# Building a CRUD API with the Dapper ORM in ASP.NET Core

December 15, 2021

ORM stands for Object Relational Mapper(ORM). It could also stand for Object Relational Mapping, which describes the method that ORMs apply.

An ORM is usually a library that makes interacting with our database easy. It acts as a spokesman between us and our database.

ORMs employ abstraction to 'mask' the unwanted details and focus on the essential and basic functionalities. After abstraction, our data is then broken into smaller parts referred to as models.

In this article, we'll learn how to build an API using Dapper to make queries.

At the end of this article, you should be able to:

- Know what ORMs are and how they work
- Create an ASP.NET Core API project with the dotnet CLI
- Able to make queries with the Dapper ORM
- Document an ASP.NET core API with Swagger.

## Prerequisites

- Basic knowledge of C#
- .NET Framework 5.0+ installed on your machine
- A code editor of your choice. I use Visual Studio Code
- Postman or any REST Client installed on your machine

## Table of Contents

# What is Dapper?

Dapper is an ORM for the .NET framework. It is lightweight, fast and since you get to write your SQL own queries, it provides a lot of flexibility. Dapper provides methods that make it easy to communicate with our database. Some of the methods are:

1. **Execute**: this method executes a command and returns the affected rows. It's usually used to perform INSERT, UPDATE, and DELETE operations.

2. **Query**: this method executes a query and maps the result. It is usually used to fetch multiple objects from the database.

3. **QueryFirst**: this method executes a query and maps the first result that matches the parameters in the query. This is used when we need just one item that matches the provided specifications.

4. **QueryFirstOrDefault**: this functions like `QueryFirst` but returns a default value if the sequence contains no elements.

5. **QuerySingle**: it executes a query and maps the result provided that there is only one item in the sequence. If there is not exactly one element in the sequence, it throws an exception when no element or more than one element is returned.

6. **QuerySingleOrDefault**: this method works like `QuerySingle` but returns a default value if no item is returned from the database.

For more information on Dapper ORM methods visit Dapper docs.

## Creating a new ASP.NET Core API project

After installing the Dotnet Framework SDK, go ahead to your console and type the command below to confirm that it is in fact, installed:

```
$ dotnet --version
5.0.301
```

If you have the framework installed, you should see the version of your SDK just below your command. If you don't, download it from here.

After making sure we have the SDK installed, we can create our project. Since we'll be building an API, we'll use the command for creating an API project. To do this, open a terminal or command prompt and navigate to the directory in which you'd like your project to be created. Now enter the command below:

```
$ dotnet new webapi -n TodoAPI
```

The `n` flag just tells dotnet what we want to call our app. This should create a new starter project for our API. Navigate into the project folder and type the following command:

It should start our server on the port shown. This is usually `5000`. When you open this project in your preferred editor, you'll see that a controller has been defined in `Controllers/WeatherForecastController.cs`.

To test this controller, open your REST client or browser and enter the link `http://localhost:5000/WeatherForecast`. You should see the data that was returned from that controller.

## Setting Up Our Database

The first thing we want to do here is add our connection string to our `appsettings.json` file. Like so:

```
"ConnectionStrings": {
   "SqlConnection": "your_connection_string"
 }
```

After that, we'll need to install a database client. For this article, we'll be using the SQL client. You can install that by doing:

```
  $ dotnet add package Microsoft.Data.SqlClient
```

The next thing is to handle migrations. This will require some external help as Dapper cannot do this for us. To create the necessary tables,

```
$ dotnet add package FluentMigrator
```

We also need FluentMigrator's runner to help run migrations. Similarly, run:

```
$ dotnet add package FluentMigrator.Runner
```

Now that we have FluentMigrator installed, we can set up our migrations. Create a `Migrations` folder at the root of your project and add the following files to it.

```
using FluentMigrator;

namespace TodoAPI.Migrations
{
    [Migration(202125100001)]
    public class Initial_202125100001 : Migration
    {
        // Drop the tables
        public override void Down()
        {
            Delete.Table("Todos");
            Delete.Table("Users");
        }

        // Create the tables
        public override void Up()
        {
```

```
                .WithColumn("Lastname").AsString(60).NotNullable()
                .WithColumn("Email").AsString(50).NotNullable();

            Create.Table("Todos")
                .WithColumn("Id").AsGuid().NotNullable().PrimaryKey()
                .WithColumn("Title").AsString(50).NotNullable()
                .WithColumn("Status").AsString(10).NotNullable()
                .WithColumn("Description").AsString().NotNullable()
                .WithColumn("UserId").AsGuid().NotNullable().ForeignKey
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using FluentMigrator;
using TodoAPI.Domain.Entities;

namespace TodoAPI.Migrations
{
    [Migration(202125100002)]
    public class Seed_202125100002 : Migration
    {
        public override void Down()
        {
            Delete.FromTable("Users");
            Delete.FromTable("Todos");
        }

        public override void Up()
        {
            List<Guid> ids = new List<Guid>{};
            List<String> names = new List<String>{"Mike", "Olumide", "P
            List<String> titles = new List<String>{"Title X", "Titte Y"
```

```
                String lastname = names[rnd.Next(names.Count)];
                String firstname = names[rnd.Next(names.Count)];
                Guid id = Guid.NewGuid();
                ids.Add(id);
                Insert.IntoTable("Users")
                    .Row(new User{
                        Firstname = firstname,
                        Lastname = lastname,
                        Email = String.Format("{0}{1}@email.co", firstn
                        Id = id
                    });

                for (int j = 0; j < 5; j++)
                {
                    Insert.IntoTable("Todos")
                        .Row(new TodoItem{
                            Title = titles[rnd.Next(titles.Count)],
                            Description = "Some pretty long string",
                            Status = (TodoStatus)rnd.Next(3),
                            UserId = id,
                            Id = Guid.NewGuid()
                        });
                }
            }
        }
    }
}
```

One migration creates the tables that we need, and the other provides us with seed data so that we have data to work within our database.

“But how do we run the migrations?", one might ask. Well, that’s where FluentMigrator’s runner comes in. So let’s create an extension that runs

Create a new folder `Extensions` and add the file below to it:

```
using System;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using FluentMigrator.Runner;

namespace TodoAPI.Extensions
{
    public static class MigrationManager
    {
        public static IHost MigrateDatabase(this IHost host)
        {
            using (var scope = host.Services.CreateScope())
            {
                var migrationService = scope.ServiceProvider.GetRequire
                try
                {
                    migrationService.ListMigrations();
                    migrationService.MigrateUp();
                }
                catch (Exception e)
                {
                    Console.WriteLine(e.Message);
                    throw;
                }
            }
            return host;
        }
    }
}
```

this extension is "used" on any instantiation of the `IHost` class. Learn more about extension methods here.

To make sure that this is called, head over to `Program.cs` and alter the code as shown below:

```
public static void Main(string[] args)
{
    CreateHostBuilder(args)
        .Build()
        .MigrateDatabase() // Add this line
        .Run();
}
```

As shown above, the migrations are run just as the app is being built and before it's run. There are a few things we need to add to our `Startup.cs` file as well.

```
public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();

        // Add this
        services.AddLogging(c => c.AddFluentMigratorConsole())
        .AddFluentMigratorCore()
        .ConfigureRunner(c => c.AddSqlServer2016()
          .WithGlobalConnectionString(Configuration.GetConnectionSt
          .ScanIn(Assembly.GetExecutingAssembly()).For.Migrations()
    }
```

above also adds configuration for our SQL server.

After doing this, start your app by running `dotnet run` and the migration should be run just before the app starts. If you encounter any errors, try to retrace your steps to find the root of the problem.

## Configuring Dapper

Next, let's configure up Dapper. To install dapper run the following command:

```
$ dotnet add package Dapper
```

After installing Dapper, create a folder `Data` and in it, add the file below:

```
using System.Data;
using Microsoft.Data.SqlClient;
using Microsoft.Extensions.Configuration;

namespace TodoAPI.Data
{
    public class DapperContext
    {
        private readonly IConfiguration _configuration;
        public DapperContext(IConfiguration configuration)
        {
            _configuration = configuration;
        }
        public IDbConnection CreateConnection()
            => new SqlConnection(_configuration.GetConnectionString("Sq
```

The class above is responsible for creating a connection to our database. Dapper is then used to communicate to our database using that connection. Don't forget to register this class as a service.

Head over to `Startup.cs` and add the line as shown below:

```
    public void ConfigureServices(IServiceCollection services)
        {
            services.AddSingleton<DapperContext>(); // Add this line to
            services.AddControllers();
        }
```

## Adding our App Logic(Models & Repositories)

First, create a new folder, `Domain`. In this new folder create three folders namely: `Entities`, `Repositories` and `DTOs`.

In the `Entities` folder is where we will define our models. Go ahead and add the following files:

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

namespace TodoAPI.Domain.Entities
{
    public class User
```

```
        [Required]
        public String Firstname { get; set; }

        [Required]
        public String Lastname { get; set; }

        [Required]
        public String Email { get; set; }
    }
}
```

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

namespace TodoAPI.Domain.Entities
{
    public class TodoItem
    {
        [Key]
        public Guid Id { get; set; }

        [Required]
        public Guid UserId { get; set; }

        [Required]
        public String Title { get; set; }

        [Required]
        public String Description { get; set; }

        public TodoStatus Status { get; set; } = TodoStatus.Todo;
    }
}
```

```
namespace TodoAPI.Domain.Entities
{
    public enum TodoStatus
    {
        Done,
        InProgress,
        Todo
    }
}
```

Next, add our Data Transfer Objects(DTOs). These will make exchanging data between requests, controllers and repositories easier and tidier.

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

using TodoAPI.Domain.Entities;

namespace TodoAPI.DTOs
{
    public class CreateTodoDTO
    {
        [Required]
        public Guid UserId { get; set; }

        [Required]
        public String Title { get; set; }

        [Required]
        public String Description { get; set; }
    }

    public class UpdateTodoDTO
    {
```

```
        [Required]
        public String Description { get; set; }

        public TodoStatus Status { get; set; }
    }
}
```

In the `Domain/Repositories/` folder, add the following files:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using TodoAPI.DTOs;
using TodoAPI.Domain.Entities;
using System;

namespace TodoAPI.Domain.Repositories
{
   public interface ITodoRepository
   {
       public Task Create(CreateTodoDTO createTodoDTO, Guid userId);

       public Task<IEnumerable<TodoItem>> GetAll();

       public Task<TodoItem> GetById(Guid id);

       public Task<IEnumerable<TodoItem>> GetByUser(Guid id);

       public Task Update(UpdateTodoDTO projectDTO, Guid id);

       public Task Delete(Guid id);
   }
}
```

```
using System.Threading.Tasks;
using TodoAPI.Domain.Entities;

namespace TodoAPI.Domain.Repositories
{
    public interface IUserRepository
    {
        public Task<IEnumerable<User>> GetAll();

        public Task<User> GetById(Guid id);
    }
}
```

These are the methods that will communicate directly with our databases with the help of Dapper. Our actual repository classes will inherit from these interfaces.

Now create a folder `Repositories` in your `Data` folder. In this folder, we will add our repositories. These classes will implement the interfaces we added earlier in the `Domain/Repositories/` folder. This ensures that we use the correct methods to communicate with our database. This folder will house the following files:

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using TodoAPI.DTOs;
using TodoAPI.Domain.Entities;
using TodoAPI.Domain.Repositories;
using System;
using Dapper;
using System.Data;
```

```
    public class UserRepository : IUserRepository
    {
        private readonly DapperContext _context;
        public UserRepository(DapperContext context)
        {
            _context = context;
        }

        public async Task<IEnumerable<User>> GetAll()
        {
            string sqlQuery = "SELECT * FROM Users";
            using(var connection = _context.CreateConnection())
            {
                var users = await connection.QueryAsync<User>(sqlQuery)
                return users.ToList();
            }
        }

        public async Task<User> GetById(Guid id)
        {
            string sqlQuery = "SELECT * FROM Users WHERE Id = @Id";
            using (var connection = _context.CreateConnection())
            {
                return await connection.QuerySingleAsync<User>(sqlQuery
            }
        }
    }
}
```

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using TodoAPI.DTOs;
using TodoAPI.Domain.Entities;
```

```
using System.Data;

namespace TodoAPI.Data.Repositories
{
    public class TodoRepository : ITodoRepository
    {
        private readonly DapperContext _context;
        public TodoRepository(DapperContext context)
        {
            _context = context;
        }

        public async Task Create(CreateTodoDTO createTodoDTO, Guid user
        {
            string sqlQuery = "INSERT into Todos (UserId, Title, Descri
            var parameters = new DynamicParameters();
            parameters.Add("Title", createTodoDTO.Title, DbType.String)
            parameters.Add("UserId", createTodoDTO.UserId, DbType.Guid)
            parameters.Add("Description", createTodoDTO.Description, Db
            parameters.Add("Status", TodoStatus.Todo, DbType.String);
            parameters.Add("Id", Guid.NewGuid(), DbType.Guid);
            Console.WriteLine(TodoStatus.Todo);
            using (var connection = _context.CreateConnection())
            {
                var r = await connection.ExecuteAsync(sqlQuery, paramet
                Console.Write(r);
            }
        }

        public async Task<IEnumerable<TodoItem>> GetAll()
        {
            string sqlQuery = "SELECT * FROM Todos";
            using (var connection = _context.CreateConnection())
            {
                var todos = await connection.QueryAsync<TodoItem>(sqlQu
                return todos.ToList();
            }
```

```
        {
            string sqlQuery = "SELECT * FROM Todos WHERE Id = @Id";
            using (var connection = _context.CreateConnection())
            {
                var todo = await connection.QuerySingleAsync<TodoItem>(
                return todo;
            }
        }

        public async Task<IEnumerable<TodoItem>> GetByUser(Guid id)
        {
            string sqlQuery = "SELECT * FROM Todos WHERE UserId = @User
            using (var connection = _context.CreateConnection())
            {
                IEnumerable<TodoItem> todos = await connection.QueryAsy
                return todos;
            }
        }

        public async Task Update(UpdateTodoDTO updateTodoDTO, Guid id)
        {
            string sqlQuery = "UPDATE Todos SET Title = @Title, Status
            var parameters = new DynamicParameters();
            parameters.Add("Title", updateTodoDTO.Title, DbType.String)
            parameters.Add("Status", updateTodoDTO.Status, DbType.Strin
            parameters.Add("Description", updateTodoDTO.Description, Db
            parameters.Add("Id", id, DbType.Guid);
            using (var connection = _context.CreateConnection())
            {
                await connection.ExecuteAsync(sqlQuery, parameters);
            }
        }

        public async Task Delete(Guid id)
        {
            string query = "DELETE FROM Todos WHERE Id = @Id";
```

```
            }

        }
    }
}
```

`Domain/Repositories/TodoRepository.cs` as shown above, has six and rightly so, given that the interface it implements has the same number of methods.

Taking the `Create` method above:

```
    public async Task Create(CreateTodoDTO createTodoDTO, Guid userId)
        {
            string sqlQuery = "INSERT into Todos (UserId, Title, Descri
            var parameters = new DynamicParameters();
            parameters.Add("Title", createTodoDTO.Title, DbType.String)
            parameters.Add("UserId", createTodoDTO.UserId, DbType.Guid)
            parameters.Add("Description", createTodoDTO.Description, Db
            parameters.Add("Status", TodoStatus.Todo, DbType.String);
            parameters.Add("Id", Guid.NewGuid(), DbType.Guid);
            using (var connection = _context.CreateConnection())
            {
                await connection.ExecuteAsync(sqlQuery, parameters);
            }
        }
```

The `GetAll` method works similarly to the `Create` method but uses the `connection.QueryAsync<TodoItem>()` method. This queries all the items and

`GetById` uses `connection.QuerySingleAsync<TodoItem>()` since the ID property is unique. As stated earlier in this article, it throws an error if more than one element is found.

`GetByUser` will fetch all the items with their `UserId` to be the same as the UserId in the query parameters. Like `GetAll` uses `connection.QueryAsync<TodoItem>()` as we want to fetch multiple items.

The `Update` and `Delete` methods work similarly to the `Create`. They also use the `connection.ExecuteAsync()` method as they do not need any data returned.

## Adding Our Controllers

In the `Controllers` folder, add the following files:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using TodoAPI.Data.Repositories;
using TodoAPI.Domain.Repositories;
using TodoAPI.DTOs;

namespace TodoAPI.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class TodosController : ControllerBase
```

```
        public TodosController(ILogger<TodosController> logger, ITodoRe
        {
            _logger = logger;
            _todosRepository = todosRepository;
        }

        [HttpGet]
        public async Task<IActionResult> GetAll()
        {
            try
            {
                 var Data = await _todosRepository.GetAll();
                return Ok(new {
                    Success = true,
                    Message = "All todo items returned.",
                    Data
                });
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                return StatusCode(500, ex.Message);
            }
        }

        [HttpGet]
        [Route("{todoId}")]
        public async Task<IActionResult> GetById(Guid todoId)
        {
            try
            {
                var todo = await _todosRepository.GetById(todoId);
                if(todo == null) return NotFound();
                return Ok(new {
                    success = true,
                    message = "One todo item returned.",
```

```
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                return StatusCode(500, ex.Message);
            }
        }

        [HttpGet]
        [Route("users/{userId}")]
        public async Task<IActionResult> GetByUserId(Guid userId)
        {
            try
            {
                var Data = await _todosRepository.GetByUser(userId);
                return Ok(new {
                    Success = true,
                    Message = "Todo items returned.",
                    Data
                });
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                return StatusCode(500, ex.Message);
            }
        }

        [HttpPost]
        public async Task<IActionResult> Create(CreateTodoDTO createTod
        {
            try
            {
                await _todosRepository.Create(createTodoDTO, userId);
                return Ok(new {
                    Success = true,
                    Message = "Todo item created."
                });
```

```
            Console.WriteLine(ex.Message);
            return StatusCode(500, ex.Message);
        }
    }

    [HttpPatch]
    [Route("{todoId}")]
    public async Task<IActionResult> Update(UpdateTodoDTO updateTod
    {
        try
        {
            await _todosRepository.Update(updateTodoDTO, todoId);
            return Ok(new {
                Success = true,
                Message = "Todo item updated."
            });
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
            return StatusCode(500, ex.Message);
        }
    }

    [HttpDelete]
    [Route("{todoId}")]
    public async Task<IActionResult> Delete(Guid todoId)
    {
        try
        {
            await _todosRepository.Delete(todoId);
            return Ok(new {
                Success = true,
                Message = "Todo deleted."
            });
        }
        catch (Exception ex)
```

```
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using TodoAPI.Data.Repositories;
using TodoAPI.Domain.Repositories;

namespace TodoAPI.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class UsersController : ControllerBase // 1
    {
        private readonly ILogger<UsersController> _logger;
        private readonly IUserRepository _userRepository;

        public UsersController(ILogger<UsersController> logger, IUserRe
        {
            _logger = logger;
            _userRepository = userRepository;
        }

        [HttpGet]
        public async Task<IActionResult> GetAll()
        {
            try
            {
```

```
                    Message = "all users returned.",
                    Data
                });
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                return StatusCode(500, ex.Message);
            }
        }

        [HttpGet] //3
        [Route("{userId}")] //4
        public async Task<IActionResult> GetById(Guid userId) //5
        {
            try
            {
                var Data = await _userRepository.GetById(userId); //6
                return Ok(new { //7
                    Success = true,
                    Message = "User fetched.",
                    Data
                });
            }
            catch (Exception ex) //8
            {
                Console.WriteLine(ex.Message);
                return StatusCode(500, ex.Message);
            }
        }
    }
}
```

exchange of data between the entities is made seamless.

Using the `UsersController` class above for reference, comment #1 is where our class is defined. It inherits from the `ControllerBase` class as seen above. ASP.NET is quite smart. It takes the letters before 'Controller' and maps the methods to their respective endpoints. In the case of `UsersController`, it maps the methods to the `users` route.

At #2, this is where our dependencies are injected. This is possible because the dependencies have earlier been registered as services.

Looking at the `GetById`, comment #3 denotes the method that this method accepts. It is a `GET` method.

#4 tells our method what route we want it to answer to. In some cases like this one, we could also pass data in the route as parameters. `userId` is the parameter we're expecting in this case.

By #5, the required data `userId` has been parsed from the route.

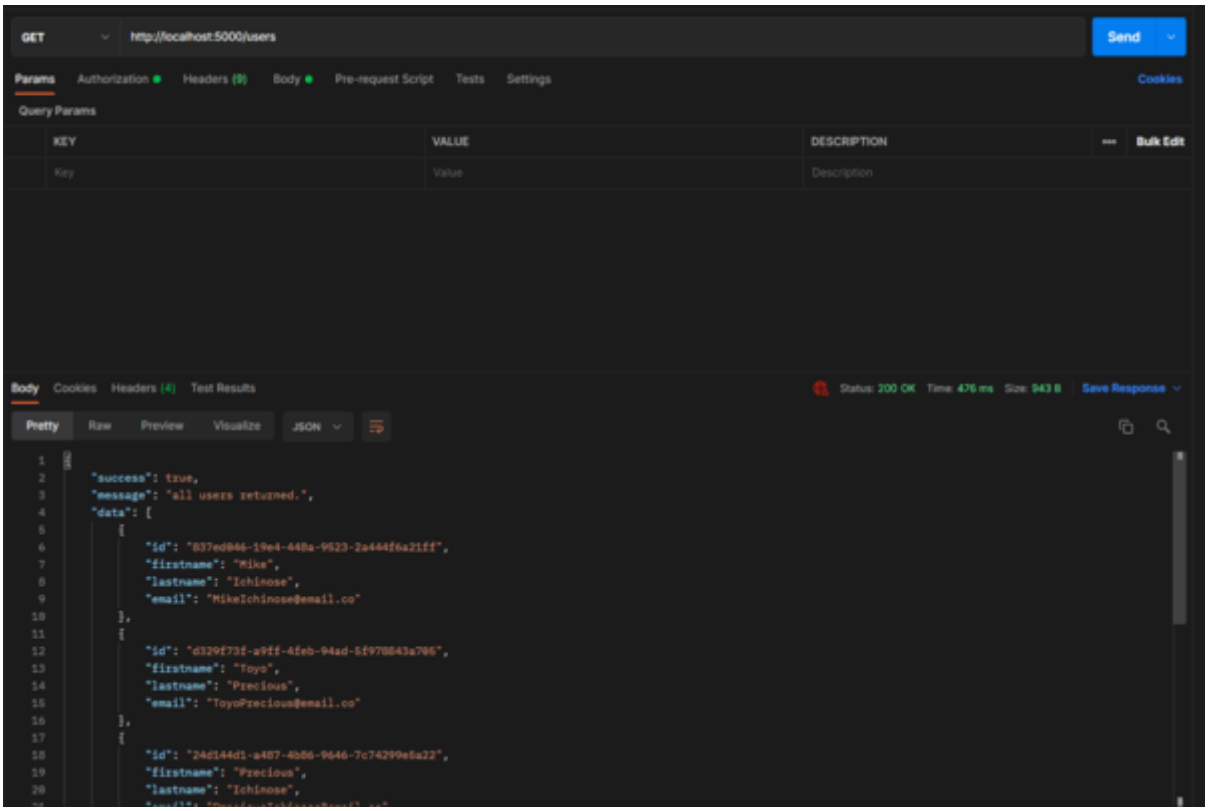
#6 is where we use `_userRepository`, which we injected earlier at #2, to communicate with the database.

#7 is where we return the data fetched as a response with a code of 200. It's a 200 response because it is wrapped with `Ok()`. Learn more about dotnet API responses here. If any error is encountered, it is caught at #8 and returned as an error message with a status code of 500.

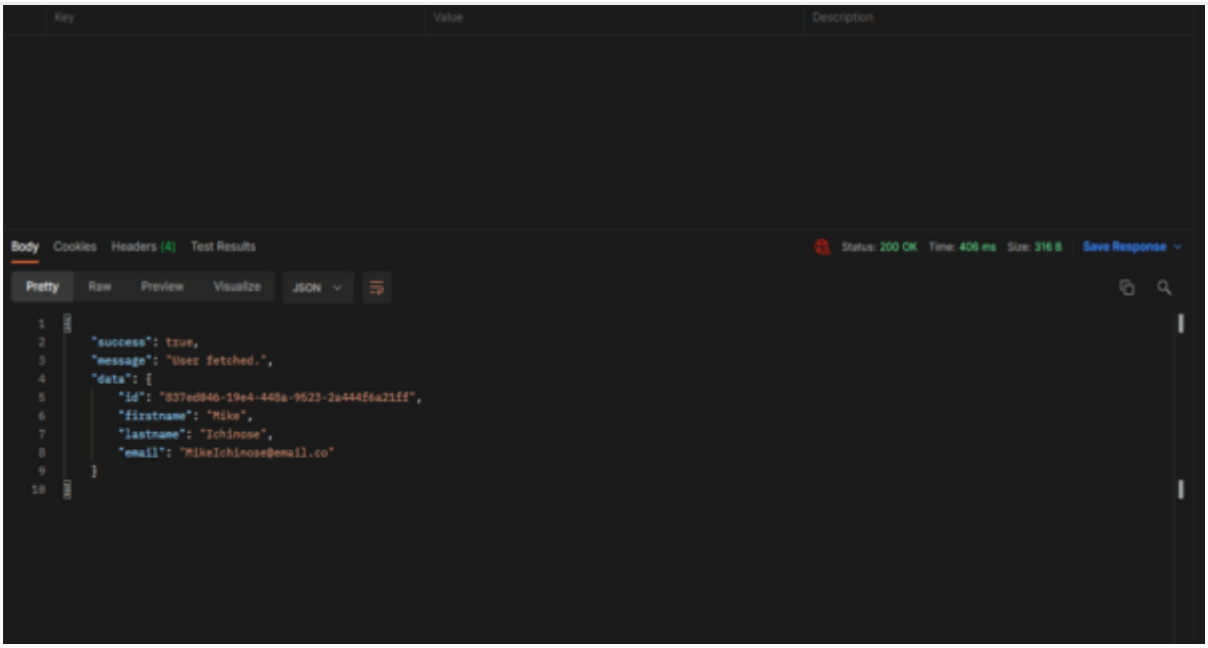
Now let's start our app and test with Postman.

## User Endpoints

**[GET] /users** - Gets all users

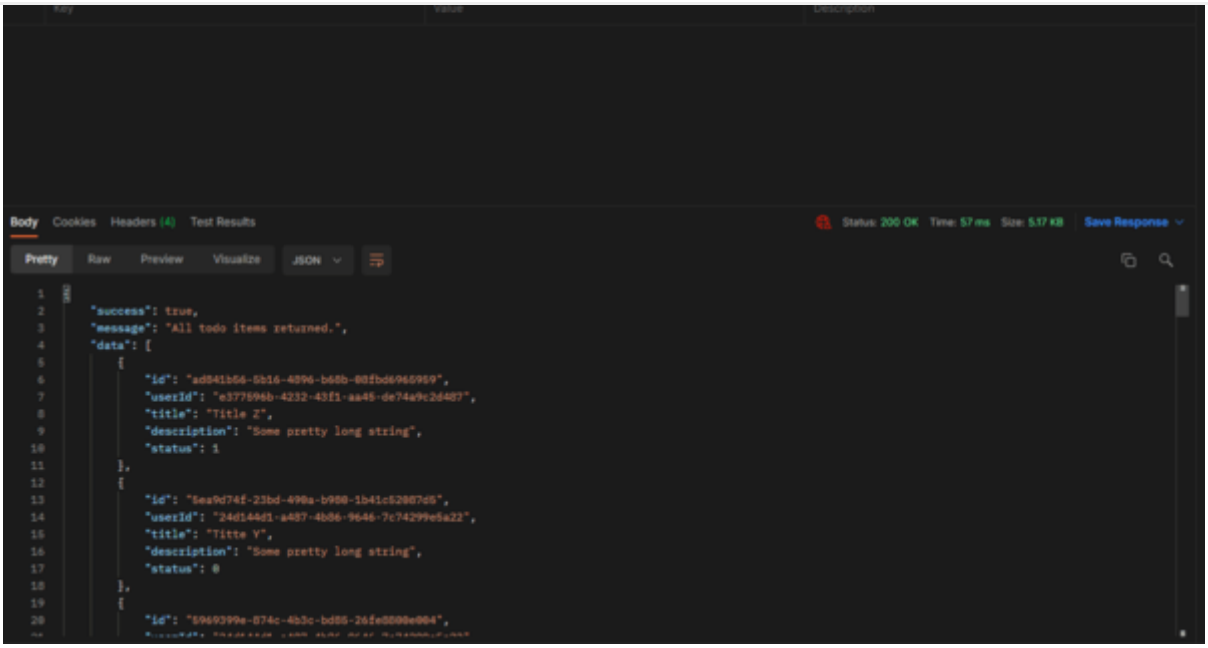



**[GET] /users/{user_id}** - Gets user by ID. We'll test this with one user's ID from the result returned above.

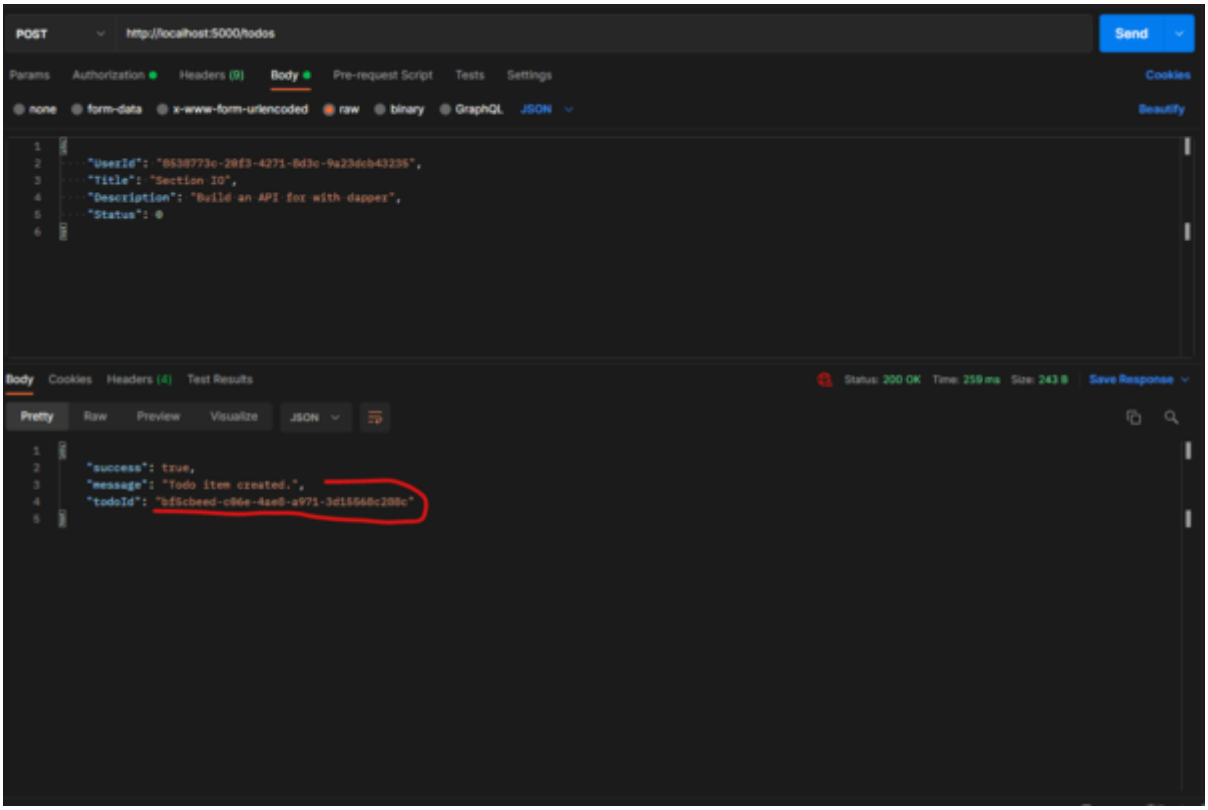



## Todos Endpoints

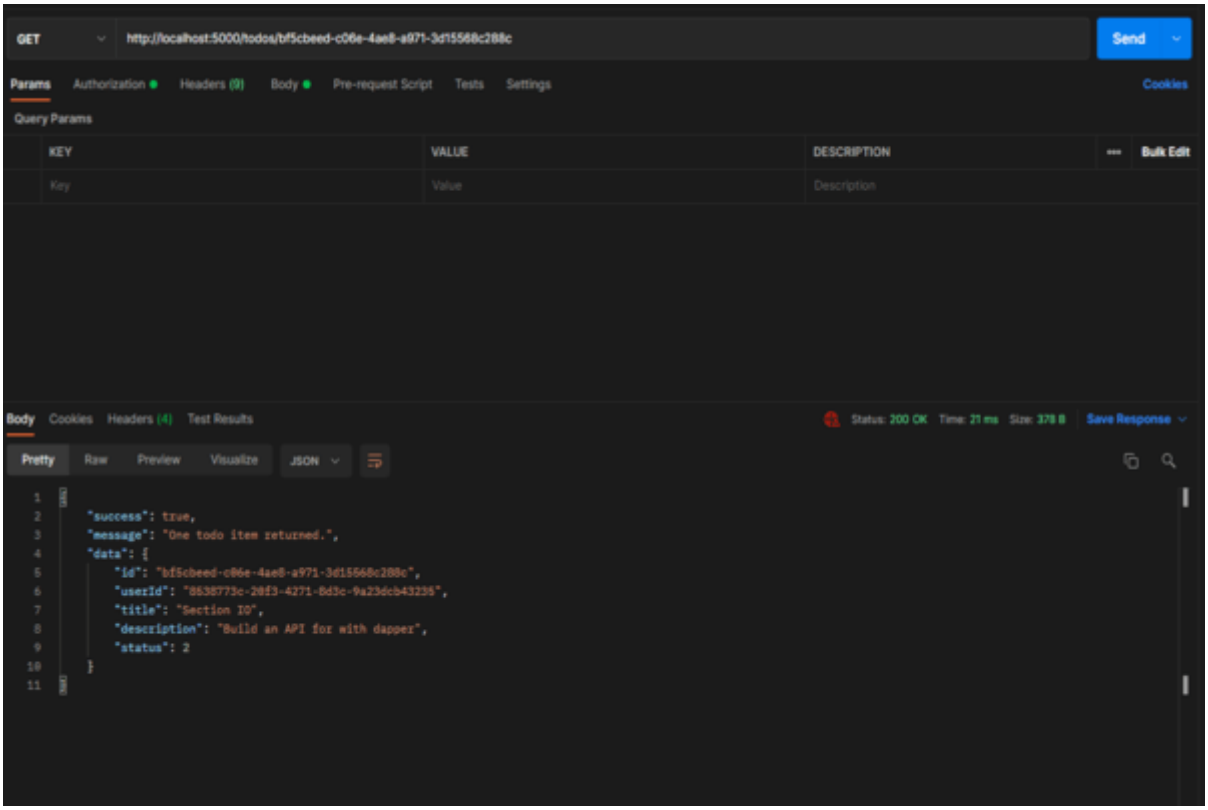
**[GET] /todos** - Gets all todo items



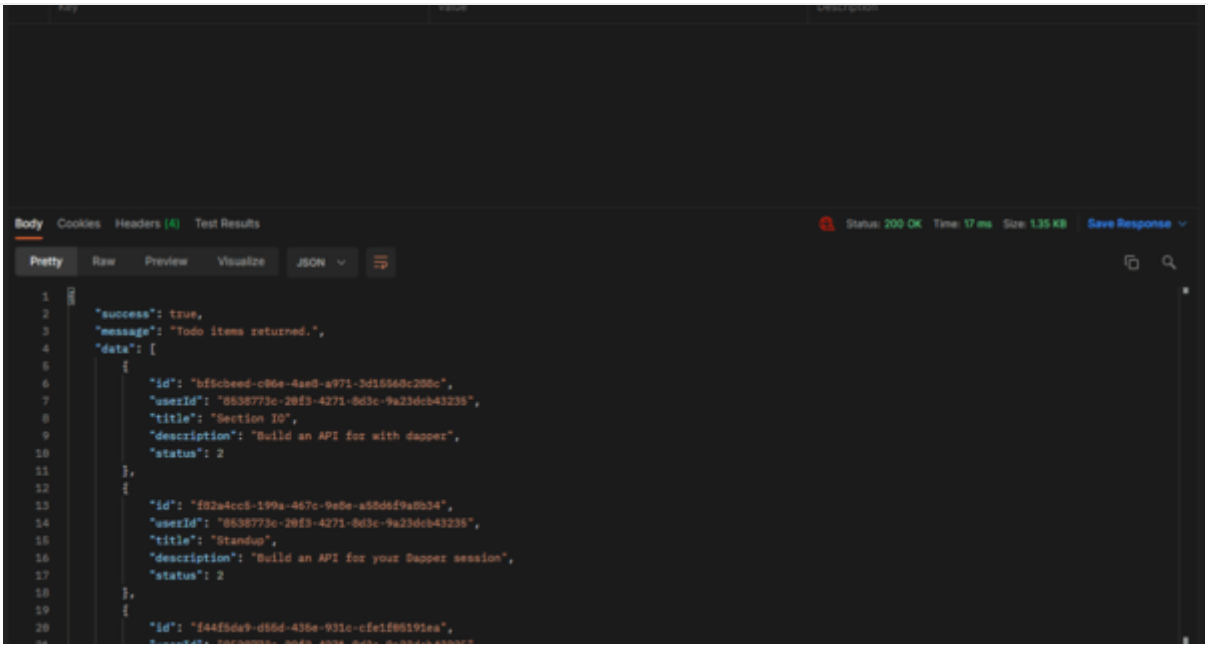

**[POST] /todos** - Creates a new todo item



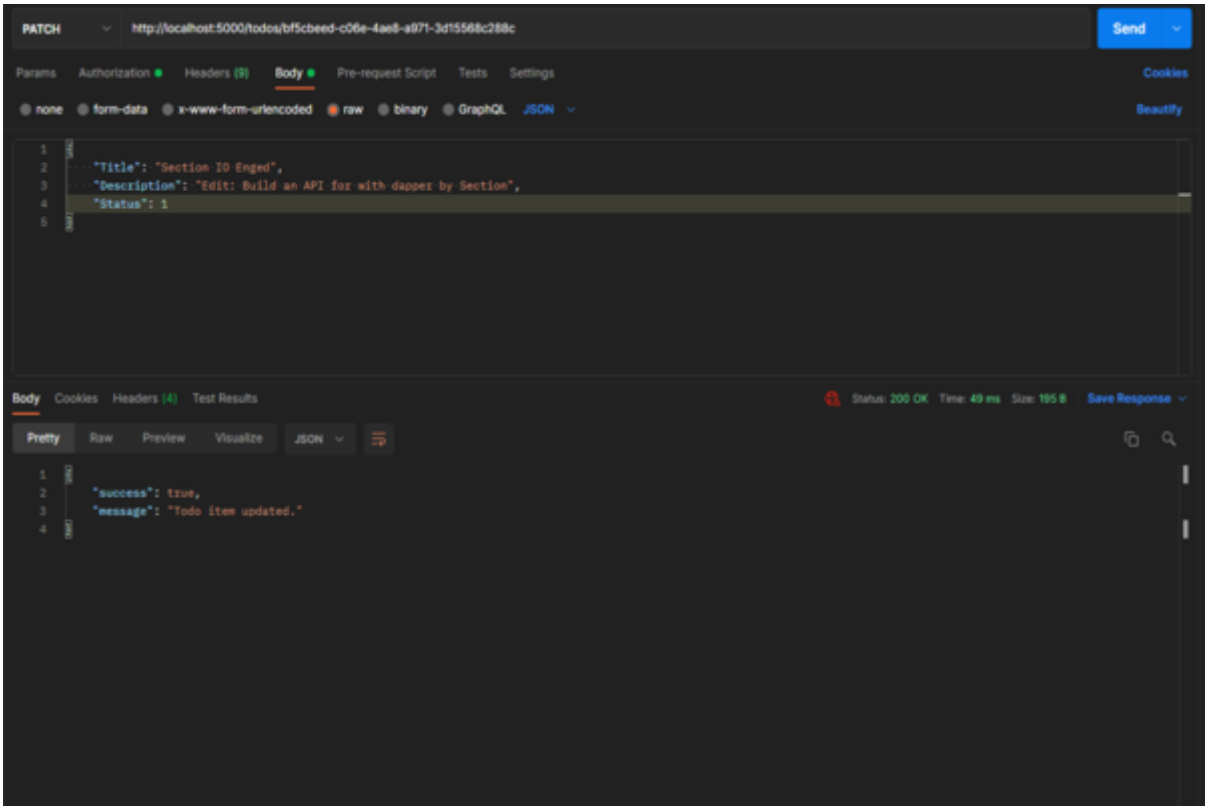
endpoint above. This tests our get-by-ID endpoint and also confirms that the todo item was indeed added to the database.

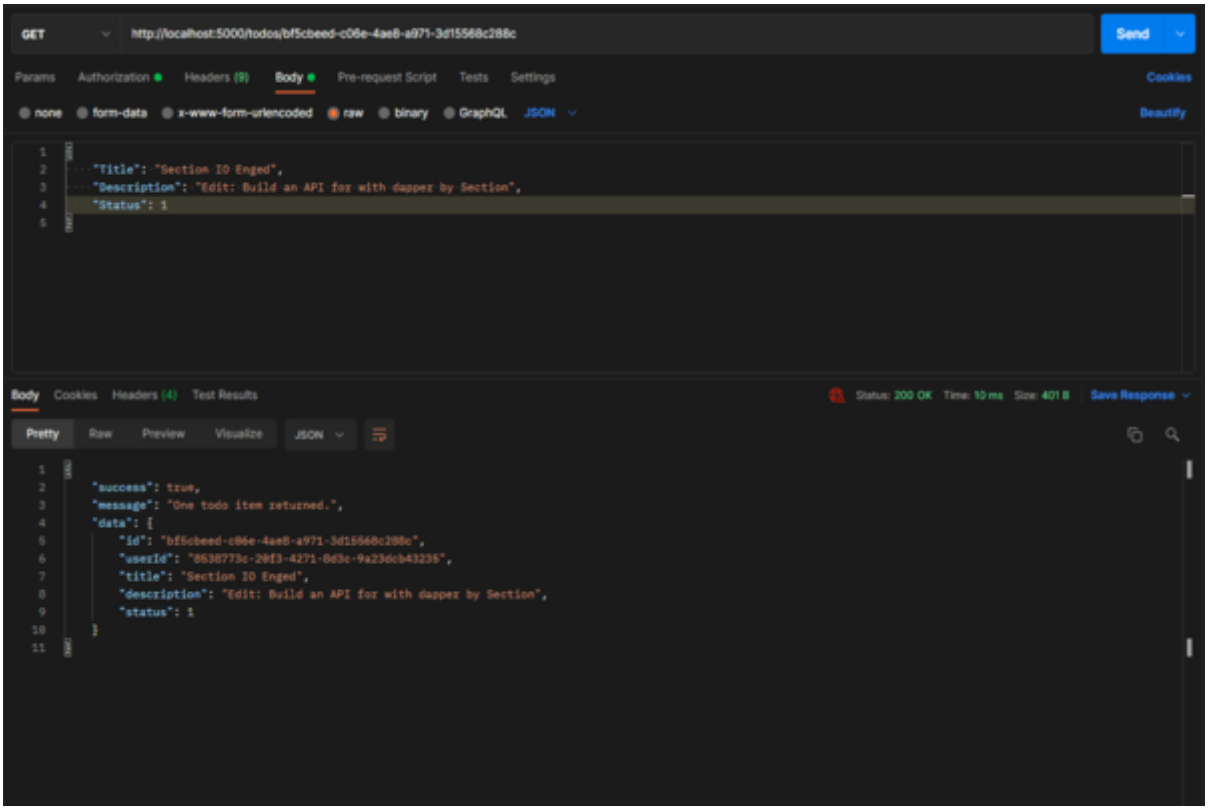



**[GET] /todos/users/{user_id}** - Gets todo items by user ID



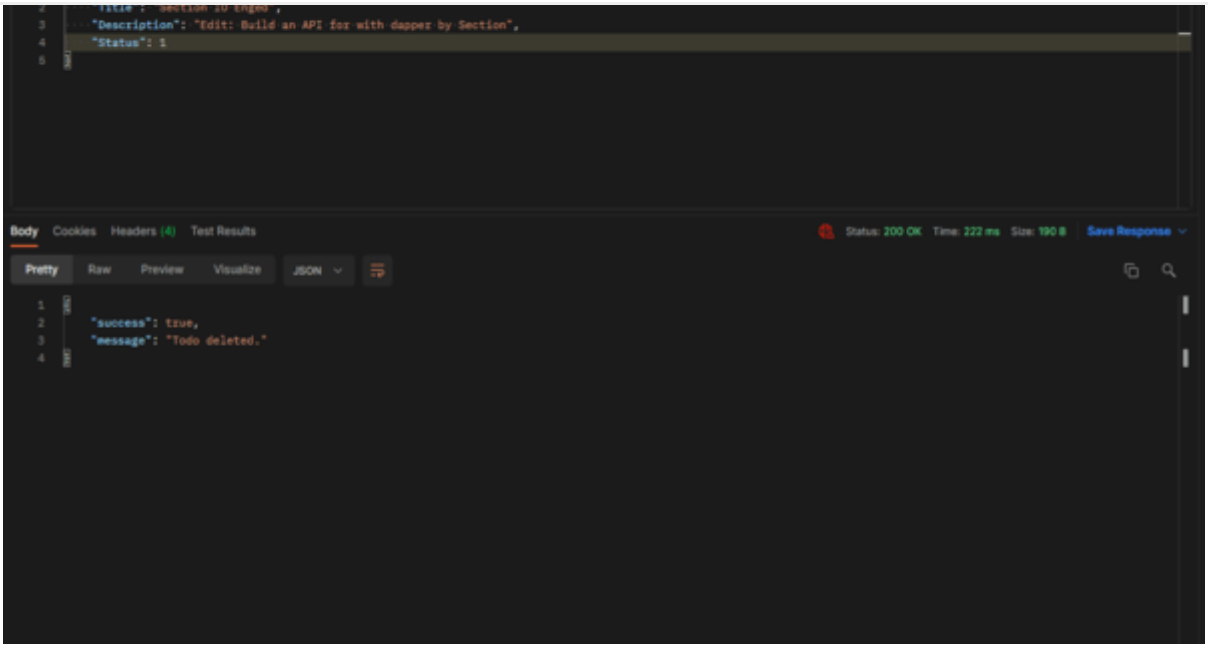

## **[PATCH] /todos** - Updates a todo item



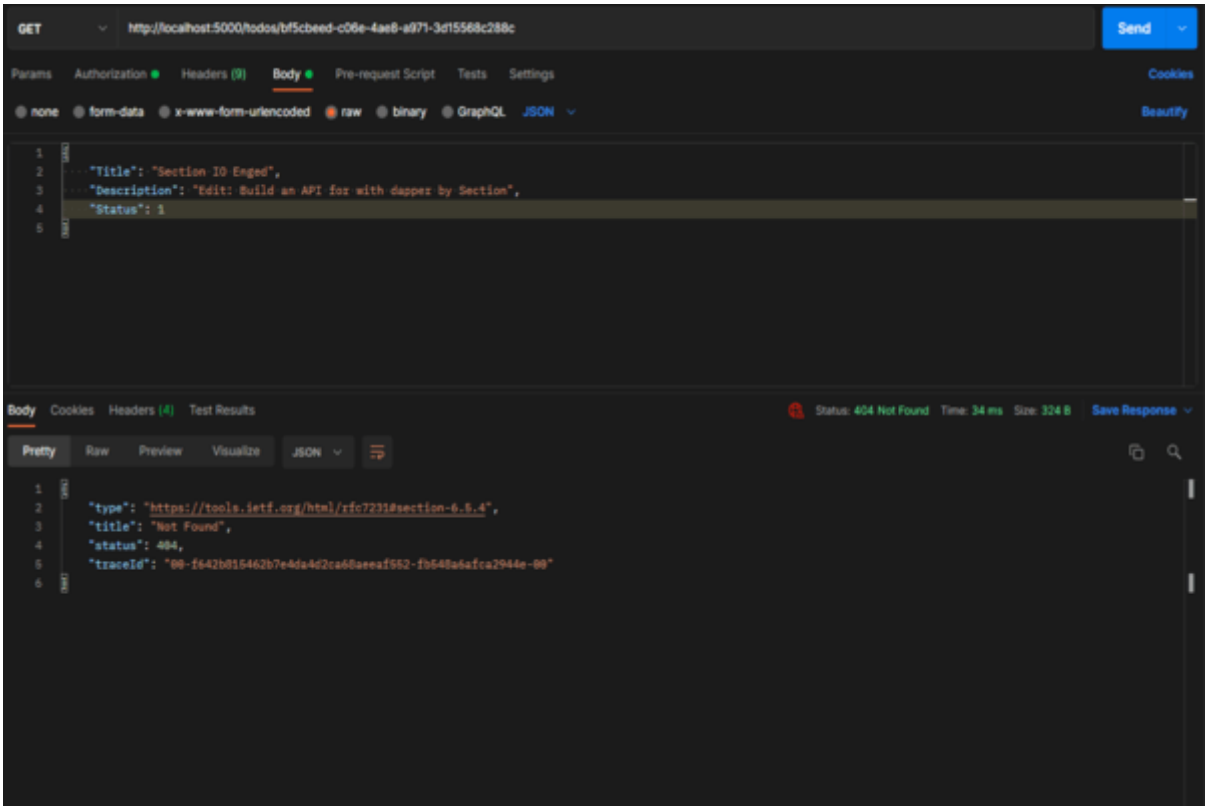



Sure enough, the fields were indeed updated as we can see above.

**[DELETE] /todos** - Deletes a todo item



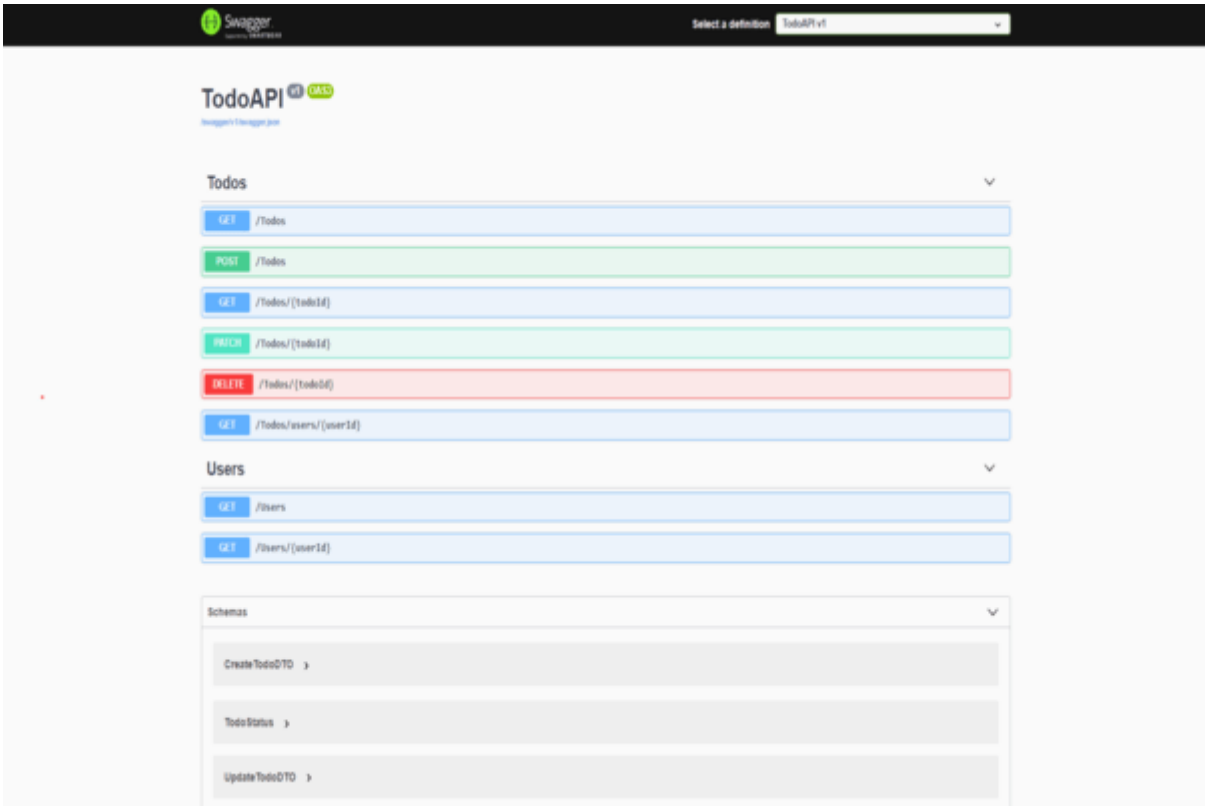

We will then try to get that item we just deleted.



want.

## Bonus: Documenting Our API

For the ASP.NET Core 5, there is built-in support for OpenAPI and Swagger UI. All you have to do is navigate to `/swagger`. You should see something similar to the image below:




## Conclusion

This article introduced you to Object Relational Mappers(ORMs) and how they work. More specifically, it explains what Dapper is and why you want to use it.

The complete code is available on GitHub. Feel free to add more features or contribute!

---

Peer Review Contributions by: Odhiambo Paul



EngEd Author Bio

Olumide is a Software developer who mostly works on backend solutions. He loves to solve problems and help people with code.

View authors full profile →

## Join our Slack community

Add to **Slack**

Sect

Help & Support

Platform Status

Section supports many open source projects including:

varnish cache logo

cloud native computing foundation logo

the linux foundation logo

lf edge logo

Section



Privacy Policy Terms of Service