# Using Dapper with ASP.NET Core Web API

In this article, we'll learn how to use Dapper in the ASP.NET Core Web API project. We'll talk about Dapper overall, how to use different queries and executions, how to execute stored procedures, and how to create multiple queries inside a transaction. We'll also create a simple repository layer to wrap the logic up to avoid using Dapper queries directly inside the controller.

---

To download the source code for this article, you can visit the **Dapper with ASP.NET Core Web API** repository.

So, let's start.

## About Dapper

Dapper is an ORM (Object-Relational Mapper) or, more precisely, a Micro ORM, which we can use to communicate with the database in our projects. We can write SQL statements using Dapper as we would in the SQL Server. Dapper performs well because it doesn't translate queries we write in .NET to SQL. It is important to know that Dapper is SQL Injection safe because we can use parameterized queries, which we should always do. One more important thing is that Dapper supports multiple database providers. It extends ADO.NET's IDbConnection and provides useful extension methods to query our database. Of course, we have to write queries compatible with our database provider.

When discussing these extension methods, we must say that Dapper supports synchronous and asynchronous method executions. We'll use the asynchronous version of those methods.

### About Extension Methods

Dapper extends the IDbConnection interface with these multiple methods:

- **Execute** – an extension method that we use to execute a command one or multiple times and return the number of affected rows
- **Query** – with this extension method, we can execute a query and map the result
- **QueryFirst** – it executes a query and maps the first result

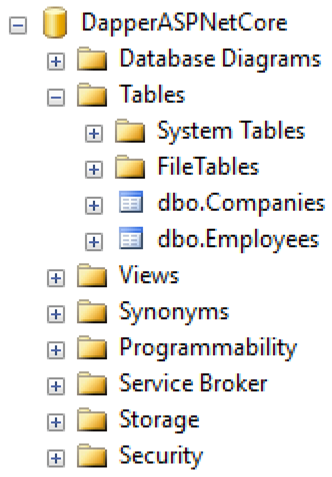
- **QueryFirstOrDefault** – we use this method to execute a query and map the first result or a default value if the sequence contains no elements
- **QuerySingle** – an extension method that executes a query and maps the result.  It throws an exception if there is not exactly one element in the sequence
- **QuerySingleOrDefault** – executes a query and maps the result or a default value if the sequence is empty. It throws an exception if there is more than one element in the sequence
- **QueryMultiple** – an extension method that executes multiple queries within the same command and maps results

As we said, Dapper provides an async version for all these methods (ExecuteAsync, QueryAsync, QueryFirstAsync, QueryFirstOrDefaultAsync, QuerySingleAsync, QuerySingleOrDefaultAsync, QueryMultipleAsync).

## Database and Web API Creation and Dapper Installation

Before using Dapper in our project, we must prepare a database and create a new Web API project. So, let's start with the database.

The first thing we'll do is create a new `ASPNetCoreDapper` database. After the database creation, you can navigate to our **source code repository** and find a script (Initial Script with Data.sql) that you can execute to create two tables and populate them with data:



If you want to learn how to create migrations and seed data with Dapper, you can read our **Dapper Migrations with FluentMigrator and ASP.NET Core** article.

Once we have our table and the data, we can create a new Web API project.

With our project in place, we can install two required packages:

Is this material useful to you? Consider subscribing and get **ASP.NET Core Web API Best Practices** eBook for **FREE!**

- Dapper – `PM> Install-Package Dapper`
- SQL Client – `PM> Install-Package Microsoft.Data.SqlClient`

Excellent.

We can continue towards the Repository Pattern creation.

## Creating Repository Pattern

In this section, we are going to create a simple repository pattern. We'll make it simple because this article is about Dapper.

If you want to learn how to create a **fully-fledged Repository Pattern**, you can read our article on that topic. Also, you can find **the async version of it** if you want to write it that way.

That said, let's start by creating a new `Entities` folder with two classes inside:

```
public class Employee

{

   public int Id { get; set; }

   public string Name { get; set; }

   public int Age { get; set; }

   public string Position { get; set; }

   public int CompanyId { get; set; }

}

public class Company

{

   public int Id { get; set; }

   public string Name { get; set; }

   public string Address { get; set; }
   public string Country { get; set; }

   public List<Employee> Employees { get; set; } = new List<Employee>();

}
```

So, these are our two model classes representing our tables in the database.

After this, we can modify the `appsettings.json` file by adding our connection string inside:

```
"ConnectionStrings": {

   "SqlConnection": "server=.; database=DapperASPNetCore; Integrated Security=true"

 },
```

Of course, feel free to modify the connection string to fit your needs.

**Also, since the Microsoft.Data.SqlClient package version 4**, if your database is not using encryption, any connection will fail by default. So, you might get an error with your first request:
Is this material useful to you? Consider subscribing and get **ASP.NET Core Web API Best Practices** eBook for **FREE!**

```
A connection was successfully established with the server, but then an error occurred during the login
process.

(provider: SSL Provider, error: 0 - The certificate chain was issued by an authority that is not trusted.)
```

To avoid this, you have to modify your connection string:

```
SqlConnection": "server=.; database=Dapper; Integrated Security=true; Encrypt=false"
```

This will prevent the error.

Now, we are going to create a new `Context` folder and a new `DapperContext` class under it:

```
public class DapperContext

{

   private readonly IConfiguration _configuration;

   private readonly string _connectionString;

   public DapperContext(IConfiguration configuration)

   {
       _configuration = configuration;

       _connectionString = _configuration.GetConnectionString("SqlConnection");

   }

   public IDbConnection CreateConnection()

       => new SqlConnection(_connectionString);

}
```

We inject the `IConfiguration` interface to enable access to the connection string from our `appsettings.json` file. Also, we create the `CreateConnection` method, which returns a new `SQLConnection` object.

For this to work, we have to add several using statements:

```
using Microsoft.Data.SqlClient;

using Microsoft.Extensions.Configuration;

using System.Data;
```

After the class creation, we can register it as a singleton service in the `Startup` class, if you are using .NET5:

```
public void ConfigureServices(IServiceCollection services)

{

    services.AddSingleton<DapperContext>();

    services.AddControllers();

}
```

**In .NET 6 and above, we can do the registration in the Program class:**

Is this material useful to you? Consider subscribing and get **ASP.NET Core Web API Best Practices** eBook for **FREE!**

```
builder.Services.AddSingleton<DapperContext>();

builder.Services.AddControllers();
```

## Repository Interfaces and Classes

Next, we are going to create a new `Contracts` folder and a single interface inside it:

```
public interface ICompanyRepository
```

```
{

}
```

Also, let's create a new `Repository` folder, and a single class inside it:

```
public class CompanyRepository : ICompanyRepository

{

   private readonly DapperContext _context;

   public CompanyRepository(DapperContext context)

   {

      _context = context;

   }

}
```

Once we start working with Dapper queries, we'll populate both files.

Finally, let's register our interface and its implementation as a service in the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)

{

   services.AddSingleton<DapperContext>();

   services.AddScoped<ICompanyRepository, CompanyRepository>();

   services.AddControllers();

}
```

**Of course, for .NET 6, the registration is slightly different:**

```
builder.Services.AddSingleton<DapperContext>();

builder.Services.AddScoped<ICompanyRepository, CompanyRepository>();

builder.Services.AddControllers();
```

That's it.

We can move on to our first query.

## Using Dapper Queries in ASP.NET Core Web API

Let’s start with an example of returning all the companies from our database.

So, the first thing we want to do is to modify our `ICompanyRepository` interface:

```
public interface ICompanyRepository

{

   public Task<IEnumerable<Company>> GetCompanies();

}
```

Then, let’s implement this method in the `CompanyRepository` class:

```
public async Task<IEnumerable<Company>> GetCompanies()

{

   var query = "SELECT * FROM Companies";

   using (var connection = _context.CreateConnection())

   {

      var companies = await connection.QueryAsync<Company>(query);

      return companies.ToList();

