


Create a Web API app on Mac or Linux with Visual Studio Code

2017-3-14 9 min to read Contributors 

In this article

[Overview](#)

[Set up your development environment](#)

[Create the project](#)

[Add support for Entity Framework Core](#)

[Add a model class](#)

[Create the database context](#)

[Add a repository class](#)

[Register the repository and EF in-memory database](#)

[Add a controller](#)

[Getting to-do items](#)

[Implement the other CRUD operations](#)

[Visual Studio Code help](#)

[Next steps](#)

By [Mike Wasson](#) and [Rick Anderson](#)

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

ASP.NET Core has built-in support for MVC building Web APIs.

See [Build a web API with ASP.NET Core MVC and Visual Studio](#) for a version of this tutorial that uses Visual Studio.

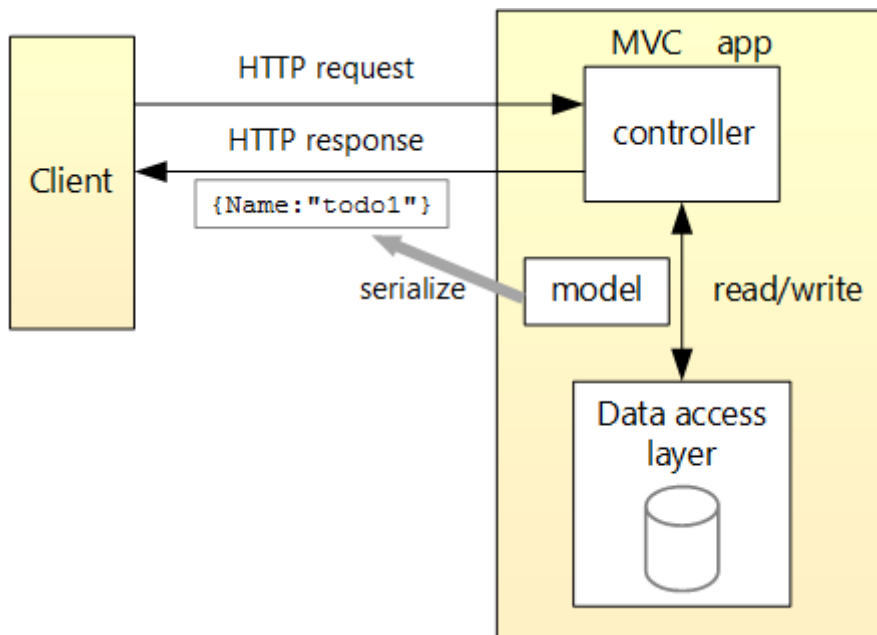
Overview

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

API	Description	Request body	Response body
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 (browser, mobile app, and so forth). We aren't writing a client in this tutorial. We'll use [Postman](#) 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 simple C# classes (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.

Set up your development environment

Download and install:

- [.NET Core](#)
- [VS Code](#)
- VS Code [C# extension](#)

Create the project

From a console, run the following commands:

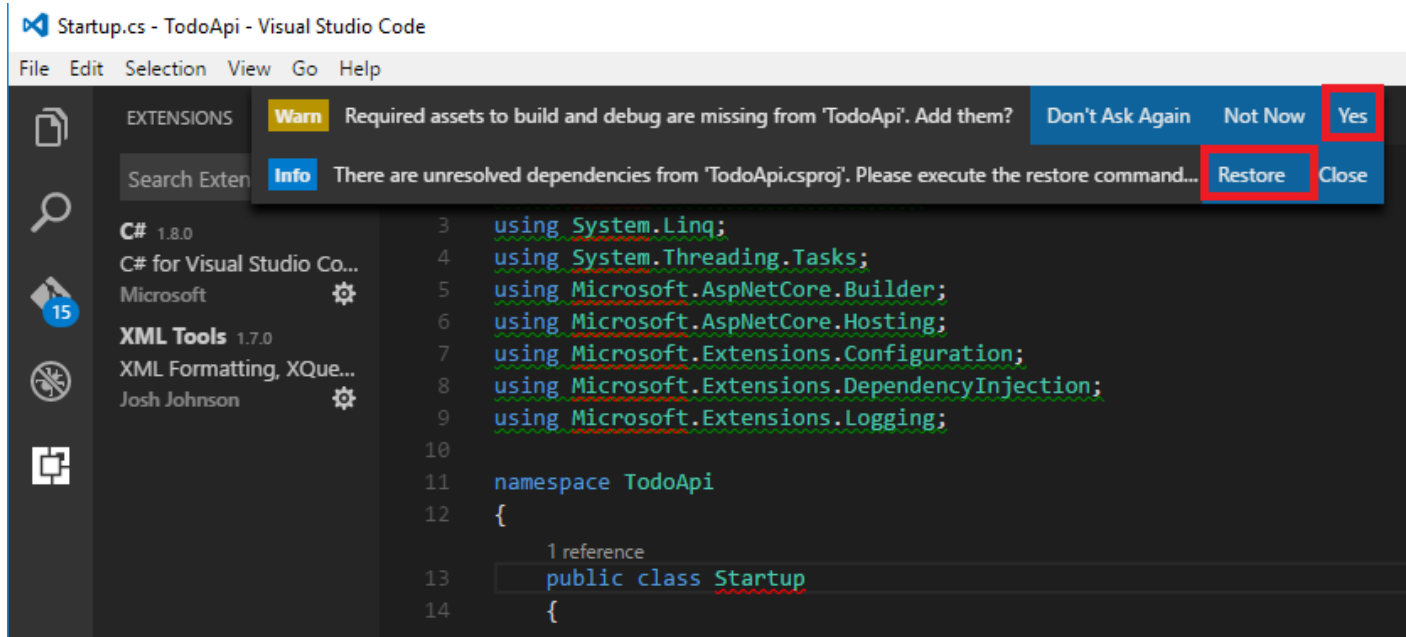
console



```
mkdir TodoApi
cd TodoApi
dotnet new webapi
```

Open the *TodoApi* folder in Visual Studio Code (VS Code) and select the *Startup.cs* file.

- Select **Yes** to the **Warn** message "Required assets to build and debug are missing from 'TodoApi'. Add them?"
- Select **Restore** to the **Info** message "There are unresolved dependencies from 'TodoApi.csproj'. Please execute the restore command..."



Press **Debug** (F5) to build and run the program. In a browser navigate to <http://localhost:5000/api/values>. The following is displayed:

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

See [Visual Studio Code help](#) for tips on using VS Code.

Add support for Entity Framework Core

Edit the *TodoApi.csproj* file to install the [Entity Framework Core InMemory](#) database provider. This database provider allows Entity Framework Core to be used with an in-memory database.

XML



```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
  <ItemGroup>
    <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>
</Project>
```

Run `dotnet restore` to download and install the EF Core InMemory DB provider. You can run `dotnet restore` from the terminal or enter `⌘⇧P` (macOS) or `Ctrl+Shift+P` (Linux) in VS Code and then type **.NET**. Select **.NET: Restore Packages**.

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

Add a `TodoItem` class with the following code:

C#



```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace TodoApi.Models
{
    ...
}
```

```
public class TodoItem
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public long Key { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

- The `[Key]` data annotation denotes the property, `Key`, is a unique identifier.
- `[DatabaseGenerated]` specifies the database will generate the key (rather than the application).
- `DatabaseGeneratedOption.Identity` specifies the database should generate integer keys when a row is inserted.

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 in the *Models* folder:

C#



```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

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

Add a repository class

A *repository* is an object that encapsulates the data layer. The *repository* contains logic for retrieving and mapping data to an entity model. Create the repository code in the *Models* folder.

Defining a repository interface named `ITodoRepository`:

C#



Copy

```
using System.Collections.Generic;

namespace TodoApi.Models
{
    public interface ITodoRepository
    {
        void Add(TodoItem item);

        IEnumerable<TodoItem> GetAll();
        TodoItem Find(long key);
        void Remove(long key);
        void Update(TodoItem item);
    }
}
```

This interface defines basic CRUD operations.

Add a `TodoRepository` class that implements `ITodoRepository`:

C#



Copy

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace TodoApi.Models
{
    public class TodoRepository : ITodoRepository
    {
        private readonly TodoContext _context;

        public TodoRepository(TodoContext context)
        {
            _context = context;

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

        public IEnumerable<TodoItem> GetAll()
        {

```

```
{
    return _context.TODOItems.ToList();
}

public void Add(TodoItem item)
{
    _context.TODOItems.Add(item);
    _context.SaveChanges();
}

public TodoItem Find(long key)
{
    return _context.TODOItems.FirstOrDefault(t => t.Key == key);
}

public void Remove(long key)
{
    var entity = _context.TODOItems.First(t => t.Key == key);
    _context.TODOItems.Remove(entity);
    _context.SaveChanges();
}

public void Update(TodoItem item)
{
    _context.TODOItems.Update(item);
    _context.SaveChanges();
}
}
```

Press **Debug** (F5) to build the app to verify you don't have any compiler errors.

Register the repository and EF in-memory database

By defining a repository interface, we can decouple the repository class from the MVC controller that uses it. Instead of instantiating a `TodoRepository` inside the controller we will inject an `ITodoRepository` using the built-in support in ASP.NET Core for [dependency injection](#).

This approach makes it easier to unit test your controllers. Unit tests should inject a mock or stub version of `ITodoRepository`. That way, the test narrowly targets the controller logic and not the data access layer.

In order to inject the repository into the controller, we need to register it with the DI container. Open the *Startup.cs* file. The code below also registers the in-memory database.

In the `ConfigureServices` method, add the highlighted code:

C#

```
// requires using TodoApi.Models; and
```



Copy

```
// using Microsoft.EntityFrameworkCore;
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase());

    services.AddMvc();

    services.AddScoped<ITodoRepository, TodoRepository>();
    //services.AddSingleton<ITodoRepository, TodoRepository>();
}
```

Add a controller

In the *Controllers* folder, create a class named `TodoController`. Add the following (and add closing braces):

C#



Copy

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

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

        public TodoController(ITodoRepository todoRepository)
        {
            _todoRepository = todoRepository;
        }
    }
}
```

This defines an empty controller class. In the next sections, we'll add methods to implement the API.

Getting to-do items

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

C#



Copy

```
[HttpGet]
public IEnumerable<TodoItem> GetAll()
```



```
{  
    return _todoRepository.GetAll();  
}  
  
[HttpGet("{id}", Name = "GetTodo")]  
public IActionResult GetById(long id)  
{  
    var item = _todoRepository.Find(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:

Code



```
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": "4f67d7c5-a2a9-4aae-b030-16003dd829ae", "Name": "Item1", "IsComplete": false}]
```

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

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, `[Route("api/[controller]")]`
- 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#



```
[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 VS Code, press F5 to launch the app. Navigate to <http://localhost:5000/api/todo> (The `Todo` controller we just created).

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#



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

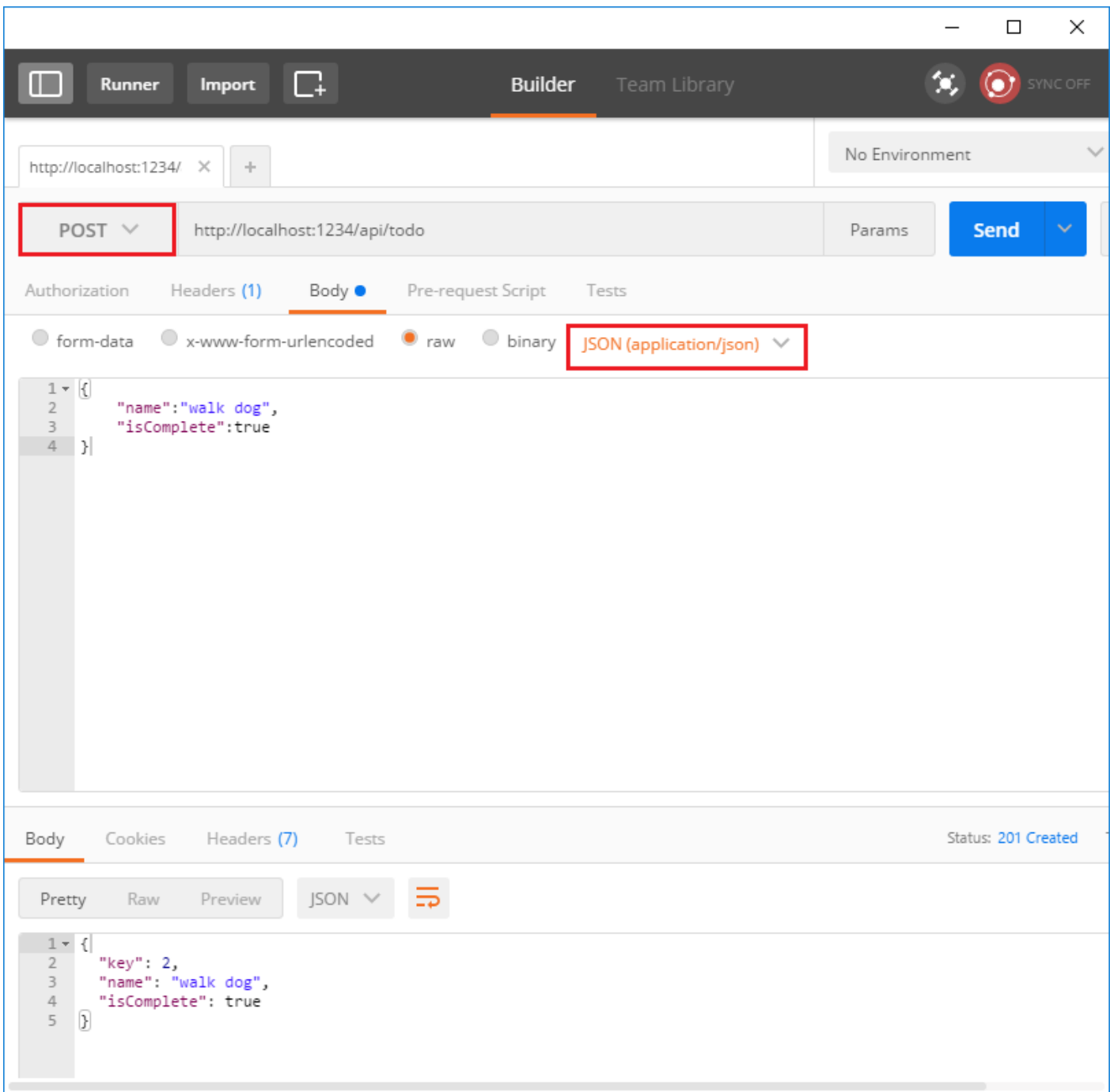
    _todoRepository.Add(item);

    return CreatedAtRoute("GetTodo", new { id = item.Key }, 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



- 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

JSON	 Copy
<pre>{ "name": "walk dog", "isComplete": true }</pre>	

- Select **Send**

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

The screenshot shows the Visual Studio Code REST Client interface. The top bar includes tabs for Runner, Import, Builder, and Team Library. The main area displays a POST request to `http://localhost:1234/api/todo`. The request body is a JSON object: `{ "name": "walk dog", "isComplete": true }`. The Headers tab is selected, showing the following headers:

- 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?QzpcY3Nwcm9qTmV3XDRcRG9jc1xhc3BuZXRjb3JlXHR1dG9yaWFsc1xmaXJzdC13ZWltYXBpXHNhbXBsZVxUb2RvQXBpXGFwaVx0b2Rv?=<code>`

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

C#



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

Update

C#



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

    var todo = _todoRepository.Find(id);
    if (todo == null)
    {
        return NotFound();
    }

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

    _todoRepository.Update(todo);
    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.

The screenshot shows the Visual Studio Code REST Client interface. At the top, there are tabs for Runner, Import, and Builder. The Builder tab is active. Below the tabs, there is a URL bar showing `http://localhost:1234/` and a dropdown menu for the environment, currently set to "No Environment".

The main area is divided into two sections. The top section is for the request, with tabs for Authorization, Headers (1), Body, Pre-request Script, and Tests. The Body tab is selected, and the content type is set to `JSON (application/json)`. The request body is a JSON object:

```
{
  "key": 1,
  "name": "Item 1",
  "isComplete": true
}
```

The bottom section shows the response, with tabs for Body, Cookies, Headers (6), and Tests. The Body tab is selected, and the status is `200 OK`. The response body is a JSON array:

```
[
  {
    "key": 1,
    "name": "Item1",
    "isComplete": false
  },
  {
    "key": 2,
    "name": "walk dog",
    "isComplete": true
  }
]
```

Delete

C#



Copy

```
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _todoRepository.Find(id);
    if (todo == null)
    {
        return NotFound();
    }

    _todoRepository.Remove(id);
    return new NoContentResult();
}
```

The response is **204 (No Content)**.

