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

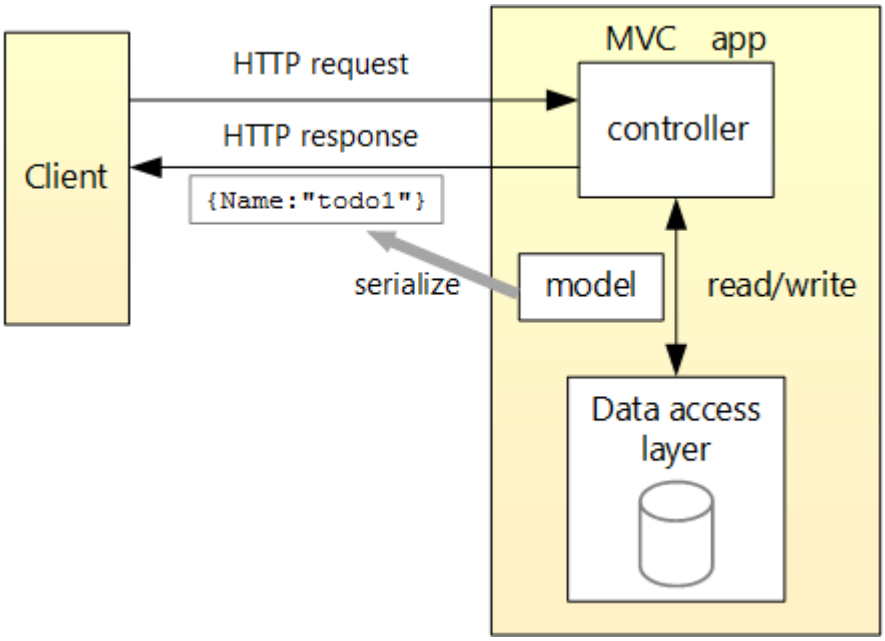
- This tutorial builds a web API for managing a list of "to-do" items. A user interface (UI) isn't created.

Overview

This tutorial creates the following API:

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.). This tutorial doesn't create a client. Postman or curl is used as the client to test the app.
- A *model* is an object that represents the data in the app. In this case, the only model is a to-do item. Models are represented as C# classes, also known as **P**lain **O**ld **C**# **O**bject (POCOs).
- A *controller* is an object that handles HTTP requests and creates the HTTP response. This app has a single controller.
- To keep the tutorial simple, the app doesn't use a persistent database. The sample app stores to-do items in an in-memory database.

Prerequisites

Visual Studio for Windows

- **ASP.NET and web development** workload
- **.NET Core cross-platform development** workload

Create the project

Follow these steps in Visual Studio:

- From the **File** menu, select **New** > **Project**.
- Select the **ASP.NET Core Web Application** template. Name the project *TodoApi* and click **OK**.
- In the **New ASP.NET Core Web Application - TodoApi** dialog, choose the ASP.NET Core version. Select the **API** template and click **OK**. Do **not** select **Enable Docker Support**.

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. Chrome, Microsoft Edge, and Firefox display the following output:

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

If using Internet Explorer, you'll be prompted to save a *values.json* file.

Add a model class

A model is an object representing the data in the app. In this case, the only model is a to-do item.

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

Note

The model classes can go anywhere in the project. The *Models* folder is used by convention for model classes.

In Solution Explorer, right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoItem* and click **Add**.

Update the `TodoItem` class with the following code:

```
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. This class is created by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

In Solution Explorer, right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.

Replace the class with the following code:

```
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 this step, the database context is registered with the dependency injection container. Services (such as the DB context) that are registered with the dependency injection (DI) container are available to the controllers.

