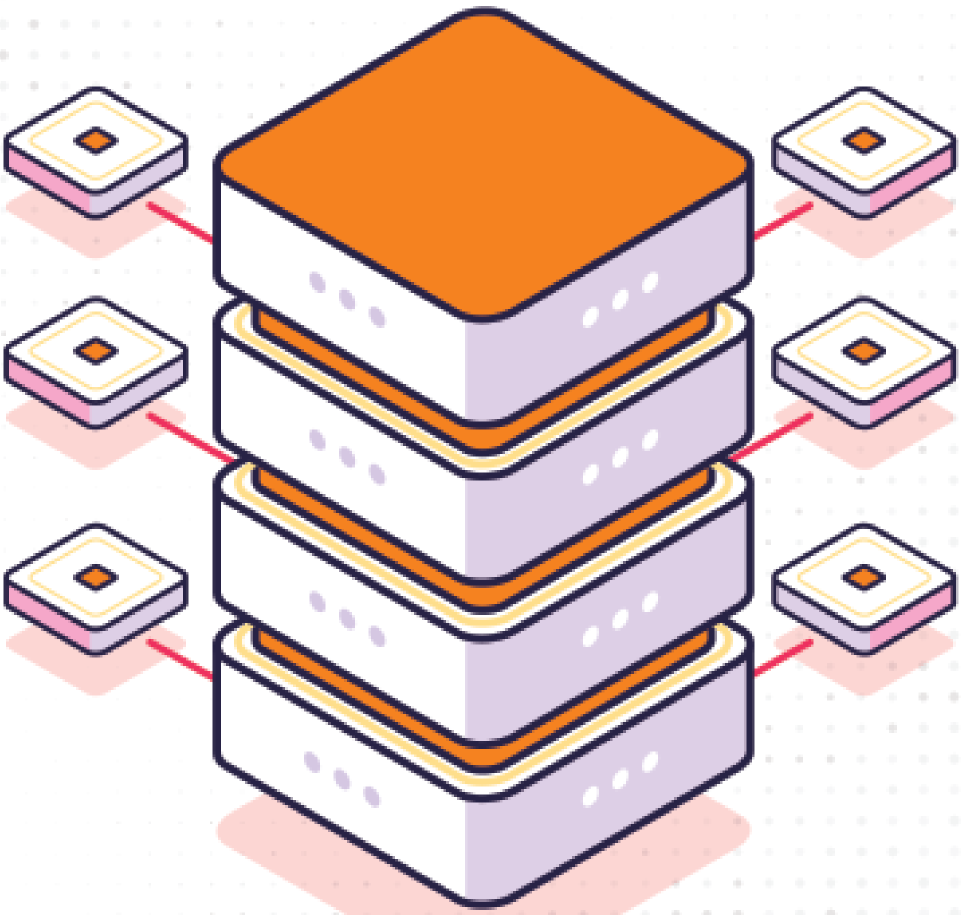
LIMITED
BUNDLE EDITION

Marinko Spasojevic
Vladimir Pecanac

ASP.NET CORE WEB API WITH DAPPER

Implementing Dapper ORM in ASP.NET Core Web API Applications



Made with ❤ by: CodeMaze

TABLE OF CONTENTS

1 About The Book

In this book, we are going to show you how to use Dapper with ASP.NET Core Web API.

We've received a lot of feedback about our main book and that our readers learned a lot but that would be great if we used Dapper instead of EntityFramework Core. So, we've decided to fulfill that wish.

That said, in this book, we are going to replace all the sections from the main book where we used EntityFramework Core with Dapper. You will learn how to handle different GET, POST, PUT, and DELETE requests, and how to use Dapper to create pagination, filtering, searching, and sorting. Also, we will show you how to use Dapper with ASP.NET Core Identity to fully support Identity logic for user's authentication and authorization.

Of course, **we strongly suggest reading the main book first before reading this one.** That's because, in this book, we won't be covering features unrelated to EF Core from the main book, there is no point in doing that since everything is clearly explained in the main book.

This means we won't cover validation, action filters, data shaping, hateoas, etc. because those are not EF Core/Dapper-related features. Also, even though we will cover pagination and other stuff here, we won't explain what pagination or sorting is, or anything similar, we will just show the step-by-step implementation. Again, there is no reason to repeat the same explanations we already have in the main book.

Our books are mainly created to help you learn ASP.NET Core Web API development, and that is the main reason for you to read the main book first, even if you don't want to use EF Core in your project.

As you will see in this book, switching from EF Core to Dapper is not a hard task to do. We will cover all the required steps to help you learn Dapper and use it in the ASP.NET Core Web API application.

We will still use the onion architecture in this book, and show you how you can implement Dapper in such a created project.

That said, we have prepared a starting project that you can find in the folder named **"start".** In that folder, you will find a fully prepared project with the onion architecture implemented. So as you can see, we will dedicate all our attention to the Dapper implementation thus making it easier to learn and digest.

2 About Dapper

Dapper is an ORM (Object-Relational Mapper) or to be more precise a Micro ORM, which we can use to communicate with the database in our projects. By using Dapper, we can write SQL statements the same way as we would do in a tool such as the query editor in SQL Server Management Studio. This means that we have to be very comfortable with writing queries in SQL. Dapper has great performance because it doesn't translate queries that we write in .NET to SQL.

It is essential to know that Dapper is SQL Injection safe because we can use parameterized queries, and that's something 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 we talk about these extension methods, we have to say that Dapper supports both synchronous and asynchronous method executions. In this book, we are going to use both synchronous methods for database creation and the asynchronous version of those methods for regular queries.

2.1 About Extension Methods

Dapper extends the `IDbConnection` interface with these multiple methods:

Execute – executes a command one or multiple times and returns the number of affected rows

Query –executes a query and maps the result

QueryFirst – this method 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 can execute a query and map the result. It throws an exception if there are 0 or more than 1 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).

Additionally, Dapper provides several methods for selecting scalar values: ExecuteScalar and ExecuteScalarAsync, which we will not use in this book.

It is important to say that Dapper provides these above Query and ExecuteScalar methods, **except the Execute method**, as strongly typed methods: QuerySingle<T>, QuerySingleOrDefault<T>, QueryFirst<T>, ExecuteScalar<T>, etc.

In this book, we will mostly use the strongly typed async methods.

Now, that we learned the basic theory about Dapper, we can move on to the database creation and migrations.

3 Dapper Database and Migrations

Mostly, when we work with Dapper, we start by creating a database first with all the tables and their relationships, and then once that's done, we continue with our C# models in the project.

This is a great approach, it works great, and we always support using the database first approach when starting a new project.

But, in our main book, we've created the database using the code-first approach where we used migrations to create a database and to seed some initial data.

That said, in this book, we are going to do the same.

Since Dapper doesn't support migrations by default, we have to use some additional help to create them. For that, we are going to use FluentMigrator.

3.1 Entities, Context, and FluentMigrator Installation

So, let's open our initial project from the "start" folder.

The first thing we are going to do is to install two required packages in the **`Repository`** project:

```
PM> Install-Package Dapper
```

And:

```
PM> Install-Package FluentMigrator.Runner
```

We need the first package, so we could work with Dapper on our project and the second one for our migrations.

The **`FluentMigrator.Runner`** has the dependency on the core **`FluentMigrator`** package and also on the

FluentMigrator.Runner.SqlServer package, so we don't have to install them separately.

Additionally, the FluentMigrator.Runner.SqlServer package has a dependency on Microsoft.Data.SqlClient package, which we will need to create the sql connection in the DapperContext class.

With the packages installed, we can move on to entity creation.

Inside the Entities project, we will find the Models folder, and inside we are going to create our two entities.

Company:

```
public class Company
{
    public Guid CompanyId { get; set; }
    public string? Name { get; set; }
    public string? Address { get; set; }
    public string? Country { get; set; }
}
```

And Employee:

```
public class Employee
{
    public Guid EmployeeId { get; set; }
    public string? Name { get; set; }
    public int Age { get; set; }
    public string? Position { get; set; }
    public Guid CompanyId { get; set; }
}
```

Comparing with the entities that we created for EntityFrameork Core, you can see that we don't use the Required or MaxLength, or any other attributes to provide additional info for the migration. We will provide that information in the migration file itself.

With the entities in place, we can add two connection strings inside the appsettings.json file:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "sqlConnection": "server=.; database=CompanyEmployeeDapper; Integrated 
Security=true; Encrypt=false",
    "masterConnection": "server=.; database=master; Integrated Security=true; 
Encrypt=false"
  },
  "AllowedHosts": "*"
}
```

We are going to use the **sqlConnection** to fetch the data from the existing database, and the **masterConnection** to check if the database exists during the database migration.

Then, let's navigate to the root of the **Repository** project and create the **DapperContext** class:

```
public class DapperContext
{
    private readonly IConfiguration _configuration;

    public DapperContext(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    public IDbConnection CreateConnection()
        => new SqlConnection(_configuration
            .GetConnectionString("sqlConnection"));

    public IDbConnection CreateMasterConnection()
        => new SqlConnection(_configuration
            .GetConnectionString("masterConnection"));
}
```

As you can see, we create two connections. One for the regular queries, and one for the master db check.

For this to work, we have to include a few namespaces:

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

Finally, we have to register our context class as a singleton service in the **Program.cs** file:

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

3.1.1 Creating the Database

To create the database for our project, we are only going to use Dapper, without the FluentMigrator package, for now. For this action, we don't need the FluentMigrator package. We will use it when we create our tables and seed our data.

That said, let's start by creating a new Database class in the Migrations folder in the main project:

```
public class Database
{
    private readonly DapperContext _context;

    public Database(DapperContext context) => _context = context;

    public void CreateDatabase(string dbName)
    {
        var query = "SELECT * FROM sys.databases WHERE name = @name";

        var parameters = new DynamicParameters();
        parameters.Add("name", dbName);

        using (var connection = _context.CreateMasterConnection())
        {
            var records = connection.Query(query, parameters);

            if (!records.Any())
                connection.Execute($"CREATE DATABASE {dbName}");
        }
    }
}
```

We have to add two additional namespaces as well:

```
using Dapper;
using Repository;
```

So, in our class, we first inject the DapperContext. Then, in the CreateDatabase method, we provide the name of the database we want to create. Also, we create an SQL query where we try to find the database with the provided name. After that, we use the DynamicParamters class to add our parameter to the dynamic list of parameters.

Once we have parameters prepared, we are creating our master connection and use it inside the `using` directive to call the `Query` method where we provide our `query` and `parameters` as arguments.

The `Query` method returns the collection as a result, and we check if it returns anything from the database. If not, we use the `connection` object to call the `Execute` method where we pass our query to create a database with a required name.

At this point, we have our class that will create a database if it doesn't already exist, and we can register this class as a service in the `Program` class:

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

To continue, we are going to create a new `MigrationManager` class in the `Extensions` folder in the main project:

```
public static class MigrationManager
{
    public static WebApplication MigrateDatabase(this WebApplication app,
        ILoggerManager logger)
    {
        using (var scope = app.Services.CreateScope())
        {
            var databaseService = scope.ServiceProvider
                .GetRequiredService<Database>();

            try
            {
                databaseService.CreateDatabase("CompanyEmployeeDapper");
            }
            catch(Exception ex)
            {
                logger.LogError($"Exception occurred during the database creation:
{ex}");
                throw;
            }
        }

        return app;
    }
}
```

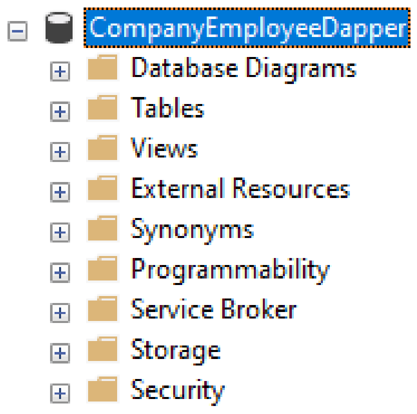
We create a `WebApplication` extension method where we get the Database service from a service provider and use it to call our `CreateDatabase` method.

Since we are extending the `WebApplication` class, we can call it inside the `Program.cs` class to ensure its execution as soon as our app starts:

```
app.MapControllers();

app.MigrateDatabase(logger);

app.Run();
```

Now, we can test this and create our database.

Let's start our application, and as soon as it starts successfully, we can verify that our database is created:



```
CompanyEmployeeDapper
    Database Diagrams
    Tables
    Views
    External Resources
    Synonyms
    Programmability
    Service Broker
    Storage
    Security
```

Of course, we don't have our tables and the data yet, so let's continue and add those missing parts.

3.2 Adding Tables To the Database

To start with adding tables to the database, we are going to create a new `InitialTables_202206160001` class under the `Migrations` folder. As you can see, we use the `{fileName_version}` pattern for the full file name:

```
[Migration(202206160001)]
public class InitialTables_202206160001 : Migration
{

}
```

We have to decorate our migration file with the **[Migration]** attribute from the **FluentMigrator** namespace and provide a version number as a parameter. Also, our class must derive from the abstract **Migration** class.

Now, we have to add two required methods inside:

```
public override void Down()
{
    Delete.Table("Employees");
    Delete.Table("Companies");
}

public override void Up()
{
    Create.Table("Companies")
        .WithColumn("CompanyId").AsGuid().NotNullable()
            .PrimaryKey().WithDefaultValue("NEWID()")
        .WithColumn("Name").AsString(50).NotNullable()
        .WithColumn("Address").AsString(60).NotNullable()
        .WithColumn("Country").AsString(50).NotNullable();

    Create.Table("Employees")
        .WithColumn("EmployeeId").AsGuid().NotNullable()
            .PrimaryKey().WithDefaultValue("NEWID()")
        .WithColumn("Name").AsString(50).NotNullable()
        .WithColumn("Age").AsInt32().NotNullable()
        .WithColumn("Position").AsString(50).NotNullable()
        .WithColumn("CompanyId").AsGuid().NotNullable()
            .ForeignKey("Companies", "CompanyId")
            .OnDelete(System.Data.Rule.Cascade);
}
```

The **Down** method gets executed if we revert this migration. It just removes our created tables.

The **Up** method will be executed as soon as we execute this migration. Here, we create two tables and provide columns for each table with different settings.

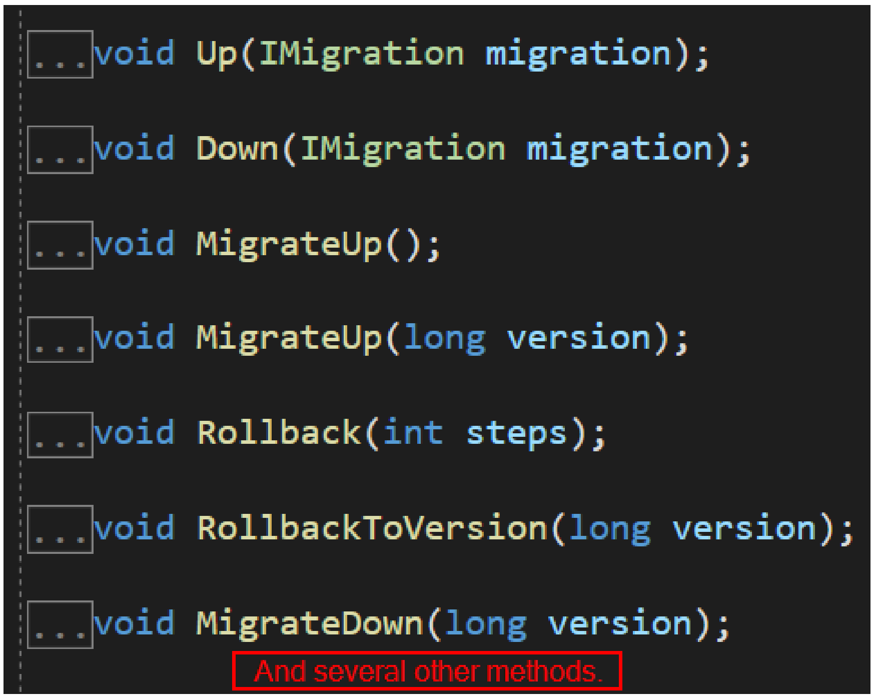
As you can see, the names of the methods describe what each method does, which is great. We use the **AsGuid**, **AsString**, and **AsInt32** methods to define the types of our columns. If we want to prevent null values for the columns, we use the **NotNullable** method. Also, we use the **WithDefaultValue** method to provide a default GUID value for our primary key columns. Finally, we use the **ForeignKey** method to create a relationship between required columns and the **OnDelete** method to specify whether we want to enable cascading delete or not. In this case, we do.

Now, to be able to start this migration as soon as our app starts, we have to modify the **MigrationManager** class:

```
public static WebApplication MigrateDatabase(this WebApplication app,
    ILoggerManager logger)
{
    using (var scope = app.Services.CreateScope())
    {
        var databaseService = scope.ServiceProvider
            .GetRequiredService<Database>();
        var migrationService = scope.ServiceProvider
            .GetRequiredService<IMigrationRunner>();

        try
        {
            databaseService.CreateDatabase("CompanyEmployeeDapper");

            migrationService.ListMigrations();
            migrationService.MigrateUp();
        }
        catch(Exception ex)
        {
            logger.LogError($"Exception occurred during the database creation: {ex}");
            throw;
        }
    }

    return app;
}
```

We are getting the **IMigrationRunner** service from the **FluentMigrator.Runner** namespace, and are using it to list all migrations and execute them.

Of course, these are just two of many methods that **IMigrationRunner** offers. If we inspect that interface, we can find all the other methods:



```
...void Up(IMigration migration);

...void Down(IMigration migration);

...void MigrateUp();

...void MigrateUp(long version);

...void Rollback(int steps);

...void RollbackToVersion(long version);

...void MigrateDown(long version);
```

The final thing we have to do is to configure **FluentMigrator** in the **ServiceExtensions** class:

```
public static void ConfigureFluentMigrator(this IServiceCollection services,
       IConfiguration configuration) => services.AddLogging(c =>
       c.AddFluentMigratorConsole())
              .AddFluentMigratorCore().ConfigureRunner(c =>
                     c.AddSqlServer2016().WithGlobalConnectionString(configuration
                                   .GetConnectionString("sqlConnection"))
                            .ScanIn(Assembly.GetExecutingAssembly())
                                   .For.Migrations());
```

We add the **FluentMigrator** logs to the console, configure the migration runner with the **AddFluentMigratorCore** method, and configure that runner with the **ConfigureRunner** method. We have to provide the SQL server support, the connection string, and the assembly to search types from.

Of course, here we need two additional namespaces:

```
using FluentMigrator.Runner;
using System.Reflection;
```

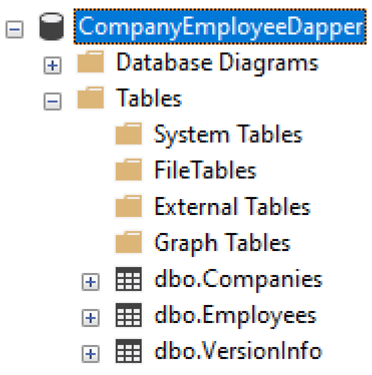
Now, all we have to do is to call this method in the **Program** class:

```
builder.Services.ConfigureFluentMigrator(builder.Configuration);
```

With this in place, we can start our app again.

Once it starts, if we inspect the console window, we are going to see different logs from FluentMigratior.

Also, we can refresh the already created database and find three tables inside:



FluentMigrator uses the VersionInfo table to keep track of migrations. So this table has the same purpose as the `EFMigrationsHistory` table that EF Core creates.

3.3 Seed Data in the Database Tables

Now, that we have our tables created, we want to populate them both.

To do that, let's create another class named `InitialSeed_202206160002` inside the `Migrations` folder:

```
[Migration(202206160002)]
public class InitialSeed_202206160002 : Migration
{

}
```

Now, let's add the `Up` method first:

```
public override void Up()
{
    Insert.IntoTable("Companies")
    .Row(new Company
    {
        CompanyId = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870"),
        Address = "583 Wall Dr. Gwynn Oak, MD 21207",
        Country = "USA",
        Name = "IT_Solutions Ltd"
    })
    .Row(new Company
    {
        CompanyId = new Guid("3d490a70-94ce-4d15-9494-5248280c2ce3"),
        Name = "Admin_Solutions Ltd",
        Address = "312 Forest Avenue, BF 923",
        Country = "USA"
    });

    Insert.IntoTable("Employees")
        .Row(new Employee
        {
            EmployeeId = new Guid("80abbca8-664d-4b20-b5de-024705497d4a"),
            Name = "Sam Raiden",
            Age = 26,
            Position = "Software developer",
            CompanyId = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870")
        })
        .Row(new Employee
        {
            EmployeeId = new Guid("86dba8c0-d178-41e7-938c-ed49778fb52a"),
            Name = "Jana McLeaf",
            Age = 30,
            Position = "Software developer",
            CompanyId = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870")
        })
        .Row(new Employee
        {
            EmployeeId = new Guid("021ca3c1-0deb-4afd-ae94-2159a8479811"),
            Name = "Kane Miller",
            Age = 35,
            Position = "Administrator",
            CompanyId = new Guid("3d490a70-94ce-4d15-9494-5248280c2ce3")
        });
}
```

Here, we insert two company rows and three employee rows. Two employees for the first company, and one employee for the second one.

Of course, as you expect, we have to add the **Down** method as well:

```
public override void Down()
{
    Delete.FromTable("Employees")
        .Row(new Employee
        {
            EmployeeId = new Guid("80abbca8-664d-4b20-b5de-024705497d4a"),
            Name = "Sam Raiden",
            Age = 26,
            Position = "Software developer",
            CompanyId = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870")
        })
        .Row(new Employee
        {
            EmployeeId = new Guid("86dba8c0-d178-41e7-938c-ed49778fb52a"),
            Name = "Jana McLeaf",
            Age = 30,
            Position = "Software developer",
            CompanyId = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870")
        })
        .Row(new Employee
        {
            EmployeeId = new Guid("021ca3c1-0deb-4afd-ae94-2159a8479811"),
            Name = "Kane Miller",
            Age = 35,
            Position = "Administrator",
            CompanyId = new Guid("3d490a70-94ce-4d15-9494-5248280c2ce3")
        });

    Delete.FromTable("Companies")
        .Row(new Company
        {
            CompanyId = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870"),
            Address = "583 Wall Dr. Gwynn Oak, MD 21207",
            Country = "USA",
            Name = "IT_Solutions Ltd"
        })
        .Row(new Company
        {
            CompanyId = new Guid("3d490a70-94ce-4d15-9494-5248280c2ce3"),
            Name = "Admin_Solutions Ltd",
            Address = "312 Forest Avenue, BF 923",
            Country = "USA"
        });
}
```

Here we use the same data, just a different method. Instead of the **`Insert`** method, we use **`Delete`**.

Of course, here we need two additional namespaces:

```
using Entities.Models;
using FluentMigrator;
```

And that’s all.

Again, all we have to do is to start our app, and we can find the data inserted in the tables.

With the database, tables, and data prepared, we can start adding repository files to the **`Repository`** project.

4 REPOSITORY AND SERVICE IMPLEMENTATION

If we inspect the `Repository` and the `Service` projects, we will find that they are missing the required files. So, in this section, we are going to add the repository and the service files in a similar fashion to what we did in the main book. Of course, we are not going to have the same repository structure because we don't need it here, but it will be pretty similar.

Also, since everything is well explained about the repository and the service layer in the main book, we will show only the implementation without any additional explanations.

That said, let's start with the required interfaces inside the `Contracts` project.

Let's first create the `ICompanyRepository` interface:

```
public interface ICompanyRepository
{
}
```

And then the `IEmployeeRepository` interface:

```
public interface IEmployeeRepository
{
}
```

After this, we can create repository user classes in the `Repository` project.

The first thing we are going to do is to create the `CompanyRepository` class:

```
public class CompanyRepository : ICompanyRepository
{
    private readonly DapperContext _context;

    public CompanyRepository(DapperContext context) => _context = context;
}
```

And then the `EmployeeRepository` class:

```
public class EmployeeRepository : IEmployeeRepository
{
    private readonly DapperContext _context;

    public EmployeeRepository(DapperContext context) => _context = context;
}
```

Now, as we did in the main book, we can create a new **IRepositoryManager** interface inside the **Contracts** project, to wrap the previous two interfaces:

```
public interface IRepositoryManager
{
    ICompanyRepository Company { get; }
    IEmployeeRepository Employee { get; }
}
```

After this interface, we are going to create the **RepositoryManager** class, in the **Repository** project, that implements the interface and provides an entry point for our repository actions:

```
public class RepositoryManager : IRepositoryManager
{
    private readonly DapperContext _dapperContext;
    private readonly Lazy<ICompanyRepository> _companyRepository;
    private readonly Lazy<IEmployeeRepository> _employeeRepository;

    public RepositoryManager(DapperContext dapperContext)
    {
        _dapperContext = dapperContext;
        _companyRepository = new Lazy<ICompanyRepository>(() => new
CompanyRepository(_dapperContext));
        _employeeRepository = new Lazy<IEmployeeRepository>(() => new
EmployeeRepository(_dapperContext));
    }

    public ICompanyRepository Company => _companyRepository.Value;
    public IEmployeeRepository Employee => _employeeRepository.Value;
}
```

After these changes, we need to register our manager class in the main project. So, let’s first modify the **ServiceExtensions** class by adding this code:

```
public static void ConfigureRepositoryManager(this IServiceCollection services) =>
       services.AddScoped<IRepositoryManager, RepositoryManager>();
```

And we have to call this method in the `Program` class:

```
builder.Services.ConfigureRepositoryManager();
```

With the repository files implemented, we can move on to the service layer implementation.

We are going to create three new interfaces inside the `Service.Contracts` project.

Let's start with the `ICompanyService` interface:

```
public interface ICompanyService
{
}
```

And then let's add another one named `IEmployeeService`:

```
public interface IEmployeeService
{
}
```

Finally, we are going to create the `IServiceManager` interface as a wrapper for the previous two interfaces:

```
public interface IServiceManager
{
    ICompanyService CompanyService { get; }
    IEmployeeService EmployeeService { get; }
}
```

Now, let's move on to the `Service` project. Here, we are going to follow the same pattern as for the repository project.

So, let's start by creating the `CompanyService` class:

```
internal sealed class CompanyService : ICompanyService
{
    private readonly IRepositoryManager _repository;
    private readonly ILoggerManager _logger;

    public CompanyService(IRepositoryManager repository, ILoggerManager logger)
    {
        _repository = repository;
        _logger = logger;
    }
}
```

Let’s do the same with the `EmployeeService` class:

```
internal sealed class EmployeeService : IEmployeeService
{
    private readonly IRepositoryManager _repository;
    private readonly ILoggerManager _logger;

    public EmployeeService(IRepositoryManager repository, ILoggerManager logger)
    {
        _repository = repository;
        _logger = logger;
    }
}
```

Finally, we will create the `ServiceManager` class as an entry point for our service functionality:

```
public sealed class ServiceManager : IServiceManager
{
    private readonly Lazy<ICompanyService> _companyService;
    private readonly Lazy<IEmployeeService> _employeeService;

    public ServiceManager(IRepositoryManager repositoryManager,
        ILoggerManager logger)
    {
        _companyService = new Lazy<ICompanyService>(() =>
            new CompanyService(repositoryManager, logger));
        _employeeService = new Lazy<IEmployeeService>(() =>
            new EmployeeService(repositoryManager, logger));
    }

    public ICompanyService CompanyService => _companyService.Value;
    public IEmployeeService EmployeeService => _employeeService.Value;
}
```

At this point, we are done with the service layer preparation. All we have to do now is to register this class as a service. To do that, let’s modify the `ServiceExtensions` class:

```
public static void ConfigureServiceManager(this IServiceCollection services) =>
       services.AddScoped<IServiceManager, ServiceManager>();
```

And call this method in the `Program` class:

```
builder.Services.ConfigureServiceManager();
```

That’s it. We are now ready to use Dapper to provide some data for our HTTP requests.

5 HANDLING GET REQUESTS

We already know everything we need to know about the controllers, routing, and naming of our resources from our main book. Also, we've seen how we can use EF Core to retrieve the data from the database.

So, since we have all that knowledge, in this chapter, we are going to show you how to replace the EF Core logic with Dapper to retrieve the same data from the database.

We are going to cover all the different GET requests that we've handled in the main book.

5.1 Getting All Companies From the Database

Let's add the first member inside the **ICompanyRepository** interface:

```
public interface ICompanyRepository
{
    Task<IEnumerable<Company>> GetAllCompanies();
}
```

Next, let's implement this method in the **CompanyRepository** class:

```
public async Task<IEnumerable<Company>> GetAllCompanies()
{
    var query = @"SELECT CompanyId, [Name], [Address], Country
                  FROM Companies
                  ORDER BY [Name]";

    using (var connection = _context.CreateConnection())
    {
        var companies = await connection.QueryAsync<Company>(query);

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

Here, we will do something similar to what we did when we created the database. We will create a **query** variable with the **SQL** statement. Then, using the **_context** object, we create a connection and store it inside the **connection** variable. After that, we call the **QueryAsync<Company>** method **from the Dapper namespace**, which returns a collection of

`Company` results, and we pass our `query` as an argument. We also need to add a using from the **`Entity.Models`** namespace

Finally, we convert the result to a list and return it.

The difference between this method and the one we used for the database creation is that here we are using the async Dapper method, which is also strongly typed.

> ***NOTE:*** *It is always a good idea to execute the SELECT statement in SQL Server Management Studio to verify that the statement returns the expected result.*

After the repository implementation, we have to implement a service layer.

Let's start with the `ICompanyService` interface modification:

```
public interface ICompanyService
{
    Task<IEnumerable<Company>> GetAllCompanies();
}
```

Here, we have to add the reference from the `Entities` project, but soon enough we will remove it.

Then, let's continue with the `CompanyService` modification:

```
public async Task<IEnumerable<Company>> GetAllCompanies()
{
    var companies = await _repository.Company.GetAllCompanies();

    return companies;
}
```

This is pretty straightforward logic where we just call our created method from the repository and return the result.

One thing to notice here is that we are not using any `try-catch` blocks because our initial project came with the global exception handler already implemented.

We already have the controller prepared in the `Controllers` folder in the `CompanyEmployees.Presentation` project:

```
[Route("api/companies")]
[ApiController]
public class CompaniesController : ControllerBase
{

}
```

Let's modify the `CompaniesController` and add a required action:

```
[Route("api/companies")]
[ApiController]
public class CompaniesController : ControllerBase
{
    private readonly IServiceManager _service;

    public CompaniesController(IServiceManager service) => _service = service;

    [HttpGet]
    public async Task<IActionResult> GetCompanies()
    {
        var companies = await _service.CompanyService.GetAllCompanies();

        return Ok(companies);
    }
}
```

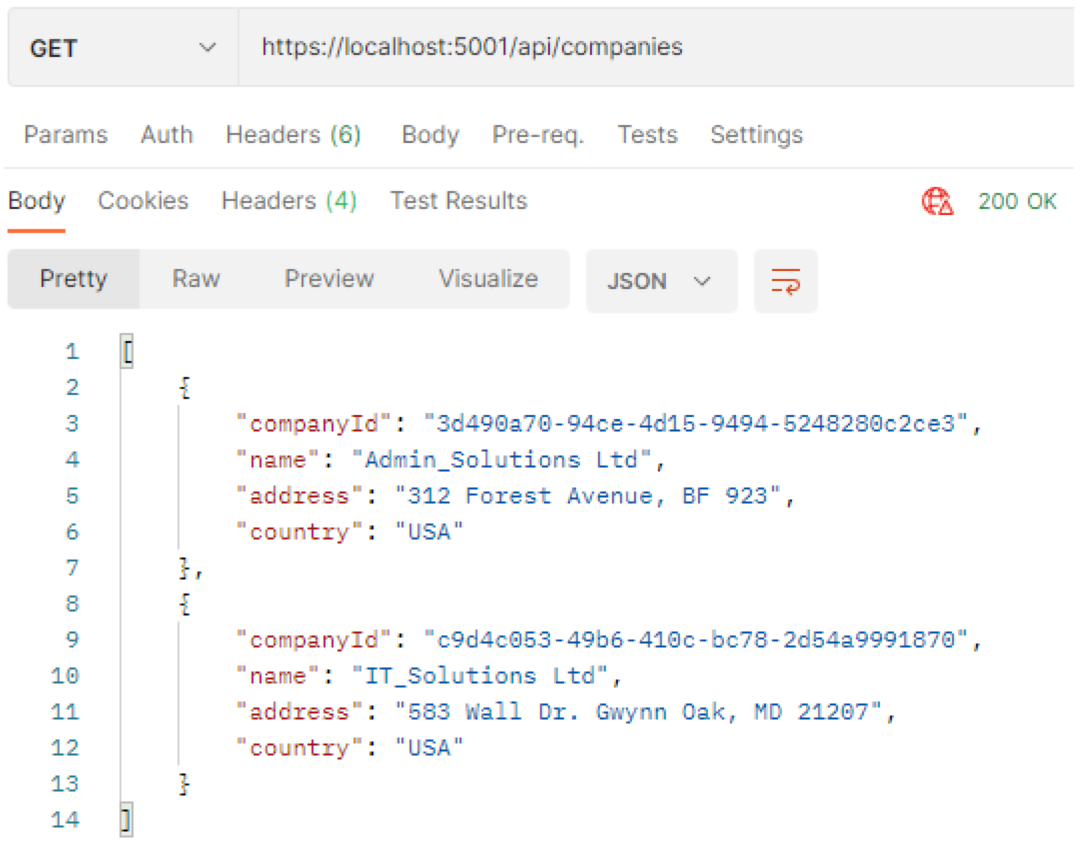
We inject our service with the constructor injection and use it to call the `GetAllCompanies` method. Then, we just return the `Ok` result where we provide the company collection as a response body. We have to add the reference from the `Service.Contracts` project

5.1.1 Testing the Result with Postman

We can use the same postman collection that we've provided as `Bonus 2-CompanyEmployeesRequests.postman_collection.json` to test our endpoint. This test request is in the **`04-Handling GET Requests folder`**.

So, let's send the first GET request:

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



```
[
    {
        "companyId": "3d490a70-94ce-4d15-9494-5248280c2ce3",
        "name": "Admin_Solutions Ltd",
        "address": "312 Forest Avenue, BF 923",
        "country": "USA"
    },
    {
        "companyId": "c9d4c053-49b6-410c-bc78-2d54a9991870",
        "name": "IT_Solutions Ltd",
        "address": "583 Wall Dr. Gwynn Oak, MD 21207",
        "country": "USA"
    }
]
```

Excellent, everything is working as planned. But we are missing something. We are using the `Company` entity to map our requests to the database and then returning it as a result to the client, and this is not a good practice. So, in the next part, we are going to learn how to improve our code with DTO classes.

5.2 Using DTOs to Return Results

In the main book, we have used the DTOs to map our results inside the service layer and then return them to the controller. For the mapping actions, we've used AutoMapper.

So, if you want the same flow, you can install AutoMapper, configure the mapping profile class, and then use it to map results inside the Service layer.

But, since Dapper can map results for us, we don’t have to use AutoMapper – at least not for mapping repository results. If you don’t want to return an Entity from the repository but, instead, return the DTO, we can use Dapper to map that results for us. Thus, we will return the DTO straight from the Repository layer.

Either way, you choose, you won’t be wrong. But in this book, we are going to use the second approach and return the DTO results from the Repository layer.

That said, let’s start by creating a new **CompanyDto** record inside the **Shared** project in the **DataTransferObjects** folder:

```
public record CompanyDto(Guid CompanyId, string Name, string FullAddress);
```

After the record creation, we are going to modify the **ICompanyRepository** interface:

```
Task<IEnumerable<CompanyDto>> GetAllCompanies();
```

Here, we are using **CompanyDto** instead of **Company** as a return type, so we have to add the reference from the **Shared** project We can now remove the reference to the **Entities** project from the **Contracts** project.

Then, we have to modify the **GetAllCompanies** method inside the repository class:

```
public async Task<IEnumerable<CompanyDto>> GetAllCompanies()
{
    var query = @"SELECT CompanyId, [Name], CONCAT([Address], ', ', Country) AS 
FullAddress
                  FROM Companies
                  ORDER BY [Name]";

    using (var connection = _context.CreateConnection())
    {
        var companies = await connection.QueryAsync<CompanyDto>(query);

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

Here we change the return type for our method and the `QueryAsync` method as well. Also, we modify the SQL query to concat `Address` and `Country` columns as `FullAddress`.At this point, the result from our `SQL` query has the same columns as the properties that we have in the `CompanyDto` record. This means the Dapper will easily map the result to the required type.

To continue, we have to modify the `IComanyService` interface:

```
public interface ICompanyService
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
}
```

Since we are not using the Company entity anymore, we can safely remove the Entities' project reference from the `Service.Contracts` project.

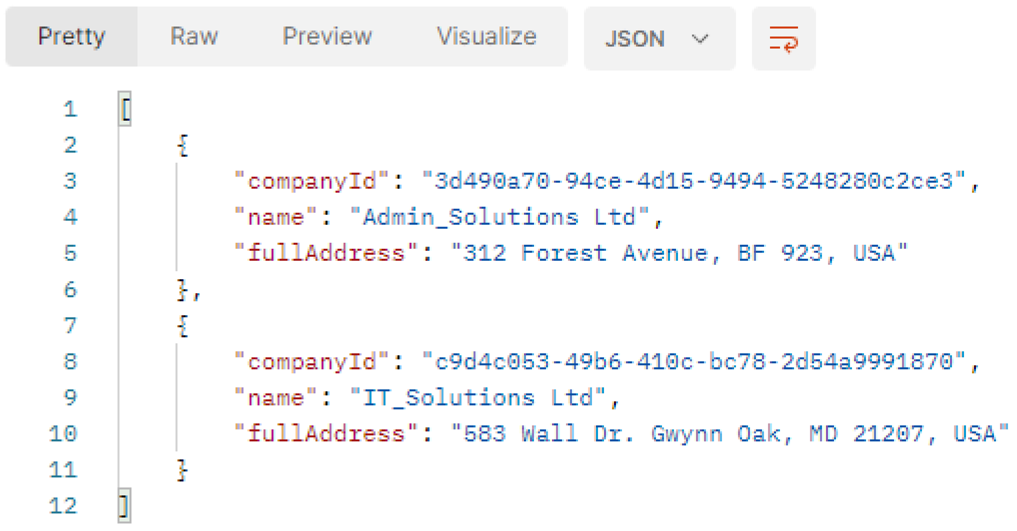
Lastly, let's modify the required method inside the service class:

```
public async Task<IEnumerable<CompanyDto>> GetAllCompanies()
{
    var companies = await _repository.Company.GetAllCompanies();

    return companies;
}
```

And that's all.

We can now send the same GET request from Postman:

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



```
[
    {
        "companyId": "3d490a70-94ce-4d15-9494-5248280c2ce3",
        "name": "Admin_Solutions Ltd",
        "fullAddress": "312 Forest Avenue, BF 923, USA"
    },
    {
        "companyId": "c9d4c053-49b6-410c-bc78-2d54a9991870",
        "name": "IT_Solutions Ltd",
        "fullAddress": "583 Wall Dr. Gwynn Oak, MD 21207, USA"
    }
]
```

This time, we get the CompanyDto result.

5.3 Different Property and Column Names

In our previous example, we are returning, as a result, the columns that are named the same as the properties we have in our **CompanyDto** record.

But what would happen if those don't match?

Well, let's modify the **Name** property to the **CompanyName** in the **CompanyDto** record:

```
public record CompanyDto(Guid CompanyId, string CompanyName, string FullAddress);
```

Now, if we run our project and send the same GET request, we will get an error. And if we inspect our console logs, we will see this error:

```
A parameterless default constructor or one matching signature (System.Guid CompanyId,
System.String Name, System.String FullAddress) is required for
Shared.DataTransferObjects.CompanyDto materialization
```

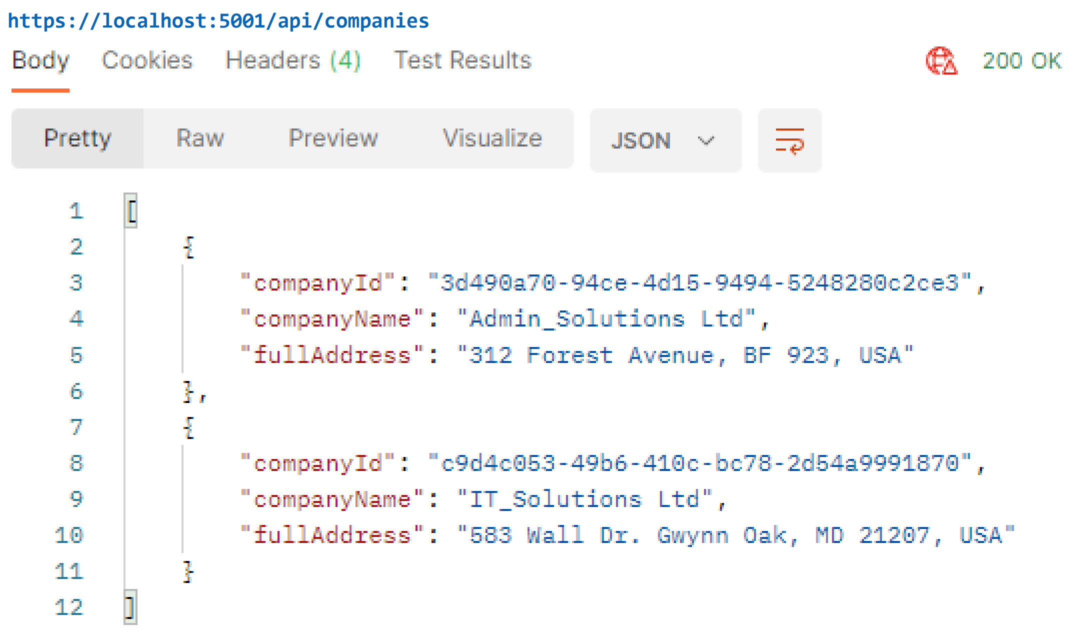
So, Dapper is unable to map our result to the required type. But the solution is pretty simple.

All we have to do is to modify the SQL query in the **GetAllCompanies** method:

```
var query = @"SELECT CompanyId, [Name] AS CompanyName,
            CONCAT([Address], ', ', Country) AS FullAddress
            FROM Companies
            ORDER BY [Name]";
```

As you can see, all we have to do is just rename the Name column to the CompanyName in the query.

Now, if we send the same request, we are going to get our result:



```
[
    {
        "companyId": "3d490a70-94ce-4d15-9494-5248280c2ce3",
        "companyName": "Admin_Solutions Ltd",
        "fullAddress": "312 Forest Avenue, BF 923, USA"
    },
    {
        "companyId": "c9d4c053-49b6-410c-bc78-2d54a9991870",
        "companyName": "IT_Solutions Ltd",
        "fullAddress": "583 Wall Dr. Gwynn Oak, MD 21207, USA"
    }
]
```

This time the mapping works perfectly.

We just wanted to turn your attention to this if you face any similar situation in your projects.

That said, let’s return all the names as they were before the modification for the query and the `CompanyDto` record.

5.4 Extracting SQL Query

The last thing we are going to do in this chapter is to extract our query to a separate class. Often we can find ourselves in a situation where we have to reuse the same SQL statement, so it is better if we place all the SQL statements in a single file and then just use them from there. Also, if we have to modify any of these statements, we can do that only in a single place.

That said, let's create a new `Queries` folder inside the `Repository` project. Inside that folder, we are going to create a new `CompanyQuery` class and transfer or SQL statement from the repository class:

```
public static class CompanyQuery
{
    public const string SelectCompanyQuery =
        @"SELECT CompanyId, [Name],
          CONCAT([Address], ', ', Country) AS FullAddress
          FROM Companies
          ORDER BY [Name]";
}
```

And then, just modify the GetAllCompanies method:

```
public async Task<IEnumerable<CompanyDto>> GetAllCompanies()
{
    var query = CompanyQuery.SelectCompanyQuery;

    using (var connection = _context.CreateConnection())
    {
        var companies = await connection.QueryAsync<CompanyDto>(query);

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

We will have to add a using for `Repostory.Queries`. With this out of the way, we can move on to the next chapter.

6 GETTING ADDITIONAL RESOURCES

In this chapter, we are going to add additional actions to our controller. Moreover, we are going to create one more controller for the Employee resource and implement an additional action in it.

6.1 Getting a Single Resource From the Database

Here, we are going to show you how to get a single company with the Dapper query and of course, how to use parameters for the queries. We've already seen, in the database creation example, how we can use dynamic parameters. But in this example, we will use anonymous parameters to show you another way to add parameters to our queries.

That said, let's start with the `ICompanyRepository` modification:

```
public interface ICompanyRepository
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
}
```

Next, let's add another `SELECT` statement in the `CompanyQuery` class:

```
public const string SelectCompanyByIdQuery =
    @"SELECT CompanyId, [Name],
      CONCAT([Address], ', ', Country) AS FullAddress
      FROM Companies
      WHERE CompanyId = @companyId";
```

Here, we have almost the same query as the previous one, but this time, we have to use the `WHERE` clause to specify that we want a specific company with the provided `@companyId` parameter.

As soon as we have our query ready, we can implement the missing member in the `CompanyRepository` class:

```
public async Task<CompanyDto> GetCompany(Guid id)
{
    var query = CompanyQuery.SelectCompanyByIdQuery;

    using (var connection = _context.CreateConnection())
    {
        var company = await connection
            .QuerySingleOrDefaultAsync<CompanyDto>(query, new { companyId = 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. Pay attention that if we want our anonymous object to be a valid parameter, **we must have the property with the same name as the parameter in the `SELECT` statement**.

Now, we can modify the `ICompanyService` interface:

```
public interface ICompanyService
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
}
```

And also the `CompanyService` class:

```
public async Task<CompanyDto> GetCompany(Guid id)
{
    var company = await _repository.Company.GetCompany(id);
    if (company is null)
        throw new CompanyNotFoundException(id);

    return company;
}
```

There is nothing new here that we didn't see in our main book.

Also, now we will have an error because we don't have this `CompanyNotFoundException` class implemented. Again, we don't want to repeat all the implementations from the main book, so you can just copy an entire `Exceptions` folder from the source code for this chapter, paste it inside the `Entities` project, and add a reference to the `Entities` project in this project.

Also, we have to modify the `ExceptionMiddlewareExtensions` class in the main project to handle a new exception:

```
if (contextFeature != null)
{
        context.Response.StatusCode = contextFeature.Error switch
        {
                NotFoundException => StatusCodes.Status404NotFound,
                _ => StatusCodes.Status500InternalServerError
        };

        logger.LogError($"Something went wrong: {contextFeature.Error}");

        await context.Response.WriteAsync(new ErrorDetails()
        {
                StatusCode = context.Response.StatusCode,
                Message = contextFeature.Error.Message
        }.ToString());
}
```

Next, add a using for `Entities.Exceptions`.

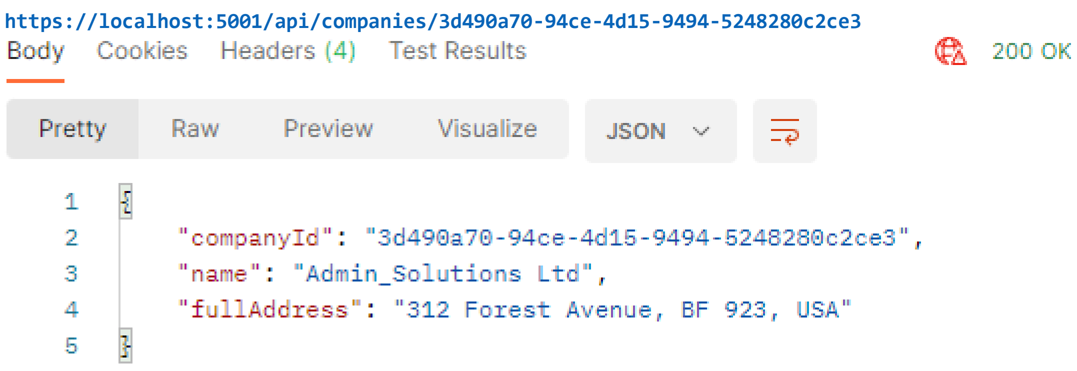
```
using Entities.Exceptions;
```

After that, we can add another action in the `CompaniesController`:

```
[HttpGet("{id:guid}")]
public async Task<IActionResult> GetCompany(Guid id)
{
    var company = await _service.CompanyService.GetCompany(id);

    return Ok(company);
}
```

At this point, we can test our app.

We will again use the GetCompany Postman request that we already prepared for you in our bonus 2 file from the `06-Getting Additional Resources` folder:


https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3

```
{
    "companyId": "3d490a70-94ce-4d15-9494-5248280c2ce3",
    "name": "Admin_Solutions Ltd",
    "fullAddress": "312 Forest Avenue, BF 923, USA"
}
```

We can see it works great.

6.2 Parent/Child Relationships with Dapper

The first thing, we are going to do here is to create a new `EmployeesController` in the `Presentation` project:

```
[Route("api/companies/{companyId}/employees")]
[ApiController]
public class EmployeesController : ControllerBase
{
    private readonly IServiceManager _service;

    public EmployeesController(IServiceManager service) => _service = service;
}
```

Add the required using statements to the controller.

For the first example, we are going to return all the employees for a single company.

Let's start with the `EmployeDto` record creation in the `Shared` project:

```
public record EmployeeDto(Guid EmployeeId, string Name, int Age, string Position);
```

Next, let's continue with editing the `IEmployeeRepository` interface, and adding the using statement for the `Shared.DataTransferObjects` project:

```
public interface IEmployeeRepository
{
    Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId);
}
```

Then, we have to create a new `EmployeeQuery` class, inside the `Repository/Queries` folder, and add a new SQL statement:

```
public static class EmployeeQuery
{
    public const string SelectEmployeesQuery =
        @"SELECT EmployeeId, [Name], Age, Position
          FROM Employees
          WHERE CompanyId = @companyId";
}
```

There is nothing new about this statement that we didn't see in the previous one.

After this, we can add a missing member inside the `EmployeeRepository` class:

```
public async Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId)
{
    var query = EmployeeQuery.SelectEmployeesQuery;

    using (var connection = _context.CreateConnection())
    {
        var employees = await connection
            .QueryAsync<EmployeeDto>(query, new { companyId });

        return employees.ToList();
    }
}
```

Here, once we have our `query` and created connection, we use the `connection` to call the `QueryAsync` method, map our result to the `EmployeeDto` type, and pass the `query` and new anonymous object as the method arguments. Finally, we convert the result to a list and return it.

Pay attention that this time we provide just the `companyId` as the property in our anonymous object, we don't have to write `companyId = companyId` (similarly to what we had in our previous method). That's because the `companyId` parameter that we accept in the repository method has the same name as the parameter in our SQL statement. Add using statements for `Contracts`, `Dapper`, `Repository.Queries` and `Shared.DataTransferObjects`.

To continue, we have to modify the `IEmployeeService` interface:

```
public interface IEmployeeService
{
    Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId);
}
```

Add a using statement for `Shared.DataTransferObjects`.

```
using Shared.DataTransferObjects;
```

And also the `EmployeeService` class:

```
public async Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId)
{
    var company = await _repository.Company.GetCompany(companyId);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

    var employees = await _repository.Employee.GetEmployees(companyId);

    return employees;
}
```

Add using statements for `Entities.Exceptions` and `Shared.DataTransferObjects`.

```
using Entities.Exceptions;
using Shared.DateTransferObjects;
```

Again, both are pretty familiar from our main book's example.

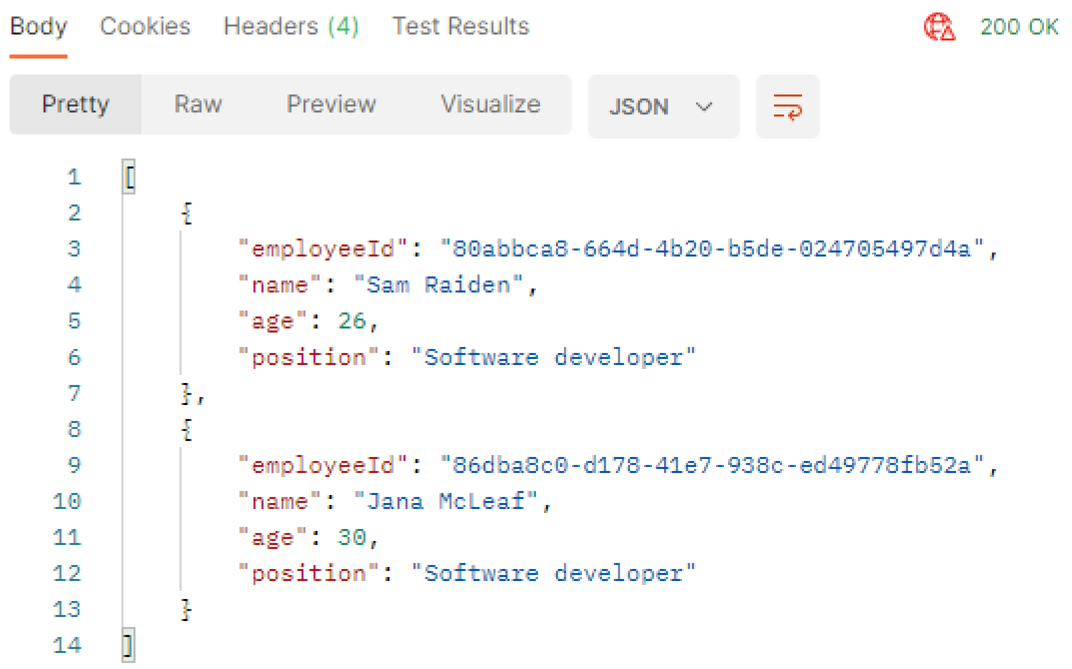
The last thing, we have to do, is to add a new action in the `EmployeesController`:

```
[HttpGet]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId)
{
    var employees = await _service.EmployeeService.GetEmployees(companyId);

    return Ok(employees);
}
```

That's all it takes.

Now, if we send a Postman request to test this using the Employees per Company request that we already prepared for you in our bonus 2 file from the 06-Getting Additional Resources folder:

`https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees`



```
[
    {
        "employeeId": "80abbca8-664d-4b20-b5de-024705497d4a",
        "name": "Sam Raiden",
        "age": 26,
        "position": "Software developer"
    },
    {
        "employeeId": "86dba8c0-d178-41e7-938c-ed49778fb52a",
        "name": "Jana McLeaf",
        "age": 30,
        "position": "Software developer"
    }
]
```

We can see our employees for the specific company.

6.3 Returning a Parent with All the Children with Dapper

In our previous example, we've fetched all the employees for a single company but without that company as part of the result. So, in this section, we will include the company in the result and learn how we can handle such a request with Dapper. You will see that the process is a bit more complicated than the ones we've seen until now.

That said, let's create a new record in the same location all the other records are placed:

```
public record CompanyWithEmployeesDto
{
    public Guid CompanyId { get; init; }
    public string? Name { get; init; }
    public string? FullAddress { get; init; }
    public List<EmployeeDto> Employees { get; init; } = new List<EmployeeDto>();
};
```

Then, we can modify the `ICompanyRepository` interface:

```
public interface ICompanyRepository
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
}
```

Before we implement this new member, we are going to add a new query in the `CompanyQuery` class:

```
public const string SelectCompaniesWithEmployeesQuery =
    @"SELECT c.CompanyId, c.[Name],
      CONCAT(c.[Address], ', ', c.Country) AS FullAddress,
      e.EmployeeId, e.[Name], e.Age, e.Position
      FROM Companies c JOIN Employees e ON c.CompanyId = e.CompanyId";
```

This is an SQL statement where we return all the companies including all the employees for a specific company.

Now, we can move on to the `CompanyRepository` class:

```
public async Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees()
{
    var query = CompanyQuery.SelectCompaniesWithEmployeesQuery;

    using (var connection = _context.CreateConnection())
    {
        var companyDict = new Dictionary<Guid, CompanyWithEmployeesDto>();

        var companies = await connection.QueryAsync<CompanyWithEmployeesDto,
EmployeeDto, CompanyWithEmployeesDto>(
            query, (company, employee) =>
            {
                if (!companyDict.TryGetValue(company.CompanyId, out var
currentCompany))
                {
                    currentCompany = company;
                    companyDict.Add(currentCompany.CompanyId, currentCompany);
                }

                currentCompany.Employees.Add(employee);

                return currentCompany;
            }, splitOn: "CompanyId, EmployeeId"
        );

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

There is quite a bit of logic here, so let’s explain it.

So, we get a query, and inside the using statement create a new connection. Then, we create a new dictionary to keep our companies in.

To extract the data from the database, we are using the `QueryAsync` method, but this time it has a new syntax we haven't seen so far.

We can see three generic types. The first two are the input types we are going to use as inputs, and the third one is the return type. This method accepts our query as a parameter, and also a `Func` delegate that accepts two parameters. These parameters have the same type as the first two input types we used in the `QueryAsync` method. Inside the delegate, we try to extract a company by its id. 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 as a result.

Also, you see that `splitOn` parameter. Dapper needs to know how to split our results into multiple objects. So, we have to provide these two parameters.

After our mapping is done, we just return a distinct result converted to a list.

After the repository implementation, we can move on to the service layer:

```
public interface ICompanyService
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
}
```

Let's implement this missing member in the `CompanyService` class:

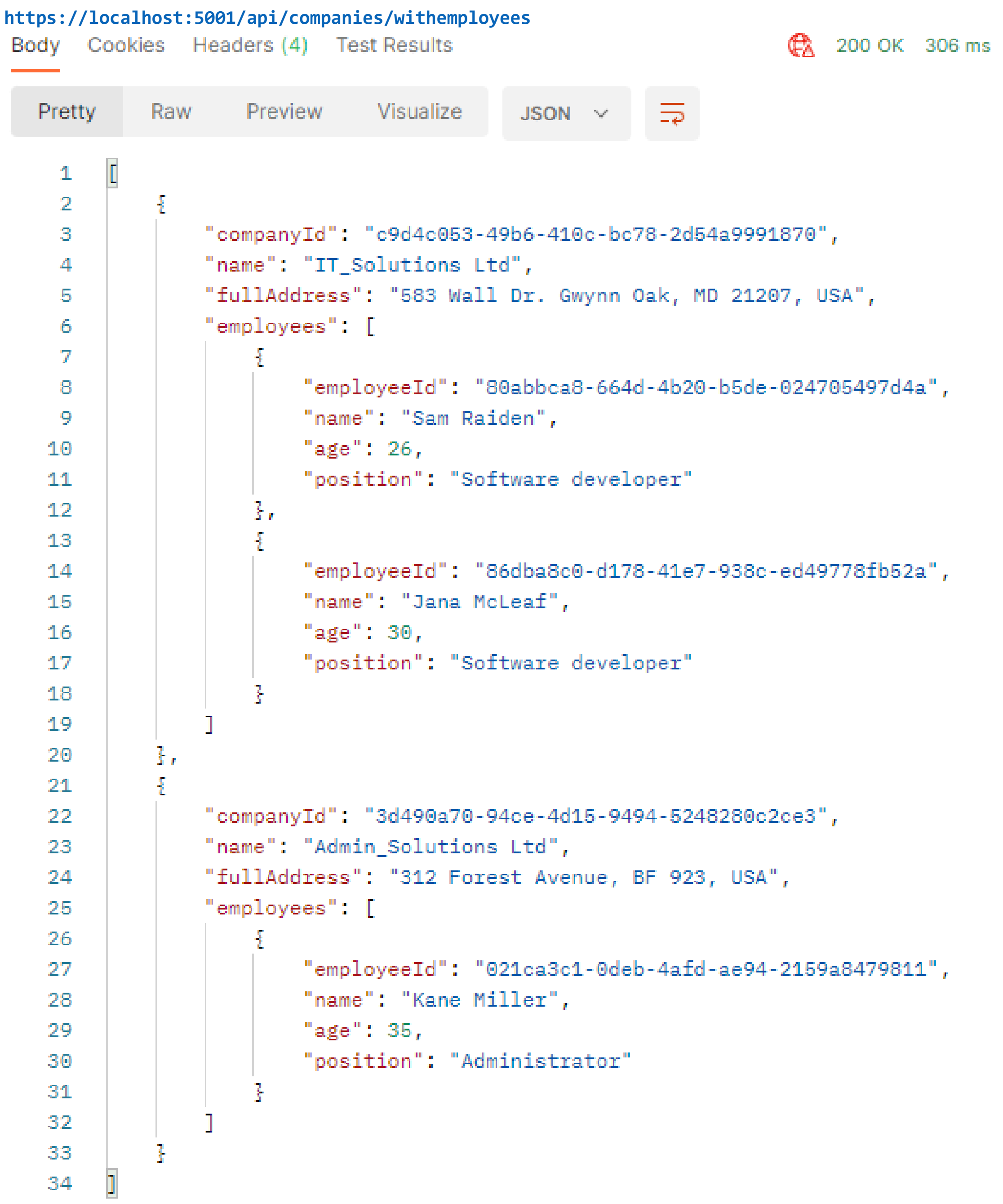
```
public async Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees()
{
    var companies = await _repository.Company.GetCompaniesWithEmployees();

    return companies;
}
```

Finally, let's add a new action in the controller:

```
[HttpGet("withEmployees")]
public async Task<IActionResult> GetCompaniesWithEmployees()
```

```
{
    var companies = await _service.CompanyService.GetCompaniesWithEmployees();

    return Ok(companies);
}
```

So, with all this in place, we can create and send a Postman request:



```
https://localhost:5001/api/companies/withemployees
```

```
[
    {
        "companyId": "c9d4c053-49b6-410c-bc78-2d54a9991870",
        "name": "IT_Solutions Ltd",
        "fullAddress": "583 Wall Dr. Gwynn Oak, MD 21207, USA",
        "employees": [
            {
                "employeeId": "80abbca8-664d-4b20-b5de-024705497d4a",
                "name": "Sam Raiden",
                "age": 26,
                "position": "Software developer"
            },
            {
                "employeeId": "86dba8c0-d178-41e7-938c-ed49778fb52a",
                "name": "Jana McLeaf",
                "age": 30,
                "position": "Software developer"
            }
        ]
    },
    {
        "companyId": "3d490a70-94ce-4d15-9494-5248280c2ce3",
        "name": "Admin_Solutions Ltd",
        "fullAddress": "312 Forest Avenue, BF 923, USA",
        "employees": [
            {
                "employeeId": "021ca3c1-0deb-4afd-ae94-2159a8479811",
                "name": "Kane Miller",
                "age": 35,
                "position": "Administrator"
            }
        ]
    }
]
```

And we can see our expected result.

6.4 Getting a Single Employee For a Company

So, as we did in previous sections, let's start with the `IEmployeeRepository` interface:

```
public interface IEmployeeRepository
{
    Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId);
    Task<EmployeeDto> GetEmployee(Guid companyId, Guid id);
}
```

Then, let's add a new query in the `EmployeeQuery` class:

```
public const string SelectEmployeeByIdAndCompanyIdQuery =
    @"SELECT EmployeeId, [Name], Age, Position
      FROM Employees
      WHERE EmployeeId = @id AND CompanyId = @companyId";
```

For this query, we accept two parameters because we want to return a single employee for a single-related company.

Next, we are going to add a new method in the `EmployeeRepository` class:

```
public async Task<EmployeeDto> GetEmployee(Guid companyId, Guid id)
{
    var query = EmployeeQuery.SelectEmployeeByIdAndCompanyIdQuery;

    using (var connection = _context.CreateConnection())
    {
        var param = new { companyId, id };
        var employee = await connection
            .QuerySingleOrDefaultAsync<EmployeeDto>(query, param);

        return employee;
    }
}
```

This is already familiar to us except we are using two parameters with the same names as the `SELECT` statement's parameters.

Now, we can move on and modify the `IEmployeeService` interface:

```
public interface IEmployeeService
{
    Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId);
    Task<EmployeeDto> GetEmployee(Guid companyId, Guid id);
}
```

And the `EmployeeService` class:

```
public async Task<EmployeeDto> GetEmployee(Guid companyId, Guid id)
{
    var company = await _repository.Company.GetCompany(companyId);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

    var employee = await _repository.Employee.GetEmployee(companyId, id);
    if (employee is null)
        throw new EmployeeNotFoundException(id);

    return employee;
}
```

We have already copied both the exception classes in our project, so we are good here.

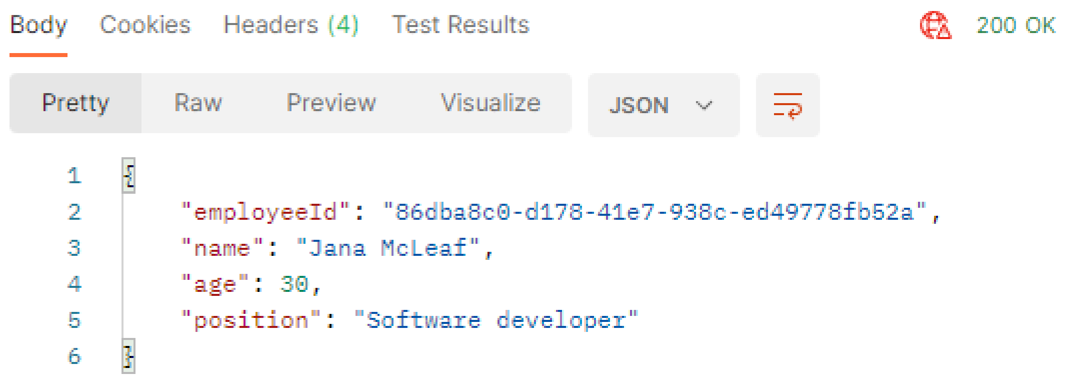
Finally, we have to add a new action in the `EmployeesController`:

```
[HttpGet("{id:guid}")]
public async Task<IActionResult> GetEmployeeForCompany(Guid companyId, Guid id)
{
    var employee = await _service.EmployeeService.GetEmployee(companyId, id);

    return Ok(employee);
}
```

We accept the `companyId` from the main URI and the `id` parameter from the action URI and then pass them both to the service method.

Now, we can send a Postman request using the Employee For Company request that we already prepared for you in our bonus 2 file from the `06-Getting Additional Resources` folder:

https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/86dba8c0-d178-41e7-938c-ed49778fb52a



```
{
    "employeeId": "86dba8c0-d178-41e7-938c-ed49778fb52a",
    "name": "Jana McLeaf",
    "age": 30,
    "position": "Software developer"
}
```

And, we can see our result, which means that everything works great.

7 CREATING RESOURCES

In this section, we are going to show you how to use Dapper to create resources in our API project. Since we already know everything we need to know about the HTTP POST requests (from the main book), let's head straight to the implementation.

7.1 Creating a Parent Resource

First, let's modify the decorator attribute for the **GetCompany** action in the **Companies** controller:

```
[HttpGet("{id:guid}", Name = "CompanyById")]
```

Next, since we want to have our DTO object for creation, we are going to create a new **CompanyForCreationDto** record in the **Shared/DataTransferObjects** folder:

```
public record CompanyForCreationDto(string Name, string Address, string Country);
```

Now, we can add another member to the **ICompanyRepsitory** interface:

```
public interface ICompanyRepository
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
}
```

As usual, we need another query in the **CompanyQuery** class:

```
public const string InsertCompanyQuery =
    @"INSERT INTO Companies (CompanyId, [Name], [Address], Country)
      OUTPUT inserted.CompanyId
      VALUES(default, @name, @address, @country);";
```

Here, we use the **INSERT** statement to insert our values into the required columns of the **Companies** table. But as you can see, we are using the **OUTPUT** clause which will return the id of the created (inserted) row. Also, we use the **default** as a value for the **companyId** column.

If we had an **int** type for the primary key, we could've used the **SELECT CAST(SCOPE_IDENTITY() as int)** expression after the **VALUES** part of the statement, which would return the **int** id of the inserted row.

With that said, let's add a missing method to the repository class:

```
public async Task<CompanyDto> CreateCompany(CompanyForCreationDto company)
{
    var query = CompanyQuery.InsertCompanyQuery;

    var param = new DynamicParameters(company);

    using (var connection = _context.CreateConnection())
    {
        var id = await connection.QuerySingleAsync<Guid>(query, param);

        return new CompanyDto(id, company.Name,
            string.Join(", ", company.Address, company.Country));
    }
}
```

Here, we assign the insert query to the **query** variable and then create parameters by using the **DynamicParameters** class. Since our **CompanyForCreationDto** parameter has the same property names as the columns we want to populate in the table, we can just add it as a parameter to the **DynamicParameters** class. You can also see that we don't provide a primary key as a parameter because it will be automatically created in the database. We've configured that in our migration for creating tables.

One more thing to mention here, if you want to be more explicit about your parameters, you can use the **Add** method to add them:

```
var param = new DynamicParameters();
param.Add("name", company.Name, DbType.String);
param.Add("address", company.Address, DbType.String);
param.Add("country", company.Country, DbType.String);
```

For this example, we will stick with the previous implementation.

After we define our parameters, we create a connection and use the **QuerySingleAsync<Guid>** method to execute our query. We are using this method, and not the ExecuteAsync method because we are returning a value from our INSERT statement.

Once we have our id value back, we return a new **CompanyDto** object as a result.

That's it regarding our repository method.

Now, we are going to modify the **ICompanyService** interface:

```
public interface ICompanyService
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
}
```

And also the **CompanyService** class:

```
public async Task<CompanyDto> CreateCompany(CompanyForCreationDto company)
{
    var companyToReturn = await _repository.Company.CreateCompany(company);

    return companyToReturn;
}
```

Finally, let's implement a required action in our controller:

```
[HttpPost]
public async Task<IActionResult> CreateCompany([FromBody] CompanyForCreationDto
company)
{
    if (company is null)
        return BadRequest("CompanyForCreationDto object is null");

    var createdCompany = await _service.CompanyService.CreateCompany(company);

    return CreatedAtRoute("CompanyById",
        new { id = createdCompany.CompanyId }, createdCompany);
}
```

This is, of course, a familiar implementation where we fetch the result from our service method and then point to the action where we can fetch a newly created company.

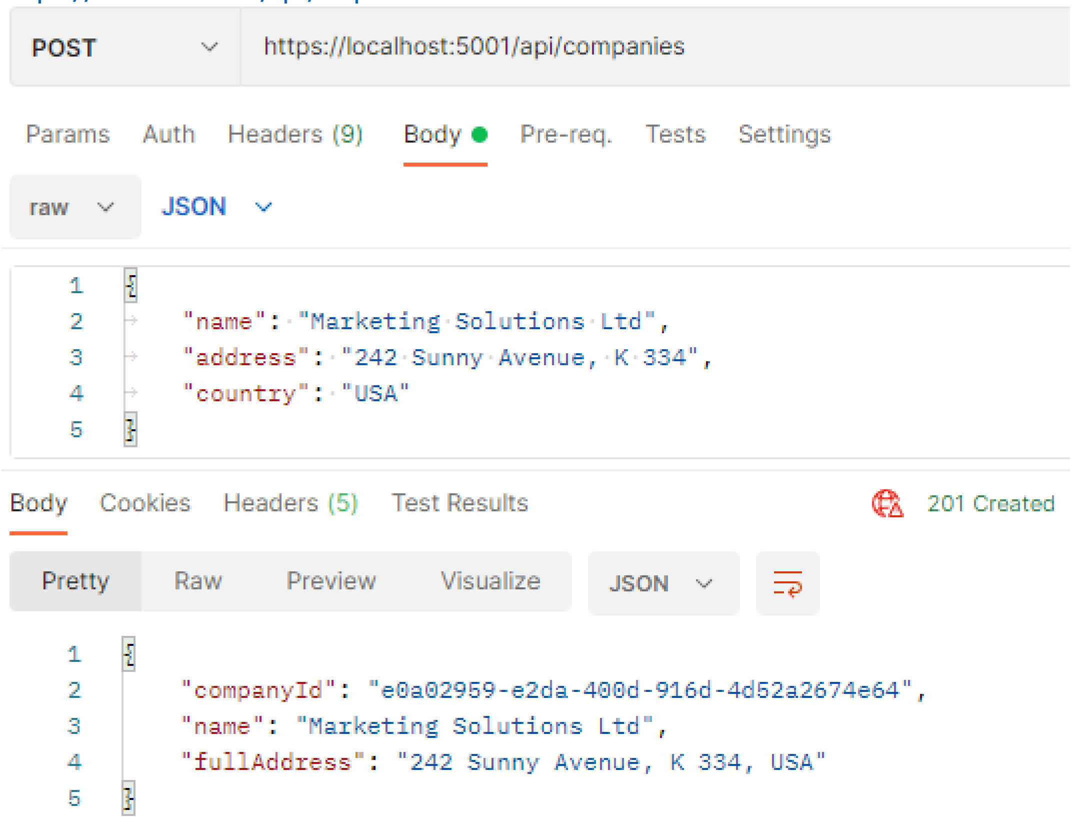
We need to add a using statement to the **Shared/DataTransferObjects** project.

```
using Shared.DataTransferObjects;
```

Of course, we will not deal with model validations or action filters, because all of that is already explained in the main book.

