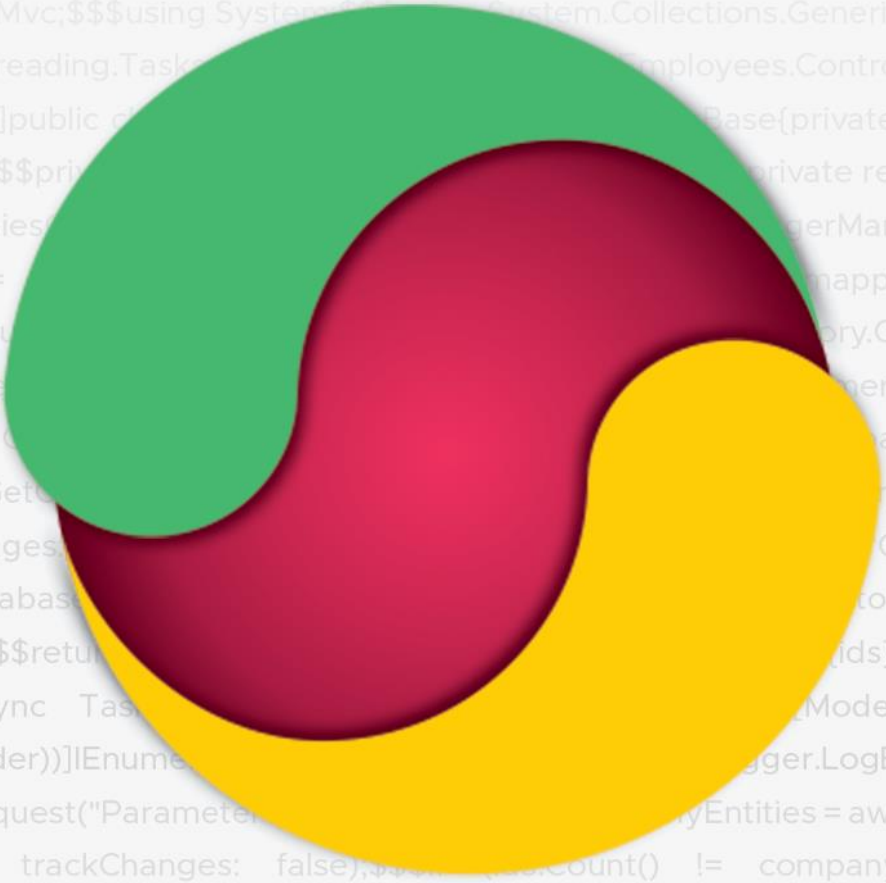
PREMIUM EDITION

Marinko Spasojevic
Vladimir Pecanac

ULTIMATE ASP.NET CORE 3 WEB API

From Complete Noob To Six-Figure Backend Developer



Made with ❤ by: CodeMaze

TABLE OF CONTENTS

About This Book

.NET Framework has been part of our lives for a long time now. .NET Core has been a major improvement in the sense that writing cross-platform and mobile-friendly web applications has never been easier. Being able to write platform-independent or containerized web applications is not a privilege like it was a few years ago. ASP.NET Core when paired with C# programming language provides us the freedom to write lightweight, optimized, flexible, and scalable web applications like we deserve.

But does that mean writing web applications is harder due to the increase in complexity?

We don't think so!

Through the years of work with our students and direct communication with our readers, we've developed an easy method that can get you up and running with ASP.NET Core Web API in a matter of days.

We make things easy to comprehend and we invest additional effort to make sure that the concepts we talk about are clear to EVERYONE without exception! Too many books are way more complicated than they should be.

This book is not.

We believe in learning through practice. Every concept, every piece of advice you find in this book is accompanied by the source code you can play around with.

But, that's not all!

Although the book is broken down into logical units, by the end you'll have a fully working and production-ready web application that can take on millions of requests and scale with ease. A step-by-step approach,

combined with a hands-on learning and gradual improvement of the codebase makes for a fun and non-frustrating learning experience.

☙ Who This Book is For

Everyone looking to improve their knowledge of building APIs by using the latest ASP.NET Core framework. Especially suited for beginners and intermediate developers who are looking for improving their knowledge or begin their journey in .NET web development.

The goal of this book is not only to improve you as a software developer but to help you get the job you want by providing real-world and down-to-earth examples.

Some of the concepts introduced towards the end of the book might be a bit complicated for a complete beginner and thus we recommend reading the book chapter by chapter and rereading some parts if necessary to be able to comprehend the material completely.

Some prior knowledge of C# is required, but other than that, every concept related to web development is described in detail in this book.

This book assumes you already have a decent knowledge in Object Oriented Programming and is here to help you organize and focus your thoughts so you can achieve greater results in your career or as a freelance developer.

If you master all the concepts in this book, we guarantee not only that you'll land yourself a great job opportunity, but that you'll get to your six-figure income goals in no-time.

☙ What You'll Learn

By the end of this book we expect you to know:

- How to get started and configure your application in a matter of minutes
- How to connect your application with a database and migrate data
- How to set up a scalable API architecture
- How to create a clean code base by using out-of-the-box middleware for global error handling and action filters
- How to Make API flexible with features such as paging, filtering, searching sorting and data shaping
- How to secure your API from unauthorized access
- How to optimize your API by making it asynchronous and by introducing caching, rate limiting and throttling
- How to ensure your API is properly tested and ready for production and then actually deploy it

And not only that but if you do put everything in this book to practice, we expect that you'll have a working application in a matter of minutes for every project you work on.

☙ What is the Best Way to Read This Book?

This book is supposed to be read in a sequence that is written. If you skip a chapter, you'll probably have a knowledge gap and you'll probably miss something important.

We've restrained ourselves from putting anything that's of no use to you in this book, and we've reduced theory as much as we could. There is still some theory here and there, but only what we think was crucial.

How This Book is Organized

This book is divided into separate chapters each one being a continuation to the previous. The model is simple and consists of two tables: Companies and Employees. It's used to simulate a simple workplace structure.

Here's a book breakdown by chapters:

- PROJECT CONFIGURATION - A simple introduction to the project and how-to guide on creating and configuring a new project. The goal of this chapter is to get things started.
- CONFIGURING A LOGGING SERVICE - We begin early with the logger service since we think logging is greatly underrated and often introduced pretty late in the real-world projects.
- DATABASE MODEL AND REPOSITORY PATTERN - We are introducing a database and EF Core as a base for the subsequent chapters. We also learn how to populate data that will be used throughout the book.
- HANDLING GET REQUESTS - Introducing the very basic principles of routing and then moving on to GET requests immediately.
- GLOBAL ERROR HANDLING - Introducing error handling on a global level early now that we have some code to work with and since we want to keep our codebase clean from the start.
- GETTING ADDITIONAL RESOURCES - Get by id resource implementation, an extension to the previous chapter with the improvements we've learned meanwhile.
- CONTENT NEGOTIATION - Content negotiation is a way to negotiate the wanted format of the response. Response formatting is important for almost any API.
- METHOD SAFETY AND METHOD IDEMPOTENCY - Before we start implementing concrete methods, we want the reader to know the difference between method types and why it matters.

- CREATING RESOURCES – This chapter is all about creating resources through POST requests. POST requests are used in forms, file uploads and other scenarios that require creating a new resource on the backend.
- WORKING WITH DELETE REQUESTS – Once the resource is created, we'll probably need a way to delete it too. "Hard" deletion is not often necessary or desirable, but sometimes we just want to dispose of the resource for good.
- WORKING WITH PUT REQUESTS – PUT requests are used to modify the existing resource on the server. In this chapter, we learn how to do it.
- WORKING WITH PATCH REQUESTS – PATCH requests are specialized requests used to modify only some parts of the resource by using a unique request format.
- VALIDATION - Now that we've implemented the important actions, we can introduce model validation and demonstrate the out-of-the-box mechanisms ASP.NET Core offers.
- ASYNCHRONOUS CODE - High volume APIs require asynchronicity, and we utilize C# features of async-await together with IQueryable to demonstrate how to make non-blocking and asynchronous calls to the API.
- ACTION FILTERS - Action filters in ASP.NET Core are a useful feature that can clean our code significantly and knowing how to utilize them is an important part of keeping our controllers clutter-free.
- PAGING - Paging is a must-have feature of every API. Together with filtering, searching, sorting and data-shaping it lifts our API to another level and gives it a dose of flexibility that is required of any serious API
- FILTERING - We're learning what filtering is and how to implement it in ASP.NET Core Web API.

- SEARCHING - We’re learning what searching is and how to implement it in ASP.NET Core Web API.
- SORTING - We’re learning what sorting is and how to implement it in ASP.NET Core Web API.
- DATA SHAPING - We’re learning what data shaping is and how to implement it in ASP.NET Core Web API.
- SUPPORTING HATEOAS - HATEAOS is one of the crucial REST requirements and one of the more advanced concepts of our book.
- WORKING WITH OPTIONS AND HEAD REQUESTS – A small chapter covering the potential usage of OPTIONS and HEAD HTTP requests.
- ROOT DOCUMENT - Root controller is a controller that usually gets hit by accident and we are going to transform it into a useful piece of code that can get API consumers started on the right track.
- VERSIONING APIS - There are four different ways of versioning API, each one has pros and cons. The reader is going to learn what those are and we’re going to implement the way we think is the best/
- CACHING – Knowing how to cache resources is a very important part of a versatile and scalable API. Caching reduces the stress on the server and saves the precious bandwidth.
- RATE LIMITING AND THROTTLING - Every high volume or even just public API needs a way to protect its endpoints from unwanted attacks and to optimize its traffic usage. This is the chapter where we’re going to tackle these concepts.
- JWT AND IDENTITY - API security is an important concept and one that cannot be avoided. It is also one of the hardest things to master and a pinnacle of our book. The reader is going to learn just how important security is and one great way to implement it.
- DOCUMENTING APIs – Swagger UI is a tool that helps us document our APIs. We are going to implement Swagger in our API and show its potential for API documentation.

- DEPLOYMENT TO IIS - Once the application is finished, we want to host it on a staging or production environment. The most used web server for .NET applications is IIS, so we are going to deploy the application to the hosting environment and run it on IIS.

About the Source Code

Source code is included with the book, and every example in the book can be found in the source code, too.

The projects are structured to follow the book chapters, and the source code from the previous chapter can be used as a starting point for the current chapter. Except of course the chapter one.

Bonus Materials

Depending on which version of the book you've bought, the book is accompanied by several bonus materials. These materials are by no means necessary but they do bring a lot of additional value. These bonus materials are:

- **Bonus 1**: The HTTP Reference Tables
  This bonus contains all the HTTP status codes, request and response headers, and MIME types that you'll ever encounter when working with HTTP and REST. You can use it as a reference material.
- **Bonus 2**: All the Postman requests we've used in this book. You copy-paste the requests from the book too, but we've prepared and organized all the requests, valid and invalid, so you don't have to.
- **Bonus 3**: The Workbook. This is a workbook with questions you can use to check if you've mastered the material so far. We haven't included the answers due to the simple fact that you'll have access to the private Facebook group where you can ask or help other community members. We believe this collaboration is a crucial part of really mastering this material.

- **Bonus 4:** A Guide ASP.NET Core Dockerization. Docker has become one of the crucial elements of cross-platform development. If you are interested in DevOps and Docker (and you should be), this is a great book that can help you finally grasp these concepts and even land you a job that has one of the highest satisfaction rates in the whole industry!
- **Bonus 5:** The Definitive Guide to Freelancing Platforms. As we've already said a few times, we want to help you all the way through. It's not important how much knowledge you can accumulate, but how you apply it. That's why we've compiled a list of Freelancing platforms you can use to sharpen your skills.
- **Bonus 6**: A private facebook group. This group is only available to those with a premium tier of the book. This is a group where you can share your experiences or ask for advice. We'll be monitoring this group closely and even join the discussions personally every day. A great place to connect and bring the game to another level.

Having all that in mind, let's get right into the material and learn how to create the best APIs together.

PROJECT CONFIGURATION

Configuration in .NET Core is very different from what we're used to in .NET Framework projects. We don't use the web.config file anymore, but instead use a built-in Configuration framework that comes out-of-the-box in .NET Core.

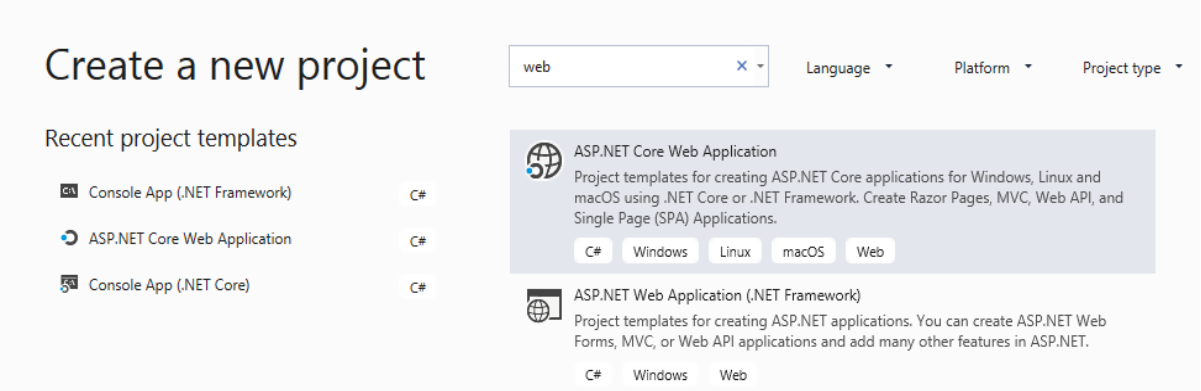
To be able to develop good applications, we need to understand how to configure our application and its services first.

In this section, we'll learn about configuration methods in the Startup class and set up our application. We will also learn how to register different services and how to use extension methods to achieve this.

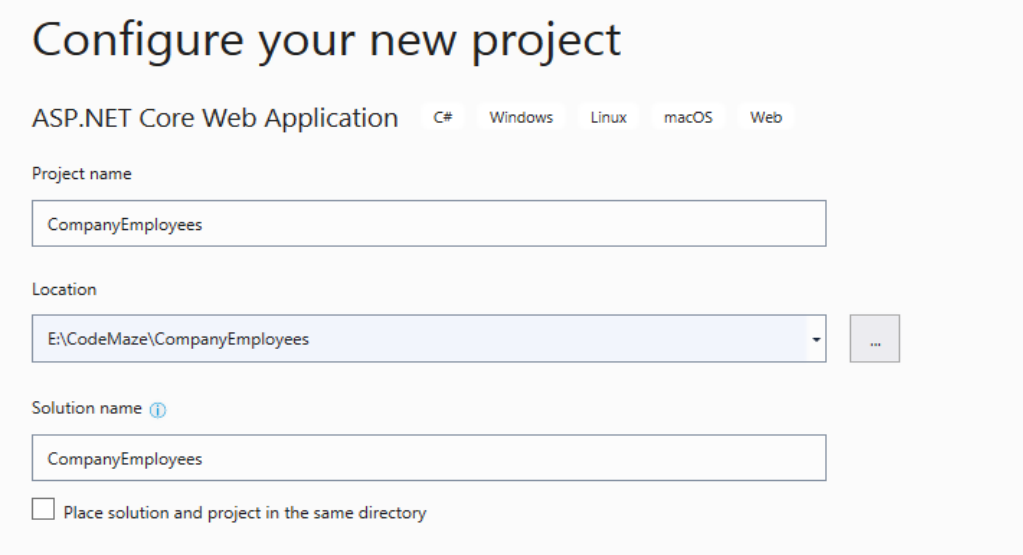
Of course, the first thing we need to do is to create a new project, so, let's dive right into it.

Creating a New Project

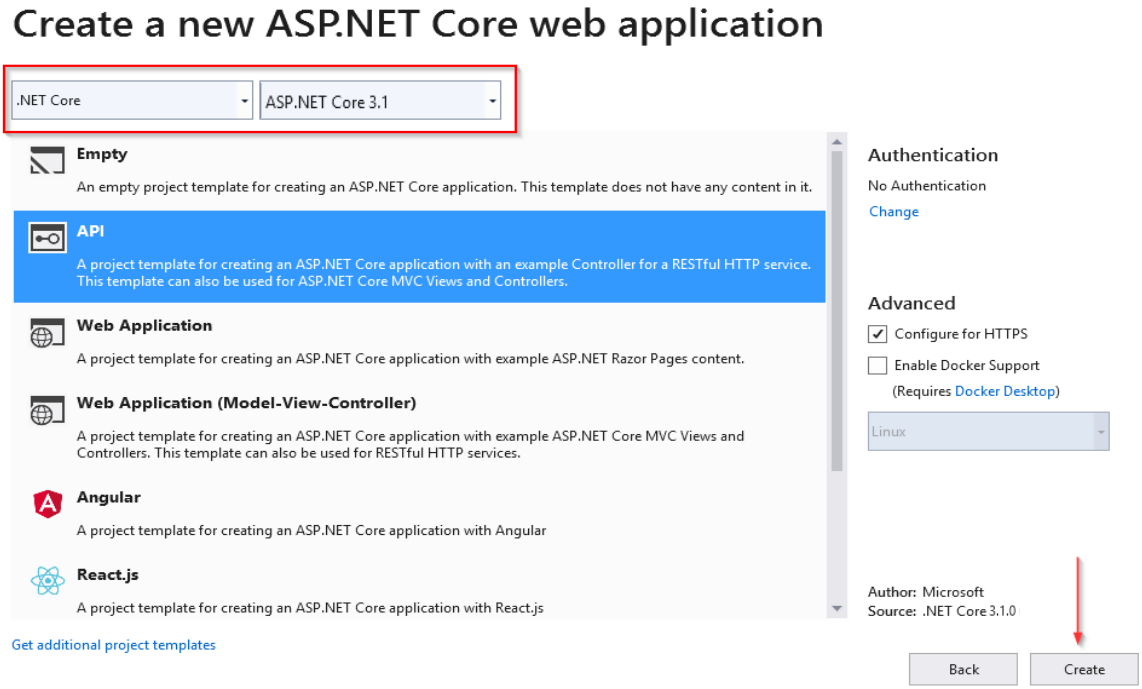
Let's open Visual Studio and create a new ASP.NET Core Web Application:



Now let’s choose a name and location for our project:



Next we want to choose a .NET Core and ASP.NET Core 3.1 from the dropdown lists respectively. Now we can proceed by clicking the Create button and the project will start initializing:



launchSettings.json File Configuration

After the project has been created, we are going to modify the `launchSettings.json` file, which can be found in the Properties section of the Solution Explorer window.

This configuration determines the launch behavior of the ASP.NET Core applications. As we can see, it contains both configurations to launch settings for IIS and self-hosted applications (Kestrel).

For now, let's change the `launchBrowser` property to `false` to prevent the web browser from launching on application start.

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:58753",
      "sslPort": 44370
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": false,
      "launchUrl": "weatherforecast"
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "CompanyEmployees": {
      "commandName": "Project",
      "launchBrowser": false,
      "launchUrl": "weatherforecast",
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

This is convenient, since we are developing a Web API project and we don't really need a browser to check our API out. We will use Postman (described later) for this purpose.

If you've checked *Configure for HTTPS* checkbox earlier in the setup phase, you will end up with two URLs in the applicationUrl section — one for HTTP, and one for HTTPS.

You'll also notice the `sslPort` property which indicates that our application, when running in IISExpress, will be configured for HTTPS (port 44370), too.

Additional info: *Take note that this HTTPS configuration is only valid in the local environment. You will have to configure a valid certificate and HTTPS redirection once you deploy the application.*

There is one more useful property for developing applications locally and that's the `launchUrl` property. This property determines which URL will the application navigate to initially. In order for `launchUrl` property to work, we need to set the `launchBrowser` property to true. So, for example, if we set the `launchUrl` property to `weatherforecast`, we will be redirected to `https://localhost:5001/weatherforecast` when we launch our application.

Program.cs and Startup.cs Explanations

`Program.cs` is the entry point to our application and it looks like this:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
```

```
                webBuilder.UseStartup<Startup>();
            });
}
```

If you are familiar with how things work in .NET Core 1.0, you will find this code considerably smaller than it used to be.

You might wonder why some parts are missing like the `UseKestrel()` or the `UseIISIntegration()`. The `CreateDefaultBuilder(args)` method encapsulates all that stuff and makes this code more readable, but it keeps all the magic present. You can still fine grain the configuration if you want to.

The `CreateDefaultBuilder(args)` method sets the default files and variables for the project and logger configuration. The fact that the logger is configured earlier in the bootstrapping process means we can log issues that happen during bootstrapping as well, which was a bit harder in previous versions.

After that, we can call `webBuilder.UseStartup<Startup>()` to initialize the `Startup` class too. The `Startup` class is mandatory in ASP.NET Core Web API projects. In the Startup class, we configure the embedded or custom services that our application needs.

When we open the Startup class, we can find the constructor and the two methods which we'll extend quite a few times during our application development.

As the method name indicates, the `ConfigureServices` method is used to do exactly that: configure our services. A service is a reusable part of the code that adds some functionality to our application.

In the `Configure` method, we are going to add different middleware components to the application's request pipeline.

Since larger applications could potentially contain a lot of different services, we can end up with a lot of clutter and unreadable code in the

`ConfigureServices` method. To make it more readable for the next person and for ourselves, we can structure the code into logical blocks and separate those blocks into extension methods.

Extension Methods and CORS Configuration

An extension method is inherently a static method. What makes it different from other static methods is that it accepts `this` as the first parameter, and `this` represents the data type of the object which will be using that extension method. We'll see what that means in a moment.

An extension method must be defined inside a static class. This kind of method extends the behavior of a type in .NET. Once we define an extension method, it can be chained multiple times on the same type of object.

So, let's start writing some code to see how it all adds up.

We are going to create a new folder `Extensions` in the project and create a new class inside that folder named `ServiceExtensions.` The ServiceExtensions class should be static.

```
public static class ServiceExtensions
{
}
```

Let's start by implementing something we need for our project immediately so we can see how extensions work.

The first thing we are going to do is to configure CORS in our application. CORS (Cross-Origin Resource Sharing) is a mechanism to give or restrict access rights to applications from different domains.

If we want to send requests from a different domain to our application, configuring CORS is mandatory. So, to start off, we'll add a code that allows all requests from all origins to be sent to our API:

```
public static void ConfigureCors(this IServiceCollection services) =>
    services.AddCors(options =>
    {
        options.AddPolicy("CorsPolicy", builder =>
            builder.AllowAnyOrigin()
            .AllowAnyMethod()
            .AllowAnyHeader());
    });
```

We are using basic CORS policy settings because allowing any origin, method, and header is okay for now. But we should be more restrictive with those settings in the production environment. More precisely, as restrictive as possible.

Instead of the `AllowAnyOrigin()` method which allows requests from any source, we can use the `WithOrigins("https://example.com")` which will allow requests from only from that concrete source. Also, instead of `AllowAnyMethod()` that allows all HTTP methods, we can use `WithMethods("POST", "GET")` that will allow only specific HTTP methods. Furthermore, you can make the same changes for the `AllowAnyHeader()` method by using, for example, the `WithHeaders("accept", "content-type")` method to allow only specific headers.

IIS Configuration

ASP.NET Core applications are by default self hosted, and if we want to host our application on IIS, we need to configure an IIS integration which will eventually help us with the deployment to IIS. To do that, we need to add the following code to the `ServiceExtensions` class:

```
public static void ConfigureIISIntegration(this IServiceCollection services) =>
    services.Configure<IISOptions>(options =>
    {

    });
```

We do not initialize any of the properties inside the options because we are fine with the default values for now. But if you need to fine tune the

configuration right away, you might want to take a look at the possible options:

Option	Default	Setting
`AutomaticAuthentication`	`true`	If `true`, the authentication middleware sets the `HttpContext.User` and responds to generic challenges. If `false`, the authentication middleware only provides an identity (`HttpContext.User`) and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication must be enabled in IIS for `AutomaticAuthentication` to function.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.
`ForwardClientCertificate`	`true`	If `true` and the `MS-ASPNETCORE-CLIENTCERT` request header is present, the `HttpContext.Connection.ClientCertificate` is populated.

Now, we mentioned extension methods are great for organizing your code and extending functionalities. Let's go back to our Startup class and modify the **`ConfigureServices`** and the **`Configure`** methods to support CORS and IIS integration now that we've written extension methods for those functionalities:

```
public void ConfigureServices(IServiceCollection services)
{
    services.ConfigureCors();
    services.ConfigureIISIntegration();

    services.AddControllers();
}
```

And let's add a few mandatory methods to our Configure method:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
            app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
```

```
    app.UseCors("CorsPolicy");

    app.UseForwardedHeaders(new ForwardedHeadersOptions
    {
        ForwardedHeaders = ForwardedHeaders.All
    });

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

We've added CORS and IIS configuration to the `ConfigureServices` method. Furthermore, CORS configuration has been added to the application's pipeline inside the `Configuration` method. But as you can see, there are some additional methods unrelated to IIS configuration. Let's go through those and learn what they do.

- `app.UseForwardedHeaders()` will forward proxy headers to the current request. This will help us during application deployment.
- `app.UseStaticFiles()` enables using static files for the request. If we don't set a path to the static files directory, it will use a `wwwroot` folder in our project by default.

Additional Code in the Startup Class

Configuration in .NET Core 3.1 is a bit different than it was in 2.2, so we have to make some changes in the Startup class. First, in the `ConfigureServices` method, instead of `AddMvc()` as used in 2.2, now we have `AddControllers()`. This method registers only the controllers in `IServiceCollection` and not Views or Pages because they are not required in the Web API project which we are building.

In the Configure method, we have `UseRouting()` and `UseAuthorization()` methods. They add routing and authorization features to our application, respectively.

Finally, we have the `UseEndpoints()` method with the `MapControllers()` method, which adds an endpoint for the controller's action to the routing without specifying any routes.

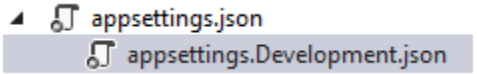
Microsoft advises that the order of adding different middlewares to the application builder is very important. So the `UseRouting()` method should be called before the `UseAuthorization()` method and `UseCors()` or `UseStaticFiles()` have to be called before the `UseRouting()` method.

Environment-Based Settings

While we develop our application, we use the "development" environment. But as soon as we publish our application, it goes to the "production" environment. Development and production environments should have different URLs, ports, connection strings, passwords, and other sensitive information.

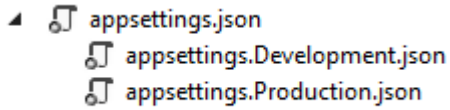
Therefore, we need to have a separate configuration for each environment and that's easy to accomplish by using .NET Core-provided mechanisms.

As soon as we create a project, we are going to see the `appsettings.json` file in the root, which is our main settings file, and when we expand it we are going to see the `appsetings.Development.json` file by default. These files are separate on the file system, but Visual Studio makes it obvious that they are connected somehow.



The apsettings.{EnvironmentSuffix}.json files are used to override the main appsettings.json file. When we use a key-value pair from the original file, we override it. We can also define environment-specific values too.

For the production environment, we should add another file: `appsettings.Production.json`:



The `appsettings.Production.json` file should contain the configuration for the production environment.

To set which environment our application runs on, we need to set up the `ASPNETCORE_ENVIRONMENT` environment variable. For example, to run the application in a production, we need to set it to the Production value on the machine we do the deployment to.

We can set the variable through the command prompt by typing `set ASPNETCORE_ENVIRONMENT=Production` in Windows or `export ASPNET_CORE_ENVIRONMENT=Production` in Linux.

ASP.NET Core applications use the value of that environment variable to decide which appsettings file to use accordingly. In this case, that will be `appsettings.Production.json`.

If we take a look at our `launchSettings.json` file, we are going to see that this variable is currently set to `Development`.

In the next chapter, we'll learn how to configure a Logger service because it's really important to have it configured as early in the project as possible.

CONFIGURING A LOGGING SERVICE

Why does logging messages matter so much during application development? While our application is in the development stage, it's easy to debug the code and find out what happened. But debugging in a production environment is not that easy.

That's why log messages are a great way to find out what went wrong and why and where the exceptions have been thrown in our code in the production environment. Logging also helps us more easily follow the flow of our application when we don’t have access to the debugger.

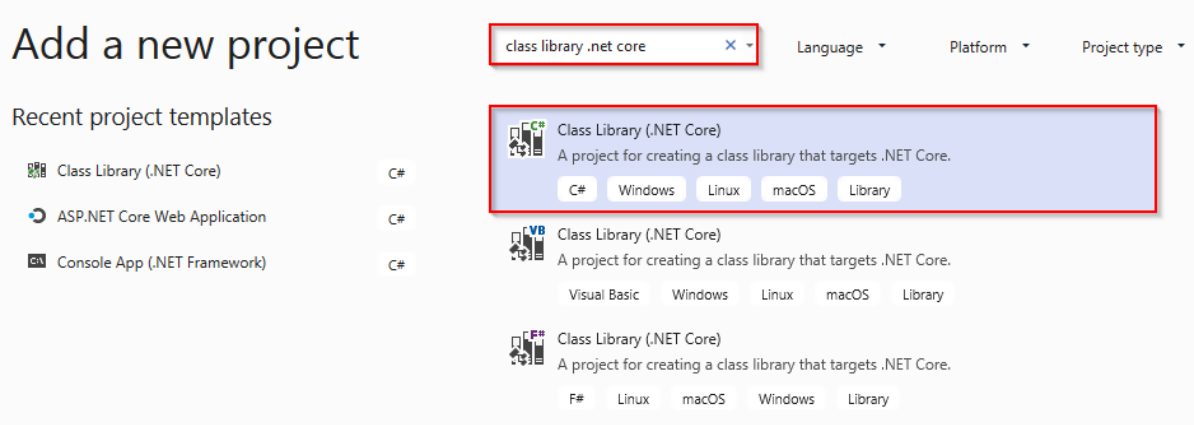
.NET Core has its own implementation of the logging mechanism, but in all our projects we prefer to create our custom logger service with the external logger library NLog.

That is exactly what we are going to do in this chapter.

Creating the Required Projects

Let’s create two new projects. In the first one named `Contracts`, we are going to keep our interfaces. We will use this project later on too, to define our contracts for the whole application. The second one, `LoggerService`, we are going to use to write our logger logic in.

To create a new project, right-click on the solution window, choose Add and then NewProject. Choose the Class Library (.NET Core) project:



Finally, name it `Contracts`. Do the same thing for the second project and name it `LoggerService`. Now that we have these projects in place, we need to reference them from our main project.

To do that, navigate to the solution explorer. Then in the `LoggerService` project, right click on `Dependencies` and choose the `AddReference` option. Under Projects, click Solution and check the `Contracts` project.

Now, in the main project right click on `Dependencies` and then click on `Add Reference`. Check the `LoggerService` checkbox to import it. Since we have referenced the `Contracts` project through the `LoggerService`, it will be available in the main project too.

Creating the ILoggerManager Interface and Installing NLog

Our logger service will contain four methods for logging our messages:

- Info messages
- Debug messages
- Warning messages
- Error messages

To achieve this, we are going to create an interface named `ILoggerManager` inside the `Contracts` project containing those four method definitions.

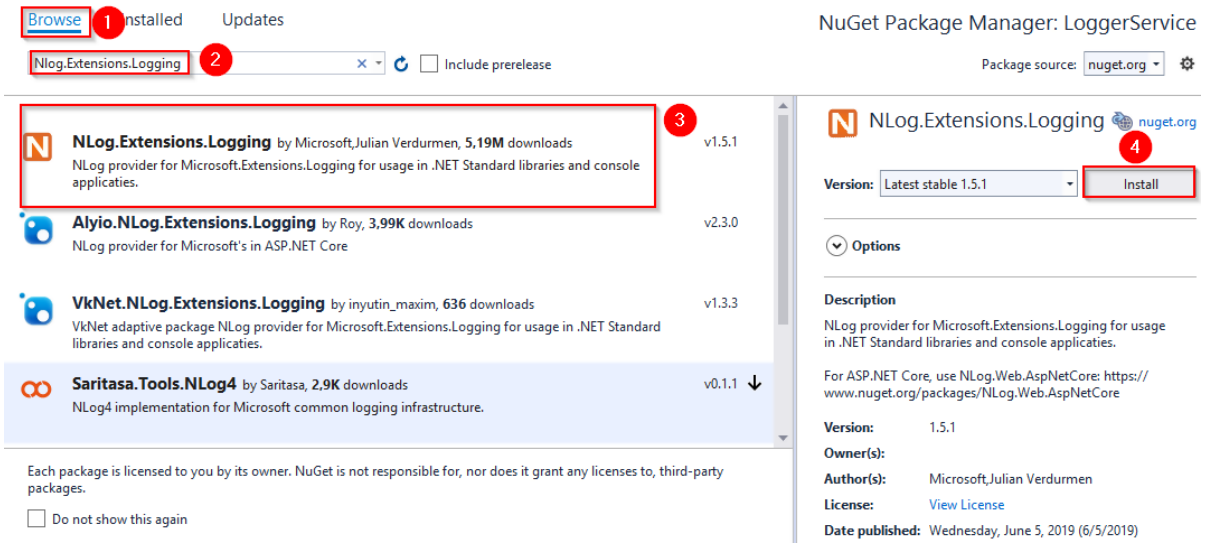
So, let's do that first:

```
public interface ILoggerManager
{
    void LogInfo(string message);
    void LogWarn(string message);
    void LogDebug(string message);
    void LogError(string message);
}
```

Before we implement this interface inside the **LoggerService** project, we need to install the **NLog** library in our **LoggerService** project. **NLog** is a logging platform for .NET which will help us create and log our messages.

We are going to show two different ways of adding the **NLog** library to our project.

1. In the **LoggerService** project, right click on the **Dependencies** and choose **Manage NuGet Packages**. After the NuGet Package Manager window appears, just follow these steps:



2. From the View menu, choose Other Windows and then click on the **Package Manager Console**. After the console appears, type:
`Install-Package NLog.Extensions.Logging -Version 1.5.1`

After a couple of seconds, **NLog** is up and running in our application.

Implementing the Interface and Nlog.Config File

In the `LoggerService` project, we are going to create a new class: `LoggerManager`. Now let's have it implement the `ILoggerManager` interface we previously defined:

```
public class LoggerManager : ILoggerManager
{
    private static ILogger logger = LogManager.GetCurrentClassLogger();

    public LoggerManager()
    {
    }

    public void LogDebug(string message)
    {
        logger.Debug(message);
    }

    public void LogError(string message)
    {
        logger.Error(message);
    }

    public void LogInfo(string message)
    {
        logger.Info(message);
    }

    public void LogWarn(string message)
    {
        logger.Warn(message);
    }
}
```

As you can see, our methods are just wrappers around NLog's methods. Both `ILogger` and `LogManager` are part of the `NLog` namespace. Now, we need to configure it and inject it into the Startup class in the `ConfigureServices` method.

NLog needs to have information about where to put log files on the file system, what the name of these files will be, and what is the minimum level of logging that we want.

We are going to define all these constants in a text file in the main project and name it `nlog.config`. You'll need to change the path of the `internal log` and `filename` parameters to your own paths.

```
<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      autoReload="true"
      internalLogLevel="Trace"

internalLogFile="d:\Projects\CompanyEmployees\Project\internal_logs\internallog.txt">

  <targets>
    <target name="logfile" xsi:type="File"

fileName="d:\Projects\CompanyEmployees/Project\logs\${shortdate}_logfile.txt"
            layout="${longdate} ${level:uppercase=true} ${message}"/>
  </targets>

  <rules>
    <logger name="*" minlevel="Debug" writeTo="logfile" />
  </rules>
</nlog>
```

Configuring Logger Service for Logging Messages

Setting up the configuration for a logger service is quite easy. First, we need to update the constructor of the `Startup` class:

```
public Startup(IConfiguration configuration)
{
    LogManager.LoadConfiguration(string.Concat(Directory.GetCurrentDirectory(),
"/nlog.config"));
    Configuration = configuration;
}
```

Basically, we are using NLog's `LogManager` static class with the `LoadConfiguration` method to provide a path to the configuration file.

The next thing we need to do is to add the logger service inside the .NET Core's IOC container. There are three ways to do that:

- By calling the `services.AddSingleton` method, we can create a service the first time we request it and then every subsequent request will call the same instance of the service. This means that all

components share the same service every time they need it and the same instance will be used for every method call.

- By calling the `services.AddScoped` method, we can create a service once per request. That means whenever we send an HTTP request to the application, a new instance of the service will be created.
- By calling the `services.AddTransient` method, we can create a service each time the application requests it. This means that if multiple components need the service, it will be created again for every single component request.

So, let's add a new method in the `ServiceExtensions` class:

```
public static void ConfigureLoggerService(this IServiceCollection services) =>
    services.AddScoped<ILoggerManager, LoggerManager>();
```

And after that, we need to modify the ConfigureServices method to include our newly created extension method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.ConfigureCors();
    services.ConfigureIISIntegration();
    services.ConfigureLoggerService();

    services.AddControllers();
}
```

Every time we want to use a logger service, all we need to do is to inject it into the constructor of the class that needs it. .NET Core will resolve that service and the logging features will be available.

This type of injecting a class is called Dependency Injection and it is built into .NET Core.

Let's learn a bit more about it.

DI, IoC, and Logger Service Testing

What is Dependency Injection (DI) exactly and what is IoC (Inversion of Control)?

Dependency injection is a technique we use to achieve the decoupling of objects and their dependencies. It means that rather than instantiating an object explicitly in a class every time we need it, we can instantiate it once and then send it to the class.

This is often done through a constructor. The specific approach we utilize is also known as the **`Constructor Injection`**.

In a system that is designed around DI, you may find many classes requesting their dependencies via their constructors. In this case, it is helpful to have a class that manages and provides dependencies to classes through the constructor.

These classes are referred to as containers or more specifically, Inversion of Control containers. An IoC container is essentially a factory that is responsible for providing instances of the types that are requested from it.

To test our logger service, we are going to use the default **`WeatherForecastController`**. You can find it in the main project in the Controllers folder. It comes with the ASP.NET Core Web API template.

In the Solution Explorer, we are going to open the Controllers folder and locate the **`WeatherForecastController`** class. Let's modify it:

```
[Route("[controller]")]
[ApiController]
public class WeatherForecastController : ControllerBase
{
    private ILoggerManager _logger;

    public WeatherForecastController(ILoggerManager logger)
    {
        _logger = logger;
    }
```

```
    [HttpGet]
    public IEnumerable<string> Get()
    {
        _logger.LogInfo("Here is info message from our values controller.");
        _logger.LogDebug("Here is debug message from our values controller.");
        _logger.LogWarn("Here is warn message from our values controller.");
        _logger.LogError("Here is an error message from our values controller.");

        return new string[] { "value1", "value2" };
    }
}
```

Now let's start the application and browse to `https://localhost:5001/weatherforecast`.

Tip: If you are using Windows 8 and having trouble starting this application on `https://localhost:5001...`, you have to add a parameter to the appsetings.Development.json file:

```
"Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1"
    }
  }
```

As a result, you will see an array of two strings. Now go to the folder that you have specified in the `nlog.config` file, and check out the result. You should see two folders: the `internal_logs` folder and the `logs` folder. Inside the `logs` folder, you should find a file with the following logs:

```
2019-09-27 11:52:01.7316 INFO Info message from our controller.
2019-09-27 11:52:01.7796 DEBUG Debug message from our controller.
2019-09-27 11:52:01.7796 WARN Warn message from our controller.
2019-09-27 11:52:01.7963 ERROR Error message from our controller.
```

That's all we need to do to configure our logger for now. We'll add some messages to our code along with the new features.

DATABASE MODEL AND REPOSITORY PATTERN

In this chapter, we are going to create a database model and transfer it to the MSSQL database by using the code first approach. So, we are going to learn how to create entities (model classes), how to work with the DbContext class, and how to use migrations to transfer our created database model to the real database. Of course, it is not enough to just create a database model and transfer it to the database. We need to use it as well, and for that, we will create a Repository pattern as a data access layer.

With the Repository pattern, we create an abstraction layer between the data access and the business logic layer of an application. By using it, we are promoting a more loosely coupled approach to access our data in the database.

Also, our code becomes cleaner, easier to maintain, and reusable. Data access logic is stored in a separate class, or sets of classes called a repository, with the responsibility of persisting the application's business model.

So, let's start with the model classes first.

Creating Models

Using the example from the second chapter of this book, we are going to extract a new Class Library (.NET Core) project named `Entities`.

Don't forget to add the reference from the main project to the Entities project.

Inside it, we are going to create a folder named `Models`, which will contain all the model classes (entities). Entities represent classes that Entity Framework Core uses to map our database model with the tables

from the database. The properties from entity classes will be mapped to the database columns.

So, in the Models folder we are going to create two classes and modify them:

```
public class Company
{
    [Column("CompanyId")]
    public Guid Id { get; set; }

    [Required(ErrorMessage = "Company name is a required field.")]
    [MaxLength(60, ErrorMessage = "Maximum length for the Name is 60 characters.")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Company address is a required field.")]
    [MaxLength(60, ErrorMessage = "Maximum length for rhe Address is 60 characte")]
    public string Address { get; set; }

    public string Country { get; set; }

    public ICollection<Employee> Employees { get; set; }
}

public class Employee
{
    [Column("EmployeeId")]
    public Guid Id { get; set; }

    [Required(ErrorMessage = "Employee name is a required field.")]
    [MaxLength(30, ErrorMessage = "Maximum length for the Name is 30 characters.")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Age is a required field.")]
    public int Age { get; set; }

    [Required(ErrorMessage = "Position is a required field.")]
    [MaxLength(20, ErrorMessage = "Maximum length for the Position is 20
characters.")]
    public string Position { get; set; }

    [ForeignKey(nameof(Company))]
    public Guid CompanyId { get; set; }
    public Company Company { get; set; }
}
```

We have created two classes: Company and Employee. Those classes contain the properties which Entity Framework Core is going to map to the columns in our tables in the database. But not all the properties will be mapped as columns. The last property of the Company class (Employees) and the last property of the Employee class (Company) are

navigational properties; these properties serve the purpose of defining the relationship between our models.

We can see several attributes in our entities. The `[Column]` attribute will specify that the `Id` property is going to be mapped with a different name in the database. The `[Required]` and `[MaxLength]` properties are here for validation purposes. The first one declares the property as mandatory and the second one defines its maximum length.

Once we transfer our database model to the real database, we are going to see how all these validation attributes and navigational properties affect the column definitions.

Context Class and the Database Connection

Now, let's create the context class, which will be a middleware component for communication with the database. It must inherit from the Entity Framework Core's `DbContext` class and it consists of `DbSet` properties, which EF Core is going to use for the communication with the database. Because we are working with the DBContext class, we need to install the `Microsoft.EntityFrameworkCore` package in the Entities project.

So, let's navigate to the root of the `Entities` project and create the `RepositoryContext` class:

```
public class RepositoryContext : DbContext
{
    public RepositoryContext(DbContextOptions options)
        : base(options)
    {
    }

    public DbSet<Company> Companies { get; set; }
    public DbSet<Employee> Employees { get; set; }
}
```

After the class modification, let's open the `appsettings.json` file and add the connection string named `sqlconnection`:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "ConnectionStrings": {
    "sqlConnection": "server=.; database=CompanyEmployee; Integrated Security=true"
  },
  "AllowedHosts": "*"
}
```

It is quite important to have the JSON object with the **ConnectionStrings** name in our **app.config** file, and soon you will see why.

We have one more step to finish the database model configuration. We need to register the **RepositoryContext** class in the application's dependency injection container as we did with the **LoggerManager** class in the previous chapter.

So, let's open the **ServiceExtensions** class and add the additional method:

```
public static void ConfigureSqlContext(this IServiceCollection services,
 IConfiguration configuration) =>
    services.AddDbContext<RepositoryContext>(opts =>
        opts.UseSqlServer(configuration.GetConnectionString("sqlConnection")));
```

With the help of the **IConfiguration configuration** parameter, we can use the **GetConnectionString** method to access the connection string from the **appsettings.json** file. Moreover, to be able to use the **UseSqlServer** method, we need to install the **Microsoft.EntityFrameworkCore.SqlServer** package. If we navigate to the **GetConnectionString** method definition, we will see that it is an extension method that uses the **ConnectionStrings** name from the **appsettings.json** file to fetch the connection string by the provided key:

```
// Summary:
//     Shorthand for GetSection("ConnectionStrings")[name].
//
// Parameters:
//   configuration:
//     The configuration.
//
//   name:
//     The connection string key.
public static string GetConnectionString(this IConfiguration configuration, string name);
```

Afterward, in the **Startup** class in the **ConfigureServices** method, we are going to add the context service to the IOC right above the **services.AddControllers()** line:

```
services.ConfigureSqlContext(Configuration);
```

Migration and Initial Data Seed

Migration is a standard process of creating and updating the database from our application. Since we are finished with the database model creation, we can transfer that model to the real database. But we need to modify our **ConfigureSqlContext** method first:

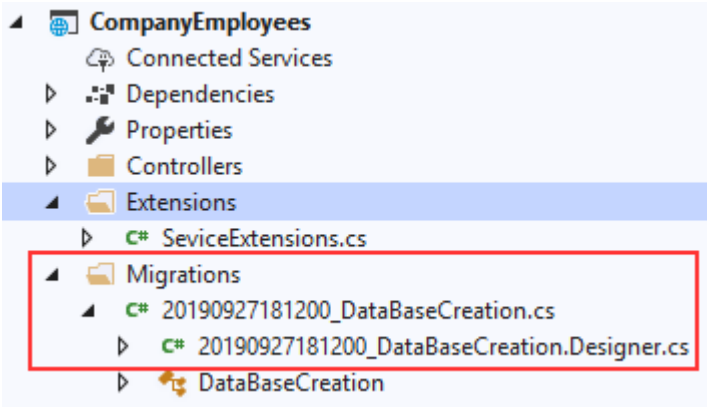
```
public static void ConfigureSqlContext(this IServiceCollection services,
IConfiguration configuration) =>
    services.AddDbContext<RepositoryContext>(opts =>
        opts.UseSqlServer(configuration.GetConnectionString("sqlConnection"), b =>
b.MigrationsAssembly("CompanyEmployees")));
```

We have to make this change because migration assembly is not in our main project, but in the **Entities** project. So, we just change the project for the migration assembly.

Before we execute our migration commands, we have to install an additional ef core library: **Microsoft.EntityFrameworkCore.Tools**

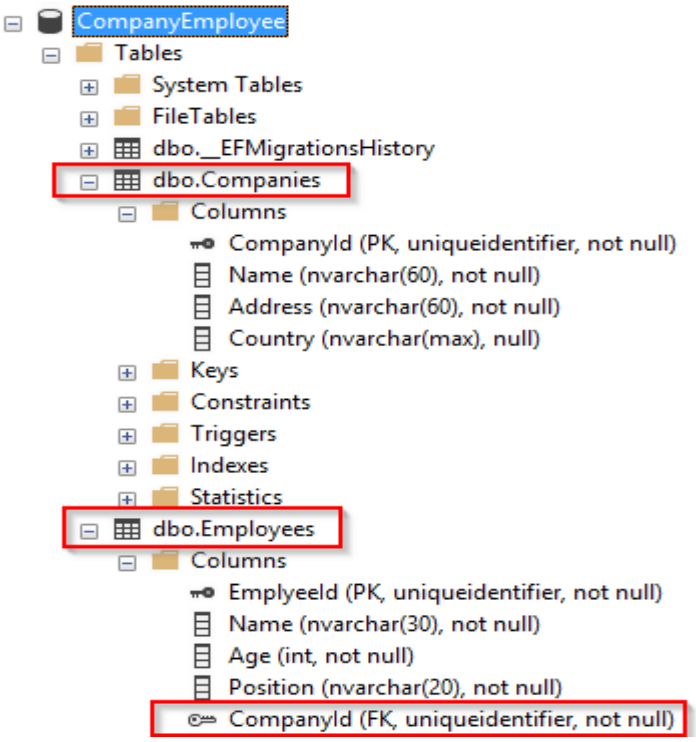
Now, let's open the Package Manager Console window and create our first migration: `PM> Add-Migration DatabaseCreation`

With this command, we are creating migration files and we can find them in the **Migrations** folder in our main project:



With those files in place, we can apply migration: `PM> Update-Database`

Excellent. We can inspect our database now:



Once we have the database and tables created, we should populate them with some initial data. To do that, we are going to create another folder called **`Configuration`** in the **`Entities`** project and add the **`CompanyConfiguration`** class:

```
public class CompanyConfiguration : IEntityTypeConfiguration<Company>
{
    public void Configure(EntityTypeBuilder<Company> builder)
```

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

Let's do the same thing for the **EmployeeConfiguration** class:

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

To invoke this configuration, we have to change the `RepositoryContext` class:

```
public class RepositoryContext: DbContext
{
    public RepositoryContext(DbContextOptions options)
    : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.ApplyConfiguration(new CompanyConfiguration());
        modelBuilder.ApplyConfiguration(new EmployeeConfiguration());
    }

    public DbSet<Company> Companies { get; set; }
    public DbSet<Employee> Employees { get; set; }
}
```

Now, we can create and apply another migration to seed these data to the database:

```
PM> Add-Migration InitialData

PM> Update-Database
```

This will transfer all the data from our configuration files to the respective tables.

Repository Pattern Logic

After establishing a connection to the database and creating one, it's time to create a generic repository that will provide us with the CRUD methods. As a result, all the methods can be called upon any repository class in our project.

Furthermore, creating the generic repository and repository classes that use that generic repository is not going to be the final step. We will go a step further and create a wrapper class around repository classes and inject it as a service in a dependency injection container.

Consequently, we will be able to instantiate this class once and then call any repository class we need inside any of our controllers.

The advantages of this approach will become clearer once we use it in the project.

That said, let's start by creating an interface for the repository inside the **Contracts** project:

```
public interface IRepositoryBase<T>
{
    IQueryable<T> FindAll(bool trackChanges);
    IQueryable<T> FindByCondition(Expression<Func<T, bool>> expression,
    bool trackChanges);
    void Create(T entity);
    void Update(T entity);
    void Delete(T entity);
}
```

Right after the interface creation, we are going to create a new Class Library (.NET Core) project with the name **Repository** and add the reference to the **Contracts** and **Entities** class libraries. Inside the **Repository** project, we are going to create an abstract class **RepositoryBase** — which is going to implement the **IRepositoryBase** interface.

We need to reference this project from the main project as well.

Additional info: *We are going to use EF Core functionalities in the Repository project. Therefore, we need to install it inside the Repository project.*

Let's add the following code to the **RepositoryBase** class:

```
public class RepositoryBase<T> : IRepositoryBase<T> where T : class
{
    protected RepositoryContext RepositoryContext;

    public RepositoryBase(RepositoryContext repositoryContext)
    {
       RepositoryContext = repositoryContext;
    }

    public IQueryable<T> FindAll(bool trackChanges) =>
        !trackChanges ?
          RepositoryContext.Set<T>()
            .AsNoTracking() :
          RepositoryContext.Set<T>();
```

```
    public IQueryable<T> FindByCondition(Expression<Func<T, bool>> expression,
    bool trackChanges) =>
        !trackChanges ?
          RepositoryContext.Set<T>()
            .Where(expression)
            .AsNoTracking() :
          RepositoryContext.Set<T>()
            .Where(expression);

    public void Create(T entity) => RepositoryContext.Set<T>().Add(entity);

    public void Update(T entity) => RepositoryContext.Set<T>().Update(entity);

    public void Delete(T entity) => RepositoryContext.Set<T>().Remove(entity);
}
```

This abstract class as well as the `IRepositoryBase` interface works with the generic type `T`. This type `T` gives even more reusability to the `RepositoryBase` class. That means we don't have to specify the exact model (class) right now for the `RepositoryBase` to work with. We can do that later on.

Moreover, we can see the `trackChanges` parameter. We are going to use it to improve our read-only query performance. When it's set to false, we attach the `AsNoTracking` method to our query to inform EF Core that it doesn't need to track changes for the required entities. This greatly improves the speed of a query.

Repository User Interfaces and Classes

Now that we have the `RepositoryBase` class, let's create the user classes that will inherit this abstract class.

By inheriting from the `RepositoryBase` class, they will have access to all the methods from it. Furthermore, every user class will have its own interface for additional model-specific methods.

This way, we are separating the logic that is common for all our repository user classes and also specific for every user class itself.

Let's create the interfaces in the **Contracts** project for the **Company** and **Employee** classes.

```
namespace Contracts
{
    public interface ICompanyRepository
    {
    }
}

namespace Contracts
{
    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 : RepositoryBase<Company>, ICompanyRepository
{
    public CompanyRepository(RepositoryContext repositoryContext)
        : base(repositoryContext)
    {
    }
}
```

And then, the **EmployeeRepository** class:

```
public class EmployeeRepository : RepositoryBase<Employee>, IEmployeeRepository
{
    public EmployeeRepository(RepositoryContext repositoryContext)
        : base(repositoryContext)
    {
    }
}
```

After these steps, we are finished creating the repository and repository user classes. But there are still more things to do.

Creating a Repository Manager

It is quite common for the API to return a response that consists of data from multiple resources; for example, all the companies and just some employees older than 30. In such a case, we would have to instantiate both of our repository classes and fetch data from their own resources.

Maybe it's not a problem when we have only two classes, but what if we need the combined logic of five or even more different classes? It would just be too complicated to pull that off.

With that in mind, we are going to create a repository manager class, which will create instances of repository user classes for us and then register it inside the dependency injection container. After that, we can inject it inside our controllers (or inside a business layer class, if we have a bigger app) with constructor injection (supported by ASP.NET Core). With the repository manager class in place, we may call any repository user class we need.

But we are also missing one important part. We have the `Create`, `Update`, and `Delete` methods in the `RepositoryBase` class, but they won't make any change in the database until we call the `SaveChanges` method. Our repository manager class will handle that as well.

That said, let's get to it and create a new interface in the `Contract` project:

```
public interface IRepositoryManager
{
    ICompanyRepository Company { get; }
    IEmployeeRepository Employee { get; }
    void Save();
}
```

And add a new class to the `Repository` project:

```
public class RepositoryManager : IRepositoryManager
{
    private RepositoryContext _repositoryContext;
```

```
    private ICompanyRepository _companyRepository;
    private IEmployeeRepository _employeeRepository;

    public RepositoryManager(RepositoryContext repositoryContext)
    {
        _repositoryContext = repositoryContext;
    }

    public ICompanyRepository Company
    {
        get
        {
            if(_companyRepository == null)
                _companyRepository = new CompanyRepository(_repositoryContext);

            return _companyRepository;
        }
    }

    public IEmployeeRepository Employee
    {
        get
        {
            if(_employeeRepository == null)
                _employeeRepository = new EmployeeRepository(_repositoryContext);

            return _employeeRepository;
        }
    }

    public void Save() => _repositoryContext.SaveChanges();
}
```

As you can see, we are creating properties that will expose the concrete repositories and also we have the `Save()` method to be used after all the modifications are finished on a certain object. This is a good practice because now we can, for example, add two companies, modify two employees, and delete one company — all in one action — and then just call the `Save` method once. All the changes will be applied or if something fails, all the changes will be reverted:

```
_repository.Company.Create(company);
_repository.Company.Create(anotherCompany);
_repository.Employee.Update(employee);
_repository.Employee.Update(anotherEmployee);
_repository.Company.Delete(oldCompany);

_repository.Save();
```

After these changes, we need to register our manager class and add a reference from the Repository to our 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 in the **Startup** class inside the **ConfigureServices** method, above the **services.AddController()** line, we have to add this code:

```
services.ConfigureRepositoryManager();
```

Excellent.

As soon as we add some methods to the specific repository classes, we are going to be able to test this logic, but we can just take a peek at how we can inject and use this repository manager.

All we have to do is to inject the **RepositoryManager** service inside the controller and we are going to see the Company and Employee properties that will provide us access to the specific repository methods:

```
[Route("[controller]")]
[ApiController]
public class WeatherForecastController : ControllerBase
{
    private readonly IRepositoryManager _repository;

    public WeatherForecastController(IRepositoryManager repository)
    {
        _repository = repository;
    }

    [HttpGet]
    public ActionResult<IEnumerable<string>> Get()
    {
        _repository.Company.AnyMethodFromCompanyRepository();
        _repository.Employee.AnyMethodFromEmployeeRepository();

        return new string[] { "value1", "value2" };
    }
}
```

We did an excellent job here. The repository layer is prepared and ready to be used to fetch data from the database.

As you can see, we have injected our repository inside the controller; this is a good practice for an application of this size. But for larger-scale applications, we would create an additional business layer between our controllers and repository logic and our RepositoryManager service would be injected inside that Business layer — thus freeing the controller from repository logic.

Now, we can continue towards handling Get requests in our application.

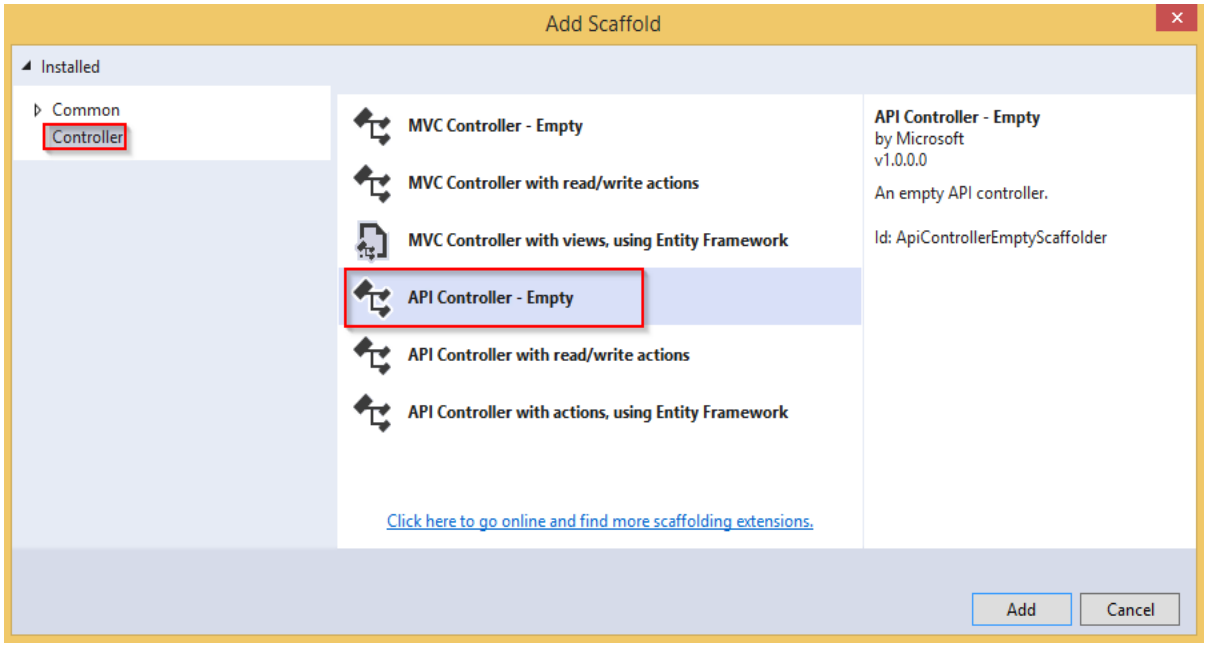
HANDLING GET REQUESTS

We're all set to add some business logic to our application. But before that, let's talk a bit about controller classes and routing because they play an important part while working with HTTP requests.

Controllers and Routing in WEB API

Controllers should only be responsible for handling requests, model validation, and returning responses to the frontend or some HTTP client. Keeping business logic away from controllers is a good way to keep them lightweight, and our code more readable and maintainable.

To create the controller, right click on the Controllers folder inside the main project and then Add=>Controller. Then from the menu, choose API Controller Class and name it `CompaniesController.cs`.



Our controller should be generated with the default code inside:

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

```
    {

    }
}
```

Every web API controller class inherits from the **ControllerBase** abstract class, which provides all necessary behavior for the derived class.

Also, above the controller class we can see this part of the code:

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

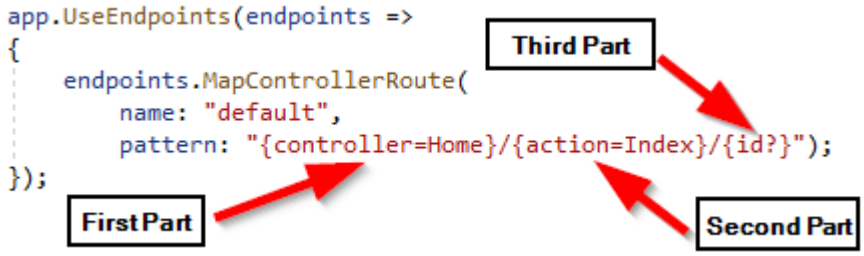
This attribute represents routing and we are going to talk more about routing inside Web APIs.

Web API routing routes incoming HTTP requests to the particular action method inside the Web API controller. As soon as we send our HTTP request, the MVC framework parses that request and tries to match it to an action in the controller.

There are two ways to implement routing in the project:

- Convention based routing and
- Attribute routing

Convention based routing is called such because it establishes a convention for the URL paths. **The first part** creates the mapping for the controller name, the **second part** creates the mapping for the action method, and **the third part** is used for the optional parameter. We can configure this type of routing in the **Startup** class in the **Configure** method:



Attribute routing uses the attributes to map the routes directly to the action methods inside the controller. Usually, we place the base route above the controller class, as you can see in our Web API controller class. Similarly, for the specific action methods, we create their routes right above them.

While working with the Web API project, the ASP.NET Core team suggests that we shouldn't use Convention-based Routing, but Attribute routing instead.

Different actions can be executed on the resource with the same URI, but with different HTTP Methods. In the same manner for different actions, we can use the same HTTP Method, but different URIs. Let's explain this quickly.

For Get request, Post, or Delete, we use the same URI `/api/companies` but we use different HTTP Methods like GET, POST or DELETE. But if we send a request for all companies or just one company, we are going to use the same GET method but different URIs `(/api/companies` for all companies and `/api/companies/{companyId}` for a single company).

We are going to understand this even more once we start implementing different actions in our controller.

Naming Our Resources

The resource name in the URI should always be a noun and not an action. That means if we want to create a route to get all companies, we should

create this route: `api/companies` and not this one: `/api/getCompanies`.

The noun used in URI represents the resource and helps the consumer to understand what type of resource we are working with. So, we shouldn't choose the noun *products* or *orders* when we work with the companies resource; the noun should always be companies. Therefore, by following this convention if our resource is employees (and we are going to work with this type of resource), the noun should be employees.

Another important part we need to pay attention to is the hierarchy between our resources. In our example, we have a Company as a principal entity and an Employee as a dependent entity. When we create a route for a dependent entity, we should follow a slightly different convention: `/api/principalResource/{principalId}/dependentResource`.

Because our employees can't exist without a company, the route for the employee's resource should be: `/api/companies/{companyId}/employees`.

With all of this in mind, we can start with the Get requests.

Getting All Companies From the Database

So let's start.

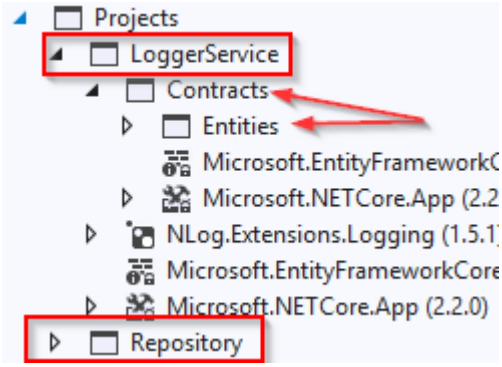
The first thing we are going to do is to change the base route from `[Route("api/[controller]")]` to `[Route("api/companies")]`. Even though the first route will work just fine, with the second example we are more specific to show that this routing should point to the `CompaniesController` class.

Now it is time to create the first action method to return all the companies from the database. Let's create a definition for the **GetAllCompanies** method in the **ICompanyRepository** interface:

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

For this to work, we need to add a reference from the **Entities** project to the **Contracts** project. But we are going to stop here for a moment to draw your attention to one important thing.

In our main project, we are referencing the LoggerService, Repository, and Entities projects. Since both the LoggerService and Repository projects have a reference for the Contracts project (which has a reference to the Entities project; we just added it) this means that the main project has a reference for the Entities project as well through the LoggerService or Repository projects. That said, we can remove the Entities reference from the main project:



Now, we can continue with the interface implementation in the **CompanyRepository** class:

```
public class CompanyRepository : RepositoryBase<Company>, ICompanyRepository
{
    public CompanyRepository(RepositoryContext repositoryContext)
        : base(repositoryContext)
    {
    }
```

```
    public IEnumerable<Company> GetAllCompanies(bool trackChanges) =>
        FindAll(trackChanges)
        .OrderBy(c => c.Name)
        .ToList();
}
```

Finally, we have to return companies by using the **GetAllCompanies** method inside the Web API controller.

The purpose of the action methods inside the Web API controllers is not only to return results. It is the main purpose, but not the only one. We need to pay attention to the status codes of our Web API responses as well. Additionally, we are going to decorate our actions with the HTTP attributes which will mark the type of the HTTP request to that action.

So, let's modify the **CompaniesController**:

```
[Route("api/companies")]
[ApiController]
public class CompaniesController : ControllerBase
{
    private readonly IRepositoryManager _repository;
    private readonly ILoggerManager _logger;