Register the DB context with the service container using the built-in support for dependency injection. Replace the contents of the *Startup.cs* file with the following code:

```
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 =>
```

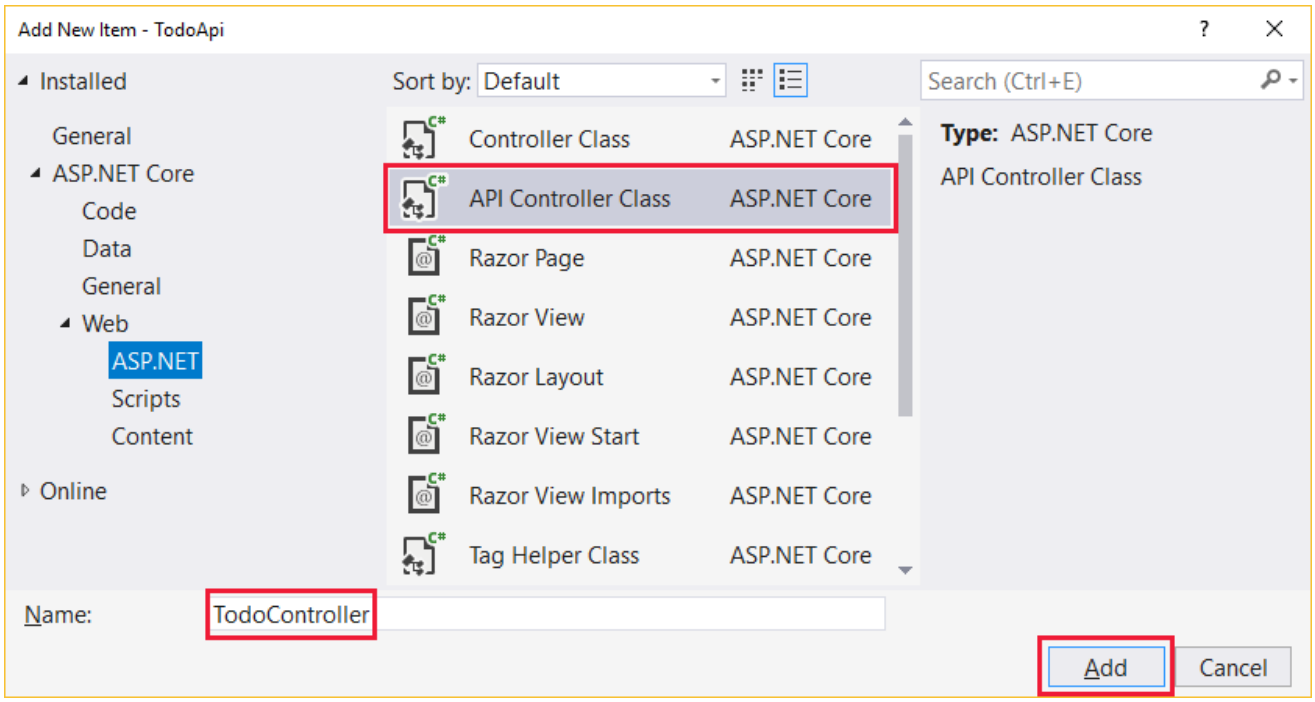
```
                opt.UseInMemoryDatabase("TodoList"));
            services.AddMvc();
        }

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

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Mvc;
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("TodoList"));
            services.AddMvc()
                    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
        }

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

The preceding code:

- Removes the unused code.
- 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 **API Controller Class** template. Name the class *TodoController*, and click **Add**.



Replace the class with the following code:

```
using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

using System.Linq;

using TodoApi.Models;


namespace TodoApi.Controllers

{

    [Route("api/[controller]")]

    public class TodoController : ControllerBase

    {

        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 API controller class without methods. In the next sections, methods are added to implement the API.

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

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoController : ControllerBase
    {
```

```
        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 API controller class without methods. In the next sections, methods are added to implement the API. The class is annotated with an `[ApiController]` attribute to enable some convenient features. For information on features enabled by the attribute, see Annotate class with ApiControllerAttribute.

The controller's 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.