   }

}
```

So, we create a `query` string variable where we store our SQL query to fetch all the companies. Then inside the `using` statement, we use our `DapperContext` object to create the `SQLConnection` object (or, to be more precise an `IDbConnection` object) by calling the `CreateConnection()` method. As you can see, as soon as we stop using our connection, we must dispose of it. Once we create a connection, we can use it to call the `QueryAsync` method and pass the query as an argument. Since the `QueryAsync()` method returns `IEnumerable<T>`, we convert it to a list as soon as we want to return a result.
It is important to notice that we use a strongly typed result from the `QueryAsync()` method: `QueryAsync<Company>(query)`. But Dapper supports anonymous results as well: `connection.QueryAsync(query)`. We are going to use the strongly typed results in our examples.

### API Controller Logic

Now, let’s create a new `CompaniesController` and modify it:

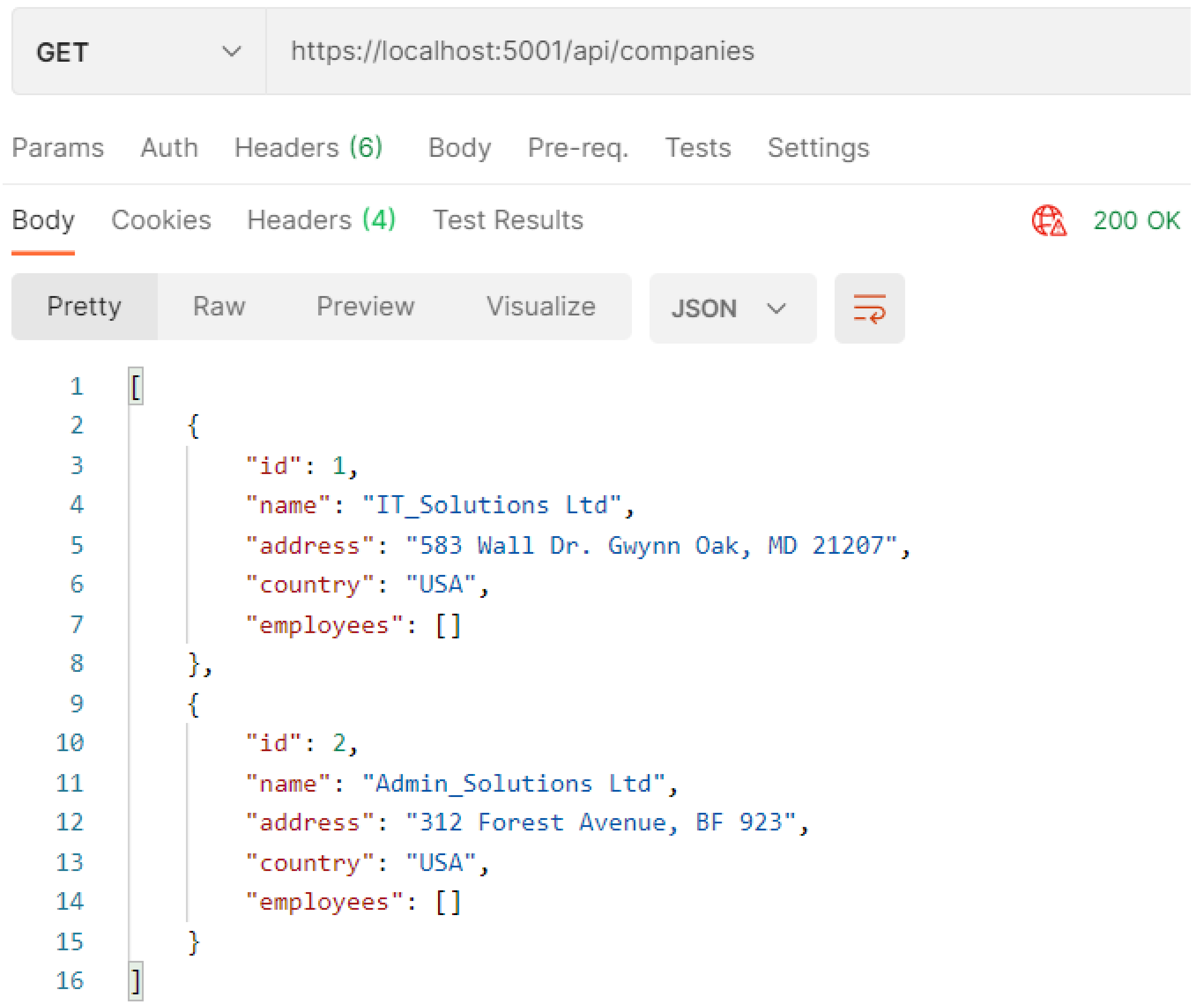
```
[Route("api/companies")]

[ApiController]

public class CompaniesController : ControllerBase

{

   private readonly ICompanyRepository _companyRepo;

   public CompaniesController(ICompanyRepository companyRepo)

   {

      _companyRepo = companyRepo;

   }

   [HttpGet]

   public async Task<IActionResult> GetCompanies()

   {

      try

      {

         var companies = await _companyRepo.GetCompanies();

         return Ok(companies);

      }

      catch (Exception ex)

      {

         //log error

         return StatusCode(500, ex.Message);

      }

   }

}
```

Here, we inject our repository via DI and use it to call our `GetCompanies` method.

A few notes here. Since we don't have any business logic, we are not creating a service layer to wrap our repository layer. For this type of application, the service layer would call repository methods and nothing more, adding an unnecessary level of complexity to the article. Of course, we always recommend using the service layer in larger-scale applications.

We'll use try-catch blocks in each action in our controller for the example's sake. But to avoid code repetition, we strongly suggest reading our **Global Error Handling** article.
Is this material useful to you? Consider subscribing and get **ASP.NET Core Web API Best Practices** eBook for **FREE!**

We'll not explain a controller's logic; we assume you are familiar with Web API development. If you want to learn more about that, you can read our **ASP.NET Core Web API** series
Now, we can start our app and test it:



```
[
    {
        "id": 1,
        "name": "IT_Solutions Ltd",
        "address": "583 Wall Dr. Gwynn Oak, MD 21207",
        "country": "USA",
        "employees": []
    },
    {
        "id": 2,
        "name": "Admin_Solutions Ltd",
        "address": "312 Forest Avenue, BF 923",
        "country": "USA",
        "employees": []
    }
]
```

Excellent.

We can see both our companies as a result.

## Different Property and Column Names

Right now, all the properties from the `Company` class have the same names as the columns inside the `Companies` table. But what would happen if those don't match?

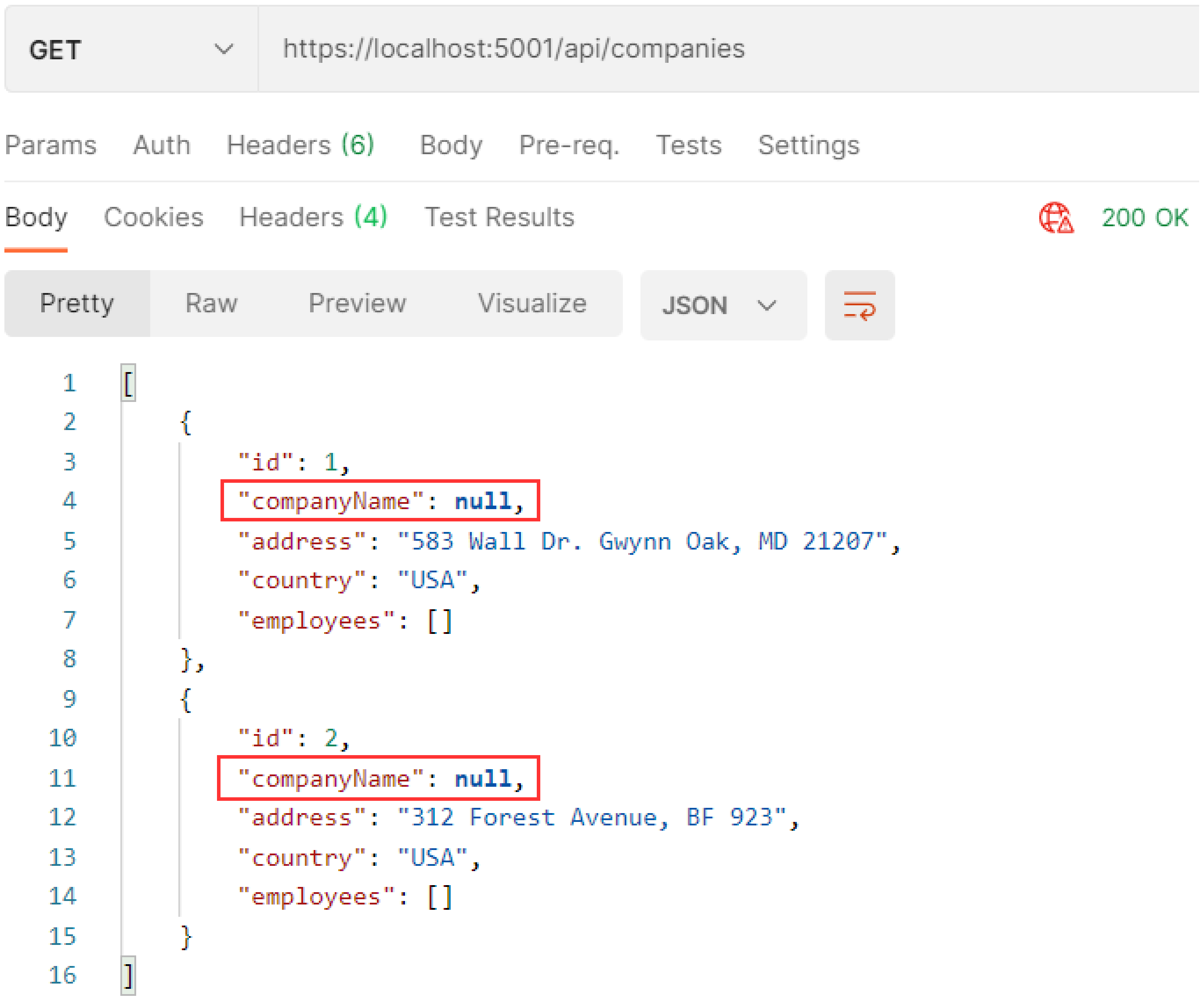
Let's check it out.

First, we are going to modify the `Name` property inside the `Company` class:

Is this material useful to you? Consider subscribing and get **ASP.NET Core Web API Best Practices** eBook for **FREE!**

```
public class Company

{

    public int Id { get; set; }

    public string CompanyName { get; set; }

    public string Address { get; set; }

    public string Country { get; set; }

    public List<Employee> Employees { get; set; } = new List<Employee>();

}
```

Now, if we run the same request, we are going to get a different result:



```
[
    {
        "id": 1,
        "companyName": null,
        "address": "583 Wall Dr. Gwynn Oak, MD 21207",
        "country": "USA",
        "employees": []
    },
    {
        "id": 2,
        "companyName": null,
        "address": "312 Forest Avenue, BF 923",
        "country": "USA",
        "employees": []
    }
]
```

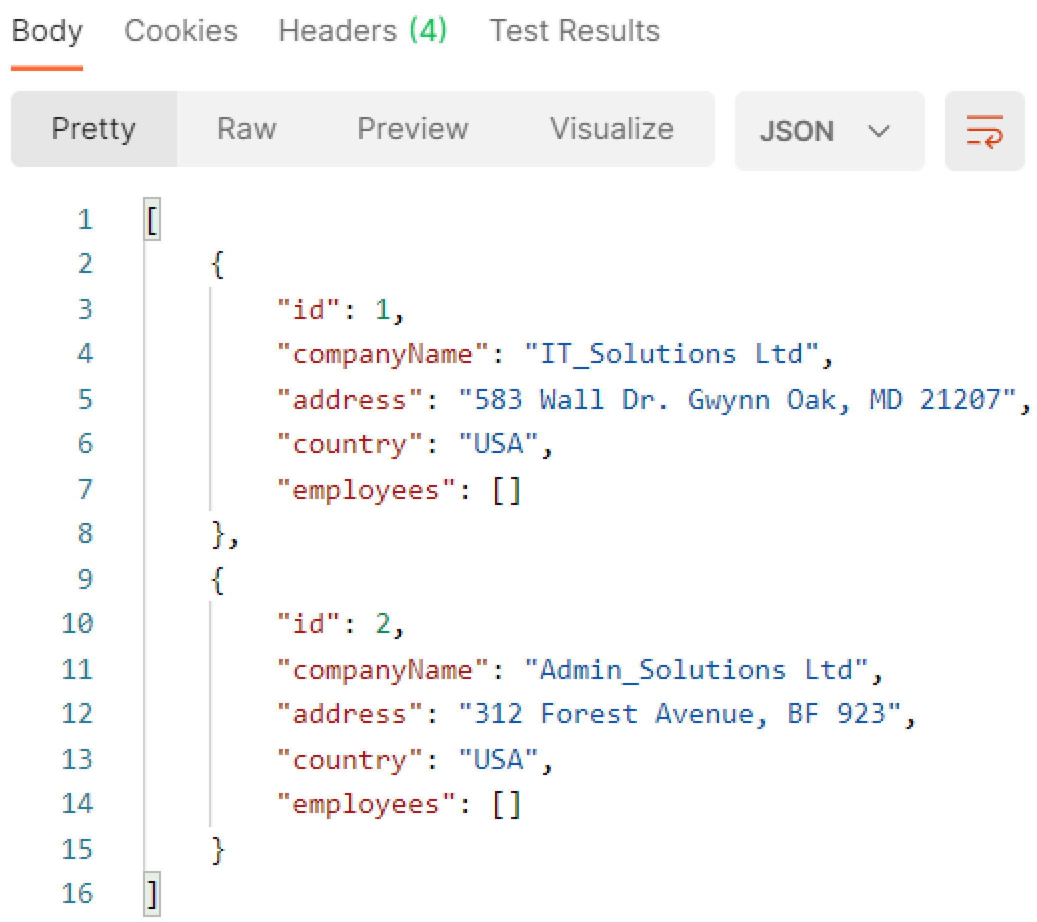
As we can see, the `companyName` property is null. This is because Dapper can't map it.

So, let's modify the query inside the `GetCompanies` method by using aliases:

```
public async Task<IEnumerable<Company>> GetCompanies()

{

  var query = "SELECT Id, Name AS CompanyName, Address, Country FROM Companies";

  using (var connection = _context.CreateConnection())

  {

    var companies = await connection.QueryAsync<Company>(query);

    return companies.ToList();

  }

}
```

As you can see, we are using the `AS` keyword to create an alias for the `Name` column.

Now, we can send the same request from Postman:



```json
[
    {
        "id": 1,
        "companyName": "IT_Solutions Ltd",
        "address": "583 Wall Dr. Gwynn Oak, MD 21207",
        "country": "USA",
        "employees": []
    },
    {
        "id": 2,
        "companyName": "Admin_Solutions Ltd",
        "address": "312 Forest Avenue, BF 923",
        "country": "USA",
        "employees": []
    }
]
```

We can see the mapping works perfectly.

Is this material useful to you? Consider subscribing and get **ASP.NET Core Web API Best Practices** eBook for **FREE!**

Let's return everything to a previous state to avoid writing aliases in future queries.

## Using Parameters With Dapper Queries

As we said at the beginning of this article, Dapper supports parameterized queries, making it 100% SQL injection-safe. It supports anonymous, dynamic, list, string, and table-valued parameters. We are mostly going to use dynamic and anonymous parameters in this article.

That said, let's start with the interface modification:

```
public interface ICompanyRepository
```

```
{
    public Task<IEnumerable<Company>> GetCompanies();
    public Task<Company> GetCompany(int id);
}
```

Then, let's add a method implementation in the `CompanyRepository` class:

```
public async Task<Company> GetCompany(int id)
{
    var query = "SELECT * FROM Companies WHERE Id = @Id";
    using (var connection = _context.CreateConnection())
    {
        var company = await connection.QuerySingleOrDefaultAsync<Company>(query, new { id });
        return company;
    }
}
```

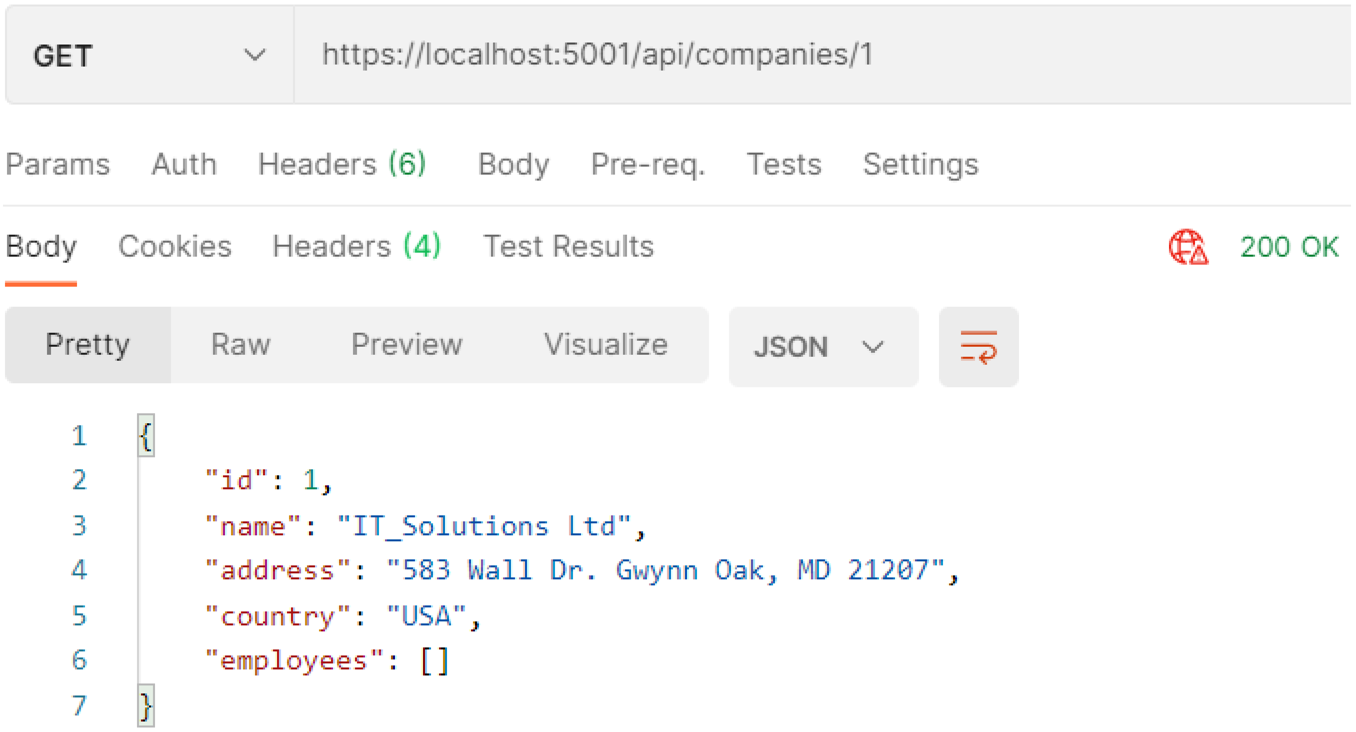
This method is almost the same as the previous one, but with one exception because we are using the `QuerySingleOrDefaultAsync` method here and provide an anonymous object as the second argument. We'll show you how to use dynamic parameters in the next example, where we'll create a new Company entity in our database.

Next, we have to modify our controller:

```
[HttpGet("{id}", Name = "CompanyById")]
public async Task<IActionResult> GetCompany(int id)
{
    try
    {
        var company = await _companyRepo.GetCompany(id);
        if (company == null)
            return NotFound();
        return Ok(company);
    }
    catch (Exception ex)
    {
        //log error
        return StatusCode(500, ex.Message);
    }
}
```

That's it.

We can test this with Postman:



https://localhost:5001/api/companies/1

```
{
    "id": 1,
    "name": "IT_Solutions Ltd",
    "address": "583 Wall Dr. Gwynn Oak, MD 21207",
    "country": "USA",
    "employees": []
}
```

Excellent. It works like a charm.

Also, if you want to read how to pass output parameters to stored procedures, you can do that in our **Passing Output Parameters to Stored Procedures** article.

## Adding a New Entity in the Database with the Execute(Async) Method

Now, we are going to handle a POST request in our API and use the `ExecuteAsync` method to create a new company entity in the database.

The first thing we are going to do is to create a new `Dto` folder, and inside it, a new `CompanyForCreationDto` class that we are going to use for the POST request:

```
public class CompanyForCreationDto

{

   public string Name { get; set; }

   public string Address { get; set; }

   public string Country { get; set; }

}
```

If you want to learn more about why we use this DTO (and we are going to use another one for the Update action), you can read our **ASP.NET Core Web API series** of articles, where we explain the reason behind this (articles **5** and **6** from the series).

After creating this class, we are going to modify our interface:

```
public interface ICompanyRepository

{

   public Task<IEnumerable<Company>> GetCompanies();

   public Task<Company> GetCompany(int id);

   public Task CreateCompany(CompanyForCreationDto company);

}
```

And, of course, let's implement this method in the repository class:

```
public async Task CreateCompany(CompanyForCreationDto company)

{

   var query = "INSERT INTO Companies (Name, Address, Country) VALUES (@Name, @Address, @Country)";

   var parameters = new DynamicParameters();
   parameters.Add("Name", company.Name, DbType.String);

   parameters.Add("Address", company.Address, DbType.String);

   parameters.Add("Country", company.Country, DbType.String);

   using (var connection = _context.CreateConnection())

   {

      await connection.ExecuteAsync(query, parameters);

   }

}
```

Here, we create our query and a dynamic parameters object (we are not using an anonymous object anymore). We populate that object with our three parameters and then call the `ExecuteAsync` method to execute our insert statement. The `ExecuteAsync` method returns `int` as a result, representing the number of affected rows in the database. So, if you need that information, you can use it by accepting the return value from this method.

Is this material useful to you? Consider subscribing and get **ASP.NET Core Web API Best Practices** eBook for **FREE!**

Now, if we call this method and pass a company for creation to it, it will create a new entity for us. But, while creating API's POST action, it is a good practice to return a link, which the API's users can use to navigate to the created entity. Our **Ultimate ASP.NET Core Web API book** explains this in great detail. We can't easily do that with the code, as we have it in this method. So, we have to modify a couple of things.

## Creating a Better API Solution

Firstly, let's modify the interface:

```
public interface ICompanyRepository

{

   public Task<IEnumerable<Company>> GetCompanies();

   public Task<Company> GetCompany(int id);

   public Task<Company> CreateCompany(CompanyForCreationDto company);

}
```

This time, we want our method to return a created company entity.

Next, we have to modify the method implementation:

```
public async Task<Company> CreateCompany(CompanyForCreationDto company)
{
  var query = "INSERT INTO Companies (Name, Address, Country) VALUES (@Name, @Address, @Country)"
+
    "SELECT CAST(SCOPE_IDENTITY() as int)";
  var parameters = new DynamicParameters();
  parameters.Add("Name", company.Name, DbType.String);
  parameters.Add("Address", company.Address, DbType.String);
  parameters.Add("Country", company.Country, DbType.String);
  using (var connection = _context.CreateConnection())
  {
    var id = await connection.QuerySingleAsync<int>(query, parameters);
    var createdCompany = new Company
    {
      Id = id,
      Name = company.Name,
      Address = company.Address,
      Country = company.Country
    };
    return createdCompany;
  }
}
```

We modify our query by adding another `SELECT` statement, which returns the last identity value created in the current scope. Then, we extract that id value by calling the `QuerySingleAsync()` method. This method executes both `INSERT` and `SELECT` statements. Once we have the Id value, we create a

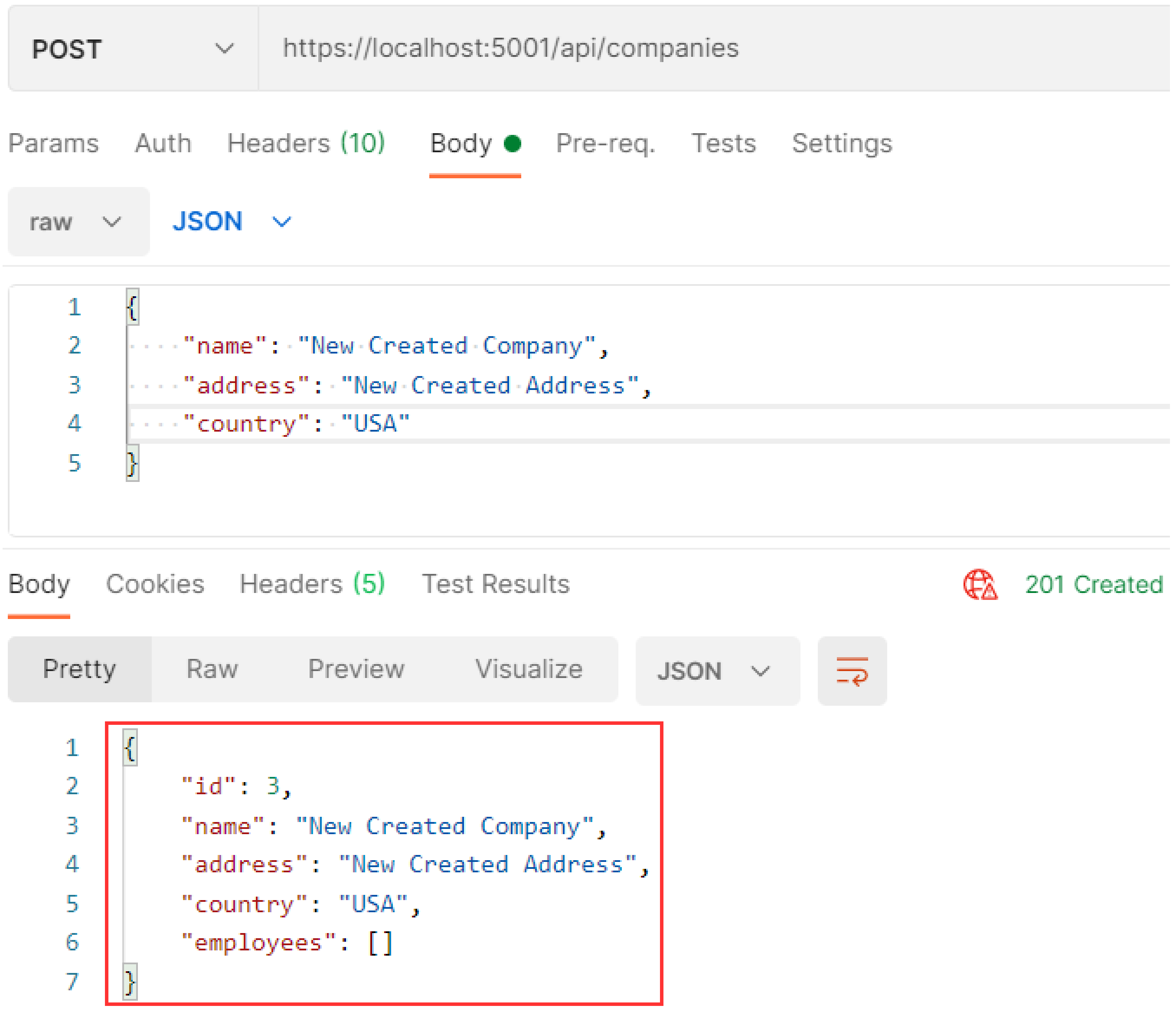
new company object with the required fields. Of course, you can do this with any mapping tool you like. Finally, we return our created entity.
With this out of the way, we can add a POST action in our controller:

```
[HttpPost]
public async Task<IActionResult> CreateCompany(CompanyForCreationDto company)
{
   try
   {
      var createdCompany = await _companyRepo.CreateCompany(company);
      return CreatedAtRoute("CompanyById", new { id = createdCompany.Id }, createdCompany);
   }
   catch (Exception ex)
   {
      //log error
      return StatusCode(500, ex.Message);
   }
}
```

As you can see, after creating a new company in the database, we return a route to fetch our newly created entity.

Let's test it:

Is this material useful to you? Consider subscribing and get **ASP.NET Core Web API Best Practices** eBook for **FREE!**



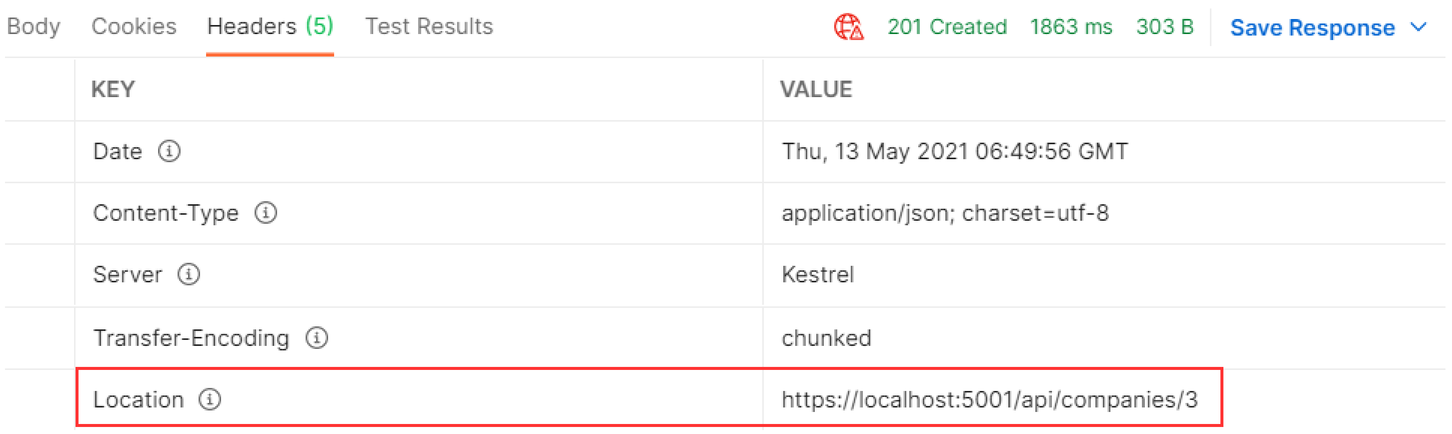
POST https://localhost:5001/api/companies

```
{
    "name": "New Created Company",
    "address": "New Created Address",
    "country": "USA"
}
```

```
{
    "id": 3,
    "name": "New Created Company",
    "address": "New Created Address",
    "country": "USA",
    "employees": []
}
```

There is our new company.

Also, if we inspect the Headers tab of the response, we are going to find a URI for this company:



| KEY | VALUE |
|---|---|
| Date | Thu, 13 May 2021 06:49:56 GMT |
| Content-Type | application/json; charset=utf-8 |
| Server | Kestrel |
| Transfer-Encoding | chunked |
| Location | https://localhost:5001/api/companies/3 |

Awesome.

Let’s move on.

## Working with Update and Delete

Working with the update and delete is pretty simple because we already have all the required knowledge. So, let’s jump straight to the code.

We are going to start with a new DTO:

```
public class CompanyForUpdateDto

{

    public string Name { get; set; }

    public string Address { get; set; }

    public string Country { get; set; }

}
```

Then, let’s modify the interface:

```
public interface ICompanyRepository

{

    public Task<IEnumerable<Company>> GetCompanies();

    public Task<Company> GetCompany(int id);

    public Task<Company> CreateCompany(CompanyForCreationDto company);

    public Task UpdateCompany(int id, CompanyForUpdateDto company);

    public Task DeleteCompany(int id);

}
```

Next, we have to implement these two methods in a repository class:

```
public async Task UpdateCompany(int id, CompanyForUpdateDto company)

{
    var query = "UPDATE Companies SET Name = @Name, Address = @Address, Country = @Country WHERE Id = @Id";
    var parameters = new DynamicParameters();
    parameters.Add("Id", id, DbType.Int32);
    parameters.Add("Name", company.Name, DbType.String);
    parameters.Add("Address", company.Address, DbType.String);
    parameters.Add("Country", company.Country, DbType.String);
    using (var connection = _context.CreateConnection())
    {
        await connection.ExecuteAsync(query, parameters);
    }
}
public async Task DeleteCompany(int id)
{
    var query = "DELETE FROM Companies WHERE Id = @Id";
    using (var connection = _context.CreateConnection())
    {
        await connection.ExecuteAsync(query, new { id });
    }
}
```

As you can see, there is nothing new with these two methods. We have a query and parameters, and we execute our statements with the `ExecuteAsync` method.

Finally, we have to add two actions to the controller:

Is this material useful to you? Consider subscribing and get **ASP.NET Core Web API Best Practices** eBook for **FREE!**

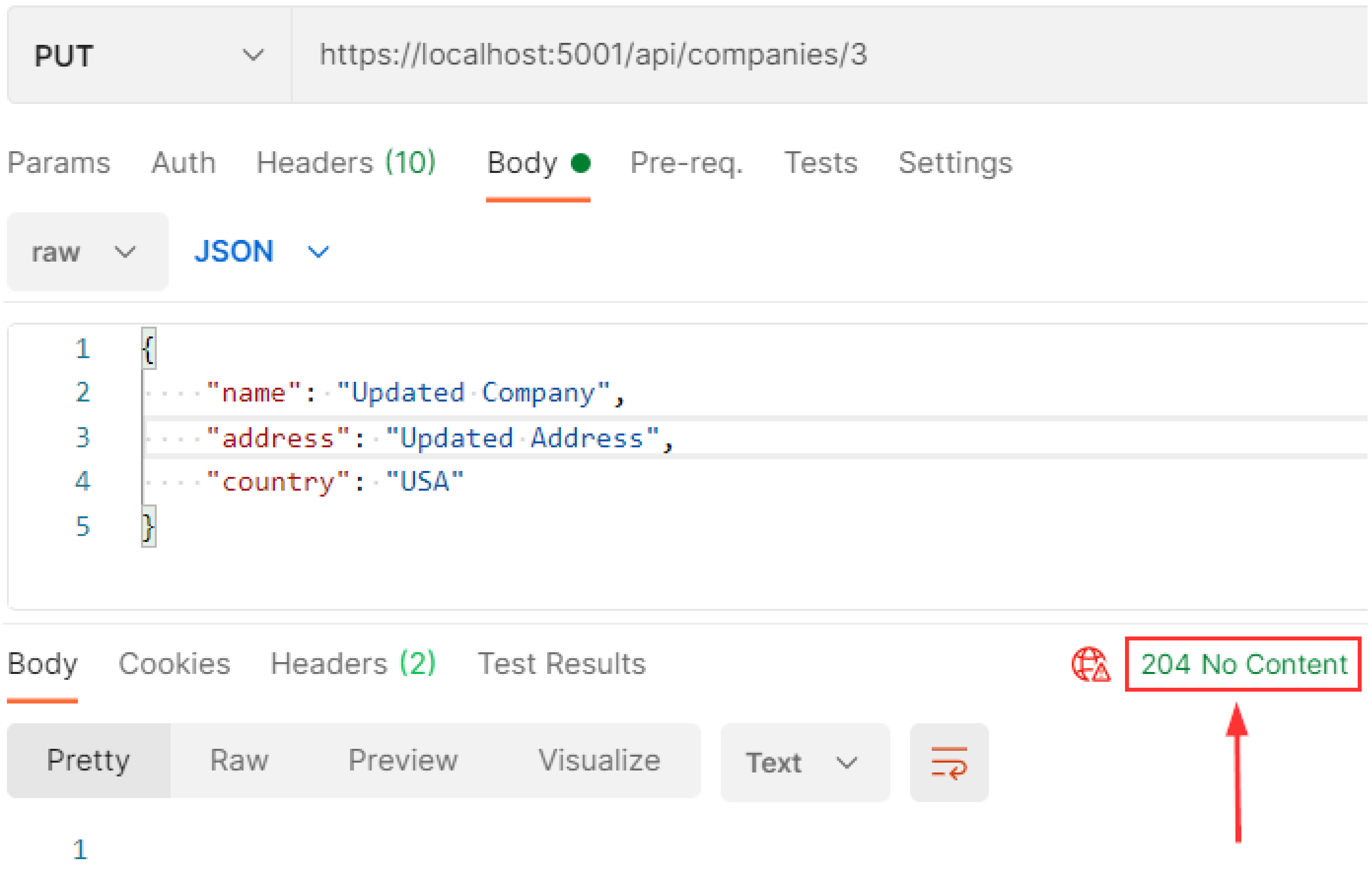
```
[HttpPut("{id}")]
public async Task<IActionResult> UpdateCompany(int id, CompanyForUpdateDto company)
```

```
{
   try
   {
      var dbCompany = await _companyRepo.GetCompany(id);
      if (dbCompany == null)
         return NotFound();
      await _companyRepo.UpdateCompany(id, company);
      return NoContent();
   }
   catch (Exception ex)
   {
      //log error
      return StatusCode(500, ex.Message);
   }
}
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteCompany(int id)
{
   try
   {
      var dbCompany = await _companyRepo.GetCompany(id);
      if (dbCompany == null)
         return NotFound();
      await _companyRepo.DeleteCompany(id);
      return NoContent();
   }
```

```
  catch (Exception ex)
  {
    //log error
    return StatusCode(500, ex.Message);
  }
}
```

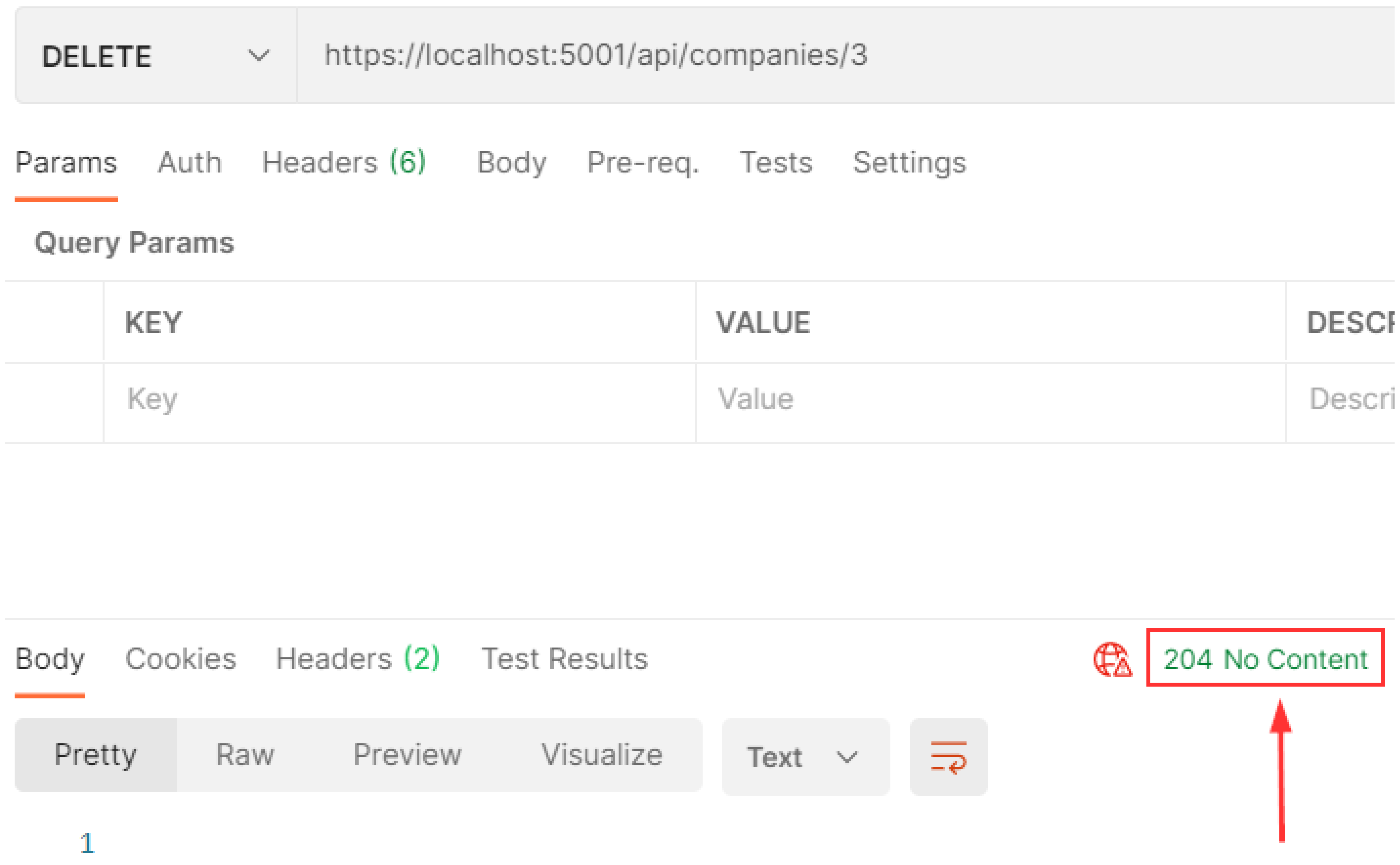
That’s it.

We can test this with Postman by sending both PUT and DELETE requests:



```
PUT https://localhost:5001/api/companies/3
```

```json
{
    "name": "Updated Company",
    "address": "Updated Address",
    "country": "USA"
}
```


DELETE https://localhost:5001/api/companies/3

204 No Content

Great job.

## Running Stored Procedures with Dapper

Before we show you how to use Dapper to call a stored procedure, we have to create one in our database:

```
USE [DapperASPNetCore]

GO

SET ANSI_NULLS ON

GO

SET QUOTED_IDENTIFIER ON

GO

CREATE PROCEDURE [dbo].[ShowCompanyForProvidedEmployeeId] @Id int

AS

SELECT c.Id, c.Name, c.Address, c.Country
```

```
FROM Companies c JOIN Employees e ON c.Id = e.CompanyId

Where e.Id = @Id

GO
```

This procedure returns a company's Name, Address, and Country with an employee with a provided Id value.

With the stored procedure in place, we can move on to the interface modification:

```
public interface ICompanyRepository

{

   public Task<IEnumerable<Company>> GetCompanies();

   public Task<Company> GetCompany(int id);

   public Task<Company> CreateCompany(CompanyForCreationDto company);

   public Task UpdateCompany(int id, CompanyForUpdateDto company);

   public Task DeleteCompany(int id);

   public Task<Company> GetCompanyByEmployeeId(int id);

}
```

Then, let's modify the class:

```
public async Task<Company> GetCompanyByEmployeeId(int id)

{

   var procedureName = "ShowCompanyForProvidedEmployeeId";

   var parameters = new DynamicParameters();

   parameters.Add("Id", id, DbType.Int32, ParameterDirection.Input);

   using (var connection = _context.CreateConnection())

   {

      var company = await connection.QueryFirstOrDefaultAsync<Company>

         (procedureName, parameters, commandType: CommandType.StoredProcedure);
```

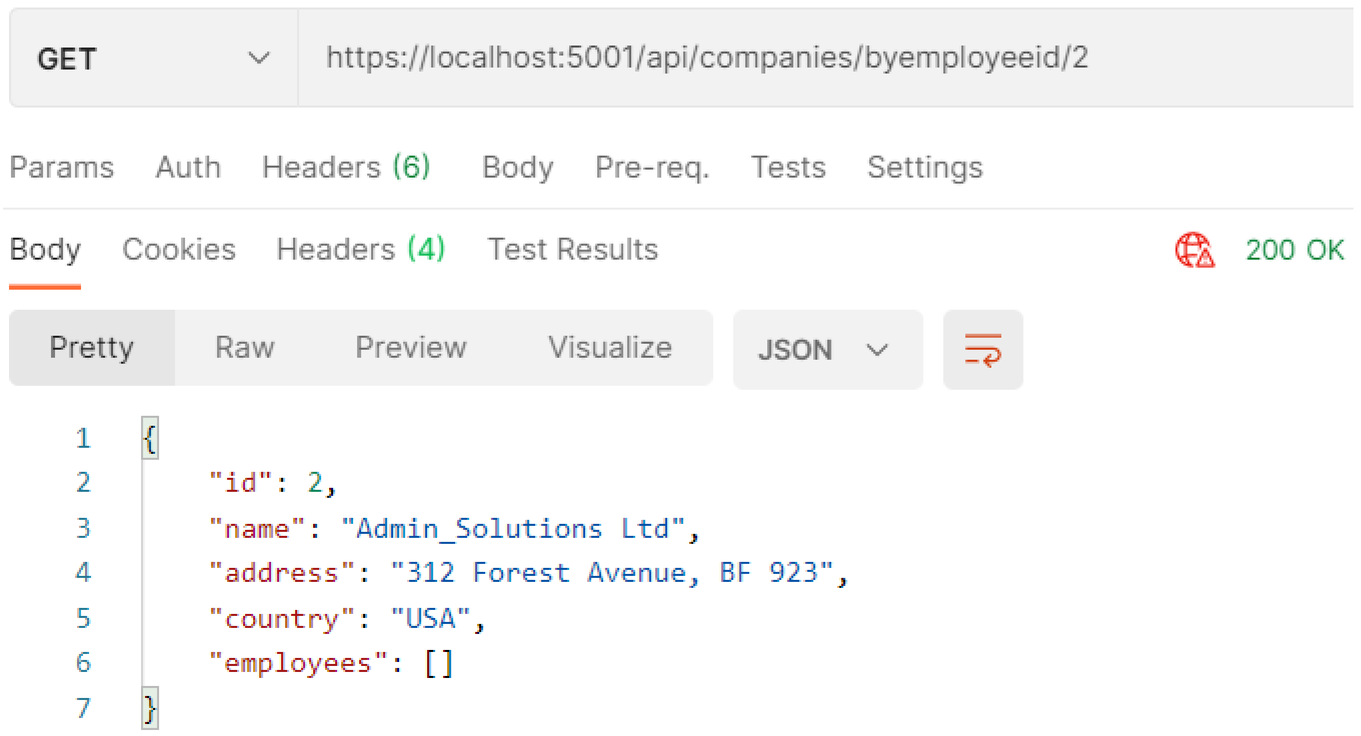
```
        return company;
    }
}
```

Here, we create a variable that contains a procedure name and a dynamic parameter object with a single parameter inside. Because our stored procedure returns a value, we use the `QueryFirstOrDefaultAsync` method to execute it. Pay attention that if your stored procedure doesn’t return a value, you can use the `ExecuteAsync` method for execution.

As usual, we have to add another action in our controller:

```
[HttpGet("ByEmployeeId/{id}")]
public async Task<IActionResult> GetCompanyForEmployee(int id)
{
    try
    {
        var company = await _companyRepo.GetCompanyByEmployeeId(id);
        if (company == null)
            return NotFound();
        return Ok(company);
    }
    catch (Exception ex)
    {
        //log error
        return StatusCode(500, ex.Message);
    }
}
```

And, of course, the test:


GET https://localhost:5001/api/companies/byemployeeid/2

Params Auth Headers (6) Body Pre-req. Tests Settings

Body Cookies Headers (4) Test Results 200 OK

Pretty Raw Preview Visualize JSON

```
{
    "id": 2,
    "name": "Admin_Solutions Ltd",
    "address": "312 Forest Avenue, BF 923",
    "country": "USA",
    "employees": []
}
```

We can see how easy it is to call a stored procedure with Dapper in our project.

## Executing Multiple SQL Statements with a Single Query

Using the `QueryMultipleAsync()` method, we can easily execute multiple SQL statements and return multiple results in a single query.

Let's see how to do that with an example.

As always, we'll modify our interface first:

```
public interface ICompanyRepository
{
   ...
   public Task<Company> GetCompanyEmployeesMultipleResults(int id);
}
```

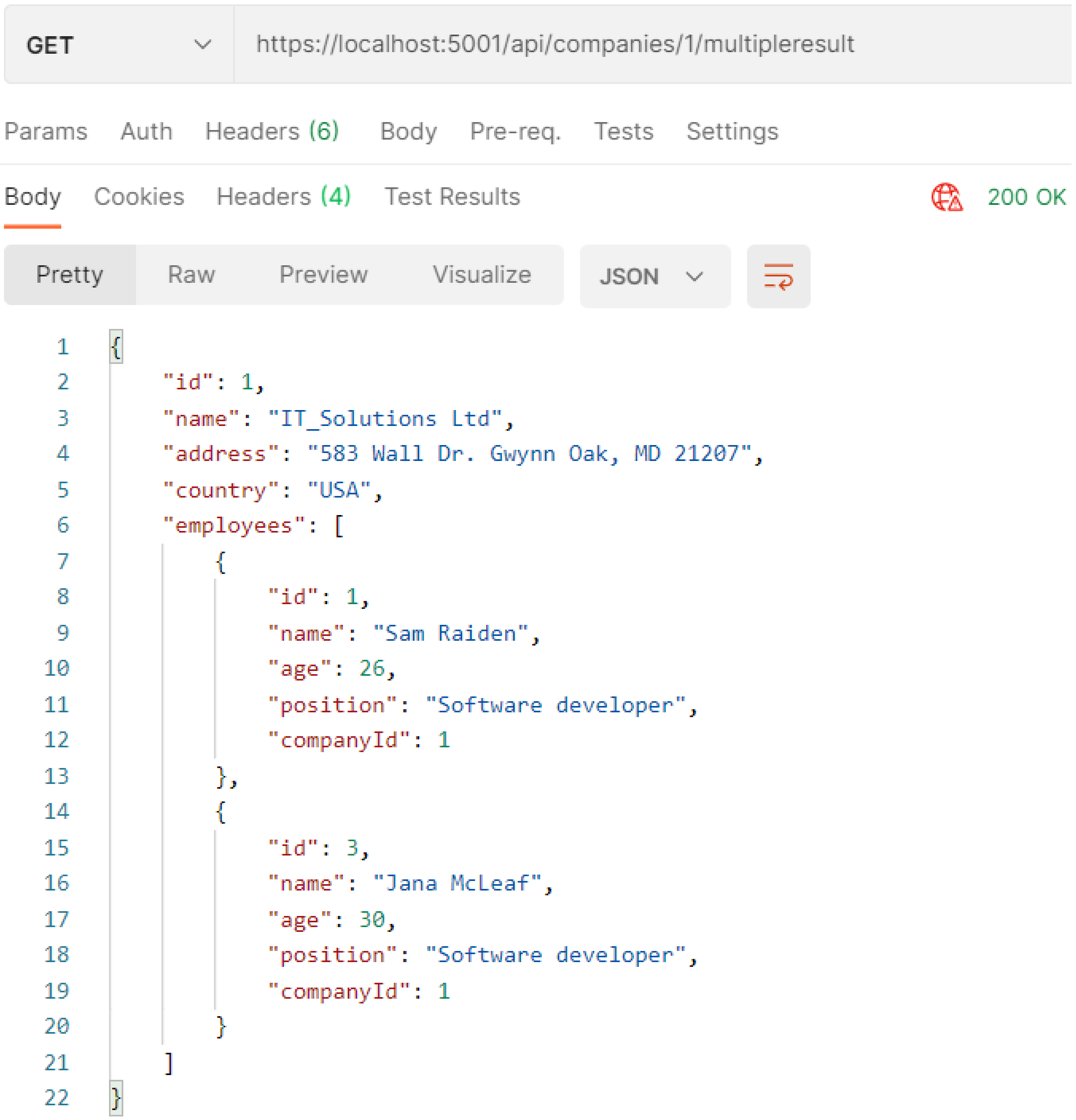
Then the implementation:

```
public async Task<Company> GetCompanyEmployeesMultipleResults(int id)
{
  var query = "SELECT * FROM Companies WHERE Id = @Id;" +

           "SELECT * FROM Employees WHERE CompanyId = @Id";

  using (var connection = _context.CreateConnection())

  using (var multi = await connection.QueryMultipleAsync(query, new { id }))

  {

     var company = await multi.ReadSingleOrDefaultAsync<Company>();

     if (company != null)

        company.Employees = (await multi.ReadAsync<Employee>()).ToList();

     return company;

  }

}
```

As you can see, our query variable contains two `SELECT` statements. The first will return a single company, and the second one will return all the employees of that company. After that, we create a connection and then use that connection to call the `QueryMultipleAsync` method. Once we get multiple results inside the `multi` variable, we can extract both results (company and employees per that company) by using the `ReadSignleOrDefaultAsync` and `ReadAsync` methods. The first method returns a single result, while the second returns a collection.
All we have to do is to create an action in the controller:

```
[HttpGet("{id}/MultipleResult")]

public async Task<IActionResult> GetCompanyEmployeesMultipleResult(int id)

{

  try

  {

     var company = await _companyRepo.GetCompanyEmployeesMultipleResults(id);

     if (company == null)

        return NotFound();

     return Ok(company);
  }
  catch (Exception ex)
  {
    //log error
    return StatusCode(500, ex.Message);
  }
}
```

And let’s test it:


GET
https://localhost:5001/api/companies/1/multipleresult
Params Auth Headers (6) Body Pre-req. Tests Settings
Body Cookies Headers (4) Test Results
200 OK
Pretty Raw Preview Visualize
JSON

```json
{
    "id": 1,
    "name": "IT_Solutions Ltd",
    "address": "583 Wall Dr. Gwynn Oak, MD 21207",
    "country": "USA",
    "employees": [
        {
            "id": 1,
            "name": "Sam Raiden",
            "age": 26,
            "position": "Software developer",
            "companyId": 1
        },
        {
            "id": 3,
            "name": "Jana McLeaf",
            "age": 30,
            "position": "Software developer",
            "companyId": 1
        }
    ]
}
```

There we go. We can see all the employees attached for a single company.

## Multiple Mapping

In a previous example, we used two SQL statements to return two results and then join them together in a single object. But usually, for such queries, we don't want to write two SQL statements. We want to use a JOIN clause and create a single SQL statement. Of course, if we write it like that, we can't use the `QueryMultipleAsync()` method anymore. We have to use a multiple mapping technique with a well-known `QueryAsync()` method.

So, let’s see how we can do it.

As usual, we are going to start with the interface modification:

```
public interface ICompanyRepository
{
   ...
   public Task<List<Company>> GetCompaniesEmployeesMultipleMapping();
}
```

Next, the implementation:

```
public async Task<List<Company>> GetCompaniesEmployeesMultipleMapping()
{
   var query = "SELECT * FROM Companies c JOIN Employees e ON c.Id = e.CompanyId";
   using (var connection = _context.CreateConnection())
   {
      var companyDict = new Dictionary<int, Company>();
      var companies = await connection.QueryAsync<Company, Employee, Company>(
         query, (company, employee) =>
         {
            if (!companyDict.TryGetValue(company.Id, out var currentCompany))
            {
               currentCompany = company;
               companyDict.Add(currentCompany.Id, currentCompany);
            }
            currentCompany.Employees.Add(employee);
            return currentCompany;
         }
```

```
    );
    return companies.Distinct().ToList();
  }
}
```

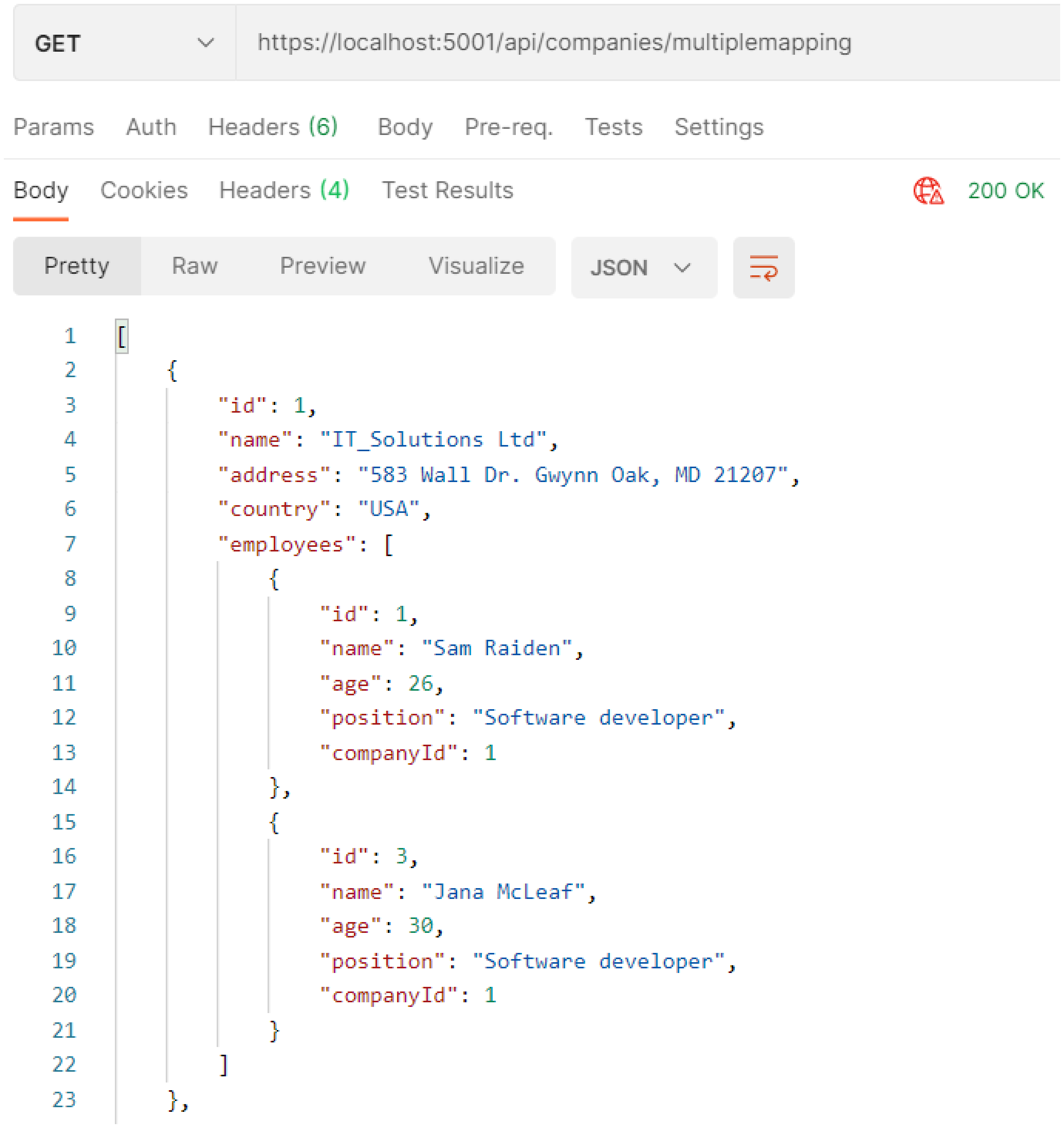
So, we create a query, and inside the using statement, a new connection. Then, we create a new dictionary to keep our companies in. To extract data from the database, we are using the `QueryAsync()` method, but it has a new syntax we haven't seen this time. We can see three generic types. The first two are the input types we'll work with, and the third is the return type. This method accepts our query as a parameter and a `Func` delegate that accepts two parameters of type `Company` end `Employee`. Inside the delegate, we try to extract a company by its Id value. If it doesn't exist, we store it inside the `currentCompany` variable and add it to the dictionary. Also, we assign all the employees to that current company and return it from a `Func` delegate.
After mapping, we return a distinct result converted to a list.

All we have left to do is to add an action to the controller:

```
[HttpGet("MultipleMapping")]
public async Task<IActionResult> GetCompaniesEmployeesMultipleMapping()
{
  try
  {
    var company = await _companyRepo.GetCompaniesEmployeesMultipleMapping();
    return Ok(company);
  }
  catch (Exception ex)
  {
    //log error
    return StatusCode(500, ex.Message);
  }
}
```

That should be it.

Let’s test it:



GET https://localhost:5001/api/companies/multiplemapping

```
[
    {
        "id": 1,
        "name": "IT_Solutions Ltd",
        "address": "583 Wall Dr. Gwynn Oak, MD 21207",
        "country": "USA",
        "employees": [
            {
                "id": 1,
                "name": "Sam Raiden",
                "age": 26,
                "position": "Software developer",
                "companyId": 1
            },
            {
                "id": 3,
                "name": "Jana McLeaf",
                "age": 30,
                "position": "Software developer",
                "companyId": 1
            }
        ]
    },
```

And we can see it works like a charm.

**One important note:** If you like the article so far, then maybe it can help even more for you to know that we updated our **Web API Premium edition** with another bonus book called **ASP.NET Core with Dapper**. In this book, you can read more about these topics: migrations with dapper, paging, searching, filtering, and sorting, and also about authorization with ASP.NET Core Identity.

We have one more thing left to cover – transactions.

## Transactions

Transactions are pretty simple to use with Dapper. We can execute it using the Dapper library (the one we already use) or the **Dappr.Transaction** library, which is the same thing as Dapper, just with the extended `IDbConnection` interface. In our example, we are going to use the Dapper library. We'll show you just the repository method where we implement transactions. All the rest is pretty simple as we repeated the steps several times in this article:

```
public async Task CreateMultipleCompanies(List<CompanyForCreationDto> companies)
{
   var query = "INSERT INTO Companies (Name, Address, Country) VALUES (@Name, @Address, @Country)";
   using (var connection = _context.CreateConnection())
   {
      connection.Open();
      using (var transaction = connection.BeginTransaction())
      {
         foreach (var company in companies)
         {
            var parameters = new DynamicParameters();
            parameters.Add("Name", company.Name, DbType.String);
            parameters.Add("Address", company.Address, DbType.String);
            parameters.Add("Country", company.Country, DbType.String);
            await connection.ExecuteAsync(query, parameters, transaction: transaction);
         }
```

```
            transaction.Commit();

        }

    }

}
```

So, there are four additional things here we haven't seen so far. First, we have to open the connection. Then, inside the `using` statement, we start our transactions by calling the `BeginTransaction()` method. In the `ExecuteAsync()` method, we specify our transaction. And finally, we call the `Commit` method to commit the transaction.

If you want to simulate an error and test that no rows will be created in the database, you can throw an exception below the `await` code line. You will find no new rows in the Companies table.

## Conclusion

There we go.

We've learned how to integrate Dapper into the ASP.NET Core project and how to use queries, executions, stored procedures, and transactions with Dapper.