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

    [HttpGet]
    public IActionResult GetCompanies()
    {
        try
        {
            var companies = _repository.Company.GetAllCompanies(trackChanges: false);

            return Ok(companies);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Something went wrong in the {nameof(GetCompanies)}
action {ex}");
            return StatusCode(500, "Internal server error");
        }
    }
}
```

Let's explain this code a bit.

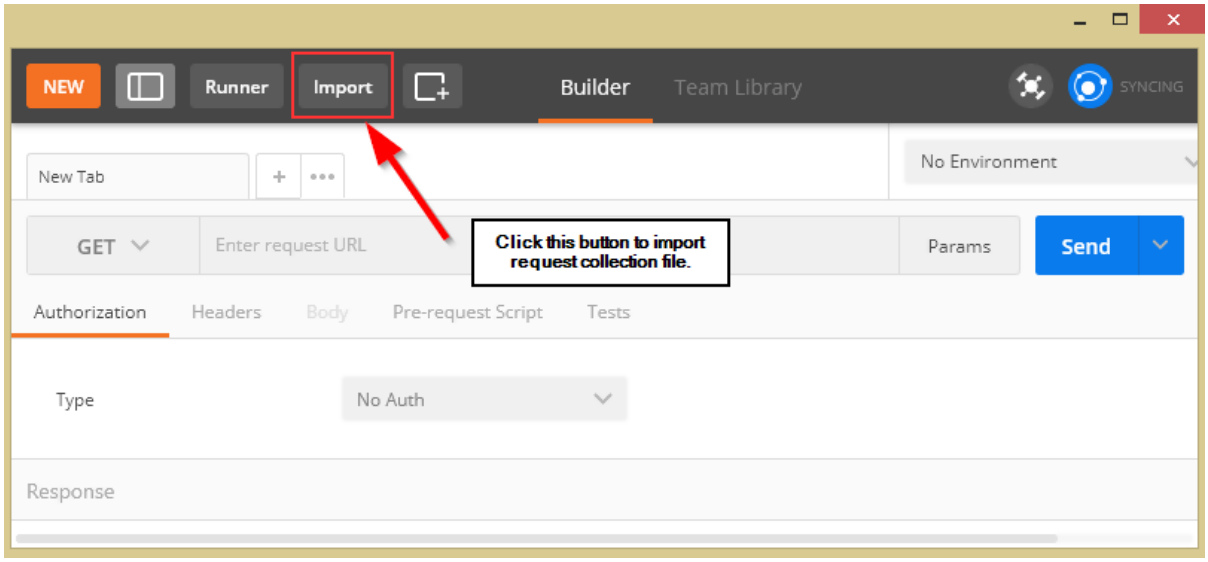
First of all, we inject the logger and repository services inside the constructor. Then by decorating the `GetCompanies` action with the `[HttpGet]` attribute, we are mapping this action to the GET request. Then, we use both injected services to log the messages and to get the data from the repository class.

The `IActionResult` interface supports using a variety of methods, which return not only the result but also the status codes. In this situation, the `OK` method returns all the companies and also the status code 200 — which stands for `OK`. If an exception occurs, we are going to return the internal server error with the status code 500.

Because there is no route attribute right above the action, the route for the `GetCompanies action` will be `api/companies` which is the route placed on top of our controller.

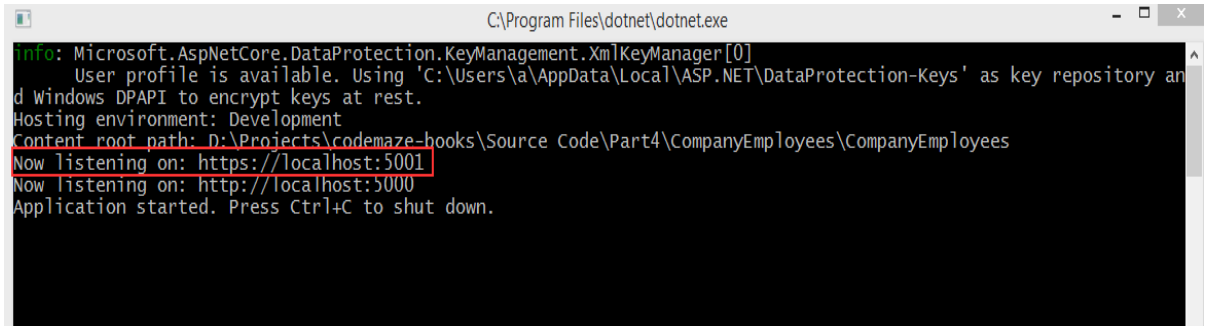
Testing the Result with Postman

To check the result, we are going to use a great tool named Postman, which helps a lot with sending requests and displaying responses. If you download our exercise files, you will find the file `CompanyEmployeesRequests,` which contains a request collection divided for each chapter of this book. You can import them in Postman to save yourself the time of manually typing them:

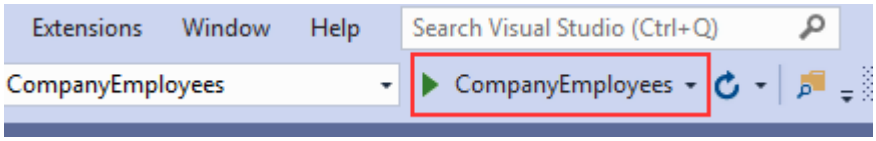


Please note that some GUID values will be different for your project, so you have to change them according to your values.

So let's start the application by pressing the F5 button and check that it is now listening on the https:localhost:5001 address:

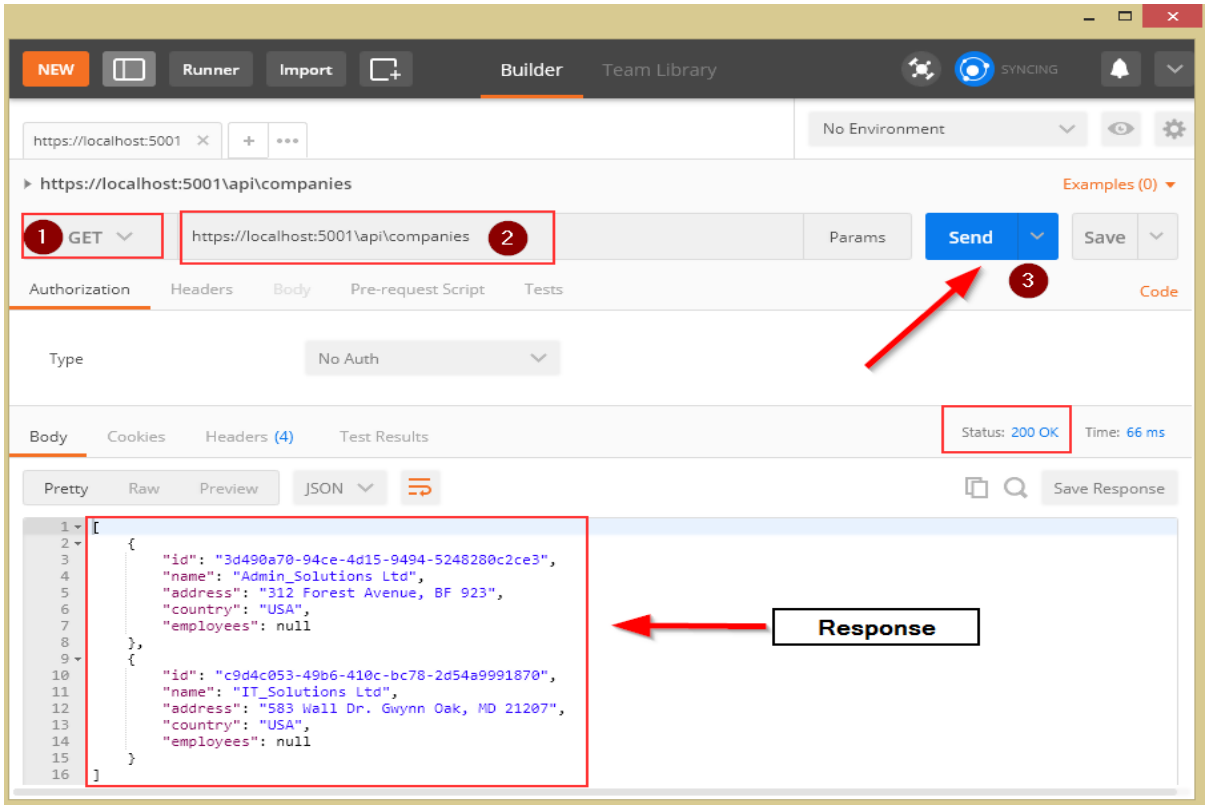


If this is not the case, you probably ran it in the IIS mode; so turn the application off and start it again, but in the CompanyEmployees mode:



Now, we can use Postman to test the result:

`http://www.companyemployees.codemaze/api/companies`



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.

DTO Classes vs. Entity Model Classes

Data transfer object (DTO) is an object that we use to transport data between the client and server applications.

So, as we said in a previous section of this book, it is not a good practice to return entities in the Web API response; we should instead use data transfer objects. But why is that?

Well, EF Core uses model classes to map them to the tables in the database and that is the main purpose of a model class. But as we saw, our models have navigational properties and sometimes we don't want to

map them in an API response. So, we can use DTO to remove any property or concatenate properties into a single property.

Moreover, there are situations where we want to map all the properties from a model class to the result — but still, we want to use DTO instead. The reason is if we change the database, we also have to change the properties in a model — but that doesn't mean our clients want the result changed. So, by using DTO, the result will stay as it was before the model changes.

As we can see, keeping these objects separate (the DTO and model classes) leads to a more robust and maintainable code in our application.

Now, when we know why should we separate DTO from a model class in our code, let's create the folder **DataTransferObjects** in the **Entities** project with the **CompanyDto** class inside:

```
public class CompanyDto
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string FullAddress { get; set; }
}
```

We have removed the **Employees** property and we are going to use the **FullAddress** property to concatenate the **Address** and **Country** properties from the **Company** class. Furthermore, we are not using validation attributes in this class, because we are going to use this class only to return a response to the client. Therefore, validation attributes are not required.

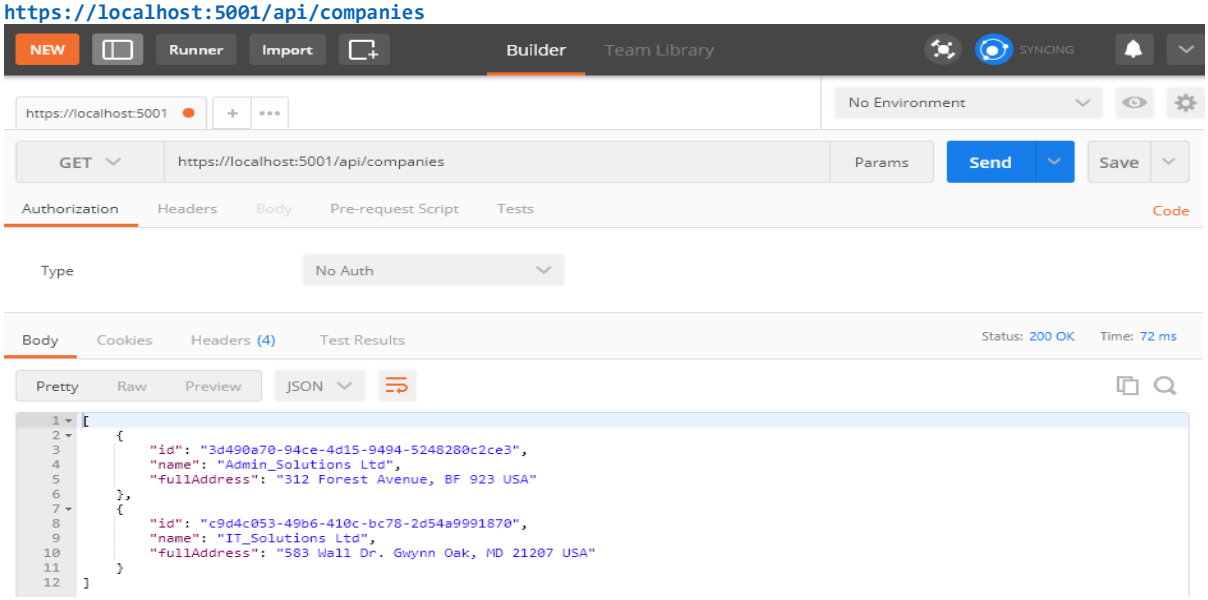
So, let's open and modify the **GetCompanies** action:

```
[HttpGet]
public IActionResult GetCompanies()
{
    try
    {
        var companies = _repository.Company.GetAllCompanies(trackChanges: false);
```

```
        var companiesDto = companies.Select(c => new CompanyDto
        {
            Id = c.Id,
            Name = c.Name,
            FullAddress = string.Join(' ', c.Address, c.Country)
        }).ToList();

        return Ok(companiesDto);
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong in the {nameof(GetCompanies)} action
{ex}");
        return StatusCode(500, "Internal server error");
    }
}
```

Let's start our application and test it with the same request from Postman:

https://localhost:5001/api/companies



This time we get our CompanyDto result, which is a more preferred way. But this can be improved as well. If we take a look at our mapping code in the `GetCompanies` action, we can see that we manually map all the properties. Sure, it is okay for few fields — but what if we have a lot more? There is a better and cleaner way to map our classes and that is by using the Automapper.

Using AutoMapper in ASP.NET Core

AutoMapper is a library that helps us with mapping objects in our applications. By using this library, we are going to remove the code for manual mapping — thus making the action readable and maintainable.

So, to install AutoMapper, let's open a Package Manager Console window and run the following command:

```
PM> Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection
```

After installation, we are going to register this library in the `ConfigureServices` method:

```
services.AddAutoMapper(typeof(Startup));
```

As soon as our library is registered, we are going to create a profile class where we specify the source and destination objects for mapping:

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        CreateMap<Company, CompanyDto>()
                .ForMember(c => c.FullAddress,
                    opt => opt.MapFrom(x => string.Join(' ', x.Address, x.Country)));
    }
}
```

The `MappingProfile` class must inherit from the AutoMapper's `Profile` class. In the constructor, we are using the `CreateMap` method where we specify the source object and the destination object to map to. Because we have the `FullAddress` property in our DTO class, which contains both the `Address` and the `Country` from the model class, we have to specify additional mapping rules with the `ForMember` method.

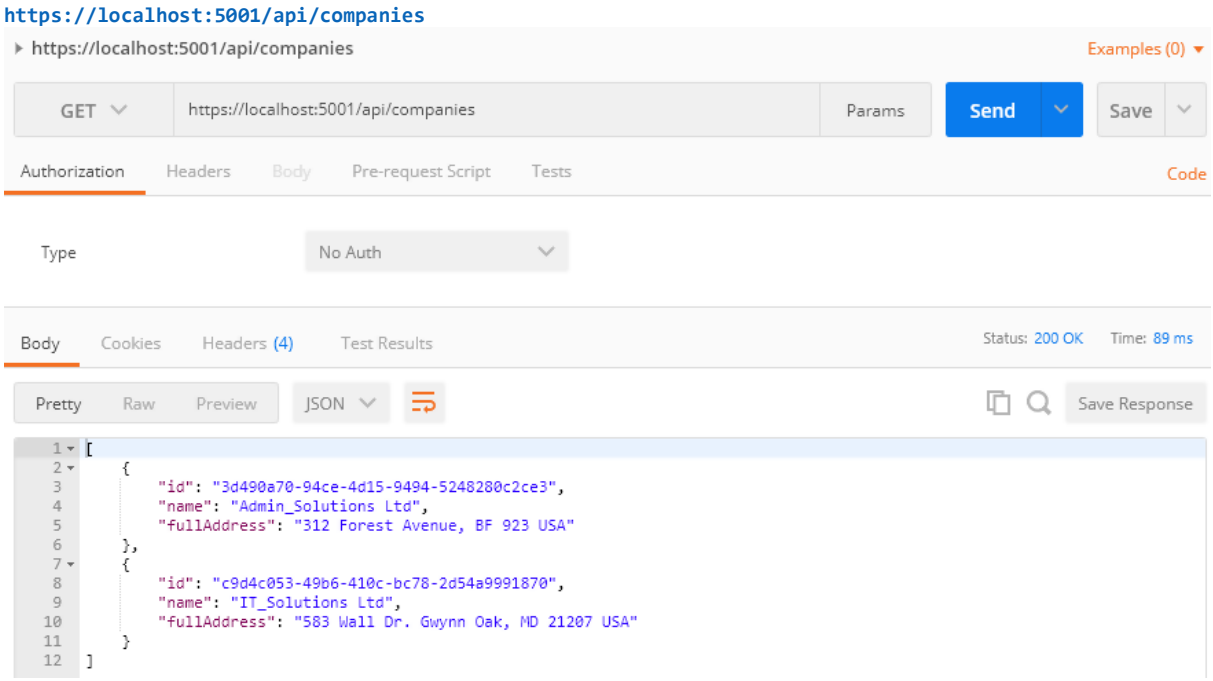
Now, we can use AutoMapper in our controller like any other service registered in IoC:

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

```
{
    private readonly IRepositoryManager _repository;
    private readonly ILoggerManager _logger;
    private readonly IMapper _mapper;

    public CompaniesController(IRepositoryManager repository, ILoggerManager logger,
IMapper mapper)
    {
        _repository = repository;
        _logger = logger;
        _mapper = mapper;
    }

    [HttpGet]
    public IActionResult GetCompanies()
    {
        try
        {
            var companies = _repository.Company.GetAllCompanies(trackChanges: false);

            var companiesDto = _mapper.Map<IEnumerable<CompanyDto>>(companies);

            return Ok(companiesDto);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Something went wrong in the {nameof(GetCompanies)}
action               {ex}");

            return StatusCode(500, "Internal server error");
        }
    }
```

Excellent.

Let’s use Postman again to send the request to test our app:



We can see that everything is working as it supposed to, but now with much better code.

GLOBAL ERROR HANDLING

Exception handling helps us deal with the unexpected behavior of our system. To handle exceptions, we use the `try-catch` block in our code as well as the `finally` keyword to clean up our resources afterwards.

Even though there is nothing wrong with the try-catch blocks in our Actions in the Web API project, we can extract all the exception handling logic into a single centralized place. By doing that, we make our actions cleaner, more readable, and the error handling process more maintainable.

In this chapter, we are going to refactor our code to use the built-in middleware and our custom middleware for global error handling to demonstrate the benefits of this approach.

Handling Errors Globally with the Built-In Middleware

The `UseExceptionHandler` middleware is a built-in middleware that we can use to handle exceptions. So, let's dive into the code to see this middleware in action.

We are going to create a new `ErrorModel` folder in the `Entities` project, and add the new class `ErrorDetails` in that folder:

```
public class ErrorDetails
{
    public int StatusCode { get; set; }
    public string Message { get; set; }

    public override string ToString() => JsonConvert.SerializeObject(this);
}
```

We are going to use this class for the details of our error message.

To continue, in the `Extensions` folder in the main project, we are going to add a new static class: `ExceptionMiddlewareExtensions.cs`.

Now, we need to modify it:

```
public static class ExceptionMiddlewareExtensions
{
    public static void ConfigureExceptionHandler(this IApplicationBuilder app,
ILoggerManager logger)
    {
        app.UseExceptionHandler(appError =>
        {
            appError.Run(async context =>
            {
                context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;
                context.Response.ContentType = "application/json";

                var contextFeature = context.Features.Get<IExceptionHandlerFeature>();
                if (contextFeature != null)
                {
                    logger.LogError($"Something went wrong: {contextFeature.Error}");

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

In the code above, we've created an extension method in which we've registered the `UseExceptionHandler` middleware. Then, we've populated the status code and the content type of our response, logged the error message, and finally returned the response with the custom created object.

Startup Class Modification

To be able to use this extension method, let's modify the `Configure` method inside the `Startup` class:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
ILoggerManager logger)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }

    app.ConfigureExceptionHandler(logger);
```

```
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseCors("CorsPolicy");

    app.UseForwardedHeaders(new ForwardedHeadersOptions
    {
        ForwardedHeaders = ForwardedHeaders.All
    });

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Finally, let's remove the `try-catch` block from our code:

```
[HttpGet]
public IActionResult GetCompanies()
{
    var companies = _repository.Company.GetAllCompanies(trackChanges: false);

    var companiesDto = _mapper.Map<IEnumerable<CompanyDto>>(companies);

    return Ok(companiesDto);
}
```

And there we go. Our action method is much cleaner now. More importantly, we can reuse this functionality to write more readable actions in the future.

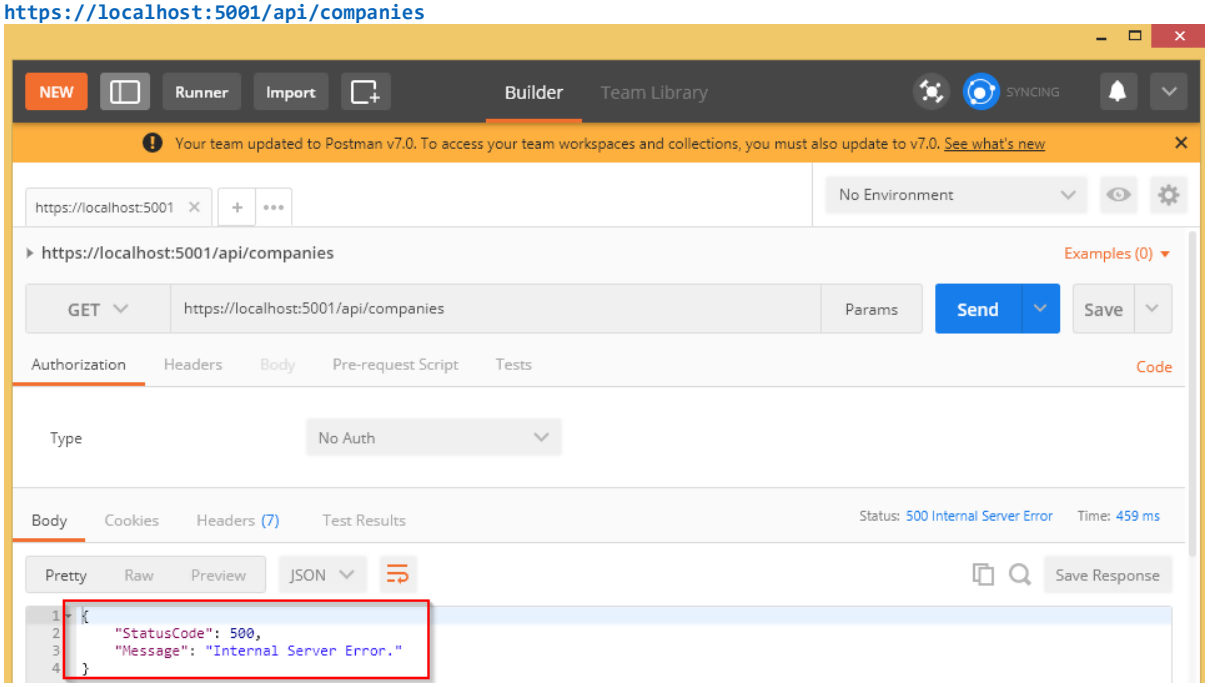
Testing the Result

To inspect this functionality, let's add the following line to the `GetCompanies` action, just to simulate an error:

```
throw new Exception("Exception");
```

And send a request from Postman:

https://localhost:5001/api/companies



We can check our log messages to make sure that logging is working as well.

GETTING ADDITIONAL RESOURCES

As of now, we can continue with GET requests by adding 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.

Getting a Single Resource From the Database

Let's start by modifying the `ICompanyRepository` interface:

```
public interface ICompanyRepository
{
    IEnumerable<Company> GetAllCompanies(bool trackChanges);
    Company GetCompany(Guid companyId, bool trackChanges);
}
```

Then, we are going to implement this interface in the `CompanyRepository.cs` file:

```
public Company GetCompany(Guid companyId, bool trackChanges) =>
    FindByCondition(c => c.Id.Equals(companyId), trackChanges)
    .SingleOrDefault();
```

Finally, let's change the `CompanyController` class:

```
[HttpGet("{id}")]
public IActionResult GetCompany(Guid id)
{
    var company = _repository.Company.GetCompany(id, trackChanges: false);
    if(company == null)
    {
        _logger.LogInfo($"Company with id: {id} doesn't exist in the database.");
        return NotFound();
    }
    else
    {
        var companyDto = _mapper.Map<CompanyDto>(company);
        return Ok(companyDto);
    }
}
```

The route for this action is `/api/companies/id` and that's because the `/api/companies` part applies from the root route (on top of the

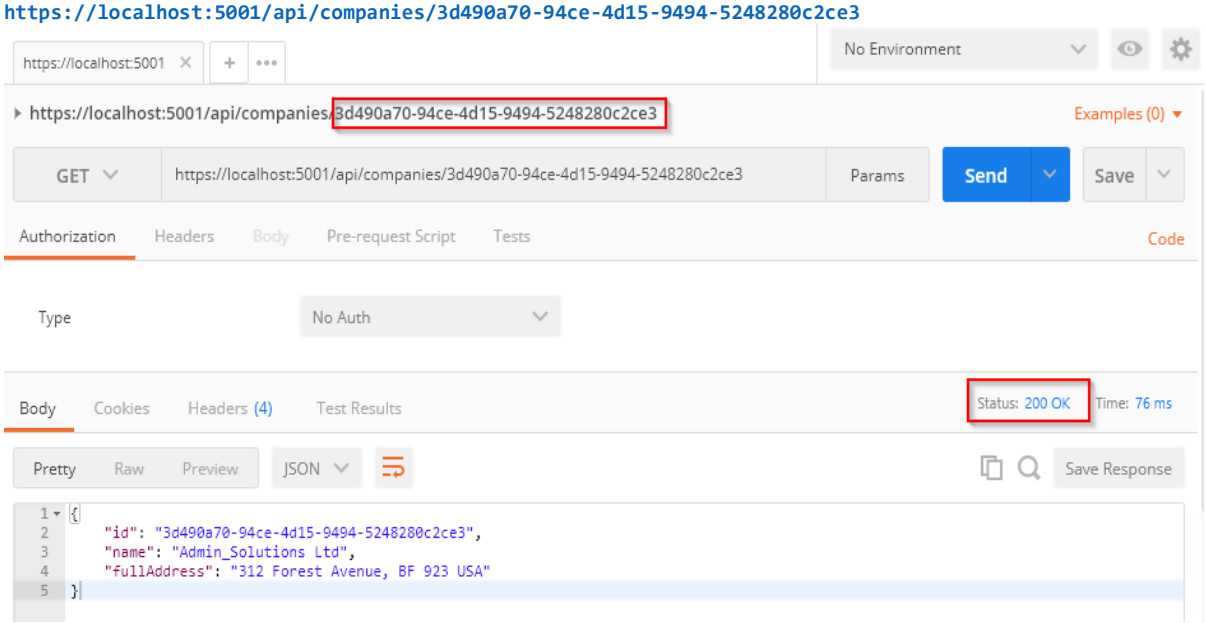
controller) and the `id` part is applied from the action attribute `[HttpGet("{id}")]`.

So, our action returns `IActionResult`, like the previous one, and we fetch a single company from the database. If it doesn't exist, we use the `NotFound` method to return a 404 status code. From this example, we can see that ASP.NET Core provides us with a variety of semantical methods that state what we can use them for, just by reading their names. The `Ok` method is for the good result (status code 200) and the `NotFound` method is for the NotFound result (status code 404).

If a company exists in the database, we just map it to the `CompanyDto` type and return it to the client.

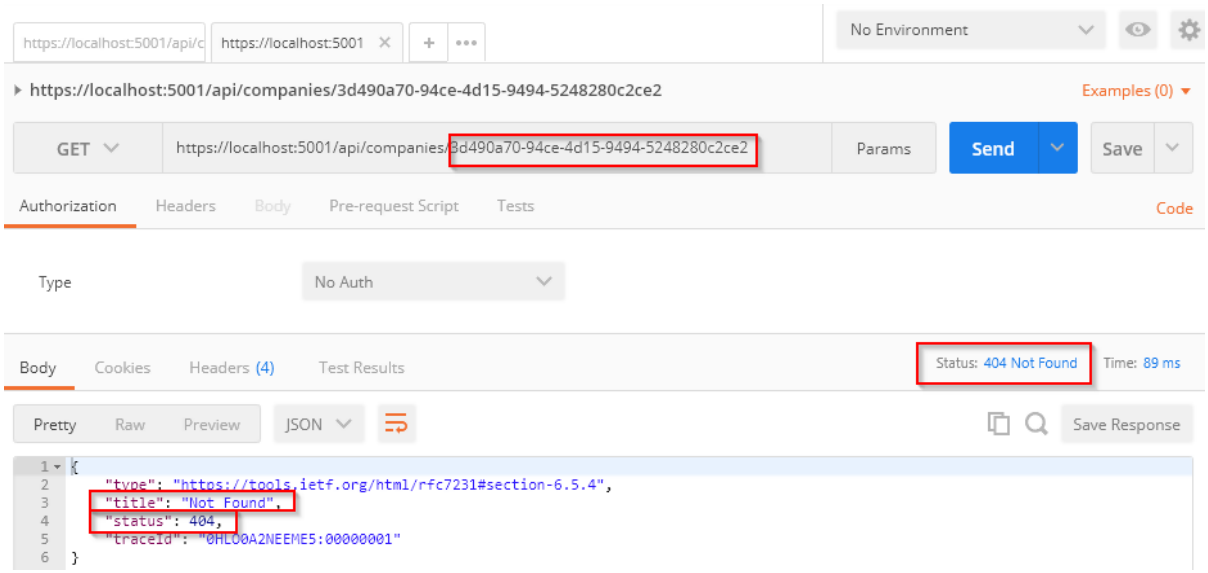
Let's use Postman to send valid and invalid requests towards our API:

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



Invalid request:

```
https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce2
```



Parent/Child Relationships in Web API

Up until now, we have been working only with the company, which is a parent (principal) entity in our API. But for each company, we have a related employee (dependent entity). Every employee must be related to a certain company and we are going to create our URIs in that manner.

That said, let's create a new controller and name it `EmployeesController`:

```
[Route("api/companies/{companyId}/employees")]
[ApiController]
public class EmployeesController : ControllerBase
{
    private readonly IRepositoryManager _repository;
    private readonly ILoggerManager _logger;
    private readonly IMapper _mapper;

    public EmployeesController(IRepositoryManager repository, ILoggerManager logger,
IMapper mapper)
    {
        _repository = repository;
        _logger = logger;
        _mapper = mapper;
    }
}
```

We are familiar with this code, but our main route is a bit different. As we said, a single employee can't exist without a company entity and this is exactly what are we exposing through this URI. To get an employee or

employees from the database, we have to specify the **companyId** parameter, and that is something all actions will have in common. For that reason, we have specified this route as our root route.

Before we create an action to fetch all the employees per company, we have to modify the **IEmployeeRepository** interface:

```
public interface IEmployeeRepository
{
    IEnumerable<Employee> GetEmployees(Guid companyId, bool trackChanges);
}
```

After interface modification, we are going to modify the **EmployeeRepository** class:

```
public IEnumerable<Employee> GetEmployees(Guid companyId, bool trackChanges) =>
    FindByCondition(e => e.CompanyId.Equals(companyId), trackChanges)
     .OrderBy(e => e.Name);
```

Finally, let's modify the Employees controller:

```
[HttpGet]
public IActionResult GetEmployeesForCompany(Guid companyId)
{
    var company = _repository.Company.GetCompany(companyId, trackChanges: false);
    if(company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
    }

        var employeesFromDb = _repository.Employee.GetEmployees(companyId,
trackChanges: false);

        return Ok(employeesFromDb);
 }
```

This code is pretty straightforward — nothing we haven't seen so far — but we need to explain just one thing. As you can see, we have the **companyId** parameter in our action and this parameter will be mapped from the main route. For that reason, we didn't place it in the [HttpGet] attribute as we did with the **GetCompany** action.

But, we know that something is wrong here because we are using a model in our response and not a data transfer object. To fix that, let's add another class in the `DataTransferObjects` folder:

```
public class EmployeeDto
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Position { get; set; }
}
```

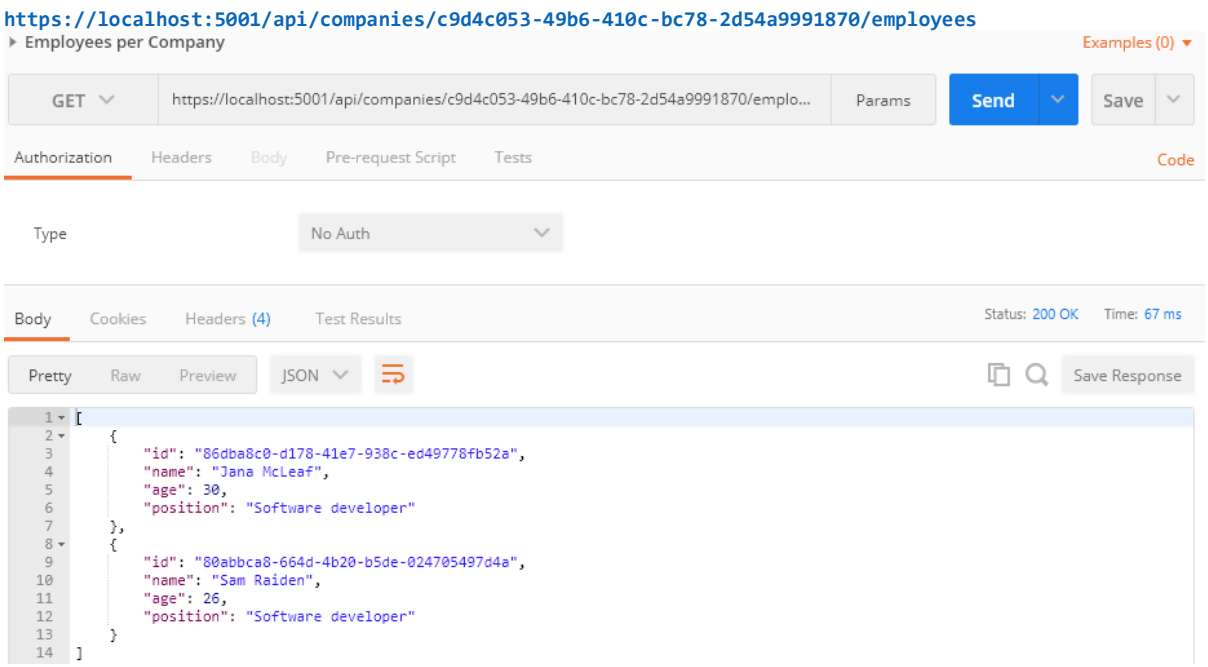
After that, let's create another mapping rule:

```
public MappingProfile()
{
    CreateMap<Company, CompanyDto>()
      .ForMember(c => c.FullAddress,
         opt => opt.MapFrom(x => string.Join(' ', x.Address, x.Country)));

    CreateMap<Employee, EmployeeDto>();
}
```

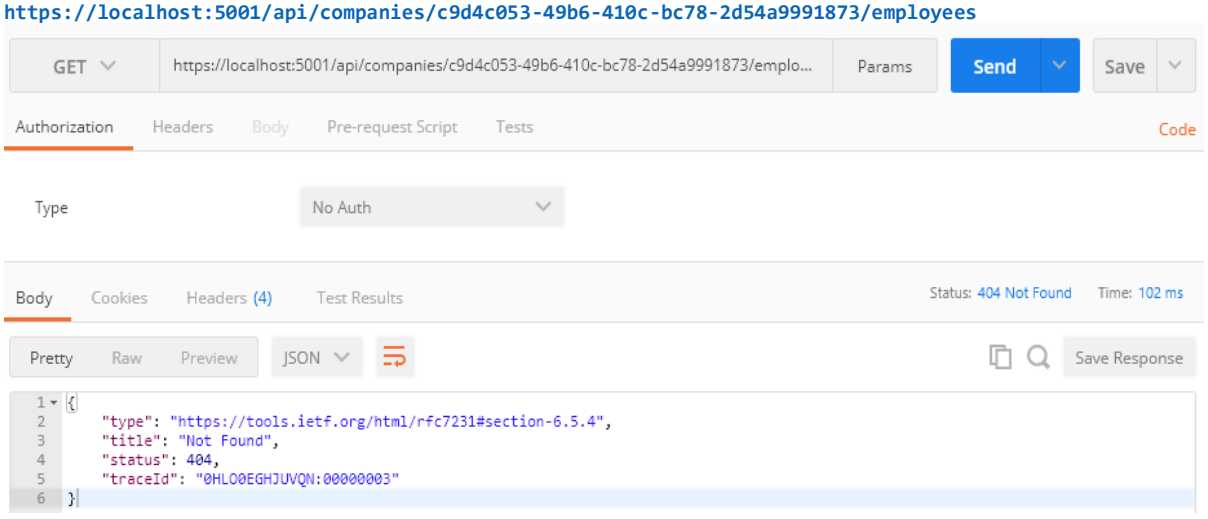
Finally, let's modify our action:

```
[HttpGet]
public IActionResult GetEmployeesForCompany(Guid companyId)
{
    var company = _repository.Company.GetCompany(companyId, trackChanges: false);
    if(company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
    }

        var employeesFromDb = _repository.Employee.GetEmployees(companyId,
trackChanges: false);

        var employeesDto = _mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

        return Ok(employeesDto);
}
```

That done, we can send a request with a valid companyId:



And with an invalid companyId:



Excellent. Let's continue by fetching a single employee.

Getting a Single Employee for Company

So, as we did in previous sections, let's start with an interface modification:

```
public interface IEmployeeRepository
{
    IEnumerable<Employee> GetEmployees(Guid companyId, bool trackChanges);
    Employee GetEmployee(Guid companyId, Guid id, bool trackChanges);
}
```

Now, let's implement this method in the **EmployeeRepository** class:

```
public Employee GetEmployee(Guid companyId, Guid id, bool trackChanges) =>
    FindByCondition(e => e.CompanyId.Equals(companyId) && e.Id.Equals(id),
trackChanges)
    .SingleOrDefault();
```

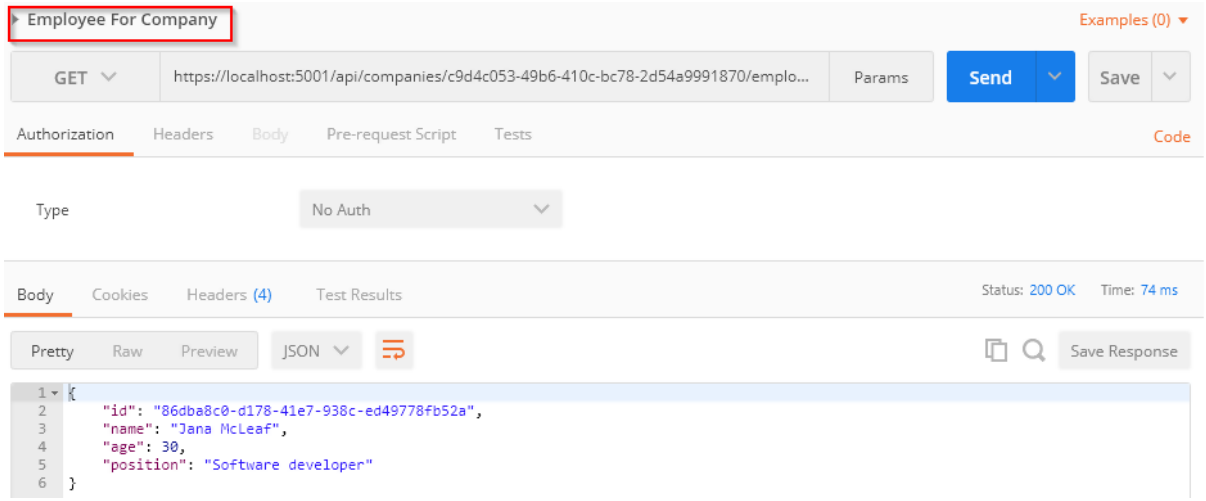
Finally, let's modify the **EmployeeController** class:

```
[HttpGet("{id}")]
public IActionResult GetEmployeeForCompany(Guid companyId, Guid id)
{
    var company = _repository.Company.GetCompany(companyId, trackChanges: false);
    if(company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
    }

    var employeeDb = _repository.Employee.GetEmployee(companyId, id, trackChanges:
false);
    if(employeeDb == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    var employee = _mapper.Map<EmployeeDto>(employeeDb);

    return Ok(employee);
}
```

Excellent.

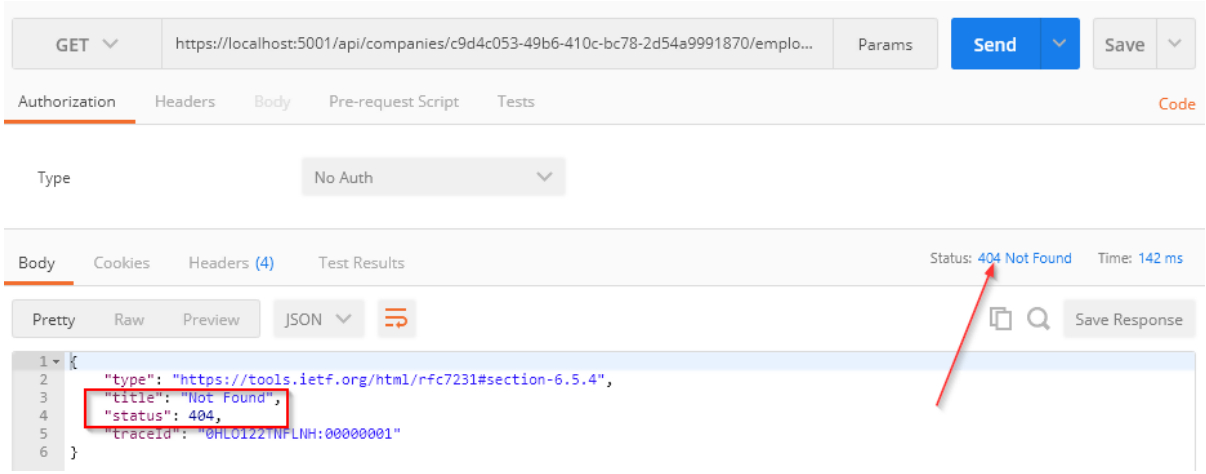
We can test this action by using already created requests from the **CompanyEmployeesRequests** file placed in the folder with the exercise files:

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



When we send the request with an invalid company or employee id:

https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/86dba8c0-d178-41e7-938c-ed49778fb52c



Our results are pretty self explanatory.

Until now, we have received only JSON formatted responses from our API. But what if we want to support some other format, like XML for example?

Well, in the next chapter we are going to learn more about Content Negotiation and enabling different formats for our responses.

CONTENT NEGOTIATION

Content negotiation is one of the quality-of-life improvements we can add to our REST API to make it more user friendly and flexible. And when we design an API, isn't that what we want to achieve in the first place?

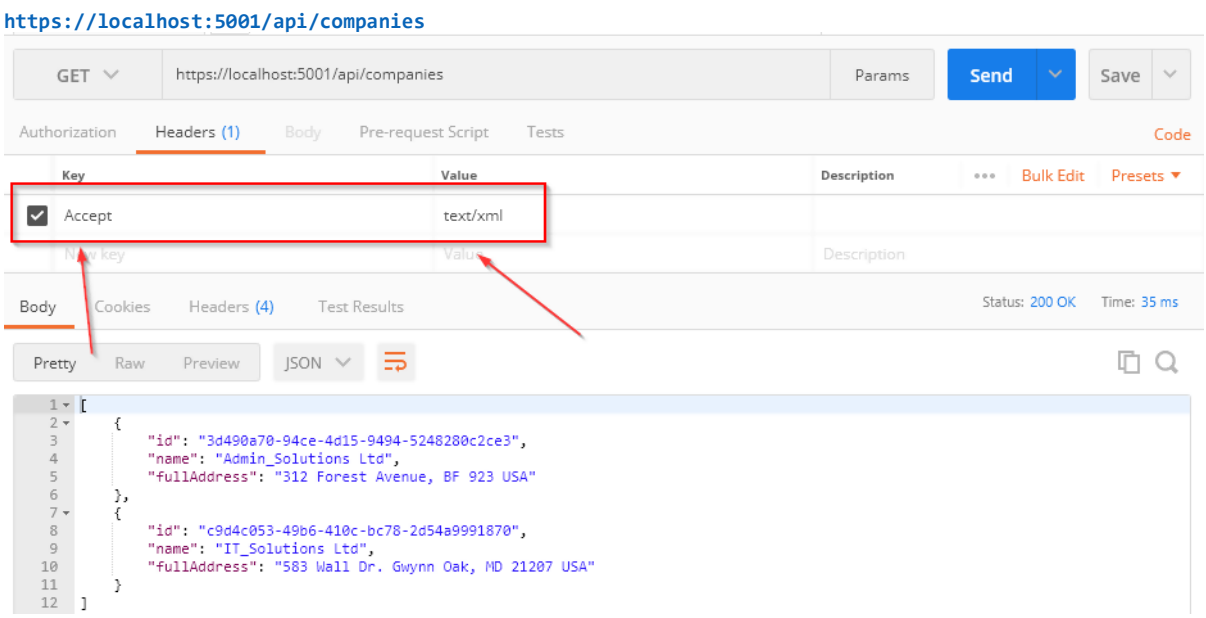
Content negotiation is an HTTP feature that has been around for a while, but for one reason or another, it is often a bit underused.

In short, content negotiation lets you choose or rather "negotiate" the content you want in to get in response to the REST API request.

What Do We Get Out of the Box?

By default, ASP.NET Core Web API returns a JSON formatted result.

We can confirm that by looking at the response from the `GetCompanies` action:



We can clearly see that the default result when calling GET on `/api/companies` returns the JSON result. We have also used the `Accept` header (as you can see in the picture above) to try forcing the server to return other media types like plain text and XML.

But that doesn't work. Why?

Because we need to configure server formatters to format a response the way we want it.

Let's see how to do that.

Changing the Default Configuration of Our Project

A server does not explicitly specify where it formats a response to JSON. But you can override it by changing configuration options through the `AddControllers` method.

We can add the following options to enable the server to format the XML response when the client tries negotiating for it:

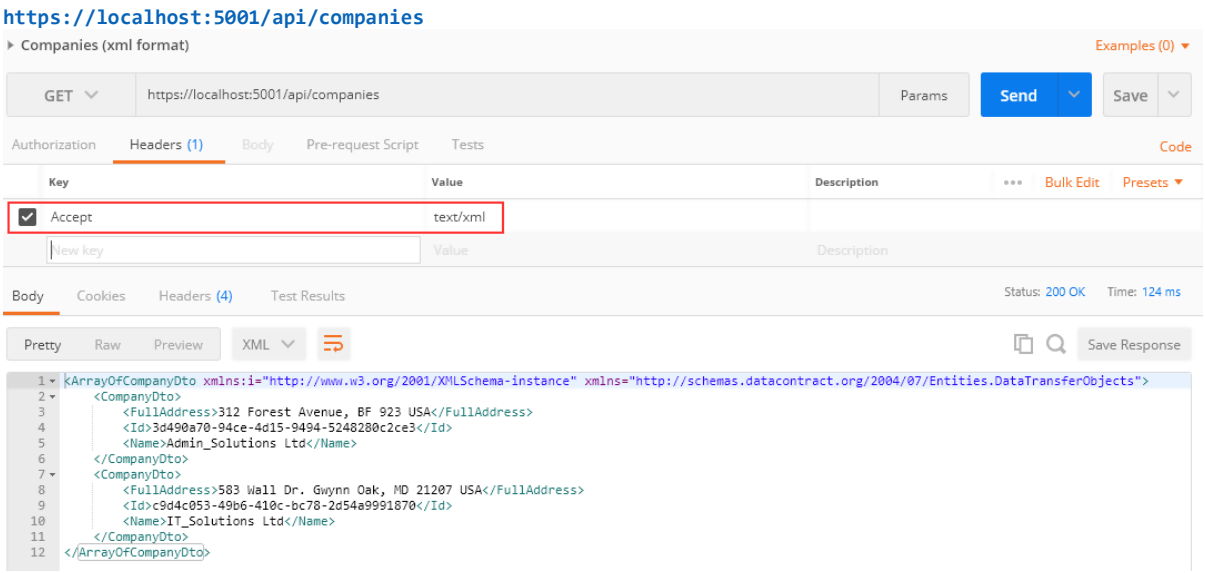
```
public void ConfigureServices(IServiceCollection services)
{
    services.ConfigureCors();
    services.ConfigureIISIntegration();
    services.ConfigureLoggerService();
    services.ConfigureSqlContext(Configuration);
    services.ConfigureRepositoryManager();
    services.AddAutoMapper(typeof(Startup));

    services.AddControllers(config =>
    {
        config.RespectBrowserAcceptHeader = true;
    }).AddXmlDataContractSerializerFormatters();
}
```

First things first, we must tell a server to respect the Accept header. After that, we just add the `AddXmlDataContractSerializerFormatters` method to support XML formatters.

Now that we have our server configured, let's test the content negotiation once more.

Testing Content Negotiation

Let's see what happens now if we fire the same request through Postman:



There is our XML response.

Now by changing the Accept header from `text/xml` to `text/json`, we can get differently formatted responses — and that is quite awesome, wouldn't you agree?

Okay, that was nice and easy.

But what if despite all this flexibility a client requests a media type that a server doesn't know how to format?

Restricting Media Types

Currently, it – the server - will default to a JSON type.

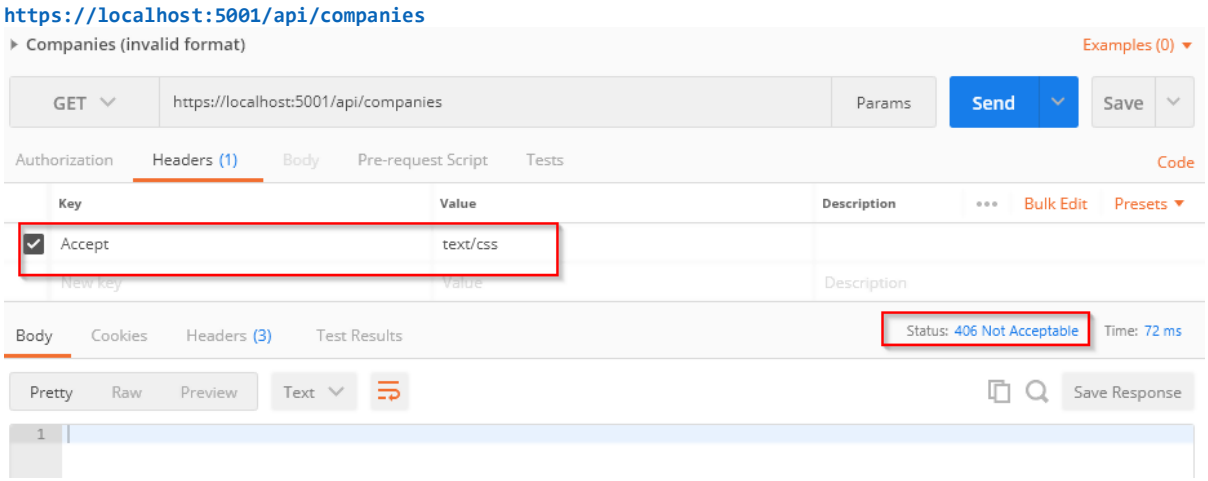
But we can restrict this behavior by adding one line to the configuration:

```
services.AddControllers(config =>
{
    config.RespectBrowserAcceptHeader = true;
    config.ReturnHttpNotAcceptable = true;
}).AddXmlDataContractSerializerFormatters();
```

We added the `ReturnHttpNotAcceptable = true` option, which tells the server that if the client tries to negotiate for the media type the server doesn't support, it should return the `406 Not Acceptable` status code.

This will make our application more restrictive and force the API consumer to request only the types the server supports. The 406 status code is created for this purpose.

Now, let's try fetching the `text/css` media type using Postman to see what happens:



And as expected, there is no response body and all we get is a nice `406 Not Acceptable` status code.

So far so good.

More About Formatters

If we want our API to support content negotiation for a type that is not "in the box," we need to have a mechanism to do this.

So, how can we do that?

ASP.NET Core supports the creation of **custom formatters**. Their purpose is to give us the flexibility to create our own formatter for any media types we need to support.

We can make the custom formatter by using the following method:

- Create an output formatter class that inherits the **TextOutputFormatter** class.
- Create an input formatter class that inherits the **TextInputformatter** class.
- Add input and output classes to the InputFormatters and OutputFormatters collections the same way we did for the XML formatter.

Now let's have some fun and implement a custom CSV formatter for our example.

Implementing a Custom Formatter

Since we are only interested in formatting responses, we need to implement only an output formatter. We would need an input formatter only if a request body contained a corresponding type.

The idea is to format a response to return the list of companies in a CSV format.

Let's add a `CsvOutputFormatter` class to our main project:

```
public class CsvOutputFormatter : TextOutputFormatter
{
    public CsvOutputFormatter()
    {
        SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/csv"));
        SupportedEncodings.Add(Encoding.UTF8);
        SupportedEncodings.Add(Encoding.Unicode);
    }

    protected override bool CanWriteType(Type type)
    {
        if (typeof(CompanyDto).IsAssignableFrom(type) ||
typeof(IEnumerable<CompanyDto>).IsAssignableFrom(type))
        {
            return base.CanWriteType(type);
        }

        return false;
    }

    public override async Task WriteResponseBodyAsync(OutputFormatterWriteContext
context, Encoding selectedEncoding)
```

```
    {
        var response = context.HttpContext.Response;
        var buffer = new StringBuilder();

        if (context.Object is IEnumerable<CompanyDto>)
        {
            foreach (var company in (IEnumerable<CompanyDto>)context.Object)
            {
                FormatCsv(buffer, company);
            }
        }
        else
        {
            FormatCsv(buffer, (CompanyDto)context.Object);
        }

        await response.WriteAsync(buffer.ToString());
    }



    private static void FormatCsv(StringBuilder buffer, CompanyDto company)
    {
        buffer.AppendLine($"{company.Id},\"{company.Name},\"{company.FullAddress}\"");
    }
}
```

There are a few things to note here:

- In the constructor, we define which media type this formatter should parse as well as encodings.
- The `CanWriteType` method is overridden, and it indicates whether or not the CompanyDto type can be written by this serializer.
- The `WriteResponseBodyAsync` method constructs the response.
- And finally, we have the `FormatCsv` method that formats a response the way we want it.

The class is pretty straightforward to implement, and the main thing that you should focus on is the `FormatCsvmethod` logic.

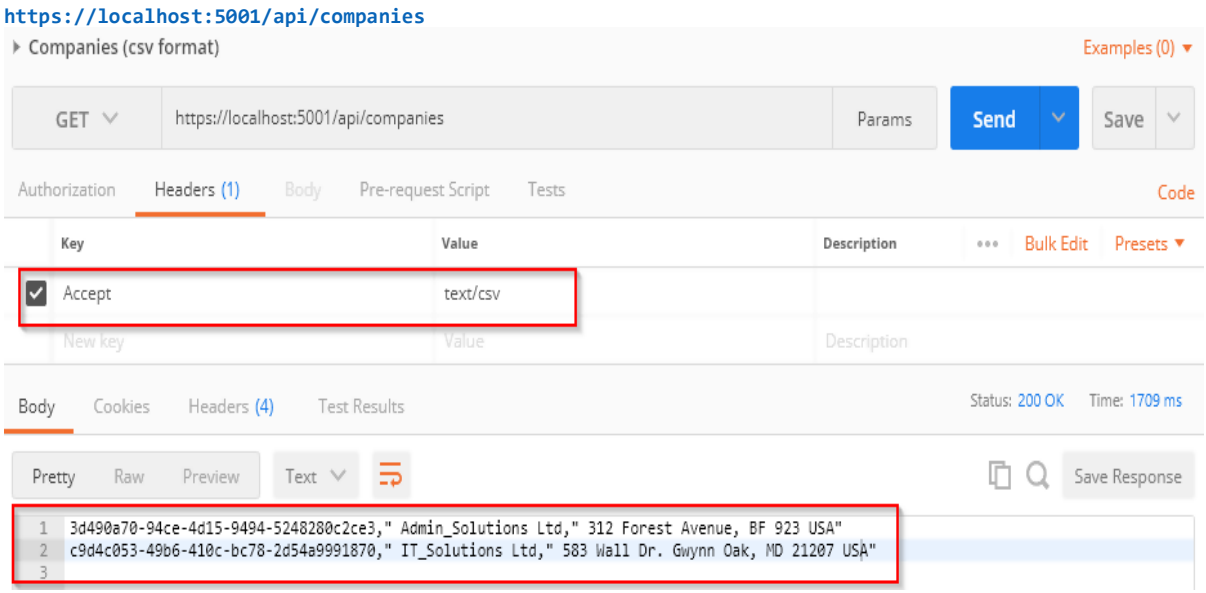
Now we just need to add the newly made formatter to the list of `OutputFormatters` in the `ServicesExtensions` class:

```
public static IMvcBuilder AddCustomCSVFormatter(this IMvcBuilder builder) =>
    builder.AddMvcOptions(config => config.OutputFormatters.Add(new
CsvOutputFormatter()));
```

And to call it in the `AddControllers`:

```
services.AddControllers(config =>
{
    config.RespectBrowserAcceptHeader = true;
    config.ReturnHttpNotAcceptable = true;
}).AddXmlDataContractSerializerFormatters()
  .AddCustomCSVFormatter();
```

Let's run this and see if it actually works. This time we will put `text/csv` as the value for the `Accept` header:



Well, what do you know, it works!

In this chapter, we finished working with GET requests in our project and we are ready to move on to the POST PUT and DELETE requests. We have a lot more ground to cover, so let's get down to business.

METHOD SAFETY AND METHOD IDEMPOTENCY

Before we start with the Create, Update, and Delete actions, we should explain two important principles in the HTTP standard. Those standards are Method Safety and Method Idempotency.

We can consider a method a safe one if it doesn't change the resource representation. So, in other words, the resource shouldn't be changed after our method is executed.

If we can call a method multiple times with the same result, we can consider that method idempotent. So in other words, the side effects of calling it once are the same as calling it multiple times.

Let's see how this applies to HTTP methods:

HTTP Method	Is it Safe?	Is it Idempotent?
GET	Yes	Yes
OPTIONS	Yes	Yes
HEAD	Yes	Yes
POST	No	No
DELETE	No	Yes
PUT	No	Yes
PATCH	No	No

As you can see, the GET, OPTIONS, and HEAD methods are both safe and idempotent, because when we call those methods they will not change the resource representation. Furthermore, we can call these methods multiple times, but they will return the same result every time.

The POST method is neither safe nor idempotent. It causes changes in the resource representation because it creates them. Also, if we call the POST method multiple times, it will create a new resource every time.

The DELETE method is not safe because it removes the resource, but it is idempotent because if we delete the same resource multiple times, we will get the same result as if we have deleted it only once.

PUT is not safe either. When we update our resource, it changes. But it is idempotent because no matter how many times we update the same resource with the same request it will have the same representation as if we have updated it only once.

Finally, the PATCH method is neither safe nor idempotent.

Now that we've learned about these principles, we can continue with our application by implementing the rest of the HTTP methods (we have already implemented GET). We can always use this table to decide which method to use for which use case.

CREATING RESOURCES

In this section, we are going to show you how to use the POST HTTP method to create resources in the database.

So, let's start.

Handling POST Requests

Firstly, let's modify the decoration attribute for the `GetCompany` action in the `Companies` controller:

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

With this modification, we are setting the name for the action. This name will come in handy in the action method for creating a new company.

We have a DTO class for the output (the GET methods), but right now we need the one for the input as well. So, let's create a new class in the `Entities/DataTransferObjects` folder:

```
public class CompanyForCreationDto
{
    public string Name { get; set; }
    public string Address { get; set; }
    public string Country { get; set; }
}
```

We can see that this DTO class is almost the same as the `Company` class, but without the Id property. We don't need that property when we create an entity.

We should pay attention to one more thing. In some projects, the input and output DTO classes are the same, but we still recommend separating them for easier maintenance and refactoring of our code. Furthermore, when we start talking about validation, we don't want to validate the output objects — but we definitely want to validate the input ones.

With all of that said and done, let's continue by modifying the **ICompanyRepository** interface:

```
public interface ICompanyRepository
{
    IEnumerable<Company> GetAllCompanies(bool trackChanges);
    Company GetCompany(Guid companyId, bool trackChanges);
    void CreateCompany(Company company);
}
```

After the interface modification, we are going to implement that interface:

```
public void CreateCompany(Company company) => Create(company);
```

We don't explicitly generate a new Id for our company; this would be done by EF Core. All we do is to set the state of the company to Added.

Before we add a new action in our **Companies** controller, we have to create another mapping rule for the **Company** and **CompanyForCreationDto** objects. Let's do this in the **MappingProfile** class:

```
public MappingProfile()
{
    CreateMap<Company, CompanyDto>()
      .ForMember(c => c.FullAddress,
         opt => opt.MapFrom(x => string.Join(' ', x.Address, x.Country)));

    CreateMap<Employee, EmployeeDto>();

    CreateMap<CompanyForCreationDto, Company>();
}
```

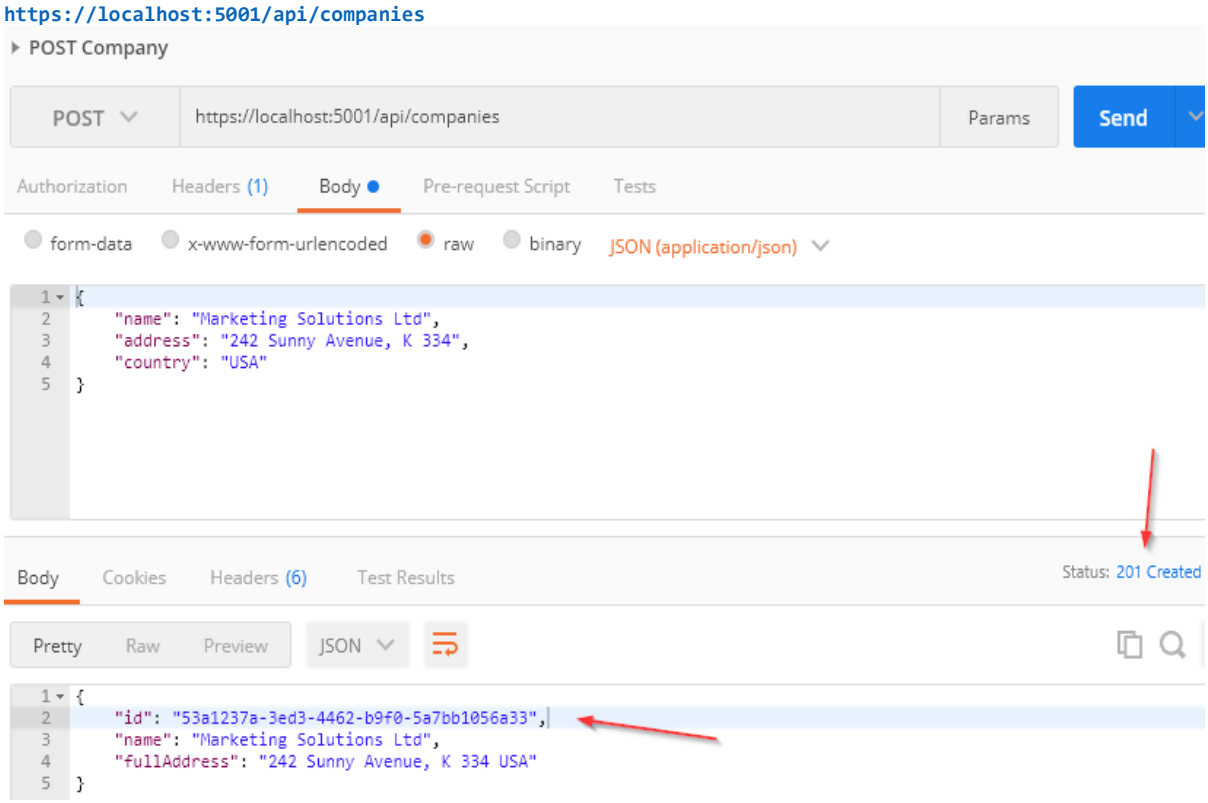
Our POST action will accept a parameter of the type **CompanyForCreationDto**, but we need the Company object for creation. Therefore, we have to create this mapping rule.

Last, let's modify the controller:

```
[HttpPost]
public IActionResult CreateCompany([FromBody]CompanyForCreationDto company)
{
    if(company == null)
    {
        _logger.LogError("CompanyForCreationDto object sent from client is null.");
```

```
        return BadRequest("CompanyForCreationDto object is null");
    }

    var companyEntity = _mapper.Map<Company>(company);

    _repository.Company.CreateCompany(companyEntity);
    _repository.Save();

    var companyToReturn = _mapper.Map<CompanyDto>(companyEntity);

    return CreatedAtRoute("CompanyById", new { id = companyToReturn.Id },
companyToReturn);
}
```

Let's use Postman to send the request and examine the result:



Code Explanation

Let's talk a little bit about this code. The interface and the repository parts are pretty clear, so we won't talk about that. But the code in the controller contains several things worth mentioning.

If you take a look at the request URI, you'll see that we use the same one as for the GetCcompanies action: `api/companies` — but this time we are using the POST request.

The `CreateCompany` method has its own `[HttpPost]` decoration attribute, which restricts it to POST requests. Furthermore, notice the company parameter which comes from the client. We are not collecting it from the URI, but from the request body. Thus the usage of the `[FromBody]` attribute. Also, the company object is a complex type; therefore, we have to use `[FromBody].`

If we wanted to, we could explicitly mark the action to take this parameter from the URI by decorating it with the `[FromUri]` attribute, though we wouldn't recommend that at all because of security reasons and the complexity of the request.

Because the `company` parameter comes from the client, it could happen that it can't be deserialized. As a result, we would need to validate it against the reference type's default value, which is null.

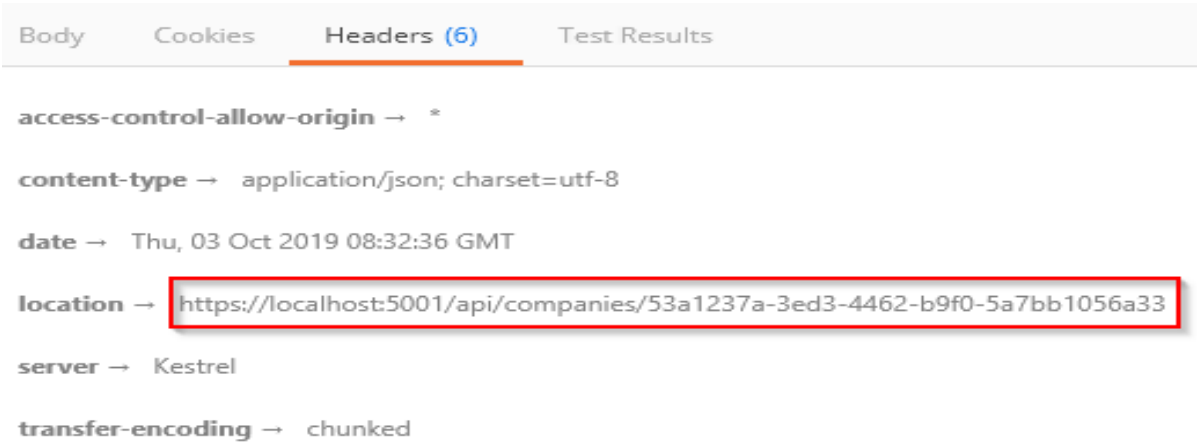
After validation, we map the company for creation to the company entity, call the repository method for creation, and call the `Save()` method to save the entity to the database. After the save action, we map the company entity to the company DTO object to return it to the client.

The last thing to mention is this part of the code:

```
CreatedAtRoute("CompanyById", new { id = companyToReturn.Id }, companyToReturn);
```

`CreatedAtRoute` will return a status code 201, which stands for `Created`. Also, it will populate the body of the response with the new company object as well as the Location attribute within the response header with the address to retrieve that company. We need to provide the name of the action, where we can retrieve the created entity.

If we take a look at the headers part of our response, we are going to see a link to retrieve the created company:



Finally, from the previous example, we can confirm that the POST method is neither safe nor idempotent. We saw that when we send the POST request, it is going to create a new resource in the database — thus changing the resource representation. Furthermore, if we try to send this request a couple of times, we will get a new object for every request (it will have a different Id for sure).

Let's continue with child resources creation.

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:

```
public class EmployeeForCreationDto
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Position { get; set; }
}
```

We don't have the `Id` property because, we are going to create that Id on the server side. But additionaly, we don't have the `CompanyId` because

we accept that parameter through the route:

```
[Route("api/companies/{companyId}/employees")]
```

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

```
public interface IEmployeeRepository
{
    IEnumerable<Employee> GetEmployees(Guid companyId, bool trackChanges);
    Employee GetEmployee(Guid companyId, Guid id, bool trackChanges);
    void CreateEmployeeForCompany(Guid companyId, Employee employee);
}
```

Of course, we have to implement this interface:

```
public void CreateEmployeeForCompany(Guid companyId, Employee employee)
{
    employee.CompanyId = companyId;
    Create(employee);
}
```

Because we are going to accept the employee DTO object in our action, but we also have to send an employee object to this repository method, we have to create an additional mapping rule in the **MappingProfile** class:

```
CreateMap<EmployeeForCreationDto, Employee>();
```

Now, we can add a new action in the EmployeesController:

```
[HttpPost]
public IActionResult CreateEmployeeForCompany(Guid companyId, [FromBody]
EmployeeForCreationDto employee)
{
    if(employee == null)
    {
        _logger.LogError("EmployeeForCreationDto object sent from client is null.");
        return BadRequest("EmployeeForCreationDto object is null");
    }

    var company = _repository.Company.GetCompany(companyId, trackChanges: false);
    if(company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
    }

    var employeeEntity = _mapper.Map<Employee>(employee);

    _repository.Employee.CreateEmployeeForCompany(companyId, employeeEntity);
    _repository.Save();
```

```
    var employeeToReturn = _mapper.Map<EmployeeDto>(employeeEntity);

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

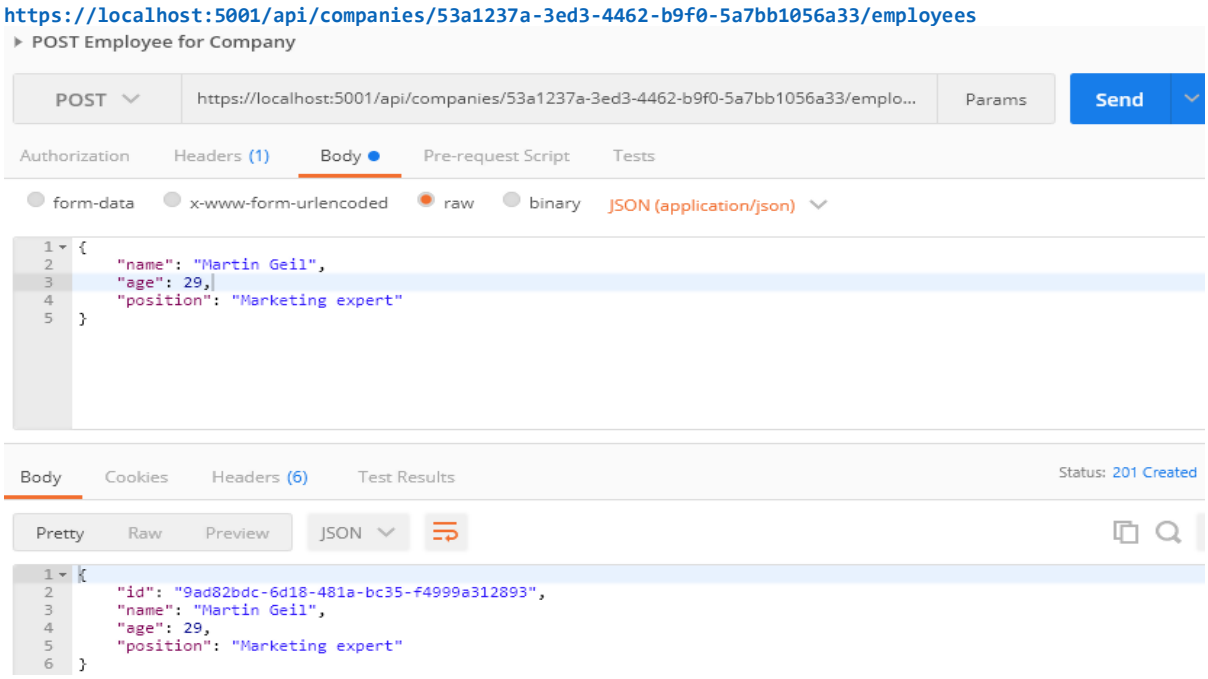
There are some differences in this code compared to the `CreateCompany` action. The first is that we have to check whether that company exists in the database because there is no point in creating an employee for a company that does not exist.

The second difference is the return statement, which now has two parameters for the anonymous object.

For this to work, we have to modify the HTTP attribute above the GetEmployeeForCompany action:

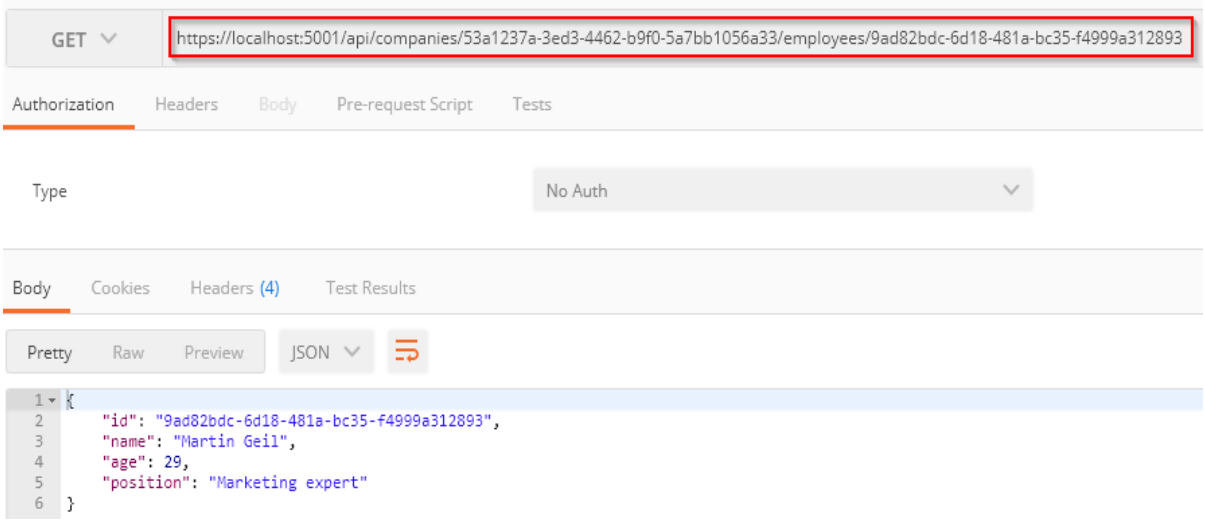
```
[HttpGet("{id}", Name = "GetEmployeeForCompany")]
```

Let's give this a try:



Excellent. A new employee was created.

If we take a look at the Headers tab, we'll see a link to fetch our newly created employee. If you copy that link and send another request with it, you will get this employee for sure:



Creating Children Resources Together with a Parent

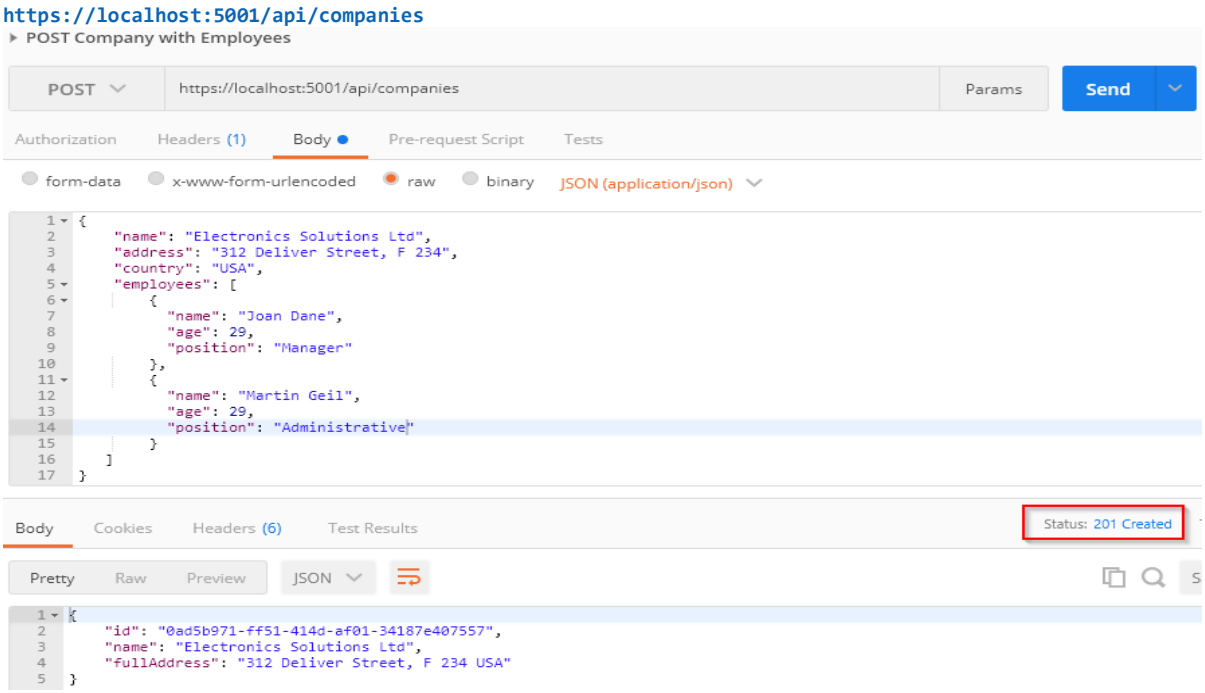
There are situations where we want to create a parent resource with its children. Rather than using multiple requests for every single child, we want to do this in the same request with the parent resource.

We are going to show you how to do this.

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

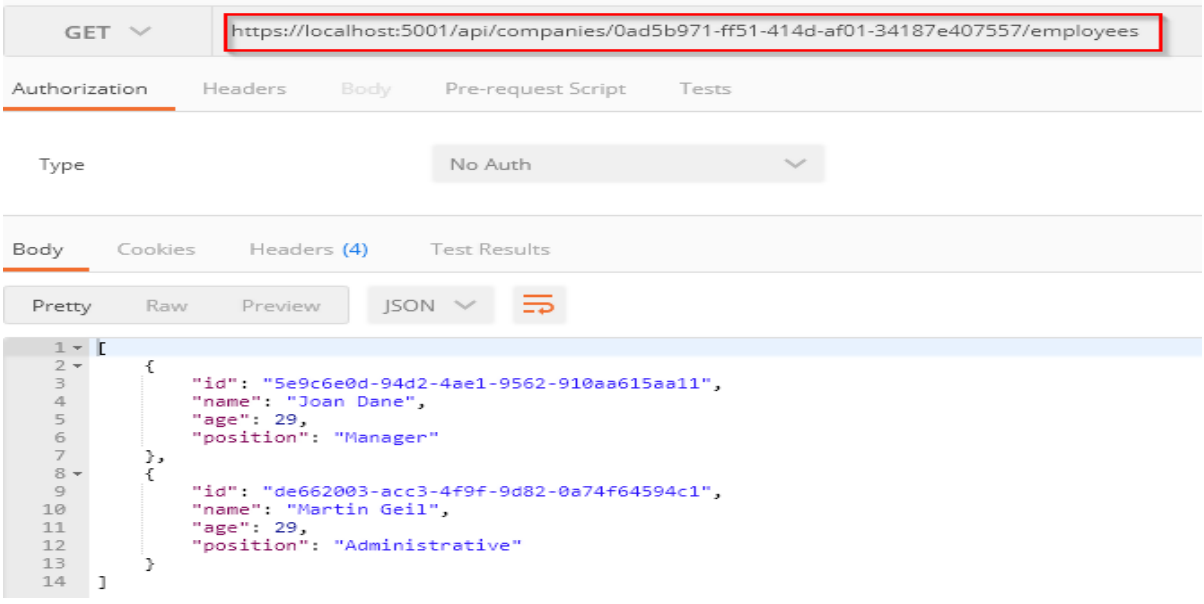
```
public class CompanyForCreationDto
{
    public string Name { get; set; }
    public string Address { get; set; }
    public string Country { get; set; }

    public IEnumerable<EmployeeForCreationDto> Employees { get; set; }
}
```

We are not going to change the action logic inside the controller nor the repository logic; everything is great there. That's all. Let's test it:



You can see that this company was created successfully.

Now we can copy the location link from the Headers tab, paste it in another Postman tab, and just add the `/employees` part:



We have confirmed that the employees were created as well.

Creating a Collection of Resources

Until now, we have been creating a single resource whether it was Company or Employee. But it is quite normal to create a collection of resources, and in this section that is something we are going to work with.

If we take a look at the `CreateCompany` action, for example, we can see that the `return` part points to the `CompanyById` route (the `GetCompany` action). That said, we don't have the GET action for the collection creating action to point to. So, before we start with the POST collection action, we are going to create the `GetCompanyCollection` action in the `Companies` controller.

But first, let's modify the `ICompanyRepository` interface:

```
IEnumerable<Company> GetByIds(IEnumerable<Guid> ids, bool trackChanges);
```

Then we have to change the `CompanyRepository` class:

```
public IEnumerable<Company> GetByIds(IEnumerable<Guid> ids, bool trackChanges) =>
    FindByCondition(x => ids.Contains(x.Id), trackChanges)
    .ToList();
```

After that, we can add a new action in the controller:

```
[HttpGet("collection/({ids})", Name = "CompanyCollection")]
public IActionResult GetCompanyCollection(IEnumerable<Guid> ids)
{
    if(ids == null)
    {
        _logger.LogError("Parameter ids is null");
        return BadRequest("Parameter ids is null");
    }

    var companyEntities = _repository.Company.GetByIds(ids, trackChanges: false);

    if(ids.Count() != companyEntities.Count())
    {
        _logger.LogError("Some ids are not valid in a collection");
        return NotFound();
    }

    var companiesToReturn = _mapper.Map<IEnumerable<CompanyDto>>(companyEntities);
    return Ok(companiesToReturn);
}
```

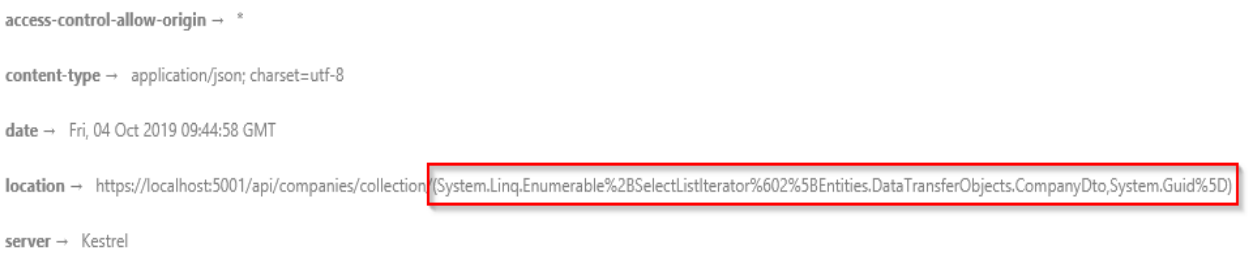
And that's it. These actions are pretty straightforward, so let's continue towards POST implementation:

```
[HttpPost("collection")]
public IActionResult CreateCompanyCollection([FromBody]
IEnumerable<CompanyForCreationDto> companyCollection)
{
    if(companyCollection == null)
    {
        _logger.LogError("Company collection sent from client is null.");
        return BadRequest("Company collection is null");
    }

    var companyEntities = _mapper.Map<IEnumerable<Company>>(companyCollection);
    foreach (var company in companyEntities)
    {
        _repository.Company.CreateCompany(company);
    }

    _repository.Save();

    var companyCollectionToReturn =
_mapper.Map<IEnumerable<CompanyDto>>(companyEntities);
    var ids = string.Join(",", companyCollectionToReturn.Select(c => c.Id));

    return CreatedAtRoute("CompanyCollection", new { ids },
companyCollectionToReturn);
}
```

So, we check if our collection is null and if it is, we return a bad request. If it isn't, then we map that collection and save all the collection elements to the database. Finally, we take all the ids as a comma-separated string and navigate to the GET action for fetching our created companies.

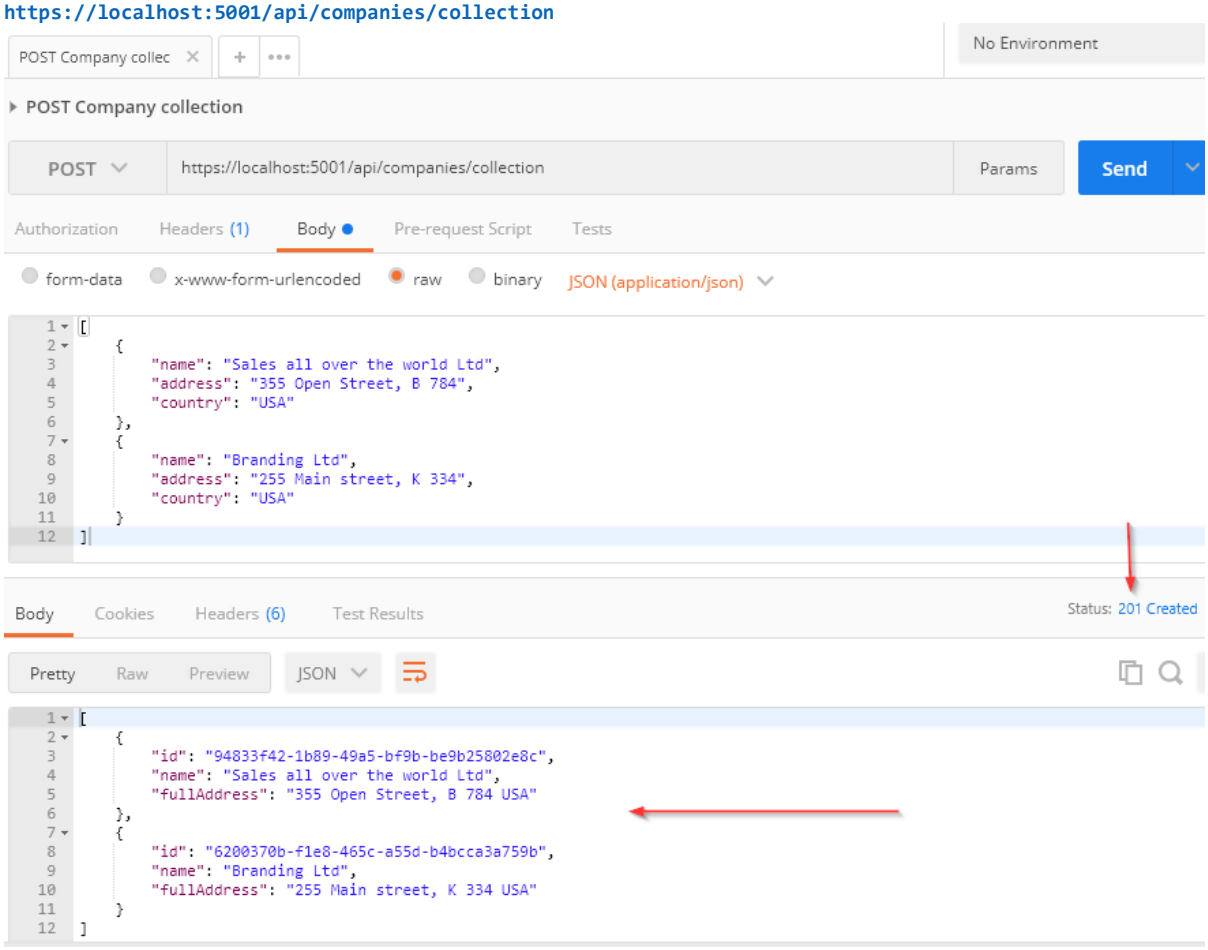
Now you may ask, why are we sending a comma-separated string when we expect a collection of ids in the **`GetCompanyCollection`** action?

Well, we can't just pass a list of ids in the **`CreatedAtRoute`** method because there is no support for the Header Location creation with the list. You may try it, but we're pretty sure you would get the location like this:

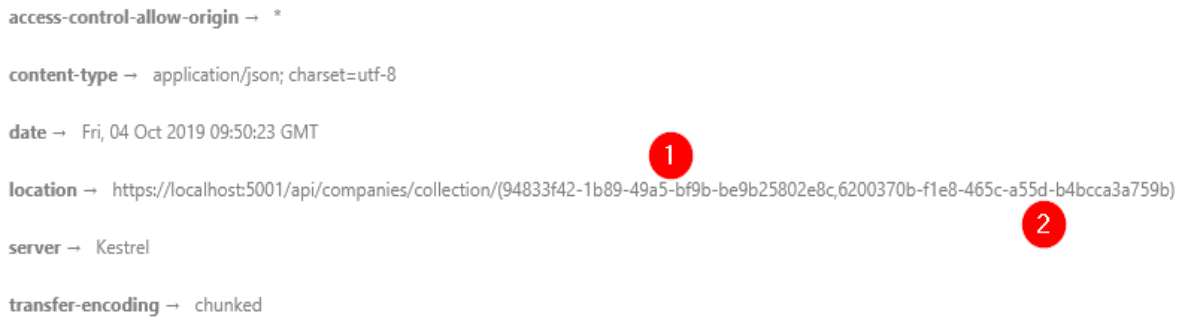


We can test our create action now:

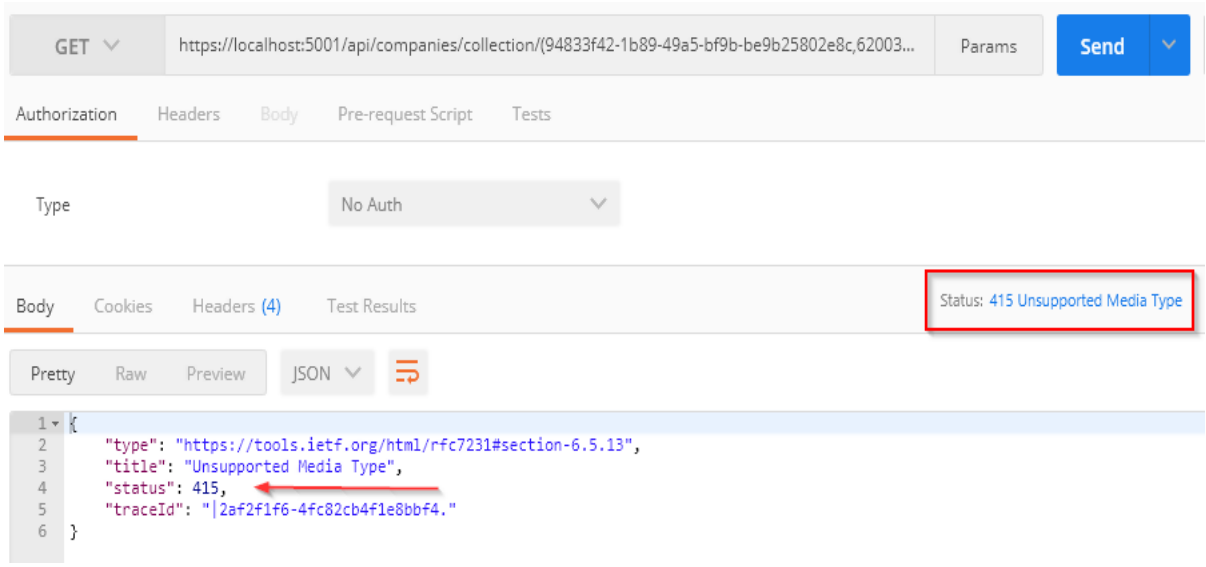
https://localhost:5001/api/companies/collection



Excellent. Let’s check the header tab:



We can see a valid location link. So, we can copy it and try to fetch our newly created companies:



But we are getting the `415 Unsupported Media Type` message. This is because our API can't bind the `string` type parameter to the `IEnumerable<Guid>` argument.

Well, we can solve this with a custom model binding.

Model Binding in API

Let's create the new folder `ModelBinders` in the main project and inside the new class `ArrayModelBinder`:

```
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;
    }
}
```

At first glance, this code might be hard to comprehend, but once we explain it, it will be easier to understand.

We are creating a model binder for the `IEnumerable` type. Therefore, we have to check if our parameter is the same type.

Next, we extract the value (a comma-separated string of GUIDs) with the `ValueProvider.GetValue()` expression. Because it is type string, we just check whether it is null or empty. If it is, we return null as a result because we have a null check in our action in the controller. If it is not, we move on.

In the `genericType` variable, with the reflection help, we store the type the `IEnumerable` consists of. In our case, it is GUID. With the `converter` variable, we create a converter to a GUID type. As you can

see, we didn't just force the GUID type in this model binder; instead, we inspected what is the nested type of the `IEnumerable` parameter and then created a converter for that exact type, thus making this binder generic.

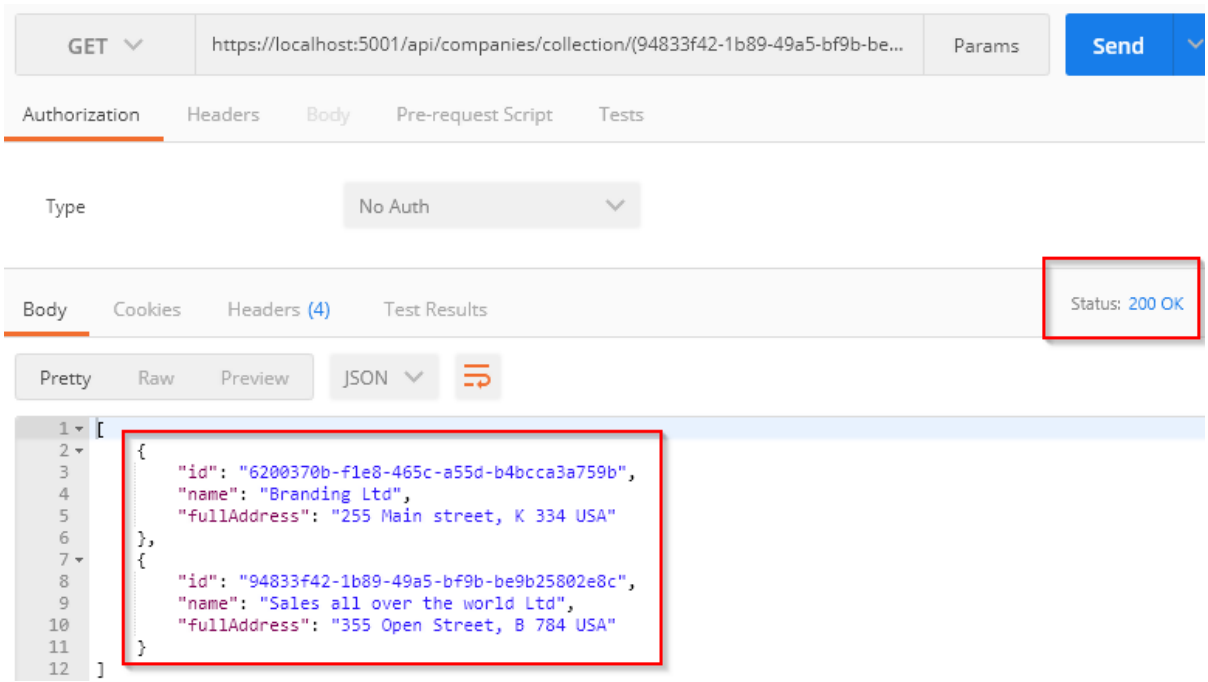
After that, we create an array of type object (`objectArray`) that consist of all the GUID values we sent to the API and then create an array of GUID types (`guidArray`), copy all the values from the `objectArray` to the `guidArray`, and assign it to the `bindingContext`.

And that is it. Now, we have just to make a slight modification in the `GetCompanyCollection` action:

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

Excellent.

Our `ArrayModelBinder` will be triggered before an action executes. It will convert the sent string parameter to the IEnumerable<Guid> type, and then the action will be executed:



Well done.

We are ready to continue towards DELETE actions.

WORKING WITH DELETE REQUESTS

Let's start this section by deleting a child resource first.

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

```
public interface IEmployeeRepository
{
    IEnumerable<Employee> GetEmployees(Guid companyId, bool trackChanges);
    Employee GetEmployee(Guid companyId, Guid id, bool trackChanges);
    void CreateEmployeeForCompany(Guid companyId, Employee employee);
    void DeleteEmployee(Employee employee);
}
```

The next step for us is to modify the **EmployeeRepository** class:

```
public void DeleteEmployee(Employee employee)
{
    Delete(employee);
}
```

Finally, we can add a delete action to the controller class:

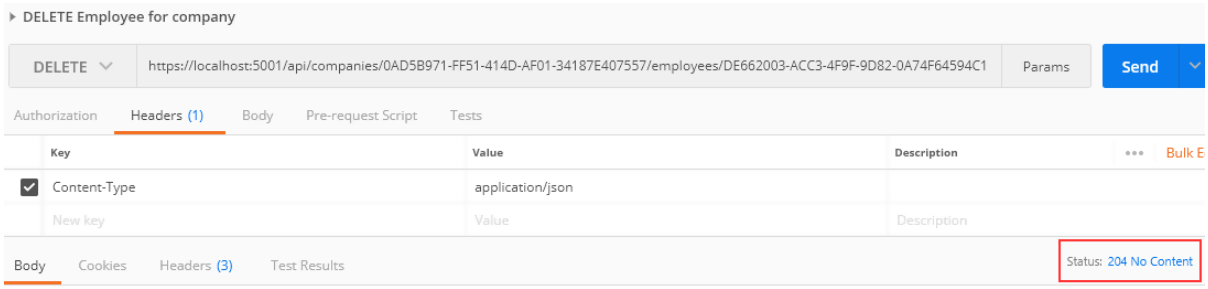
```
[HttpDelete("{id}")]
public IActionResult DeleteEmployeeForCompany(Guid companyId, Guid id)
{
    var company = _repository.Company.GetCompany(companyId, trackChanges: false);
    if(company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
    }

    var employeeForCompany = _repository.Employee.GetEmployee(companyId, id,
trackChanges: false);
    if(employeeForCompany == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    _repository.Employee.DeleteEmployee(employeeForCompany);
    _repository.Save();

    return NoContent();
}
```

There is nothing new with this action. We collect the companyId from the root route and the employee's id from the passed argument. We have to check if the company exists. Then, we check for the employee entity.

Finally, we delete our employee and return the `NoContent()` method, which returns the status code `204 No Content`.

Let's test this:

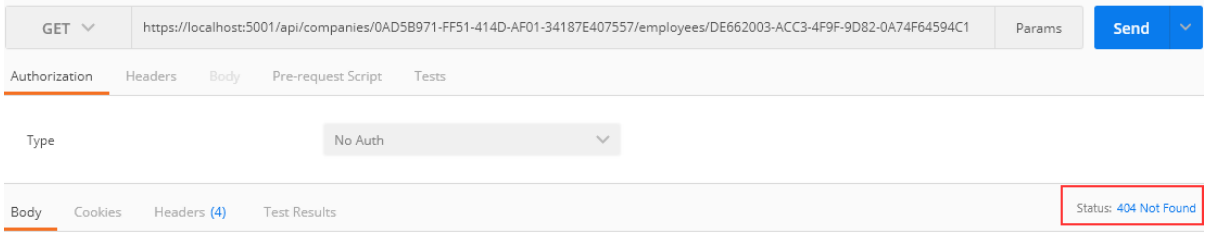
`https://localhost:5001/api/companies/0AD5B971-FF51-414D-AF01-34187E407557/employees/DE662003-ACC3-4F9F-9D82-0A74F64594C1`



Excellent. It works great.

You can try to get that employee from the database, but you will get 404 for sure:

`https://localhost:5001/api/companies/0AD5B971-FF51-414D-AF01-34187E407557/employees/DE662003-ACC3-4F9F-9D82-0A74F64594C1`



We can see that the DELETE request isn't safe because it deletes the resource, thus changing the resource representation. But if we try to send this delete request one or even more times, we would get the same 404 result because the resource doesn't exist anymore. That's what makes the DELETE request idempotent.

Deleting a Parent Resource with its Children

With Entity Framework Core, this action is pretty simple. With the basic configuration, cascade deleting is enabled, which means deleting a parent resource will automatically delete all of its children. We can confirm that from the migration file:

```
modelBuilder.Entity("Entities.Models.Employee", b =>
    {
        b.HasOne("Entities.Models.Company", "Company")
            .WithMany("Employees")
            .HasForeignKey("CompanyId")
            .OnDelete(DeleteBehavior.Cascade)
            .IsRequired();
    });
```

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

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

```
public interface ICompanyRepository
{
    IEnumerable<Company> GetAllCompanies(bool trackChanges);
    Company GetCompany(Guid companyId, bool trackChanges);
    void CreateCompany(Company company);
    IEnumerable<Company> GetByIds(IEnumerable<Guid> ids, bool trackChanges);
    void DeleteCompany(Company company);
}
```

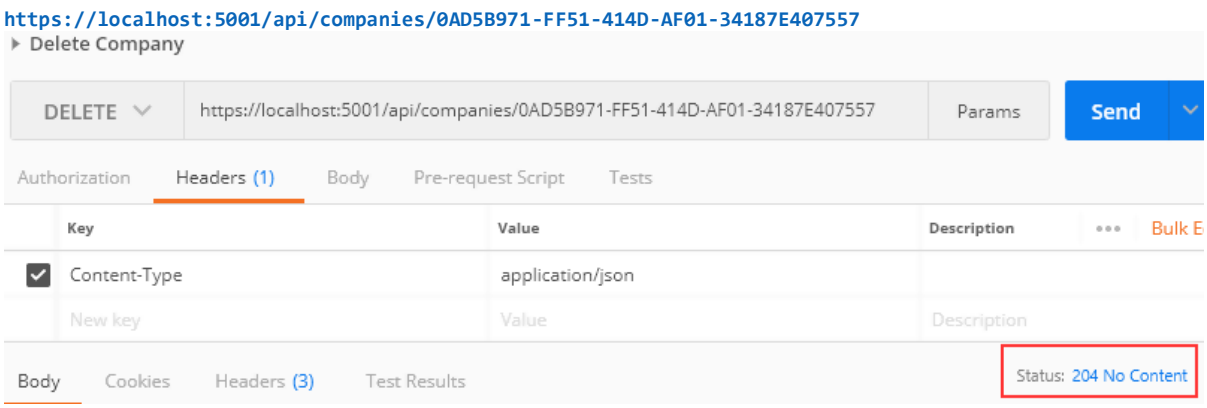
Then let's modify the repository class:

```
public void DeleteCompany(Company company)
{
    Delete(company);
}
```

Finally, let's modify our controller:

```
[HttpDelete("{id}")]
public IActionResult DeleteCompany(Guid id)
{
    var company = _repository.Company.GetCompany(id, trackChanges: false);
    if(company == null)
    {
        _logger.LogInfo($"Company with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    _repository.Company.DeleteCompany(company);
    _repository.Save();

    return NoContent();
}
```

And let's test our action:



It works.

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 to the PUT requests.

WORKING WITH PUT REQUESTS

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

Updating Employee

In the previous sections, we first changed our interface, then the repository class, and finally the controller. But for the update, this doesn't have to be the case.

Let's go step by step.

The first thing we are going to do is to create another DTO class for update purposes:

```
public class EmployeeForUpdateDto
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Position { get; set; }
}
```

We do not require the Id property because it will be accepted through the URI, like with the DELETE requests. Additionally, this DTO contains the same properties as the DTO for creation, but there is a conceptual difference between those two DTO classes. One is for updating and the other is for creating. Furthermore, once we get to the validation part, we will understand the additional difference between those two.

Because we have additional DTO class, we require an additional mapping rule:

```
CreateMap<EmployeeForUpdateDto, Employee>();
```

Now, when we have all of these, let's modify the `EmployeesController`:

```
[HttpPut("{id}")]
public IActionResult UpdateEmployeeForCompany(Guid companyId, Guid id, [FromBody]
       EmployeeForUpdateDto employee)
{
    if(employee == null)
    {
        _logger.LogError("EmployeeForUpdateDto object sent from client is null.");
        return BadRequest("EmployeeForUpdateDto object is null");
    }

    var company = _repository.Company.GetCompany(companyId, trackChanges: false);
    if(company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
    }

    var employeeEntity = _repository.Employee.GetEmployee(companyId, id, trackChanges:
true);
    if(employeeEntity == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    _mapper.Map(employee, employeeEntity);
    _repository.Save();

    return NoContent();
}
```

We are using the `PUT` attribute with the `id` parameter to annotate this action. That means that our route for this action is going to be: `api/companies/{companyId}/employees/{id}.`

As you can see, we have three checks in our code and they are familiar to us. But we have one difference. Pay attention to the way we fetch the `company` and the way we fetch the `employeeEntity`. Do you see the difference?

The `trackChanges` parameter is set to `true` for the `employeeEntity`. That's because we want EF Core to track changes on this entity. This means that as soon as we change any property in this entity, EF Core will set the state of that entity to `Modified`.

As you can see, we are mapping from the `employee` object (we will change just the age property in a request) to the `employeeEntity` — thus changing the state of the `employeeEntity` object to Modified.

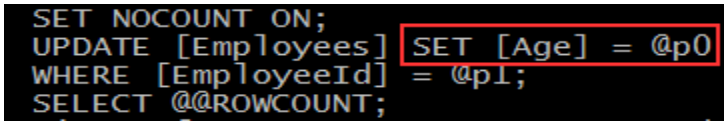
Because our entity has a modified state, it is enough to call the `Save` method without any additional update actions. As soon as we call the `Save` method, our entity is going to be updated in the database.

Finally, we return the 204 NoContent status.

We can test our action:

And it works; we get the 204 No Content status.

We can check our executed query through EF Core to confirm that only the Age column is updated:



Excellent.

You can send the same request with the invalid company id or employee id. In both cases, you should get a 404 response, which is a valid response for this kind of situation.

Additional note: As you can see, we have changed only the `Age` property, but we have sent all the other properties with unchanged values as well. Therefore, `Age` is only updated in the database. But if we send the object with just the `Age` property, without the other properties, those other properties will be set to their default values and the whole object will be updated — not just the `Age` column. That's because the PUT is a request for a full update. This is very important to know.

About the Update Method from the RepositoryBase Class

Right now, you might be asking: "Why do we have the Update method in the RepositoryBase class if we are not using it?"

The update action we just executed is a connected update (an update where we use the same context object to fetch the entity and to update it). But sometimes we can work with disconnected updates. This kind of update action uses different context objects to execute fetch and update actions or sometimes we can receive an object from a client with the Id property set as well, so we don't have to fetch it from the database. In that situation, all we have to do is to inform EF Core to track changes on that entity and to set its state to modified. We can do both actions with the `Update` method from our `RepositoryBase` class. So, you see, having that method is crucial as well.

One note, though. If we use the `Update` method from our repository, even if we change just the Age property, all properties will be updated in the database.

Inserting Resources while Updating One

While updating a parent resource, we can create child resources as well without too much effort. EF Core helps us a lot with that process. Let's see how.

The first thing we are going to do is to create a DTO class for update:

```
public class CompanyForUpdateDto
{
    public string Name { get; set; }
    public string Address { get; set; }
    public string Country { get; set; }

    public IEnumerable<EmployeeForCreationDto> Employees { get; set; }
}
```

After this, let's create a new mapping rule:

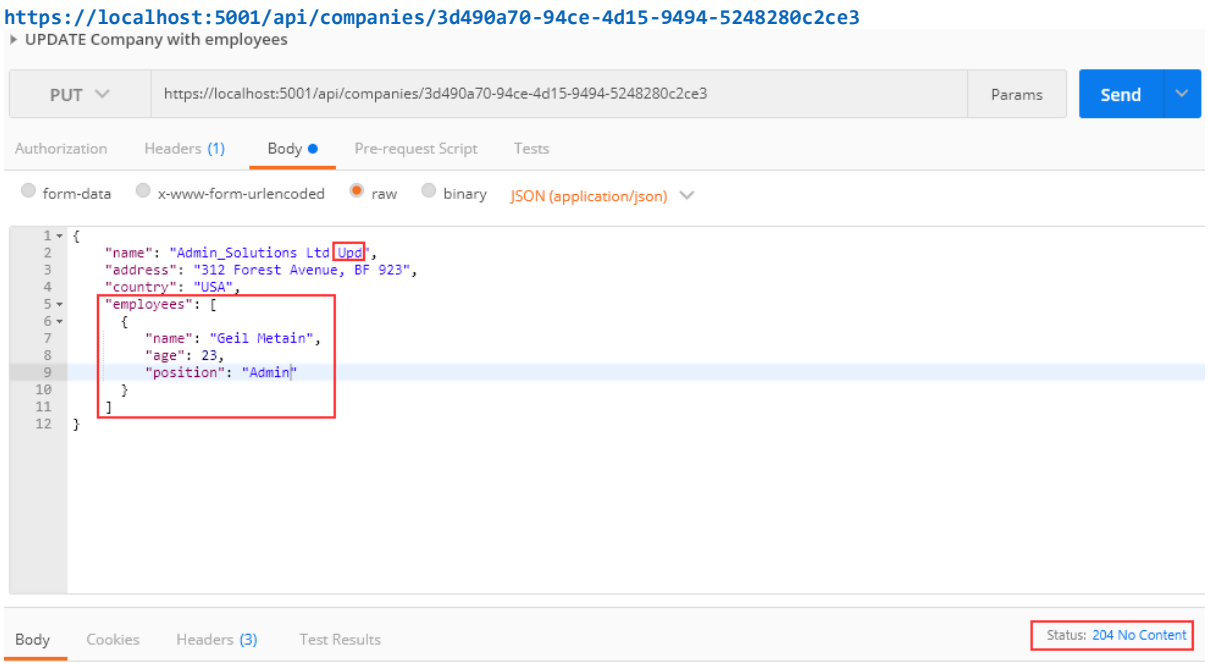
```
CreateMap<CompanyForUpdateDto, Company>();
```

Right now, we can modify our controller:

```
[HttpPut("{id}")]
public IActionResult UpdateCompany(Guid id, [FromBody] CompanyForUpdateDto company)
{
    if(company == null)
    {
        _logger.LogError("CompanyForUpdateDto object sent from client is null.");
        return BadRequest("CompanyForUpdateDto object is null");
    }

    var companyEntity = _repository.Company.GetCompany(id, trackChanges: true);
    if(companyEntity == null)
    {
        _logger.LogInfo($"Company with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    _mapper.Map(company, companyEntity);
    _repository.Save();

    return NoContent();
}
```

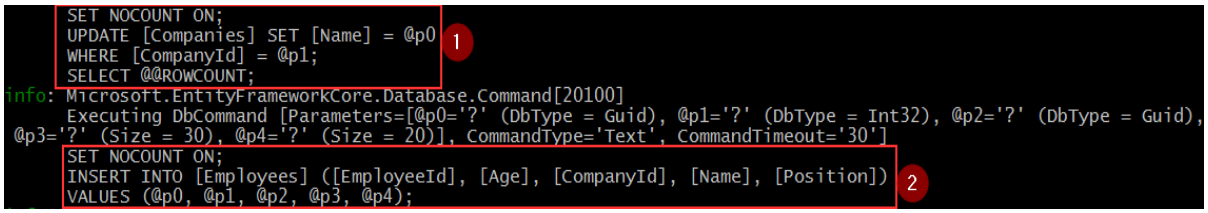
That's it. You can see that this action is almost the same as the employee update action.

Let's test this now:



We modify the name of the company and attach an employee as well. As a result, we can see **204**, which means that the entity has been updated. But what about that new employee?

Let's inspect our query:



You can see that we have created the **employee** entity in the database. So, EF Core does that job for us because we track the **company** entity. As soon as mapping occurs, EF Core sets the state for the **company** entity to **modified** and for all the employees to **added**. After we call the **Save** method, the **Name** property is going to be modified and the **employee** entity is going to be created in the database.

We are finished with the PUT requests, so let's continue with PATCH.

WORKING WITH PATCH REQUESTS

In the previous chapter, we worked with the PUT request to fully update our resource. But if we want to update our resource only partially, we should use PATCH.

The partial update isn't the only difference between PATCH and PUT. The request body is different as well. For the Company PATCH request, for example, we should use `[FromBody]JsonPatchDocument<Company>` and not `[FromBody]Company` as we did with the PUT requests.

Additionally, for the PUT request's media type, we have used `application/json` — but for the PATCH request's media type, we should use `application/json-patch+json`. Even though the first one would be accepted in ASP.NET Core for the PATCH request, the recommendation by REST standards is to use the second one.

Let's see what the PATCH request body looks like:

```
[
  {
    "op": "replace",
    "path": "/name",
    "value":  "new name"
  },
  {
    "op": "remove",
    "path": "/name"
  }
]
```

The square brackets represent an array of operations. Every operation is placed between curly brackets. So, in this specific example, we have two operations: Replace and Remove represented by the `op` property. The `path` property represents the object's property that we want to modify and the `value` property represents a new value.

In this specific example, for the first operation, we `replace` the value of the `name` property to a `new name`. In the second example, we `remove` the `name` property, thus setting its value to default.

There are six different operations for a PATCH request:

OPERATION	REQUEST BODY	EXPLANATION
Add	`{ "op": "add", "path": "/name", "value": "new value" }`	Assigns a new value to a required property.
Remove	`{ "op": "remove", "path": "/name" }`	Sets a default value to a required property.
Replace	`{ "op": "replace", "path": "/name", "value": "new value" }`	Replaces a value of a required property to a new value.
Copy	`{ "op": "copy", "from": "/name", "path": "/title" }`	Copies the value from a property in the “from” part to the property in the “path” part.
Move	`{ "op": "move", "from": "/name", "path": "/title" }`	Moves the value from a property in the “from” part to a property in the “path” part.
Test	`{ "op": "test", "path": "/name", "value": "new value" }`	Tests if a property has a specified value.

After all this theory, we are ready to dive into the coding part.

Applying PATCH to the Employee Entity

Before we start with the controller modification, we have to install two required libraries:

- The `Microsoft.AspNetCore.JsonPatch` library to support the usage of `JsonPatchDocument` in our controller and

- The `Microsoft.AspNetCore.Mvc.NewtonsoftJson` library to support request body conversion to a `PatchDocument` once we send our request.

Once the installation is completed, we have to add the `NewtonsoftJson` configuration to `IServiceCollection`:

```
services.AddControllers(config =>
{
    config.RespectBrowserAcceptHeader = true;
    config.ReturnHttpNotAcceptable = true;
    config.OutputFormatters.Add(new CsvOutputFormatter());
}).AddNewtonsoftJson()
  .AddXmlDataContractSerializerFormatters()
  .AddCustomCSVFormatter();
```

Add it before the Xml and CSV formatters. Now we can continue.

We will require a mapping from the `Employee` type to the `EmployeeForUpdateDto` type. Therefore, we have to create a mapping rule for that.

If we take a look at the `MappingProfile` class, we will see that we have a mapping from the `EmployeeForUpdateDto` to the `Employee` type:

```
CreateMap<EmployeeForUpdateDto, Employee>();
```

But we need it another way. To do so, we are not going to create an additional rule; we can just use the `ReverseMap` method to help us in the process:

```
CreateMap<EmployeeForUpdateDto, Employee>().ReverseMap();
```

The `ReverseMap` method is also going to configure this rule to execute reverse mapping if we ask for it.

Now, we can modify our controller:

```
[HttpPatch("{id}")]
public IActionResult PartiallyUpdateEmployeeForCompany(Guid companyId, Guid id,
[FromBody] JsonPatchDocument<EmployeeForUpdateDto> patchDoc)
{
    if(patchDoc == null)
```

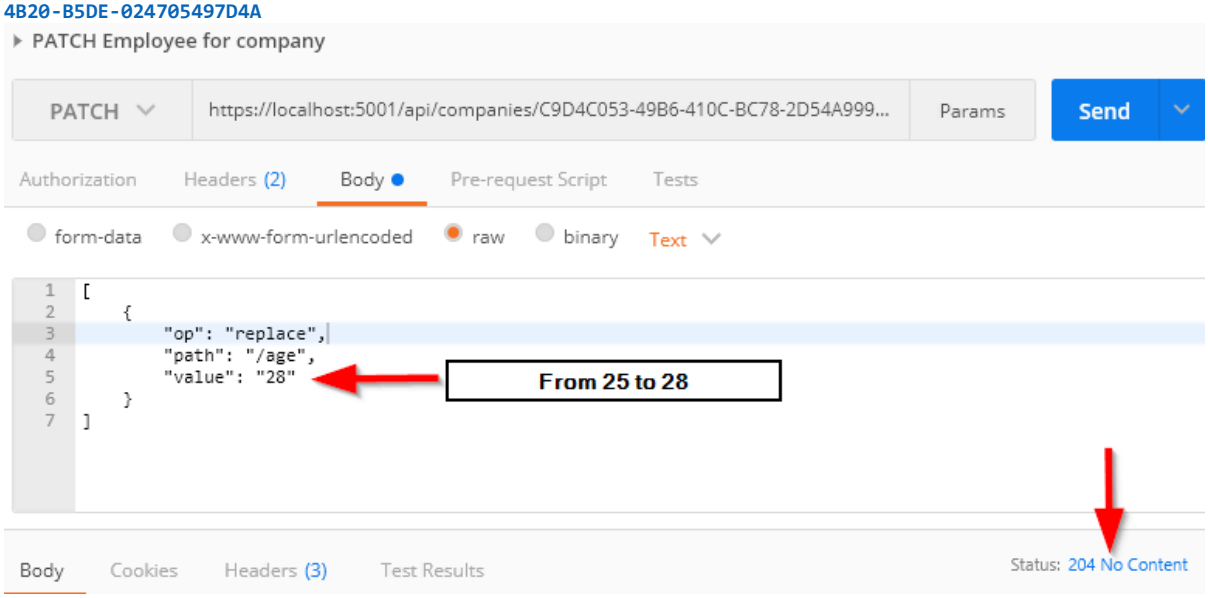
```
    {
        _logger.LogError("patchDoc object sent from client is null.");
        return BadRequest("patchDoc object is null");
    }

    var company = _repository.Company.GetCompany(companyId, trackChanges: false);
    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
    }

    var employeeEntity = _repository.Employee.GetEmployee(companyId, id, trackChanges:
true);
    if (employeeEntity == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    var employeeToPatch = _mapper.Map<EmployeeForUpdateDto>(employeeEntity);

    patchDoc.ApplyTo(employeeToPatch);

    _mapper.Map(employeeToPatch, employeeEntity);

    _repository.Save();

    return NoContent();
}
```

You can see that our action signature is different from the PUT actions. We are accepting the **JsonPatchDocument** from the request body. After that, we have a familiar code where we check the **patchDoc** for null value and if the company and employee exist in the database. Then, we map from the **Employee** type to the **EmployeeForUpdateDto** type; it is important for us to do that because the **patchDoc** variable can apply only to the **EmployeeForUpdateDto** type. After apply is executed, we map again to the **Employee** type (from **employeeToPatch** to **employeeEntity**) and save changes in the database.

Now, we can send a couple of requests to test this code:

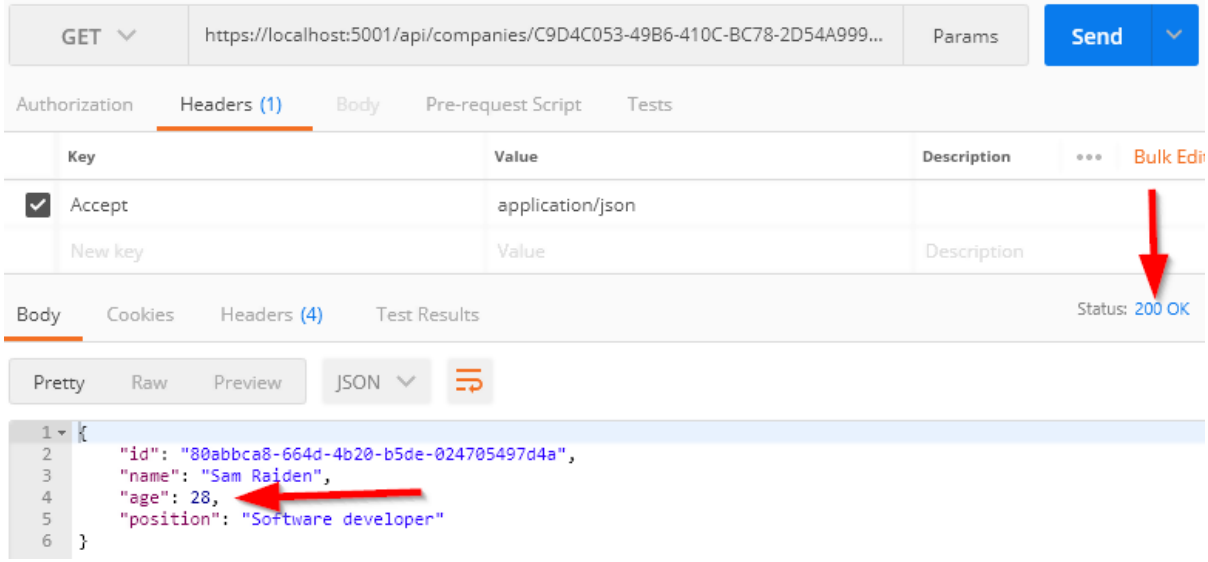
Let’s first send the replace operation:

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



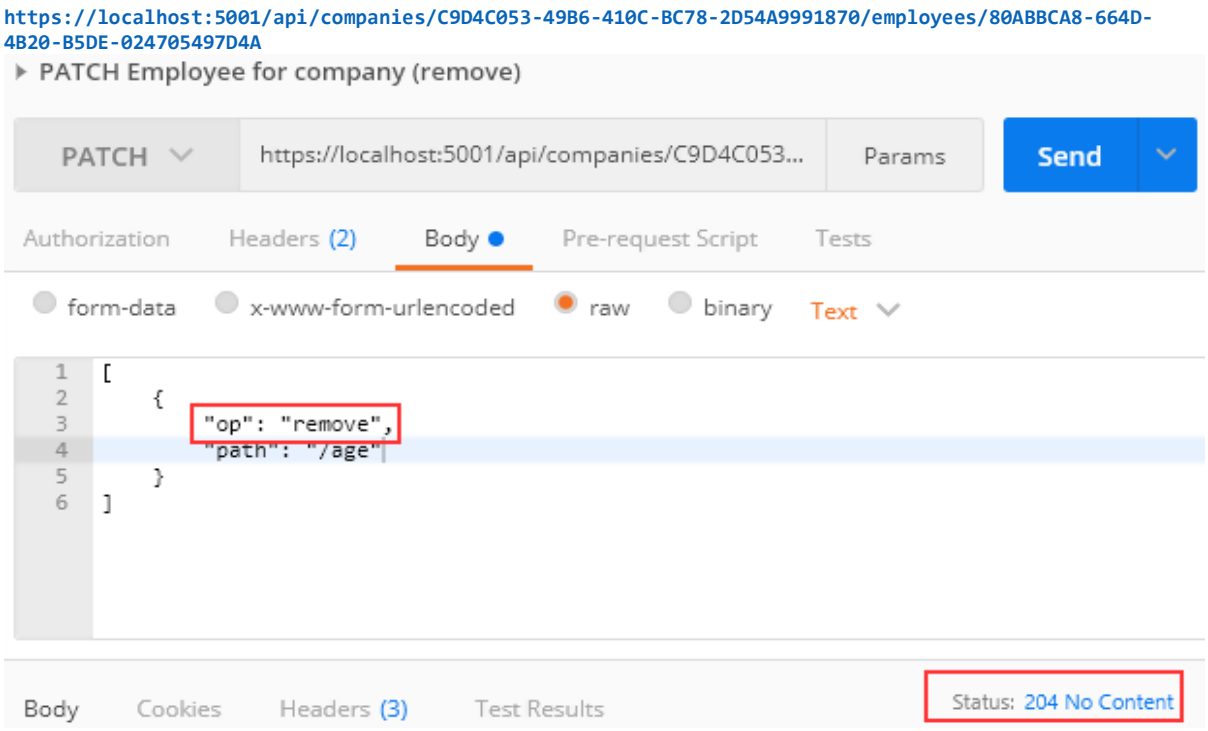
It works; we get the 204 No Content message. Let's check the same employee:

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



And we see that the `Age` property has been changed.

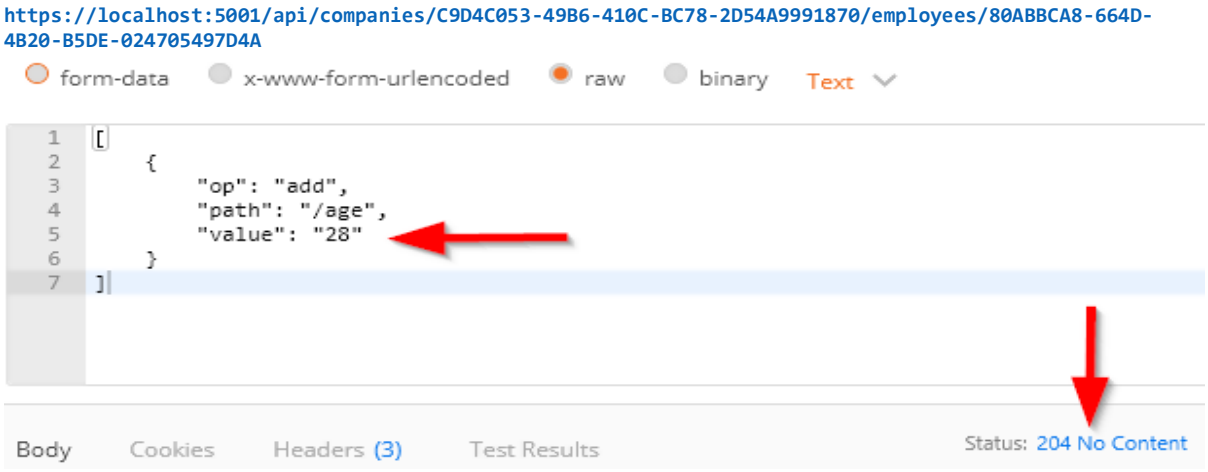
Let's send a remove operation in a request:



This works as well. Now, if we check our employee, its age is going to be set to 0 (the default value for the int type):

```
{
    "id": "80abbca8-664d-4b20-b5de-024705497d4a",
    "name": "Sam Raiden",
    "age": 0,
    "position": "Software developer"
}
```

Finally, let's return a value of 28 for the Age property:



Let’s check the employee now:


```
{
    "id": "80abbca8-664d-4b20-b5de-024705497d4a",
    "name": "Sam Raiden",
    "age": 28,
    "position": "Software developer"
}
```


Excellent.

Everything is working well.

VALIDATION

While writing API actions, we have a set of rules that we need to check. If we take a look at the `Company` class, we can see different data annotation attributes above our properties:

```
public class Company
{
    [Column("CompanyId")]
    5 references
    public Guid Id { get; set; }

    [Required(ErrorMessage = "Company name is a required field.")]
    [MaxLength(60, ErrorMessage = "Maximum length for the Name is 60 characters.")]
    3 references
    public string Name { get; set; }

    [Required(ErrorMessage = "Company address is a required field.")]
    [MaxLength(60, ErrorMessage = "Maximum length for the Address is 60 characters.")]
    3 references
    public string Address { get; set; }

    3 references
    public string Country { get; set; }

    2 references
    public ICollection<Employee> Employees { get; set; }
}
```

Those attributes serve the purpose to validate our model object while creating or updating resources in the database. But we are not making use of them yet.

In this chapter, we are going to show you how to validate our model objects and how to return an appropriate response to the client if the model is not valid. So, we need to validate the input and not the output of our controller actions. This means that we are going to apply this validation to the POST, PUT, and PATCH requests, but not for the GET request.

To validate against validation rules applied by Data Annotation attributes, we are going to use the concept of `ModelState`. It is a dictionary containing the state of the model and model binding validation.

Once we send our request, the rules defined by Data Annotation attributes are checked. If one of the rules doesn't check out, the

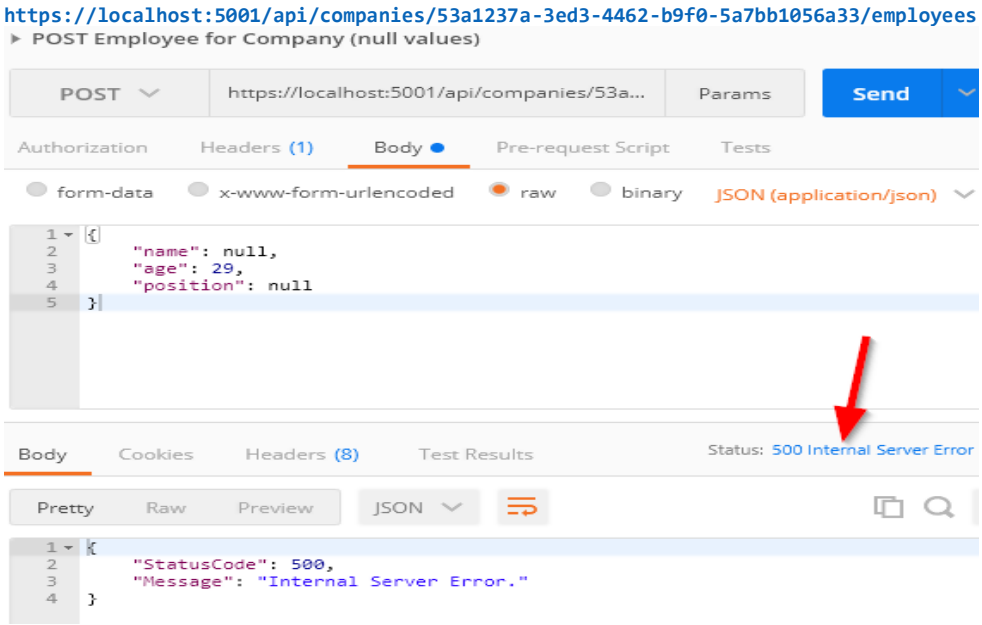
appropriate error message will be returned. We are going to use the `ModelState.IsValid` expression to check for those validation rules.

Finally, the response status code, when validation fails, should be `422 Unprocessable Entity`. That means that the server understood the content type of the request and the syntax of the request entity is correct, but it was unable to process validation rules applied on the entity inside the request body.

So, with all this in mind, we are ready to implement model validation in our code.

Validation while Creating Resource

Let's send another request for the `CreateEmployee` action, but this time with the invalid request body:

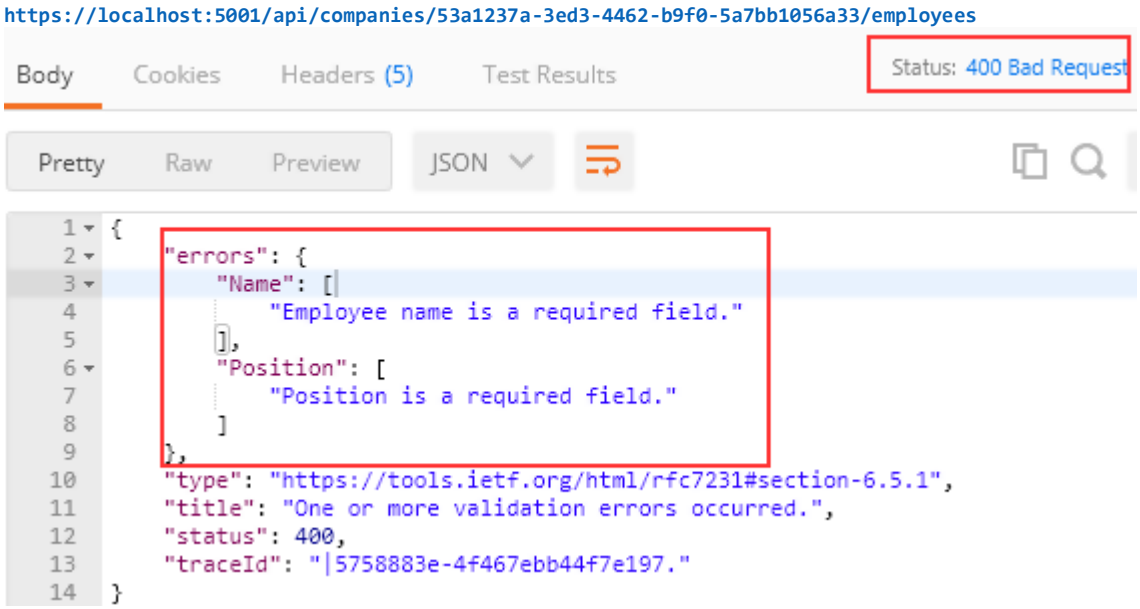


And we get the `500 Internal Server Error`, which is a generic message when something unhandled happens in our code. But this is not good. This means that the server made an error, which is not the case. In this case, we, as a consumer, sent a wrong model to the API — thus the error message should be different.

In order to fix this, let's modify our `EmployeeForCreationDto` class because that's what we deserialize the request body to:

```
public class EmployeeForCreationDto
{
    [Required(ErrorMessage = "Employee name is a required field.")]
    [MaxLength(30, ErrorMessage = "Maximum length for the Name is 30 characters.")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Age is a required field.")]
    public int Age { get; set; }

    [Required(ErrorMessage = "Position is a required field.")]
    [MaxLength(20, ErrorMessage = "Maximum length for the Position is 20
characters.")]
    public string Position { get; set; }
}
```

Once we have the rules applied, we can send the same request again:



You can see that our validation rules have been applied and verified as well. ASP.NET Core validates the model object as soon as the request gets to the action.

But the status code for this response is 400 Bad Request. That is acceptable, but as we said, there is a status code that better fits this kind of situation. It is 422 Unprocessable Entity.

To return 422 instead of 400, the first thing we have to do is to suppress the `BadRequest` error when the `ModelState` is invalid. We are going to do that by adding this code into the `Startup` class in the `ConfigureServices` method:

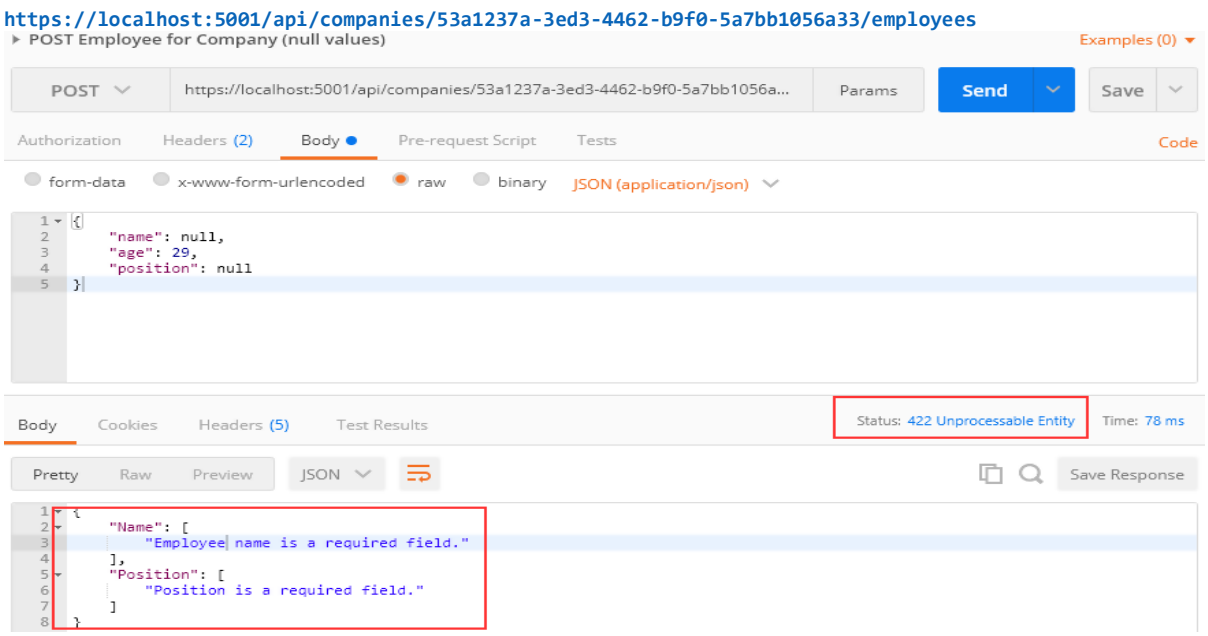
```
services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressModelStateInvalidFilter = true;
});
```

Then, we have to modify our action:

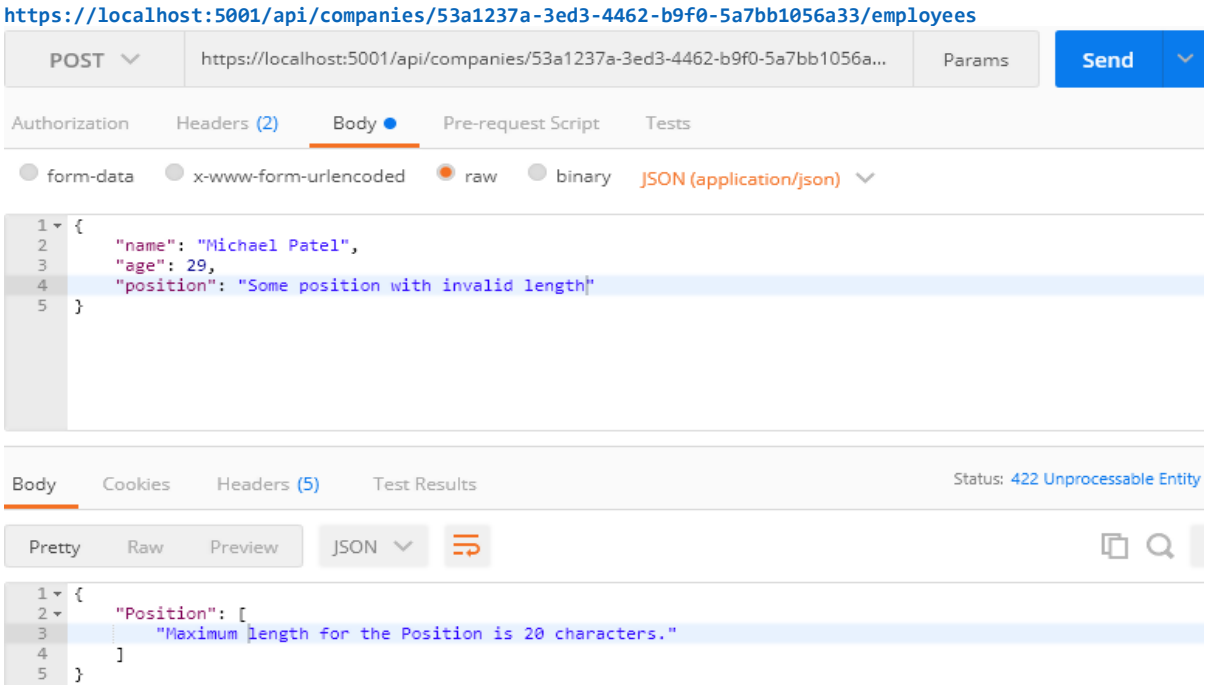
```
[HttpPost]
public IActionResult CreateEmployeeForCompany(Guid companyId, [FromBody]
EmployeeForCreationDto employee)
{
    if(employee == null)
    {
        _logger.LogError("EmployeeForCreationDto object sent from client is null.");
        return BadRequest("EmployeeForCreationDto object is null");
    }

    if(!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the EmployeeForCreationDto object");
        return UnprocessableEntity(ModelState);
    }

    … the rest of the code …

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

And that is all.

Let’s send our request one more time:



Let’s send an additional request to test the max length rule:

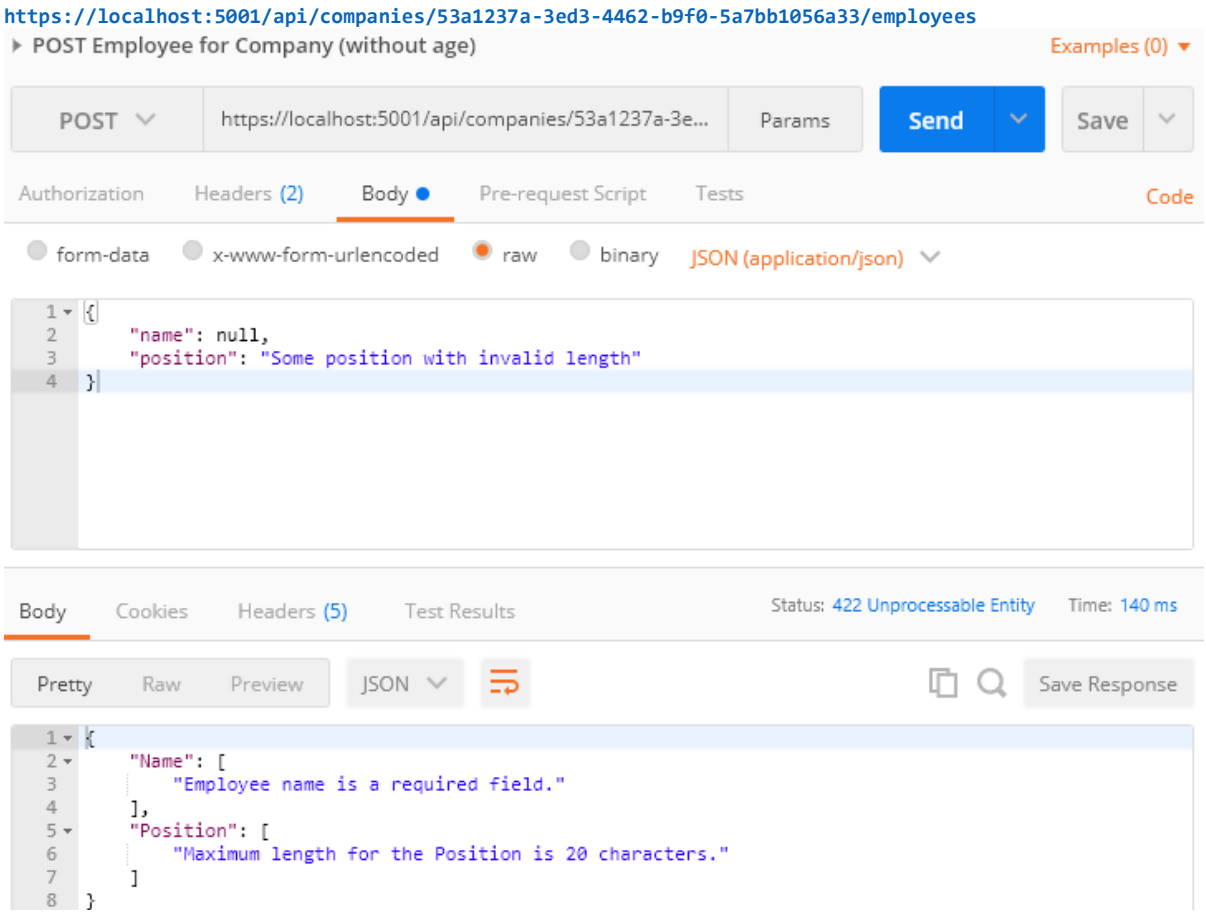


Excellent. It is working as expected.

The same actions can be applied for the `CreateCompany` action and `CompanyForCreationDto` class — and if you check the source code for this chapter, you will find it implemented.

Validating Int Type

Let's create one more request with the request body without the `age` property:



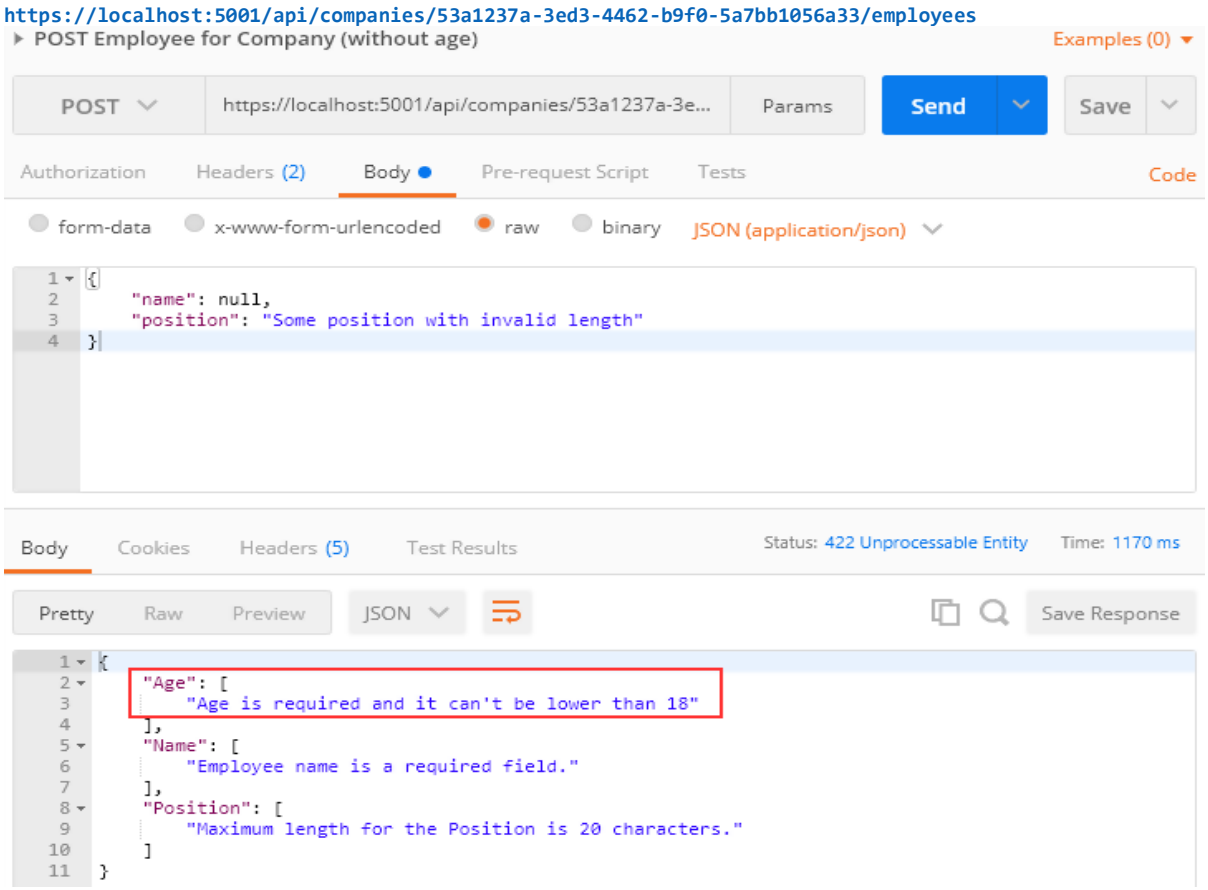
We can clearly see that the `age` property hasn't been sent, but in the response body, we don't see the error message for the `age` property next to other error messages. That is because the age is of type int and if we don't send that property, it would be set to a default value, which is 0.

So, on the server side, validation for the `Age` property will pass, because it is not null.

In order to prevent this type of behavior, we have to modify the data annotation attribute on top of the `Age` property in the `EmployeeForCreationDto` class:

```
[Range(18, int.MaxValue, ErrorMessage = "Age is required and it can't be lower than 18")]
public int Age { get; set; }
```

Now, let's try to send the same request one more time:



Now, we have the `Age` error message in our response.

If we want, we can add the custom error messages in our action:

```
ModelState.AddModelError(string key, string errorMessage)
```

With this expression, the additional error message will be included with all the other messages.

Validation for PUT Requests

The validation for PUT requests shouldn't be different from POST requests (except in some cases), but there are still things we have to do to at least optimize our code.

But let's go step by step.

First, let's add Data Annotation Attributes to the **EmployeeForUpdateDto** class:

```
public class EmployeeForUpdateDto
{
   [Required(ErrorMessage = "Employee name is a required field.")]
   [MaxLength(30, ErrorMessage = "Maximum length for the Name is 30 characters.")]
   public string Name { get; set; }

   [Range(18, int.MaxValue, ErrorMessage = "Age is required and it can't be lower than
18")]
   public int Age { get; set; }

   [Required(ErrorMessage = "Position is a required field.")]
   [MaxLength(20, ErrorMessage = "Maximum length for the Position is 20 characters.")]
   public string Position { get; set; }
}
```

Once we have done this, we realize we have a small problem. If we compare this class with the DTO class for creation, we are going to see that they are the same. Of course, we don't want to repeat ourselves, thus we are going to add some modifications.

Let's create a new class in the **DataTransferObjects** folder:

```
public abstract class EmployeeForManipulationDto
{
    [Required(ErrorMessage = "Employee name is a required field.")]
    [MaxLength(30, ErrorMessage = "Maximum length for the Name is 30 characters.")]
    public string Name { get; set; }

    [Range(18, int.MaxValue, ErrorMessage = "Age is required and it can't be lower
than 18")]
    public int Age { get; set; }

    [Required(ErrorMessage = "Position is a required field.")]
    [MaxLength(20, ErrorMessage = "Maximum length for the Position is 20
characters.")]
    public string Position { get; set; }
}
```

We create this class as an abstract class because we want our creation and update DTO classes to inherit from it:

```
public class EmployeeForUpdateDto : EmployeeForManipulationDto
{
}
```

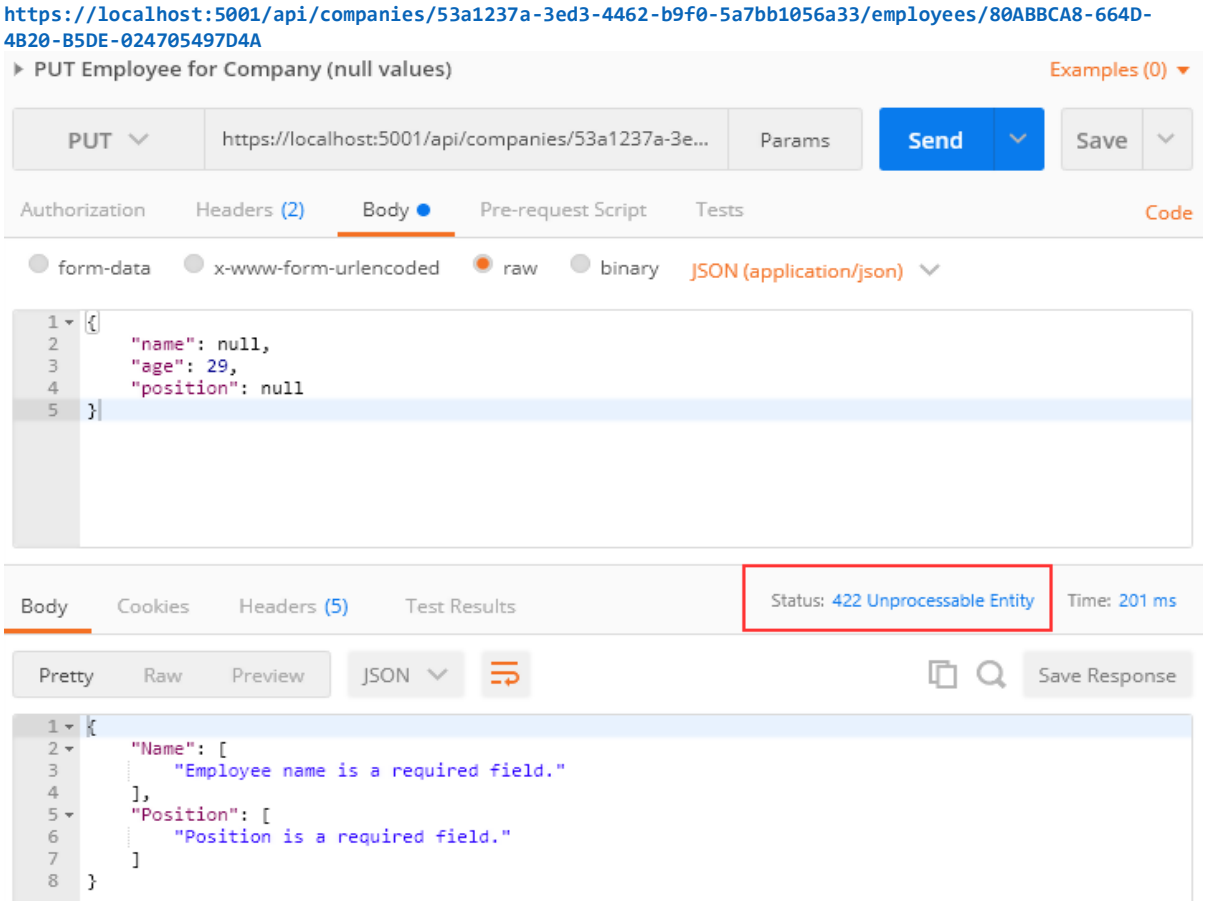
```
public class EmployeeForCreationDto : EmployeeForManipulationDto
{
}
```

Now, we can modify the **UpdateEmployeeForCompany** action by adding the model validation right after the null check:

```
if(employee == null)
{
    _logger.LogError("EmployeeForUpdateDto object sent from client is null.");
    return BadRequest("EmployeeForUpdateDto object is null");
}

if (!ModelState.IsValid)
{
    _logger.LogError("Invalid model state for the EmployeeForUpdateDto object");
    return UnprocessableEntity(ModelState);
}
```

The same process can be applied to the Company DTO classes and create action. You can find it implemented in the source code for this chapter.

Let's test this:



Great.

Everything works well.

Validation for PATCH Requests

The validation for PATCH requests is a bit different from the previous ones. We are using the ModelState concept again, but this time we have to place it in the `ApplyTo` method first:

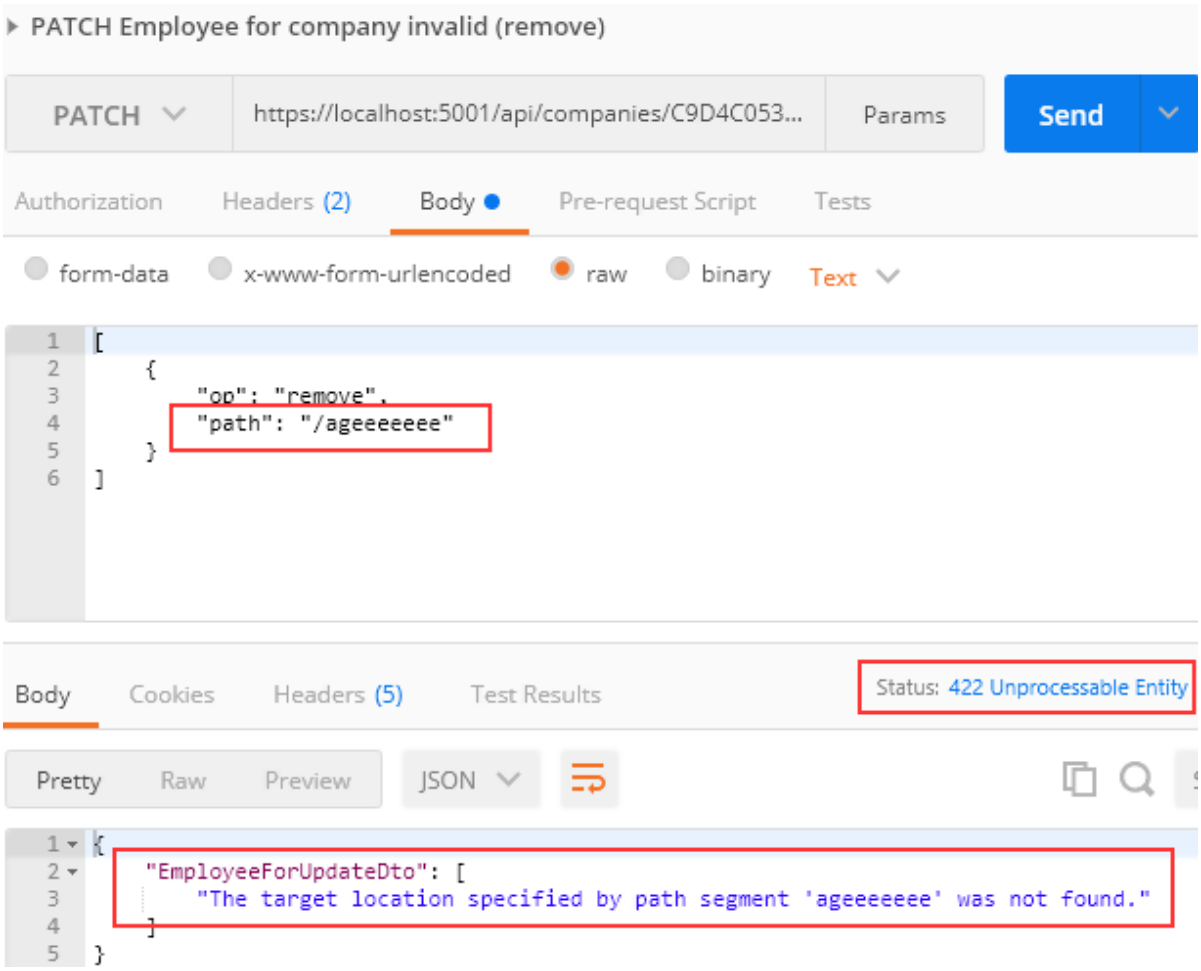
```
patchDoc.ApplyTo(employeeToPatch, ModelState);
```

Right below, we can add our familiar validation logic:

```
patchDoc.ApplyTo(employeeToPatch, ModelState);

if(!ModelState.IsValid)
{
    _logger.LogError("Invalid model state for the patch document");
    return UnprocessableEntity(ModelState);
}

_mapper.Map(employeeToPatch, employeeEntity);
```

Let's test this now:

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



You can see that it works as it supposed to.

But, we have a small problem now. What if we try to send a remove operation, but for the valid path:

We can see it passes, but this is not good. If you can remember, we said that the `remove` operation will set the value for the included property to its default value, which is 0. But in the `EmployeeForUpdateDto` class, we have a `Range` attribute which doesn't allow that value to be below 18. So, where is the problem?

Let's illustrate this for you:

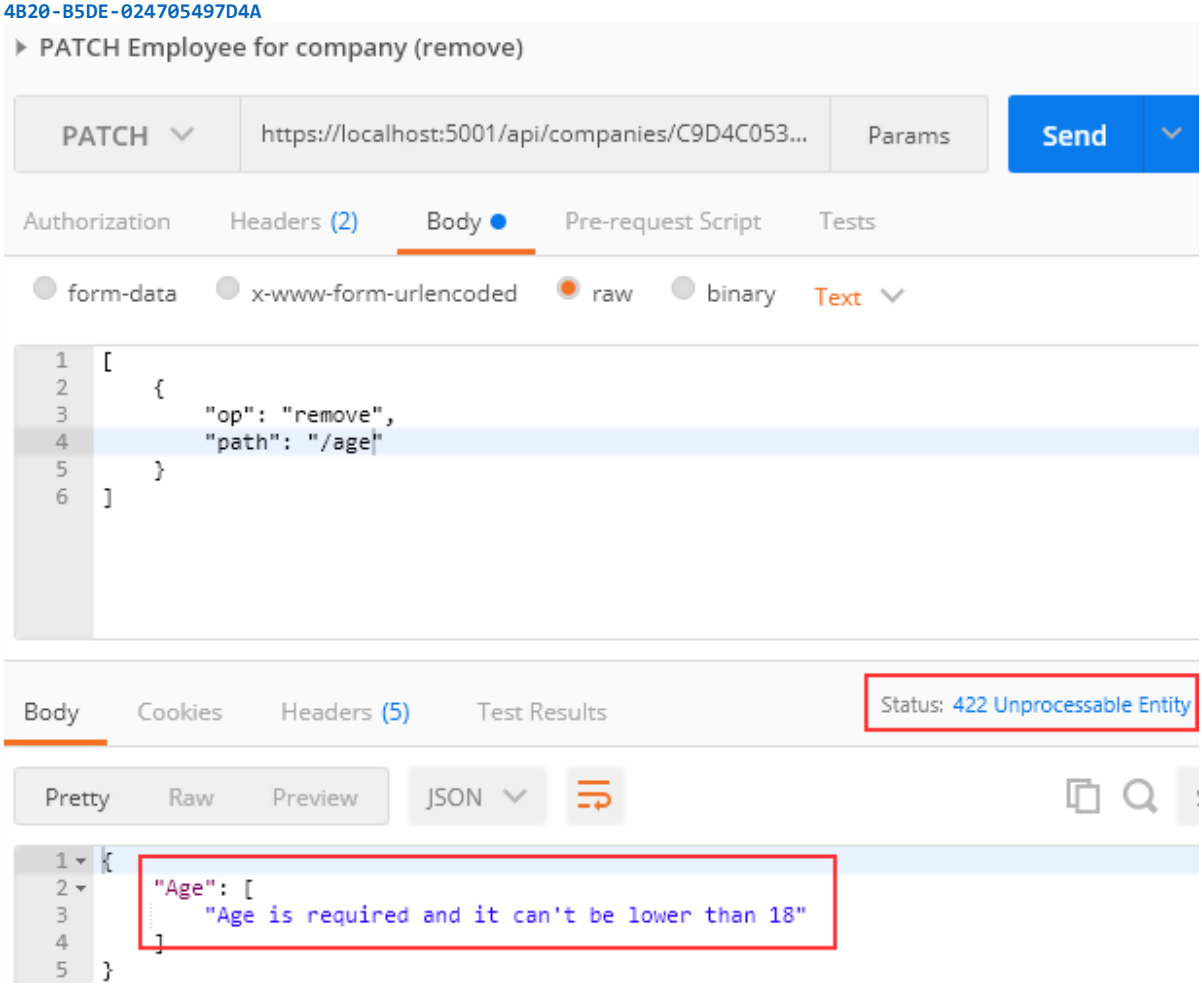
As you can see, we are validating the patchDoc which is completely valid at this moment, but we save employeeEntity to the database. So, we need some additional validation to prevent an invalid `employeeEntity` from being saved to the database:

```
var employeeToPatch = _mapper.Map<EmployeeForUpdateDto>(employeeEntity);

patchDoc.ApplyTo(employeeToPatch, ModelState);

TryValidateModel(employeeToPatch);

if(!ModelState.IsValid)
{
    _logger.LogError("Invalid model state for the patch document");
    return UnprocessableEntity(ModelState);
}
```

We can use the `TryValidateModel` method to validate the already patched `employeeToPatch` instance. This will trigger a validation and every error will make ModelState invalid. After that, we execute a familiar validation check.

Now, we can test this again:

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



And we get 422, which is the expected status code.

ASYNCHRONOUS CODE

In this chapter, we are going to convert synchronous code to asynchronous inside ASP.NET Core. First, we are going to learn a bit about asynchronous programming and why should we write async code. Then we are going to use our code from the previous chapters and rewrite it in an async manner.

We are going to modify the code, step by step, to show you how easy is to convert synchronous code to asynchronous code. Hopefully, this will help you understand how asynchronous code works and how to write it from scratch in your applications.

What is Asynchronous Programming?

Async programming is a parallel programming technique which allows the working process to run separately from the main application thread. As soon as the work completes, it informs the main thread about the result whether it was successful or not.

By using async programming, we can avoid performance bottlenecks and enhance the responsiveness of our application.

How so?

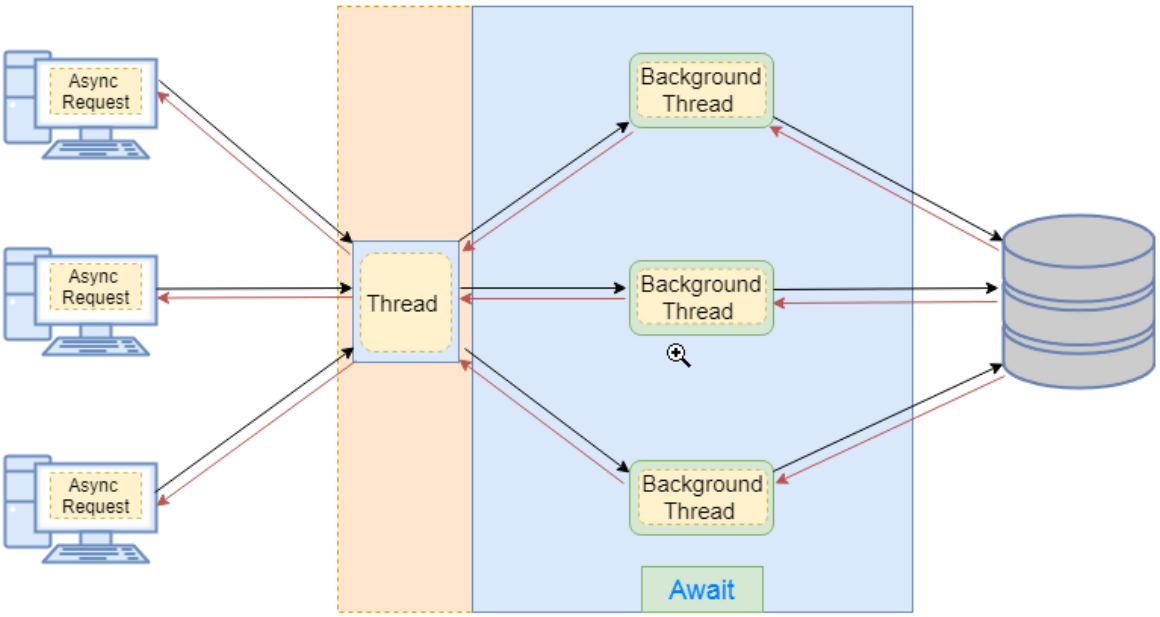
Because we are not sending requests to the main thread and blocking it while waiting for the responses anymore (as long as it takes). Now, when we send a request to the main thread, it delegates a job to a background thread — thus freeing itself for another request. Eventually, a background thread finishes its job and returns it to the main thread. Then the main thread returns the result to the requester.

It is very important to understand that if we send a request to an endpoint and it takes the application three or more seconds to process

that request, we probably won't be able to execute this request any faster in async mode. It is going to take the same amount of time as the sync request.

The only advantage is that in async mode the main thread won't be blocked three or more seconds; thus, it will be able to process other requests.

Here is a visual representation of asynchronous workflow:



Now that we've cleared that out, we can learn how to implement asynchronous code in .NET Core.

Async, Await Keywords, and Return Types

The `async` and `await` keywords play a crucial part in asynchronous programming. By using those keywords, we can easily write asynchronous methods without too much effort.

For example, if we want to create a method in an asynchronous manner, we need to add the `async` keyword next to the method's return type:

```
async Task<IEnumerable<Company>> GetAllCompaniesAsync()
```

By using the `async` keyword, we are enabling the `await` keyword and modifying how method results are handled (from synchronous to asynchronous):

```
await FindAllAsync();
```

In asynchronous programming, we have three return types:

- Task<TResult>, for an async method that returns a value.
- Task, for an async method that does not return a value.
- void, which we can use for an event handler.

What does this mean?

Well, we can look at this through synchronous programming glasses. If our sync method returns an int, then in the async mode it should return `Task<int>` — or if the sync method returns `IEnumerable<string>`, then the async method should return `Task<IEnumerable<string>>`.

But if our sync method returns no value (has a void for the return type), then our async method should return `Task`. This means that we can use the `await` keyword inside that method, but without the return keyword.

You may wonder now, why not return Task all the time? Well, we should use void only for the asynchronous event handlers which require a void return type. Other than that, we should always return a Task.

From C# 7.0 onward, we can specify any other return type if that type includes a GetAwaiter method.

Now, when we have all the information, let's do some refactoring in our completely synchronous code.

The IRepositoryBase Interface and the RepositoryBase Class Explanation

We won't be changing the mentioned interface and class. That's because we want to leave a possibility for the repository user classes to have either sync or async method execution. Sometimes, the async code could become slower than the sync one because EF Core's async commands take slightly longer to execute (due to extra code for handling the threading), so leaving this option is always a good choice.

It is general advice to use async code wherever it is possible, but if we notice that our async code runes slower, we should switch back to the sync one.

Modifying the ICompanyRepository Interface and the CompanyRepository Class

In the `Contracts` project, we can find the `ICompanyRepository` interface with all the synchronous method signatures which we should change.

So, let's do that:

```
public interface ICompanyRepository
{
    Task<IEnumerable<Company>> GetAllCompaniesAsync(bool trackChanges);
    Task<Company> GetCompanyAsync(Guid companyId, bool trackChanges);
    void CreateCompany(Company company);
    Task<IEnumerable<Company>> GetByIdsAsync(IEnumerable<Guid> ids, bool
trackChanges);
    void DeleteCompany(Company company);
}
```

The `Create` and `Delete` method signatures are left synchronous. That's because in these methods, we are not making any changes in the database. All we're doing is changing the state of the entity to Added and Deleted.

So, in accordance with the interface changes, let's modify our `CompanyRepository.cs` class, which we can find in the `Repository` project:

```
public async Task<IEnumerable<Company>> GetAllCompaniesAsync(bool trackChanges) =>
    await FindAll(trackChanges)
    .OrderBy(c => c.Name)
    .ToListAsync();

public async Task<Company> GetCompanyAsync(Guid companyId, bool trackChanges) =>
    await FindByCondition(c => c.Id.Equals(companyId), trackChanges)
    .SingleOrDefaultAsync();

public async Task<IEnumerable<Company>> GetByIdsAsync(IEnumerable<Guid> ids, bool
trackChanges) =>
    await FindByCondition(x => ids.Contains(x.Id), trackChanges)
    .ToListAsync();
```

We only have to change these methods in our repository class.

IRepositoryManager and RepositoryManager Changes

If we inspect the mentioned interface and the class, we will see the `Save` method, which just calls the EF Core's `SaveChanges` method. We have to change that as well:

```
public interface IRepositoryManager
{
    ICompanyRepository Company { get; }
    IEmployeeRepository Employee { get; }
    Task SaveAsync();
}
```

And class modification:

```
public Task SaveAsync() => _repositoryContext.SaveChangesAsync();
```

Because the `SaveAsync(), ToListAsync()...` methods are awaitable, we may use the `await` keyword; thus, our methods need to have the `async` keyword and `Task` as a return type.

Using the `await` keyword is not mandatory, though. Of course, if we don't use it, our `SaveAsync()` method will execute synchronously — and that is not our goal here.

Controller Modification

Finally, we need to modify all of our actions in the `CompaniesController` to work asynchronously.

So, let's first start with the `GetCompanies` method:

```
[HttpGet]
public async Task<IActionResult> GetCompanies()
{
    var companies = await _repository.Company.GetAllCompaniesAsync(trackChanges:
false);

    var companiesDto = _mapper.Map<IEnumerable<CompanyDto>>(companies);

    return Ok(companiesDto);
}
```

We haven't changed much in this action. We've just changed the return type and added the `async` keyword to the method signature. In the method body, we can now await the `GetAllCompaniesAsync()` method. And that is pretty much what we should do in all the actions in our controller.

So, let's modify all the other actions.

GetCompany:

```
[HttpGet("{id}", Name = "CompanyById")]
public async Task<IActionResult> GetCompany(Guid id)
{
    var company = await _repository.Company.GetCompanyAsync(id, trackChanges: false);
    if (company == null)
    {
        _logger.LogInfo($"Company with id: {id} doesn't exist in the database.");
        return NotFound();
    }
    else
    {
        var companyDto = _mapper.Map<CompanyDto>(company);
        return Ok(companyDto);
    }
}
```

GetCompanyCollection:

```
[HttpGet("collection/({ids})", Name = "CompanyCollection")]
```

```
public async Task<IActionResult> GetCompanyCollection([ModelBinder(BinderType =
typeof(ArrayModelBinder))]IEnumerable<Guid> ids)
{
    if(ids == null)
    {
        _logger.LogError("Parameter ids is null");
        return BadRequest("Parameter ids is null");
    }

    var companyEntities = await _repository.Company.GetByIdsAsync(ids, trackChanges:
false);

    if(ids.Count() != companyEntities.Count())
    {
        _logger.LogError("Some ids are not valid in a collection");
        return NotFound();
    }

    var companiesToReturn = _mapper.Map<IEnumerable<CompanyDto>>(companyEntities);
    return Ok(companiesToReturn);
}
```

CreateCompany:

```
[HttpPost]
public async Task<IActionResult> CreateCompany([FromBody]CompanyForCreationDto
company)
{
    if(company == null)
    {
        _logger.LogError("CompanyForCreationDto object sent from client is null.");
        return BadRequest("CompanyForCreationDto object is null");
    }

    if (!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the CompanyForCreationDto object");
        return UnprocessableEntity(ModelState);
    }

    var companyEntity = _mapper.Map<Company>(company);

    _repository.Company.CreateCompany(companyEntity);
    await _repository.SaveAsync();

    var companyToReturn = _mapper.Map<CompanyDto>(companyEntity);

    return CreatedAtRoute("CompanyById", new { id = companyToReturn.Id },
companyToReturn);
}
```

CreateCompanyCollection:

```
[HttpPost("collection")]
public async Task<IActionResult> CreateCompanyCollection([FromBody]
IEnumerable<CompanyForCreationDto> companyCollection)
```

```
{
    if(companyCollection == null)
    {
        _logger.LogError("Company collection sent from client is null.");
        return BadRequest("Company collection is null");
    }

    var companyEntities = _mapper.Map<IEnumerable<Company>>(companyCollection);
    foreach (var company in companyEntities)
    {
        _repository.Company.CreateCompany(company);
    }

    await _repository.SaveAsync();

    var companyCollectionToReturn =
        _mapper.Map<IEnumerable<CompanyDto>>(companyEntities);
    var ids = string.Join(",", companyCollectionToReturn.Select(c => c.Id));

    return CreatedAtRoute("CompanyCollection", new { ids },
companyCollectionToReturn);
}
```

DeleteCompany:

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteCompany(Guid id)
{
    var company = await _repository.Company.GetCompanyAsync(id, trackChanges: false);
    if(company == null)
    {
        _logger.LogInfo($"Company with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    _repository.Company.DeleteCompany(company);
    await _repository.SaveAsync();

    return NoContent();
}
```

UpdateCompany:

```
[HttpPut("{id}")]
public async Task<IActionResult> UpdateCompany(Guid id, [FromBody] CompanyForUpdateDto
company)
{
    if(company == null)
    {
        _logger.LogError("CompanyForUpdateDto object sent from client is null.");
        return BadRequest("CompanyForUpdateDto object is null");
    }

    if(!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the CompanyForUpdateDto object");
```

```
        return UnprocessableEntity(ModelState);
    }

    var companyEntity = await _repository.Company.GetCompanyAsync(id, trackChanges:
true);
    if(companyEntity == null)
    {
        _logger.LogInfo($"Company with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    _mapper.Map(company, companyEntity);
    await _repository.SaveAsync();

    return NoContent();
}
```

Excellent. Now we are talking async.

Of course, we have the **Employee** entity as well and all of these steps have to be implemented for the **EmployeeRepository** class, **IEmployeeRepository** interface, and **EmployeesController**.

You can always refer to the source code for this chapter if you have any trouble implementing async code for the **Employee** entity.

After the async implementation in the Employee classes, you can try to send different requests (from any chapter) to test your async actions. All of them should work as before, without errors, but this time in an asynchronous manner.

ACTION FILTERS

Filters in .NET offer a great way to hook into the MVC action invocation pipeline. Therefore, we can use filters to extract code which can be reused and make our actions cleaner and maintainable. Some filters are already provided by .NET like the authorization filter, and there are the custom ones that we can create ourselves.

There are different filter types:

- **Authorization filters** – They run first to determine whether a user is authorized for the current request.
- **Resource filters** – They run right after the authorization filters and are very useful for caching and performance.
- **Action filters** – They run right before and after action method execution.
- **Exception filters** – They are used to handle exceptions before the response body is populated.
- **Result filters** – They run before and after the execution of the action methods result.

In this chapter, we are going to talk about Action filters and how to use them to create cleaner and reusable code in our Web API.

Action Filters Implementation

To create an Action filter, we need to create a class that inherits either from the `IActionFilter` interface, the `IAsyncActionFilter` interface, or the `ActionFilterAttribute` class — which is the implementation of `IActionFilter`, `IAsyncActionFilter`, and a few different interfaces as well:

```
public abstract class ActionFilterAttribute : Attribute, IActionFilter,
IFilterMetadata, IAsyncActionFilter, IResultFilter, IAsyncResultFilter, IOrderedFilter
```

To implement the synchronous Action filter that runs before and after action method execution, we need to implement the `OnActionExecuting` and `OnActionExecuted` methods:

```
namespace ActionFilters.Filters
{
    public class ActionFilterExample : IActionFilter
    {
        public void OnActionExecuting(ActionExecutingContext context)
        {
            // our code before action executes
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            // our code after action executes
        }
    }
}
```

We can do the same thing with an asynchronous filter by inheriting from `IAsyncActionFilter`, but we only have one method to implement — the `OnActionExecutionAsync`:

```
namespace ActionFilters.Filters
{
    public class AsyncActionFilterExample : IAsyncActionFilter
    {
        public async Task OnActionExecutionAsync(ActionExecutingContext context,
        ActionExecutionDelegate next)
        {
            // execute any code before the action executes
            var result = await next();
            // execute any code after the action executes
        }
    }
}
```

The Scope of Action Filters

Like the other types of filters, the action filter can be added to different scope levels: Global, Action, and Controller.

If we want to use our filter globally, we need to register it inside the `AddControllers()` method in the `ConfigureServices` method:

```
services.AddControllers(config =>
{
    config.Filters.Add(new GlobalFilterExample());
});
```

But if we want to use our filter as a service type on the Action or Controller level, we need to register it in the same `ConfigureServices` method, but as a service in the IoC container:

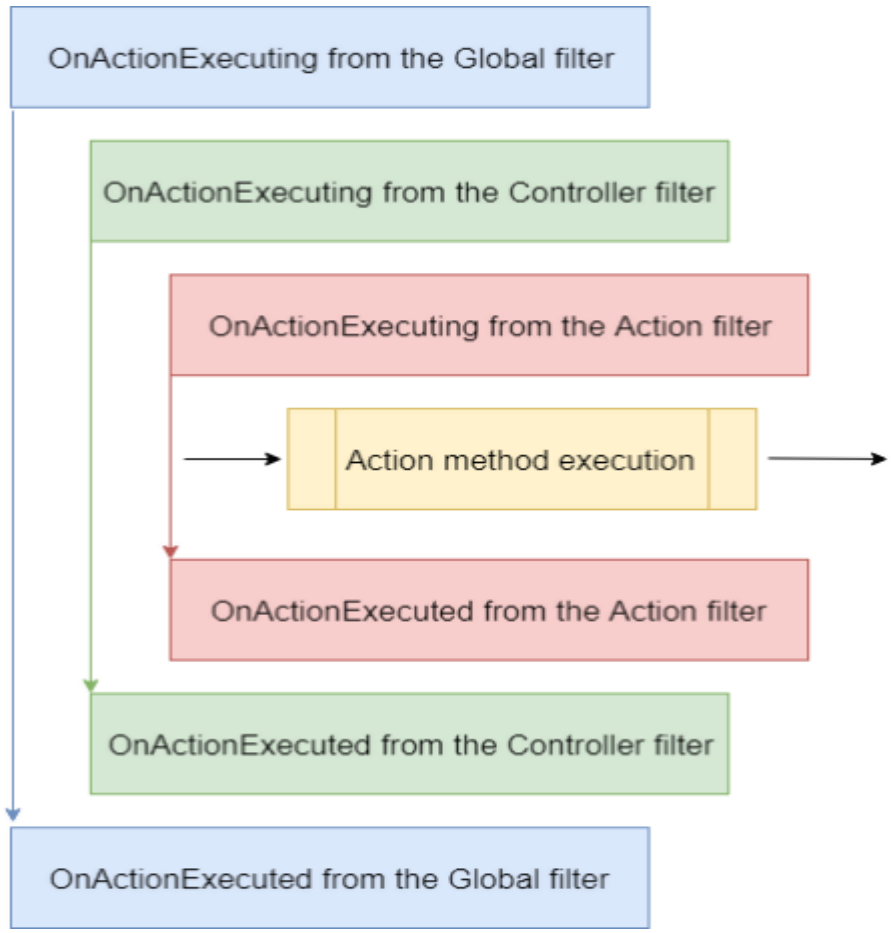
```
services.AddScoped<ActionFilterExample>();
services.AddScoped<ControllerFilterExample>();
```

Finally, to use a filter registered on the Action or Controller level, we need to place it on top of the Controller or Action as a ServiceType:

```
namespace AspNetCore.Controllers
{
    [ServiceFilter(typeof(ControllerFilterExample))]
    [Route("api/[controller]")]
    [ApiController]
    public class TestController : ControllerBase
    {
        [HttpGet]
        [ServiceFilter(typeof(ActionFilterExample))]
        public IEnumerable<string> Get()
        {
            return new string[] { "example", "data" };
        }

    }
}
```

Order of Invocation

The order in which our filters are executed is as follows:



Of course, we can change the order of invocation by adding the `Order` property to the invocation statement:

```
namespace AspNetCore.Controllers
{
    [ServiceFilter(typeof(ControllerFilterExample), Order = 2)]
    [Route("api/[controller]")]
    [ApiController]
    public class TestController : ControllerBase
    {
        [HttpGet]
        [ServiceFilter(typeof(ActionFilterExample), Order = 1)]
        public IEnumerable<string> Get()
        {
            return new string[] { "example", "data" };
        }

    }
}
```

Or something like this on top of the same action:

```
[HttpGet]
[ServiceFilter(typeof(ActionFilterExample), Order = 2)]
[ServiceFilter(typeof(ActionFilterExample2), Order = 1)]
public IEnumerable<string> Get()
{
    return new string[] { "example", "data" };
}
```

Improving the Code with Action Filters

Our actions are clean and readable without `try-catch` blocks due to global exception handling, but we can improve them even further.

So, let's start with the validation code from the POST and PUT actions.

Validation with Action Filters

If we take a look at our POST and PUT actions, we can notice the repeated code in which we validate our `Company` model:

```
if(company == null)
{
    _logger.LogError("CompanyForCreationDto object sent from client is null.");
    return BadRequest("CompanyForCreationDto object is null");
}

if (!ModelState.IsValid)
{
    _logger.LogError("Invalid model state for the CompanyForCreationDto object");
    return UnprocessableEntity(ModelState);
}
```

We can extract that code into a custom Action Filter class, thus making this code reusable and the action cleaner.

So, let's do that.

Let's create a new folder in our solution explorer, and name it `ActionFilters`. Then inside that folder, we are going to create a new class `ValidationFilterAttribute`:

```
public class ValidationFilterAttribute : IActionFilter
{
    private readonly ILoggerManager _logger;
    public ValidationFilterAttribute(ILoggerManager logger)
    {
        _logger = logger;
    }
```

```
    public void OnActionExecuting(ActionExecutingContext context) { }

    public void OnActionExecuted(ActionExecutedContext context){}
}
```

Now we are going to modify the **OnActionExecuting** method:

```
public void OnActionExecuting(ActionExecutingContext context)
{
    var action = context.RouteData.Values["action"];
    var controller = context.RouteData.Values["controller"];

    var param = context.ActionArguments
        .SingleOrDefault(x => x.Value.ToString().Contains("Dto")).Value;
    if (param == null)
    {
        _logger.LogError($"Object sent from client is null. Controller: {controller},
action: {action}");
        context.Result = new BadRequestObjectResult($"Object is null. Controller:
{controller}, action: {action}");
        return;
    }

    if (!context.ModelState.IsValid)
    {
        _logger.LogError($"Invalid model state for the object. Controller:
{controller}, action: {action}");
        context.Result = new UnprocessableEntityObjectResult(context.ModelState);
    }
}
```

Next, let's register this action filter in the **ConfigureServices** method:

```
services.AddScoped<ValidationFilterAttribute>();
```

Finally, let's remove that validation code from our actions and call this action filter as a service:

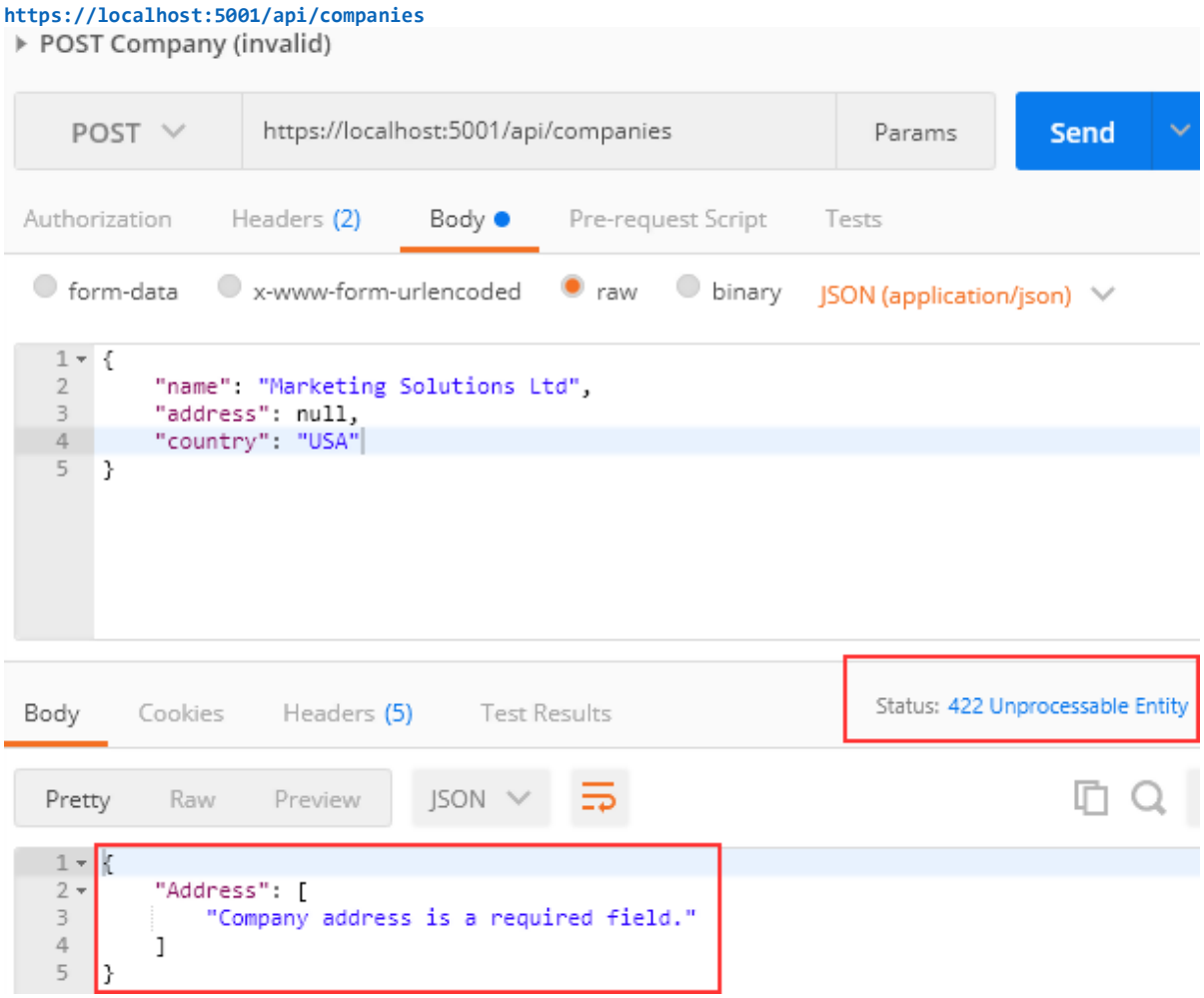
```
[HttpPost]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> CreateCompany([FromBody]CompanyForCreationDto
company)
{
    var companyEntity = _mapper.Map<Company>(company);

    _repository.Company.CreateCompany(companyEntity);
    await _repository.SaveAsync();

    var companyToReturn = _mapper.Map<CompanyDto>(companyEntity);

    return CreatedAtRoute("CompanyById", new { id = companyToReturn.Id },
    companyToReturn);
}
```

```
[HttpPut("{id}")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> UpdateCompany(Guid id, [FromBody] CompanyForUpdateDto
company)
{
    var companyEntity = await _repository.Company.GetCompanyAsync(id, trackChanges:
    true);
    if(companyEntity == null)
    {
        _logger.LogInfo($"Company with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    _mapper.Map(company, companyEntity);
    await _repository.SaveAsync();

    return NoContent();
}
```

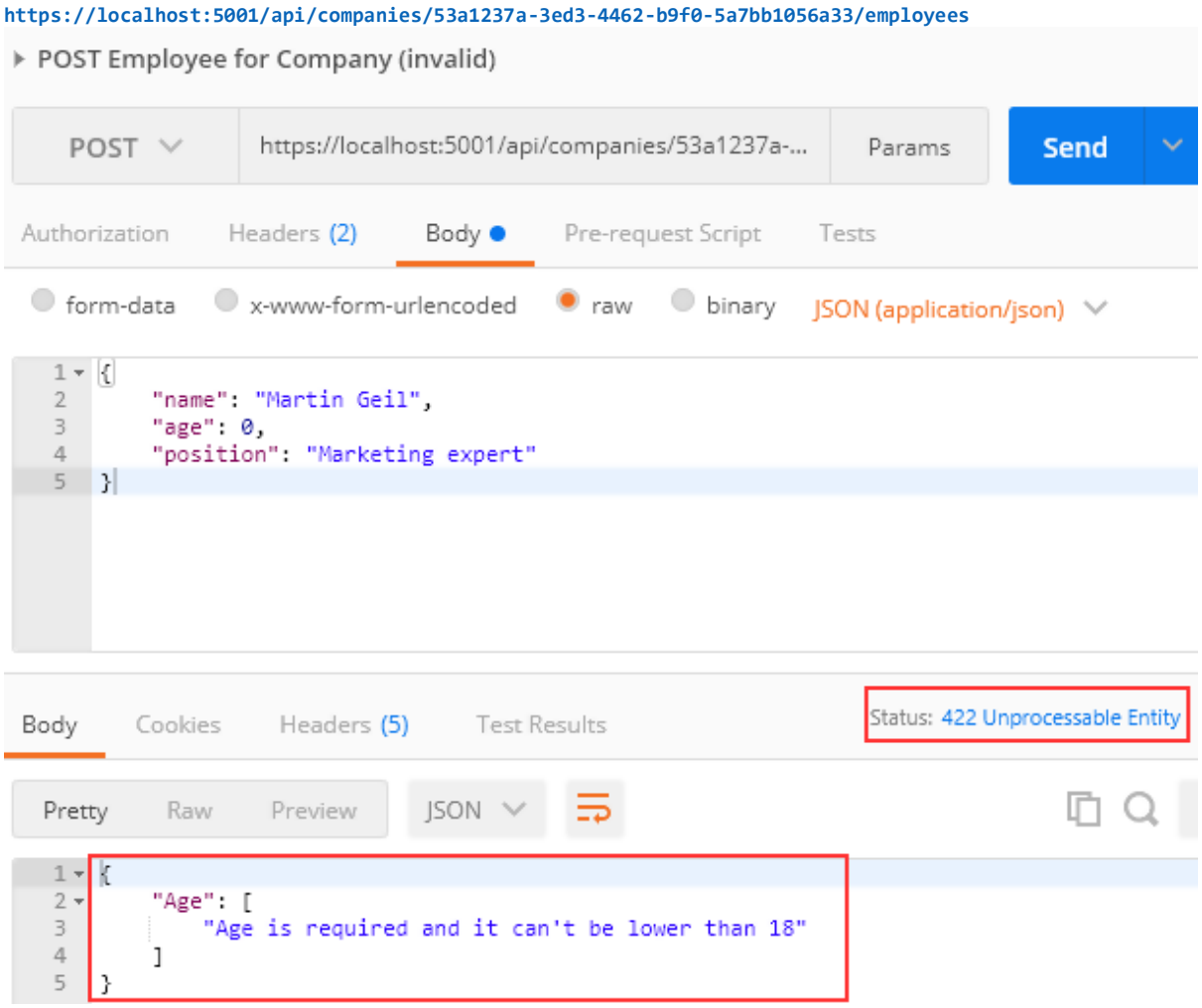
Excellent.

This code is much cleaner and more readable now without the validation part. Furthermore, the validation part is now reusable for the POST and PUT actions for both the Company and Employee DTO objects.

If we send a POST request, for example, with the invalid model we will get the required response:



We can apply this action filter to the POST and PUT actions in the `EmployeesController` the same way we did in the `CompaniesController` and test it as well:



Dependency Injection in Action Filters

If we take a look at our `DeleteCompany` and `UpdateCompany` actions, we are going to see the code where we fetch the company by id from the database and check if it exists:

```
if (company == null)
{
    _logger.LogInfo($"Company with id: {id} doesn't exist in the database.");
    return NotFound();
}
```

That's something we can extract to the Action Filter class as well, thus making it reusable in all the actions.

Of course, we need to inject our **repository** into a new ActionFilter class by using dependency injection.

Having said that, let's create another Action Filter class: **ValidateCompanyExistsAttribute** in the **ActionFilters** folder and modify it:

```
public class ValidateCompanyExistsAttribute : IAsyncActionFilter
{
    private readonly IRepositoryManager _repository;
    private readonly ILoggerManager _logger;
    public ValidateCompanyExistsAttribute(IRepositoryManager repository,
ILoggerManager logger)
    {
        _repository = repository;
        _logger = logger;
    }

    public async Task OnActionExecutionAsync(ActionExecutingContext context,
ActionExecutionDelegate next)
    {
        var trackChanges = context.HttpContext.Request.Method.Equals("PUT") ? true :
false;
        var id = (Guid)context.ActionArguments["id"];
        var company = await _repository.Company.GetCompanyAsync(id, trackChanges);

        if (company == null)
        {
            _logger.LogInfo($"Company with id: {id} doesn't exist in the database.");
            context.Result = new NotFoundResult();
        }
        else
        {
            context.HttpContext.Items.Add("company", company);
            await next();
        }
    }
}
```

We are using the async version of the action filter because we fetch our entity in an async manner. Two things to notice here. The first is that we check a type of request and only if it is a PUT request we set the trackChanges to true. The second thing is if we find the entity in the database, we store it in **HttpContext** because we need that entity in our action methods and we don't want to query the database two times (we would lose more than we gain if we double that action).

Now, let's register this filter:

```
services.AddScoped<ValidateCompanyExistsAttribute>();
```

And let's modify our actions:

```
[HttpDelete("{id}")]
[ServiceFilter(typeof(ValidateCompanyExistsAttribute))]
public async Task<IActionResult> DeleteCompany(Guid id)
{
    var company = HttpContext.Items["company"] as Company;

    _repository.Company.DeleteCompany(company);
    await _repository.SaveAsync();

    return NoContent();
}

[HttpPut("{id}")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
[ServiceFilter(typeof(ValidateCompanyExistsAttribute))]
public async Task<IActionResult> UpdateCompany(Guid id, [FromBody] CompanyForUpdateDto
company)
{
    var companyEntity = HttpContext.Items["company"] as Company;

    _mapper.Map(company, companyEntity);
    await _repository.SaveAsync();

    return NoContent();
}
```

Now our actions look great without code repetition.

You can test these actions with the prepared (Delete and Put) requests in Postman. Of course, the implementation for the **`EmployeesController`** is almost the same (except some differences in a filter implementation).

So, let's see how to do that:

```
public class ValidateEmployeeForCompanyExistsAttribute : IAsyncActionFilter
{
    private readonly IRepositoryManager _repository;
    private readonly ILoggerManager _logger;

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

```
    public async Task OnActionExecutionAsync(ActionExecutingContext context,
ActionExecutionDelegate next)
    {
        var method = context.HttpContext.Request.Method;
        var trackChanges = (method.Equals("PUT") || method.Equals("PATCH")) ? true :
false;

        var companyId = (Guid)context.ActionArguments["companyId"];
        var company = await _repository.Company.GetCompanyAsync(companyId, false);

        if (company == null)
        {
            _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
            context.Result = new NotFoundResult();
            return;
        }

        var id = (Guid)context.ActionArguments["id"];
        var employee = await _repository.Employee.GetEmployeeAsync(companyId, id,
trackChanges);

        if(employee == null)
        {
            _logger.LogInfo($"Employee with id: {id} doesn't exist in the database.");
            context.Result = new NotFoundResult();
        }
        else
        {
            context.HttpContext.Items.Add("employee", employee);
            await next();
        }
    }
}
```

Then the registration part:

```
services.AddScoped<ValidateEmployeeForCompanyExistsAttribute>();
```

And finally, the controller modification.

Delete:

```
[HttpDelete("{id}")]
[ServiceFilter(typeof(ValidateEmployeeForCompanyExistsAttribute))]
public async Task<IActionResult> DeleteEmployeeForCompany(Guid companyId, Guid id)
{
    var employeeForCompany = HttpContext.Items["employee"] as Employee;

    _repository.Employee.DeleteEmployee(employeeForCompany);
    await _repository.SaveAsync();

    return NoContent();
}
```

Update:

```
[HttpPut("{id}")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
[ServiceFilter(typeof(ValidateEmployeeForCompanyExistsAttribute))]
public async Task<IActionResult> UpdateEmployeeForCompany(Guid companyId, Guid id,
[FromBody] EmployeeForUpdateDto employee)
{
    var employeeEntity = HttpContext.Items["employee"] as Employee;

    _mapper.Map(employee, employeeEntity);
    await _repository.SaveAsync();

    return NoContent();
}
```

And Patch:

```
[HttpPatch("{id}")]
[ServiceFilter(typeof(ValidateEmployeeForCompanyExistsAttribute))]
public async Task<IActionResult> PartiallyUpdateEmployeeForCompany(Guid companyId,
Guid id, [FromBody] JsonPatchDocument<EmployeeForUpdateDto> patchDoc)
{
    if(patchDoc == null)
    {
        _logger.LogError("patchDoc object sent from client is null.");
        return BadRequest("patchDoc object is null");
    }

    var employeeEntity = HttpContext.Items["employee"] as Employee;

    var employeeToPatch = _mapper.Map<EmployeeForUpdateDto>(employeeEntity);

    patchDoc.ApplyTo(employeeToPatch, ModelState);

    TryValidateModel(employeeToPatch);

    if(!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the patch document");
        return UnprocessableEntity(ModelState);
    }

    _mapper.Map(employeeToPatch, employeeEntity);

    await _repository.SaveAsync();

    return NoContent();
}
```

These changes can be tested as well with prepared requests in our Postman document.

One last thing.

If we take a look at the Employees and the Companies controller, we will find the `GetEmployeeForCompany` action and the `GetCompany` action. For both actions, we can implement these "ExistsAttribute" filters, but then those actions must be synchronous. That's because there will be no async code left. It is up to you whether you want to implement them or not.

PAGING

We have covered a lot of interesting features while creating our Web API project, but there are still things to do.

So, in this chapter, we're going to learn how to implement paging in ASP.NET Core Web API. It is one of the most important concepts in building RESTful APIs.

If we inspect the `GetEmployeesForCompany` action in the `EmployeesController`, we can see that we return all the employees for the single company.

But we don't want to return a collection of all resources when querying our API. That can cause performance issues and it's in no way optimized for public or private APIs. It can cause massive slowdowns and even application crashes in severe cases.

Of course, we should learn a little more about Paging before we dive into code implementation.

What is Paging?

Paging refers to **getting partial results from an API**. Imagine having millions of results in the database and having your application try to return all of them at once.

Not only that would be an **extremely ineffective** way of returning the results, but it could also possibly have **devastating effects on the application itself or the hardware it runs on**. Moreover, every client has limited memory resources and it needs to restrict the number of shown results.

Thus, we need a way to return a set number of results to the client in order to avoid these consequences. Let's see how we can do that.

Paging Implementation

Mind you, we don't want to change the base repository logic or implement any business logic in the controller.

What we want to achieve is something like this: `https://localhost:5001/api/companies/companyId/employees?pageNumber=2&pageSize=2`. This should return the second set of two employees we have in our database.

We also want to constrain our API not to return all the employees even if someone calls `https://localhost:5001/api/companies/companyId/employees`.

Let's start with the controller modification by modifying the `GetEmployeesForCompany` action:

```
[HttpGet]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId, [FromQuery]
EmployeeParameters employeeParameters)
{
    var company = await _repository.Company.GetCompanyAsync(companyId, trackChanges:
false);
    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
    }

    var employeesFromDb = await _repository.Employee.GetEmployeesAsync(companyId,
employeeParameters, trackChanges: false);

    var employeesDto = _mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

    return Ok(employeesDto);
}
```

A few things to take note of here:

- We're calling the `GetEmployeesForCompany` method from the `EmployeeRepository`, which doesn't exist yet, but we'll implement it soon.

- We're using `[FromQuery]` to point out that we'll be using query parameters to define which page and how many employees we are requesting.
- The `EmployeeParameters` class is the container for the actual parameters for the Employee entity.

We also need to actually create the `EmployeeParameters` class. So, let's first create a `RequestFeatures` folder in the `Entities` project and then inside, create the required classes:

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

As you can see, we create an abstract class to hold the common properties for all the entities in our project, and a single EmployeeParameters class that will hold the specific parameters. It is empty now, but soon it won't be.

In the abstract class, we are using the `maxPageSize` constant to restrict our API to a maximum of 50 rows per page. We have two public properties – `PageNumber` and `PageSize`. If not set by the caller, `PageNumber` will be set to 1, and `PageSize` to 10.

Now we can return to the controller and import a using directive for the `EmployeeParameters` class:

```
using Entities.RequestFeatures;
```

After that change, let's implement the most important part — the repository logic. We need to modify the `GetEmployeesAsync` method in the `IEmployeeRepository` interface and the `EmployeeRepository` class.

So, first the interface modification:

```
public interface IEmployeeRepository
{
    Task<IEnumerable<Employee>> GetEmployeesAsync(Guid companyId, EmployeeParameters
employeeParameters, bool trackChanges);
    Task<Employee> GetEmployeeAsync(Guid companyId, Guid id, bool trackChanges);
    void CreateEmployeeForCompany(Guid companyId, Employee employee);
    void DeleteEmployee(Employee employee);
}
```

And the repository logic:

```
public async Task<IEnumerable<Employee>> GetEmployeesAsync(Guid companyId,
EmployeeParameters employeeParameters, bool trackChanges) =>
    await FindByCondition(e => e.CompanyId.Equals(companyId), trackChanges)
    .OrderBy(e => e.Name)
    .Skip((employeeParameters.PageNumber - 1) * employeeParameters.PageSize)
    .Take(employeeParameters.PageSize)
    .ToListAsync();
```

Okay, the easiest way to explain this is by example.

Say we need to get the results for the third page of our website, counting 20 as the number of results we want. That would mean we want to skip the first ((3 – 1) * 20) = 40 results, then take the next 20 and return them to the caller.

Does that make sense?

Concrete Query

Before we continue, we should create additional employees for the company with the id: `C9D4C053-49B6-410C-BC78-2D54A9991870`. We

are doing this because we have only a small number of employees per company and we need more of them for our example. You can use a predefined request in Part16 in Postman, and just change the request body with the following objects:

{
"name": "Mihael Worth",
"age": 30,
"position": "Marketing expert"
}
	{
	"name": "John Spike",
	"age": 30,
	"position": "Marketing expert II"
	}
		{
		"name": "Nina Hawk",
		"age": 26,
		"position": "Marketing expert II"
		}

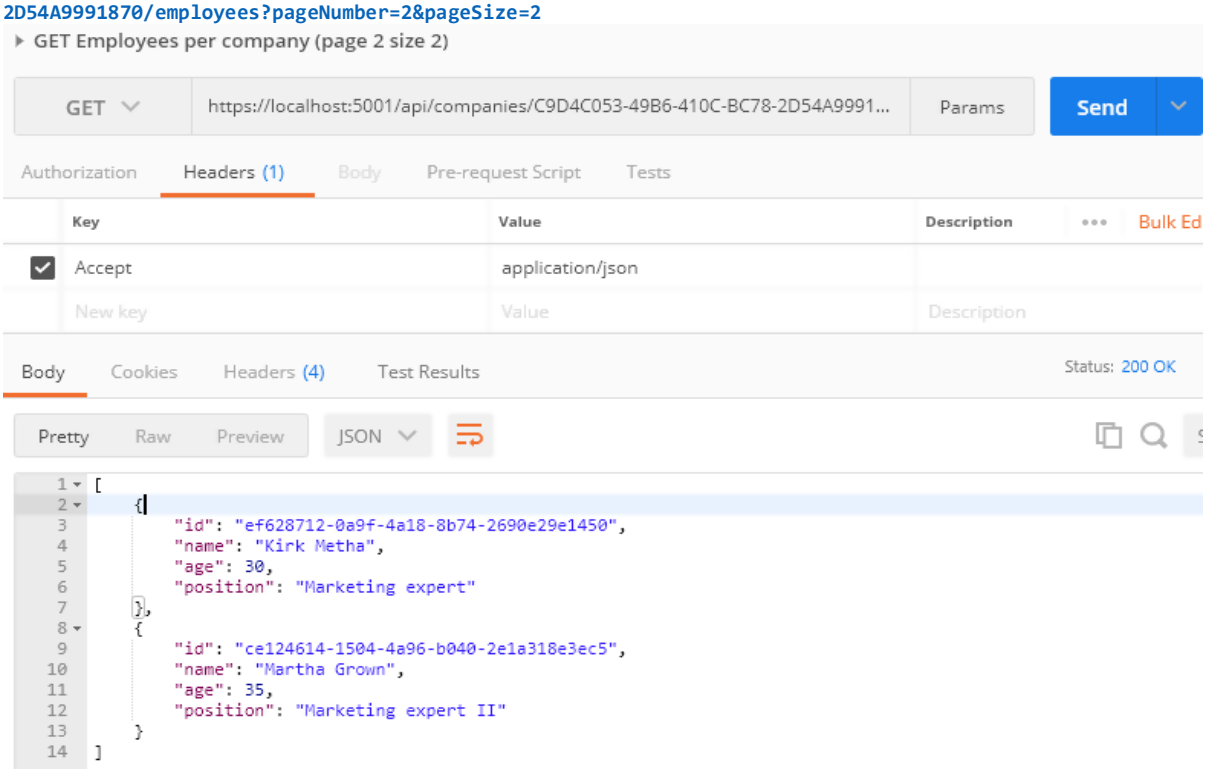
{
"name": "Mihael Fins",
"age": 30,
"position": "Marketing expert"
}
	{
	"name": "Martha Grown",
	"age": 35,
	"position": "Marketing expert II"
	}
		{
		"name": "Kirk Metha",
		"age": 30,
		"position": "Marketing expert"
		}

Now we should have eight employees for this company, and we can try a request like this:

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

So, we request page two with two employees:

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



If that's what you got, you're on the right track.

We can check our result in the database:

	EmployeeId	Name	Age	Position	CompanyId
1	86DBA8C0-D178-41E7-938C-ED49778FB52A	Jana McLeaf	30	Software developer	C9D4C053-49B6-410C-BC78-2D54A9991870
2	5E9C6E0D-94D2-4AE1-9562-910AA615AA11	Joan Dane	29	Manager	0AD5B971-FF51-414D-AF01-34187E407557
3	64EC59E9-94F5-4B14-B781-92899D2971D5	John Spike	32	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
4	021CA3C1-0DEB-4AFD-AE94-2159A8479811	Kane Miller	35	Administrator	3D490A70-94CE-4D15-9494-5248280C2CE3
5	EF628712-0A9F-4A18-8B74-2690E29E1450	Kirk Metha	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870
6	CE124614-1504-4A96-B040-2E1A318E3EC5	Martha Grown	35	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
7	DE662003-ACC3-4F9F-9D82-0A74F64594C1	Martin Geil	29	Administrative	0AD5B971-FF51-414D-AF01-34187E407557
8	9AD82BDC-6D18-481A-BC35-F4999A312893	Martin Geil	29	Marketing expert	53A1237A-3ED3-4462-B9F0-5A7BB1056A33
9	5436E51D-690A-467E-9600-2DBEEBCAA0DD	Mihael Fins	30	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
10	2E2D6091-5E69-49A4-A2EC-3AA8AABEE078	Mihael Worth	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870
11	2557B87E-388A-4679-B6CC-A514977DB5EF	Nina Hawk	26	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
12	80ABBCA8-664D-4B20-B5DE-024705497D4A	Sam Raiden	26	Software developer	C9D4C053-49B6-410C-BC78-2D54A9991870

And we can see that we have a correct data returned.

Now, what can we do to improve this solution?

Improving the Solution

Since we're returning just a subset of results to the caller, we might as well have a `PagedList` instead of `List`.

`PagedList` will inherit from the `List` class and will add some more to it. We can also move the skip/take logic to the `PagedList` since it makes more sense.

So, let's first create a new `MetaData` class in the `Entities/RequestFeatures` folder:

```
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;
}
```

Then, we are going to implement the `PagedList` class in the same folder:

```
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);
    }

    public static PagedList<T> ToPagedList(IEnumerable<T> source, int pageNumber, int
pageSize)
    {
        var count = source.Count();
        var items = source
            .Skip((pageNumber - 1) * pageSize)
            .Take(pageSize).ToList();

        return new PagedList<T>(items, count, pageNumber, pageSize);
    }
}
```

As you can see, we've transferred the skip/take logic to the static method inside of the `PagedList` class. And in the `MetaData` class, we've added a few more properties that will come in handy as metadata for our response.

`HasPrevious` is true if the `CurrentPage` is larger than 1, and `HasNext` is calculated if the `CurrentPage` is smaller than the number of total pages. `TotalPages` is calculated by dividing the number of items by the page size and then rounding it to the larger number, since a page needs to exist even if there is only one item on it.

Now that we've cleared that up, let's change our `EmployeeRepository` and `EmployeesController` accordingly.

Let's start with the interface modification:

```
Task<PagedList<Employee>> GetEmployeesAsync(Guid companyId, EmployeeParameters
employeeParameters, bool trackChanges);
```

Then, let's change the repository class:

```
public async Task<PagedList<Employee>> GetEmployeesAsync(Guid companyId,
EmployeeParameters employeeParameters, bool trackChanges)
{
    var employees = await FindByCondition(e => e.CompanyId.Equals(companyId),
trackChanges)
        .OrderBy(e => e.Name)
        .ToListAsync();

    return PagedList<Employee>
       .ToPagedList(employees, employeeParameters.PageNumber,
employeeParameters.PageSize);
 }
```

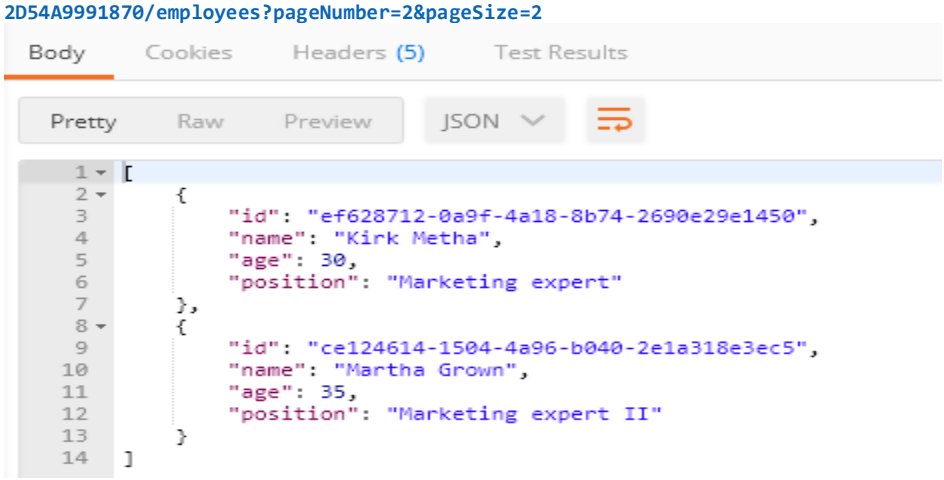
And finally, let's modify the controller:

```
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId, [FromQuery]
EmployeeParameters employeeParameters)
{
    var company = await _repository.Company.GetCompanyAsync(companyId, trackChanges:
false);
    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
     }
```

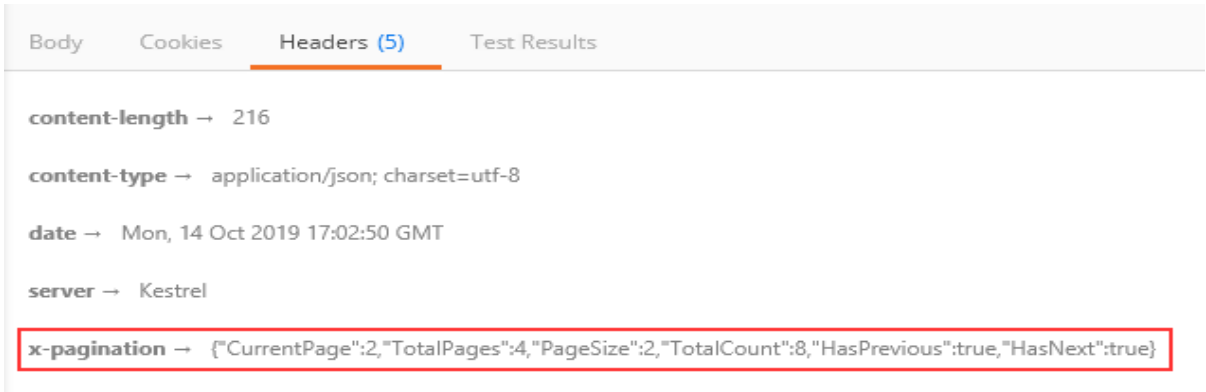
```
    var employeesFromDb = await _repository.Employee.GetEmployeesAsync(companyId,
employeeParameters, trackChanges: false);

    Response.Headers.Add("X-Pagination",
JsonConvert.SerializeObject(employeesFromDb.MetaData));

    var employeesDto = _mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

    return Ok(employeesDto);
 }
```

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

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



But now we have some additional useful information in the X-Pagination response header:



As you can see, all of our metadata is here. We can use this information when building any kind of frontend pagination to our benefit. You can play around with different requests to see how it works in other scenarios.

We could also use this data to generate links to the previous and next pagination page on the backend, but that is part of the HATEOAS and is out of the scope of this chapter.

FILTERING

In this chapter, we are going to cover filtering in ASP.NET Core Web API. We'll learn what filtering is, how it's different from searching, and how to implement it in a real-world project.

While not critical as paging, filtering is still an important part of a flexible REST API, so we need to know how to implement it in our API projects. Filtering helps us get the exact result set we want instead of all the results without any criteria.

What is Filtering?

Filtering is a mechanism to retrieve results by **providing some kind of criterion**. We can write many kinds of filters to get results by type of class property, value range, date range, or anything else.

When implementing filtering, you are always restricted by the predefined set of options you can set in your request. For example, you can send a date value to request an employee, but you won't have much success.

On the front end, filtering is usually implemented as checkboxes, radio buttons, or dropdowns. This kind of implementation limits you to only those options that are available to create a valid filter.

Take for example a car-selling website. When filtering the cars you want, you would ideally want to select:

- Car manufacturer as a category from a list or a dropdown
- Car model from a list or a dropdown
- Is it new or used with radio buttons
- The city where the seller is as a dropdown
- The price of the car is an input field (numeric)
- ….

You get the point. So, the request would look something like this:

```
https://bestcarswebsite.com/sale?manufacturer=ford&model=expedition&state=used&city=washington&price_from=30000&price_to=50000
```

Or even like this:

```
https://bestcarswebsite.com/sale/filter?data[manufacturer]=ford&[model]=expedition&[state]=used&[city]=washington&[price_from]=30000&[price_to]=50000
```

Now that we know what filtering is, let's see how it's different from searching.

How is Filtering Different from Searching?

When searching for results, we usually have only one input and that's the one you use to search for anything within a website.

So in other words, you send a string to the API and the API is responsible for using that string to find any results that match it.

On our car website, we would use the search field to find the "Ford Expedition" car model and we would get all the results that match the car name "Ford Expedition." Thus, this search would return every "Ford Expedition" car available.

We can also improve the search by implementing search terms like Google does, for example. If the user enters the Ford Expedition without quotes in the search field, we would return both what's relevant to Ford and Expedition. But if the user puts quotes around it, we would search the entire term "Ford Expedition" in our database.

It makes a better user experience.

Example:

```
https://bestcarswebsite.com/sale/search?name=ford focus
```

Using search doesn't mean we can't use filters with it. It makes perfect sense to use filtering and searching together, so we need to take that into account when writing our source code.

But enough theory.

Let's implement some filters.

How to Implement Filtering in ASP.NET Core Web API

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.

We would need a query like this one:

```
https://localhost:5001/api/companies/companyId/employees?minAge=26&maxAge=29
```

But, we want to be able to do this too:

```
https://localhost:5001/api/companies/companyId/employees?minAge=26
```

Or like this:

```
https://localhost:5001/api/companies/companyId/employees?maxAge=29
```

Okay, we have a specification. Let's see how to implement it.

We've already implemented paging in our controller, so we have the necessary infrastructure to extend it with the filtering functionality. We've used the `EmployeeParameters` class, which inherits from the `RequestParameters` class, to define the query parameters for our paging request.

Let's extend the `EmployeeParameters` class:

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

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

We've added two unsigned int properties (to avoid negative year values): `MinAge` and `MaxAge`.

Since the default uint value is 0, we don't need to explicitly define it; 0 is okay in this case. For `MaxAge`, we want to set it to the max int value. If we don't get it through the query params, we have something to work with. It doesn't matter if someone sets the age to 300 through the params; it won't affect the results.

We've also added a simple validation property – ValidAgeRange. Its purpose is to tell us if the max-age is indeed greater then min-age. If it's not, we want to let the API user know that he/she is doing something wrong.

Okay, now that we have our parameters ready, we can modify the `GetEmployeesForCompany` action by adding validation check as a first statement:

```
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId, [FromQuery]
EmployeeParameters employeeParameters)
{
    if(!employeeParameters.ValidAgeRange)
        return BadRequest("Max age can't be less than min age.");

...the rest of the code...

}
```

As you can see, there's not much to it. We've added our validation check and a BadRequest response with a short message to the API user.

That should do it for the controller.

Let's get to the implementation in our `EmployeeRepository` class:

```
public async Task<PagedList<Employee>> GetEmployeesAsync(Guid companyId,
EmployeeParameters employeeParameters, bool trackChanges)
{
    var employees = await FindByCondition(e => e.CompanyId.Equals(companyId) && (e.Age
>= employeeParameters.MinAge && e.Age <= employeeParameters.MaxAge), trackChanges)
      .OrderBy(e => e.Name)
      .ToListAsync();

    return PagedList<Employee>
     .ToPagedList(employees, employeeParameters.PageNumber,
employeeParameters.PageSize);
}
```

Actually, at this point, the implementation is rather simple too.

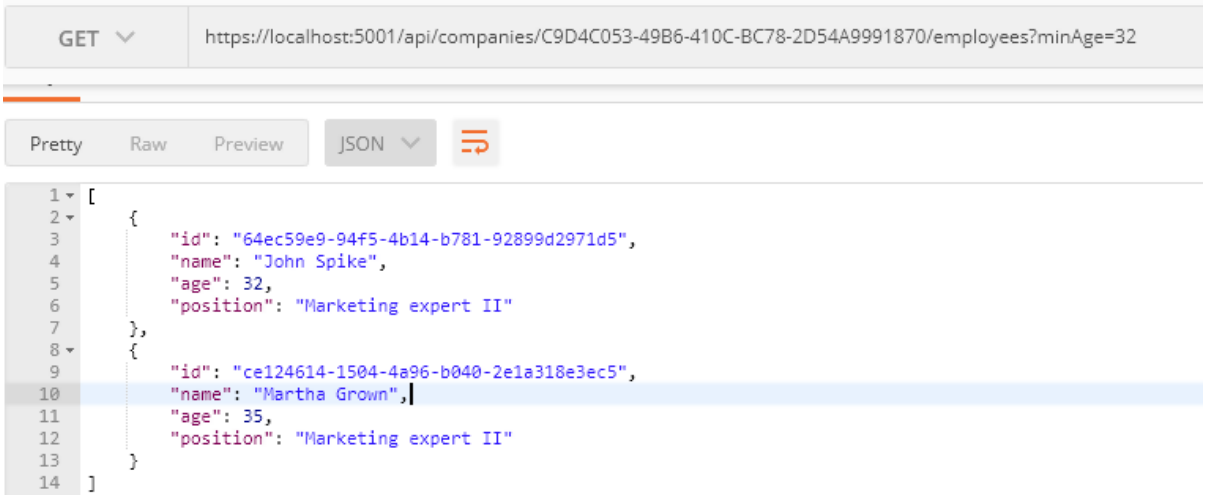
We are using the `FindByCondition` method to find all the employees with an `Age` between the `MaxAge` and the `MinAge`.

Let's try it out.

Sending and Testing a Query

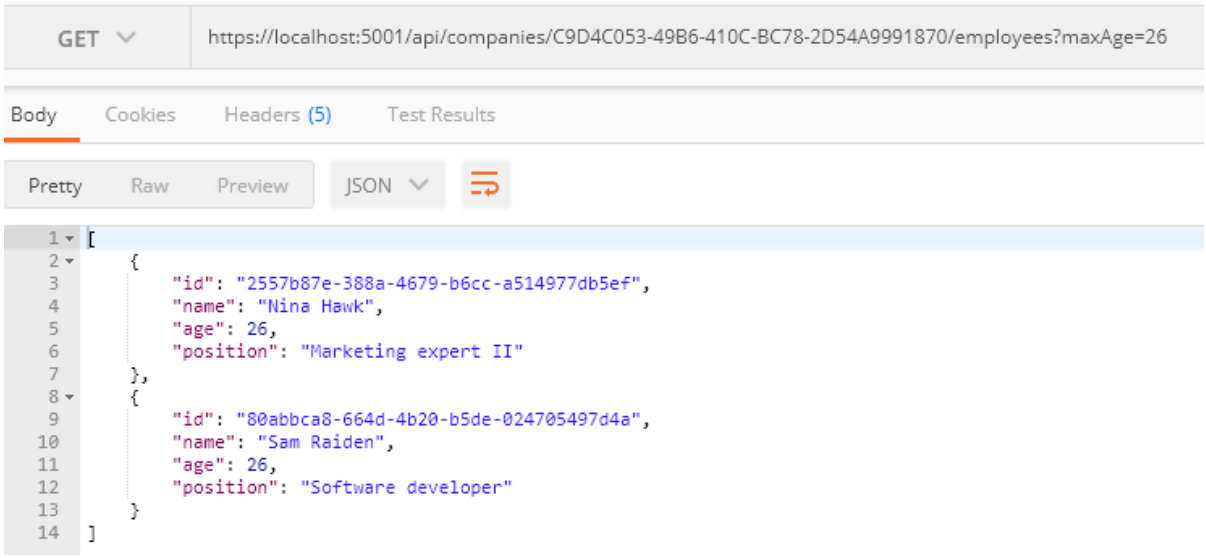
Let's send a first request with only a MinAge parameter:

`https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?minAge=32`



Next, let's send one with only a MaxAge parameter:

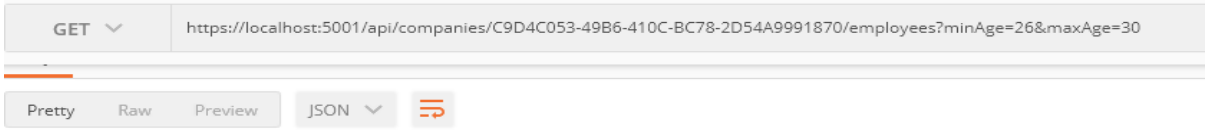
https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?maxAge=26




```
[
    {
        "id": "2557b87e-388a-4679-b6cc-a514977db5ef",
        "name": "Nina Hawk",
        "age": 26,
        "position": "Marketing expert II"
    },
    {
        "id": "80abbca8-664d-4b20-b5de-024705497d4a",
        "name": "Sam Raiden",
        "age": 26,
        "position": "Software developer"
    }
]
```

After that, we can combine those two:

https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?minAge=26&maxAge=30




```
[
    {
        "id": "86dba8c0-d178-41e7-938c-ed49778fb52a",
        "name": "Jana McLeaf",
        "age": 30,
        "position": "Software developer"
    },
    {
        "id": "ef628712-0a9f-4a18-8b74-2690e29e1450",
        "name": "Kirk Metha",
        "age": 30,
        "position": "Marketing expert"
    },
    {
        "id": "5436e51d-690a-467e-9600-2dbeebcaa0dd",
        "name": "Mihael Fins",
        "age": 30,
        "position": "Marketing expert II"
    },
    {
        "id": "2e2d6091-5e69-49a4-a2ec-3aa8aabee078",
        "name": "Mihael Worth",
        "age": 30,
        "position": "Marketing expert"
    },
    {
        "id": "2557b87e-388a-4679-b6cc-a514977db5ef",
        "name": "Nina Hawk",
        "age": 26,
        "position": "Marketing expert II"
    },
    {
        "id": "80abbca8-664d-4b20-b5de-024705497d4a",
        "name": "Sam Raiden",
        "age": 26,
        "position": "Software developer"
    }
]
```

And finally, we can test the filter with the paging:

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

Excellent. The filter is implemented and we can move on to the searching part.

SEARCHING

In this chapter, we're going to tackle the topic of searching in ASP.NET Core Web API. Searching is one of those functionalities that can make or break your API, and the level of difficulty when implementing it can vary greatly depending on your specifications.

If you need to implement a basic searching feature where you are just trying to search one field in the database, you can easily implement it. On the other hand, if it's a multi-column, multi-term search, you would probably be better off with some of the great search libraries out there like Lucene.NET which are already optimized and proven.

What is Searching?

There is no doubt in our minds that you've seen a search field on almost every website on the internet. It's easy to find something when we are familiar with the website structure or when a website is not that large.

But if we want to find the most relevant topic for us, we don't know what we're going to find, or maybe we're first-time visitors to a large website, we're probably going to use a search field.

In our simple project, one use case of a search would be to find an employee by name.

Let's see how we can achieve that.

Implementing Searching in Our Application

Since we're going to implement the most basic search in our project, the implementation won't be complex at all. We have all we need infrastructure-wise since we already covered paging and filtering. We'll just extend our implementation a bit.

What we want to achieve is something like this:

`https://localhost:5001/api/companies/companyId/employees?searchTerm=Mihael Fins`

This should return just one result: Mihael Fins. Of course, the search needs to work together with filtering and paging, so that's one of the things we'll need to keep in mind too.

Like we did with filtering, we're going to extend our `EmployeeParameters` class first since we're going to send our search query as a query parameter:

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

    public bool ValidAgeRange => MaxAge > MinAge;

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

Simple as that.

Now we can write queries with `name="term"` in them.

The next thing we need to do is actually implement the search functionality in our `EmployeeRepository` class:

```
public async Task<PagedList<Employee>> GetEmployeesAsync(Guid companyId,
EmployeeParameters employeeParameters, bool trackChanges)
{
    var employees = await FindByCondition(e => e.CompanyId.Equals(companyId),
trackChanges)
        .FilterEmployees(employeeParameters.MinAge, employeeParameters.MaxAge)
        .Search(employeeParameters.SearchTerm)
        .OrderBy(e => e.Name)
        .ToListAsync();

    return PagedList<Employee>
        .ToPagedList(employees, employeeParameters.PageNumber,
employeeParameters.PageSize);
}
```

As you can see, we have made two changes here. The first is modifying the filter logic and the second is adding the `Search` method for the

searching functionality. But these methods (FilterEmployees and Search) are not created yet, so let's create them.

In the **Repository** project, we are going to create the new folder **Extensions** and inside of that folder the new class **RepositoryEmployeeExtensions**:

```
public static class RepositoryEmployeeExtensions
{
    public static IQueryable<Employee> FilterEmployees(this IQueryable<Employee>
employees, uint minAge, uint maxAge) =>
        employees.Where(e => (e.Age >= minAge && e.Age <= maxAge));

    public static IQueryable<Employee> Search(this IQueryable<Employee> employees,
string searchTerm)
    {
        if (string.IsNullOrWhiteSpace(searchTerm))
            return employees;

        var lowerCaseTerm = searchTerm.Trim().ToLower();

        return employees.Where(e => e.Name.ToLower().Contains(lowerCaseTerm));
    }
}
```

So, we are just creating our extension methods to update our query until it is executed in the repository. Now, all we have to do is add a using directive to the EmployeeRepository class:

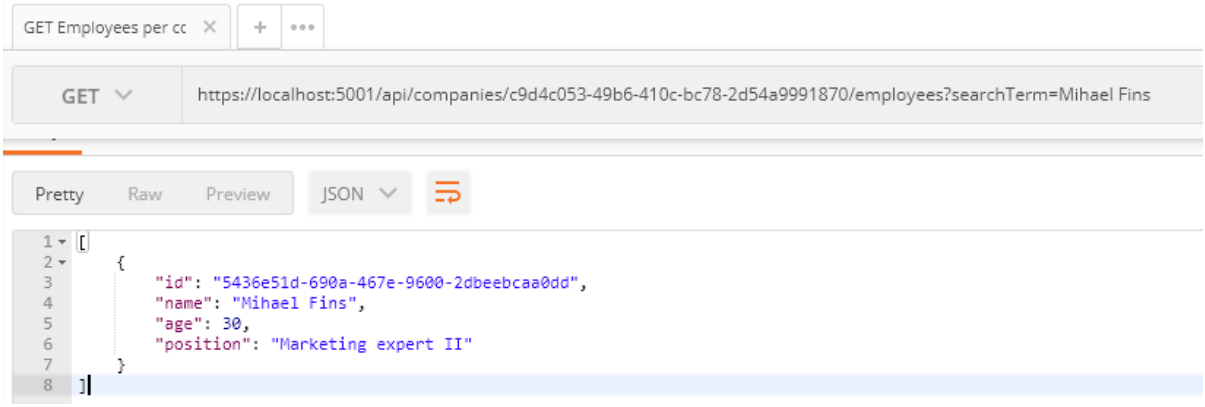
```
using Repository.Extensions;
```

That's it for our implementation. As you can see, it isn't that hard since it is the most basic search and we already had an infrastructure set.

Testing Our Implementation

Let's send a first request with the value Mihael Fins for the search term:

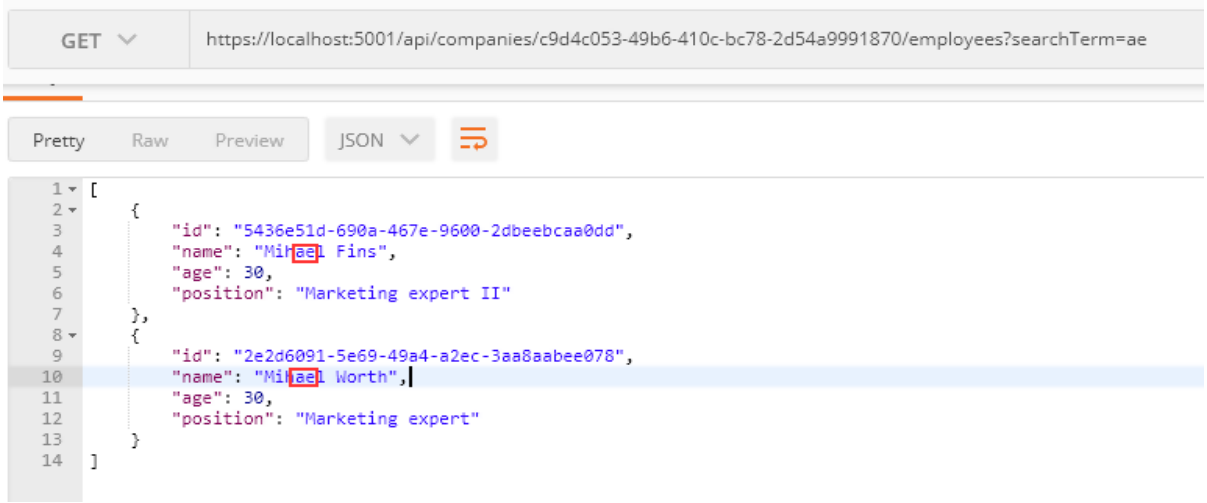
https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?searchTerm=Mihael Fins



This is working great.

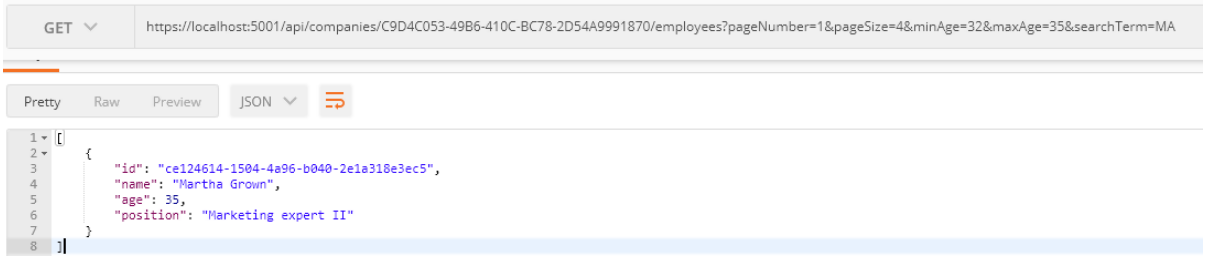
Now, let's find all employees that contain the letters "ae":

https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?searchTerm=ae



Great. One more request with the paging and filtering:

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



And this works as well.

That’s it! We’ve successfully implemented and tested our search functionality.

If we check the Headers tab for each request, we will find valid x-pagination as well.

SORTING

In this chapter, we're going to talk about sorting in ASP.NET Core Web API. Sorting is a commonly used mechanism that every API should implement. Implementing it in ASP.NET Core is not difficult due to the flexibility of LINQ and good integration with EF Core.

So, let's talk a bit about sorting.

What is Sorting?

Sorting, in this case, refers to ordering our results in a preferred way using our query string parameters. We are not talking about sorting algorithms nor are we going into the how's of implementing a sorting algorithm.

What we're interested in, however, is how do we make our API sort our results the way we want it to.

Let's say we want our API to sort employees by their name in ascending order, and then by their age.

To do that, our API call needs to look something like this:

```
https://localhost:5001/api/companies/companyId/employees?orderBy=name,age desc
```

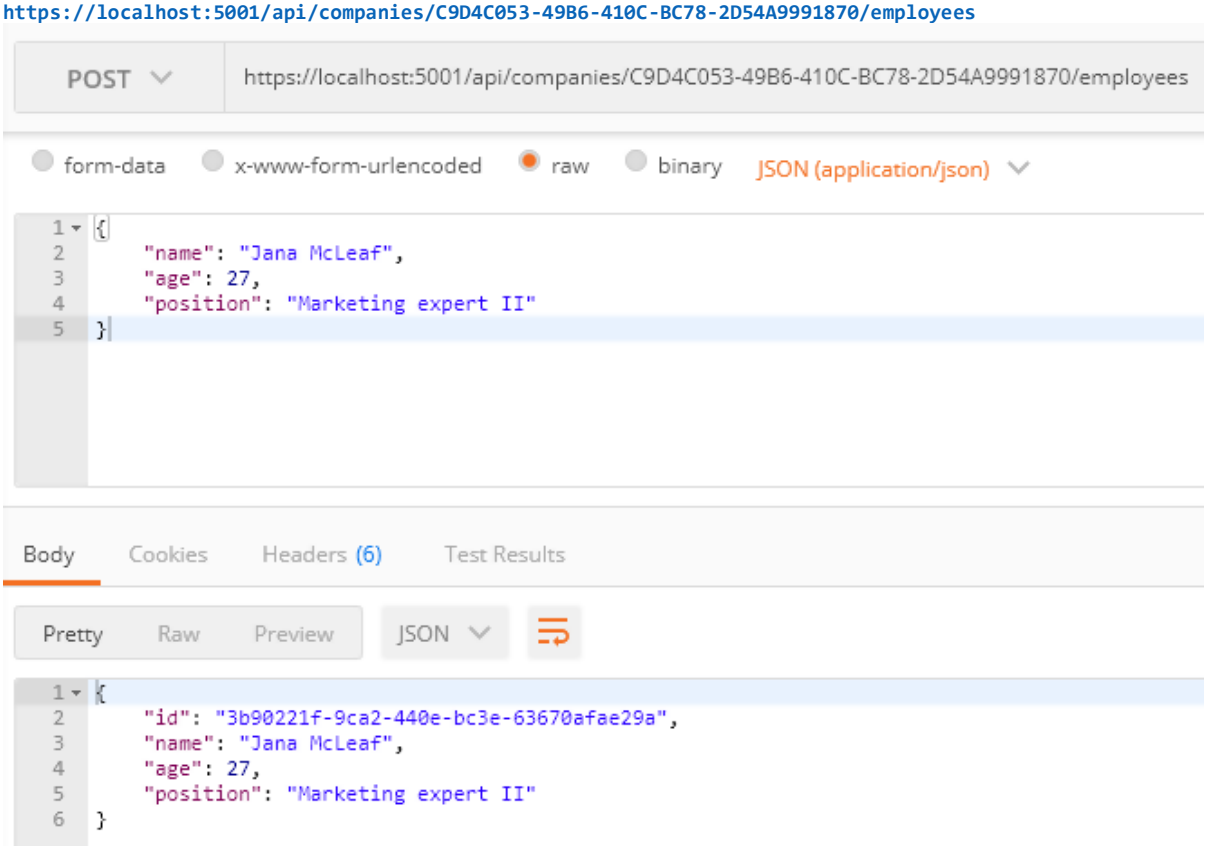
Our API needs to take all the parameters into consideration and sort our results accordingly. In our case, this means sorting results by their name; then, if there are employees with the same name, sorting them by the age property.

So, these are our employees for the IT_Solutions Ltd company:

	EmployeeId	Name	Age	Position	CompanyId
1	80ABBCA8-664D-4B20-B5DE-024705497D4A	Sam Raiden	26	Software developer	C9D4C053-49B6-410C-BC78-2D54A9991870
2	EF628712-0A9F-4A18-8B74-2690E29E1450	Kirk Metha	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870
3	5436E51D-690A-467E-9600-2DBEEBCAA0DD	Mihael Fins	30	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
4	CE124614-1504-4A96-B040-2E1A318E3EC5	Martha Grown	35	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
5	2E2D6091-5E69-49A4-A2EC-3AA8AABEE078	Mihael Worth	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870
6	64EC59E9-94F5-4B14-B781-92899D2971D5	John Spike	32	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
7	2557B87E-388A-4679-B6CC-A514977DB5EF	Nina Hawk	26	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
8	86DBA8C0-D178-41E7-938C-ED49778FB52A	Jana McLeaf	30	Software developer	C9D4C053-49B6-410C-BC78-2D54A9991870

For the sake of demonstrating this example (sorting by name and then by age), we are going to add one more Jana McLeaf to our database with the age of 27. You can add whatever you want to test the results:

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



Great, now we have the required data to test our functionality properly.

And of course, like with all other functionalities we have implemented so far (paging, filtering, and searching), we need to implement this to work well with everything else. We should be able to get the paginated, filtered, and sorted data, for example.

Let's see one way to go around implementing this.

How to Implement Sorting in ASP.NET Core Web API

As with everything else so far, first, we need to extend our `RequestParameters` class to be able to send requests with the orderBy clause in them:

```
public 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. We want to sort our results by name, even if it hasn't been stated explicitly in the request.

That said, let's modify the `EmployeeParameters` class to enable the default sorting condition for `Employee` if none was stated:

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

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

    public bool ValidAgeRange => MaxAge > MinAge;

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

```
}
```

Next, we're going to dive right into the implementation of our sorting mechanism, or rather, our ordering mechanism.

One thing to note is that we'll be using the **System.Linq.Dynamic.Core NuGet package** to dynamically create our **OrderBy** query on the fly. So, feel free to install it in the **Repository** project and add a using directive in the **RepositoryEmployeeExtensions** class:

```
using System.Linq.Dynamic.Core;
```

Now, we can add the new extension method **Sort** in our **RepositoryEmployeeExtensions** class:

```
public static IQueryable<Employee> Sort(this IQueryable<Employee> employees, string
orderByQueryString)
{
    if (string.IsNullOrWhiteSpace(orderByQueryString))
        return employees.OrderBy(e => e.Name);

    var orderParams = orderByQueryString.Trim().Split(',');
    var propertyInfos = typeof(Employee).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") ? "descending" : "ascending";

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

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

    if (string.IsNullOrWhiteSpace(orderQuery))
        return employees.OrderBy(e => e.Name);

    return employees.OrderBy(orderQuery);
}
```

Okay, there are a lot of things going on here, so let's take it step by step and see what exactly we've done.

Implementation – Step by Step

First, let start with the method definition. It has two arguments — one for the list of employees as IQueryable<Employee> and the other for the ordering query. If we send a request like this one: `https://localhost:5001/api/companies/companyId/employees?orderBy=name,age desc`, our `orderByQueryString` will be `name,age desc`.

We begin by executing some basic check against the orderByQueryString. If it is null or empty, we just return the same collection ordered by name.

```
if (string.IsNullOrWhiteSpace(orderByQueryString))
        return employees.OrderBy(e => e.Name);
```

Next, we are splitting our query string to get the individual fields:

```
var orderParams = orderByQueryString.Trim().Split(',');
```

We're also using a bit of reflection to prepare the list of `PropertyInfo` objects that represent the properties of our `Employee` class. We need them to be able to check if the field received through the query string really exists in the `Employee` class:

```
var propertyInfos = typeof(Employee).GetProperties(BindingFlags.Public |
BindingFlags.Instance);
```

That prepared, we can actually run through all the parameters and check for their existence:

```
if (string.IsNullOrWhiteSpace(param))
    continue;

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

If we don't find such a property, we skip the step in the foreach loop and go to the next parameter in the list:

```
if (objectProperty == null)
    continue;
```

If we do find the property, we return it and additionally check if our parameter contains "desc" at the end of the string. We use that to decide how we should order our property:

```
var direction = param.EndsWith(" desc") ? "descending" : "ascending";
```

We use the `StringBuilder` to build our query with each loop:

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

Now that we've looped through all the fields, we are just removing excess commas and doing one last check to see if our query indeed has something in it:

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

if (string.IsNullOrWhiteSpace(orderQuery))
    return employees.OrderBy(e => e.Name);
```

Finally, we can order our query:

```
return employees.OrderBy(orderQuery);
```

At this point, the `orderQuery` variable should contain the "`Name ascending, DateOfBirth descending`" string. That means it will order our results first by `Name` in ascending order, and then by `DateOfBirth` in descending order.

The standard LINQ query for this would be:

```
employees.OrderBy(e => e.Name).ThenByDescending(o => o.Age);
```

This is a neat little trick to form a query when you don't know in advance how you should sort.

Once we have done this, all we have to do is to modify the `GetEmployeesAsync` method:

```
public async Task<PagedList<Employee>> GetEmployeesAsync(Guid companyId,
EmployeeParameters employeeParameters, bool trackChanges)
{
    var employees = await FindByCondition(e => e.CompanyId.Equals(companyId),
trackChanges)
        .FilterEmployees(employeeParameters.MinAge, employeeParameters.MaxAge)
        .Search(employeeParameters.SearchTerm)
        .Sort(employeeParameters.OrderBy)
        .ToListAsync();

    return PagedList<Employee>
        .ToPagedList(employees, employeeParameters.PageNumber,
employeeParameters.PageSize);
}
```

And that's it! We can test this functionality now.

Testing Our Implementation

First, let's try out the query we've been using as an example:

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

And this is the result:

```
[
    {
        "id": "86dba8c0-d178-41e7-938c-ed49778fb52a",
        "name": "Jana McLeaf",
        "age": 30,
        "position": "Software developer"
    },
    {
        "id": "3b90221f-9ca2-440e-bc3e-63670afae29a",
        "name": "Jana McLeaf",
        "age": 27,
        "position": "Marketing expert II"
    },
    {
        "id": "64ec59e9-94f5-4b14-b781-92899d2971d5",
        "name": "John Spike",
        "age": 32,
        "position": "Marketing expert II"
    },
    {
        "id": "ef628712-0a9f-4a18-8b74-2690e29e1450",
        "name": "Kirk Metha",
        "age": 30,
        "position": "Marketing expert"
    },
    {
        "id": "ce124614-1504-4a96-b040-2e1a318e3ec5",
        "name": "Martha Grown",
        "age": 35,
        "position": "Marketing expert II"
    },
    {
        "id": "5436e51d-690a-467e-9600-2dbeebcaa0dd",
        "name": "Mihael Fins",
        "age": 30,
        "position": "Marketing expert II"
    }, ...
```


As you can see, the list is sorted by Name ascending. Since we have two Jana's, they were sorted by Age descending.

We have prepared additional requests which you can use to test this functionality with Postman. So, feel free to do it.

Improving the Sorting Functionality

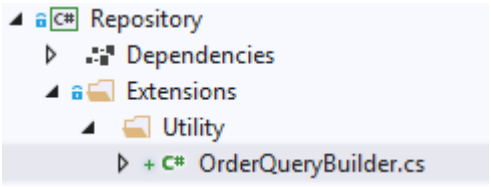
Right now, sorting only works with the Employee entity, but what about the Company? It is obvious that we have to change something in our implementation if we don't want to repeat our code while implementing sorting for the Company entity.

That said, let's modify the `Sort` extension method:

```
public static IQueryable<Employee> Sort(this IQueryable<Employee> employees, string
orderByQueryString)
{
    if (string.IsNullOrWhiteSpace(orderByQueryString))
        return employees.OrderBy(e => e.Name);

    var orderQuery = OrderQueryBuilder.CreateOrderQuery<Employee>(orderByQueryString);

    if (string.IsNullOrWhiteSpace(orderQuery))
        return employees.OrderBy(e => e.Name);

     return employees.OrderBy(orderQuery);
 }
```

So, we are extracting a logic that can be reused in the `CreateOrderQuery<T>` method. But of course, we have to create that method.

Let's create a `Utility` folder in the `Extensions` folder with the new class `OrderQueryBuilder`:

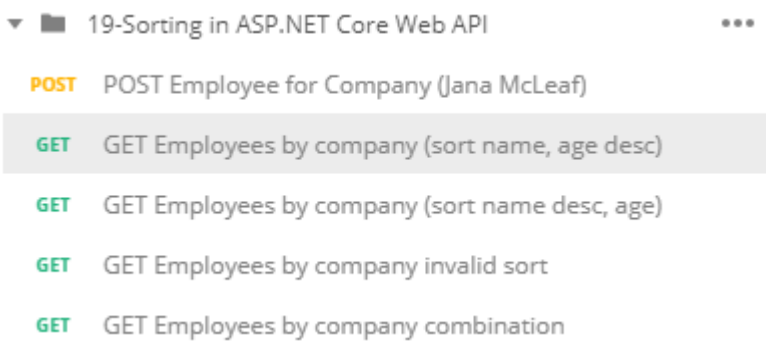


Now, let's modify that class:

```
public static class OrderQueryBuilder
{
    public static string CreateOrderQuery<T>(string orderByQueryString)
    {
        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") ? "descending" : "ascending";
```

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

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

        return orderQuery;
    }
}
```

And there we go. Not too many changes, but we did a great job here. You can test this solution with the prepared requests in Postman and you'll get the same result for sure:



But now, this functionality is reusable.

DATA SHAPING

In this chapter, we are going to talk about a neat concept called data shaping and how to implement it in ASP.NET Core Web API. To achieve that, we are going to use similar tools to the previous section. Data shaping is not something that every API needs, but it can be very useful in some cases.

Let's start by learning what data shaping is exactly.

What is Data Shaping?

Data shaping is a great way to reduce the amount of traffic sent from the API to the client. It **enables the consumer of the API to select (shape) the data by choosing the fields through the query string**.

What this means is something like:

```
https://localhost:5001/api/companies/companyId/employees?fields=name,age
```

By giving the consumer a way to select just the fields it needs, we can potentially **reduce the stress on the API**. On the other hand, **this is not something every API needs**, so we need to think carefully and decide whether we should implement its implementation because it has a bit of reflection in it.

And we know for a fact that reflection takes its toll and slows our application down.

Finally, as always, data shaping should work well together with the concepts we've covered so far – paging, filtering, searching, and sorting.

First, we are going to implement an employee-specific solution to data shaping. Then we are going to make it more generic, so it can be used by any entity or any API.

Let's get to work.

How to Implement Data Shaping

First things first, we need to extend our `RequestParameters` class since we are going to add a new feature to our query string and we want it to be available for any entity:

```
public string Fields { get; set; }
```

We've added the `Fields` property and now we can use fields as a query string parameter.

Let's continue by creating a new interface in the `Contracts` project:

```
public interface IDataShaper<T>
{
    IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities, string
fieldsString);
    ExpandoObject ShapeData(T entity, string fieldsString);
}
```

The `IDataShaper` defines two methods that should be implemented — one for the single entity and one for the collection of entities. Both are named `ShapeData`, but they have different signatures.

Notice how we use the `ExpandoObject` as a return type. We need to do that in order to shape our data the way we want it.

To implement this interface, we are going to create the new folder `DataShaping` in the `Repository` project and the new class `DataShaper`:

```
public class DataShaper<T> : IDataShaper<T> where T : class
{
    public PropertyInfo[] Properties { get; set; }

    public DataShaper()
    {
        Properties = typeof(T).GetProperties(BindingFlags.Public |
BindingFlags.Instance);
    }

    public IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities, string
fieldsString)
    {
         var requiredProperties = GetRequiredProperties(fieldsString);
```

```
         return FetchData(entities, requiredProperties);
    }

    public ExpandoObject ShapeData(T entity, string fieldsString)
    {
        var requiredProperties = GetRequiredProperties(fieldsString);

        return FetchDataForEntity(entity, requiredProperties);
    }

    private IEnumerable<PropertyInfo> GetRequiredProperties(string fieldsString)
    {
        var requiredProperties = new List<PropertyInfo>();

        if (!string.IsNullOrWhiteSpace(fieldsString))
        {
            var fields = fieldsString.Split(',',
StringSplitOptions.RemoveEmptyEntries);

            foreach (var field in fields)
            {
                var property = Properties
                    .FirstOrDefault(pi => pi.Name.Equals(field.Trim(),
StringComparison.InvariantCultureIgnoreCase));

                if (property == null)
                    continue;

                requiredProperties.Add(property);
            }
        }
        else
        {
            requiredProperties = Properties.ToList();
        }

        return requiredProperties;
    }

    private IEnumerable<ExpandoObject> FetchData(IEnumerable<T> entities,
IEnumerable<PropertyInfo> requiredProperties)
    {
        var shapedData = new List<ExpandoObject>();

        foreach (var entity in entities)
        {
            var shapedObject = FetchDataForEntity(entity, requiredProperties);
            shapedData.Add(shapedObject);
        }

        return shapedData;
    }

    private ExpandoObject FetchDataForEntity(T entity, IEnumerable<PropertyInfo>
requiredProperties)
    {
        var shapedObject = new ExpandoObject();
```

```
        foreach (var property in requiredProperties)
        {
            var objectPropertyValue = property.GetValue(entity);
            shapedObject.TryAdd(property.Name, objectPropertyValue);
        }

        return shapedObject;
    }
}
```

There is quite a lot of code here, so let's break it down.

Step-by-Step Implementation

We have one public property in this class – `Properties`. It's an array of PropertyInfo's that we're going to pull out of the input type, whatever it is — Company or Employee in our case:

```
public PropertyInfo[] Properties { get; set; }

public DataShaper()
{
    Properties = typeof(T).GetProperties(BindingFlags.Public | BindingFlags.Instance);
}
```

So, here it is. In the constructor, we get all the properties of an input class.

Next, we have the implementation of our two public `ShapeData` methods:

```
public IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities, string
fieldsString)
{
    var requiredProperties = GetRequiredProperties(fieldsString);

    return FetchData(entities, requiredProperties);
}

public ExpandoObject ShapeData(T entity, string fieldsString)
{
    var requiredProperties = GetRequiredProperties(fieldsString);

    return FetchDataForEntity(entity, requiredProperties);
}
```

Both methods rely on the `GetRequiredProperties` method to parse the input string that contains the fields we want to fetch.

The `GetRequiredProperties` method does the magic. It parses the input string and returns just the properties we need to return to the controller:

```
private IEnumerable<PropertyInfo> GetRequiredProperties(string fieldsString)
{
    var requiredProperties = new List<PropertyInfo>();

    if (!string.IsNullOrWhiteSpace(fieldsString))
    {
        var fields = fieldsString.Split(',', StringSplitOptions.RemoveEmptyEntries);

        foreach (var field in fields)
        {
            var property = Properties
              .FirstOrDefault(pi => pi.Name.Equals(field.Trim(),
StringComparison.InvariantCultureIgnoreCase));

            if (property == null)
                continue;

            requiredProperties.Add(property);
        }
    }
    else
    {
        requiredProperties = Properties.ToList();
    }

    return requiredProperties;
}
```

As you can see, there's nothing special about it. If the `fieldsString` is not empty, we split it and check if the fields match the properties in our entity. If they do, we add them to the list of required properties.

On the other hand, if the `fieldsString` is empty, all properties are required.

Now, `FetchData` and `FetchDataForEntity` are the private methods to extract the values from these required properties we've prepared.

The `FetchDataForEntity` method does it for a single entity:

```
private ExpandoObject FetchDataForEntity(T entity, IEnumerable<PropertyInfo>
requiredProperties)
{
    var shapedObject = new ExpandoObject();

    foreach (var property in requiredProperties)
    {
```

```
            var objectPropertyValue = property.GetValue(entity);
            shapedObject.TryAdd(property.Name, objectPropertyValue);
        }

        return shapedObject;

    }
```

As you can see, we loop through the `requiredProperties`. Then, using a bit of reflection, we extract the values and add them to our ExpandoObject. ExpandoObject implements `IDictionary<string,object>`, so we can use the `TryAdd` method to add our property using its name as a key and the value as a value for the dictionary.

This way, we dynamically add just the properties we need to our dynamic object.

The `FetchData` method is just an implementation for multiple objects. It utilizes the `FetchDataForEntity` method we've just implemented:

```
private IEnumerable<ExpandoObject> FetchData(IEnumerable<T> entities,
IEnumerable<PropertyInfo> requiredProperties)
{
    var shapedData = new List<ExpandoObject>();

    foreach (var entity in entities)
    {
        var shapedObject = FetchDataForEntity(entity, requiredProperties);
        shapedData.Add(shapedObject);
    }

    return shapedData;
}
```

To continue, let's register the `DataShaper` class in the `IServiceCollection` in the `ConfigureServices` method:

```
services.AddScoped <IDataShaper<EmployeeDto>, DataShaper<EmployeeDto>>();
```

As you can see, during the registration, we provide the type to work with.

Finally, we can modify the `EmployeesController` by modifying the constructor:

```
private readonly IDataShaper<EmployeeDto> _dataShaper;

public EmployeesController(IRepositoryManager repository, ILoggerManager logger,
IMapper mapper, IDataShaper<EmployeeDto> dataShaper)
{
    _repository = repository;
    _logger = logger;
    _mapper = mapper;
    _dataShaper = dataShaper;
}
```

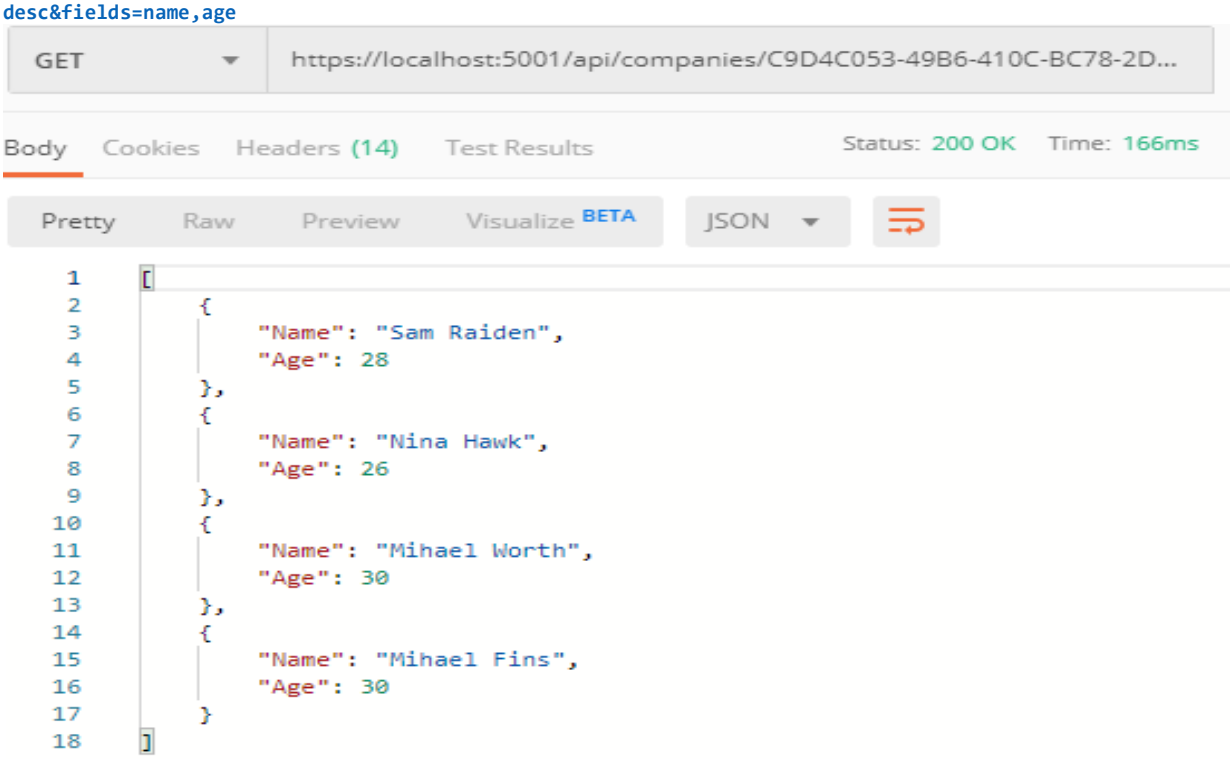
We are injecting it inside the controller because we don't have a service layer in this app. We could have created it, but it would be an overhead for the app this size. But for bigger apps, we recommend creating a service layer and transferring all the mappings and data shaping logic inside it.

And the return statement of the **GetEmployeesForCompany** actions:

```
return Ok(_dataShaper.ShapeData(employeesDto, employeeParameters.Fields));
```

Now, we can test our solution:

https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?pageNumber=1&pageSize=4&minAge=26&maxAge=32&searchTerm=A&orderBy=name desc&fields=name,age




```
[
    {
        "Name": "Sam Raiden",
        "Age": 28
    },
    {
        "Name": "Nina Hawk",
        "Age": 26
    },
    {
        "Name": "Mihael Worth",
        "Age": 30
    },
    {
        "Name": "Mihael Fins",
        "Age": 30
    }
]
```

Excellent. Everything is working like a charm.

Resolving XML Serialization Problems

Let's send the same request one more time, but this time with the different accept header (text/xml):

```
<ArrayOfArrayOfKeyValueOfstringanyType xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://s
    <ArrayOfKeyValueOfstringanyType>
        <KeyValueOfstringanyType>
            <Key>Name</Key>
            <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:string">Sam Raiden</Value>
        </KeyValueOfstringanyType>
        <KeyValueOfstringanyType>
            <Key>Age</Key>
            <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:int">26</Value>
        </KeyValueOfstringanyType>
    </ArrayOfKeyValueOfstringanyType>
    <ArrayOfKeyValueOfstringanyType>
        <KeyValueOfstringanyType>
            <Key>Name</Key>
            <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:string">Nina Hawk</Value>
        </KeyValueOfstringanyType>
        <KeyValueOfstringanyType>
            <Key>Age</Key>
            <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:int">26</Value>
        </KeyValueOfstringanyType>
    </ArrayOfKeyValueOfstringanyType>
```

As you can see, it works — but it looks pretty ugly and unreadable. But that's how the `XmlDataContractSerializerOutputFormatter` serializes our `ExpandoObject` by default.

We can fix that, but the logic is out of the scope of this book. Of course, we have implemented the solution in our source code. So, if you want, you can use it in your project.

All you have to do is to create the `Entity` class and copy the content from our `Entity` class that resides in the `Entities/Models` folder.

After that, just modify the `IDataShaper` interface and the `DataShaper` class by using the `Entity` type instead of the `ExpandoObject` type. Again, you can check our implementation if you have any problems.

After all those changes, once we send the same request, we are going to see a much better result:

```
https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?pageNumber=1&pageSize=4&minAge=26&maxAge=32&searchTerm=A&orderBy=name desc&fields=name,age
```

GET https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-

Pretty Raw Preview Visualize BETA XML

```
1  <ArrayOfEntity xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
2      <Entity>
3          <Name>Sam Raiden</Name>
4          <Age>28</Age>
5      </Entity>
6      <Entity>
7          <Name>Nina Hawk</Name>
8          <Age>26</Age>
9      </Entity>
10     <Entity>
11         <Name>Mihael Worth</Name>
12         <Age>30</Age>
13     </Entity>
14     <Entity>
15         <Name>Mihael Fins</Name>
16         <Age>30</Age>
17     </Entity>
18 </ArrayOfEntity>
```

If XML serialization is not important to you, you can keep using `ExpandoObject` — but if you want a nicely formatted XML response, this is the way to go.

As you can see, data shaping is an exciting and neat little feature that can really make our APIs flexible and reduce our network traffic. If we have a high-volume traffic API, data shaping should work just fine. On the other hand, it's not a feature that we should use lightly because it utilizes reflection and dynamic typing to get things done.

As with all other functionalities, we need to be careful when and if we should implement data shaping. Performance tests might come in handy even if we do implement it.

SUPPORTING HATEOAS

In this section, we are going to talk about one of the most important concepts in building RESTful APIs — HATEOAS and learn how to implement HATEOAS in ASP.NET Core Web API. This part relies heavily on the concepts we've implemented so far in paging, filtering, searching, sorting, and especially data shaping and builds upon the foundations we've put down in these parts.

What is HATEOAS and Why is it so Important?

HATEOAS (Hypermedia as the Engine of Application State) is a very important REST constraint. Without it, a REST API cannot be considered RESTful and many of the benefits we get by implementing a REST architecture are unavailable.

Hypermedia refers to any kind of content that contains links to media types such as documents, images, videos, etc.

REST architecture allows us to generate hypermedia links in our responses dynamically and thus make navigation much easier. To put this into perspective, think about a website that uses hyperlinks to help you navigate to different parts of it. You can achieve the same effect with HATEOAS in your REST API.

Imagine a website that has a home page and you land on it, but there are no links anywhere. You need to scrape the website or find some other way to navigate it to get to the content you want. We're not saying that the website is the same as a REST API, but you get the point.

The power of being able to explore an API on your own can be very useful.

Let's see how that works.

Typical Response with HATEOAS Implemented

Once we implement HATEOAS in our API, we are going to have this type of response:


```
"value": [
    {
        "Name": "Sam Raiden",
        "Age": 26,
        "Links": [
            {
                "href": "https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/80abbca8-664d-4b20-b5de-024705497d4a?fields=name,age",
                "rel": "self",
                "method": "GET"
            },
            {
                "href": "https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/80abbca8-664d-4b20-b5de-024705497d4a",
                "rel": "delete_employee",
                "method": "DELETE"
            },
            {
                "href": "https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/80abbca8-664d-4b20-b5de-024705497d4a",
                "rel": "update_employee",
                "method": "PUT"
            },
            {
                "href": "https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/80abbca8-664d-4b20-b5de-024705497d4a",
                "rel": "partially_update_employee",
                "method": "PATCH"
            }
        ]
    },  ...
```


As you can see, we got the list of our employees and for each employee all the actions we can perform on them. And so on...

So, it's a nice way to make an API self-discoverable and evolvable.

What is a Link?

According to RFC5988, a link is "a typed connection between two resources that are identified by Internationalised Resource Identifiers (IRIs)". Simply put, we use links to traverse the internet or rather the resources on the internet.

Our responses contain an array of links, which consist of a few properties according to the RFC:

- href - represents a target URI.
- rel - represents a link relation type, which means it describes how the current context is related to the target resource.
- method - we need an HTTP method to know how to distinguish the same target URIs.

Pros/Cons of Implementing HATEOAS

So, what are all the benefits we can expect when implementing HATEOAS?

HATEOAS is not trivial to implement, but the rewards we reap are worth it. Here are the things we can expect to get when we implement HATEOAS:

- API becomes self-discoverable and explorable.
- A client can use the links to implement its logic, it becomes much easier, and any changes that happen in the API structure are directly reflected onto the client.
- The server drives the application state and URL structure and not vice versa.
- The link relations can be used to point to developer documentation.
- Versioning through hyperlinks becomes easier.
- Reduced invalid state transaction calls.
- API is evolvable without breaking all the clients.

We can do so much with HATEOAS. But since it's not easy to implement all these features, we should keep in mind the scope of our API and if we need all this. There is a great difference between a high volume public API and some internal API that is needed to communicate between parts of the same system.

That is more than enough theory for now. Let's get to work and see what the concrete implementation of HATEOAS looks like.

Adding Links in the Project

Let’s begin with the concept we know so far, and that’s the link. In the `Entities` project, we are going to create the `LinkModels` folder and inside a new `Link` class:

```
public class Link
{
    public string Href { get; set; }
    public string Rel { get; set; }
    public string Method { get; set; }

    public Link()
    { }

    public Link(string href, string rel, string method)
    {
        Href = href;
        Rel = rel;
        Method = method;
    }
}
```

Note that we have an empty constructor, too. We'll need that for XML serialization purposes, so keep it that way.

Next, we need to create a class that will contain all of our links — **LinkResourceBase**:

```
public class LinkResourceBase
{
    public LinkResourceBase()
    {}

    public List<Link> Links { get; set; } = new List<Link>();
}
```

And finally, since our response needs to describe the root of the controller, we need a wrapper for our links:

```
public class LinkCollectionWrapper<T> : LinkResourceBase
{
    public List<T> Value { get; set; } = new List<T>();

    public LinkCollectionWrapper()
    {}

    public LinkCollectionWrapper(List<T> value)
    {
        Value = value;
    }
}
```

This class might not make too much sense right now, but stay with us and it will become clear later down the road. For now, let's just assume we wrapped our links in another class for response representation purposes.

Since our response will contain links too, we need to extend the XML serialization rules so that our XML response returns the properly formatted links. Without this, we would get something like: `<Links>System.Collections.Generic.List`1[Entites.Models.Link]<Links>`. So, in the `Entities/Models/Entity` class, we need to extend the `WriteLinksToXml` method to support links:

```
private void WriteLinksToXml(string key, object value, XmlWriter writer)
{
    writer.WriteStartElement(key);

    if (value.GetType() == typeof(List<Link>))
    {
        foreach (var val in value as List<Link>)
        {
            writer.WriteStartElement(nameof(Link));
            WriteLinksToXml(nameof(val.Href), val.Href, writer);
            WriteLinksToXml(nameof(val.Method), val.Method, writer);
            WriteLinksToXml(nameof(val.Rel), val.Rel, writer);
            writer.WriteEndElement();
        }
    }
    else
    {
        writer.WriteString(value.ToString());
    }

    writer.WriteEndElement();
}
```

So, we check if the type is `List<Link>`. If it is, we iterate through all the links and call the method recursively for each of the properties: href, method, and rel.

That's all we need for now. We have a solid foundation to implement HATEOAS in our controllers.

Additional Project Changes

When we generate links, HATEOAS strongly relies on having the ids available to construct the links for the response. Data shaping, on the other hand, enables us to return only the fields we want. So, if we want only the `name` and `age` fields, the `id` field won't be added. To solve that, we have to apply some changes.

The first thing we are going to do is to add a `ShapedEntity` class in the `Entities/Models` folder:

```
public class ShapedEntity
{
    public ShapedEntity()
    {
        Entity = new Entity();
    }

    public Guid Id { get; set; }
    public Entity Entity { get; set; }
}
```

With this class, we expose the `Entity` and the `Id` property as well.

Now, we have to modify the `IDataShaper` interface and the `DataShaper` class by replacing all `Entity` usage with `ShapedEntity`.

In addition to that, we need to extend the `FetchDataForEntity` method in the `DataShaper` class to get the `id` separately:

```
private ShapedEntity FetchDataForEntity(T entity, IEnumerable<PropertyInfo>
requiredProperties)
{
    var shapedObject = new ShapedEntity();

    foreach (var property in requiredProperties)
    {
        var objectPropertyValue = property.GetValue(entity);
        shapedObject.Entity.TryAdd(property.Name, objectPropertyValue);
    }

    var objectProperty = entity.GetType().GetProperty("Id");
    shapedObject.Id = (Guid)objectProperty.GetValue(entity);

    return shapedObject;
}
```

Finally, let's add the `LinkResponse` class in the `LinkModels` folder; that will help us with the response once we start with the HATEOAS implementation:

```
public class LinkResponse
{
    public bool HasLinks { get; set; }

    public List<Entity> ShapedEntities { get; set; }
```

```
    public LinkCollectionWrapper<Entity> LinkedEntities { get; set; }

    public LinkResponse()
    {
        LinkedEntities = new LinkCollectionWrapper<Entity>();
        ShapedEntities = new List<Entity>();
    }
}
```

With this class, we are going to know whether our response has links. If it does, we are going to use the `LinkedEntities` property. Otherwise, we are going to use the `ShapedEntities` property.

Adding Custom Media Types

What we want to do is to enable links in our response only if it is explicitly asked for. To do that, we are going to introduce custom media types.

Before we start, let's see how we can create a custom media type. A custom media type should look something like this: `application/vnd.codemaze.hateoas+json`. To compare it to the typical json media type which we use by default: `application/json`.

So let's break down the different parts of a custom media type:

- vnd – vendor prefix; it's always there.
- codemaze – vendor identifier; we've chosen codemaze, because why not?
- hateoas – media type name.
- json – suffix; we can use it to describe if we want json or an XML response, for example.

Now, let's implement that in our application.

Registering Custom Media Types

First, we want to register our new custom media types in the middleware. Otherwise, we'll just get a `406 Not Acceptable` message.

Let's add a new extension method to our `ServiceExtensions`:

```
public static void AddCustomMediaTypes(this IServiceCollection services)
{
    services.Configure<MvcOptions>(config =>
    {
        var newtonsoftJsonOutputFormatter = config.OutputFormatters
            .OfType<NewtonsoftJsonOutputFormatter>()?.FirstOrDefault();

        if (newtonsoftJsonOutputFormatter != null)
        {
            newtonsoftJsonOutputFormatter
              .SupportedMediaTypes
              .Add("application/vnd.codemaze.hateoas+json");
        }

        var xmlOutputFormatter = config.OutputFormatters
.OfType<XmlDataContractSerializerOutputFormatter>()?.FirstOrDefault();

        if (xmlOutputFormatter != null)
        {
            xmlOutputFormatter
              .SupportedMediaTypes
              .Add("application/vnd.codemaze.hateoas+xml");
        }
    });
}
```

We are registering two new custom media types for the JSON and XML output formatters. This ensures we don't get a 406 Not Acceptable response.

Add that to the `Startup.cs` class in the `ConfigureServices` method, just after the `AddControllers` method:

```
services.AddCustomMediaTypes();
```

Excellent. The registration process is done.

Implementing a Media Type Validation Filter

Now, since we've implemented custom media types, we want our Accept header to be present in our requests so we can detect when the user requested the HATEOAS-enriched response.

To do that, we'll implement an ActionFilter which will validate our Accept header and media types:

```
public class ValidateMediaTypeAttribute : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
```

```
    {
        var acceptHeaderPresent =
context.HttpContext.Request.Headers.ContainsKey("Accept");

        if (!acceptHeaderPresent)
        {
            context.Result = new BadRequestObjectResult($"Accept header is missing.");
            return;
        }

        var mediaType =
context.HttpContext.Request.Headers["Accept"].FirstOrDefault();

        if (!MediaTypeHeaderValue.TryParse(mediaType, out MediaTypeHeaderValue
outMediaType))
        {
            context.Result = new BadRequestObjectResult($"Media type not present.
Please add Accept header with the required media type.");
            return;
        }

        context.HttpContext.Items.Add("AcceptHeaderMediaType", outMediaType);
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
    }
}
```

We check for the existence of the Accept header first. If it's not present, we return BadRequest. If it is, we parse the media type — and if there is no valid media type present, we return BadRequest.

Once we've passed the validation checks, we pass the parsed media type to the HttpContext of the controller.

Now, we have to register the filter in the ConfigureServices method:

```
services.AddScoped<ValidateMediaTypeAttribute>();
```

And to decorate the **GetEmployeesForCompany** action:

```
[HttpGet]
[ServiceFilter(typeof(ValidateMediaTypeAttribute))]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId, [FromQuery]
EmployeeParameters employeeParameters)
```

Great job.

Finally, we can work on the HATEOAS implementation.

Implementing HATEOAS

We are going to start by creating a new **Utility** folder in the main project and the **EmployeeLinks** class in it. Let's start by adding the required dependencies inside the class:

```
public class EmployeeLinks
{
    private readonly LinkGenerator _linkGenerator;
    private readonly IDataShaper<EmployeeDto> _dataShaper;

    public EmployeeLinks(LinkGenerator linkGenerator, IDataShaper<EmployeeDto>
dataShaper)
    {
        _linkGenerator = linkGenerator;
        _dataShaper = dataShaper;
    }

}
```

We are going to use **LinkGenerator** to generate links for our responses and **IDataShaper** to shape our data. As you can see, the shaping logic is now extracted from the controller.

After dependencies, we are going to add the first method:

```
public LinkResponse TryGenerateLinks(IEnumerable<EmployeeDto> employeesDto, string
fields, Guid companyId, HttpContext httpContext)
{
    var shapedEmployees = ShapeData(employeesDto, fields);

    if (ShouldGenerateLinks(httpContext))
        return ReturnLinkdedEmployees(employeesDto, fields, companyId, httpContext,
shapedEmployees);

    return ReturnShapedEmployees(shapedEmployees);
}
```

So, our method accepts four parameters. The **employeeDto** collection, the **fields** that are going to be used to shape the previous collection, **companyId** because routes to the employee resources contain the Id from the company, and **httpContext** which holds information about media types.

The first thing we do is shape our collection. Then if the `httpContext` contains the required media type, we add links to the response. On the other hand, we just return our shaped data.

Of course, we have to add those not implemented methods:

```
private List<Entity> ShapeData(IEnumerable<EmployeeDto> employeesDto, string fields)
=>
    _dataShaper.ShapeData(employeesDto, fields)
        .Select(e => e.Entity)
        .ToList();
```

The ShapeData method executes data shaping and extracts only the entity part without the Id property.

Let's add two additional methods:

```
private bool ShouldGenerateLinks(HttpContext httpContext)
{
    var mediaType = (MediaTypeHeaderValue)httpContext.Items["AcceptHeaderMediaType"];

    return mediaType.SubTypeWithoutSuffix.EndsWith("hateoas",
StringComparison.InvariantCultureIgnoreCase);
}

private LinkResponse ReturnShapedEmployees(List<Entity> shapedEmployees) => new
LinkResponse { ShapedEntities = shapedEmployees };
```

In the `ShouldGenerateLinks` method, we extract the media type from the `httpContext`. If that media type ends with `hateoas`, the method returns true; otherwise, it returns false. `ReturnShapedEmployees` just returns a new `LinkResponse` with the `ShapedEntities` property populated. By default, the `HasLinks` property is false.

After these methods, we have to add the `ReturnLinkedEmployees` method as well:

```
private LinkResponse ReturnLinkdedEmployees(IEnumerable<EmployeeDto> employeesDto,
string fields, Guid companyId, HttpContext httpContext, List<Entity> shapedEmployees)
{
    var employeeDtoList = employeesDto.ToList();

    for (var index = 0; index < employeeDtoList.Count(); index++)
    {
```

```
        var employeeLinks = CreateLinksForEmployee(httpContext, companyId,
employeeDtoList[index].Id, fields);
        shapedEmployees[index].Add("Links", employeeLinks);
    }

    var employeeCollection = new LinkCollectionWrapper<Entity>(shapedEmployees);
    var linkedEmployees = CreateLinksForEmployees(httpContext, employeeCollection);

    return new LinkResponse { HasLinks = true, LinkedEntities = linkedEmployees };
}
```

As you can see, we iterate through each employee and create links for it by calling the `CreateLinksForEmployee` method. Then, we just add it to the `shapedEmployees` collection. After that, we wrap the collection and create links that are important for the entire collection by calling the `CreateLinksForEmployees` method.

Finally, we have to add those two new methods that create links:

```
private List<Link> CreateLinksForEmployee(HttpContext httpContext, Guid companyId,
Guid id, string fields = "")
{
    var links = new List<Link>
    {
        new Link(_linkGenerator.GetUriByAction(httpContext, "GetEmployeeForCompany",
values: new { companyId, id, fields }),
        "self",
        "GET"),
        new Link(_linkGenerator.GetUriByAction(httpContext,
"DeleteEmployeeForCompany", values: new { companyId, id }),
        "delete_employee",
        "DELETE"),
        new Link(_linkGenerator.GetUriByAction(httpContext,
"UpdateEmployeeForCompany", values: new { companyId, id }),
        "update_employee",
        "PUT"),
        new Link(_linkGenerator.GetUriByAction(httpContext,
"PartiallyUpdateEmployeeForCompany", values: new { companyId, id }),
        "partially_update_employee",
        "PATCH")
    };

    return links;
}

private LinkCollectionWrapper<Entity> CreateLinksForEmployees(HttpContext httpContext,
LinkCollectionWrapper<Entity> employeesWrapper)
{
    employeesWrapper.Links.Add(new Link(_linkGenerator.GetUriByAction(httpContext,
"GetEmployeesForCompany", values: new { }),
    "self",
    "GET"));
```

```
    return employeesWrapper;
}
```

There are a few things to note here.

We need to consider the fields while creating the links, since we might be using it in our requests. We are creating the links by using the LinkGenerator's `GetUriByAction` method — which accepts `HttpContext`, the name of the action, and the values that need to be used to make the URL valid. In the case of the EmployeesController, we send the company id, employee id, and fields.

And that is it regarding this class.

Now, we have to register this class in the `ConfigureServices` method:

```
services.AddScoped<EmployeeLinks>();
```

Once registered, we can inject it in the `EmployeesController`:

```
private readonly IRepositoryManager _repository;
private readonly ILoggerManager _logger;
private readonly IMapper _mapper;
private readonly EmployeeLinks _employeeLinks;

public EmployeesController(IRepositoryManager repository, ILoggerManager logger,
IMapper mapper, EmployeeLinks employeeLinks)
{
    _repository = repository;
    _logger = logger;
    _mapper = mapper;
    _employeeLinks = employeeLinks;
}
```

As you can see, we don't have the DataShaper injected anymore.

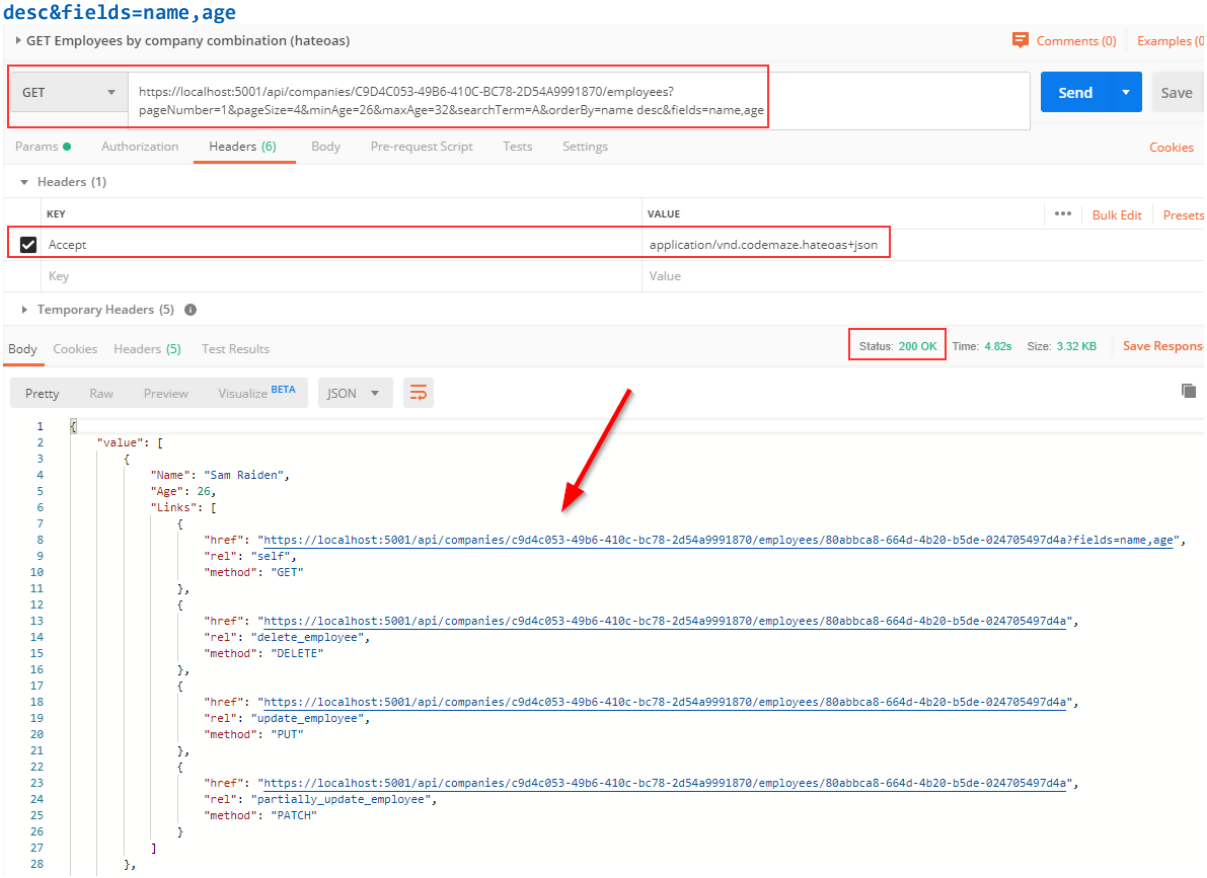
All we have left to do is to slightly modify the `GetEmployeesForCompany` action:

```
[HttpGet]
[ServiceFilter(typeof(ValidateMediaTypeAttribute))]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId, [FromQuery]
EmployeeParameters employeeParameters)
{
    //The first part of the action omitted for the clarity

    var employeesDto = _mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);
```

```
    var links = _employeeLinks.TryGenerateLinks(employeesDto,
employeeParameters.Fields, companyId, HttpContext);

    return links.HasLinks ? Ok(links.LinkedEntities) : Ok(links.ShapedEntities);
}
```

Excellent. We can test this now:

https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?pageNumber=1&pageSize=4&minAge=26&maxAge=32&searchTerm=A&orderBy=name desc&fields=name,age



You can test this with the xml media type as well (we have prepared the request in Postman for you).

WORKING WITH OPTIONS AND HEAD REQUESTS

In one of the previous chapters (Method Safety and Method Idempotency), we talked about different HTTP requests. Until now, we have been working with all request types except OPTIONS and HEAD. So, let's cover them as well.

OPTIONS HTTP Request

The Options request can be used to request information on the communication options available upon a certain URI. It allows consumers to determine the options or different requirements associated with a resource. Additionally, it allows us to check the capabilities of a server without forcing action to retrieve a resource.

Basically, Options should inform us whether we can Get a resource or execute any other action (POST, PUT, or DELETE). All of the options should be returned in the Allow header of the response as a comma-separated list of methods.

Let's see how we can implement the Options request in our example.

OPTIONS Implementation

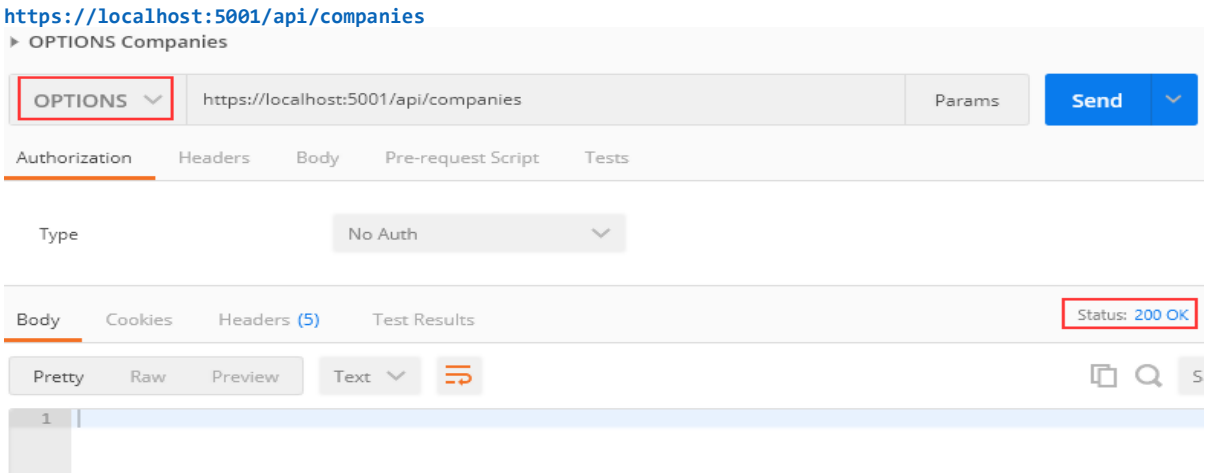
We are going to implement this request in the `CompaniesController` — so, let's open it and add an additional action:

```
[HttpOptions]
public IActionResult GetCompaniesOptions()
{
    Response.Headers.Add("Allow", "GET, OPTIONS, POST");

    return Ok();
}
```

We have to decorate our action with the `HttpOptions` attribute. As we said, the available options should be returned in the `Allow` response header, and that is exactly what we are doing here. The URI for this

action is `/api/companies`, so we state which actions can be executed for that certain URI. Finally, the Options request should return the 200 OK status code. We have to understand that the response, if it is empty, must include the `content-length` field with the value of zero. We don't have to add it by ourselves because ASP.NET Core takes care of that for us.

Let's try this:



As you can see, we are getting a 200 OK response. Let's inspect the Headers tab:

Everything works as expected.

Let's move on.

Head HTTP Request

Head is identical to Get, but without a response body. This type of request could be used to obtain information about validity, accessibility, and recent modifications of the resource.

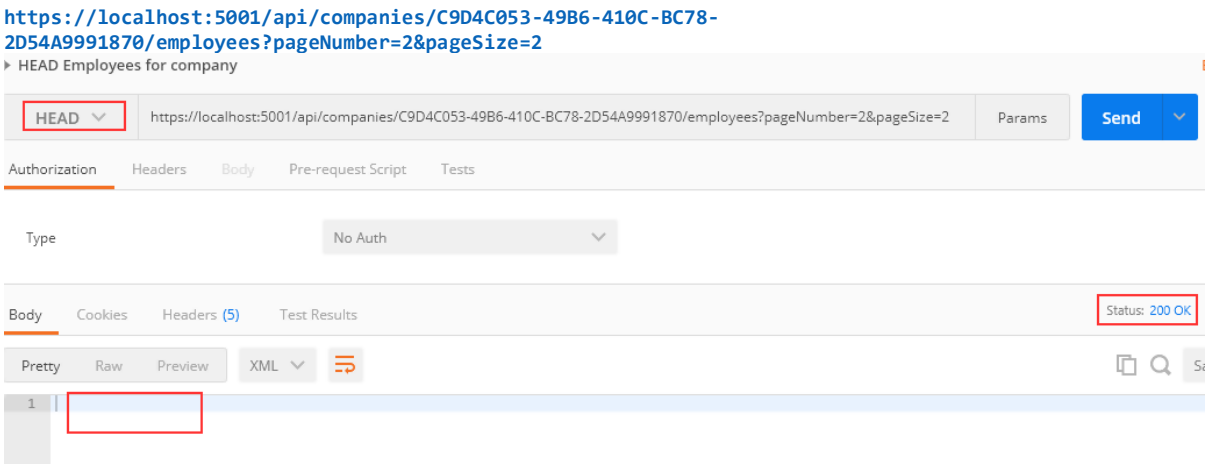
HEAD Implementation

Let's open the `EmployeesController`, because that's where we are going to implement this type of request. As we said, the Head request must return exactly the same response as the Get request — just without the response body. That means it should include the paging information in the response as well.

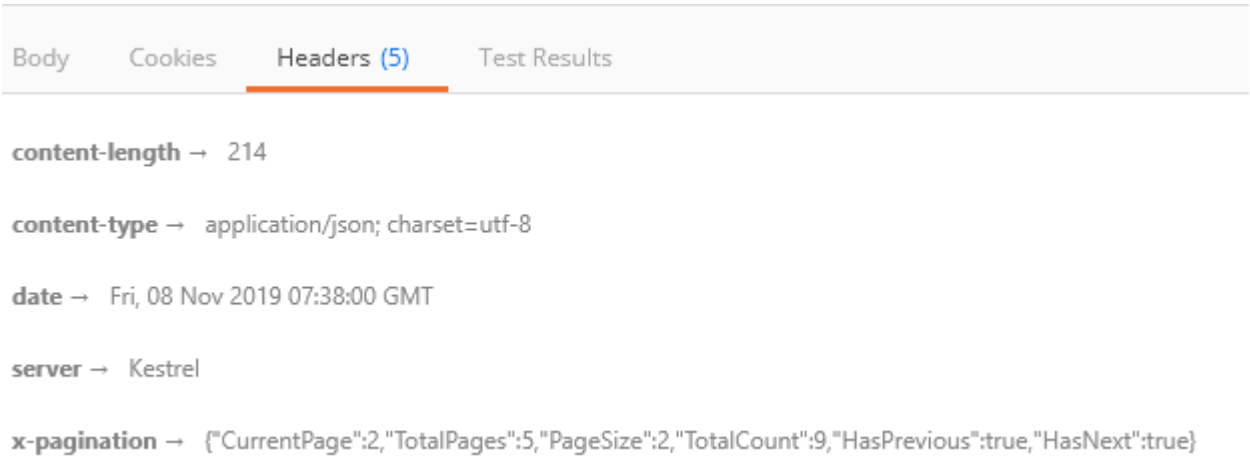
Now, you may think that we have to write a completely new action and also repeat all the code inside, but that is not the case. All we have to do is add the `HttpHead` attribute below `HttpGet`:

```
[HttpGet]
[HttpHead]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId, [FromQuery]
EmployeeParameters employeeParameters)
```

We can test this now:



As you can see, we receive a 200 OK status code with the empty body. Let's check the Headers part:



You can see the x-pagination link included in the Headers part of the response. Additionally, all the parts of the x-pagination link are populated — which means that our code was successfully executed, but the response body hasn't been included.

Excellent.

We now have support for the Http OPTIONS and HEAD requests.

ROOT DOCUMENT

In this section, we are going to create a starting point for the consumers of our API. This starting point is also known as the Root Document. The Root Document is the place where consumers can learn how to interact with the rest of the API.

Root Document Implementation

This document should be created at the api root, so let's start by creating a new controller:

```
[Route("api")]
[ApiController]
public class RootController : ControllerBase
{
}
```

We are going to generate links towards the API actions. Therefore, we have to inject `LinkGenerator`:

```
[Route("api")]
[ApiController]
public class RootController : ControllerBase
{
    private readonly LinkGenerator _linkGenerator;

    public RootController(LinkGenerator linkGenerator)
    {
        _linkGenerator = linkGenerator;
    }
}
```

In this controller, we only need a single action, `GetRoot`, which will be executed with the GET request on the `/api` URI.

There are several links that we are going to create in this action. The link to the document itself and links to actions available on the URIs at the root level (actions from the Companies controller). We are not creating links to employees, because they are children of the company — and in our API if we want to fetch employees, we have to fetch the company first.

If we inspect our **CompaniesController**, we can see that **GetCompanies** and **CreateCompany** are the only actions on the root URI level (api/companies). Therefore, we are going to create links only to them.

Before we start with the **GetRoot** action, let's add a name for the **CreateCompany** and **GetCompanies** actions in the **CompaniesController**:

```
[HttpGet(Name = "GetCompanies")]
public async Task<IActionResult> GetCompanies()
```

```
[HttpPost(Name = "CreateCompany")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> CreateCompany([FromBody]CompanyForCreationDto
company)
```

We are going to use the **Link** class to generate links:

```
public class Link
{
    public string Href { get; set; }
    public string Rel { get; set; }
    public string Method { get; set; }
    …
}
```

This class contains all the required properties to describe our actions while creating links in the **GetRoot** action. The **Href** property defines the URI to the action, the **Rel** property defines the identification of the action type, and the **Method** property defines which HTTP method should be used for that action.

Now, we can create the **GetRoot** action:

```
[HttpGet(Name = "GetRoot")]
public IActionResult GetRoot([FromHeader(Name = "Accept")] string mediaType)
{
    if(mediaType.Contains("application/vnd.codemaze.apiroot"))
    {
        var list = new List<Link>
        {
            new Link
            {
                Href = _linkGenerator.GetUriByName(HttpContext, nameof(GetRoot), new
{}),
                Rel = "self",
                Method = "GET"
```

```
            },
            new Link
            {
                Href = _linkGenerator.GetUriByName(HttpContext, "GetCompanies", new
{}),
                Rel = "companies",
                Method = "GET"
            },
            new Link
            {
                Href = _linkGenerator.GetUriByName(HttpContext, "CreateCompany", new
{}),
                Rel = "create_company",
                Method = "POST"
            }
        };

        return Ok(list);
    }

    return NoContent();
}
```

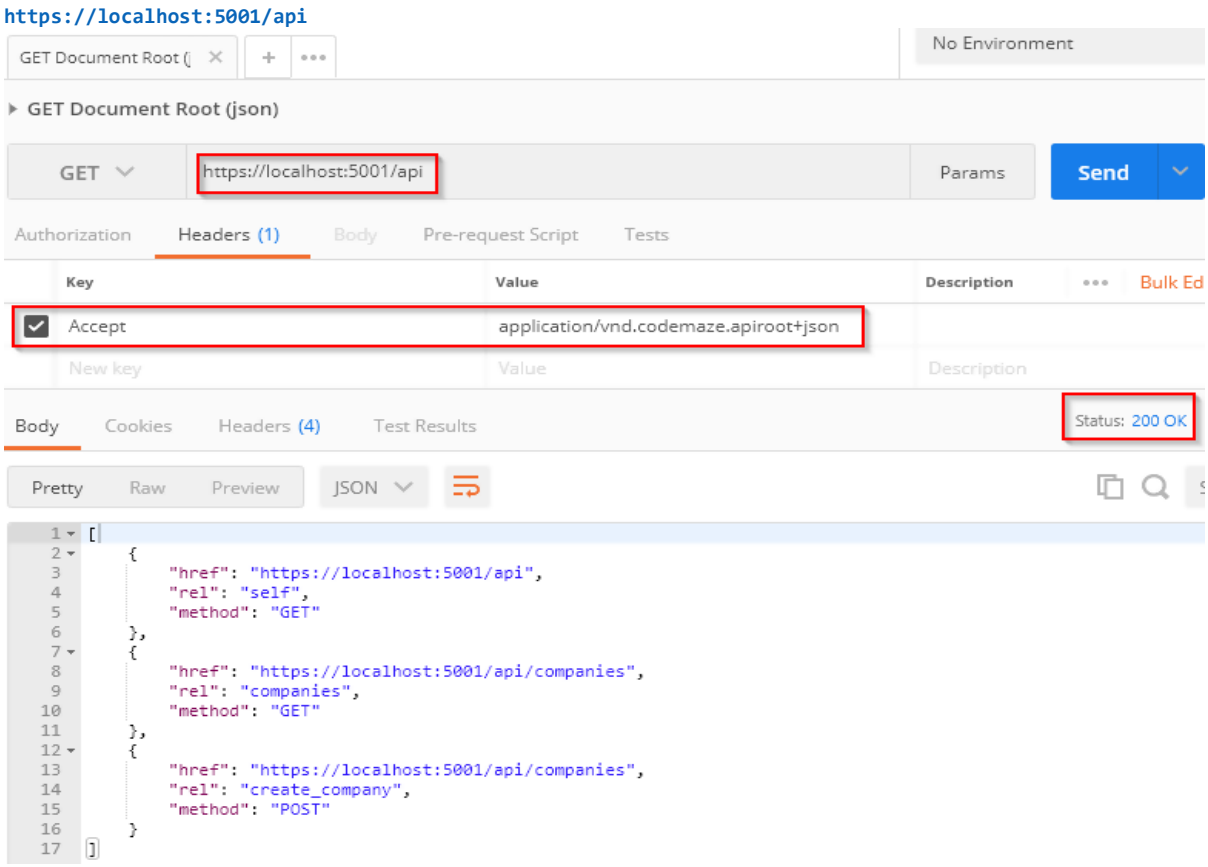
As you can see, we generate links only if a custom media type is provided from the Accept header. Otherwise, we return `NoContent()`. To generate links, we use the `GetUriByName` method from the `LinkGenerator` class.

That said, we have to register our custom media types for the json and xml formats. To do that, we are going to extend the AddCustomMediaTypes extension method:

```
public static void AddCustomMediaTypes(this IServiceCollection services)
{
    services.Configure<MvcOptions>(config =>
    {
        var newtonsoftJsonOutputFormatter = config.OutputFormatters
            .OfType<NewtonsoftJsonOutputFormatter>()?.FirstOrDefault();

        if (newtonsoftJsonOutputFormatter != null)
        {
            newtonsoftJsonOutputFormatter
              .SupportedMediaTypes.Add("application/vnd.codemaze.hateoas+json");
            newtonsoftJsonOutputFormatter
              .SupportedMediaTypes.Add("application/vnd.codemaze.apiroot+json");
        }

        var xmlOutputFormatter = config.OutputFormatters
              .OfType<XmlDataContractSerializerOutputFormatter>()?.FirstOrDefault();

        if (xmlOutputFormatter != null)
        {
            xmlOutputFormatter
              .SupportedMediaTypes.Add("application/vnd.codemaze.hateoas+xml");
```

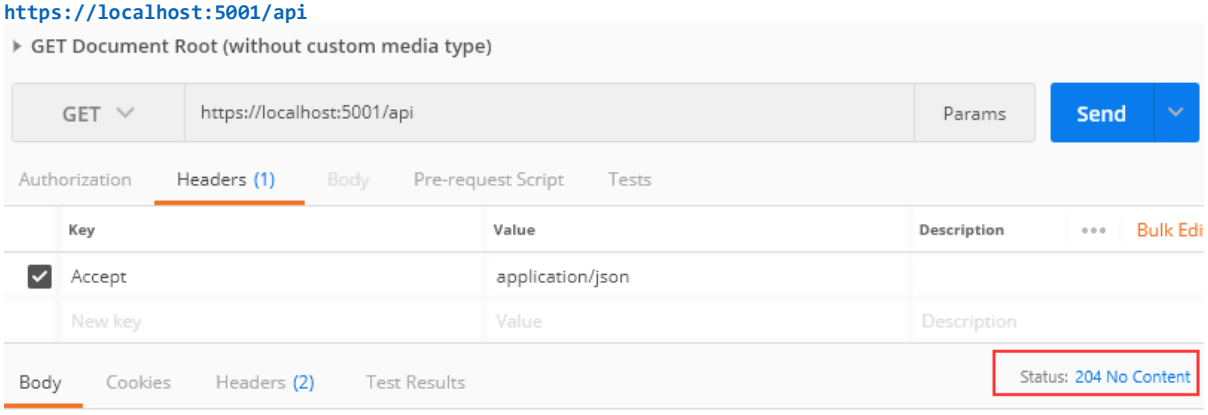
```
            xmlOutputFormatter
              .SupportedMediaTypes.Add("application/vnd.codemaze.apiroot+xml");
        }
    });
}
```

We can now inspect our result:



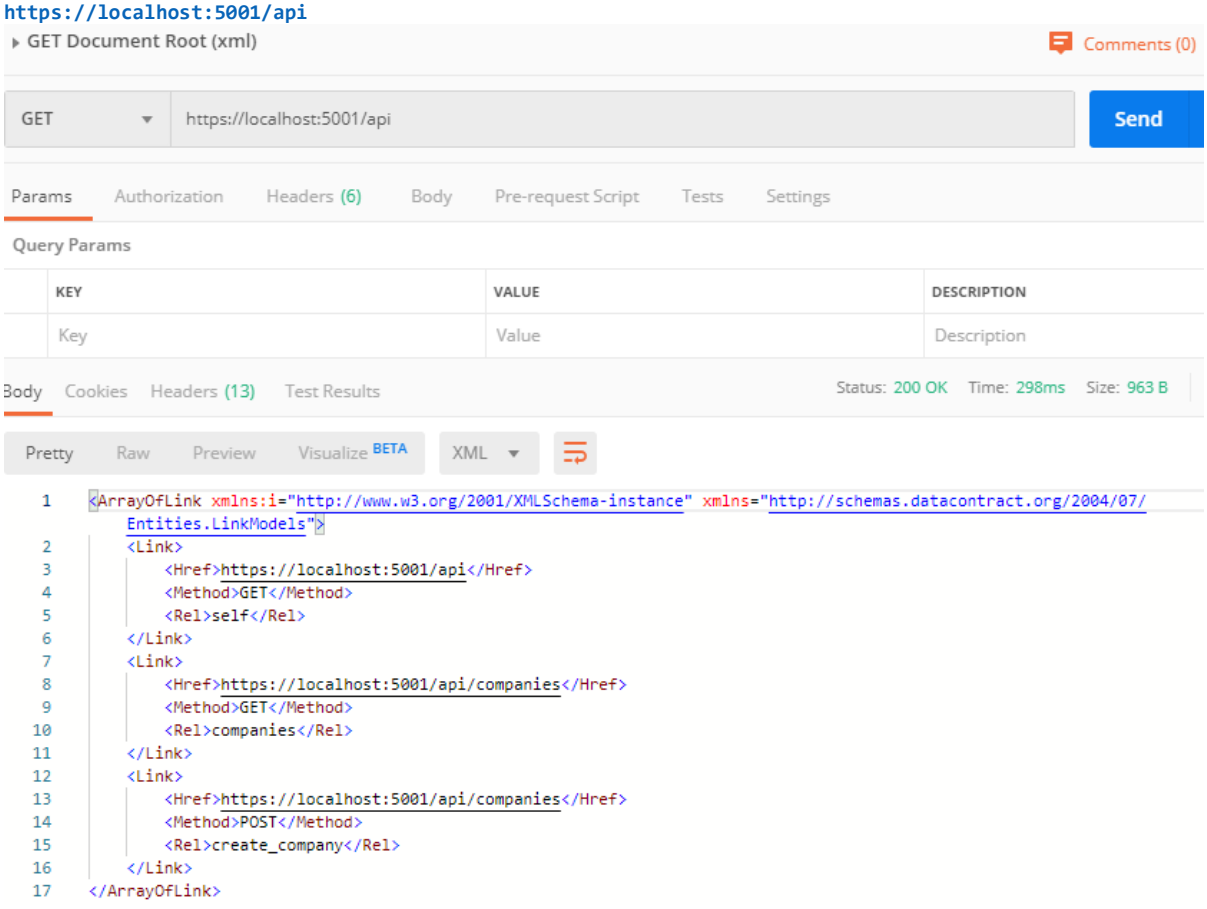
This works great.

Let's test what is going to happen if we don't provide the custom media type:



Well, we get the `204 No Content` message as expected.

Of course, you can test the xml request as well:



VERSIONING APIS

As our project grows, so does our knowledge; therefore, we have a better understanding of how to improve our system. Moreover, requirements change over time — thus, our API has to change as well.

When we implement some breaking changes, we want to ensure that we don't do anything that will cause our API consumers to change their code. Those breaking changes could be:

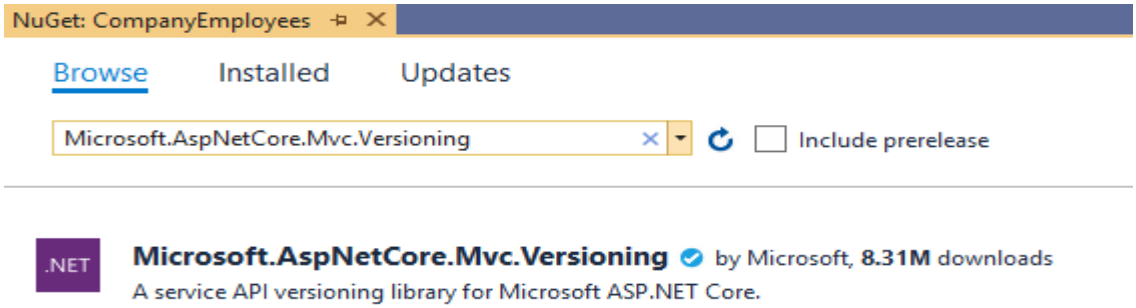
- Renaming fields, properties, or resource URIs.
- Changes in the payload structure.
- Modifying response codes or HTTP Verbs.
- Redesigning our API endpoints.

If we have to implement some of these changes in the already working API, the best way is to apply versioning to prevent breaking our API for the existing API consumers.

There are different ways to achieve API versioning and there is no guidance that favors one way over another. So, we are going to show you different ways to version an API, and you can choose which one suits you best.

Required Package Installation and Configuration

In order to start, we have to install the `Microsoft.AspNetCore.Mvc.Versioning` library in the main project:



This library is going to help us a lot in versioning our API.

After the installation, we have to add the versioning service in the service collection and to configure it. So, let's create a new extension method in the `ServiceExtensions` class:

```
public static void ConfigureVersioning(this IServiceCollection services)
{
    services.AddApiVersioning(opt =>
    {
        opt.ReportApiVersions = true;
        opt.AssumeDefaultVersionWhenUnspecified = true;
        opt.DefaultApiVersion = new ApiVersion(1, 0);
    });
}
```

With the `AddApiVersioning` method, we are adding service API versioning to the service collection. We are also using a couple of properties to initially configure versioning:

- `ReportApiVersions` adds the API version to the response header.
- `AssumeDefaultVersionWhenUnspecified` does exactly that. It specifies the default API version if the client doesn't send one.
- `DefaultApiVersion` sets the default version count.

After that, we are going to use this extension in the `ConfigureServices` method:

```
services.ConfigureVersioning();
```

API versioning is installed and configured, and we can move on.

Versioning Examples

Before we continue, let's create another controller: `CompaniesV2Controller` (for example's sake), which will represent a new version of our existing one. It is going to have just one Get action:

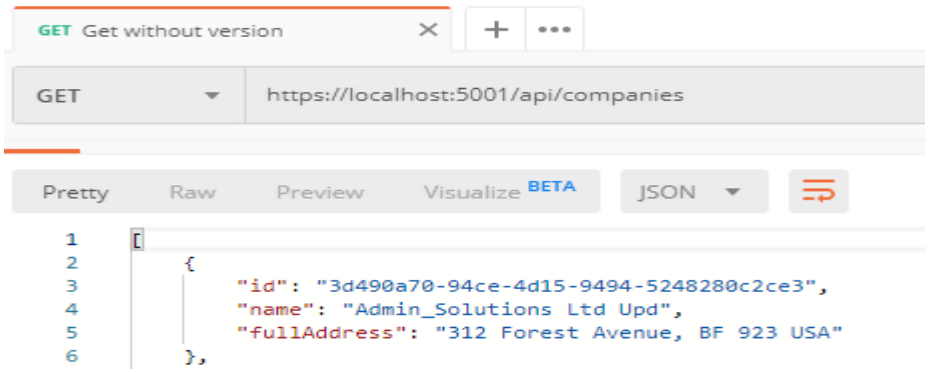
```
[ApiVersion("2.0")]
[Route("api/companies")]
[ApiController]
public class CompaniesV2Controller : ControllerBase
{
    private readonly IRepositoryManager _repository;

    public CompaniesV2Controller(IRepositoryManager repository)
    {
        _repository = repository;
    }

    [HttpGet]
    public async Task<IActionResult> GetCompanies()
    {
        var companies = await _repository.Company.GetAllCompaniesAsync(trackChanges:
false);

        return Ok(companies);
    }
}
```

By using the [ApiVersion("2.0")] attribute, we are stating that this controller is version 2.0. In the Get action, we are not returning a DTO to the client, but we return the entity itself. Let's version our original controller as well:

```
[ApiVersion("1.0")]
[Route("api/companies")]
[ApiController]
public class CompaniesController : ControllerBase
```

If you remember, we configured versioning to use 1.0 as a default API version (`opt.AssumeDefaultVersionWhenUnspecified = true;`). Therefore, if a client doesn't state the required version, our API will use this one:

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



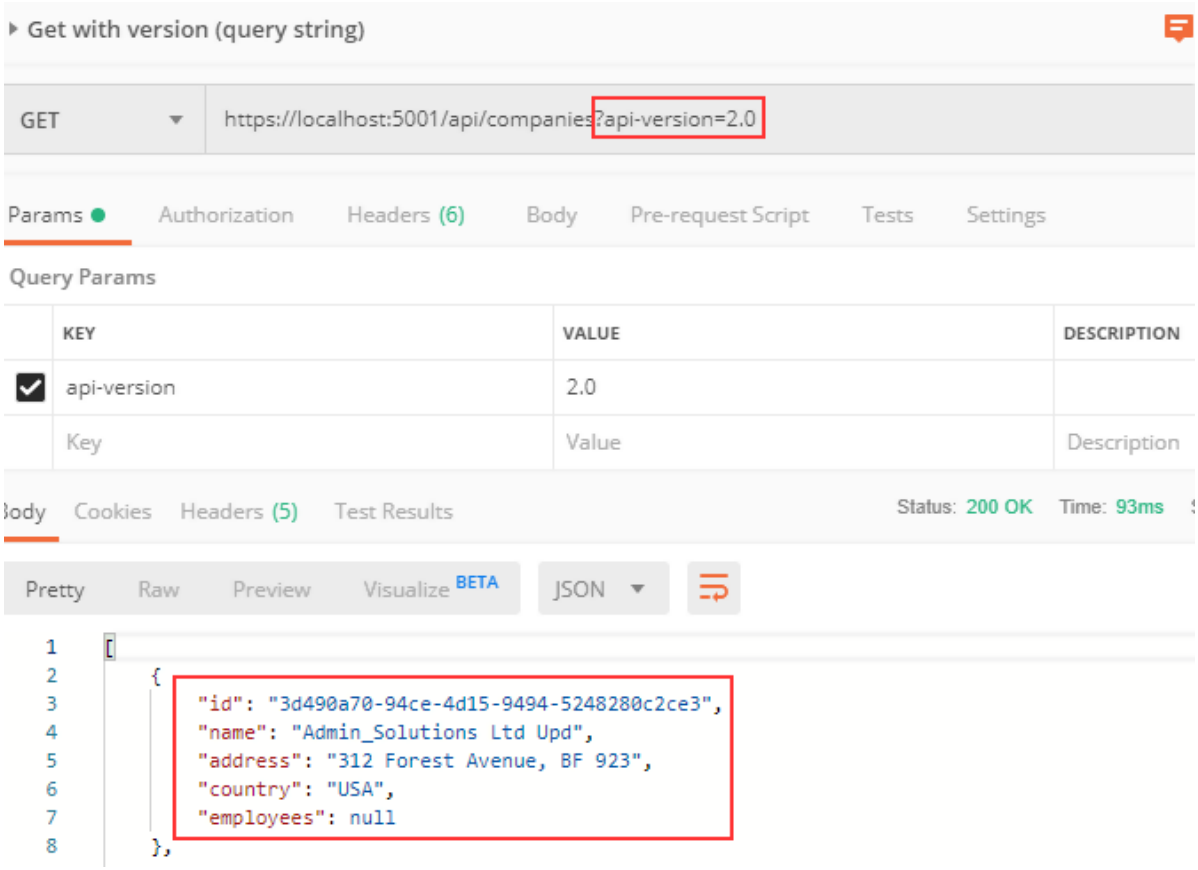
You can see that we have the `fullAddress` property in a result, which means that our original controller was called even though we didn't provide an API version in a request.

Now, let's see how we can provide a version inside the request.

Using Query String

We can provide a version within the request by using a query string in the URI. Let's test this with an example:

```
https://localhost:5001/api/companies?api-version=2.0
```



As you can see, the Company entity is returned as a response body and not CompanyDto. Therefore, we are sure that version 2.0 was called.

Additionally, we can inspect the response headers to make sure that version 2.0 is used:

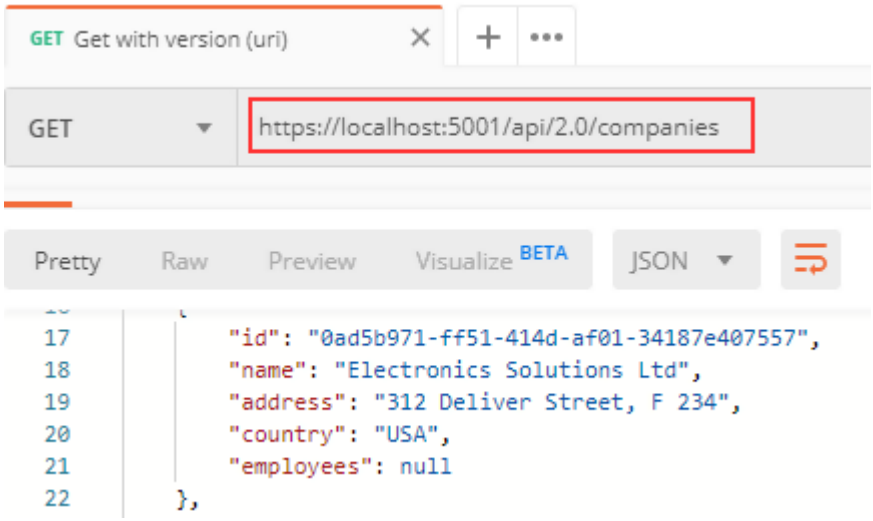
Using URL Versioning

For URL versioning to work, we have to modify the route in our controller:

```
[ApiVersion("2.0")]
[Route("api/{v:apiversion}/companies")]
[ApiController]
public class CompaniesV2Controller : ControllerBase
```

Now, we can test it:

https://localhost:5001/api/2.0/companies



One thing to mention, we can't use the query string pattern to call the companies v2 controller anymore. We can use it for version 1.0, though.

HTTP Header Versioning

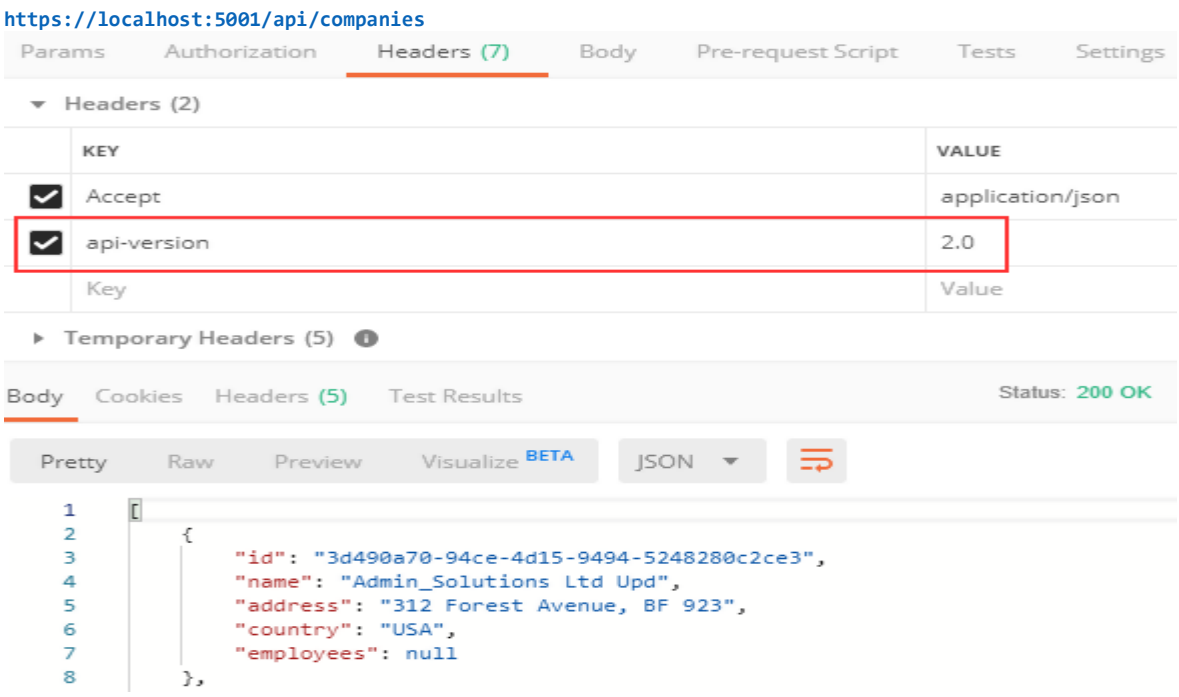
If we don't want to change the URI of the API, we can send the version in the HTTP Header. To enable this, we have to modify our configuration:

```
public static void ConfigureVersioning(this IServiceCollection services)
{
    services.AddApiVersioning(opt =>
    {
        opt.ReportApiVersions = true;
        opt.AssumeDefaultVersionWhenUnspecified = true;
        opt.DefaultApiVersion = new ApiVersion(1, 0);
        opt.ApiVersionReader = new HeaderApiVersionReader("api-version");
    });
}
```

And to revert the Route change in our controller:

```
[ApiVersion("2.0")]
[Route("api/companies")]
```

Let's test these changes:



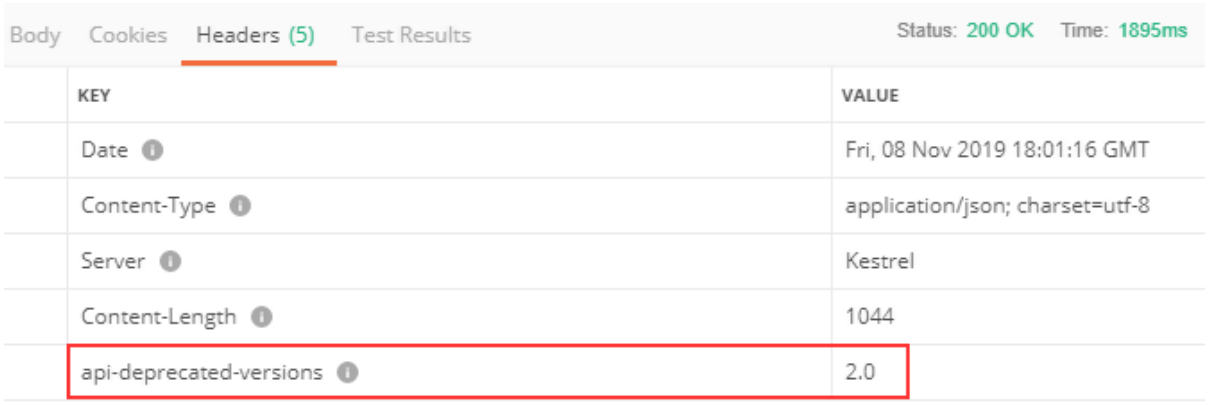
If we want to support query string versioning, we should use a `new QueryApiVersionReader` class instead.

Deprecating Versions

If we want to deprecate version of an API, but don't want to remove it completely, we can use the Deprecated property for that purpose:

```
[ApiVersion("2.0", Deprecated = true)]
```

We will be able to work with that API, but we will be notified that this version is deprecated:



Using Conventions

If we have a lot of versions of a single controller, we can assign these versions in the configuration instead:

```
opt.Conventions.Controller<CompaniesController>().HasApiVersion(new ApiVersion(1, 0));
opt.Conventions.Controller<CompaniesV2Controller>().HasDeprecatedApiVersion(new
ApiVersion(2, 0));
```

Now, we can remove the `[ApiVersion]` attribute from the controllers.

Of course, there are a lot more features that the installed library provides for us — but with the mentioned ones, we have covered quite enough to version our APIs.

CACHING

In this section, we are going to learn about caching resources. Caching can improve the quality and performance of our app a lot, but again, it is something first we need to look at as soon as some bug appears. To cover resource caching, we are going to work with HTTP Cache. Additionally, we are going to talk about cache expiration, validation, and cache-control headers.

About Caching

We want to use cache in our app because it can significantly improve performance. Otherwise, it would be useless. The main goal of caching is to eliminate the need to send requests towards the API in many cases and also to send full responses in other cases.

To reduce the number of sent requests, caching uses the **expiration mechanism**, which helps reduce network round trips. Furthermore, to eliminate the need to send full responses, the cache uses the **validation mechanism**, which reduces network bandwidth. We can now see why these two are so important when caching resources.

The cache is a separate component that accepts requests from the API's consumer. It also accepts the response from the API and stores that response if they are cacheable. Once the response is stored, if a consumer requests the same response again, the response from the cache should be served.

But the cache behaves differently depending on what cache type is used.

Cache Types

There are three types of caches: Client Cache, Gateway Cache, and Proxy Cache.

The client cache lives on the client (browser); thus, it is a private cache. It is private because it is related to a single client. So every client consuming our API has a private cache.

The gateway cache lives on the server and is a shared cache. This cache is shared because the resources it caches are shared over different clients.

The proxy cache is also a shared cache, but it doesn't live on the server nor in the client side. It lives on the network.

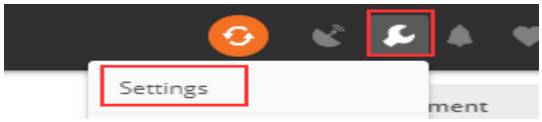
With the private cache, if five clients request the same response for the first time, every response will be served from the API and not from the cache. But if they request the same response again, that response should come from the cache (if it's not expired). This is not the case with the shared cache. The response from the first client is going to be cached, and then the other four clients will receive the cached response if they request it.

Response Cache Attribute

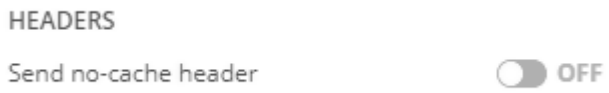
So, to cache some resources, we have to know whether or not it's cacheable. The response header helps us with that. The one that is used most often is Cache-Control: `Cache-Control: max-age=180`. This states that the response should be cached for 180 seconds. For that, we use the `ResponseCache` attribute. But of course, this is just a header. If we want to cache something, we need a cache-store. For our example, we are going to use Response caching middleware provided by ASP.NET Core.

Adding Cache Headers

Before we start, let's open Postman and modify the settings to support caching:

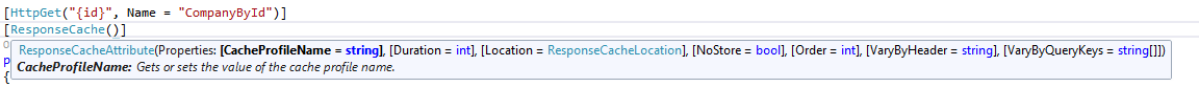


In the General tab under Headers, we are going to turn off the Send no-cache header:



Great. We can move on.

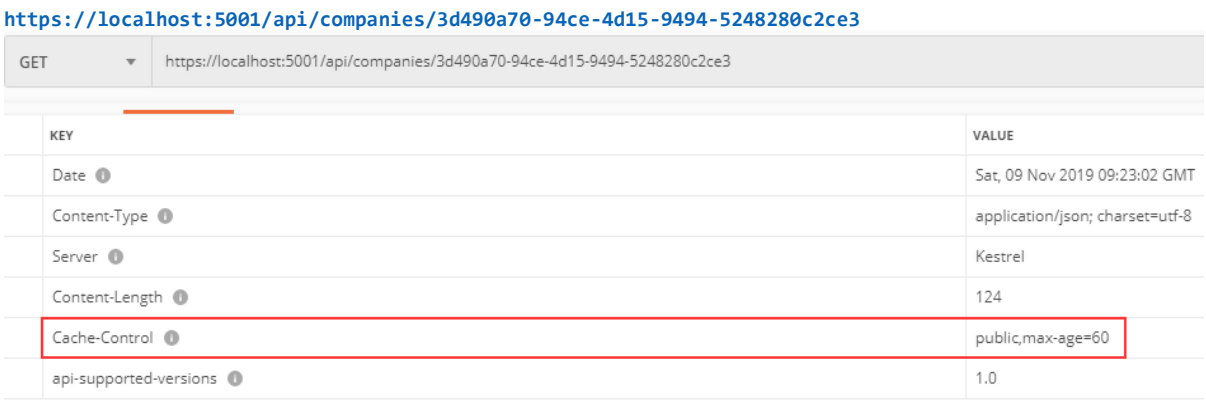
Let's assume we want to cache the result from the `GetCompany` action:



As you can see, we can work with different properties in the ResponseCache attribute — but for now, we are going to use `Duration` only:

```
[HttpGet("{id}", Name = "CompanyById")]
[ResponseCache(Duration = 60)]
public async Task<IActionResult> GetCompany(Guid id)
```

And that is it. We can inspect our result now:



You can see that the Cache-Control header was created with a public cache and a duration of 60 seconds. But as we said, this is just a header; we need a cache-store to cache the response. So, let's add one.

Adding Cache-Store

The first thing we are going to do is add an extension method in the `ServiceExtensions` class:

```
public static void ConfigureResponseCaching(this IServiceCollection services) =>
services.AddResponseCaching();
```

We register response caching in the IOC container, and now we have to call this method in the `ConfigureServices` method:

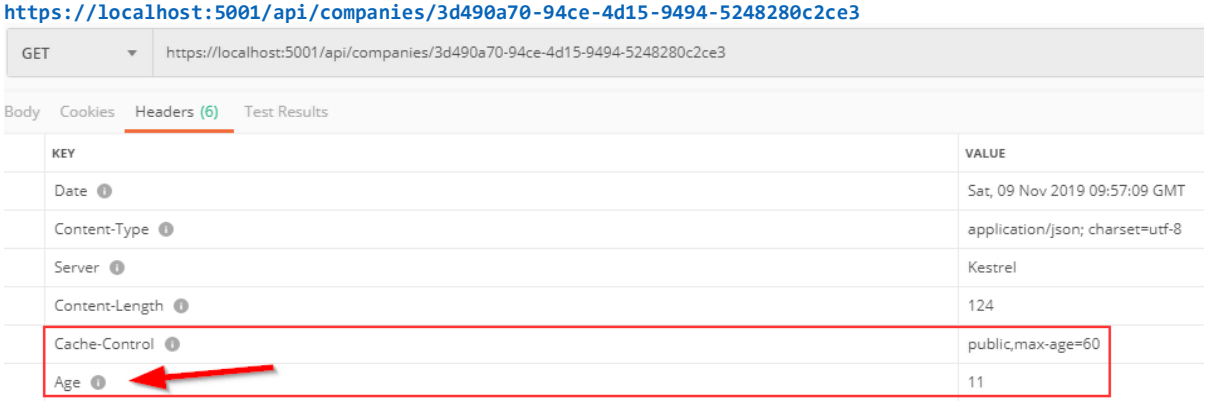
```
services.ConfigureResponseCaching();
```

Additionally, we have to add caching to the application middleware in the `Configure` method right above `UseRouting()`:

```
app.UseResponseCaching();

app.UseRouting();
```

Now, we can start our application and send the same `GetCompany` request. It will generate the Cache-Control header. After that, before 60 seconds pass, we are going to send the same request and inspect the headers:

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



You can see the additional Age header that indicates the number of seconds the object has been stored in the cache. Basically, it means that we received our second response from the cache-store. We can confirm that from the console as well:

```
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'CompanyEmployees.Controllers.CompaniesController.GetCompany (CompanyEmployees)'
info: Microsoft.AspNetCore.ResponseCaching.ResponseCachingMiddleware[26]
      The response has been cached.
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 2149.4242ms 200 application/json; charset=utf-8
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3
info: Microsoft.AspNetCore.ResponseCaching.ResponseCachingMiddleware[22]
      Serving response from cache.
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 16.321ms 200 application/json; charset=utf-8
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3
info: Microsoft.AspNetCore.ResponseCaching.ResponseCachingMiddleware[22]
      Serving response from cache.
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 22.945ms 200 application/json; charset=utf-8
```

If we send several requests within the 60 seconds, the Age property will increment. After the expiration period passes, the response will be sent from the API, cached again, and the Age header will not be generated.

Additionally, we can use cache profiles to apply the same rules to different resources. If you look at the picture that shows all the properties we can use with **ResponseCacheAttribute**, you can see that there are a lot of properties. Configuring all of them on top of the action or controller could lead to less readable code. Therefore, we can use **CacheProfiles** to extract that configuration.

To do that, we are going to modify **AddControllers** in the **ConfigureServices** method:

```
services.AddControllers(config =>
{
    config.RespectBrowserAcceptHeader = true;
    config.ReturnHttpNotAcceptable = true;
    config.CacheProfiles.Add("120SecondsDuration", new CacheProfile { Duration = 120
});
})…
```

We set up only Duration, but you can add additional properties as well. Now, let's implement this profile on top of the Companies controller:

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

```
[ApiController]
[ResponseCache(CacheProfileName = "120SecondsDuration")]
```

We have to mention that this cache rule will apply to all the actions inside the controller except the ones that already have the ResponseCache atribute applied.

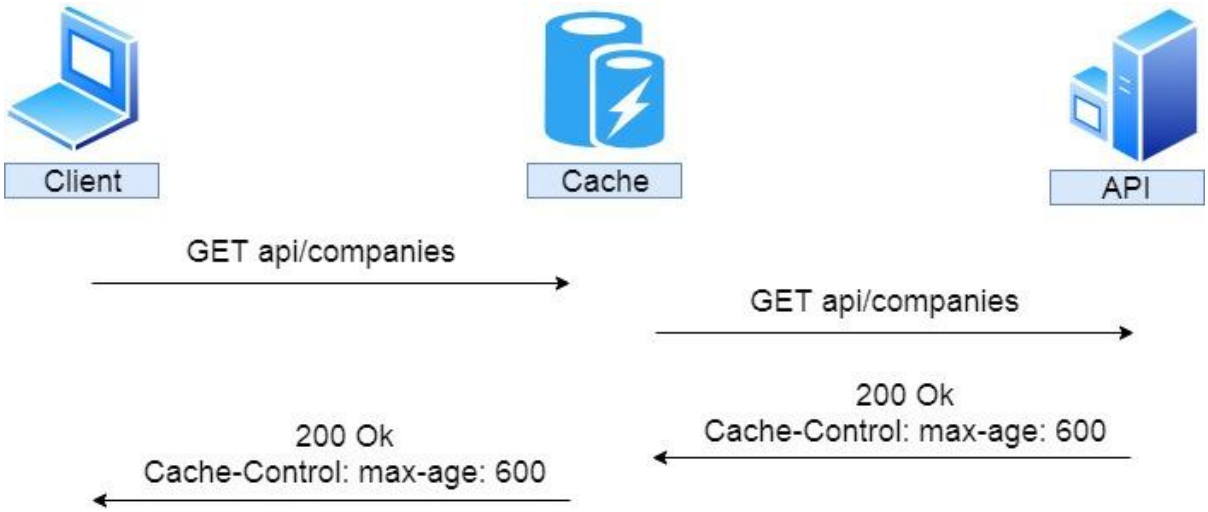
That said, once we send the request to `GetCompany`, we will still have the maximum age of 60. But once we send the request to `GetCompanies`:

There you go. Now, let's talk some more about the Expiration and Validation models.

Expiration Model

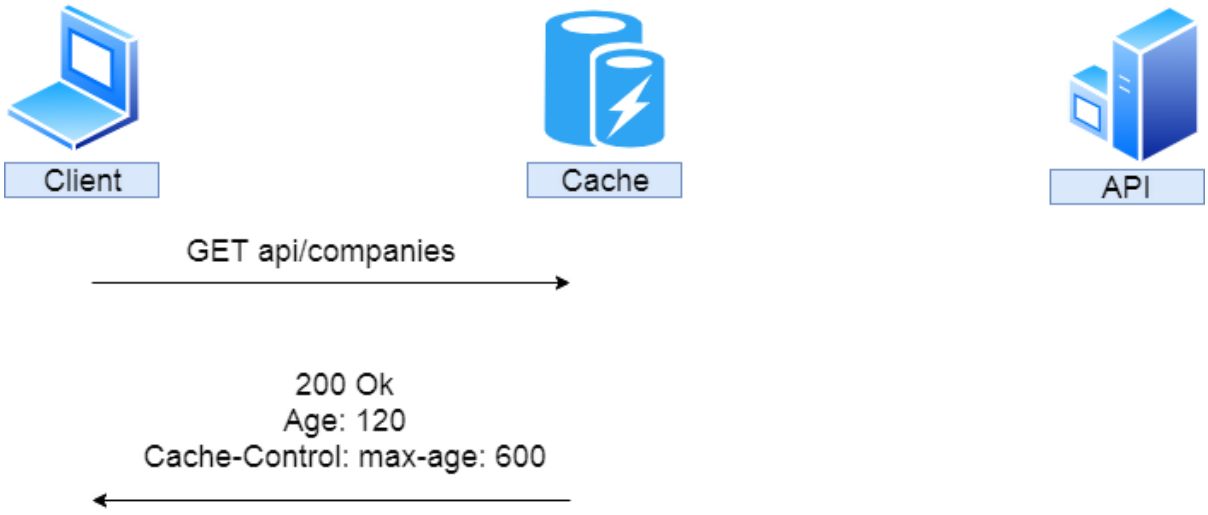
The expiration model allows the server to recognize whether or not the response has expired. As long as the response is fresh, it will be served from the cache. To achieve that, the Cache-Control header is used. We have seen this in the previous example.

Let's look at the diagram to see how caching works:



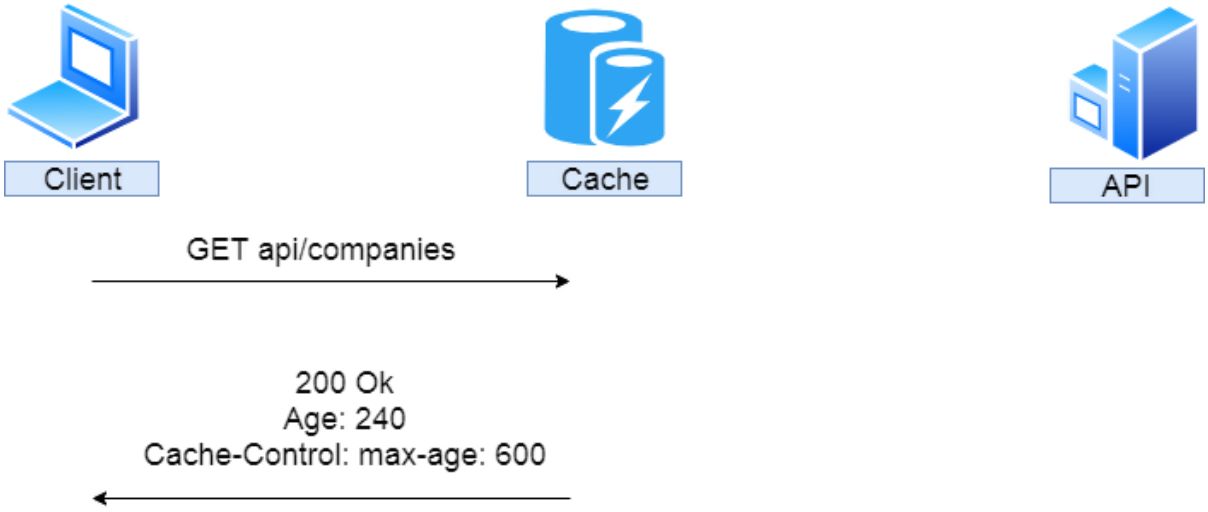
So, the client sends a request to get companies. There is no cached version of that response; therefore, the request is forwarded to the API. The API returns the response with the Cache-Control header with a 10-minute expiration period; it is being stored in the cache and forwarded to the client.

If after two minutes, the same response has been requested:



We can see that the cached response was served with an additional Age header with 120 seconds or two minutes. If this is a private cache, that is where it stops. That's because the private cache is stored in the browser and another client will hit the API for the same response. But if this is a

shared cache and another client requests the same response after an additional two minutes:



The response is served from the cache with an additional two minutes added to the Age header.

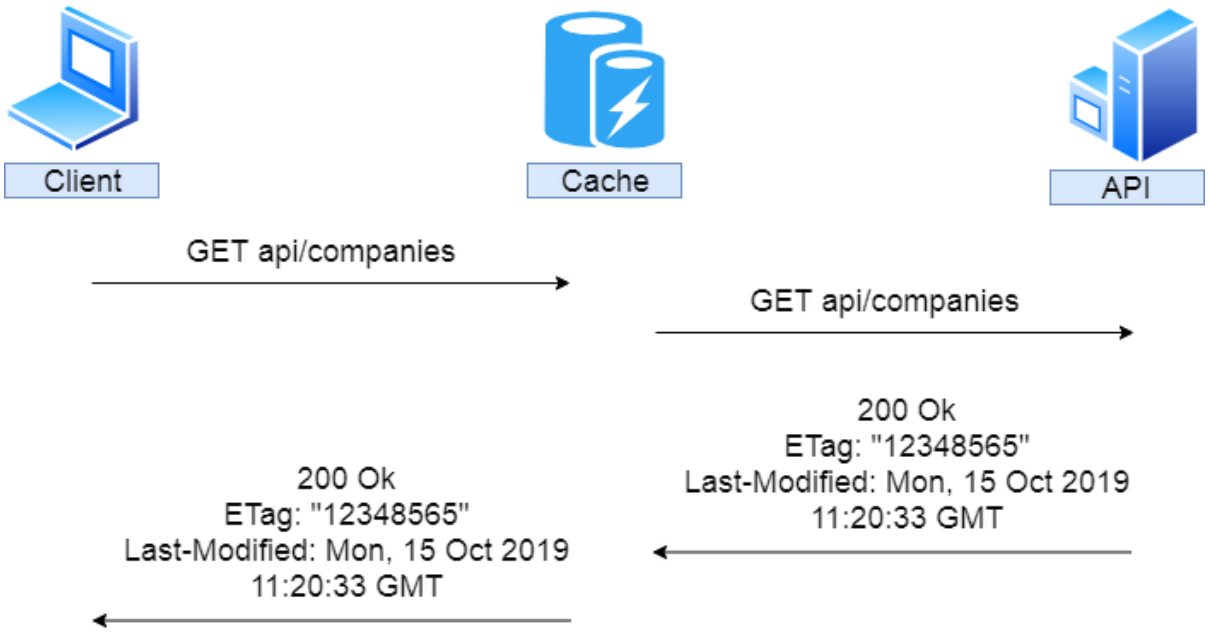
We saw how the Expiration model works, now let's inspect the Validation model.

Validation Model

The validation model is used to validate the freshness of the response. So it checks if the response is cached and still usable. Let's assume we have a shared cached GetCompany response for 30 minutes. If someone updates that company after five minutes, without validation the client would receive the wrong response for another 25 minutes — not the updated one.

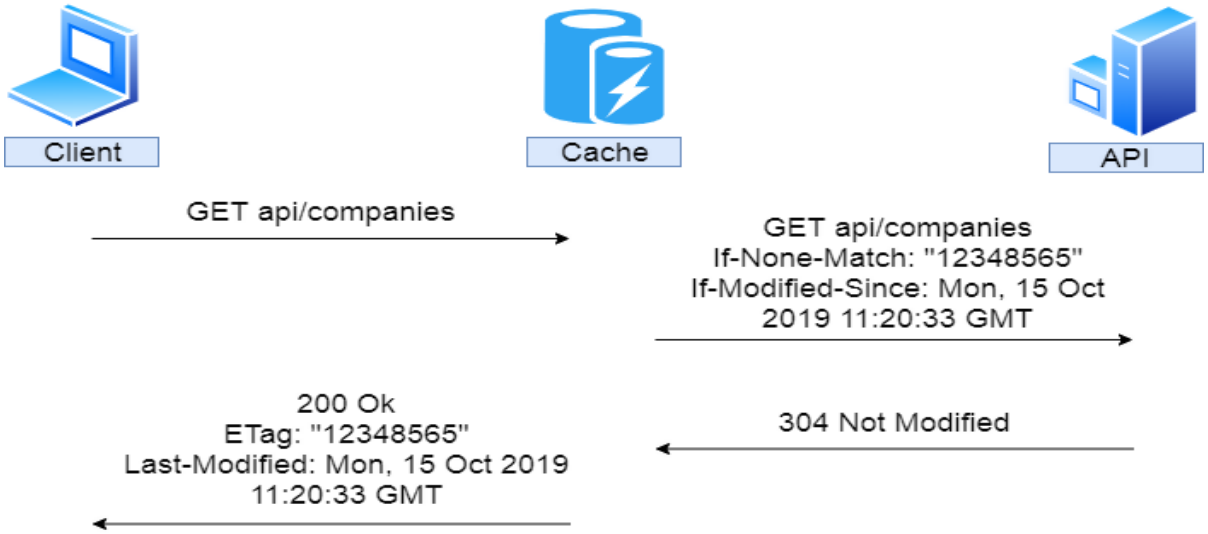
To prevent that, we use validators. The HTTP standard advises using Last-Modified and ETag validators in combination if possible.

Let's see how validation works:



So again, the client sends a request, it is not cached, and so it is forwarded to the API. Our API returns the response that contains the Etag and Last-Modified headers. That response is cached and forwarded to the client.

After two minutes, the client sends the same request:



So, the same request is sent, but we don't know if the response is valid. Therefore, the cache forwards that request to the API with the additional headers If-None-Match — which is set to the Etag value — and If-

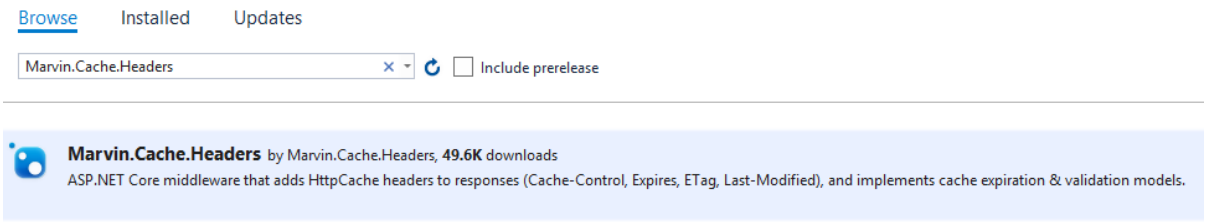
Modified-Since — which is set to the Last-Modified value. If this request checks out against the validators, our API doesn't have to recreate the same response; it just sends a 304 Not Modified status. After that, the regular response is served from the cache. Of course, if this doesn't check out, the new response must be generated.

That brings us to the conclusion that for the shared cache if the response hasn't been modified, that response has to be generated only once.

Let's see all of these in an example.

Supporting Validation

We have to install the `Marvin.Cache.Headers` library in the main project. This library supports HTTP cache headers like Cach-Controle, Expires, Etag, and Last-Modified and also implements validation and expiration models:



Now, let's modify the `ServiceExtensions` class:

```
public static void ConfigureHttpCacheHeaders(this IServiceCollection services) =>
services.AddHttpCacheHeaders();
```

We are going to add additional configuration later.

Then, let's modify the `ConfigureServices` method:

```
services.ConfigureResponseCaching();
services.ConfigureHttpCacheHeaders();
```

And finally, let's modify the `Configure` method:

```
app.UseResponseCaching();
app.UseHttpCacheHeaders();
```

To test this, we have to remove or comment out `ResponseCache` attributes in the `CompaniesController`. The installed library will provide that for us.

Now, let's send the `GetCompany` request:

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

As you can see, we have all the required headers generated. The default expiration is set to 60 seconds and if we send this request one more time, we are going to get an additional Age header.

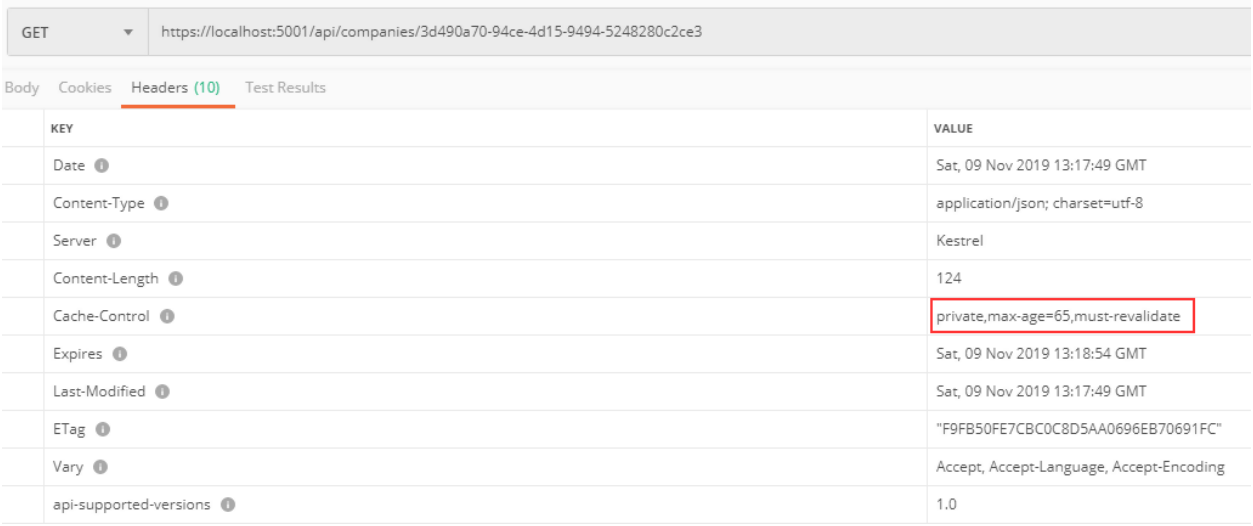
Configuration

We can globally configure our expiration and validation headers. To do that, let's modify the `ConfigureHttpCacheHeaders` method:

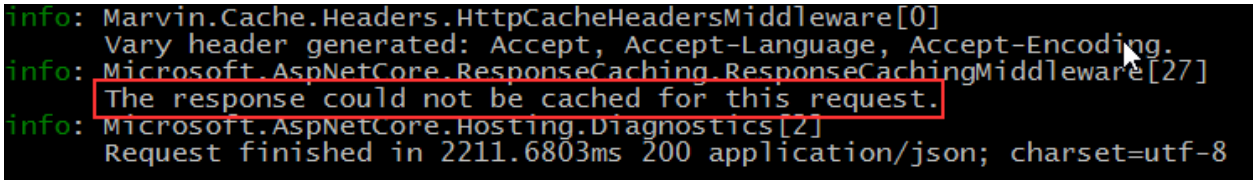
```
public static void ConfigureHttpCacheHeaders(this IServiceCollection services) =>
  services.AddHttpCacheHeaders(
      (expirationOpt) =>
      {
           expirationOpt.MaxAge = 65;
           expirationOpt.CacheLocation = CacheLocation.Private;
       },
       (validationOpt) =>
       {
           validationOpt.MustRevalidate = true;
       });
```

After that, we are going to send the same request for the single company:

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



You can see that the changes are implemented. Now, this is a private cache with an age of 65 seconds. Because it is a private cache, our API won't cache it:



Other then global configuration, we can apply it on the resource level (on action or controller). The overriding rules are the same. Configuration on the action level will override the configuration on the controller or global level. Also, the configuration on the controller level will override the global level configuration.

To apply a resource level configuration, we have to use the `HttpCacheExpiration` and `HttpCacheValidation` attributes:

```
[HttpGet("{id}", Name = "CompanyById")]
[HttpCacheExpiration(CacheLocation = CacheLocation.Public, MaxAge = 60)]
[HttpCacheValidation(MustRevalidate = false)]
public async Task<IActionResult> GetCompany(Guid id)
```

Once we send the `GetCompanies` request, we are going to see global values:

But if we send the `GetCompany` request:

You can see that it is public and you can inspect the console to see the cached response.

Using ETag and Validation

First, we have to mention that the `ResponseCaching` library doesn't correctly implement the validation model. Also, using the authorization header is a problem. We are going to show you alternatives later. But for now, we can simulate how validation with Etag should work.

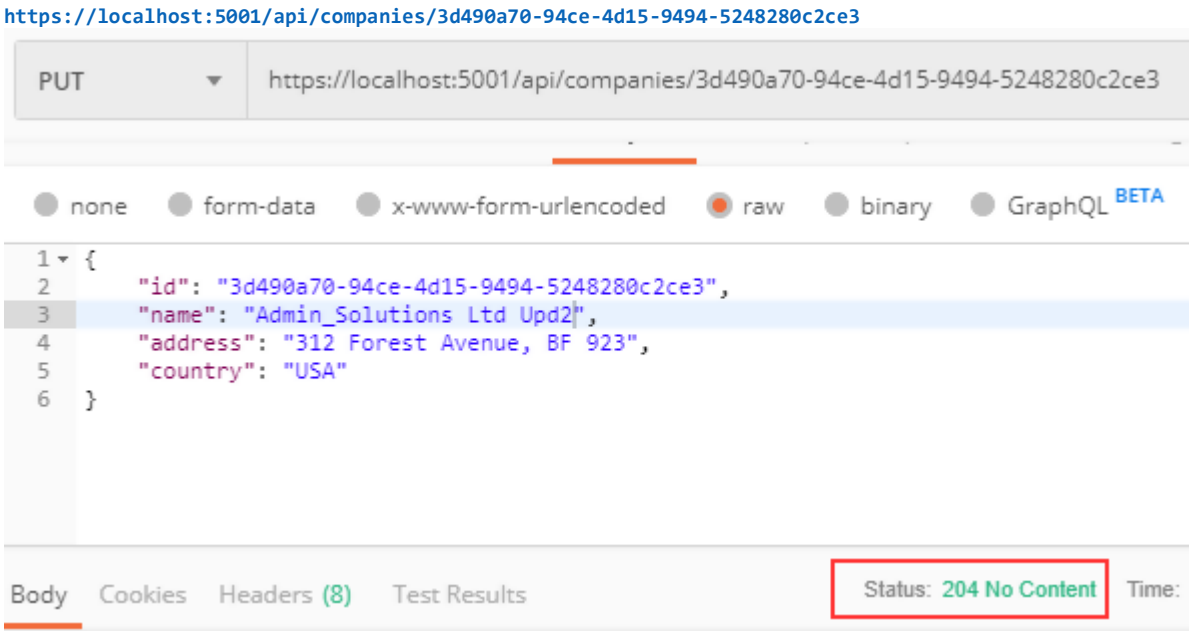
So, let's restart our app to have a fresh application, and send a `GetCompany` request one more time. In a header, we are going to get our ETag. Let's copy the Etag's value and use another `GetCompany` request:

We send the `If-None-Match` tag with the value of our Etag. And we can see as a result we get `304 Not Modified`.

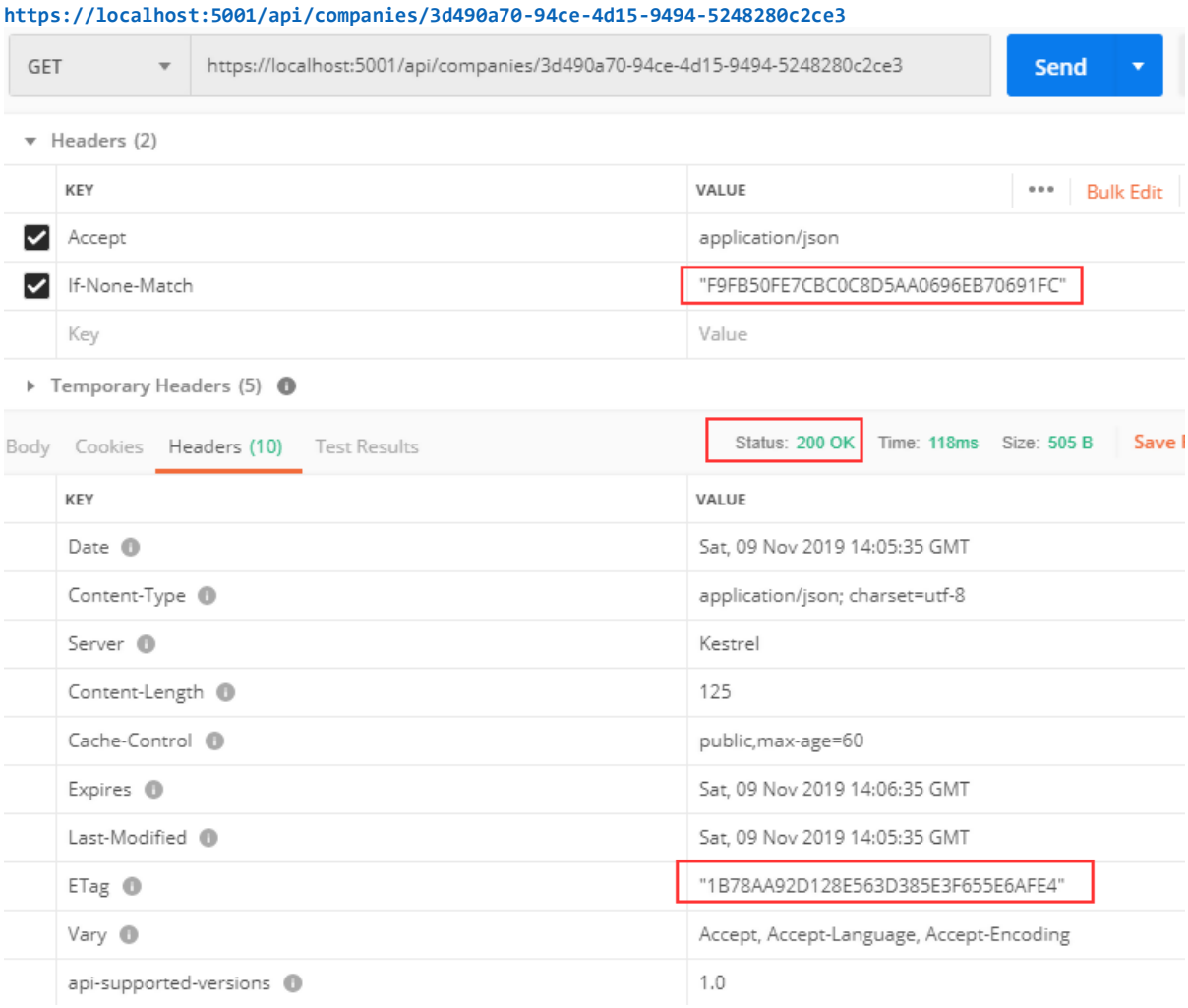
But this is not a valid situation. As we said, the client should send a valid request and it is up to the Cache to add an `If-None-Match` tag. In our example, which we sent from Postman, we simulated that. Then, it is up to the server to return a 304 message to the cache and then the cache should return the same response.

But anyhow, we have managed to show you how validation works.

If we update that company:



And then send the same request with the same If-None-Match value:



You can see that we get 200 OK, and that ETag is different because the resource changed.

So, we saw how validation works and also concluded that the ResponseCaching library is not that good for validation — it is much better for just expiration.

But then, what are the alternatives?

There are a lot of alternatives, such as:

- Varnish - https://varnish-cache.org/
- Apache Traffic Server - https://trafficserver.apache.org/
- Squid - http://www.squid-cache.org/

They implement caching correctly. And if you want to have expiration and validation, you should combine them with the Marvin library and you are good to go. But those servers are not that trivial to implement.

There is another option: CDN (Content Delivery Network). CDN uses HTTP caching and is used by various sites on the internet. The good thing with CDN is we don't need to set up a cache server by ourselves, but unfortunately we have to pay for it. The previous cache servers we presented are free to use. So, it's up to you to decide what suits you best.

RATE LIMITING AND THROTTLING

Rate Limiting allows us to protect our API against too many requests that can deteriorate our API's performance. API is going to reject requests that exceed the limit. Throttling queues exceeded requests for possible later processing. The API will eventually reject the request if processing cannot occur after a certain number of attempts.

For example, we can configure our API to create a limitation of 100 requests/hour per client. Or additionally, we can limit a client to the maximum 1,000 requests/day per IP and 100 requests/hour. We can even limit the number of requests for a specific resource in our API; for example, 50 requests to `api/companies`.

To provide information about rate limiting, we use the response headers. They are separated between Allowed requests, which all start with the X-Rate-Limit and Disallowed requests.

The Allowed requests header contains the following information :

- X-Rate-Limit-Limit – rate limit period.
- X-Rate-Limit-Remaining – number of remaining requests.
- X-Rate-Limit-Reset – date/time information about resetting the request limit.

For the disallowed requests, we use a 429 status code; that stands for too many requests. This header may include the Retry-After response header and should explain details in the response body.

Implementing Rate Limiting

To start, we have to install the `AspNetCoreRateLimit` library:


AspNetCoreRateLimit by Stefan Prodan, Cristi Pufu, **469K** downloads
ASP.NET Core rate limiting middleware

Then, we have to add it to the service collection. This library uses a memory cache to store its counters and rules. Therefore, we have to add the MemoryCache to the service collection as well.

That said, let's add the MemoryCache:

```
services.AddMemoryCache();
```

After that, we are going to create another extension method in the `ServiceExtensions` class:

```
public static void ConfigureRateLimitingOptions(this IServiceCollection services)
{
    var rateLimitRules = new List<RateLimitRule>
    {
        new RateLimitRule
        {
            Endpoint = "*",
            Limit= 3,
            Period = "5m"
        }
    };

    services.Configure<IpRateLimitOptions>(opt =>
    {
        opt.GeneralRules = rateLimitRules;
    });

    services.AddSingleton<IRateLimitCounterStore, MemoryCacheRateLimitCounterStore>();
    services.AddSingleton<IIpPolicyStore, MemoryCacheIpPolicyStore>();
    services.AddSingleton<IRateLimitConfiguration, RateLimitConfiguration>();
}
```

We create a rate limit rules first, for now just one, stating that three requests are allowed in a five-minute period for any endpoint in our API. Then, we configure IpRateLimitOptions to add the created rule. Finally, we have to register rate limit stores and configuration as singleton. They serve the purpose of storing rate limit counters and policies as well as adding configuration.

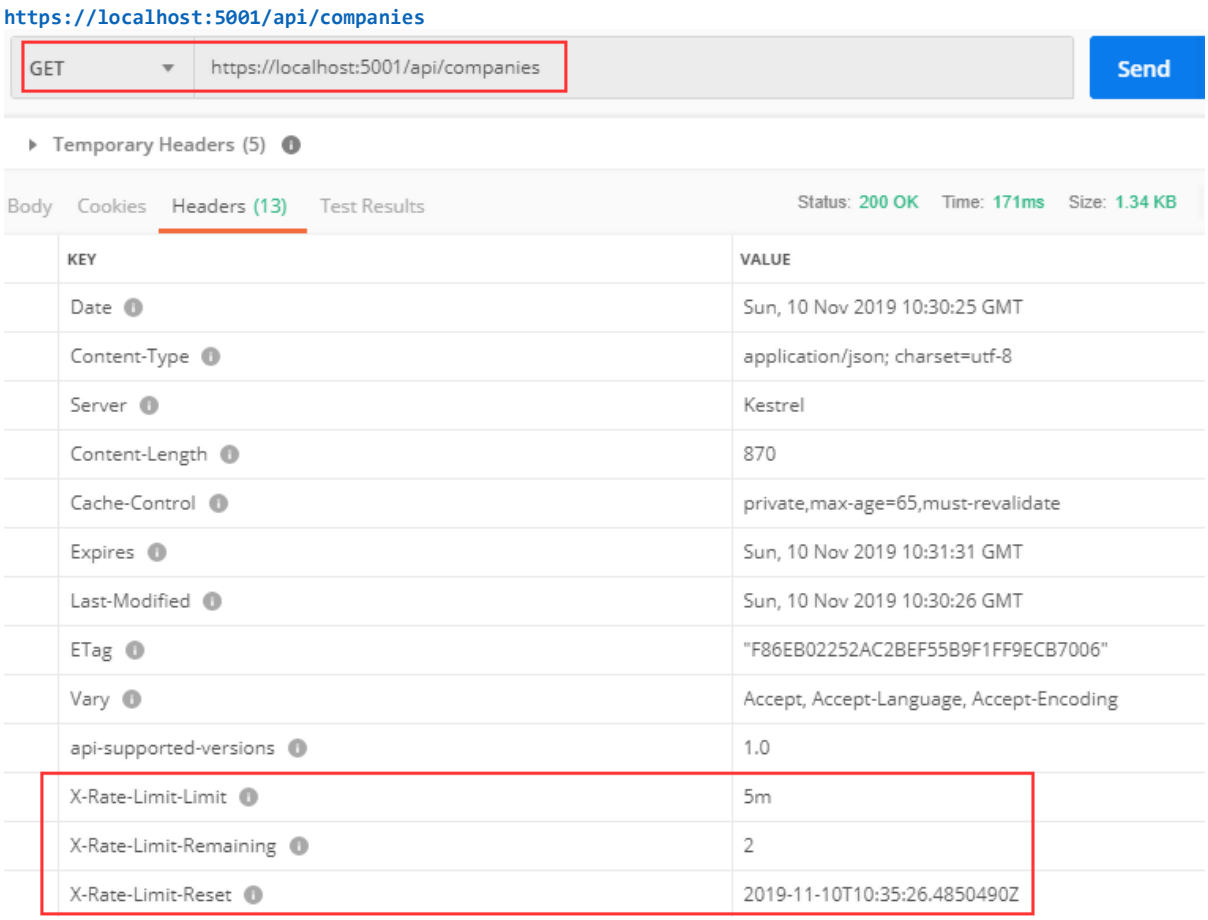
Now, we have to modify the `ConfigureServices` method:

```
services.AddMemoryCache();

services.ConfigureRateLimitingOptions();
services.AddHttpContextAccessor();
```

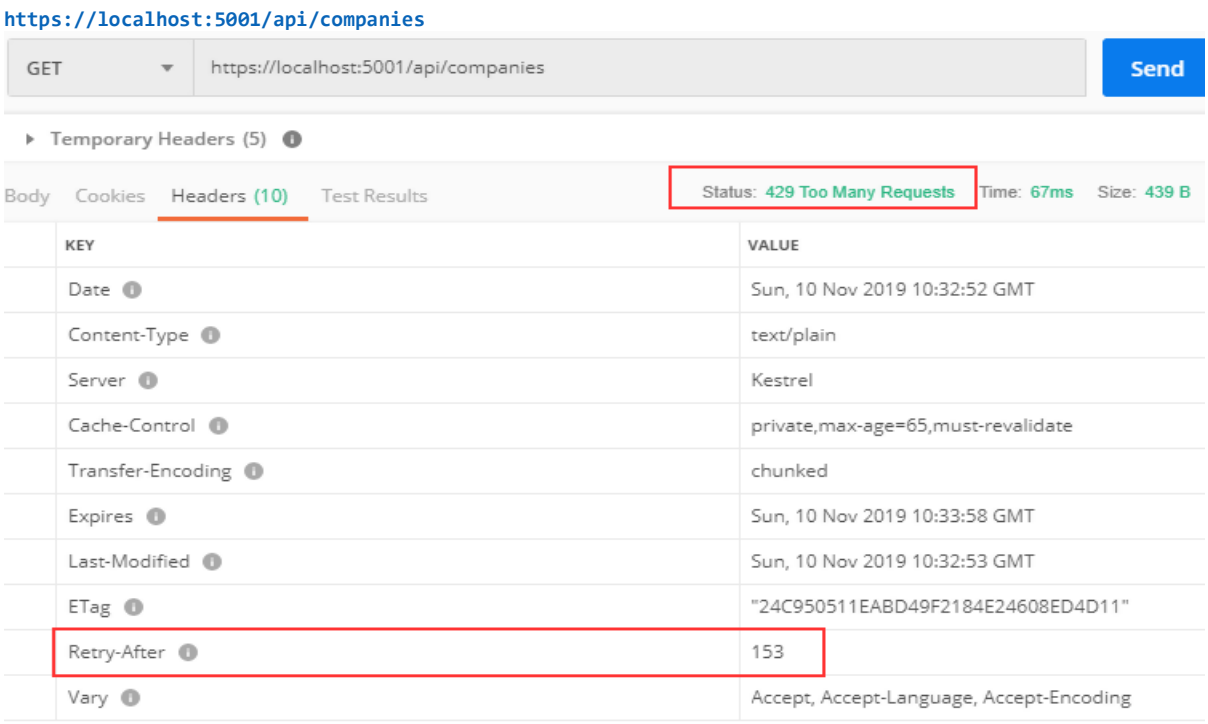
Finally, we have to add it to the request pipeline in the `Configure` method:

```
app.UseIpRateLimiting();

app.UseRouting();
```

And that is it. We can test this now:

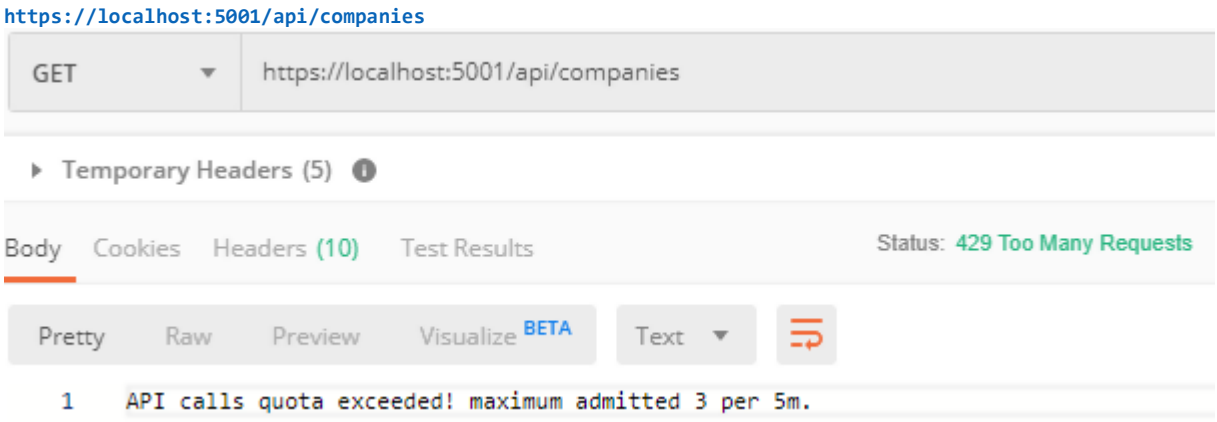


So, we can see that we have two requests remaining and the time to reset the rule. If we send an additional three requests in the five-minute period of time, we are going to get a different response:



The status code is 429 Too Many Requests and we have the Retry-After header.

We can inspect the body as well:



So, our rate limiting works.

There are a lot of options that can be configured with Rate Limiting and you can read more about them on the AspNetCoreRateLimit GitHub page.

JWT AND IDENTITY

User authentication is an important part of any application. It refers to the process of confirming the identity of an application's users. Implementing it properly could be a hard job if you are not familiar with the process. Also, it could take a lot of time that could be spent on different features of an application.

So, in this section, we are going to learn about authentication and authorization in ASP.NET Core by using Identity and JWT (Json Web Token). We are going to explain step by step how to integrate Identity in the existing project and then how to implement JWT for the authentication and authorization actions.

ASP.NET Core provides us with both functionalities, making implementation even easier.

So, let's start with Identity integration.

Implementing Identity in ASP.NET Core Project

Asp.NET Core Identity is the membership system for web applications that includes membership, login, and user data. It provides a rich set of services that help us with creating users, hashing their passwords, creating a database model, and the authentication overall.

That said, let's start with the integration process.

The first thing we have to do is to install the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` library in the `Entities` project:

After the installation, we are going to create a new **User** class in the **Entities/Models** folder:

```
public class User : IdentityUser
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Our class inherits from the **IdentityUser** class that has been provided by the ASP.NET Core Identity. It contains different properties and we can extend it with our own as well.

After that, we have to modify the **RepositoryContext** class:

```
public class RepositoryContext : IdentityDbContext<User>
{
    public RepositoryContext(DbContextOptions options)
    : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.ApplyConfiguration(new CompanyConfiguration());
        modelBuilder.ApplyConfiguration(new EmployeeConfiguration());
    }

    public DbSet<Company> Companies { get; set; }
    public DbSet<Employee> Employees { get; set; }
}
```

So, our class now inherits from the **IdentityDbContext** class and not **DbContext** because we want to integrate our context with Identity. Additionally, we call the **OnModelCreating** method from the base class. This is required for migration to work properly.

Now, we have to move on to the configuration part.

To do that, let's create a new extension method in the **ServiceExtensions** class:

```
public static void ConfigureIdentity(this IServiceCollection services)
{
```

```
    var builder = services.AddIdentityCore<User>(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;
    });

    builder = new IdentityBuilder(builder.UserType, typeof(IdentityRole),
builder.Services);
    builder.AddEntityFrameworkStores<RepositoryContext>()
        .AddDefaultTokenProviders();
}
```

With the `AddIdentityCore` method, we are adding and configuring Identity for the specific type; in this case, the `User` type. As you can see, we use different configuration parameters that are pretty self-explanatory on their own. Identity provides us with even more features to configure, but these are sufficient for our example.

Then, we create an Identity builder and add `EntityFrameworkStores` implementation with the default token providers.

Now, let's modify the `ConfigureServices` method:

```
services.AddAuthentication();
services.ConfigureIdentity();
```

And, let's modify the `Configure` method:

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

That's it. We have prepared everything we need.

Creating Tables and Inserting Roles

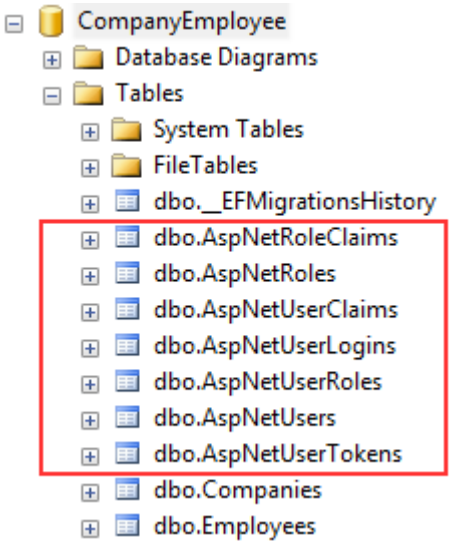
Creating tables is quite an easy process. All we have to do is to create and apply migration. So, let's create a migration:

```
PM> Add-Migration CreatingIdentityTables
```

And then apply it:

```
PM> Update-Database
```

If we check our database now, we are going to see additional tables:



For our project, the AspNetRoles, AspNetUserRoles, and AspNetUsers tables will be quite enough. If you open the AspNetUsers table, you will see additional FirstName and LastName columns.

Now, let's insert several roles in the AspNetRoles table, again by using migrations. The first thing we are going to do is to create the **RoleConfiguration** class in the **Entities/Configuration** folder:

```
public class RoleConfiguration : IEntityTypeConfiguration<IdentityRole>
{
    public void Configure(EntityTypeBuilder<IdentityRole> builder)
    {
        builder.HasData(
        new IdentityRole
        {
            Name = "Manager",
            NormalizedName = "MANAGER"
        },
        new IdentityRole
        {
            Name = "Administrator",
            NormalizedName = "ADMINISTRATOR"
        }
    );
}
```

And let's modify the `OnModelCreating` method in the `RepositoryContext` class:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.ApplyConfiguration(new CompanyConfiguration());
    modelBuilder.ApplyConfiguration(new EmployeeConfiguration());
    modelBuilder.ApplyConfiguration(new RoleConfiguration());
}
```

Finally, let's create and apply migration:

```
PM> Add-Migration AddedRolesToDb

PM> Update-Database
```

If you check the AspNetRoles table, you will find two new roles created.

User Creation

For this, we have to create a new controller:

```
[Route("api/authentication")]
[ApiController]
public class AuthenticationController: ControllerBase
{
    private readonly ILoggerManager _logger;
    private readonly IMapper _mapper;
    private readonly UserManager<User> _userManager;
    public AuthenticationController (ILoggerManager logger, IMapper mapper,
UserManager<User> userManager)
    {
        _logger = logger;
        _mapper = mapper;
        _userManager = userManager;
    }
}
```

So, this is a familiar code except for the `UserManager<TUser>` part. That service is provided by Identity and it provides APIs for managing users. We don't have to inject our repository here because UserManager provides us all we need for this example.

The next thing we have to do is to create a `UserForRegistrationDto` class in the `DataTransferObjects` folder:

```
public class UserForRegistrationDto
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    [Required(ErrorMessage = "Username is required")]
    public string UserName { get; set; }
    [Required(ErrorMessage = "Password is required")]
    public string Password { get; set; }
    public string Email { get; set; }
    public string PhoneNumber { get; set; }
    public ICollection<string> Roles { get; set; }
}
```

Then, let's create a mapping rule in the **MappingProfile** class:

```
CreateMap<UserForRegistrationDto, User>();
```

Finally, it is time to create the **RegisterUser** action:

```
[HttpPost]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> RegisterUser([FromBody] UserForRegistrationDto
userForRegistration)
{
    var user = _mapper.Map<User>(userForRegistration);

    var result = await _userManager.CreateAsync(user, userForRegistration.Password);
    if(!result.Succeeded)
    {
        foreach (var error in result.Errors)
        {
            ModelState.TryAddModelError(error.Code, error.Description);
        }

        return BadRequest(ModelState);
    }

    await _userManager.AddToRolesAsync(user, userForRegistration.Roles);

    return StatusCode(201);
}
```

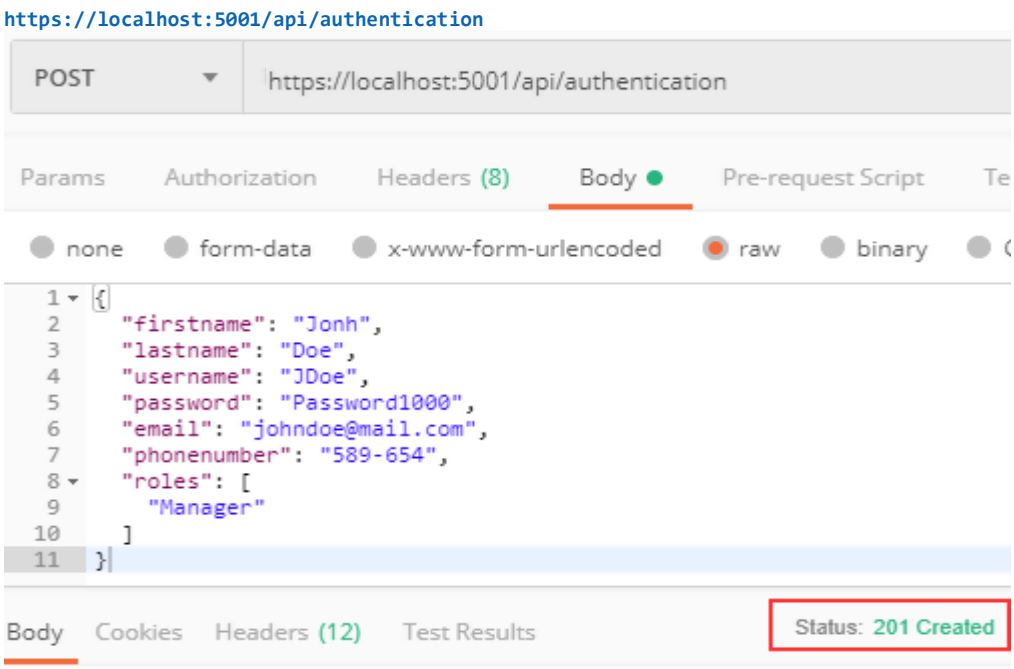
We are implementing our existing action filter for the entity and model validation on top of our action. After that, we map the DTO object to the User object and call the **CreateAsync** method to create that specific user in the database. The **CreateAsync** method will save the user to the database if the action succeeds or it will return error messages. If it returns error messages, we add them to the model state.

Finally, if a user is created, we connect it to its roles — the default one or the ones sent from the client side — and return `201 created`.

If you want, before calling `AddToRoleAsync` or `AddToRolesAsync`, you can check if roles exist in the database. But for that, you have to inject `RoleManager<TRole>` and use the `RoleExistsAsync` method. Now, we can test this.

Before we continue, we should increase a rate limit from 3 to 30 (`ServiceExtensions` class, `ConfigureRateLimitingOptions` method) just to not stand in our way while we're testing the different features of our application.

Let's send a valid request first:



And we get 201, which means that the user has been created and added to the role. We can send additional invalid requests to test our Action and Identity features.

If the model is invalid:

https://localhost:5001/api/authentication

```
{
    "UserName": [
        "Username is required"
    ]
}
```

If the password is invalid:

https://localhost:5001/api/authentication

```
{
    "PasswordTooShort": [
        "Passwords must be at least 10 characters."
    ],
    "PasswordRequiresDigit": [
        "Passwords must have at least one digit ('0'-'9')."
    ]
}
```

Finally, if we want to create a user with the same user name and email:

https://localhost:5001/api/authentication

```
{
    "DuplicateEmail": [
        "Email 'johndoe@mail.com' is already taken."
    ],
    "DuplicateUserName": [
        "User name 'JDoe' is already taken."
    ]
}
```

Excellent. Everything is working as planned. We can move on to the JWT implementation.

Big Picture

Before we get into the implementation of authentication and authorization, let's have a quick look at the big picture. There is an application that has a login form. A user enters its username and password and presses the login button. After pressing the login button, a client (e.g., web browser) sends the user's data to the server's API endpoint:

When the server validates the user's credentials and confirms that the user is valid, it's going to send an encoded JWT to the client. A JSON web token is a JavaScript object that can contain some attributes of the logged-in user. It can contain a username, user subject, user roles, or some other useful information.

About JWT

JSON web tokens enable a secure way to transmit data between two parties in the form of a JSON object. It's an open standard and it's a popular mechanism for web authentication. In our case, we are going to use JSON web tokens to securely transfer a user's data between the client and the server.

JSON web tokens consist of three basic parts: the header, the payload, and the signature.

One real example of a JSON web token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHM
I6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

Every part of all three parts is shown in a different color. The first part of JWT is the header, which is a JSON object encoded in the base64 format. The header is a standard part of JWT and we don't have to worry about it.

It contains information like the type of token and the name of the algorithm:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

After the header, we have a payload which is also a JavaScript object encoded in the base64 format. The payload contains some attributes about the logged-in user. For example, it can contain the user id, the user subject, and information about whether a user is an admin user or not. **JSON web tokens are not encrypted** and can be decoded with any base64 decoder, so please **never include sensitive information in the Payload:**

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Finally, we have the signature part. Usually, the server uses the signature part to verify whether the token contains valid information, the information which the server is issuing. It is a digital signature that gets generated by combining the header and the payload. Moreover, it's based on a secret key that only the server knows:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  superSecertKey
) secret base64 encoded
```

So, if malicious users try to modify the values in the payload, they have to recreate the signature; for that purpose, they need the secret key only known to the server. At the server side, we can easily verify if the values

are original or not by comparing the original signature with a new signature computed from the values coming from the client.

So, we can easily verify the integrity of our data just by comparing the digital signatures. This is the reason why we use JWT.

JWT Configuration

Let's start by modifying the appsettings.json file:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "ConnectionStrings": {
    "sqlConnection": "server=.; database=CompanyEmployee; Integrated Security=true"
  },
  "JwtSettings": {
    "validIssuer": "CodeMazeAPI",
    "validAudience": "https://localhost:5001"
  },
  "AllowedHosts": "*"
}
```

We just store the issuer and audience information in the appsettings.json file. We are going to talk more about that in a minute. As you probably remember, we require a secret key on the server side. So, we are going to create one and store it in the environment variable because this is much safer than storing it inside the project.

To create an environment variable, we have to open the cmd window as an administrator and type the following command:

```
setx SECRET "CodeMazeSecretKey" /M
```

This is going to create a system environment variable with the name SECRET and the value CodeMazeSecretKey. By using /M we specify that we want a system variable and not local.

Great.

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.GetSection("validIssuer").Value,
            ValidAudience = jwtSettings.GetSection("validAudience").Value,
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey))
        };
    });
}
```

First, we extract the `JwtSettings` from the `appconfig.json` file and extract our environment variable (If you keep getting null for the secret key, try restarting the Visual Studio or even your computer).

Then, we register the JWT authentication middleware by calling the method `AddAuthentication` on the `IServiceCollection` interface. Next, we specify the authentication scheme `JwtBearerDefaults.AuthenticationScheme` as well as `ChallengeScheme`. We also provide some parameters that will be used while validating JWT. For this to work, we have to install the `Microsoft.AspNetCore.Authentication.JwtBearer` library.

Excellent.

We've successfully configured the JWT authentication.

According to the configuration, the token is going to be valid if:

- The issuer is the actual server that created the token (ValidateIssuer=true)
- The receiver of the token is a valid recipient (ValidateAudience=true)
- The token has not expired (ValidateLifetime=true)
- The signing key is valid and is trusted by the server (ValidateIssuerSigningKey=true)

Additionally, we are providing values for the issuer, the audience, and the secret key that the server uses to generate the signature for JWT.

All we have to do is to call this method in the ConfigureServices method:

```
services.ConfigureIdentity();
services.ConfigureJWT(Configuration);
```

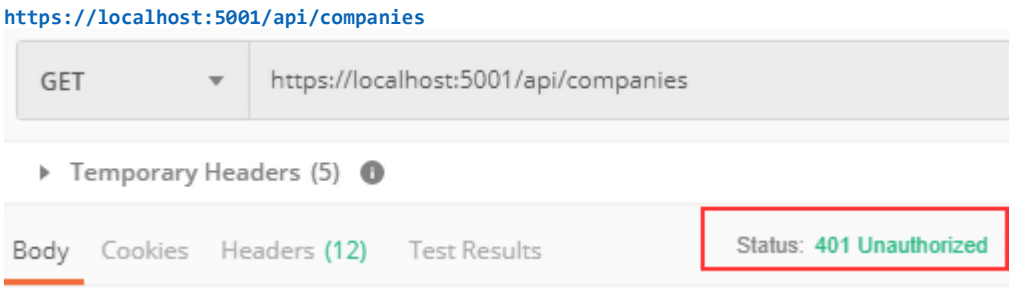
And that is it. We can now protect our endpoints.

Protecting Endpoints

Let's open the `CompaniesController` and add an additional attribute above the `GetCompanies` action:

```
[HttpGet(Name = "GetCompanies"), Authorize]
public async Task<IActionResult> GetCompanies()
```

To test this, let's send a request to get all companies:



We see the protection works. We get a 401 Unauthorized message, which is expected because an unauthorized user tried to access the protected endpoint. So, what we need is our user to be authenticated and to have a valid token.

Implementing Authentication

Let's begin with the UserForAuthenticationDto class:

```
public class UserForAuthenticationDto
{
    [Required(ErrorMessage = "User name is required")]
    public string UserName { get; set; }

     [Required(ErrorMessage = "Password name is required")]
     public string Password { get; set; }
 }
```

We are going to have some complex logic for the authentication and the token generation actions; therefore, it is best to extract these actions in another service.

That said, let's create a new **IAuthenticationManager** interface in the **Contracts** project:

```
public interface IAuthenticationManager
{
    Task<bool> ValidateUser(UserForAuthenticationDto userForAuth);
    Task<string> CreateToken();
}
```

Next, let's create the **AuthenticationManager** class and implement this interface:

```
public class AuthenticationManager : IAuthenticationManager
{
    private readonly UserManager<User> _userManager;
    private readonly IConfiguration _configuration;

    private User _user;

    public AuthenticationManager(UserManager<User> userManager, IConfiguration
configuration)
    {
        _userManager = userManager;
        _configuration = configuration;
    }

    public async Task<bool> ValidateUser(UserForAuthenticationDto userForAuth)
    {
        _user = await _userManager.FindByNameAsync(userForAuth.UserName);

        return (_user != null && await _userManager.CheckPasswordAsync(_user,
userForAuth.Password));
    }
```

```
    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.GetSection("validIssuer").Value,
            audience: jwtSettings.GetSection("validAudience").Value,
            claims: claims,
            expires:
DateTime.Now.AddMinutes(Convert.ToDouble(jwtSettings.GetSection("expires").Value)),
            signingCredentials: signingCredentials
        );

        return tokenOptions;
    }
}
```

In the `ValidateUser` method, we check whether the user exists in the database and if the password matches. The UserManager<TUser> class provides the `FindByNameAsync` method to find the user by user name

and the `CheckPasswordAsync` to verify the user's password against the hashed password from the database.

The CreateToken method does exactly that — it creates a token. It does that by collecting information from the private methods and serializing token options with the `WriteToken` method.

We have three private methods as well. The `GetSignInCredentials` method returns our secret key as a byte array with the security algorithm. The `GetClaims` method creates a list of claims with the user name inside and all the roles the user belongs to. The last method, `GenerateTokenOptions`, creates an object of the JwtSecurityToken type with all of the required options. We can see the expires parameter as one of the token options. We would extract it from the appsettings.json file as well, but we don't have it there. So, we have to add it:

```
"JwtSettings": {
    "validIssuer": "CodeMazeAPI",
    "validAudience": "https://localhost:5001",
    "expires": 5
  }
```

After that, we want to register this class in the `IServiceCollection`:

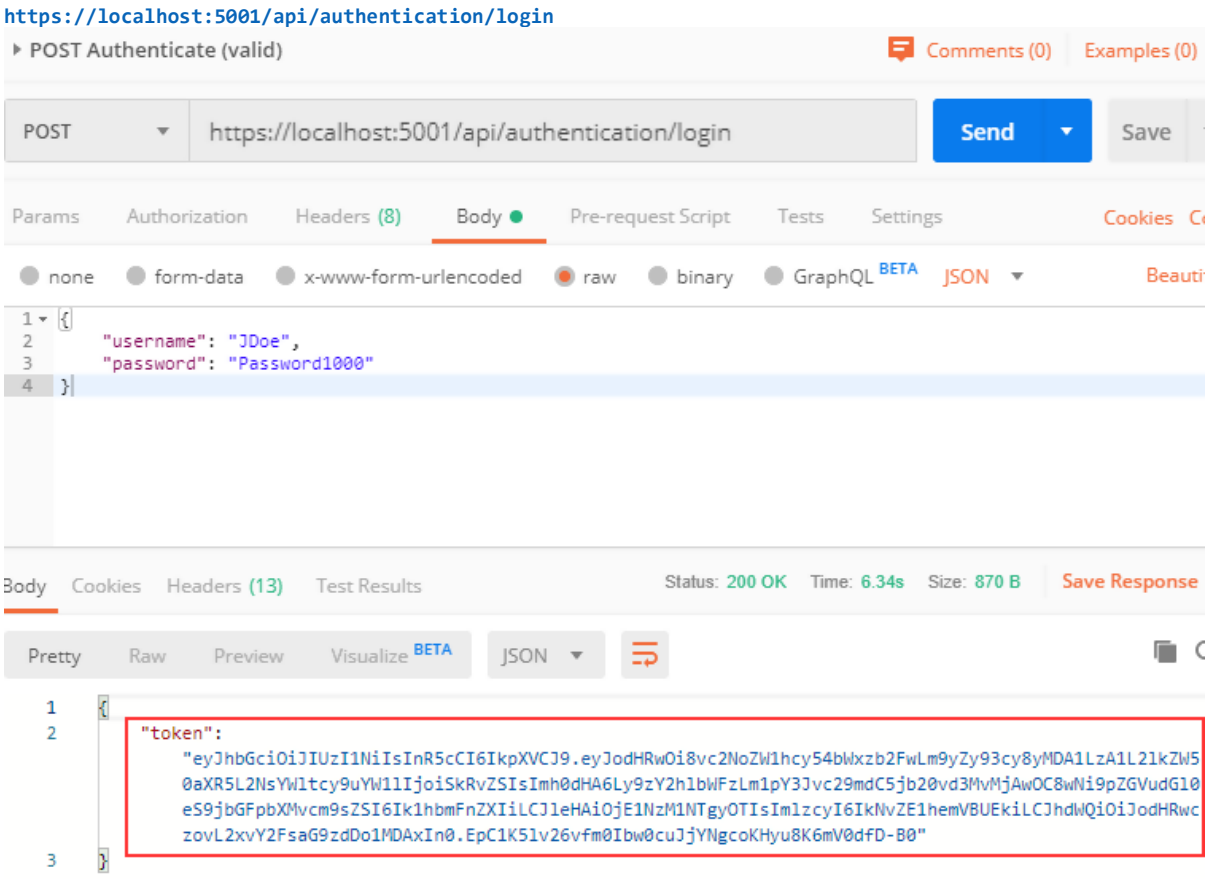
```
services.AddScoped<IAuthenticationManager, AuthenticationManager>();
```

Finally, we have to modify the `AuthenticationController`:

```
[Route("api/authentication")]
[ApiController]
public class AuthenticationController : ControllerBase
{
    private readonly ILoggerManager _logger;
    private readonly IMapper _mapper;
    private readonly UserManager<User> _userManager;
    private readonly IAuthenticationManager _authManager;
    public AuthenticationController(ILoggerManager logger, IMapper mapper,
UserManager<User> userManager, IAuthenticationManager authManager)
    {
        _logger = logger;
        _mapper = mapper;
        _userManager = userManager;
        _authManager = authManager;
    }
```

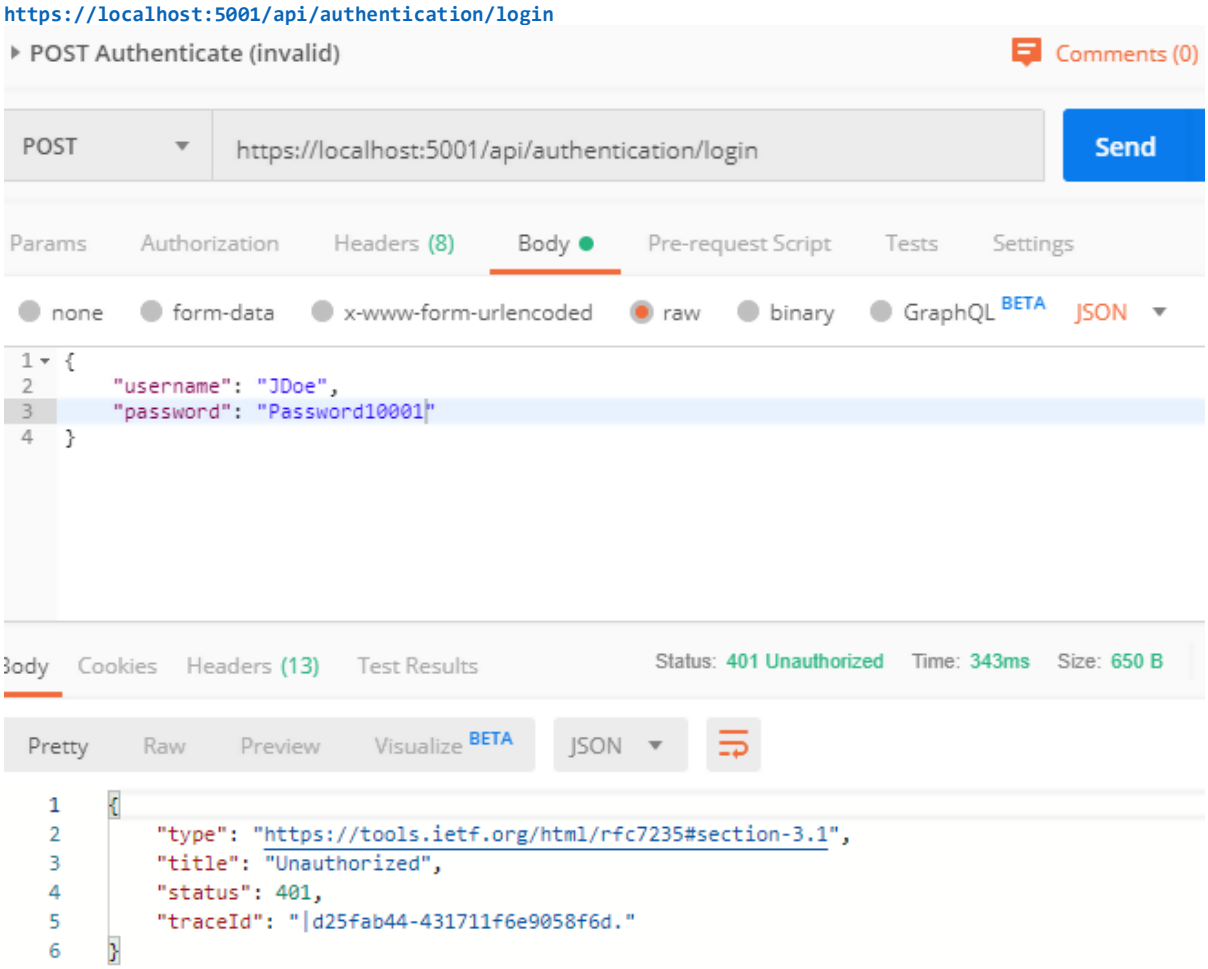
```
    //Previous action

    [HttpPost("login")]
    [ServiceFilter(typeof(ValidationFilterAttribute))]
    public async Task<IActionResult> Authenticate([FromBody] UserForAuthenticationDto
user)
    {
        if (!await _authManager.ValidateUser(user))
        {
            _logger.LogWarn($"{nameof(Authenticate)}: Authentication failed. Wrong
user name or password.");
            return Unauthorized();
        }

        return Ok(new { Token = await _authManager.CreateToken() });
    }
}
```

There is really nothing special in this controller. If validation fails, we return the 401 Unauthorized response; otherwise, we return our created token:



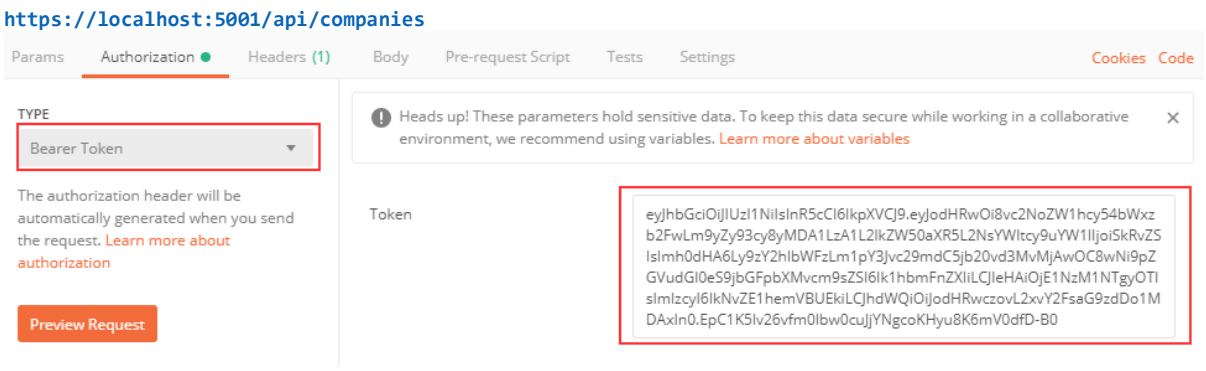
Excellent. We can see our token generated.

Now, let's send invalid credentials:

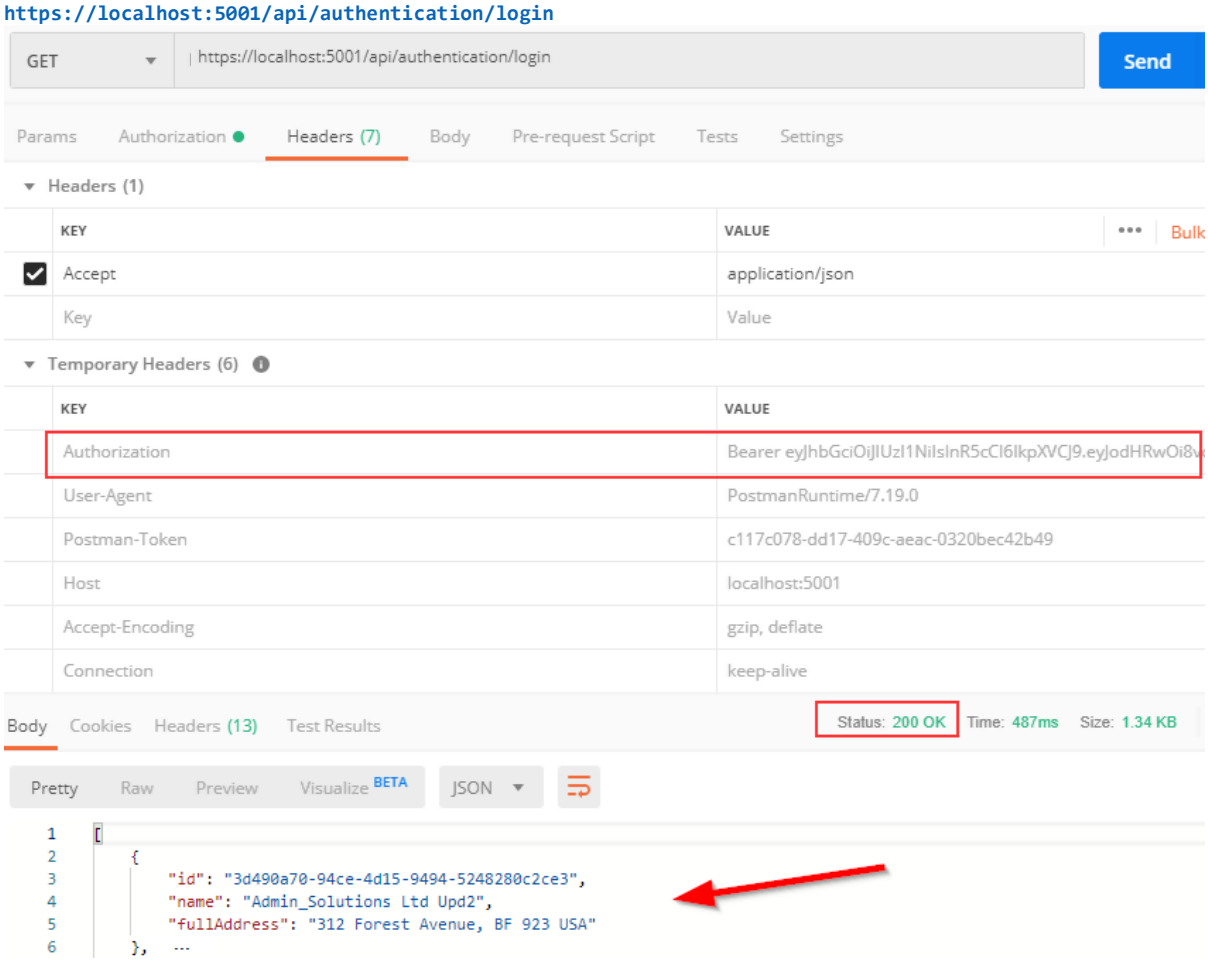


And we get a 401 Unauthorized message.

Right now if we send a request to the `GetCompanies` action, we are still going to get the 401 Unauthorized response even though we have successful authentication. That's because we didn't provide our token in a request header and our API has nothing to authorize against. To solve that, we are going to create another GET request, and in the Authorization header choose the header type and paste the token from the previous request:



Now, we can send the request again:



Excellent. It works like a charm.

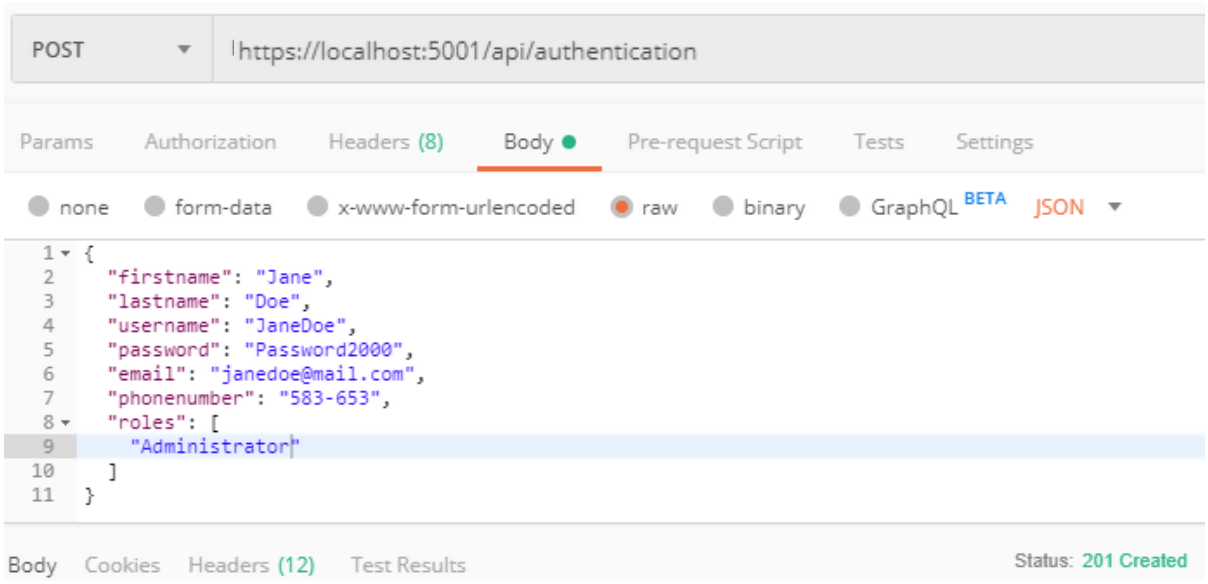
Role-Based Authorization

Right now, even though authentication and authorization are working as expected, every single authenticated user can access the `GetCompanies` action. What if we don't want that type of behavior? For example, we

want to allow only managers to access it. To do that, we have to make one simple change:

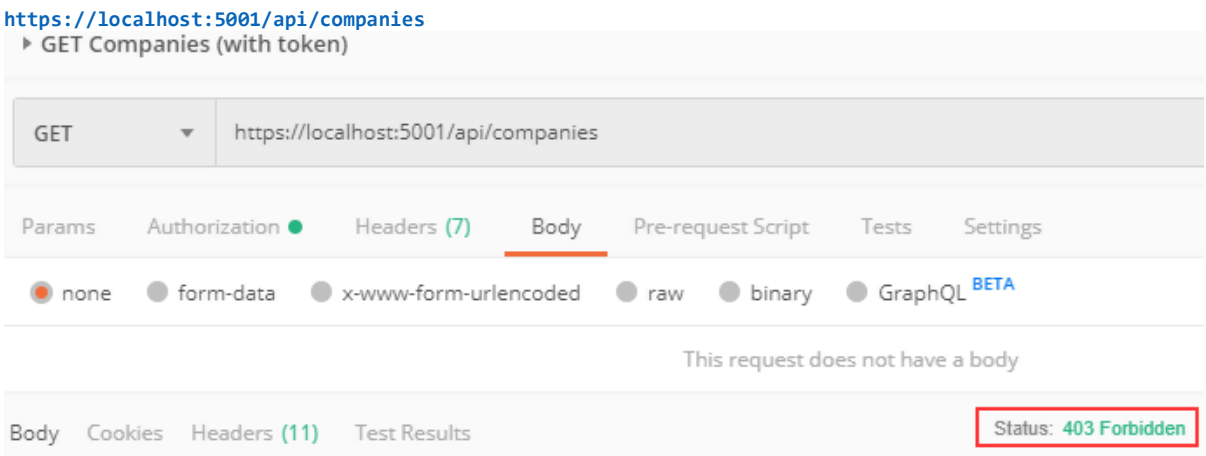
```
[HttpGet(Name = "GetCompanies"), Authorize(Roles = "Manager")]
public async Task<IActionResult> GetCompanies()
```

And that is it. To test this, let's create another user with the Administrator role (the second role from the database):



We get 201.

After we send an authentication request for Jane Doe, we are going to get a new token. Let's use that token to send the request towards the `GetCompanies` action:



As you can see, we get a 403 Forbidden message because this user is not allowed to access the required endpoint. If we login with John Doe and use his token, we are going to get a successful response for sure. Of course, we don't have to place an Authorize attribute only on top of the action; we can place it on the controller level as well. For example, we can place just [Authorize] on the controller level to allow only authorized users to access all the actions in that controller; also, we can place the [Authorize (Role=…)] on top of any action in that controller to state that only a user with that specific role has access to that action.

One more thing. Our token expires after five minutes from the creation point. So, if we try to send another request after that period, we are going to get the 401 Unauthorized status for sure. Feel free to try.

DOCUMENTING API WITH SWAGGER

Developers who consume our API might be trying to solve important business problems with it. Hence, it is very important for them to understand how to use our API effectively. This is where API documentation comes into the picture.

API documentation is the process of giving instructions on how to effectively use and integrate an API. Hence, it can be thought of as a concise reference manual containing all the information required to work with the API, with details about functions, classes, return types, arguments, and more, supported by tutorials and examples.

So, having the proper documentation for our API enables consumers to integrate our APIs as quickly as possible and move forward with their development. Furthermore, this also helps them understand the value and usage of our API, improves the chances for our API's adoption, and makes our APIs easier to maintain and support.

About Swagger

Swagger is a language-agnostic specification for describing REST APIs. Swagger is also referred to as OpenAPI. It allows us to understand the capabilities of a service without looking at the actual implementation code.

Swagger minimizes the amount of work needed while integrating an API. Similarly, it also helps API developers document their APIs quickly and accurately.

Swagger Specification is an important part of the Swagger flow. By default, a document named `swagger.json` is generated by the Swagger tool which is based on our API. It describes the capabilities of our API and how to access it via HTTP.

Swagger Integration Into Our Project

We can use the Swashbuckle package to easily integrate Swagger into our .NET Core Web API project. It will generate the Swagger specification for the project as well. Additionally, the Swagger UI is also contained within Swashbuckle.

There are three main components in the Swashbuckle package:

- `Swashbuckle.AspNetCore.Swagger`: This contains the Swagger object model and the middleware to expose SwaggerDocument objects as JSON.
- `Swashbuckle.AspNetCore.SwaggerGen`: A Swagger generator that builds SwaggerDocument objects directly from our routes, controllers, and models.
- `Swashbuckle.AspNetCore.SwaggerUI`: An embedded version of the Swagger UI tool. It interprets Swagger JSON to build a rich, customizable experience for describing web API functionality.

So, the first thing we are going to do is to install the required library. Let's open the Package Manager Console window and type the following command:

```
PM> Install-Package Swashbuckle.AspNetCore -version 5.0.0
```

After a couple of seconds, the package will be installed. Now, we have to configure the Swagger Middleware. To do that, we are going to add a new method in the `ServiceExtensions` class:

```
public static void ConfigureSwagger(this IServiceCollection services)
{
    services.AddSwaggerGen(s =>
    {
        s.SwaggerDoc("v1", new OpenApiInfo { Title = "Code Maze API", Version = "v1"
});
        s.SwaggerDoc("v2", new OpenApiInfo { Title = "Code Maze API", Version = "v2"
});
    });
}
```

We are creating two versions of SwaggerDoc because if you remember, we have two versions for the Companies controller and we want to separate them in our documentation.

The next step is to call this method in the `ConfigureServices` method:

```
services.ConfigureSwagger();
```

And finally, in the `Configure` method, we are going to add it to the application's execution pipeline together with the UI feature:

```
app.UseSwagger();
app.UseSwaggerUI(s =>
{
    s.SwaggerEndpoint("/swagger/v1/swagger.json", "Code Maze API v1");
    s.SwaggerEndpoint("/swagger/v2/swagger.json", "Code Maze API v2");
});
```

Finally, let's sligthly modify the Companies and CompaniesV2 controllers:

```
[Route("api/companies")]
[ApiController]
[ApiExplorerSettings(GroupName = "v1")]
public class CompaniesController : ControllerBase

[Route("api/companies")]
[ApiController]
[ApiExplorerSettings(GroupName = "v2")]
public class CompaniesV2Controller : ControllerBase
```

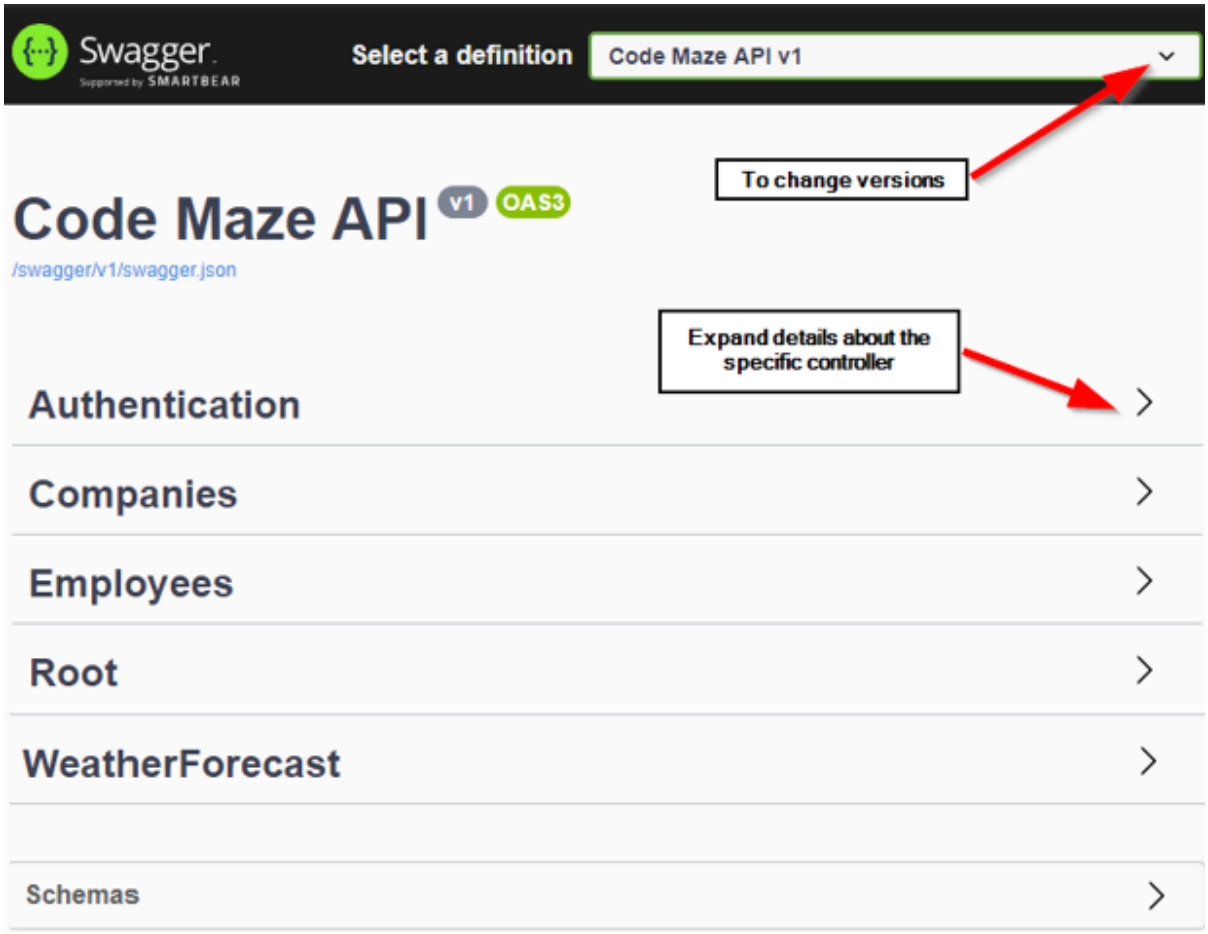
With this change, we state that the CompaniesController belongs to group v1 and the CompaniesV2Controller belongs to group v2. All the other controllers will be included in both groups because they are not versioned. Which is what we want.

And that is all. We have prepared the basic configuration.

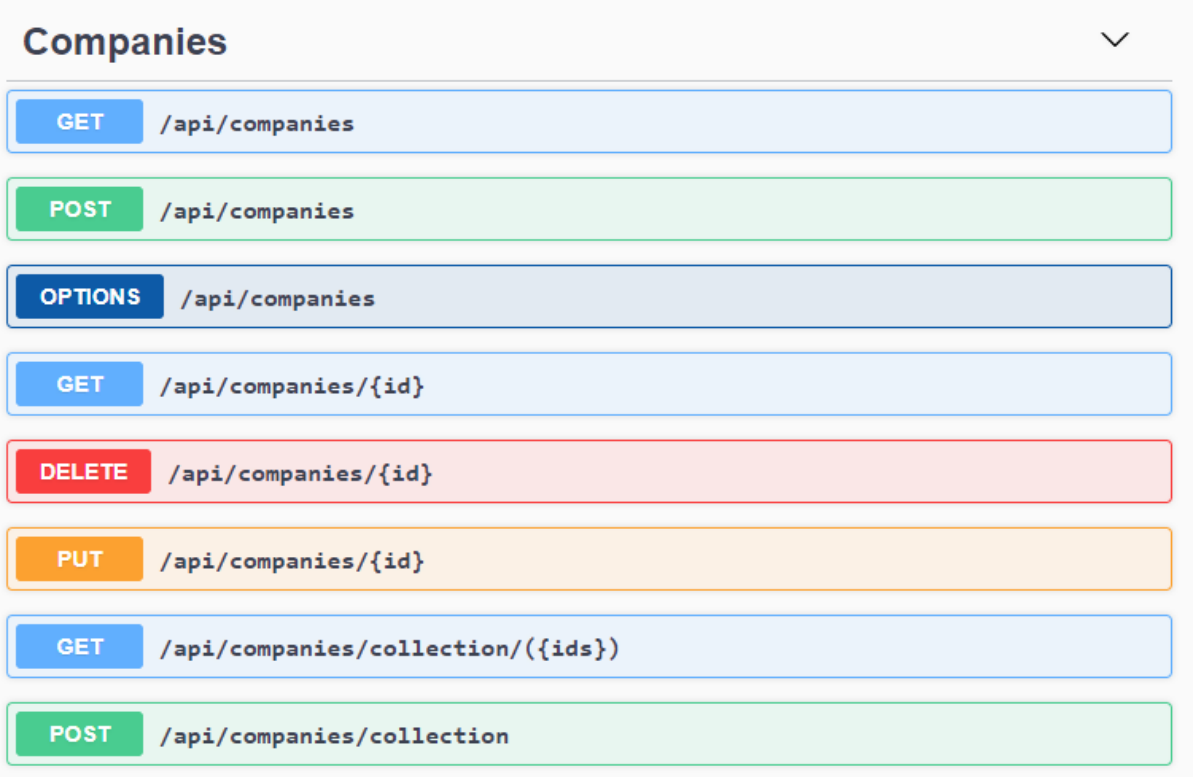
Now, we can start our app, open the browser, and navigate to `https://localhost:5001/swagger/v1/swagger.json`. Once the page is up, you are going to see a json document containing all the controllers and actions without the v2 companies controller. Of course, if you change

v1 to v2 in the URL, you are going to see all the controllers — including v2 companies, but without v1 companies.

Additionally, let's navigate to `https://localhost:5001/swagger/index.html`:

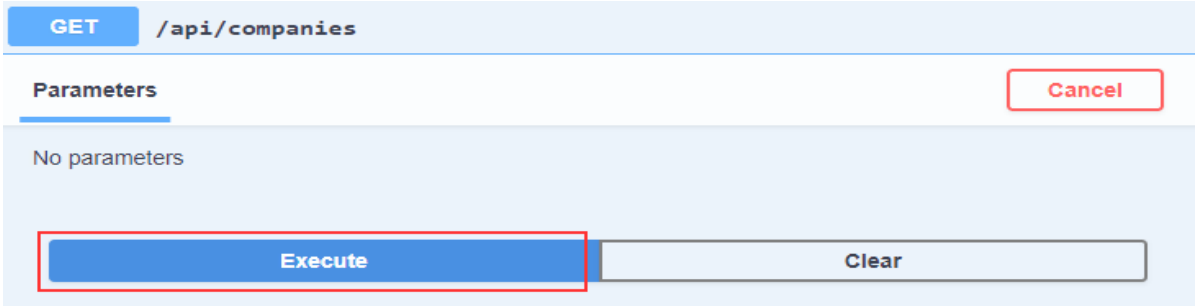


If we click on a specific controller to expand its details, we are going to see all the actions inside:

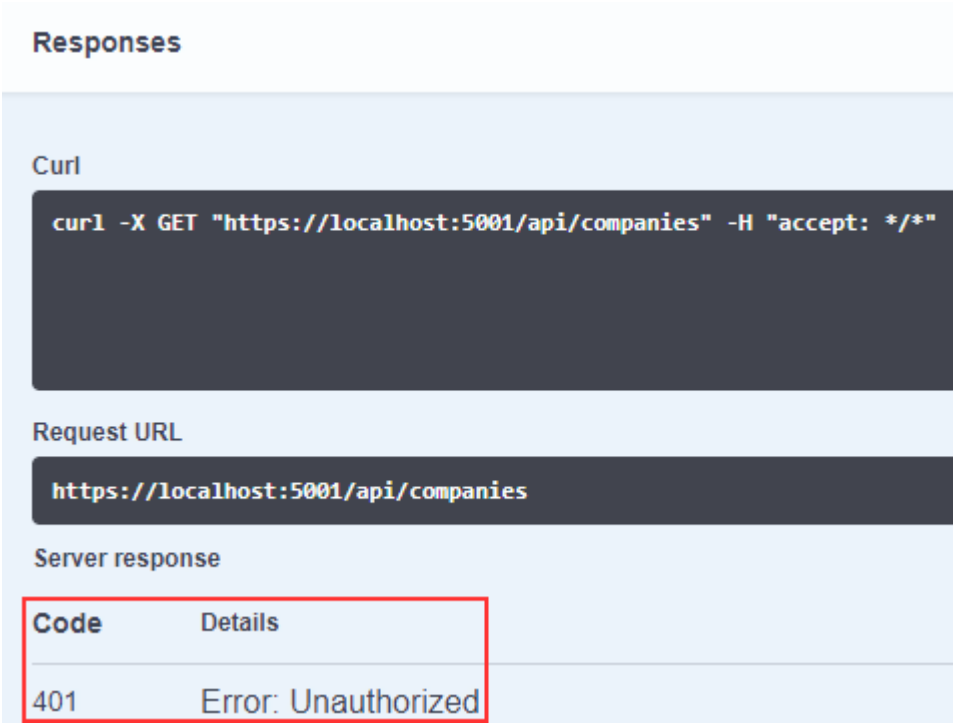


Once we click on an action method, we can see detailed information like parameters, response, and example values. There is also an option to try out each of those action methods by clicking the `Try it out` button.

So, let's try it with the /api/companies action:



Once we click the Execute button, we are going to see that we get our response:



And this is an expected response. We are not authorized. To enable authorization, we have to add some modifications.

Adding Authorization Support

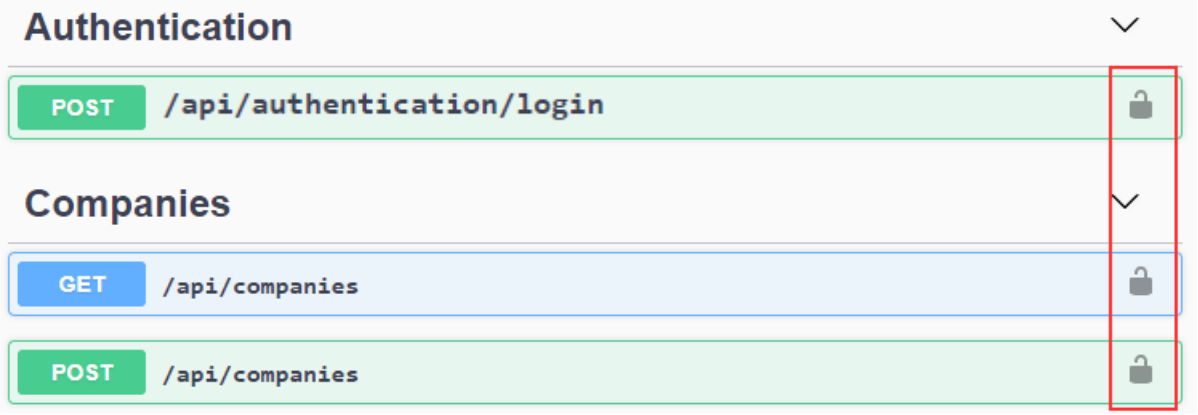
To add authorization support, we need to modify the `ConfigureSwagger` method:

```
public static void ConfigureSwagger(this IServiceCollection services)
{
    services.AddSwaggerGen(s =>
    {
        s.SwaggerDoc("v1", new OpenApiInfo { Title = "Code Maze API", Version = "v1"
});
        s.SwaggerDoc("v2", new OpenApiInfo { Title = "Code Maze API", Version = "v2"
});

        s.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
        {
            In = ParameterLocation.Header,
            Description = "Place to add JWT with Bearer",
            Name = "Authorization",
            Type = SecuritySchemeType.ApiKey,
            Scheme = "Bearer"
        });

        s.AddSecurityRequirement(new OpenApiSecurityRequirement()
        {
            {
```

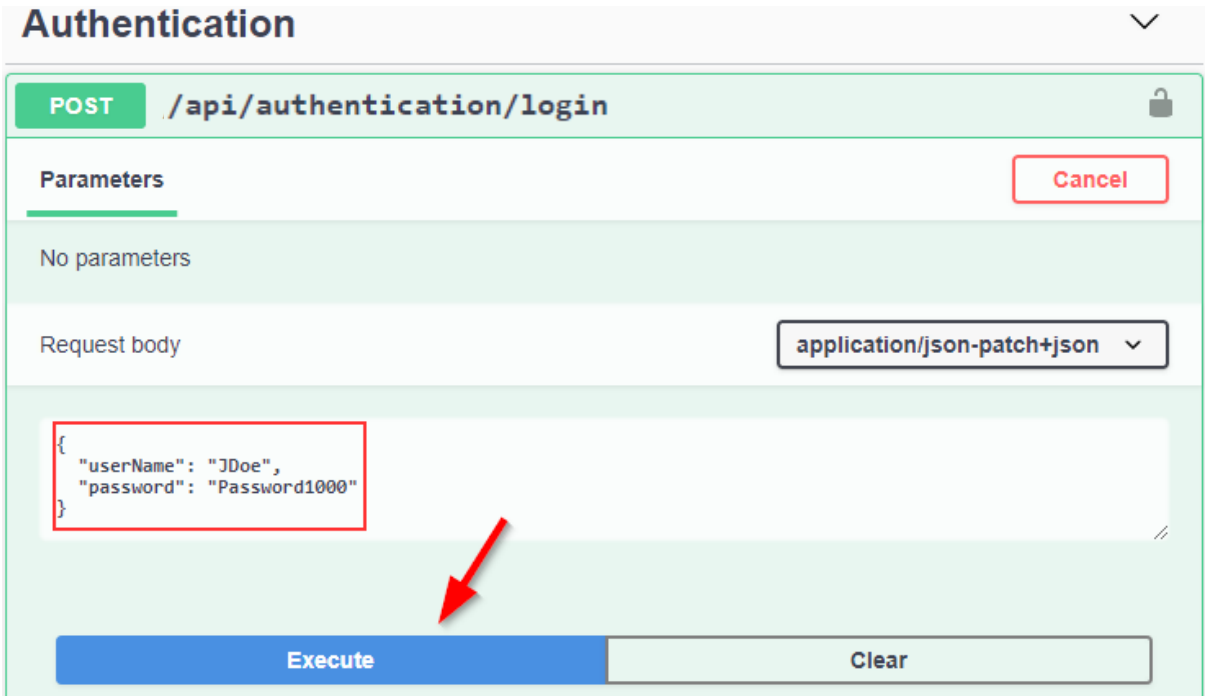
```
                new OpenApiSecurityScheme
                {
                    Reference = new OpenApiReference
                    {
                        Type = ReferenceType.SecurityScheme,
                        Id = "Bearer"
                    },
                    Name = "Bearer",
                },
                new List<string>()
            }
        });
    });
}
```

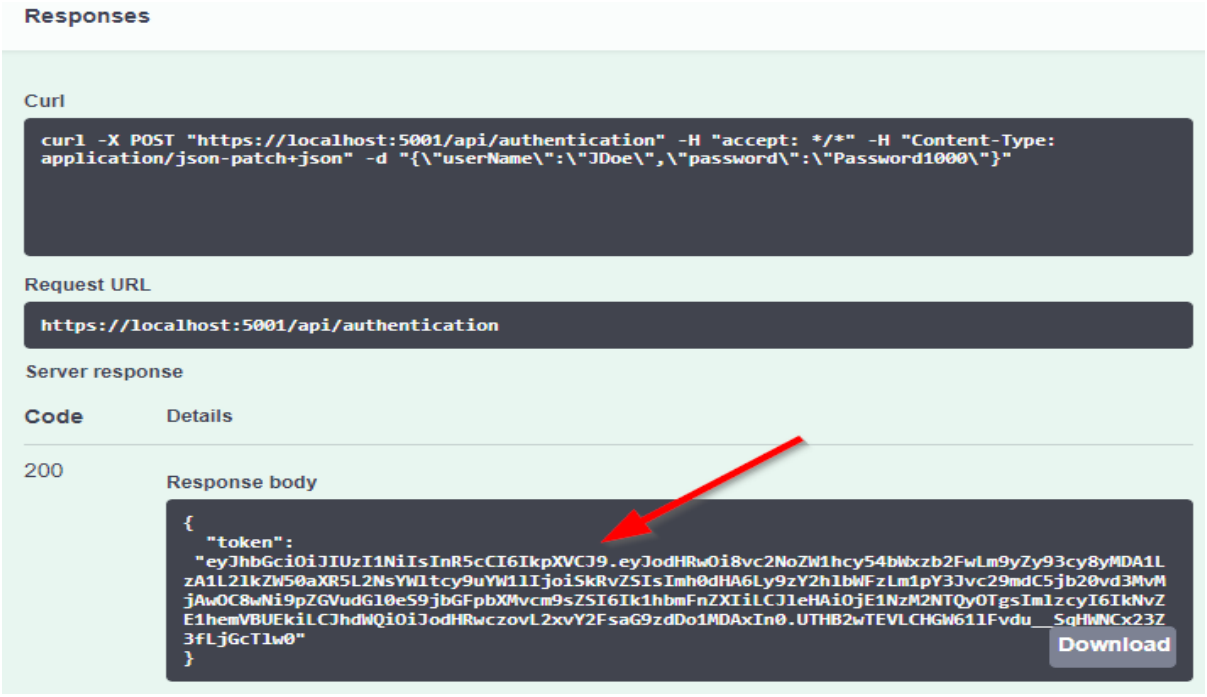
With this modification, we are adding the security definition in our swagger configuration. Now, we can start our app again and navigate to the index.html page.

The first thing we are going to notice is the Authorize options for requests:



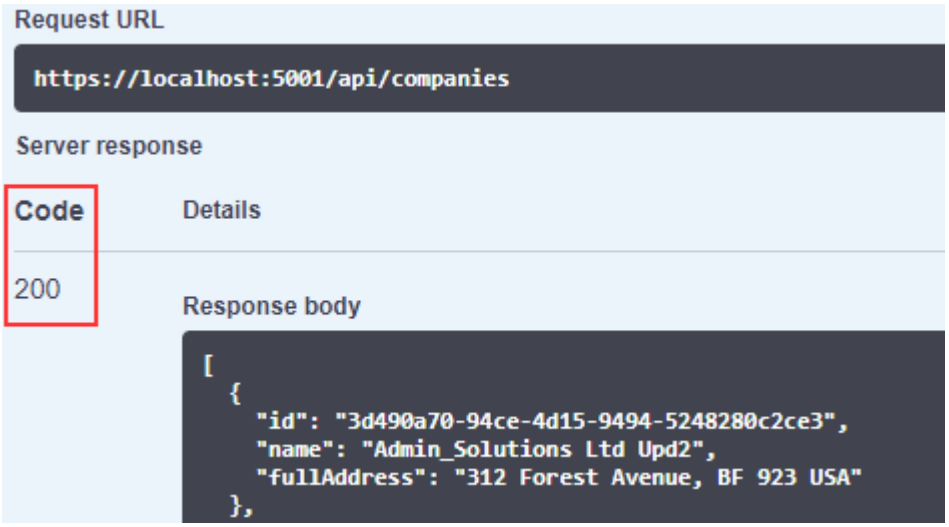
We are going to use that in a moment. But let's get our token first. For that, let's open the api/authentication/login action, click try it out, add credentials, and copy the received token:





Once we have copied the token, we are going to click on the authorization button for the /api/companies request, paste it with the Bearer in front of it, and click Authorize:

After authorization, we are going to click on the Close button and try our request:



And we get our response. Excellent job.

Extending Swagger Configuration

Swagger provides options for extending the documentation and customizing the UI. Let's explore some of those.

First, let's see how we can specify the API info and description. The configuration action passed to the `AddSwaggerGen()` method adds

information such as Contact, License, and Description. Let's provide some values for those:

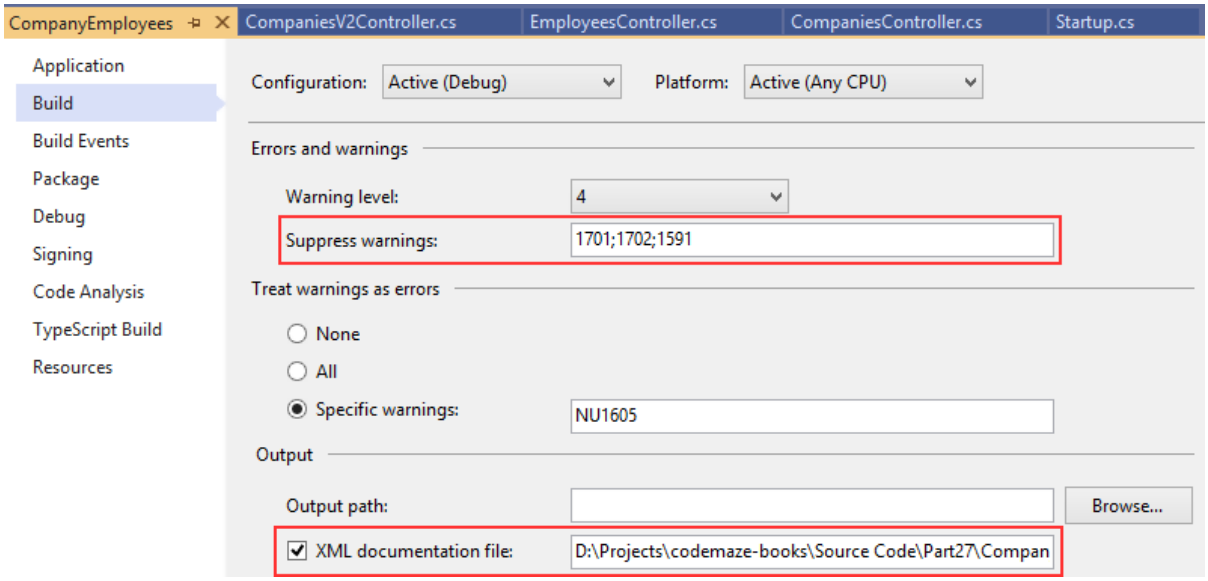
```
s.SwaggerDoc("v1", new OpenApiInfo
{
    Title = "Code Maze API",
    Version = "v1",
    Description = "CompanyEmployees API by CodeMaze",
    TermsOfService = new Uri("https://example.com/terms"),
    Contact = new OpenApiContact
    {
        Name = "John Doe",
        Email = "John.Doe@gmail.com",
        Url = new Uri("https://twitter.com/johndoe"),
    },
    License = new OpenApiLicense
    {
        Name = "CompanyEmployees API LICX",
        Url = new Uri("https://example.com/license"),
    }
});
…
```

We have implemented this just for the first version, but you get the point.

Now, let's run the application once again and explore the Swagger UI:

For enabling XML comments, we need to do the following steps:

- In the Build tab of the main project properties, check the box labeled XML documentation file. Let's keep the auto-generated file path.
- Suppress warning 1591, which will now give warnings about any method, class, or field that doesn't have triple-slash comments.



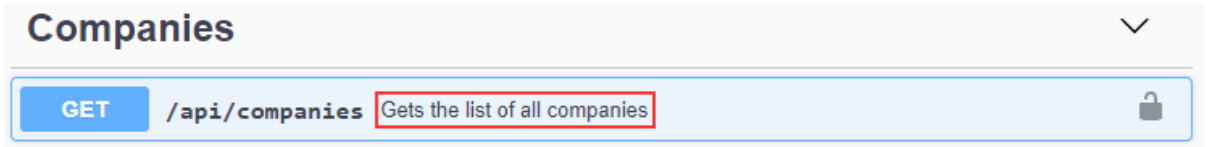
Now, let's modify our configuration:

```
s.SwaggerDoc("v2", new OpenApiInfo { Title = "Code Maze API", Version = "v2" });

var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
s.IncludeXmlComments(xmlPath);
```

Next, adding triple-slash comments to the action method enhances the Swagger UI by adding a description to the section header:

```
/// <summary>
/// Gets the list of all companies
/// </summary>
/// <returns>The companies list</returns>
[HttpGet(Name = "GetCompanies"), Authorize(Roles = "Manager")]
public async Task<IActionResult> GetCompanies()
```

And this is the result:



The developers who consume our APIs are usually more interested in what it returns — specifically the response types and error codes. Hence, it is very important to describe our response types. These are denoted using XML comments and data annotations.

Let’s enhance the response types a little bit:

```
/// <summary>
/// Creates a newly created company
/// </summary>
/// <param name="company"></param>
/// <returns>A newly created company</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
/// <response code="422">If the model is invalid</response>
[HttpPost(Name = "CreateCompany")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(422)]
```

Code	Description	Links
201	Returns the newly created item	*No links*
400	If the item is null Media type: text/plain Example Value | Schema	*No links*
422	If the model is invalid	*No links*

```
{
  "type": "string",
  "title": "string",
  "status": 0,
  "detail": "string",
  "instance": "string",
  "extensions": {
    "additionalProp1": {},
    "additionalProp2": {},
    "additionalProp3": {}
  }
}
```

DEPLOYMENT TO IIS

Before we start the deployment process, we would like to point out one important thing. We should always try to deploy an application on at least a local machine to somehow simulate the production environment as soon as we start with development. That way, we are able to observe how the application behaves in a production environment from the beginning of the development process.

That leads us to the conclusion that the deployment process should not be the last step of the application's lifecycle. We should deploy our application to the staging environment as soon as we start building it.

That said, let's start with the deployment process.

Creating Publish Files

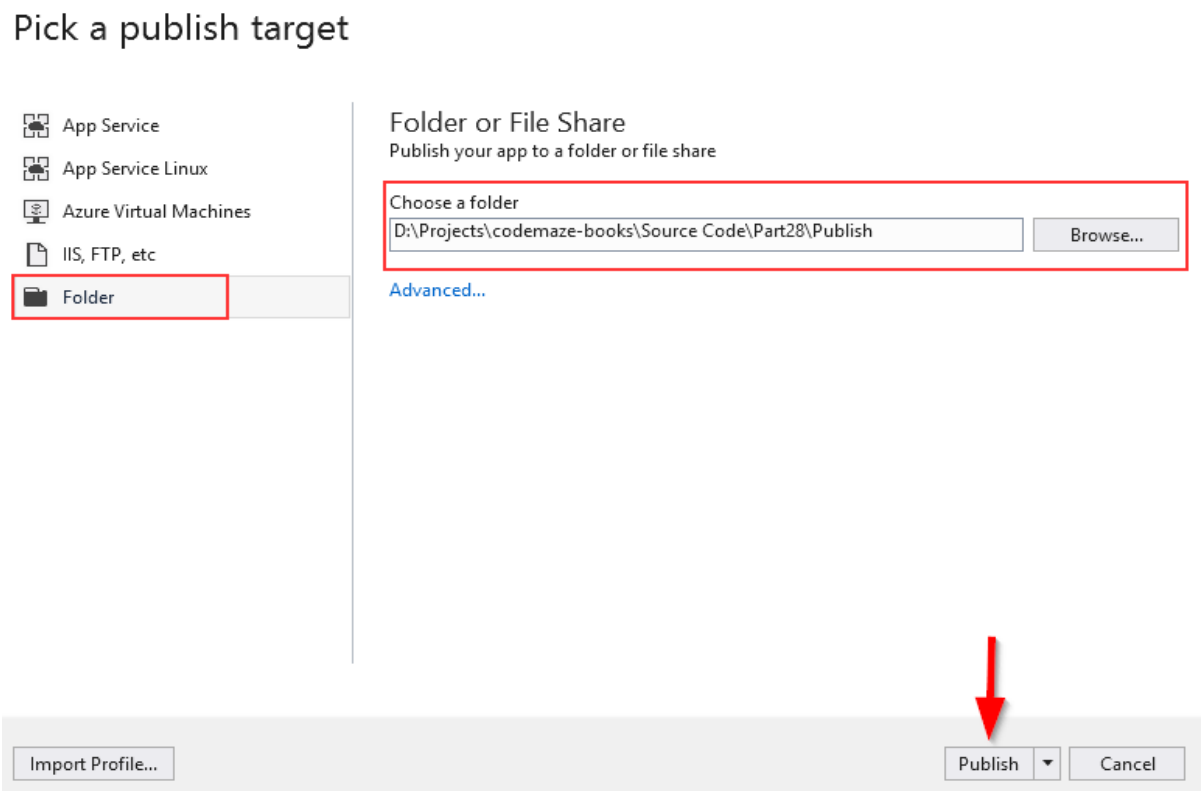
Before we create publish files, we have to do one thing in our project. In the previous section, we integrated Swagger in our application and it is using an xml file for the xml documentation. What we have to do is to enable that file to be published with all the other published files from our application.

To do that, let's find the `CompanyEmployees.xml` file in the main project, right-click on it, and choose `Properties`. In the next window, for the `Copy to Output Directory` option, we are going to choose `Copy always`.

That's it. We can move on.

Let's create a folder on the local machine with the name `Publish`. Inside that folder, we want to place all of our files for the deployment. After the folder creation, let's right-click on the main project in the Solution Explorer window and click publish option:

In the "Pick a publish target" window, we are going to choose the Folder option and point to the location of the Publish folder we just created:



Visual Studio is going to do its job and publish the required files in the specified folder.

Windows Server Hosting Bundle

Prior to any further action, let's install the .NET Core Windows Server Hosting bundle on our system to install .NET Core Runtime. Furthermore, with this bundle, we are installing the .NET Core Library and the ASP.NET Core Module. This installation will create a reverse proxy between IIS and the Kestrel server, which is crucial for the deployment process.

If you have a problem with missing SDK after installing the Hosting Bundle, follow this solution suggested by Microsoft:

Installing the .NET Core Hosting Bundle modifies the PATH when it installs the .NET Core runtime to point to the 32-bit (x86) version of .NET Core (C:\Program Files (x86)\dotnet\). This can result in missing SDKs when the 32-bit (x86) .NET Core dotnet command is used (No .NET Core SDKs were detected). To resolve this problem, move C:\Program Files\dotnet\ to a position before C:\Program Files (x86)\dotnet\ on the PATH environment variable.

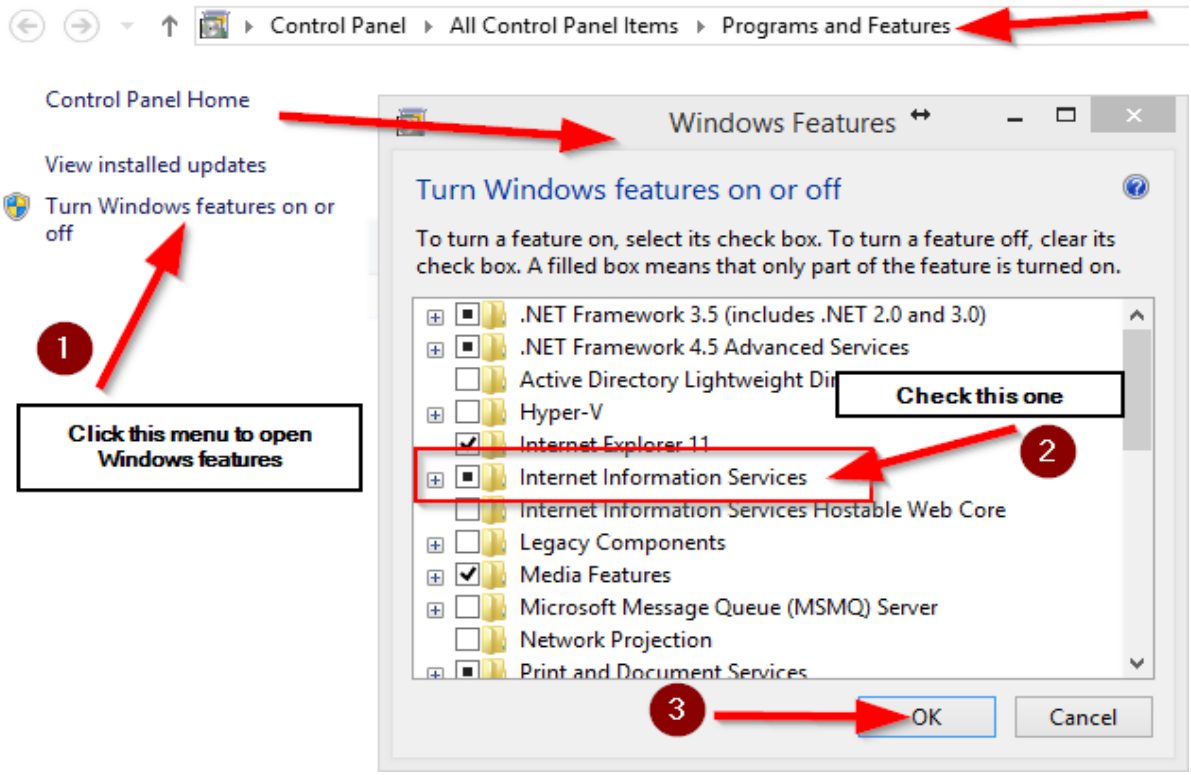
After the installation, we are going to locate the Windows hosts file on C:\Windows\System32\drivers\etc and add the following record at the end of the file:

```
127.0.0.1 www.companyemployees.codemaze
```

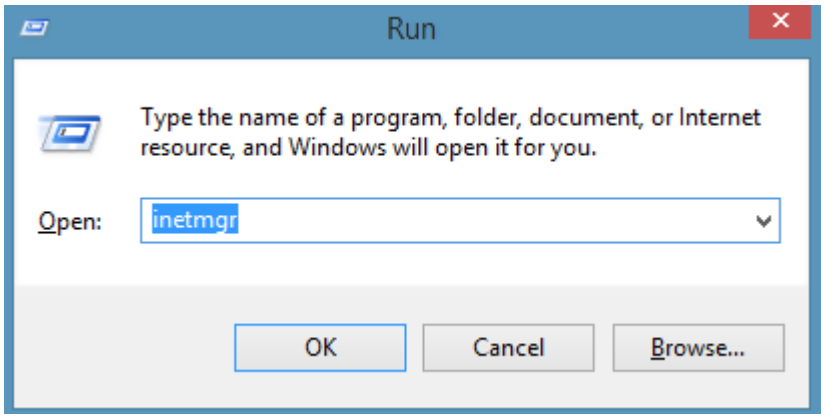
After that, we are going to save the file.

Installing IIS

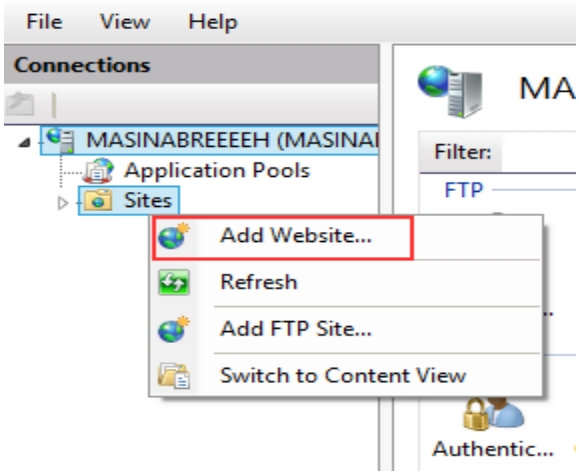
If you don't have IIS installed on your machine, you need to install it by opening ControlPanel and then Programs and Features:



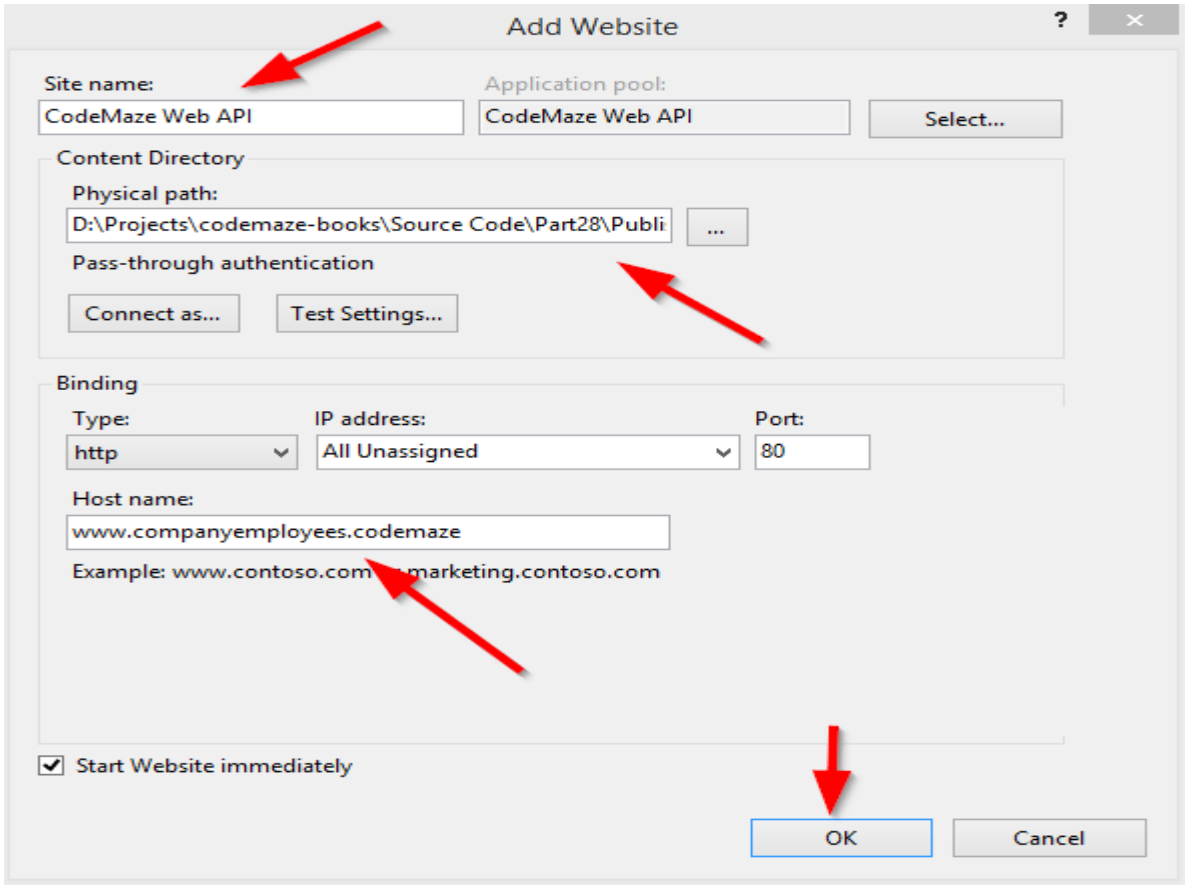
After the IIS installation finishes, let's open the Run window (windows key + R) and type: `inetmgr` to open the IIS manager:



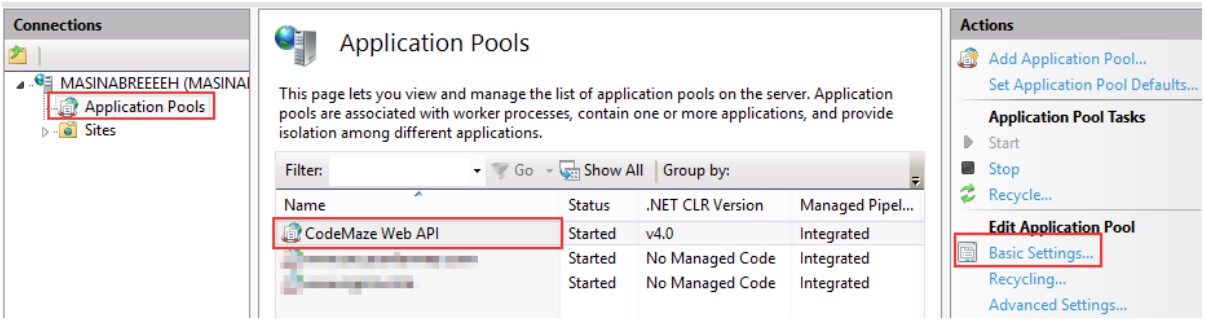
Now, we can create a new website:



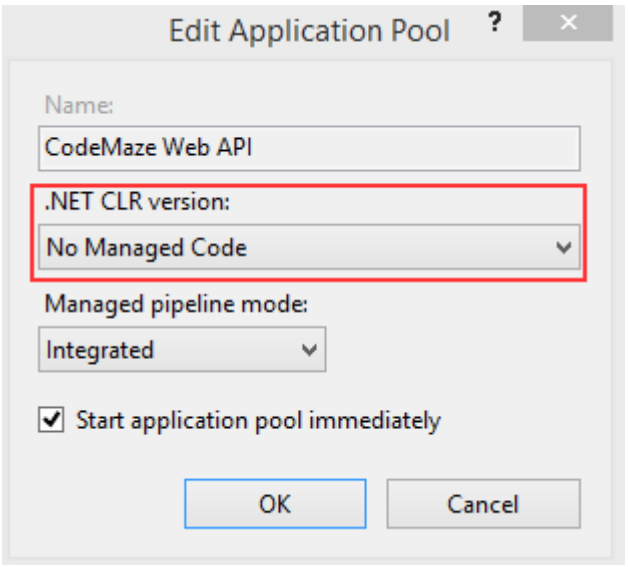
In the next window, we need to add a name to our site and a path to the published files:



After this step, we are going to have our site inside the “sites” folder in the IIS Manager. Additionally, we need to set up some basic settings for our application pool:



After we click on the `Basic Settings` link, let's configure our application pool:



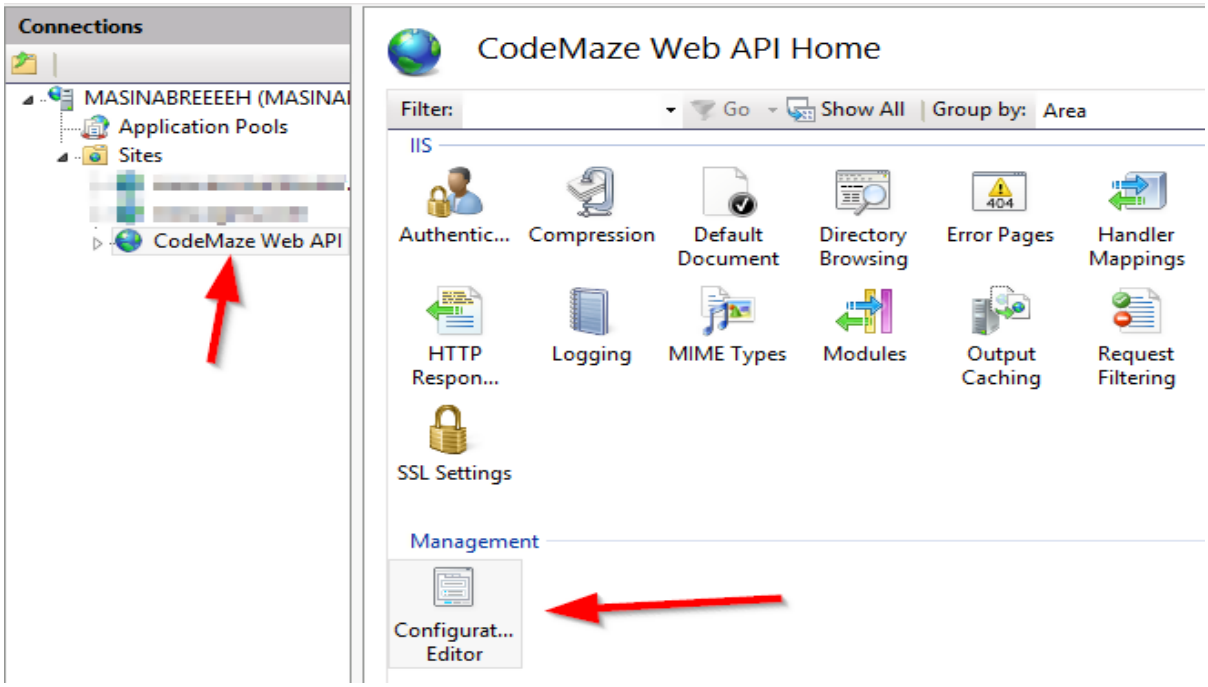
ASP.NET Core runs in a separate process and manages the runtime. It doesn't rely on loading the desktop CLR (.NET CLR). The Core Common Language Runtime for .NET Core is booted to host the app in the worker process. Setting the .NET CLR version to No Managed Code is optional but recommended.

Our website and the application pool should be started automatically.

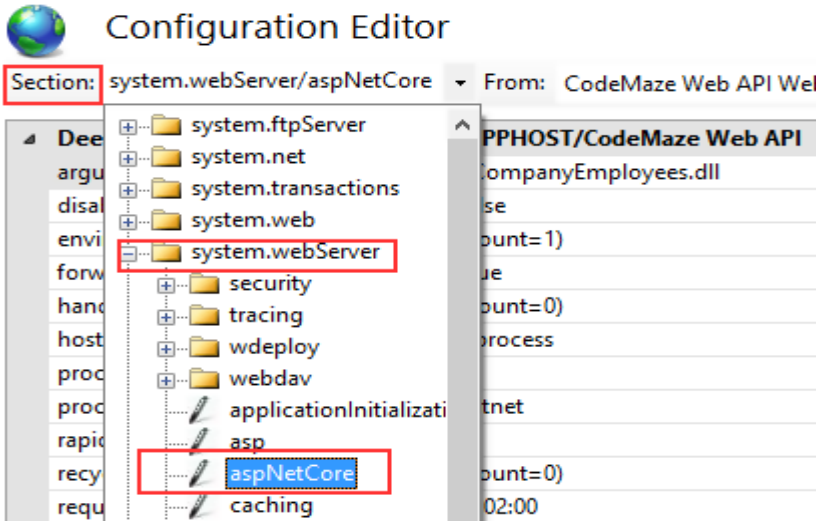
Configuring Environment File

In the section where we configured JWT, we had to use a secret key that we placed in the environment file. Now, we have to provide to IIS the name of that key and the value as well.

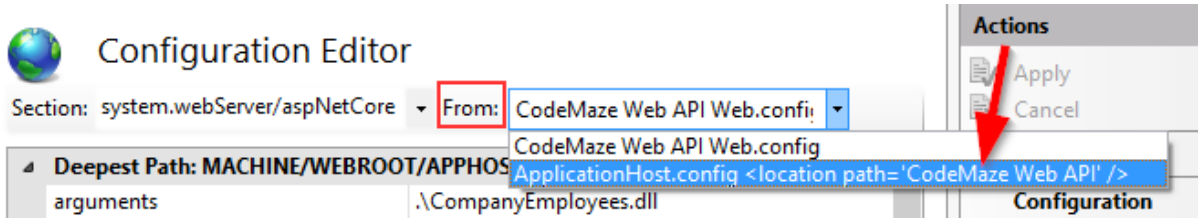
The first step is to click on our site in IIS and open `Configuration Editor`:



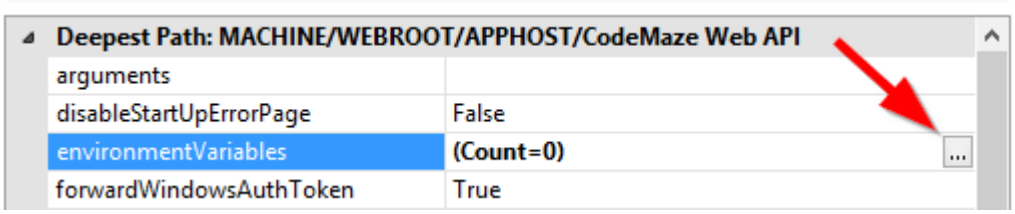
Then, in the section box, we are going to choose `system.webServer/aspNetcore`:



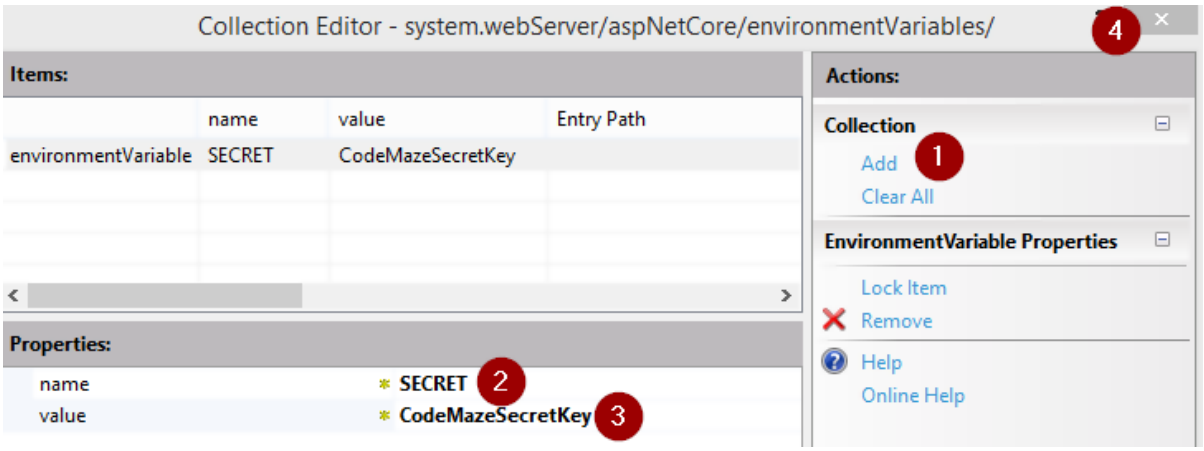
From the "From" combo box, we are going to choose `ApplicationHost.config`:



After that, we are going to select environment variables:



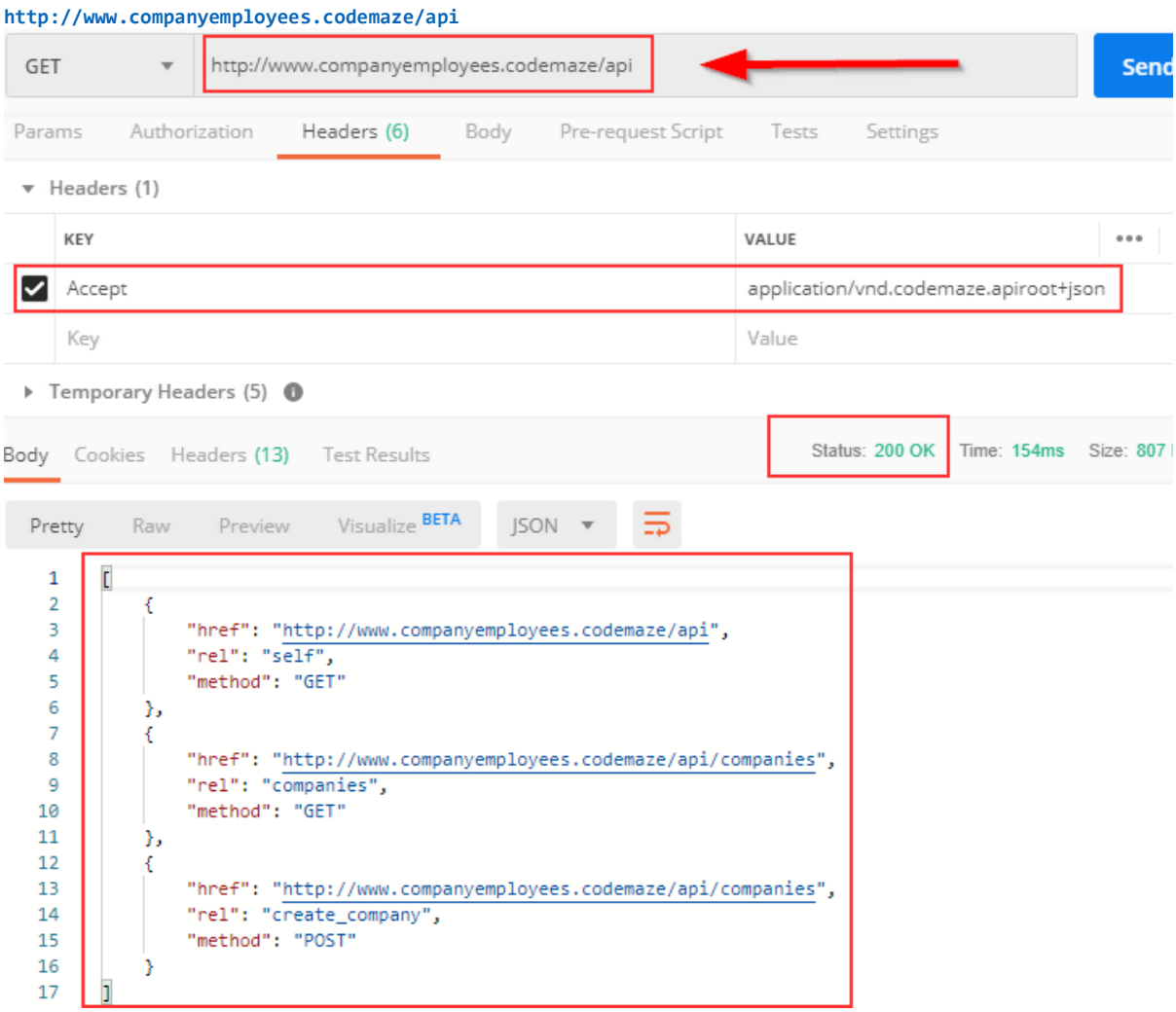
Click Add and type the name and the value of our variable:



As soon as we click the close button, we should click apply in the next window, restart our application in IIS, and we are good to go.

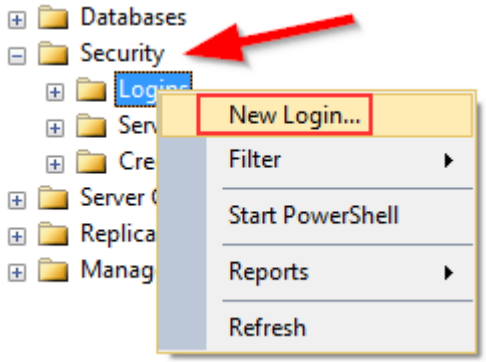
Testing Deployed Application

Let's open Postman and send a request for the Root document:

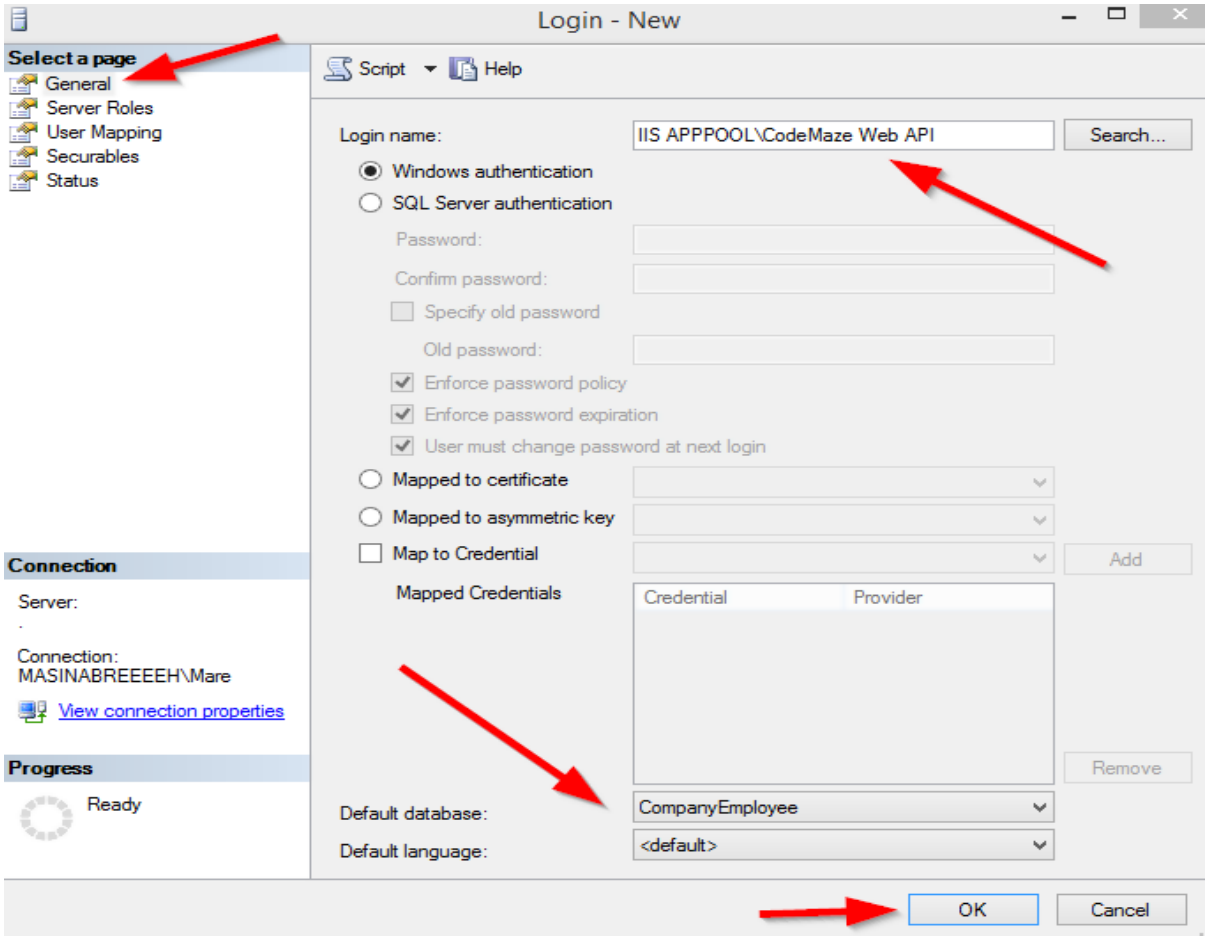


We can see that our API is working as expected. If it's not, and you have a problem related to web.config in IIS, try reinstalling the Server Hosting Bundle package.

But we still have one more thing to do. We have to add a login to the SQL Server for `IIS APPPOOL\CodeMaze Web Api` and grant permissions to the database. So, let's open the SQL Server Management Studio and add a new login:



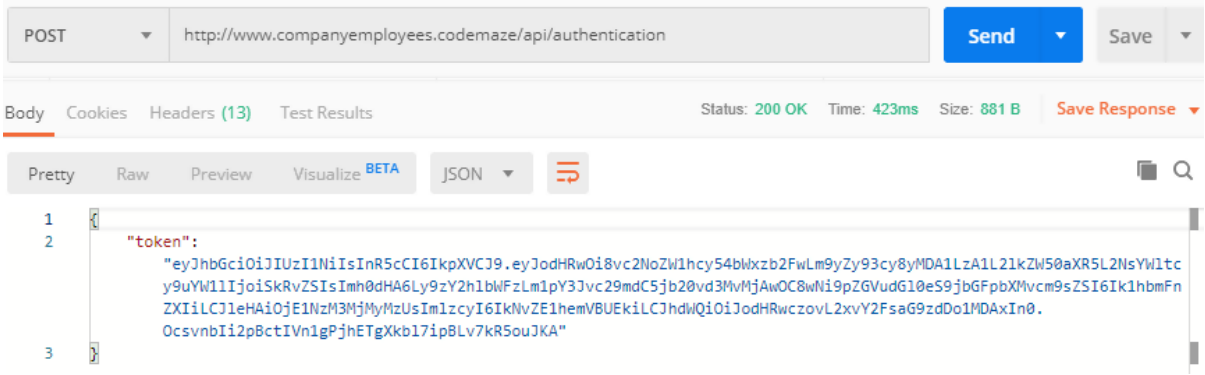
In the next window, we are going to add our user:



After that, we are going to expand the Logins folder, right-click on our user, and choose Properties. There, under UserMappings, we have to select the CompanyEmployee database and grant the dbwriter and dbreader roles.

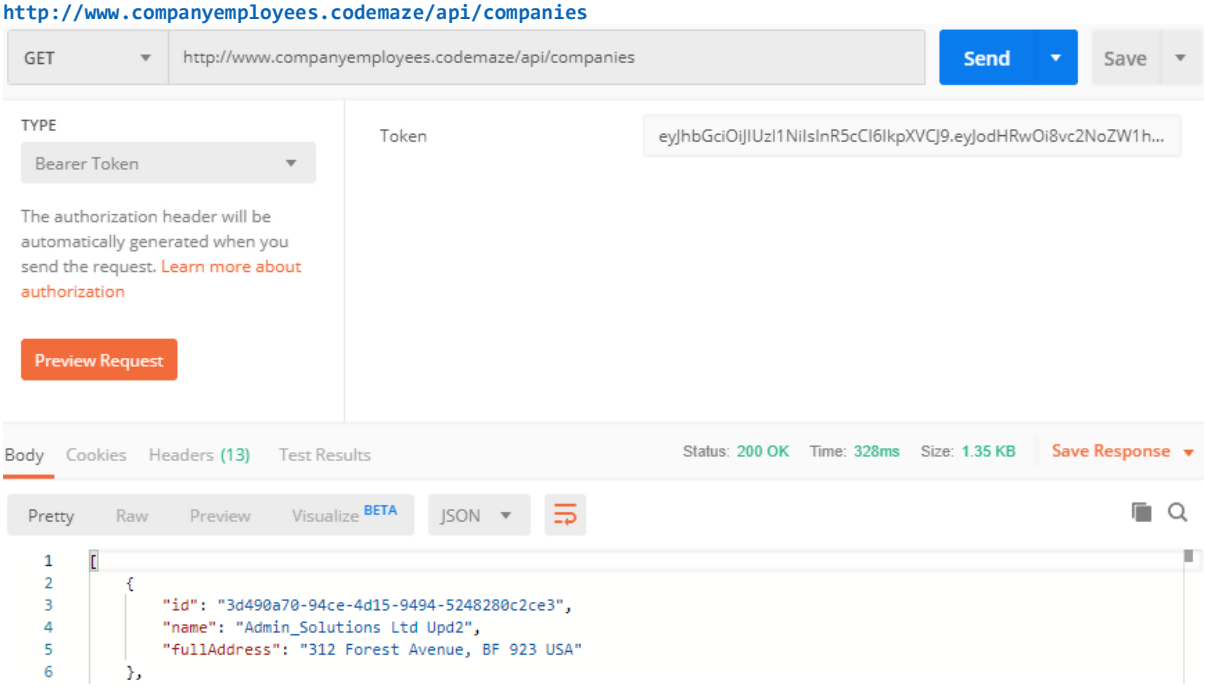
Now, we can try to send the Authentication request:

http://www.companyemployees.codemaze/api/authentication



Excellent; we have our token. Now, we can send the request to the GetCompanies action with the generated token:

http://www.companyemployees.codemaze/api/companies



And there we go. Our API is published and working as expected.