Get to-do items

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

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

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TodoItems.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    return Ok(item);
}
[HttpGet]
public ActionResult<List<TodoItem>> GetAll()
{
    return _context.TodoItems.ToList();
}

[HttpGet("{id}", Name = "GetTodo")]
public ActionResult<TodoItem> GetById(long id)
{
    var item = _context.TodoItems.Find(id);
    if (item == null)
```

```
    {
        return NotFound();
    }
    return item;
}
```

These methods implement the two GET methods:

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

Here's a sample HTTP response for the `GetAll` method:

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

Later in the tutorial, I'll show how the HTTP response can be viewed with Postman or curl.

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:

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

```
namespace TodoApi.Controllers
```

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

namespace TodoApi.Controllers

{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoController : ControllerBase
    {
        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 **Todo**Controller and the root name is "todo". ASP.NET Core routing is case insensitive.
- If the `[HttpGet]` attribute has a route template (such as `[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 `GetById` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetById` is invoked, it assigns the value of `"{id}"` in the URL to the method's `id` parameter.

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TodoItems.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    return Ok(item);
}

 [HttpGet("{id}", Name = "GetTodo")]
public ActionResult<TodoItem> GetById(long id)
```

```
{
    var item = _context.TodoItems.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    return item;
}
```

`Name = "GetTodo"` creates a named route. Named routes:

- Enable the app to create an HTTP link using the route name.
- Are explained later in the tutorial.

Return values

The `GetAll` method returns a collection of `TodoItem` objects. 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. Returning NotFound returns an HTTP 404 response.
- Otherwise, the method returns 200 with a JSON response body. Returning Ok results in an HTTP 200 response.

In contrast, the `GetById` method returns the ActionResult<T> 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. Returning NotFound returns an HTTP 404 response.

- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

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. Navigate to the `Todo` controller at `http://localhost:<port>/api/todo`.

Implement the other CRUD operations

In the following sections, `Create`, `Update`, and `Delete` methods are added to the controller.

Create

Add the following `Create` method:

```
 [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);
}
```

