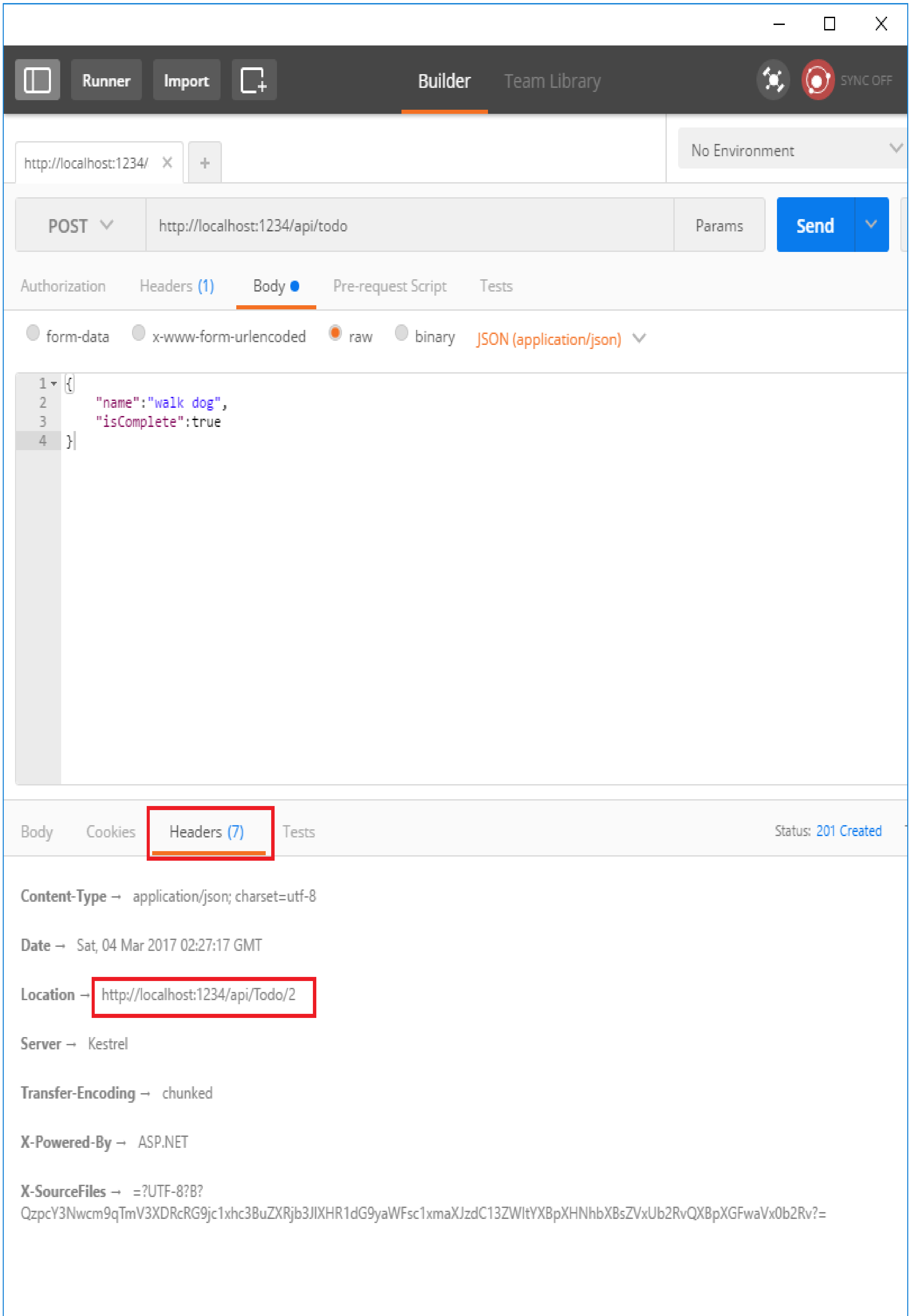
```
{
  "key": 2,
  "name": "walk dog",
  "isComplete": true
}
```

- Set the HTTP method to `POST`
- Select the Body radio button
- Select the raw radio button
- Set the type to JSON
- In the key-value editor, enter a Todo item such as

JSONCopy

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

- Select Send
- Select the Headers tab in the lower pane and copy the Location header:



http://localhost:1234/

POST http://localhost:1234/api/todo

JSON (application/json)

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

Status: 201 Created

Content-Type → application/json; charset=utf-8

Date → Sat, 04 Mar 2017 02:27:17 GMT

Location → http://localhost:1234/api/Todo/2

Server → Kestrel

Transfer-Encoding → chunked

X-Powered-By → ASP.NET

X-SourceFiles → =?UTF-8?B?QzpcY3Nwcm9qTmV3XDRcRG9jc1xhc3BuZXRjb3JlXHR1dG9yaWFsc1xmaXJzdC13ZWItYXBpXHNhbXBsZVxUb2RvQXBpXGFwaVx0b2Rv?=

You can use the Location header URI to access the resource you just created. Recall the `GetById` method created the `"GetTodo"` named route:

C#Copy

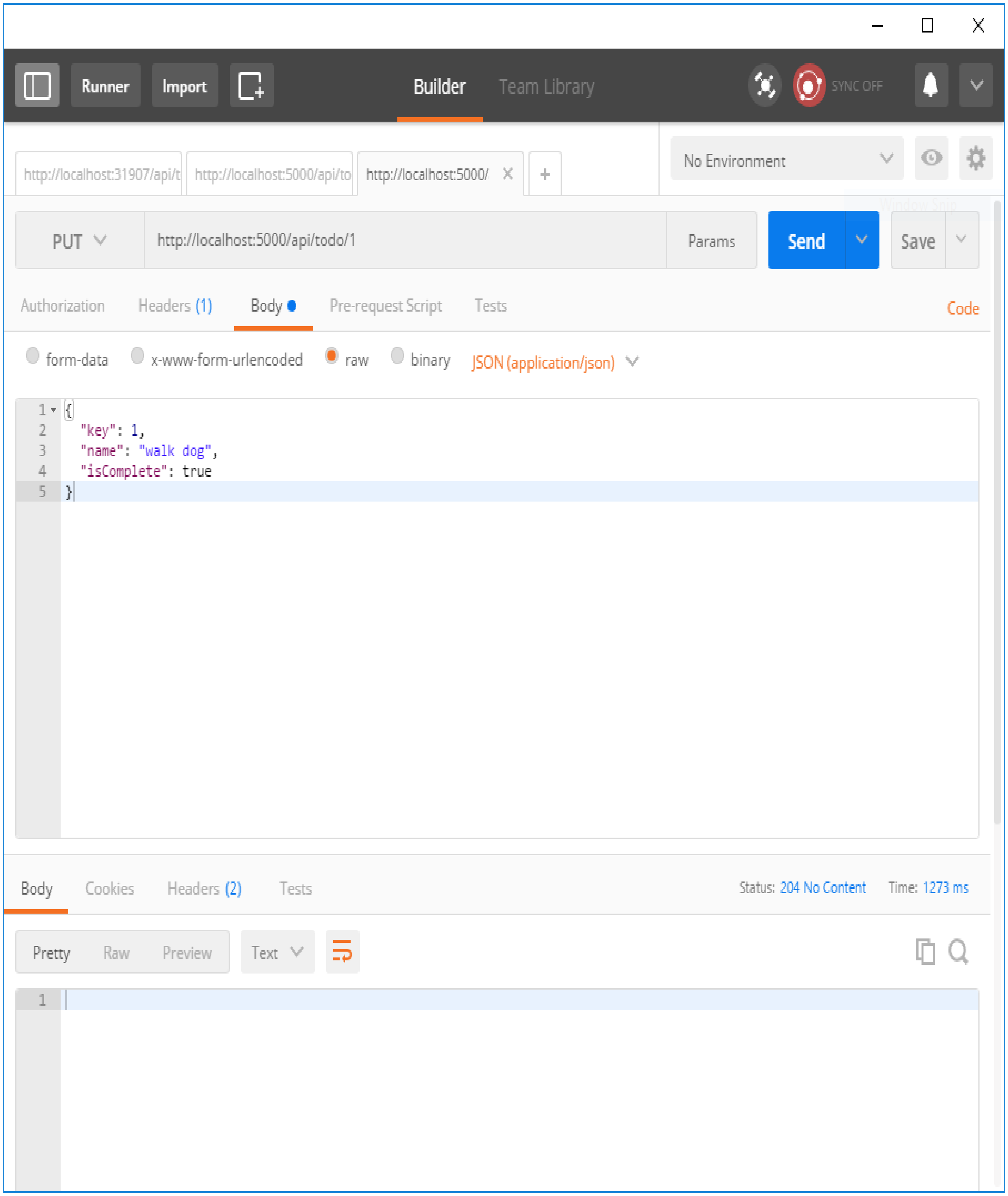
```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
```

Update

C#Copy

```
[HttpPut("{id}")]
public IActionResult Update(long id, [FromBody] TodoItem item)
{
    if (item == null || item.Id != id)
    {
        return BadRequest();
    }

    var todo = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    todo.IsComplete = item.IsComplete;
    todo.Name = item.Name;

    _context.TodoItems.Update(todo);
    _context.SaveChanges();
    return new NoContentResult();
}
```

`Update` is similar to `Create`, but uses HTTP PUT. The response is 204 (No Content). According to the HTTP spec, a PUT request requires the client to send the entire updated entity, not just the deltas. To support partial updates, use HTTP PATCH.



PUT http://localhost:5000/api/todo/1

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

Status: 204 No Content Time: 1273 ms

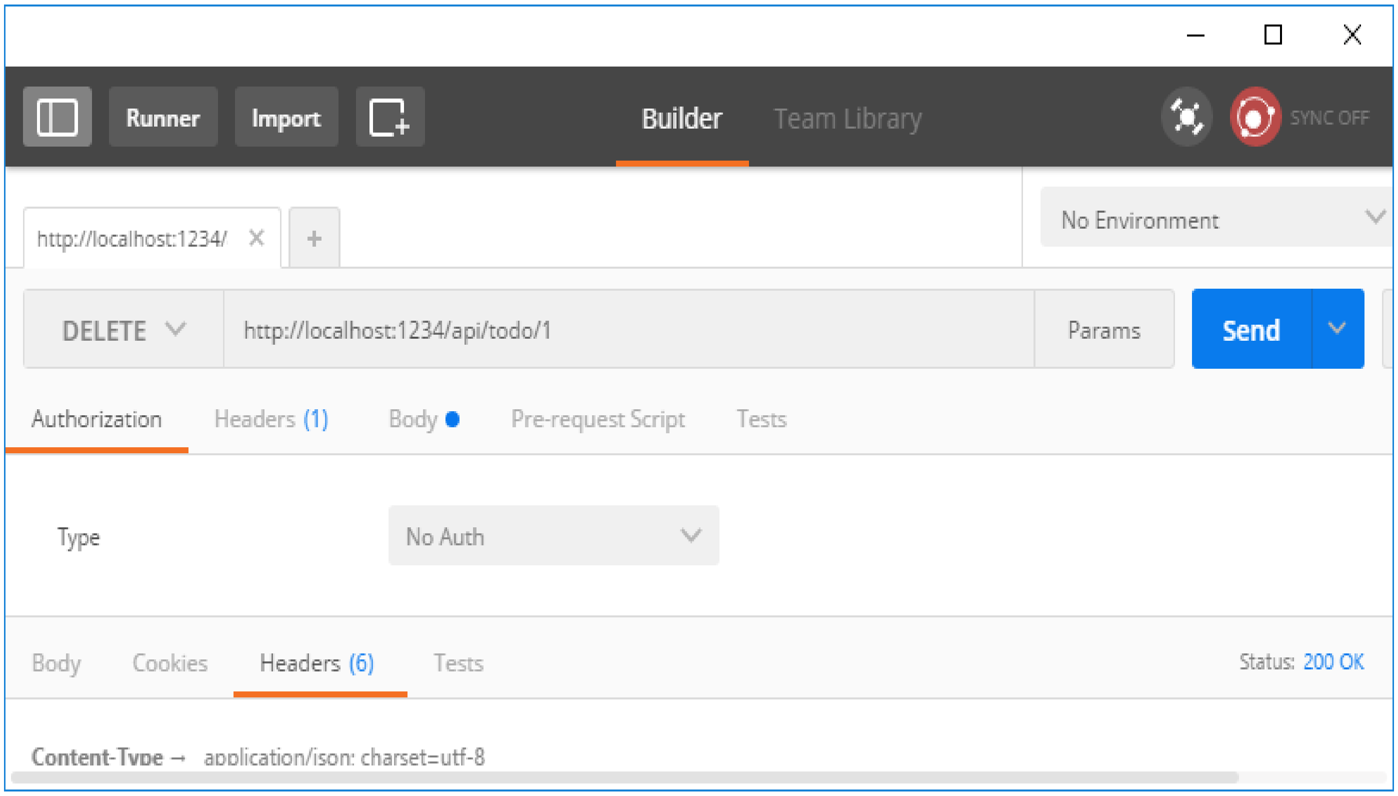
5

Delete

C#Copy

```
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todo);
    _context.SaveChanges();
    return new NoContentResult();
}
```

The response is 204 (No Content).



DELETE http://localhost:1234/api/todo/1

Type: No Auth

Status: 200 OK

Content-Type → application/json; charset=utf-8

Next steps

- ASP.NET Web API Help Pages using Swagger
- Routing to Controller Actions
- For information about deploying your API, see Publishing and Deployment.
- View or download sample code. See how to download.
- Postman