So, let's test it by sending a POST request using the POST Company request that we already prepared for you in our bonus 2 file from the `09-Creating Resources` folder:

https://localhost:5001/api/companies



https://localhost:5001/api/companies

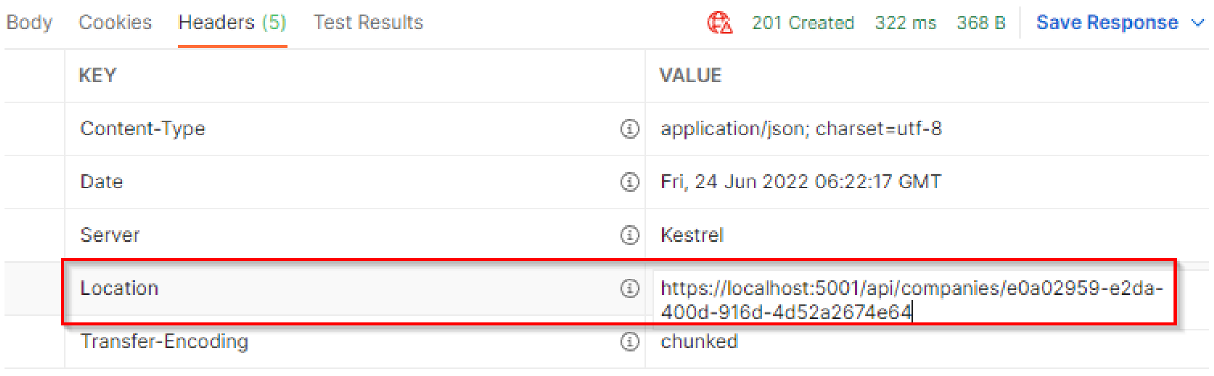
```
{
    "name": "Marketing Solutions Ltd",
    "address": "242 Sunny Avenue, K 334",
    "country": "USA"
}
```

201 Created

```
{
    "companyId": "e0a02959-e2da-400d-916d-4d52a2674e64",
    "name": "Marketing Solutions Ltd",
    "fullAddress": "242 Sunny Avenue, K 334, USA"
}
```

We can see that we get the `201 Created` result, and also a newly created company in the response body.

Of course, if we check the Headers tab, we will find the Location header:



KEY	VALUE
Content-Type	application/json; charset=utf-8
Date	Fri, 24 Jun 2022 06:22:17 GMT
Server	Kestrel
Location	https://localhost:5001/api/companies/e0a02959-e2da-400d-916d-4d52a2674e64
Transfer-Encoding	chunked

You can always use that URI from the Location header and send the GET request to verify that the company is created.

7.2 Creating a Child Resource

While creating our company, we created the DTO object required for the CreateCompany action. So, for employee creation, we are going to do the same thing by creating a new **EmployeeForCreationDto** record in the **Shared/DataTransferObjects** folder:

```
public record EmployeeForCreationDto(string Name, int Age, string Position);
```

The next step is to modify the **IEmployeeRepository** interface:

```
public interface IEmployeeRepository
{
    Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId);
    Task<EmployeeDto> GetEmployee(Guid companyId, Guid id);
    Task<EmployeeDto> CreateEmployeeForCompany(Guid companyId,
        EmployeeForCreationDto employeeDto);
}
```

Before we add this method to the repository class, we are going to create a new query in the **EmployeeQuery** class:

```
public const string InsertEmployeeWithOutputQuery =
    @"INSERT INTO Employees (EmployeeId, Name, Age, Position, CompanyId)
      OUTPUT inserted.EmployeeId
      VALUES (default, @name, @age, @position, @id)";
```

This is the same query as we had in our previous example. Just here, we have to accept the **@id** as a parameter for the **CompanyId** column.

Now, we can add the missing method to the EmployeeRepository class:

```
public async Task<EmployeeDto> CreateEmployeeForCompany(Guid companyId,
    EmployeeForCreationDto employeeDto)
{
    var query = EmployeeQuery.InsertEmployeeWithOutputQuery;

    var param = new DynamicParameters(employeeDto);
    param.Add("id", companyId, DbType.Guid);

    using (var connection = _context.CreateConnection())
    {
        var id = await connection.QuerySingleAsync<Guid>(query, param);

        return new EmployeeDto(id, employeeDto.Name,
            employeeDto.Age, employeeDto.Position);
    }
}
```

This is also a similar implementation, just here, we have an additional `id` parameter to pass to our query.

Also to be able to use DbType enumeration, we have to include one more namespace:

```
using System.Data;
```

After this implementation, we can modify the `IEmployeeService` interface:

```
public interface IEmployeeService
{
    Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId);
    Task<EmployeeDto> GetEmployee(Guid companyId, Guid id);
    Task<EmployeeDto> CreateEmployeeForCompany(Guid companyId,
        EmployeeForCreationDto employeeDto);
}
```

And also the `EmployeeService` class:

```
public async Task<EmployeeDto> CreateEmployeeForCompany(Guid companyId,
    EmployeeForCreationDto employeeDto)
{
    var company = await _repository.Company.GetCompany(companyId);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

    var employee = await _repository.Employee
        .CreateEmployeeForCompany(companyId, employeeDto);

    return employee;
}
```

As soon as we have our created employee returned from a service, we can implement a missing action in the `EmployeesController`:

```
[HttpPost]
public async Task<IActionResult> CreateEmployeeForCompany(Guid companyId,
    [FromBody] EmployeeForCreationDto employee)
{
    if (employee is null)
        return BadRequest("EmployeeForCreationDto object is null");

    var employeeToReturn = await _service.EmployeeService
        .CreateEmployeeForCompany(companyId, employee);

    return CreatedAtRoute("GetEmployeeForCompany",
        new { companyId, id = employeeToReturn.EmployeeId },
        employeeToReturn);
}
```

Of course, for this to work, we have to modify the `HttpGet` attribute of the `GetEmployeeForCompany` action:

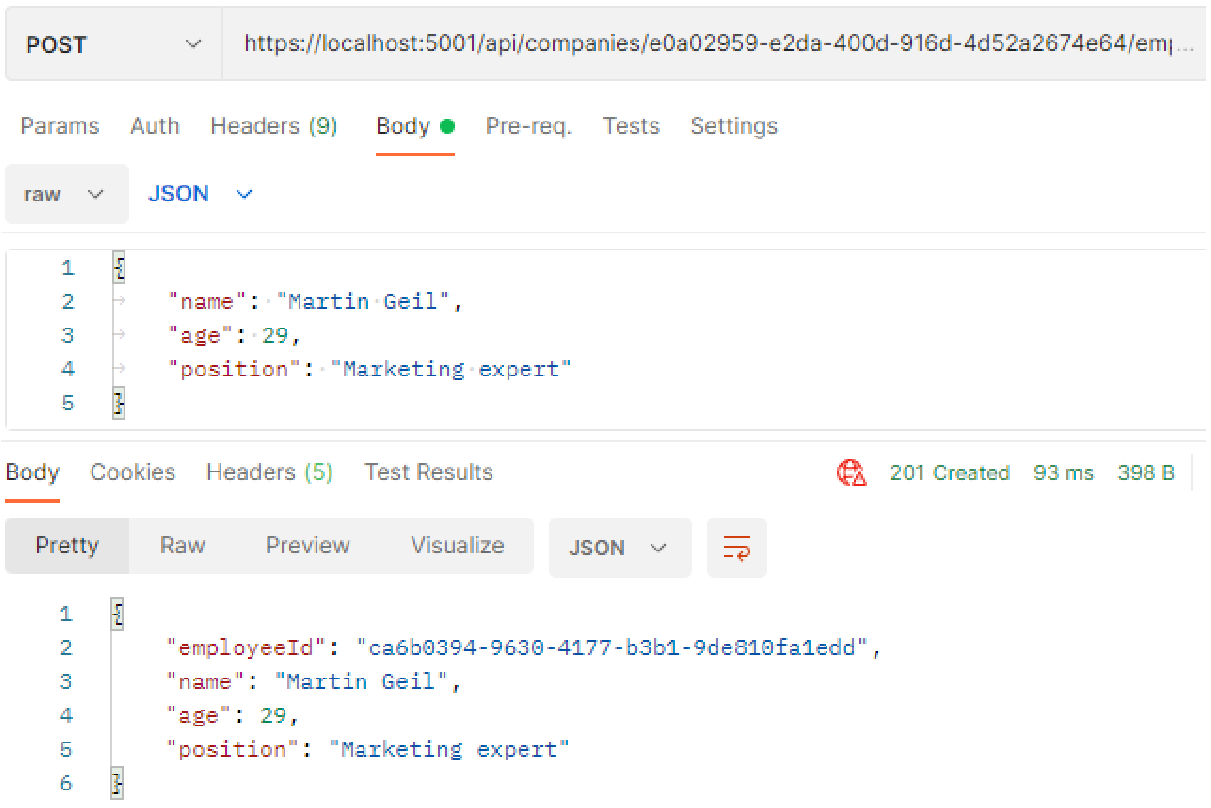
```
[HttpGet("{id:guid}", Name = "GetEmployeeForCompany")]
```

We also need to add a using statement to the `Shared/DataTransferObjects` project.

```
using Shared.DataTransferObjects;
```

Now, we can test this using the POST Employee for Company request that we already prepared for you in our bonus 2 file from the `09-Creating Resources` folder, but you will have to be sure to use a company id that exists in the `Company` table:

`https://localhost:5001/api/companies/e0a02959-e2da-400d-916d-4d52a2674e64/employees`


POST
https://localhost:5001/api/companies/e0a02959-e2da-400d-916d-4d52a2674e64/em|...
Params
Auth
Headers (9)
Body
Pre-req.
Tests
Settings
raw
JSON

```
{
    "name": "Martin Geil",
    "age": 29,
    "position": "Marketing expert"
}
```

```
{
    "employeeId": "ca6b0394-9630-4177-b3b1-9de810fa1edd",
    "name": "Martin Geil",
    "age": 29,
    "position": "Marketing expert"
}
```

And again, we can see a created employee as a response. Also, we can find the `Location` header in the `Headers` tab.

7.3 Creating Children Together with a Parent

There are situations where we want to create a parent resource with its children. Rather than using multiple requests for every single child, we want to do this in the same request with the parent resource.

We are going to show you how to do this. Also, we will see how to use transactions to make sure that all the actions are executed.

The first thing we are going to do is extend the `CompanyForCreationDto` record:

```
public record CompanyForCreationDto(string Name, string Address, string Country,
    IEnumerable<EmployeeForCreationDto>? Employees);
```

Now, let’s modify the `ICompanyRepository` interface:

```
public interface ICompanyRepository
```

```
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<CompanyDto> CreateCompanyWithEmployees(CompanyForCreationDto company);
}
```

We are not going to add any new queries for the company, but we are going to add a new query to the **EmployeeQuery** class:

```
public const string InsertEmployeeNoOutputQuery =
    @"INSERT INTO Employees (EmployeeId, Name, Age, Position, CompanyId)
      VALUES (default, @name, @age, @position, @id)";
```

This time, we are not returning any output value because we will use this query to create multiple employees for a single company. Let's add this method to the **CompanyRepository**:

```
public async Task<CompanyDto> CreateCompanyWithEmployees(CompanyForCreationDto
company)
{
    var query = CompanyQuery.InsertCompanyQuery;

    var param = new DynamicParameters(company);

    using (var connection = _context.CreateConnection())
    {
        connection.Open();

        using (var trans = connection.BeginTransaction())
        {
            var id = await connection
                .QuerySingleAsync<Guid>(query, param, transaction: trans);

            var queryEmp = EmployeeQuery.InsertEmployeeNoOutputQuery;

            var empList = company.Employees
                .Select(e => new { e.Name, e.Age, e.Position, id });

            await connection.ExecuteAsync(queryEmp, empList, transaction: trans);

            trans.Commit();

            return new CompanyDto(id, company.Name,
                string.Join(", ", company.Address, company.Country));
        }
    }
}
```

Here, our **company** parameter contains a collection of employees we want to create alongside its properties.

So, first, we use an existing query for the company creation and also create a list of parameters using the `DynamicParameters` class. Then, we can see the first difference once we create our connection. We have to open it explicitly. We have to do that because we are going to use transactions here and a transaction expects an already opened connection. Up until now, Dapper was automatically opening a connection for us.

Once we open our connection, we use the connection object to call the `BeginTransaction` method, which begins a new transaction. We have to use a transaction here because we are executing multiple operations that alter the database. When we have a situation like this, we want to make sure that all the operations are executed. In other words, if any operation fails, we don't want any other operation to succeed. So, with a transaction, either all operations execute successfully or no changes will be made on top of the database.

As soon as we create our transaction, we execute the company insert query with the `QuerySingleAsync` method and return the id of the created company. You can notice that now we pass three arguments for this method. We have to add our transaction object as an argument when we use transactions in our queries.

After we execute this query, we create another one for the employees – `queryEmp`. Then, we just transform the employee list into a new collection of anonymous objects, where each object has the `Name`, `Age`, and `Position` properties and for the `CompanyId` property we set an `id` from the newly created company.

Then, we call the `ExecuteAsync` method to execute our query and pass three arguments: the query, the list of objects, and the transaction.

Pay attention that the ExecuteAsync method allows us to pass a list as a second argument. That means we don't have to create a foreach loop and then call the ExecuteAsync method for each object in the list.

Once we execute the second query, we call the Commit method to commit our transaction.

Finally, we return a CompanyDto as a result.

If you want, you can always throw an exception after the QuerySingleAsync method execution, just to simulate that something went wrong during a transaction. You will see that even though we execute our first query if we don't call the Commit method, we won't be able to find any new company in our table.

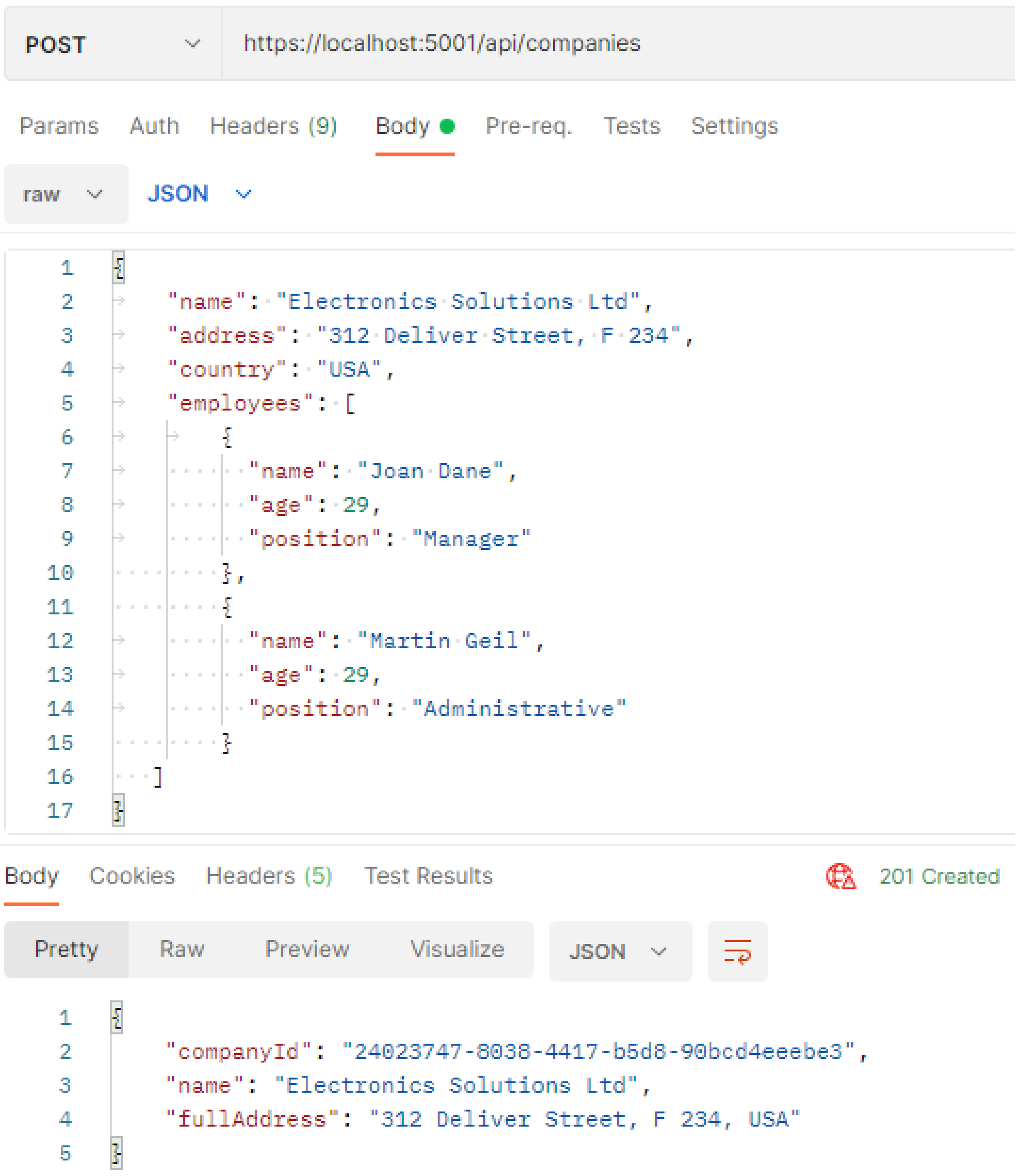
Now, let's just modify the CreateCompany method inside the CompanyService class:

```
public async Task<CompanyDto> CreateCompany(CompanyForCreationDto company)
{
    if (company.Employees is not null && company.Employees.Any())
        return await _repository.Company.CreateCompanyWithEmployees(company);
    else
        return await _repository.Company.CreateCompany(company);
}
```

So here we check if our company parameter has populated employees. If we find any, we execute the CreateCompanyWithEmployees repository method. Otherwise, we just call the previous CreateCompany method.

Now, to test this, we can send a new POST request with a company and two employees as a request body using the POST Company with Employees request that we already prepared for you in our bonus 2 file from the 09-Creating Resources folder:

https://localhost:5001/api/companies



```
{
    "name": "Electronics Solutions Ltd",
    "address": "312 Deliver Street, F 234",
    "country": "USA",
    "employees": [
        {
            "name": "Joan Dane",
            "age": 29,
            "position": "Manager"
        },
        {
            "name": "Martin Geil",
            "age": 29,
            "position": "Administrative"
        }
    ]
}
```

```
{
    "companyId": "24023747-8038-4417-b5d8-90bcd4eeebe3",
    "name": "Electronics Solutions Ltd",
    "fullAddress": "312 Deliver Street, F 234, USA"
}
```

And we can see that the request was executed successfully.

If we try to send a GET request to fetch all the employees for this company:

https://localhost:5001/api/companies/24023747-8038-4417-b5d8-90bcd4eeebe3/employees

We will get two employees for sure as a response.

7.4 Creating a Collection of Resources

The last part of this chapter is about creating a collection of resources. But as you already know from the main book, we can't go straight away for the POST action implementation. First, we have to implement a GET action that accepts a collection of ids as a parameter.

That said, let's add another member to the **ICompanyRepository** interface:

```
public interface ICompanyRepository
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<CompanyDto> CreateCompanyWithEmployees(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids);
}
```

Then, we need a new query in the **CompanyQuery** class:

```
public const string SelectCompaniesForMultipleIdsQuery =
    @"SELECT CompanyId, [Name], CONCAT([Address], ', ', Country) AS FullAddress
      FROM Companies
      WHERE CompanyId IN @ids";
```

We use the **IN** operator inside the **WHERE** clause to extract all the company rows with the required ids.

Now, the **CompanyRepository** class modification:

```
public async Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids)
{
    var query = CompanyQuery.SelectCompaniesForMultipleIdsQuery;

    using (var connection = _context.CreateConnection())
    {
        var companies = await connection
            .QueryAsync<CompanyDto>(query, new { ids });

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

There is nothing new here. We are creating our query, creating a connection, and using the **QueryAsync** method to execute our query where we pass the **query** and an anonymous object with our collection of **ids** as arguments.

Lastly, we convert our result to a list and return it.

After the repository, we have to add some changes to the service layer starting with the **ICompanyService** interface:

```
public interface ICompanyService
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids);
}
```

And then, let's add this missing member to a service class:

```
public async Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids)
{
    if (ids is null)
        throw new IdParametersBadRequestException();

    var companies = await _repository.Company.GetByIds(ids);

    if (ids.Count() != companies.Count())
        throw new CollectionByIdsBadRequestException();

    return companies;
}
```

As you did with all the previous exception classes, you can copy-paste all the bad request exception classes from our source code because we've already seen their implementation. Also, we have to modify the **ConfigureExceptionHandler** method in the **ExceptionMiddlewareExtensions** class:

```
context.Response.StatusCode = contextFeature.Error switch
{
       NotFoundException => StatusCodes.Status404NotFound,
       BadRequestException => StatusCodes.Status400BadRequest,
       _ => StatusCodes.Status500InternalServerError
};
```

Lastly, we can add a new action in the **CompaniesController**:

```
[HttpGet("collection/({ids})", Name = "CompanyCollection")]
public async Task<IActionResult> GetCompanyCollection(IEnumerable<Guid> ids)
```

```
{
    var companies = await _service.CompanyService.GetByIds(ids);

    return Ok(companies);
}
```

And that's it. This action is pretty straightforward, so let's continue toward the POST implementation.

As usual, we will start with the `ICompanyRepository` interface:

```
public interface ICompanyRepository
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<CompanyDto> CreateCompanyWithEmployees(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids);
    Task<IEnumerable<CompanyDto>> CreateCompanyCollection
        (IEnumerable<CompanyForCreationDto> companies);
}
```

We will accept the `CompanyForCreationDto` collection as a parameter and after the successful creation of multiple companies, we are going to return the `CompanyDto` collection as a result.

We don’t need a separate query for this method since we will reuse an existing one.

That said, let’s add a new method inside the repository class:

```
public async Task<IEnumerable<CompanyDto>> CreateCompanyCollection
    (IEnumerable<CompanyForCreationDto> companies)
{
    var companyList = new List<CompanyDto>();

    using (var connection = _context.CreateConnection())
    {
        connection.Open();

        using (var trans = connection.BeginTransaction())
        {
            foreach (var company in companies)
            {
                var query = CompanyQuery.InsertCompanyQuery;
                var param = new DynamicParameters(company);

                var id = await connection
                    .QuerySingleAsync<Guid>(query, param, transaction: trans);

                companyList.Add(new CompanyDto(id, company.Name,
                    string.Join(", ", company.Address, company.Country)));
            }

            trans.Commit();

            return companyList;
        }
    }
}
```

Here, we first create an empty list that we will populate with our created companies and return as a result. Then, we create a connection, open it, and also begin a transaction. Next, foreach company in our collection parameter, we use an existing query, create parameters, and extract the id of the created company by calling the QuerySingleAsync<Guid> method. Of course, for this method, we provide our `query`, `param`, and `transaction` as arguments. After the successful creation, we add a new `CompanyDto` object to a list.

As soon as we are done with the entire collection, we commit our transaction and return our `companyList` as a result.

Now, we can modify the `ICompanyService` interface:

```
public interface ICompanyService
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids);
    Task<(IEnumerable<CompanyDto> companies, string ids)>
        CreateCompanyCollection(IEnumerable<CompanyForCreationDto> companyCollection);
}
```

And also the `CompanyService` class:

```
public async Task<(IEnumerable<CompanyDto> companies, string ids)>
    CreateCompanyCollection(IEnumerable<CompanyForCreationDto> companyCollection)
{
    if (companyCollection is null)
        throw new CompanyCollectionBadRequest();

    var companies = await _repository.Company
        .CreateCompanyCollection(companyCollection);

    var ids = string.Join(",", companies.Select(c => c.CompanyId));
    return (companies, ids);
}
```

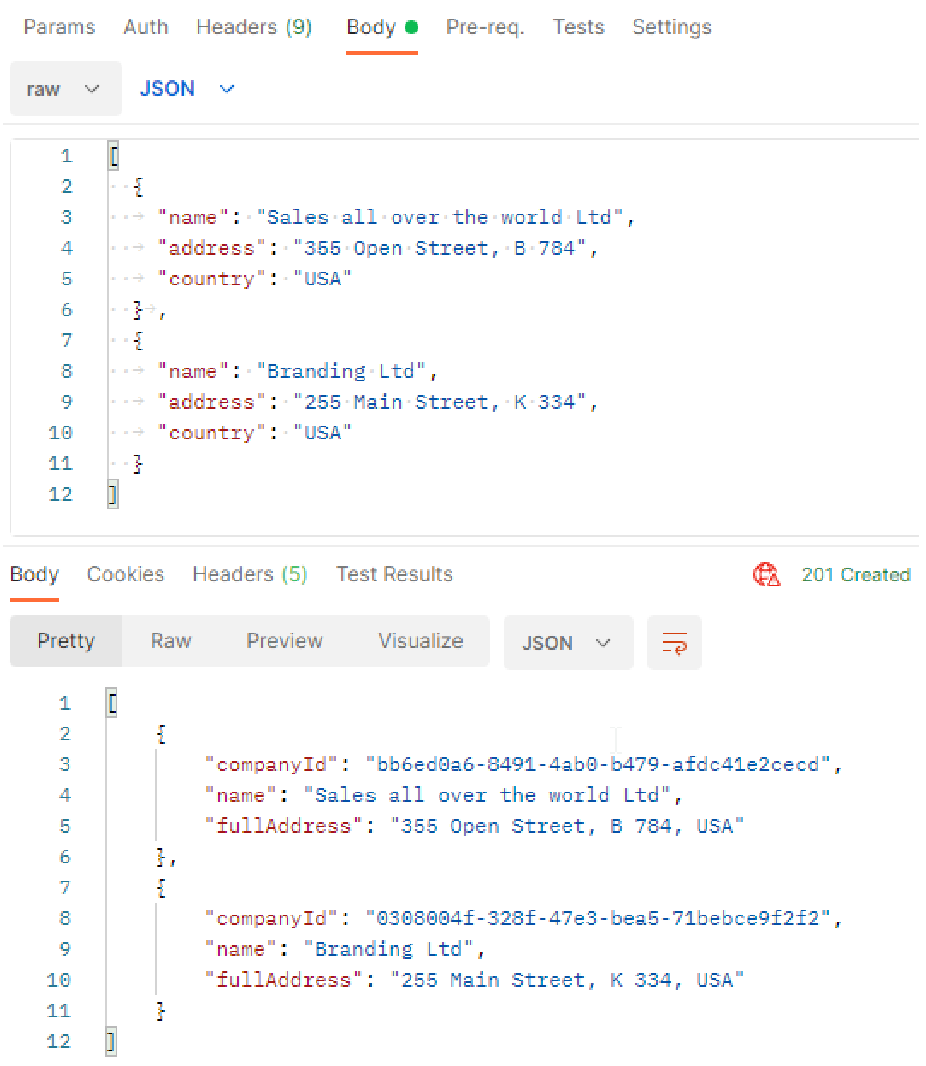
In this method, we execute our repository method, extract all the created companies and also all the ids from those companies, and return a tuple with both the companies and ids as parameters.

Finally, we have to modify our controller:

```
[HttpPost("collection")]
public async Task<IActionResult> CreateCompanyCollection([FromBody]
    IEnumerable<CompanyForCreationDto> companyCollection)
{
    var result = await _service.CompanyService
        .CreateCompanyCollection(companyCollection);

    return CreatedAtRoute("CompanyCollection",
        new { result.ids }, result.companies);
}
```

We can now test this with Postman using the POST Company collection request that we already prepared for you in our bonus 2 file from the `09-Creating Resources` folder:

`https://localhost:5001/api/companies/collection`

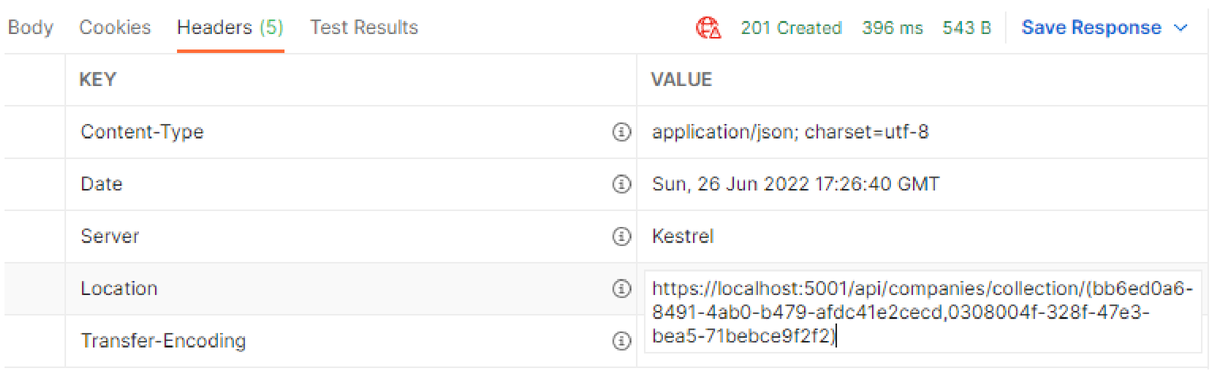


```
[
  {
    "name": "Sales all over the world Ltd",
    "address": "355 Open Street, B 784",
    "country": "USA"
  },
  {
    "name": "Branding Ltd",
    "address": "255 Main Street, K 334",
    "country": "USA"
  }
]
```

```
[
    {
        "companyId": "bb6ed0a6-8491-4ab0-b479-afdc41e2cecd",
        "name": "Sales all over the world Ltd",
        "fullAddress": "355 Open Street, B 784, USA"
    },
    {
        "companyId": "0308004f-328f-47e3-bea5-71bebce9f2f2",
        "name": "Branding Ltd",
        "fullAddress": "255 Main Street, K 334, USA"
    }
]
```

And we can see both our companies created. So this works as expected.

Also, if you inspect the Headers tab, you will find the link to use to fetch these created companies:



KEY	VALUE
Content-Type	application/json; charset=utf-8
Date	Sun, 26 Jun 2022 17:26:40 GMT
Server	Kestrel
Location	https://localhost:5001/api/companies/collection/(bb6ed0a6-8491-4ab0-b479-afdc41e2cecd,0308004f-328f-47e3-bea5-71bebce9f2f2)
Transfer-Encoding	

But as you remember from our main book, we can't just copy-paste this into another request and send it. That's because we will get a `415 Unsupported Media Type` response.

And again, as you remember, we had to implement a custom model binder to fix this.

So, all we have to do is to create a `ModelBinders` folder in the `Presentation` project and inside it create a new `ArrayModelBinder` class:

```
public class ArrayModelBinder : IModelBinder
{
	public Task BindModelAsync(ModelBindingContext bindingContext)
	{
		if (!bindingContext.ModelMetadata.IsEnumerableType)
		{
			bindingContext.Result = ModelBindingResult.Failed();
			return Task.CompletedTask;
		}

		var providedValue = bindingContext.ValueProvider
			.GetValue(bindingContext.ModelName)
			.ToString();
		if (string.IsNullOrEmpty(providedValue))
		{
			bindingContext.Result = ModelBindingResult.Success(null);
			return Task.CompletedTask;
		}

		var genericType = bindingContext.ModelType
			.GetTypeInfo().GenericTypeArguments[0];

		var converter = TypeDescriptor.GetConverter(genericType);

		var objectArray = providedValue
			.Split(new[] { "," }, StringSplitOptions.RemoveEmptyEntries)
			.Select(x => converter.ConvertFromString(x.Trim()))
			.ToArray();

		var guidArray = Array
			.CreateInstance(genericType, objectArray.Length);

		objectArray.CopyTo(guidArray, 0);
		bindingContext.Model = guidArray;

		bindingContext.Result = ModelBindingResult
			.Success(bindingContext.Model);

		return Task.CompletedTask;
	}
}
```

There's no need to explain this implementation since everything is already explained in our main book.

Also, for this to work, we need the required using directives:

```
using Microsoft.AspNetCore.Mvc.ModelBinding;
using System.ComponentModel;
using System.Reflection;
```

Now, in our controller, we have to modify the `GetCompanyCollection` action:

```
public async Task<IActionResult> GetCompanyCollection([ModelBinder(BinderType =
    typeof(ArrayModelBinder))]IEnumerable<Guid> ids)
```

And use add another using directive:

```
using CompanyEmployees.Presentation.ModelBinders;
```

Visual Studio will provide two different namespaces to resolve the error, so be sure to pick the right one.

And that's it. You can now use Postman to fetch the newly created companies with the already provided URI from the Headers tab of the previous POST request. You will get two new companies as a result.

8 WORKING WITH DELETE REQUESTS

In this chapter, we will see how we can handle DELETE requests in our API and use Dapper to delete resources from the database.

As with all the other chapters in this book, we will mainly focus on the Dapper implementation and will not repeat the same explanations we already have in the main book.

That said, let's start with the request where we want to delete a single employee record from the database.

To start with that, we are going to modify the `IEmployeeRepository` interface:

```
public interface IEmployeeRepository
{
    Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId);
    Task<EmployeeDto> GetEmployee(Guid companyId, Guid id);
    Task<EmployeeDto> CreateEmployeeForCompany(Guid companyId,
        EmployeeForCreationDto employeeDto);
    Task DeleteEmployee(Guid employeeId);
}
```

Compared to the implementation from the main book with EF Core, we can see that with Dapper it is enough to send just the `employeeId` as a parameter instead of the whole employee object.

Next, we have to add our query inside the `EmployeeQuery` class:

```
public const string DeleteEmployeeQuery =
    @"DELETE FROM Employees
      WHERE EmployeeId = @employeeId";
```

This is a plain simple SQL delete statement where we delete a single employee row based on the primary key parameter.

With this in place, we can add a missing method to the `EmployeeRepository` class:

```
public async Task DeleteEmployee(Guid employeeId)
{
    var query = EmployeeQuery.DeleteEmployeeQuery;
    using (var connection = _context.CreateConnection())
    {
        await connection.ExecuteAsync(query, new { employeeId });
    }
}
```

Nothing new here. After we get our query and create a connection, we just use the `ExecuteAsync` method to execute our query with the provided employeeId inside the anonymous object as an argument.

Of course, the `ExecuteAsync` method returns an int – the number of affected rows, so if you need that info, you can always retrieve it.

With this out of the way, we can modify the `IEmployeeService` interface:

```
public interface IEmployeeService
{
    Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId);
    Task<EmployeeDto> GetEmployee(Guid companyId, Guid id);
    Task<EmployeeDto> CreateEmployeeForCompany(Guid companyId,
        EmployeeForCreationDto employeeDto);
    Task DeleteEmployeeForCompany(Guid companyId, Guid employeeId);
}
```

And also the `EmployeeService` class:

```
public async Task DeleteEmployeeForCompany(Guid companyId, Guid employeeId)
{
    var company = await _repository
        .Company.GetCompany(companyId);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

    var employeeForCompany = _repository
        .Employee.GetEmployee(companyId, employeeId);
    if (employeeForCompany is null)
        throw new EmployeeNotFoundException(employeeId);

    await _repository.Employee.DeleteEmployee(employeeId);
}
```

Here, we just verify that both a company (the parent of an employee we want to delete) and an employee exist in the database, and if they do, we call the repository method to delete that employee record.

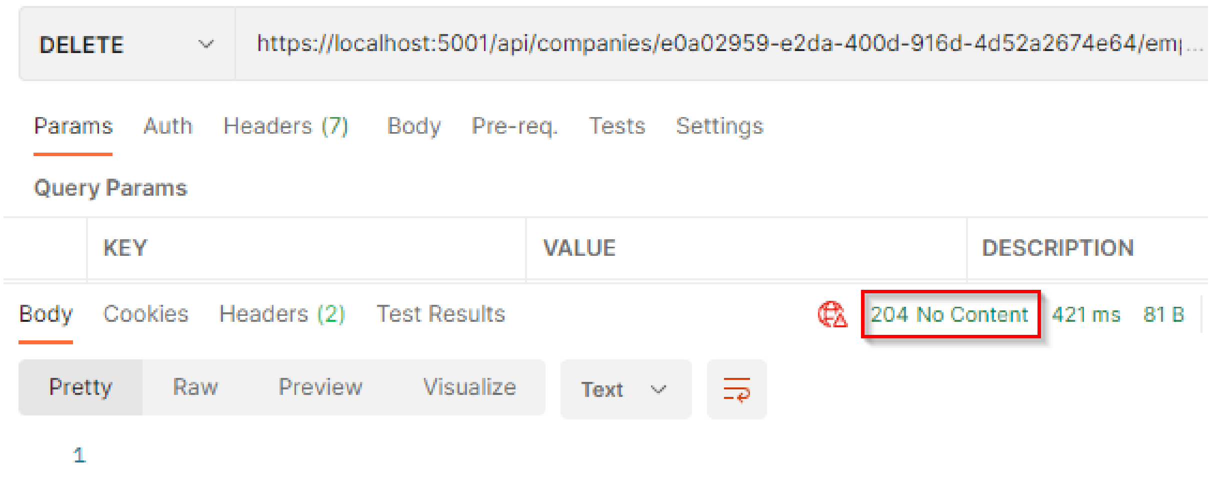
Finally, we are going to add a new action in the `EmployeesController`:

```
[HttpDelete("{id:guid}")]
public async Task<IActionResult> DeleteEmployeeForCompany(Guid companyId, Guid id)
{
    await _service.EmployeeService.DeleteEmployeeForCompany(companyId, id);

    return NoContent();
}
```

With this, our implementation is over.

After making sure that we are using an existing company id and employee id, we can test it with Postman using the DELETE Employee for company request that we already prepared for you in our bonus 2 file from the 10-Working with DELETE Requests folder:

https://localhost:5001/api/companies/e0a02959-e2da-400d-916d-4d52a2674e64/employees/ca6b0394-9630-4177-b3b1-9de810fa1edd



We see that we get a 204 No Content response, which means that the request was successfully executed.

Also, you can try to fetch this deleted employee, and you will get a 404 error response.

8.1 Deleting a Parent Resource with its Children

With the relationship setup as we already have from our migration file, this is pretty easy to accomplish. If we inspect the `InitialTables_202206160001` class, we can find the configuration where we've enabled the cascading delete:

```
Create.Table("Employees")
    .WithColumn("EmployeeId").AsGuid().NotNullable()
        .PrimaryKey().WithDefaultValue("NEWID()")
    .WithColumn("Name").AsString(50).NotNullable()
    .WithColumn("Age").AsInt32().NotNullable()
    .WithColumn("Position").AsString(50).NotNullable()
    .WithColumn("CompanyId").AsGuid().NotNullable()
        .ForeignKey("Companies", "CompanyId")
        .OnDelete(System.Data.Rule.Cascade);
```

So, all we have to do is to create logic for deleting the parent resource.

Well, let's do that following the same steps as in a previous example:

```
public interface ICompanyRepository
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<CompanyDto> CreateCompanyWithEmployees(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids);
    Task<IEnumerable<CompanyDto>> CreateCompanyCollection
        (IEnumerable<CompanyForCreationDto> companies);
    Task DeleteCompany(Guid id);
}
```

Next, let's add a query to the `CompanyQuery` class:

```
public const string DeleteCompanyQuery =
    @"DELETE FROM Companies
      WHERE CompanyId = @id";
```

After this, we can modify the repository class:

```
public async Task DeleteCompany(Guid id)
{
    var query = CompanyQuery.DeleteCompanyQuery;

    using (var connection = _context.CreateConnection())
    {
        await connection.ExecuteAsync(query, new { id });
    }
}
```

Both the query and this method are almost the same as the ones we used in our previous section.

To continue, we have to modify the `ICompanyService` interface:

```
public interface ICompanyService
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids);
    Task<(IEnumerable<CompanyDto> companies, string ids)>
        CreateCompanyCollection(IEnumerable<CompanyForCreationDto> companyCollection);
    Task DeleteCompany(Guid companyId);
}
```

And of course, to implement this missing member:

```
public async Task DeleteCompany(Guid companyId)
{
    var company = await _repository.Company.GetCompany(companyId);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

    await _repository.Company.DeleteCompany(companyId);
}
```

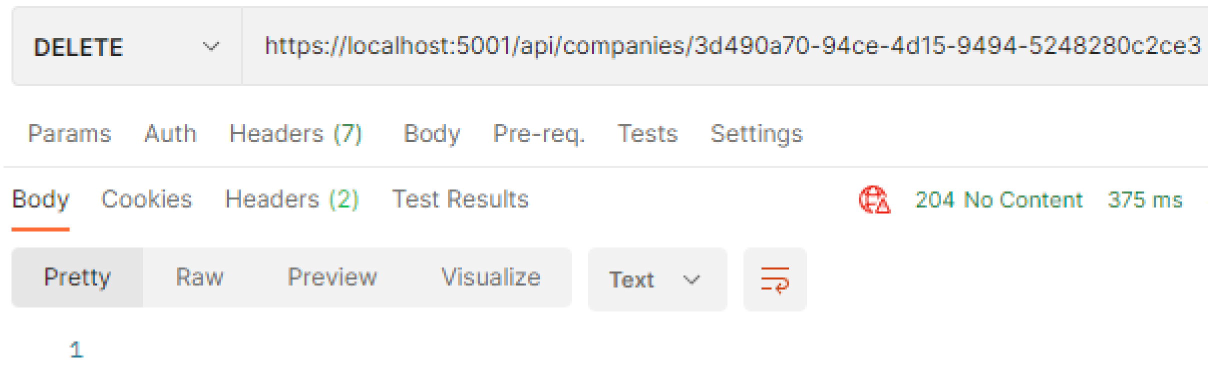
Finally, let's add a new action to the `CompaniesController`:

```
[HttpDelete("{id:guid}")]
public async Task<IActionResult> DeleteCompany(Guid id)
{
    await _service.CompanyService.DeleteCompany(id);

    return NoContent();
}
```

We again return a 204 NoContent if everything goes well.

After making sure that we are using an existing company id, to test this, we will send a DELETE request from Postman using the DELETE Company request that we already prepared for you in our bonus 2 file from the 10-Working with DELETE Requests folder:

https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3


DELETE https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3
204 No Content 375 ms

And again, we get a 204 No Content response.

You can check in your database that this company alongside its children doesn’t exist anymore.

There we go. We have finished working with DELETE requests and we are ready to continue with the PUT requests.

9 Working with PUT Requests

In this section, we are going to show you how to update a resource using a PUT request with Dapper. We are going to update a child resource first and then show you how to execute an insert while updating a parent resource.

9.1 Updating Employee

Before we start the implementation, we are going to create a new **EmployeeForUpdateDto** record:

```
public record EmployeeForUpdateDto(string Name, int Age, string Position);
```

Next, let's modify the **IEmployeeRepository** interface:

```
public interface IEmployeeRepository
{
    Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId);
    Task<EmployeeDto> GetEmployee(Guid companyId, Guid id);
    Task<EmployeeDto> CreateEmployeeForCompany(Guid companyId,
        EmployeeForCreationDto employeeDto);
    Task DeleteEmployee(Guid employeeId);
    Task UpdateEmployee(Guid employeeId, EmployeeForUpdateDto employee);
}
```

Since Dapper doesn't track changes the way EF Core does, we can't just modify the service layer as we did in the main book. Therefore, we are starting our implementation on the repository level.

That said, let's continue with the update statement inside the **EmployeeQuery** class:

```
public const string UpdateEmployeeQuery =
    @"UPDATE Employees
      SET [Name] = @name, Age = @age, Position = @position
      WHERE EmployeeId = @employeeId";
```

This is a regular **UPDATE SQL** statement where we update our row based on the primary key value.

Now, we can modify the repository class by adding another method to it:

```
public async Task UpdateEmployee(Guid employeeId, EmployeeForUpdateDto employee)
{
    var query = EmployeeQuery.UpdateEmployeeQuery;

    var param = new DynamicParameters(employee);
    param.Add("employeeId", employeeId, DbType.Guid);

    using (var connection = _context.CreateConnection())
    {
        await connection.ExecuteAsync(query, param);
    }
}
```

Since our `employee` parameter has all the properties we need for our SQL parameters, we use it inside the `DynamicParameters` class to provide parameters for our Dapper execution. We add a parameter for `employeeId` of type `Guid`, and then we just create a connection and execute our query calling the `ExecuteAsync` method.

With this done, we can move on to the `IEmployeeService` modification:

```
public interface IEmployeeService
{
    Task<IEnumerable<EmployeeDto>> GetEmployees(Guid companyId);
    Task<EmployeeDto> GetEmployee(Guid companyId, Guid id);
    Task<EmployeeDto> CreateEmployeeForCompany(Guid companyId,
        EmployeeForCreationDto employeeDto);
    Task DeleteEmployeeForCompany(Guid companyId, Guid employeeId);
    Task UpdateEmployeeForCompany(Guid companyId, Guid id,
        EmployeeForUpdateDto employee);
}
```

And also the `EmployeeService` class modification:

```
public async Task UpdateEmployeeForCompany(Guid companyId, Guid id,
EmployeeForUpdateDto employee)
{
    var company = await _repository.Company.GetCompany(companyId);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

    var employeeDto = await _repository.Employee.GetEmployee(companyId, id);
    if (employeeDto is null)
        throw new EmployeeNotFoundException(id);

    await _repository.Employee.UpdateEmployee(id, employee);
}
```

As usual, we have to check whether the parent company entity exists in the database. If it does, we check whether the employee entity that we want to update exists as well. Finally, if both checks out, we call the update method in the repository.

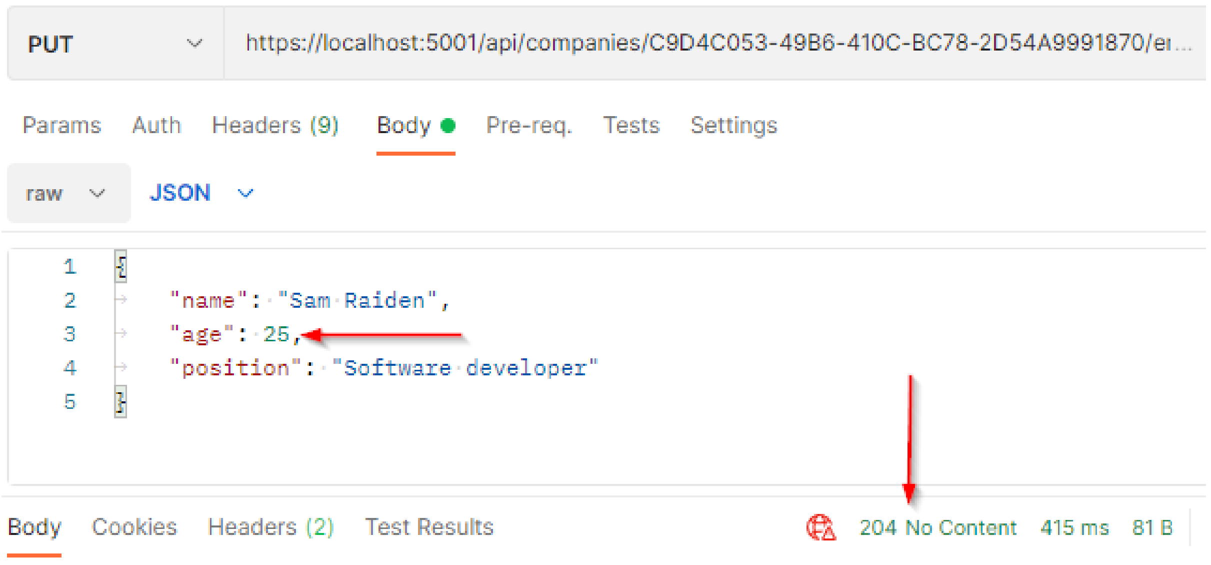
Finally, let's modify the `EmployeesController`:

```
[HttpPut("{id:guid}")]
public async Task<IActionResult> UpdateEmployeeForCompany(Guid companyId, Guid id,
    [FromBody] EmployeeForUpdateDto employee)
{
    if (employee is null)
        return BadRequest("EmployeeForUpdateDto object is null");

    await _service.EmployeeService
        .UpdateEmployeeForCompany(companyId, id, employee);

    return NoContent();
}
```

There is nothing new here that we didn't learn from the main book.

Since this example is working with data that we created in the migrations, we can be sure that we have an existing company id and employee id to use in this example, and we can test it with Postman using the UPDATE Employee for company request that we already prepared for you in our bonus 2 file from the 11-Working with PUT Requests folder:

https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBCA8-664D-4B20-B5DE-024705497D4A


PUT https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/er...

```
{
    "name": "Sam Raiden",
    "age": 25,
    "position": "Software developer"
}
```

204 No Content 415 ms 81 B

And we can see we get 204 as a result. Previously, the age of this employee was 26.

Now, as we said, Dapper doesn't support change tracking as EF Core does, thus we can't update just the modified properties. And to be honest here, you probably wouldn't even notice the difference in performance between the two implementations, **so we shouldn't overengineer our solution unless it is really necessary.** Everything should be tested and if proven that the update action slows down the execution then we should either implement tracking ourselves or use one of the external libraries that support change tracking like Dapper.Contrib or Dapper.Rainbow. Of course, each of these libraries has its limitations, so in most cases using raw dapper queries is usually the way to go.

9.2 Inserting Resources while Updating One

While updating a parent resource, we can create child resources as well. We will use transactions here to help us ensure database consistency.

That said, let's start with the **`CompanyForUpdateDto`** creation:

```
public record CompanyForUpdateDto(string Name, string Address,
    string Country, IEnumerable<EmployeeForCreationDto> Employees);
```

Next, we are going to modify the **`ICompanyRepository`** interface:

```
public interface ICompanyRepository
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<CompanyDto> CreateCompanyWithEmployees(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids);
    Task<IEnumerable<CompanyDto>> CreateCompanyCollection
        (IEnumerable<CompanyForCreationDto> companies);
    Task DeleteCompany(Guid id);
    Task UpsertCompany(Guid id, CompanyForUpdateDto company);
}
```

We call this method **`UpsertCompany`** because we will use it to update the company and also insert one or multiple employees for that company.

Then, we need a new query to update the company:

```
public const string UpdateCompanyQuery =
    @"UPDATE Companies
      SET [Name] = @name, Address = @address, Country = @country
      WHERE CompanyId = @id";
```

And then, we can implement the `UpsertCompany` method inside the repository class:

```
public async Task UpsertCompany(Guid id, CompanyForUpdateDto company)
{
    var query = CompanyQuery.UpdateCompanyQuery;

    var param = new DynamicParameters(company);
    param.Add("id", id, DbType.Guid);

    using (var connection = _context.CreateConnection())
    {
        connection.Open();

        using (var trans = connection.BeginTransaction())
        {
            await connection.ExecuteAsync(query, param, transaction: trans);

            var queryEmp = EmployeeQuery.InsertEmployeeNoOutputQuery;

            var empList = company.Employees
                .Select(e => new { e.Name, e.Age, e.Position, id });

            await connection.ExecuteAsync(queryEmp, empList, transaction: trans);

            trans.Commit();
        }
    }
}
```

This method accepts an `id` of the company we want to update and the `company` parameter with all the employees we want to insert.

Once we get the update query and add parameters for the update company action, we create the connection, open it, and begin a transaction. Then we call the `ExecuteAsync` method to execute our update statement and provide our query, parameters, and a transaction as arguments.

After that, we use our existing insert employee statement without an output, extract employee parameters inside a collection and call the same `ExecuteAsync` method again. But this time, we provide a collection of anonymous objects as an argument next to the query and transaction.

Finally, we commit our transaction.

We have to add a using statement for `System.Data`:

```
using System.Data;
```

On the service layer, we have to add a new member to the interface:

```
public interface ICompanyService
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids);
    Task<(IEnumerable<CompanyDto> companies, string ids)>
        CreateCompanyCollection(IEnumerable<CompanyForCreationDto> companyCollection);
    Task DeleteCompany(Guid companyId);
    Task UpdateCompany(Guid companyId, CompanyForUpdateDto companyForUpdate);
}
```

And implement it in the service class:

```
public async Task UpdateCompany(Guid companyId, CompanyForUpdateDto companyForUpdate)
{
    var companyEntity = await _repository.Company.GetCompany(companyId);
    if (companyEntity is null)
        throw new CompanyNotFoundException(companyId);

    if (companyForUpdate.Employees is not null && companyForUpdate.Employees.Any())
    {
        await _repository.Company.UpsertCompany(companyId, companyForUpdate);
    }
}
```

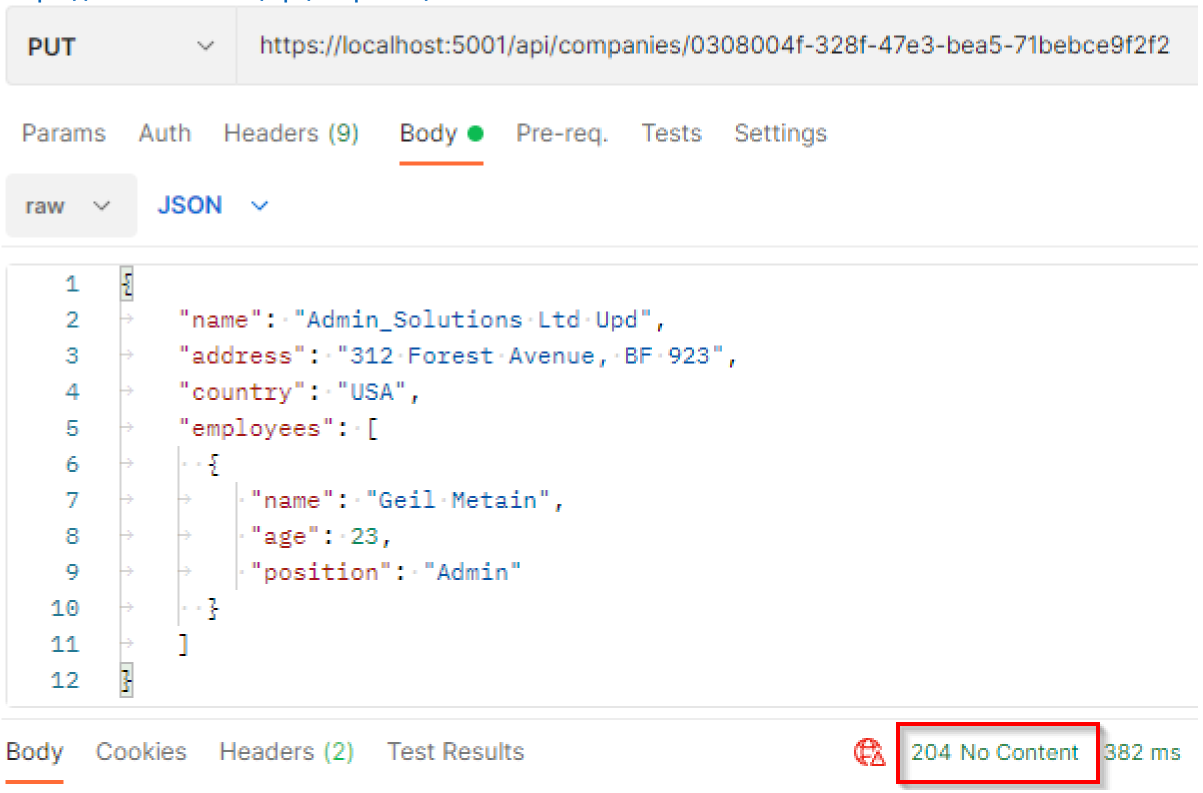
Finally, we have to modify the controller:

```
[HttpPut("{id:guid}")]
public async Task<IActionResult> UpdateCompany(Guid id, [FromBody] CompanyForUpdateDto
company)
{
    if (company is null)
        return BadRequest("CompanyForUpdateDto object is null");

    await _service.CompanyService.UpdateCompany(id, company);

    return NoContent();
}
```

And that’s it.

Since this example is working with data that we created in the migrations, we can be sure that we have an existing company id, and we can test it with Postman using the UPDATE Company with employees request that we already prepared for you in our bonus 2 file from the `11-Working with PUT Requests` folder:

https://localhost:5001/api/companies/0308004f-328f-47e3-bea5-71bebce9f2f2



```
{
    "name": "Admin_Solutions Ltd Upd",
    "address": "312 Forest Avenue, BF 923",
    "country": "USA",
    "employees": [
      {
        "name": "Geil Metain",
        "age": 23,
        "position": "Admin"
      }
    ]
}
```

And we can see a 204 No Content response. Of course, if you inspect the company with the provided id, you will find the modified columns and also a new employee for that company.

Now, we are ready to see how Dapper works with stored procedures.

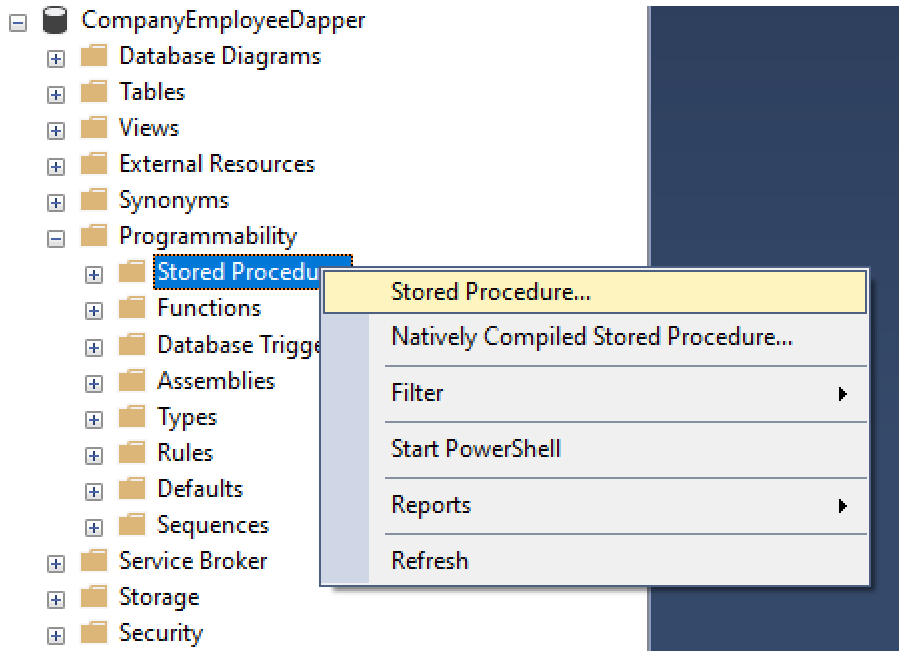
10 Working with Stored Procedures

Dapper is a great and easy tool to use with stored procedures. And in this section, we are going to see how.

A lot of times, we have to create a bit of complex SQL logic, and creating that inside our C# class can be pretty unreadable and hard to maintain. Therefore, we use stored procedures to keep that complex logic in, and then just call it with Dapper and consume the result.

To show you how to use Dapper to call stored procedures, we have to create a stored procedure first.

To create one, we are going to expand our database, expand the `Programmability` folder, and then right-click on the `Stored Procedures` folder and click the `Stored Procedure` option:



Once we do that, we will get a template to create our stored procedure.

Now, let’s modify the template:

```
CREATE PROCEDURE [ShowCompanyByEmployeeId]
       @Id uniqueidentifier
AS
```

```
BEGIN
   SET NOCOUNT ON;

   SELECT c.CompanyId, c.[Name], CONCAT(c.[Address], ', ', c.Country) AS FullAddress
   FROM Companies c JOIN Employees e ON c.CompanyId = e.CompanyId
   WHERE e.EmployeeId = @Id
END
GO
```

Here, we just select the company based on the employee's id value.

Now, we can hit that **Execute** button and create a stored procedure.

That's it regarding the SQL part. We can go back to Visual Studio.

Let's start with the **ICompanyRepository** modification:

```
public interface ICompanyRepository
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<CompanyDto> CreateCompanyWithEmployees(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids);
    Task<IEnumerable<CompanyDto>> CreateCompanyCollection
        (IEnumerable<CompanyForCreationDto> companies);
    Task DeleteCompany(Guid id);
    Task UpsertCompany(Guid id, CompanyForUpdateDto company);
    Task<CompanyDto> GetCompanyByEmployeeId(Guid employeeId);
}
```

Next, we can modify the repository class:

```
public async Task<CompanyDto> GetCompanyByEmployeeId(Guid employeeId)
{
    var procName = "ShowCompanyByEmployeeId";

    using (var connection = _context.CreateConnection())
    {
        var company = await connection
            .QueryFirstOrDefaultAsync<CompanyDto>(procName,
            new { id = employeeId }, commandType: CommandType.StoredProcedure);

        return company;
    }
}
```

We can see that the logic is almost the same as we had in some of our previous examples. After we create a **connection**, we use the **QueryFirstOrDefaultAsync** method and provide the name of the procedure, the parameter, and the command type as arguments.

This method will execute our stored procedure, and map the result to the CompanyDto type.

Once the result is mapped, we return it to the caller.

One thing to mention here is if we want to be more declarative about our parameters we can use this syntax:

```
var param = new DynamicParameters();
param.Add("id", employeeId, DbType.Guid, ParameterDirection.Input);
```

You can see that we can add the direction of the parameter as Input. Of course, we have other options as well as Output, ReturnValue, and InputOutput.

With this out of the way, let's modify the `ICompanyService` interface:

```
public interface ICompanyService
{
    Task<IEnumerable<CompanyDto>> GetAllCompanies();
    Task<CompanyDto> GetCompany(Guid id);
    Task<IEnumerable<CompanyWithEmployeesDto>> GetCompaniesWithEmployees();
    Task<CompanyDto> CreateCompany(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIds(IEnumerable<Guid> ids);
    Task<(IEnumerable<CompanyDto> companies, string ids)>
        CreateCompanyCollection(IEnumerable<CompanyForCreationDto> companyCollection);
    Task DeleteCompany(Guid companyId);
    Task UpdateCompany(Guid companyId, CompanyForUpdateDto companyForUpdate);
    Task<CompanyDto> GetCompanyByEmployeeId(Guid employeeId);
}
```

And also, let's modify the service class:

```
public async Task<CompanyDto> GetCompanyByEmployeeId(Guid employeeId)
{
    var company = await _repository.Company.GetCompanyByEmployeeId(employeeId);

    return company;
}
```

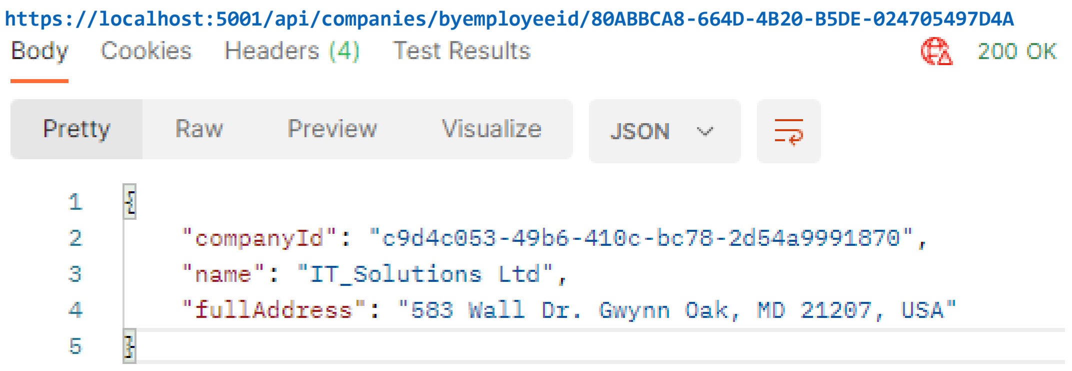
Of course, we can always check here if an employee with this id exists in the database, but for now, let's just focus on getting our result from the repository method.

Finally, let's add another action to the controller:

```
[HttpGet("byemployeeid/{id:guid}")]
public async Task<IActionResult> GetCompanyByEmployeeId(Guid id)
```

```
{
    var company = await _service.CompanyService.GetCompanyByEmployeeId(id);

    return Ok(company);
}
```

Let’s test it by creating a new request in Postman, using one of the employee ids from a company that we created earlier:



```
https://localhost:5001/api/companies/byemployeeid/80ABBCA8-664D-4B20-B5DE-024705497D4A
```

```
{
    "companyId": "c9d4c053-49b6-410c-bc78-2d54a9991870",
    "name": "IT_Solutions Ltd",
    "fullAddress": "583 Wall Dr. Gwynn Oak, MD 21207, USA"
}
```

We can see the result, which means that our call to stored procedure works as well.

11 Paging with ASP.NET Core Web API and Dapper

By reading our main Ultimate ASP.NET Core Web API book, you have learned what paging is, and the step-by-step implementation from the basic steps to some improved solutions.

As we know all of that, in this chapter, we will see how to implement the same functionality using Dapper instead of EF Core. We'll show you the entire implementation process, but as with the previous sections of this book, we won't spend too much time explaining things that are already explained in our main book.

That said, let's start with the implementation.

11.1 Paging Implementation

The first thing we are going to do is to create a new **RequestFeatures** folder in the **Shared** project. Then, we are going to add a new **RequestParameters** class inside the newly created folder:

```
public abstract class RequestParameters
{
	const int maxPageSize = 50;
	public int PageNumber { get; set; } = 1;

	private int _pageSize = 10;
	public int PageSize
	{
		get
		{
			return _pageSize;
		}
		set
		{
			_pageSize = (value > maxPageSize) ? maxPageSize : value;
		}
	}
}
```

Here, we are using the **maxPageSize** constant to restrict our API to a maximum of 50 rows per page. We also have two public properties –

PageNumber and PageSize. If not set by the caller, PageNumber will be set to 1, and PageSize to 10.

Then let's add one more EmployeeParameters class, in the same folder:

```
public class EmployeeParameters : RequestParameters
{
}
```

As we already said, in this chapter, we won't implement a basic solution and then improve it, we will jump straight on to the improved version. There is no need to repeat all the same steps and explanations we did in the main book.

That said, let's add two more classes in the same folder.

The MetaData class:

```
public class MetaData
{
	public int CurrentPage { get; set; }
	public int TotalPages { get; set; }
	public int PageSize { get; set; }
	public int TotalCount { get; set; }

	public bool HasPrevious => CurrentPage > 1;
	public bool HasNext => CurrentPage < TotalPages;
}
```

This class contains metadata for our paging response.

And the PagedList class:

```
public class PagedList<T> : List<T>
{
	public MetaData MetaData { get; set; }

	public PagedList(List<T> items, int count, int pageNumber, int pageSize)
	{
		MetaData = new MetaData
		{
			TotalCount = count,
			PageSize = pageSize,
			CurrentPage = pageNumber,
			TotalPages = (int)Math.Ceiling(count / (double)pageSize)
		};

		AddRange(items);
	}
}
```

As we suggested in the last part of the pagination section from the main book, we will pass to this class only the required items that we already fetched from the database (skip and take will already be applied).

With both additional classes in place, we can modify the **IEmployeeRepository** interface:

```
public interface IEmployeeRepository
{
    Task<PagedList<EmployeeDto>> GetEmployees(Guid companyId,
        EmployeeParameters employeeParameters);
    Task<EmployeeDto> GetEmployee(Guid companyId, Guid id);
    Task<EmployeeDto> CreateEmployeeForCompany(Guid companyId,
        EmployeeForCreationDto employeeDto);
    Task DeleteEmployee(Guid employeeId);
    Task UpdateEmployee(Guid employeeId, EmployeeForUpdateDto employee);
}
```

We've changed the return type of our method and also add a new parameter to the signature. We need to add a using statement for Shared.RequestFeatures.

Now, we can add our query. But before we do that, we want to mention that we want two things from our query, the number of employees per company, and all the required employees based on the page size and page number parameters. So basically, we will execute two queries in a single Dapper request and extract both results. We will see how Dapper allows us to do that.

Let's continue with the query first. We will modify the **SelectEmployeesQuery** query inside the **EmployeeQuery** class:

```
public const string SelectEmployeesQuery =
    @"SELECT COUNT(e.EmployeeId)
      FROM Employees AS e
      WHERE e.CompanyId = @companyId;

      SELECT e.EmployeeId, e.[Name], e.[Age], e.Position
      FROM Employees AS e
      WHERE e.CompanyId = @companyId
      ORDER BY e.[Name]
      OFFSET @Skip ROWS FETCH NEXT @Take ROWS ONLY";
```

So, in the first **SELECT** statement, we extract the number of employees for the provided **companyId** parameter. And in the second one, we extract these employees for the provided companyId parameter, sort them by name, and also use the **OFFSET @Skip ROWS** to skip the required number of rows and also use the **FETCH NEXT @Take ROWS ONLY** to take the required number of rows. This means we will pass three parameters to this query, companyId, skip, and take.

Once we have our query in place, we can modify the **GetEmployees** method in the class:

```
public async Task<PagedList<EmployeeDto>> GetEmployees(Guid companyId,
    EmployeeParameters employeeParameters)
{
    var skip = (employeeParameters.PageNumber - 1) * employeeParameters.PageSize;

    var param = new DynamicParameters();
    param.Add("companyId", companyId, DbType.Guid);
    param.Add("skip", skip, DbType.Int32);
    param.Add("take", employeeParameters.PageSize, DbType.Int32);
    var query = EmployeeQuery.SelectEmployeesQuery;

    using (var connection = _context.CreateConnection())
    using (var multi = await connection.QueryMultipleAsync(query, param))
    {
        var count = await multi.ReadSingleAsync<int>();
        var employees = (await multi.ReadAsync<EmployeeDto>()).ToList();

        return new PagedList<EmployeeDto>(employees, count,
            employeeParameters.PageNumber, employeeParameters.PageSize);
    }
}
```

Here we change the signature of our method by changing the return type and also adding a new parameter. Then, we calculate the skip value, add all three parameters as **DynamicParameters**, and use our prepared query.

Next, we create a connection, and then we use the **QueryMultipleAsync** method to execute multiple statements in a single query. This method returns a set of results, and we can extract each of those with the **ReadSingleAsync<T>** method for a single result, and the **ReadAsync<T>** method for a collection result.

Once we have both results, we return a new PagedList with all the required arguments. We need to add a using statement for **Shared.RequestFeatures**.

So, you can see how nice and easy is to execute multiple statements with Dapper.

Now, we can move on with the **IEmployeeService** modification:

```
public interface IEmployeeService
{
    Task<(IEnumerable<EmployeeDto> employees, MetaData metaData)>
        GetEmployees(Guid companyId, EmployeeParameters employeeParameters);
    Task<EmployeeDto> GetEmployee(Guid companyId, Guid id);
    Task<EmployeeDto> CreateEmployeeForCompany(Guid companyId,
        EmployeeForCreationDto employeeDto);
    Task DeleteEmployeeForCompany(Guid companyId, Guid employeeId);
    Task UpdateEmployeeForCompany(Guid companyId, Guid id,
        EmployeeForUpdateDto employee);
}
```

This method now returns a tuple of the **EmployeeDto** collection and **MetaData**. Again, we need to add a using statement for Shared.RequestFeatures.

As soon as we are done with the interface changes, we can implement them in the service class:

```
public async Task<(IEnumerable<EmployeeDto> employees, MetaData metaData)>
    GetEmployees(Guid companyId, EmployeeParameters employeeParameters)
{
    var company = await _repository.Company.GetCompany(companyId);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

    var employeesWithMetaData = await _repository.Employee
        .GetEmployees(companyId, employeeParameters);

    var employees = employeesWithMetaData
        .Select(e => new EmployeeDto(e.EmployeeId, e.Name, e.Age, e.Position));

    return (employees: employees, metaData: employeesWithMetaData.MetaData);
}
```

As soon as we call the repository method and get the requested result, we extract all the employees from the paged result and return both the collection and the metadata. Again, we need to add a using statement for `Shared.RequestFeatures`.

Finally, we have to modify our action inside the controller:

```
[HttpGet]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId,
    [FromQuery] EmployeeParameters employeeParameters)
{
    var pagedResult = await _service
        .EmployeeService.GetEmployees(companyId, employeeParameters);

    Response.Headers.Add("X-Pagination",
        JsonSerializer.Serialize(pagedResult.metaData));

    return Ok(pagedResult.employees);
}
```

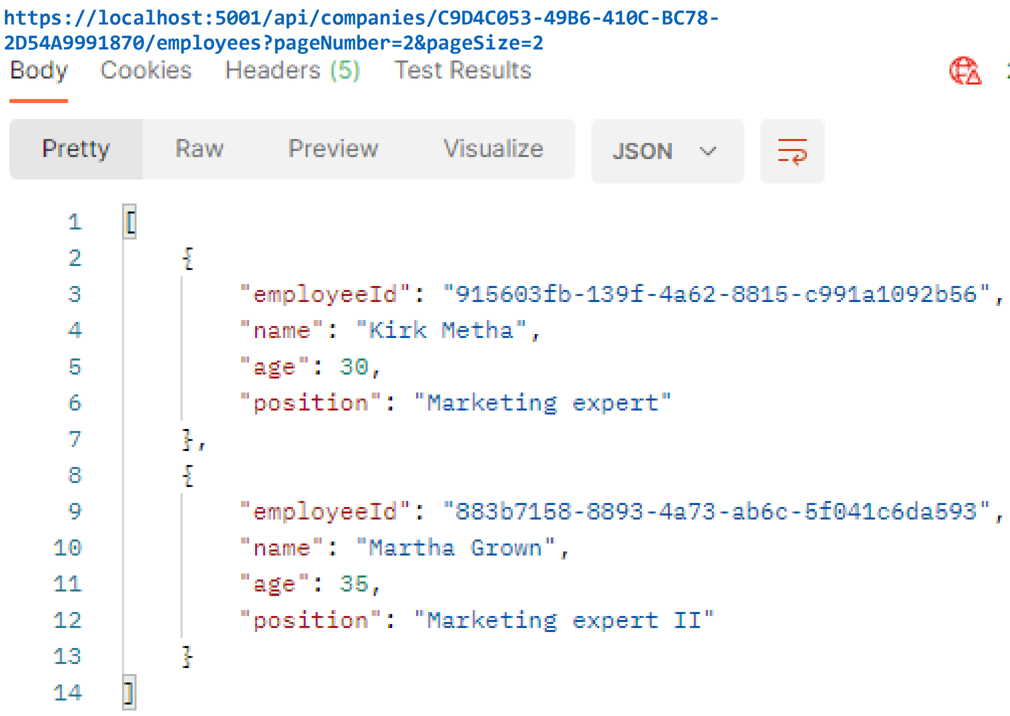
This is the same implementation from our book. Once more, we need to add a using statement for `Shared.RequestFeatures` and another for `System.Text.Json`.

Now, before we test this, we can use the table from the main book and create a few new employees for the company with the id: `C9D4C053-49B6-410C-BC78-2D54A9991870`.

{ "name": "Mihael Worth", "age": 30, "position": "Marketing expert" }	{ "name": "John Spike", "age": 32, "position": "Marketing expert II" }	{ "name": "Nina Hawk", "age": 26, "position": "Marketing expert II" }
{ "name": "Mihael Fins", "age": 30, "position": "Marketing expert" }	{ name": "Martha Grown", "age": 35, "position": "Marketing expert II" }	{ "name": "Kirk Metha", "age": 30, "position": "Marketing expert" }

We can use the empty POST Employee for Company request from our Postman collection file (16-Paging in ASP.NET Core Web API).

Once we have entered all the employees, we can test our solution using the GET Employees for company (page 2 size 2) request that we already prepared for you in our bonus 2 file from the 16-Paging in ASP.NET Core Web API folder:

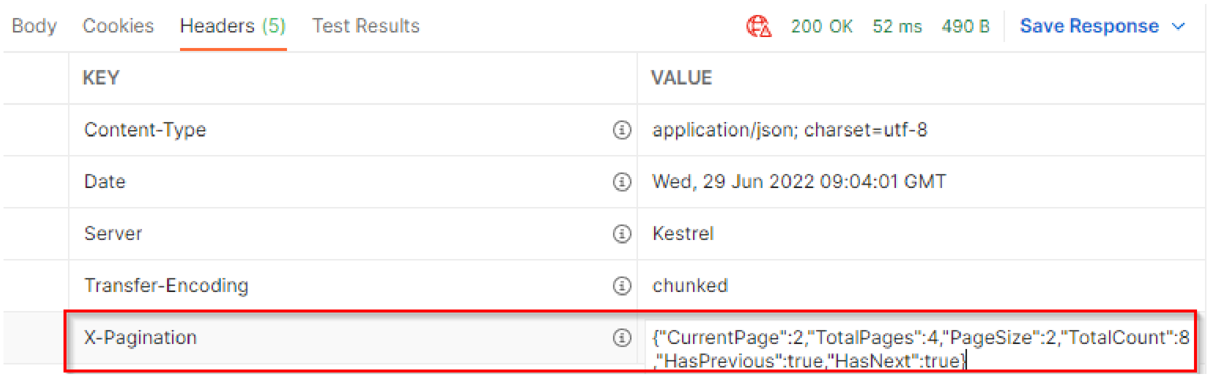


```
https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-
2D54A9991870/employees?pageNumber=2&pageSize=2
```

```
[
    {
        "employeeId": "915603fb-139f-4a62-8815-c991a1092b56",
        "name": "Kirk Metha",
        "age": 30,
        "position": "Marketing expert"
    },
    {
        "employeeId": "883b7158-8893-4a73-ab6c-5f041c6da593",
        "name": "Martha Grown",
        "age": 35,
        "position": "Marketing expert II"
    }
]
```

And we can see we have the correct result.

Also, we can inspect the **Headers** tab of the response, and we will find our metadata:



KEY	VALUE
Content-Type	application/json; charset=utf-8
Date	Wed, 29 Jun 2022 09:04:01 GMT
Server	Kestrel
Transfer-Encoding	chunked
X-Pagination	{"CurrentPage":2,"TotalPages":4,"PageSize":2,"TotalCount":8,"HasPrevious":true,"HasNext":true}

Great. We have the Paging implemented. Play around with the pageNumber and pageSize query parameters, paying attention to the data and X-Pagination header values returned.

12 FILTERING WITH DAPPER AND WEB API

In this chapter, we are going to cover filtering in ASP.NET Core Web API with Dapper. We already know what the filtering is – from the main book – so we will go straight to the implementation part.

12.1 Filtering Implementation

We have the `Age` property in our `Employee` class. Let's say we want to find out which employees are between the ages of 26 and 29. We also want to be able to enter just the starting age — and not the ending one — and vice versa.

To implement that, we have to modify our `EmployeeParameters` class:

```
public class EmployeeParameters : RequestParameters
{
	public int MinAge { get; set; } = 0;
	public int MaxAge { get; set; } = int.MaxValue;

	public bool ValidAgeRange => MaxAge > MinAge;
}
```

Here we have two `int` properties and both assigned initial values. We also have a validation property to verify that `MaxAge` is greater than `MinAge`. We are not using the `uint` types here, as we did in the main book, to avoid some Dapper/SQL `uint` errors and warnings.

Now, we can modify the `GetEmployees` service method by adding a validation check as a first statement:

```
public async Task<(IEnumerable<EmployeeDto> employees, MetaData metaData)>
    GetEmployees(Guid companyId, EmployeeParameters employeeParameters)
{
    if (!employeeParameters.ValidAgeRange)
        throw new MaxAgeRangeBadRequestException();

    var company = await _repository.Company.GetCompany(companyId);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

    var employeesWithMetaData = await _repository.Employee
        .GetEmployees(companyId, employeeParameters);
    var employees = employeesWithMetaData
        .Select(e => new EmployeeDto(e.EmployeeId, e.Name, e.Age, e.Position));

    return (employees: employees, metaData: employeesWithMetaData.MetaData);
}
```

Of course, as you did with all the previous exception classes, you can borrow this one from the source code and place it inside the **`Entities/Exceptions`** folder.

Then, we have to modify the existing query inside the **`EmployeeQuery`** class:

```
public const string SelectEmployeesQuery =
    @"SELECT COUNT(e.EmployeeId)
      FROM Employees AS e
      WHERE e.CompanyId = @companyId AND (e.Age >= @minAge AND e.Age <= @maxAge);

      SELECT e.EmployeeId, e.[Name], e.[Age], e.Position
      FROM Employees AS e
      WHERE e.CompanyId = @companyId AND (e.Age >= @minAge AND e.Age <= @maxAge)
      ORDER BY e.[Name]
      OFFSET @Skip ROWS FETCH NEXT @Take ROWS ONLY";
```

Here we add one more condition inside the **`WHERE`** clause. But we do that for both queries because we don't want to count all the employees if we have a filtering condition. This also means that we have to pass these two new parameters from the repository method:

```
public async Task<PagedList<EmployeeDto>> GetEmployees(Guid companyId,
    EmployeeParameters employeeParameters)
{
    var skip = (employeeParameters.PageNumber - 1) * employeeParameters.PageSize;

    var param = new DynamicParameters();
    param.Add("companyId", companyId, DbType.Guid);
    param.Add("skip", skip, DbType.Int32);
    param.Add("take", employeeParameters.PageSize, DbType.Int32);
    param.Add("minAge", employeeParameters.MinAge, DbType.Int32);
    param.Add("maxAge", employeeParameters.MaxAge, DbType.Int32);
    var query = EmployeeQuery.SelectEmployeesQuery;
...
```

And, that's all it takes.

We can now test this with all the Postman requests provided in our bonus file from the 17-Filtering in ASP.NET Core Web API folder, and we will get the expected results.

13 Searching with Dapper and Web API

In this chapter, we are going to see that the search implementation is pretty similar to the filter implementation, just we need to add a condition to our query. Of course, that condition is the main part of the searching functionality.

13.1 Searching Implementation

Let's start with the **EmployeeParameters** class modification:

```
public class EmployeeParameters : RequestParameters
{
        public int MinAge { get; set; } = 0;
        public int MaxAge { get; set; } = int.MaxValue;

        public bool ValidAgeRange => MaxAge > MinAge;

        public string? SearchTerm { get; set; }
}
```

The next thing we are going to do is to modify our query:

```
public const string SelectEmployeesQuery =
    @"SELECT COUNT(e.EmployeeId)
      FROM Employees AS e
      WHERE e.CompanyId = @companyId AND (e.Age >= @minAge AND e.Age <= @maxAge)
      AND ((@searchTerm LIKE N'') OR (CHARINDEX(@searchTerm, LOWER(e.[Name])) > 0));

      SELECT e.EmployeeId, e.[Name], e.[Age], e.Position
      FROM Employees AS e
      WHERE e.CompanyId = @companyId AND (e.Age >= @minAge AND e.Age <= @maxAge)
      AND ((@searchTerm LIKE N'') OR (CHARINDEX(@searchTerm, LOWER(e.[Name])) > 0))
      ORDER BY e.[Name]
      OFFSET @Skip ROWS FETCH NEXT @Take ROWS ONLY";
```

Here, we add our new condition and it consists of two parts. The first part is using the **LIKE** operator, which we need if we don't send the search parameter at all. On the other hand, if we send the search parameter, we use the **CHARINDEX** function that returns the position of our search parameter in the **Name** column. If it returns a number greater than zero it means the query found our search parameter in the column and will return that row in the result.

Now, we have to modify our repository method:

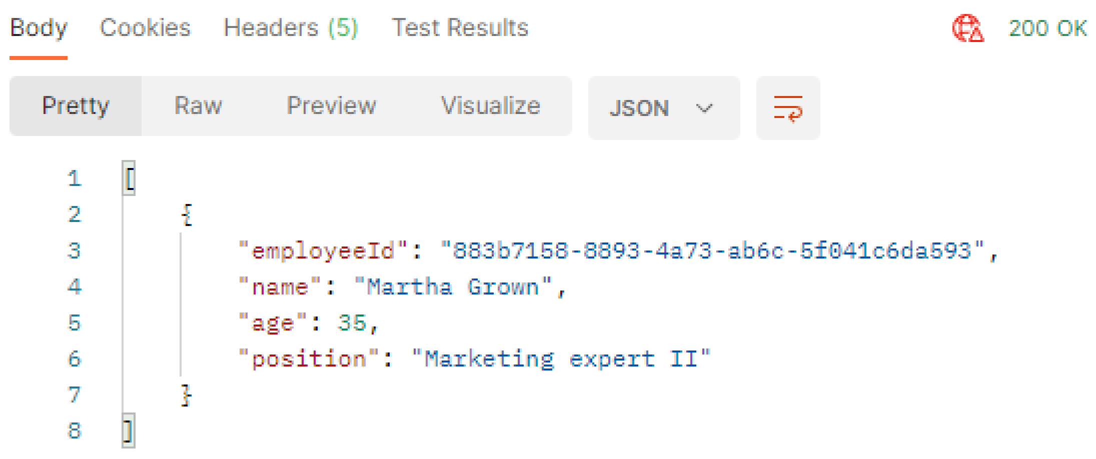
```
public async Task<PagedList<EmployeeDto>> GetEmployees(Guid companyId,
    EmployeeParameters employeeParameters)
{
    var skip = (employeeParameters.PageNumber - 1) * employeeParameters.PageSize;
    var searchTerm = !string.IsNullOrEmpty(employeeParameters.SearchTerm) ?
        employeeParameters.SearchTerm.Trim().ToLower() : string.Empty;

    var param = new DynamicParameters();
    param.Add("companyId", companyId, DbType.Guid);
    param.Add("skip", skip, DbType.Int32);
    param.Add("take", employeeParameters.PageSize, DbType.Int32);
    param.Add("minAge", employeeParameters.MinAge, DbType.Int32);
    param.Add("maxAge", employeeParameters.MaxAge, DbType.Int32);
    param.Add("searchTerm", searchTerm, DbType.String);
    var query = EmployeeQuery.SelectEmployeesQuery;
...
```

Here, we extract our search parameter inside the `searchTerm` variable by using the ternary operator. If the parameter exists, we just transform it to lower case. Otherwise, we store an empty string in our variable. Then, we just add this variable as a parameter for our Dapper query.

Again, as with the previous chapter, you can use already prepared requests in the 18-Searching in ASP.NET Core Web API folder to test this functionality.

We'll take a combination of features as an example:

https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?pageNumber=1&pageSize=4&minAge=32&maxAge=35&searchTerm=MA



```
[
    {
        "employeeId": "883b7158-8893-4a73-ab6c-5f041c6da593",
        "name": "Martha Grown",
        "age": 35,
        "position": "Marketing expert II"
    }
]
```

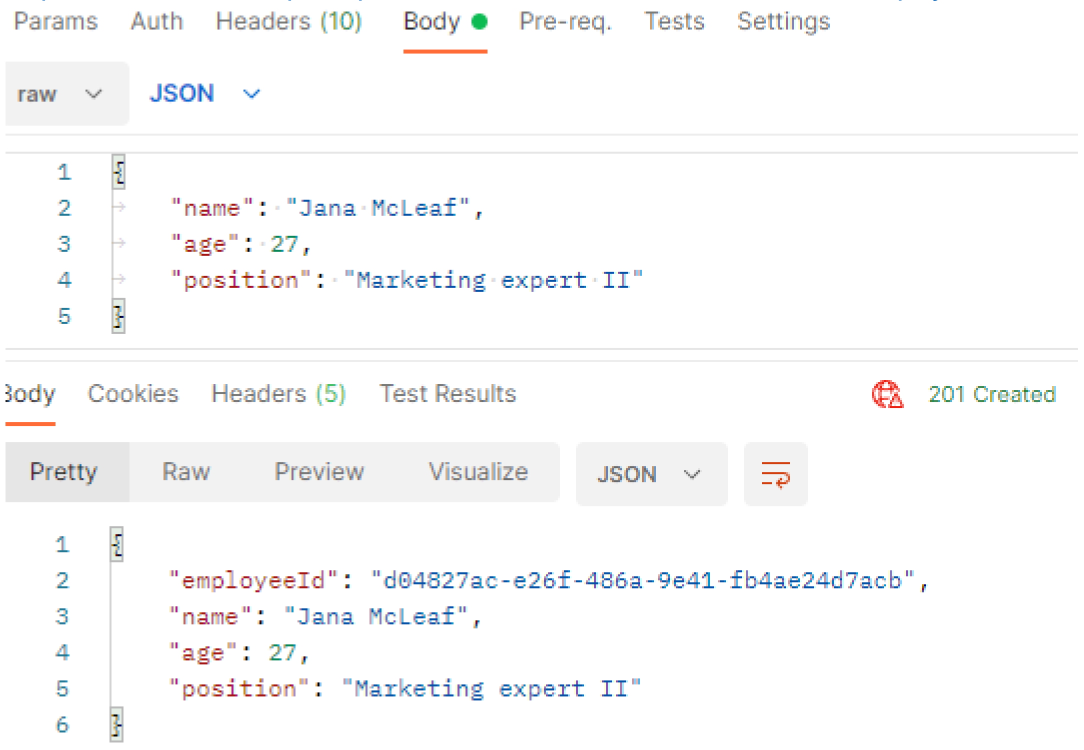
14 Sorting with Dapper and Web API

In our main book when we have discussed this functionality, we've shown the initial implementation and then the improved one. Now, since we already know all the basics, we will show just the final sorting implementation with the improved code.

That said, before we start with the implementation, we have to add one more employee to the database so we could properly show sorting examples by name and then by age.

To do that, let's execute our Postman request POST Employee for Company (Jana McLeaf), from the `19-Sorting in ASP.NET Core Web API` folder:

`https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees`



```
{
    "name": "Jana McLeaf",
    "age": 27,
    "position": "Marketing expert II"
}
```

```
{
    "employeeId": "d04827ac-e26f-486a-9e41-fb4ae24d7acb",
    "name": "Jana McLeaf",
    "age": 27,
    "position": "Marketing expert II"
}
```

Great. Now that we have all the employees in our database, we can start with the sorting implementation.

14.1 Sorting Implementation

The first thing we are going to do is to modify the **RequestParameters** class:

```
public abstract class RequestParameters
{
	const int maxPageSize = 50;
	public int PageNumber { get; set; } = 1;

	private int _pageSize = 10;
	public int PageSize
	{
		get
		{
			return _pageSize;
		}
		set
		{
			_pageSize = (value > maxPageSize) ? maxPageSize : value;
		}
	}

	public string? OrderBy { get; set; }
}
```

As you can see, the only thing we've added is the **OrderBy** property and we added it to the **RequestParameters** class because we can reuse it for other entities.

Now, we want to have a default sorting mechanism if no sort parameter is provided. For the employee entity, we want to use the name column:

```
public class EmployeeParameters : RequestParameters
{
	public EmployeeParameters() => OrderBy = "name";

	public int MinAge { get; set; } = 0;
	public int MaxAge { get; set; } = int.MaxValue;

	public bool ValidAgeRange => MaxAge > MinAge;

	public string? SearchTerm { get; set; }
}
```

In the constructor, we set the **OrderBy** property to name by default.

Next, we are going to create a new **OrderQueryBuilder** class inside the **Repository** project, and add a reusable code that we can use for different entities to create the sorting string:

```
public static class OrderQueryBuilder
{
	public static string CreateOrderQuery<T>(string orderByQueryString, char alias)
	{
		var orderParams = orderByQueryString.Trim().Split(',');
		var propertyInfos = typeof(T)
			.GetProperties(BindingFlags.Public | BindingFlags.Instance);
		var orderQueryBuilder = new StringBuilder();

		foreach (var param in orderParams)
		{
			if (string.IsNullOrWhiteSpace(param))
				continue;

			var propertyFromQueryName = param.Split(" ")[0];
			var objectProperty = propertyInfos
				.FirstOrDefault(pi => pi.Name
				    .Equals(propertyFromQueryName,
				       StringComparison.InvariantCultureIgnoreCase));

			if (objectProperty == null)
				continue;

			var direction = param.EndsWith(" desc") ? "desc" : "asc";

			orderQueryBuilder
			    .Append($"{alias}.{objectProperty.Name.ToString()}
{direction}, ");
		}

		var orderQuery = orderQueryBuilder.ToString().TrimEnd(',', ' ');

		return string.IsNullOrEmpty(orderQuery) ? "name asc" : orderQuery;
	}
}
```

This code is a bit different than the one we have in the main book, and we've marked the differences. The first difference is that we pass our alias to this method as a parameter – the one we use in our query. The second is that SQL can't recognize the `ascending` or `descending` keywords as we have in our main book's example. Here, we have to use either `asc` or `desc`. The last difference is just using the ternary operator here to return a default sorting string if our created `orderQuery` is null or empty. We need to add a using statement for `System.Reflection`.

Now, we have to modify the query itself inside the `EmployeeQuery` class:

```
public static string SelectEmployeesQuery(string orderBy) =>
    @$"SELECT COUNT(e.EmployeeId)
        FROM Employees AS e
        WHERE e.CompanyId = @companyId AND (e.Age >= @minAge AND e.Age <= @maxAge)
        AND ((@searchTerm LIKE N'') OR (CHARINDEX(@searchTerm, LOWER(e.[Name])) > 0));

        SELECT e.EmployeeId, e.[Name], e.[Age], e.Position
        FROM Employees AS e
        WHERE e.CompanyId = @companyId AND (e.Age >= @minAge AND e.Age <= @maxAge)
        AND ((@searchTerm LIKE N'') OR (CHARINDEX(@searchTerm, LOWER(e.[Name])) > 0))
        ORDER BY {orderBy}
        OFFSET @Skip ROWS FETCH NEXT @Take ROWS ONLY";
```

As we can see, this is no longer a constant string but a static method that accepts a single string parameter and returns a string.

Finally, we have to modify the repository method:

```
public async Task<PagedList<EmployeeDto>> GetEmployees(Guid companyId,
    EmployeeParameters employeeParameters)
{
    var skip = (employeeParameters.PageNumber - 1) * employeeParameters.PageSize;
    var searchTerm = !string.IsNullOrEmpty(employeeParameters.SearchTerm) ?
        employeeParameters.SearchTerm.Trim().ToLower() : string.Empty;
    var orderBy = OrderQueryBuilder
        .CreateOrderQuery<EmployeeDto>(employeeParameters.OrderBy, 'e');
    var query = EmployeeQuery.SelectEmployeesQuery(orderBy);

    var param = new DynamicParameters();

...
```

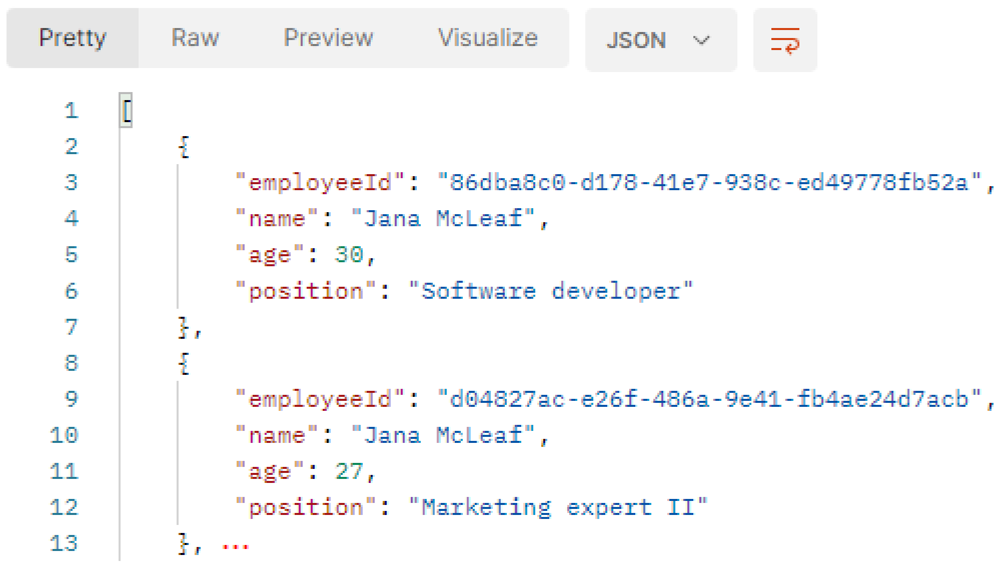
So, we create the `orderBy` variable and store the sorting string that we get by calling the `CreateOrderQuery` method and passing two arguments. The first one is the parameter sent from the client, and the second one is the alias that we use in our query. We also use our modified `SelectEmployeesQuery` method to create our complete SQL statement with the `orderBy` string.

And that's all.

We can now use all the requests from our Postman bonus file **19-Sorting in ASP.NET Core Web API** folder to test this implementation and confirm that everything works well.

For example, let's use the first one:

`https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?orderBy=name,age desc`



```
[
    {
        "employeeId": "86dba8c0-d178-41e7-938c-ed49778fb52a",
        "name": "Jana McLeaf",
        "age": 30,
        "position": "Software developer"
    },
    {
        "employeeId": "d04827ac-e26f-486a-9e41-fb4ae24d7acb",
        "name": "Jana McLeaf",
        "age": 27,
        "position": "Marketing expert II"
    }, ...
```

And we can see our employees are sorted by the name ascending first and then by the age descending.

15 Dapper and Identity Authentication

In our main book, in JWT and Refresh Token sections, we've learned a lot about the authentication, authorization, roles, and refresh token actions with ASP.NET Core Identity.

Now, that we have a fair amount of knowledge on the topic, we are going to expand that knowledge by adding Dapper into the story.

Of course, by default, ASP.NET Core Identity makes use of the Entity Framework Core data model, to work with the database. We've seen how easy it was to integrate both inside a single project. But with Dapper, it is a different story. There is no default Identity support for Dapper.

Still, we have several options to work with.

The first option, as strange as it might sound, is not to mix Dapper with Identity, but to use EF Core instead. This is the easiest solution. Even Dapper creators support the combination of both EF Core and Dapper in the same project if it makes sense. For us, this makes sense because using EF Core is restricted only to the authentication/authorization actions, and it doesn't compromise any of the Dapper queries for the main project's logic. So, we would only use EF Core for the registration, login, verify credentials, etc. actions. This means that the queries related to the main business logic are still written with Dapper and we have complete control over the SQL statements and the performance of our queries, which matters the most. Of course, using EF Core with Identity is explained in our main book.

The second option is to write our wrappers for the ASP.NET Core Identity providers. We've seen solutions like this one, where you wrap only what you need and register your wrapper stores with the Identity package. We will not cover that in this book since it would take a lot of time and code to cover all that needs to be covered.

The third option is to use one of the existing packages created by other developers who already needed the wrappers we've talked about in the second option. This is a nice solution, but all of those packages are not extendable as the EF Core is. As you remember, with EF Core, we can add new properties to the `User` class that inherits from the `IdentityUser` and those properties would be migrated as columns in our database.

But, where is the package - there is the GitHub open source project. So, instead of a package installation, we can use the source project, include it in our solution and modify it to fit our needs. That's exactly what we are going to show in this section.

15.1 Integrating AspNetCore.Identity.Dapper Project

We are going to use an open-source project (https://github.com/simonfaltum/AspNetCore.Identity.Dapper), clone it, and integrate it into our solution.

That said, in our source code (15 - Dapper and Identity), we can find the `AspNetCore.Identity.Dapper` project, which we can copy and then paste inside your solution. Then, we have to add this project as an existing project to our solution (right-click on a solution => Add => Existing Project), and also add a reference from that project to our main project.

We have already made all the required changes in the project to support the additional properties we've worked with in our main book (FirstName, LastName, RefreshToken, and RefreshTokenExpiryTime).

Let's guide you through the changes.

If we expand the imported project and open the Models folder, we are going to see two classes. Let's open the `ApplicationUser` class:

```
public class ApplicationUser : IdentityUser
{
    public int UserType { get; set; }
    public bool IsActive { get; set; } = true;
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public string? RefreshToken { get; set; }
    public DateTime RefreshTokenExpiryTime { get; set; } = DateTime.UtcNow;

    internal List<Claim>? Claims { get; set; }
    internal List<UserRole>? Roles { get; set; }
    internal List<UserLoginInfo>? Logins { get; set; }
    internal List<UserToken>? Tokens { get; set; }
}
```

This class already extends the `IdentityUser` class, and we've extended it even more with our required properties.

We have modified one more class in the project, the `UsersProvider` class, and we can find it inside the `Providers` folder. In this class, we've changed the SQL statements for the `CreateAsync` and `UpdateAsync` methods, to include additional parameters for the first name, last name, and refresh token fields.

We didn't change anything else in the project. The project as it is right now perfectly fits our needs.

Now, to configure Identity in our main project, we are going to open the `ServiceExtensions` class, and add another method:

```
public static void ConfigureIdentity(this IServiceCollection services, IConfiguration
configuration)
{
       services.AddIdentity<ApplicationUser, ApplicationRole>(o =>
       {
              o.Password.RequireDigit = true;
              o.Password.RequireLowercase = false;
              o.Password.RequireUppercase = false;
              o.Password.RequireNonAlphanumeric = false;
              o.Password.RequiredLength = 10;
              o.User.RequireUniqueEmail = true;
       })
       .AddDapperStores(opt =>
       {
              opt.ConnectionString =
configuration.GetConnectionString("SqlConnection");
       })
       .AddDefaultTokenProviders();
}
```

This code is almost the same as the one we used in the main book to register Identity. Of course, there are some differences. Here, we use the ApplicationUser and ApplicationRole classes, and also we call the AddDapperStores method to register the Dapper implementation of the ASP.NET Core Identity stores. This method accepts an Action<DBProviderOptions> delegate as a parameter, and if we inspect the DBProviderOptions class in our Identity.Dapper project, we can see that it expects two properties to be configured:

```
public class DBProviderOptions
{
    public string DbSchema { get; set; } = "dbo";

    public string? ConnectionString { get; set; }
}
```

The DbSchema has a default dbo value and that's the reason why we didn't configure it next to the ConnectionString in our ConfigureIdentity method. Our schema is dbo so we don't have to change anything with that. If your schema is a different one, you can configure that next to the connection string configuration in the AddDapperStores method.

For this, we need to include some additional namespaces:

```
using AspNetCore.Identity.Dapper;
using AspNetCore.Identity.Dapper.Models;
using Microsoft.AspNetCore.Identity;
```

We can use AspNetCore.Identity namespace because it is already installed inside the Identity.Dapper project.

Now, we can navigate to the Program class, and call our extension method:

```
builder.Services.ConfigureServiceManager();
builder.Services.AddAuthentication();
builder.Services.ConfigureIdentity(builder.Configuration);
```

Here we have to add the authentication first and then call our method.

And, let’s also add the authentication middleware to the application’s request pipeline:

```
app.UseAuthentication();
app.UseAuthorization();
```

That’s it. We can move on to database creation.

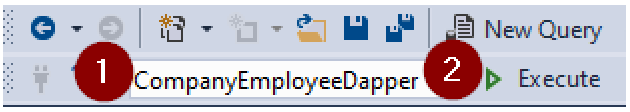
15.2 Creating Tables and Inserting Roles

To create all the required tables for Identity usage, we are going to use an already prepared SQL script. The name of the script is IdentityTables, and we can find it in our source code for this chapter.

With this script, we create all seven tables needed for Identity to work, and we also add our extra required columns inside the AspNetUsers table:

```
CREATE TABLE [AspNetUsers](
       [Id] [nvarchar](450) NOT NULL,
       [FirstName] [nvarchar](50) NULL,
       [LastName] [nvarchar](50) NULL,
       [UserName] [nvarchar](256) NULL,
       [NormalizedUserName] [nvarchar](256) NULL,
       [Email] [nvarchar](256) NULL,
       [NormalizedEmail] [nvarchar](256) NULL,
       [EmailConfirmed] [bit] NOT NULL,
       [PasswordHash] [nvarchar](max) NULL,
       [SecurityStamp] [nvarchar](max) NULL,
       [ConcurrencyStamp] [nvarchar](max) NULL,
       [PhoneNumber] [nvarchar](max) NULL,
       [PhoneNumberConfirmed] [bit] NOT NULL,
       [TwoFactorEnabled] [bit] NOT NULL,
       [LockoutEnd] [datetimeoffset](7) NULL,
       [LockoutEnabled] [bit] NOT NULL,
       [AccessFailedCount] [int] NOT NULL,
       [UserType] [int] NOT NULL,
       [IsActive] [bit] NOT NULL,
       [RefreshToken] [nvarchar](max) NULL,
       [RefreshTokenExpiryTime] [datetime] NULL,
 CONSTRAINT [PK_AspNetUsers] PRIMARY KEY CLUSTERED
```

So, we can just open the file in our SQL Management Studio (File => Open =>File), select the database we want these tables in, and hit the `Execute` button:



Now that we have our tables, we can add roles.

As we did in our main book, we are going to use migrations to add the roles.

So, in the `Migrations` folder, let's create a new `AddRolesToDb_202218060003` class:

```
[Migration(202218060003)]
public class AddRolesToDb_202218060003 : Migration
{
    public override void Down()
    {
        Delete.FromTable("AspNetRoles")
            .Row(new ApplicationRole
            {
                Name = "Manager",
                NormalizedName = "MANAGER"
            })
            .Row(new ApplicationRole
            {
                Name = "Administrator",
                NormalizedName = "ADMINISTRATOR"
            });
    }

    public override void Up()
    {
        Insert.IntoTable("AspNetRoles")
            .Row(new ApplicationRole
            {
                Name = "Manager",
                NormalizedName = "MANAGER"
            })
            .Row(new ApplicationRole
            {
                Name = "Administrator",
                NormalizedName = "ADMINISTRATOR"
            });
    }
}
```

We are already familiar with this code and FluentMigrator so there is nothing new here. We need to add two using statements:

```
using AspNetCore.Identity.Dapper.Models;
using FluentMigrator;
```

At this point, we can start our app. Once it's started, we can inspect the AspNetRoles table and verify that both roles are migrated successfully.

Now, we have everything prepared and we can follow all the steps from the main book to implement the registration, login, and refresh token logic.

15.3 User Creation

To start with the user creation action, we are going to add a new controller inside the `Presentation/Controllers` folder:

```
[Route("api/authentication")]
[ApiController]
public class AuthenticationController : ControllerBase
{
       private readonly IServiceManager _service;

       public AuthenticationController(IServiceManager service) => _service = service;
}
```

Add using statements:

```
using Microsoft.AspnetCore.Mvc;
using Service.Contracts;
```

The next thing we have to do is to create a `UserForRegistrationDto` record in the `Shared/DataTransferObjects` folder:

```
public record UserForRegistrationDto
{
    public string? FirstName { get; init; }
    public string? LastName { get; init; }
    [Required(ErrorMessage = "Username is required")]
    public string? UserName { get; init; }
    [Required(ErrorMessage = "Password is required")]
    public string? Password { get; init; }
    public string? Email { get; init; }
    public string? PhoneNumber { get; init; }
    public ICollection<string>? Roles { get; init; }
}
```

Add a using statement:

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

To continue, we are going to create a new `IAuthenticationService` interface inside the `Service.Contracts` project:

```
public interface IAuthenticationService
{
    Task<IdentityResult> RegisterUser(UserForRegistrationDto userForRegistration);
}
```

Of course, the **IdentityResult** comes from the **Microsoft.Extensions.Identity.Core** package, we have to install it inside the **Service.Contracts** project. Next, add a using statement for the Shared.DataTransferObjects project.

Now that we have the interface, we need to create an **AuthenticationService** class inside the **Service** project:

```
internal sealed class AuthenticationService : IAuthenticationService
{
    private readonly ILoggerManager _logger;
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly IConfiguration _configuration;

    public AuthenticationService(ILoggerManager logger,
        UserManager<ApplicationUser> userManager, IConfiguration configuration)
    {
        _logger = logger;
        _userManager = userManager;
        _configuration = configuration;
    }
}
```

For this to work, **we have to add the reference** from the **AspNetCore.Identity.Dapper** project to the **Service** project, and also we have to include a couple of namespaces:

```
using AspNetCore.Identity.Dapper.Models;
using Contracts;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Configuration;
using Service.Contracts;

using Shared.DataTransferObjects;
```

Now, we can implement a missing method:

```
public async Task<IdentityResult> RegisterUser(UserForRegistrationDto
userForRegistration)
{
    var user = new ApplicationUser
    {
        FirstName = userForRegistration.FirstName,
        LastName = userForRegistration.LastName,
        Email = userForRegistration.Email,
        UserName = userForRegistration.UserName,
        PhoneNumber = userForRegistration.PhoneNumber
    };
    var result = await _userManager.CreateAsync(user, userForRegistration.Password);

    if (result.Succeeded)
    {
        await _userManager.AddToRolesAsync(user, userForRegistration.Roles);
    }

    return result;
}
```

Since we don't use `AutoMapper`, we have to do the mapping ourselves. Everything else is the same as what we've seen in the main book using EF Core. So, we can see that as we normally use the `UserManager` store for our required actions, we now have a custom implementation of that store.

We want to provide this service to the caller through `ServiceManager` and for that, we have to modify the `IServiceManager` interface first:

```
public interface IServiceManager
{
    ICompanyService CompanyService { get; }
    IEmployeeService EmployeeService { get; }
    IAuthenticationService AuthenticationService { get; }
}
```

And also the `ServiceManager` class:

```
public sealed class ServiceManager : IServiceManager
{
    private readonly Lazy<ICompanyService> _companyService;
    private readonly Lazy<IEmployeeService> _employeeService;
    private readonly Lazy<IAuthenticationService> _authenticationService;

    public ServiceManager(IRepositoryManager repositoryManager,
        ILoggerManager logger, UserManager<ApplicationUser> userManager,
        IConfiguration configuration)
    {
        _companyService = new Lazy<ICompanyService>(() =>
            new CompanyService(repositoryManager, logger));
        _employeeService = new Lazy<IEmployeeService>(() =>
            new EmployeeService(repositoryManager, logger));
        _authenticationService = new Lazy<IAuthenticationService>(() =>
            new AuthenticationService(logger, userManager, configuration));
    }

    public ICompanyService CompanyService => _companyService.Value;
    public IEmployeeService EmployeeService => _employeeService.Value;
    public IAuthenticationService AuthenticationService
        => _authenticationService.Value;
}
```

We will add the required using statements:

```
using AspNetCore.Identity.Dapper.Models;
using Contracts;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Configuration;
using Service.Contracts;
```

Finally, it is time to create the `RegisterUser` action:

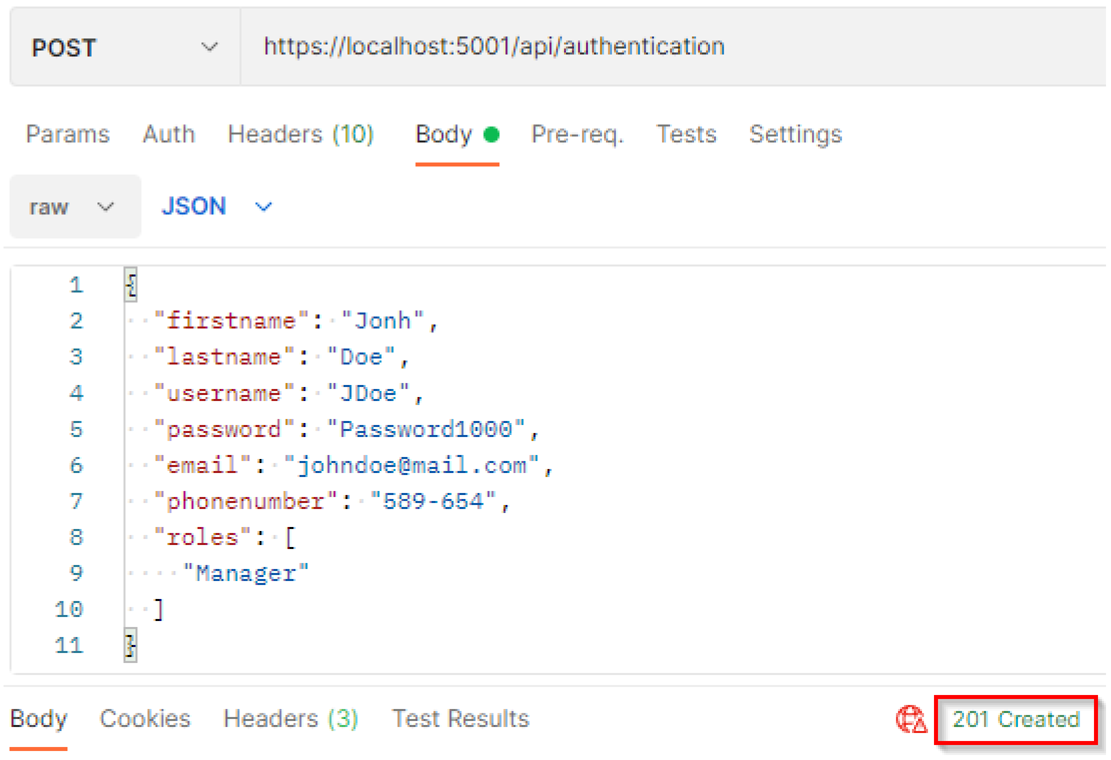
```
[HttpPost]
public async Task<IActionResult> RegisterUser([FromBody] UserForRegistrationDto
userForRegistration)
{
	var result = await _service.AuthenticationService
		.RegisterUser(userForRegistration);
	if (!result.Succeeded)
	{
		foreach (var error in result.Errors)
		{
			ModelState.TryAddModelError(error.Code, error.Description);
		}

		return BadRequest(ModelState);
	}

	return StatusCode(201);
}
```

We will add the required using statement:

```
using Shared.DataTransferObjects;
```

With all this in place, we can now test this with Postman using the POST Register user request that we already prepared for you in our bonus 2 file from the `27-JWT and Identity in ASP.NET Core` folder:

https://localhost:5001/api/authentication



https://localhost:5001/api/authentication

```
{
  "firstname": "Jonh",
  "lastname": "Doe",
  "username": "JDoe",
  "password": "Password1000",
  "email": "johndoe@mail.com",
  "phonenumber": "589-654",
  "roles": [
    "Manager"
  ]
}
```

And we get the 201 Created result.

You can also check the database to see that you have a new user created (AspNetUsers table) and also a role assigned to that user (AspNetUserRoles table).

Also, if you run other Postman requests (invalid ones) you will get the required error responses.

15.4 User Authentication and JWT

Since we have explained everything we need to know about JWT in our main book, we can move straight away to the implementation of the authentication action.

To start with it, we are going to modify the `appsettings.json` file:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "sqlConnection": "server=.; database=CompanyEmployeeDapper; Integrated
Security=true; Encrypt=false",
    "masterConnection": "server=.; database=master; Integrated Security=true;
Encrypt=false"
  },
  "JwtSettings": {
    "validIssuer": "CodeMazeAPI",
    "validAudience": "https://localhost:5001"
  },
  "AllowedHosts": "*"
}
```

We also have to create a secret key inside the environment variable with the familiar command:

```
setx SECRET "CodeMazeSecretKey" /M
```

We can now modify the `ServiceExtensions` class:

```
public static void ConfigureJWT(this IServiceCollection services, IConfiguration
configuration)
{
	var jwtSettings = configuration.GetSection("JwtSettings");
	var secretKey = Environment.GetEnvironmentVariable("SECRET");

	services.AddAuthentication(opt =>
	{
		opt.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
		opt.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
	})
	.AddJwtBearer(options =>
	{
		options.TokenValidationParameters = new TokenValidationParameters
		{
			ValidateIssuer = true,
			ValidateAudience = true,
			ValidateLifetime = true,
			ValidateIssuerSigningKey = true,

			ValidIssuer = jwtSettings["validIssuer"],
			ValidAudience = jwtSettings["validAudience"],
			IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey))
		};
	});
}
```

Again, there is nothing different here from what we have in our main book. Just, for this to work, we have to install the `Microsoft.AspNetCore.Authentication.JwtBearer` library.

Also, we require some additional namespaces:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Identity;
using Microsoft.IdentityModel.Tokens;
using System.Text;
```

After this, all we have to do is to call this method in the `Program` class:

```
builder.Services.AddAuthentication();
builder.Services.ConfigureIdentity(builder.Configuration);
builder.Services.ConfigureJWT(builder.Configuration);
```

Great.

Now, let's protect our `GetCompanies` endpoint inside the `CompaniesController` by using the `[Authorize]` attribute:

```
[HttpGet]
[Authorize]
public async Task<IActionResult> GetCompanies()
{
    var companies = await _service.CompanyService.GetAllCompanies();

    return Ok(companies);
}
```

This attribute resides inside the `Microsoft.AspNetCore.Authorization` namespace.

At this point, if we try to get all the companies with an unauthorized user, we will get the 401 Unauthorized response.

So, we need to implement our authentication.

To do that, let's start with the `UserForAuthenticationDto` record:

```
public record UserForAuthenticationDto
{
       [Required(ErrorMessage = "User name is required")]
       public string? UserName { get; init; }
       [Required(ErrorMessage = "Password name is required")]
       public string? Password { get; init; }
}
```

Adding a using statement:

```
using System.ComponentModel.Annotations;
```

To continue, let's modify the `IAuthenticationService` interface:

```
public interface IAuthenticationService
{
    Task<IdentityResult> RegisterUser(UserForRegistrationDto userForRegistration);
    Task<bool> ValidateUser(UserForAuthenticationDto userForAuth);
    Task<string> CreateToken();
}
```

Now, let's add a new private variable inside the `AuthenticationService` class:

```
private readonly UserManager<ApplicationUser> _userManager;
private readonly IConfiguration _configuration;
private ApplicationUser? _user;
```

And then implement the missing methods:

```
public async Task<bool> ValidateUser(UserForAuthenticationDto userForAuth)
{
    _user = await _userManager.FindByNameAsync(userForAuth.UserName);

    var result = (_user != null &&
        await _userManager.CheckPasswordAsync(_user, userForAuth.Password));
    if (!result)
        _logger.LogWarn($"{nameof(ValidateUser)}: Authentication failed. Wrong user
name or password.");

    return result;
}

public async Task<string> CreateToken()
{
    var signingCredentials = GetSigningCredentials();
    var claims = await GetClaims();
    var tokenOptions = GenerateTokenOptions(signingCredentials, claims);

    return new JwtSecurityTokenHandler().WriteToken(tokenOptions);
}

private SigningCredentials GetSigningCredentials()
{
    var key = Encoding.UTF8.GetBytes(Environment.GetEnvironmentVariable("SECRET"));
    var secret = new SymmetricSecurityKey(key);

    return new SigningCredentials(secret, SecurityAlgorithms.HmacSha256);
}

private async Task<List<Claim>> GetClaims()
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.Name, _user.UserName)
    };

    var roles = await _userManager.GetRolesAsync(_user);
    foreach (var role in roles)
    {
        claims.Add(new Claim(ClaimTypes.Role, role));
    }

    return claims;
}

private JwtSecurityToken GenerateTokenOptions
    (SigningCredentials signingCredentials, List<Claim> claims)
{
    var jwtSettings = _configuration.GetSection("JwtSettings");
    var tokenOptions = new JwtSecurityToken(
        issuer: jwtSettings["validIssuer"],
        audience: jwtSettings["validAudience"],
        claims: claims,
        expires: DateTime.Now.AddMinutes(Convert.ToDouble(jwtSettings["expires"])),
        signingCredentials: signingCredentials
    );

    return tokenOptions;
}
```

For this to work, we require a few more namespaces:

```
using System.IdentityModel.Tokens.Jwt;
using Microsoft.IdentityModel.Tokens;
using System.Text;
using System.Security.Claims;
```

After the implementation of our missing methods and a few private helper methods, we just have to modify the appsettings file to add the expires property for the `JwtSettings` object:

```
"JwtSettings": {
  "validIssuer": "CodeMazeAPI",
  "validAudience": "https://localhost:5001",
  "expires": 5
},
```

As you can see, everything is still the same as the logic that we have used in our main book.

The last thing we have to do is to add a new action inside the `AuthenticationController`:

```
[HttpPost("login")]
public async Task<IActionResult> Authenticate([FromBody] UserForAuthenticationDto
user)
{
       if (!await _service.AuthenticationService.ValidateUser(user))
              return Unauthorized();

       return Ok(new
       {
```

```
                Token = await _service.AuthenticationService.CreateToken()
        });
}
```

Of course, we don't have our validation logic here because we didn't work with action filters in this book.

And that's it.

We can now use our Postman requests that we already prepared for you in our bonus 2 file from the `27-JWT and Identity in ASP.NET Core` folder to authenticate the user and obtain the token. Then, we can use that token inside a GET request to get access to the protected endpoint.

We will not show the refresh token implementation here because, as you can see, everything is the same as the logic in the main book regarding the implementation. So, just follow the mentioned steps and you will be good to go. Of course, you don't have to add the RefreshToken and Expiry properties and columns, since we already did that during the imported project modification.

The most important part was implementing the Identity stores, where we used Dapper (which was the main goal), and registering those custom stores as well. After that was done, as you could've seen, we were able to use Identity stores and methods as we did with EF Core.