The preceding code is an HTTP POST method, as 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.

```
 [HttpPost]
public IActionResult Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

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

The preceding code is an HTTP POST method, as indicated by the [HttpPost] attribute. MVC gets the value of the to-do item from the body of the HTTP request.

The `CreatedAtRoute` method:

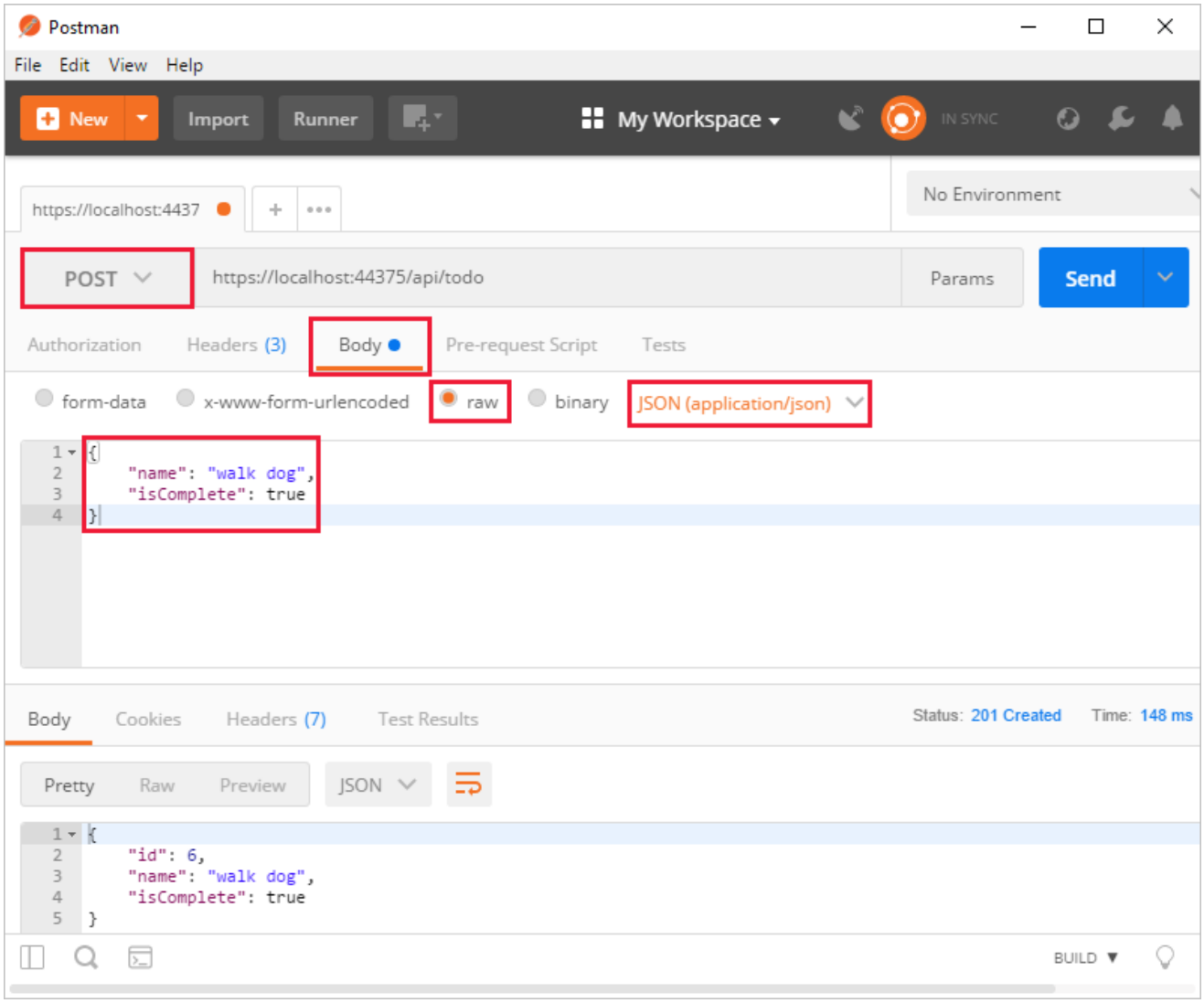
- Returns a 201 response. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. See 10.2.2 201 Created.
- Uses the "GetTodo" named route to create the URL. The "GetTodo" named route is defined in `GetById`:

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

```
{
    var item = _context.TodoItems.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    return Ok(item);
}
 [HttpGet("{id}", Name = "GetTodo")]
public ActionResult<TodoItem> GetById(long id)
{
    var item = _context.TodoItems.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    return item;
}
```

Use Postman to send a Create request

- Start the app.
- Open Postman.



- Update the port number in the localhost URL.
- Set the HTTP method to *POST*.
- Click the **Body** tab.
- Select the **raw** radio button.
- Set the type to *JSON (application/json)*.
- Enter a request body with a to-do item resembling the following JSON:

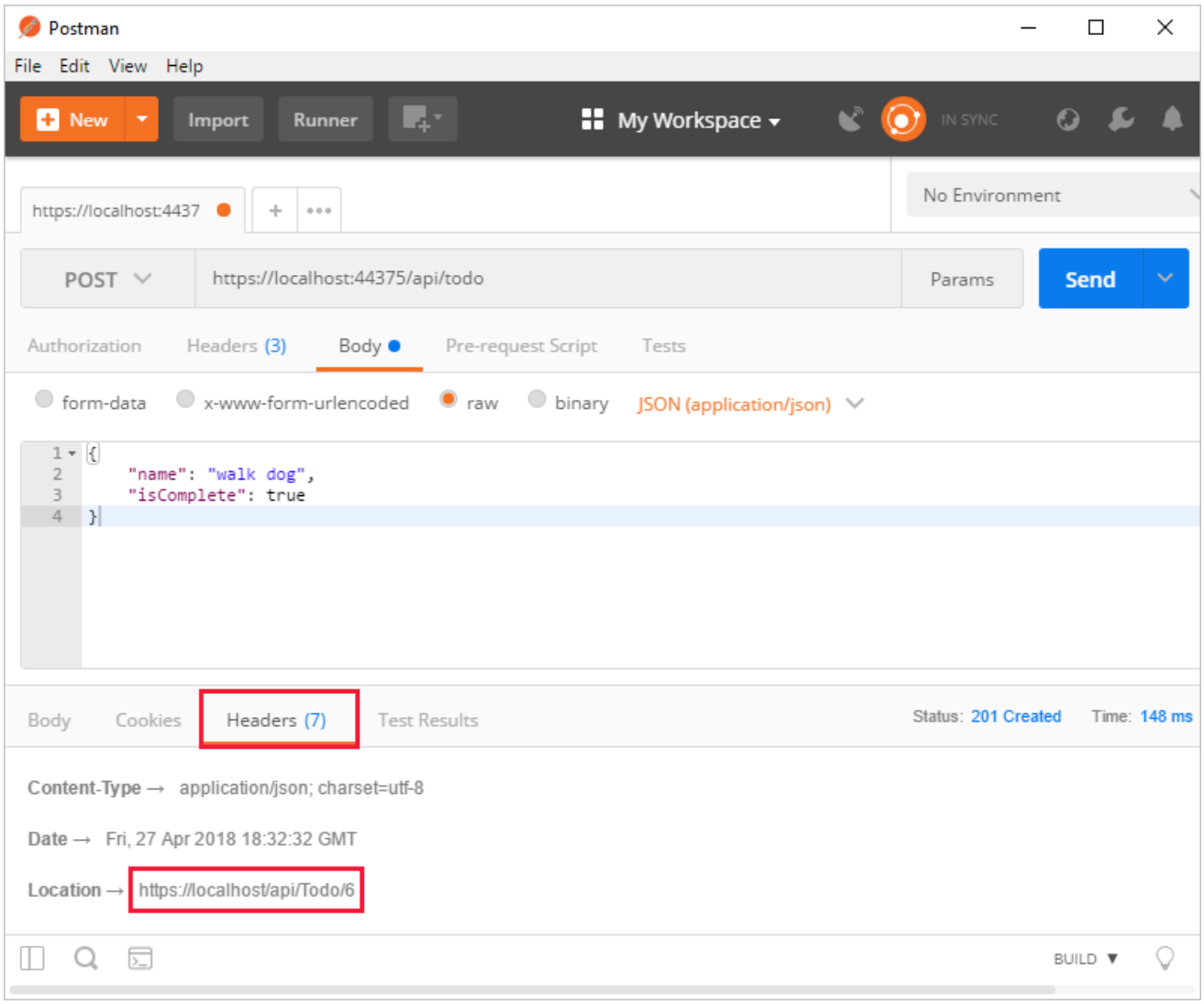
```
{
  "name":"walk dog",
  "isComplete":true
}
```

- Click the **Send** button.

  Tip

  If no response displays after clicking **Send**, disable the **SSL certification verification** option. This is found under **File** > **Settings**. Click the **Send** button again after disabling the setting.

  Click the **Headers** tab in the **Response** pane and copy the **Location** header value:



  The Location header URI can be used to access the new item.

  Update

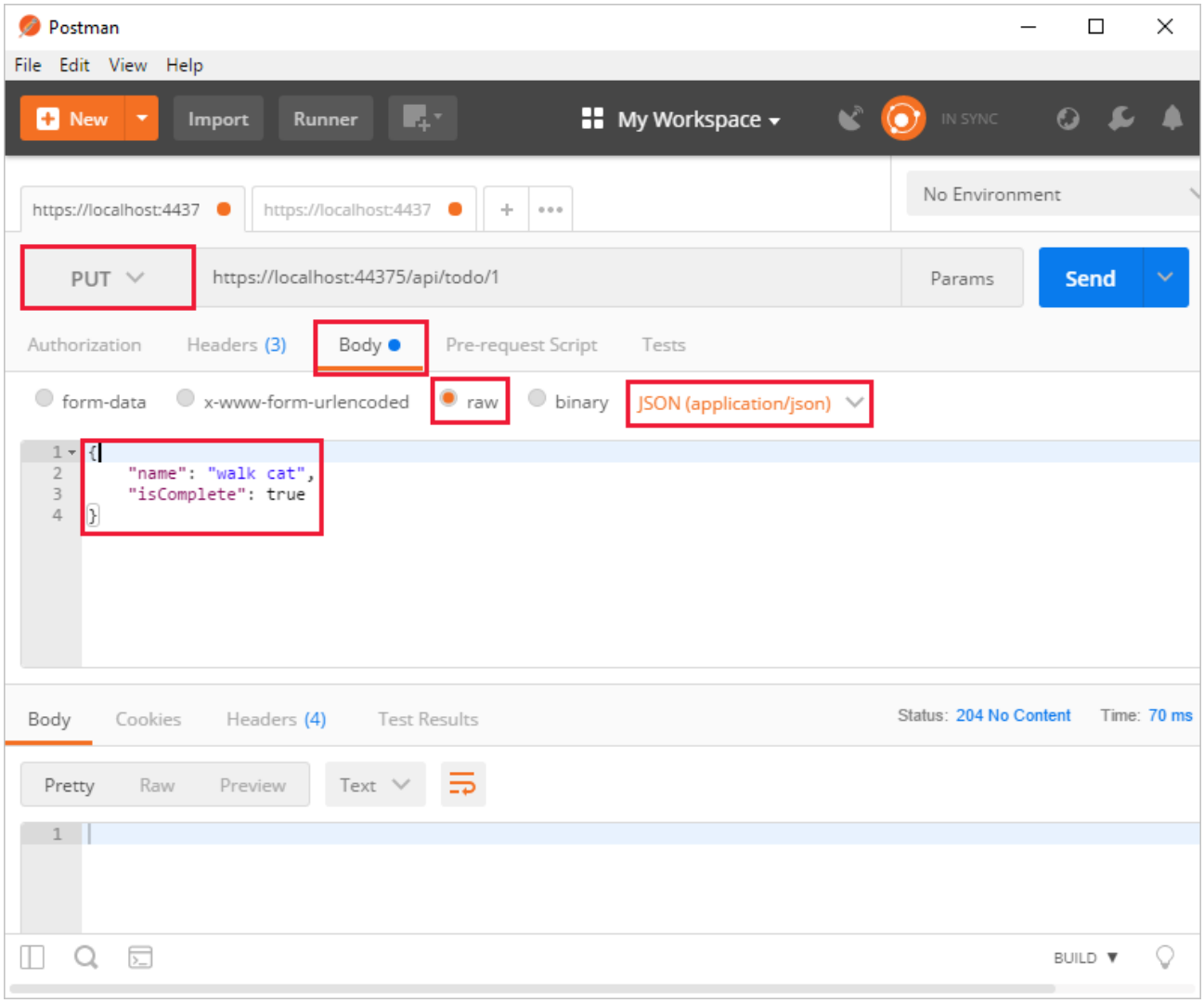
Add the following `Update` method:

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

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

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

    _context.TodoItems.Update(todo);
    _context.SaveChanges();
    return NoContent();
}
 [HttpPut("{id}")]
public IActionResult Update(long id, TodoItem item)
{
    var todo = _context.TodoItems.Find(id);
```

```
    if (todo == null)
    {
        return NotFound();
    }

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

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

`Update` is similar to `Create`, 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 deltas. To support partial updates, use HTTP PATCH.

Use Postman to update the to-do item's name to "walk cat":



Delete

Add the following `Delete` method:

```
 [HttpDelete("{id}")]

public IActionResult Delete(long id)

{

    var todo = _context.TodoItems.Find(id);

    if (todo == null)

    {

        return NotFound();
```

```
    }


    _context.TodoItems.Remove(todo);

    _context.SaveChanges();

    return NoContent();

}
```

The `Delete` response is 204 (No Content).

Use Postman to delete the to-do item: