

PREMIUM

Marinko Spasojevic

Vladimir Pecanac

ULTIMATE ASP.NET CORE WEB API

SECOND EDITION

From **Zero** To
Six-Figure Backend Developer



Made with ❤ by:





Ultimate ASP.NET Core Web API



TABLE OF CONTENTS

1 PROJECT CONFIGURATION	1
1.1 Creating a New Project	1
1.2 launchSettings.json File Configuration	2
1.3 Program.cs Class Explanations.....	4
1.4 Extension Methods and CORS Configuration.....	7
1.5 IIS Configuration	9
1.6 Additional Code in the Program Class.....	11
1.7 Environment-Based Settings	12
1.8 ASP.NET Core Middleware	13
1.8.1 Creating a First Middleware Component.....	16
1.8.2 Working with the Use Method.....	18
1.8.3 Using the Map and MapWhen Methods	20
1.8.4 Using MapWhen Method	21
2 CONFIGURING A LOGGING SERVICE	23
2.1 Creating the Required Projects	23
2.2 Creating the ILoggerManager Interface and Installing NLog	24
2.3 Implementing the Interface and Nlog.Config File.....	26
2.4 Configuring Logger Service for Logging Messages	27
2.5 DI, IoC, and Logger Service Testing	29
3 ONION ARCHITECTURE IMPLEMENTATION	31
3.1 About Onion Architecture	32



3.1.1	Advantages of the Onion Architecture	33
3.1.2	Flow of Dependencies.....	33
3.2	Creating Models	34
3.3	Context Class and the Database Connection.....	36
3.4	Migration and Initial Data Seed.....	39
3.5	Repository Pattern Logic	42
3.6	Repository User Interfaces and Classes	44
3.7	Creating a Repository Manager	45
3.8	Adding a Service Layer.....	47
3.9	Registering RepositoryContext at a Runtime.....	50
4	HANDLING GET REQUESTS	52
4.1	Controllers and Routing in WEB API.....	52
4.2	Naming Our Resources.....	57
4.3	Getting All Companies From the Database	57
4.4	Testing the Result with Postman.....	61
4.5	DTO Classes vs. Entity Model Classes	62
4.6	Using AutoMapper in ASP.NET Core.....	65
5	GLOBAL ERROR HANDLING	70
5.1	Handling Errors Globally with the Built-In Middleware.....	70
5.2	Program Class Modification	72
5.3	Testing the Result	73



6 GETTING ADDITIONAL RESOURCES	75
6.1 Getting a Single Resource From the Database.....	75
6.1.1 Handling Invalid Requests in a Service Layer	77
6.2 Parent/Child Relationships in Web API	80
6.3 Getting a Single Employee for Company.....	83
7 CONTENT NEGOTIATION	87
7.1 What Do We Get Out of the Box?.....	87
7.2 Changing the Default Configuration of Our Project	88
7.3 Testing Content Negotiation.....	89
7.4 Restricting Media Types	91
7.5 More About Formatters	92
7.6 Implementing a Custom Formatter	93
8 METHOD SAFETY AND METHOD IDEMPOTENCY	96
9 CREATING RESOURCES	98
9.1 Handling POST Requests	98
9.2 Code Explanation	101
9.2.1 Validation from the ApiController Attribute.....	102
9.3 Creating a Child Resource	105
9.4 Creating Children Resources Together with a Parent	108
9.5 Creating a Collection of Resources	109
9.6 Model Binding in API.....	115



10 WORKING WITH DELETE REQUESTS.....	119
10.1 Deleting a Parent Resource with its Children	121
11 WORKING WITH PUT REQUESTS	123
11.1 Updating Employee	123
11.1.1 About the Update Method from the RepositoryBase Class	127
11.2 Inserting Resources while Updating One	127
12 WORKING WITH PATCH REQUESTS	130
12.1 Applying PATCH to the Employee Entity	131
13 VALIDATION	138
13.1 ModelState, Rerun Validation, and Built-in Attributes	138
13.1.1 Rerun Validation	139
13.1.2 Built-in Attributes	140
13.2 Custom Attributes and IValidatableObject	141
13.3 Validation while Creating Resource	143
13.3.1 Validating Int Type	146
13.4 Validation for PUT Requests	147
13.5 Validation for PATCH Requests.....	149
14 ASYNCHRONOUS CODE.....	154
14.1 What is Asynchronous Programming?	154
14.2 Async, Await Keywords and Return Types.....	156
14.2.1 Return Types of the Asynchronous Methods	158
14.2.2 The IRepositoryBase Interface and the RepositoryBase Class Explanation	159



14.3 Modifying the ICompanyRepository Interface and the CompanyRepository Class	159
14.4 IRepositoryManager and RepositoryManager Changes	160
14.5 Updating the Service layer	161
14.6 Controller Modification.....	163
14.7 Continuation in Asynchronous Programming	166
14.8 Common Pitfalls	167
15 ACTION FILTERS	169
15.1 Action Filters Implementation.....	169
15.2 The Scope of Action Filters	170
15.3 Order of Invocation.....	171
15.4 Improving the Code with Action Filters	173
15.5 Validation with Action Filters	173
15.6 Refactoring the Service Layer	176
16 PAGING	181
16.1 What is Paging?	181
16.2 Paging Implementation.....	182
16.3 Concrete Query	185
16.4 Improving the Solution	187
16.4.1 Additional Advice	190
17 FILTERING	192
17.1 What is Filtering?	192



17.2 How is Filtering Different from Searching?.....	193
17.3 How to Implement Filtering in ASP.NET Core Web API	194
17.4 Sending and Testing a Query.....	196
18 SEARCHING	199
18.1 What is Searching?.....	199
18.2 Implementing Searching in Our Application	199
18.3 Testing Our Implementation	201
19 SORTING	204
19.1 What is Sorting?.....	204
19.2 How to Implement Sorting in ASP.NET Core Web API	206
19.3 Implementation – Step by Step.....	208
19.4 Testing Our Implementation	210
19.5 Improving the Sorting Functionality	211
20 DATA SHAPING	213
20.1 What is Data Shaping?	213
20.2 How to Implement Data Shaping	214
20.3 Step-by-Step Implementation	216
20.4 Resolving XML Serialization Problems.....	221
21 SUPPORTING HATEOAS	224
21.1 What is HATEOAS and Why is it so Important?.....	224
21.1.1 Typical Response with HATEOAS Implemented	225



21.1.2	What is a Link?.....	225
21.1.3	Pros/Cons of Implementing HATEOAS	226
21.2	Adding Links in the Project	227
21.3	Additional Project Changes	228
21.4	Adding Custom Media Types.....	230
21.4.1	Registering Custom Media Types	230
21.4.2	Implementing a Media Type Validation Filter	231
21.5	Implementing HATEOAS.....	233
22	WORKING WITH OPTIONS AND HEAD REQUESTS.....	241
22.1	OPTIONS HTTP Request	241
22.2	OPTIONS Implementation	241
22.3	Head HTTP Request.....	243
22.4	HEAD Implementation.....	243
23	ROOT DOCUMENT	245
23.1	Root Document Implementation	245
24	VERSIONING APIs	250
24.1	Required Package Installation and Configuration	250
24.2	Versioning Examples	251
24.2.1	Using Query String	253
24.2.2	Using URL Versioning	254
24.2.3	HTTP Header Versioning	255
24.2.4	Deprecating Versions	256
24.2.5	Using Conventions	257
25	CACHING.....	258



25.1 About Caching	258
25.1.1 Cache Types	258
25.1.2 Response Cache Attribute	259
25.2 Adding Cache Headers.....	259
25.3 Adding Cache-Store.....	261
25.4 Expiration Model	264
25.5 Validation Model.....	266
25.6 Supporting Validation.....	267
25.6.1 Configuration	268
25.7 Using ETag and Validation.....	270
26 RATE LIMITING AND THROTTLING	273
26.1 Implementing Rate Limiting.....	273
27 JWT, IDENTITY, AND REFRESH TOKEN	277
27.1 Implementing Identity in ASP.NET Core Project.....	277
27.2 Creating Tables and Inserting Roles.....	280
27.3 User Creation	281
27.4 Big Picture	287
27.5 About JWT.....	287
27.6 JWT Configuration.....	289
27.7 Protecting Endpoints	291
27.8 Implementing Authentication	292
27.9 Role-Based Authorization.....	297



28 REFRESH TOKEN.....	300
28.1 Why Do We Need a Refresh Token	302
28.2 Refresh Token Implementation	303
28.3 Token Controller Implementation	307
29 BINDING CONFIGURATION AND OPTIONS PATTERN.....	311
29.1 Binding Configuration	312
29.2 Options Pattern.....	314
29.2.1 Using IOptions	315
29.2.2 IOptionsSnapshot and IOptionsMonitor.....	317
30 DOCUMENTING API WITH SWAGGER	320
30.1 About Swagger.....	320
30.2 Swagger Integration Into Our Project.....	321
30.3 Adding Authorization Support	325
30.4 Extending Swagger Configuration	328
31 DEPLOYMENT TO IIS	332
31.1 Creating Publish Files.....	332
31.2 Windows Server Hosting Bundle	334
31.3 Installing IIS.....	335
31.4 Configuring Environment File	338
31.5 Testing Deployed Application	339
32 BONUS 1 - RESPONSE PERFORMANCE IMPROVEMENTS.....	343



32.1 Adding Response Classes to the Project	343
32.2 Service Layer Modification	345
32.3 Controller Modification	347
32.4 Testing the API Response Flow	349
 33 BONUS 2 - INTRODUCTION TO CQRS AND MEDIATR WITH ASP.NET CORE WEB API	 352
33.1 About CQRS and Mediator Pattern.....	352
33.1.1 CQRS	352
33.1.2 Advantages and Disadvantages of CQRS	354
33.1.3 Mediator Pattern.....	355
33.2 How MediatR facilitates CQRS and Mediator Patterns	356
33.3 Adding Application Project and Initial Configuration.....	356
33.4 Requests with MediatR.....	359
33.5 Commands with MediatR.....	365
33.5.1 Update Command	367
33.5.2 Delete Command	369
33.6 MediatR Notifications	370
33.7 MediatR Behaviors	373
33.7.1 Adding Fluent Validation	374
33.7.2 Creating Decorators with MediatR PipelineBehavior	375
33.7.3 Validating null Object	379



1 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.

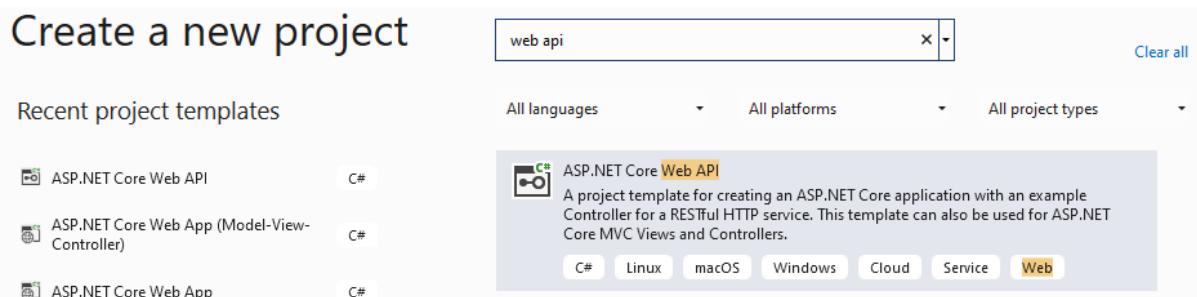
To be able to develop good applications, we need to understand how to configure our application and its services first.

In this section, we'll learn about configuration in the Program 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.

1.1 Creating a New Project

Let's open Visual Studio, we are going to use VS 2022, and create a new ASP.NET Core Web API Application:



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



Configure your new project

ASP.NET Core Web Application C# Windows Linux macOS Web

Project name

CompanyEmployees

Location

E:\CodeMaze\CompanyEmployees



Solution name ⓘ

CompanyEmployees

Place solution and project in the same directory

Next, we want to choose a .NET 6.0 from the dropdown list. Also, we don't want to enable OpenAPI support right now. We'll do that later in the book on our own. Now we can proceed by clicking the Create button and the project will start initializing:

Additional information

ASP.NET Core Web API C# Linux macOS Windows Cloud Service Web

Framework ⓘ

.NET 6.0 (Long-term support)

Authentication type ⓘ

None

Configure for HTTPS ⓘ

Enable Docker ⓘ

Docker OS ⓘ

Linux

Use controllers (uncheck to use minimal APIs) ⓘ

Enable OpenAPI support ⓘ

Back

Create

1.2 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": "https://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:1629",
      "sslPort": 44370
    }
  },
  "profiles": {
    "CompanyEmployees": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "launchUrl": "weatherforecast",
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": false,
      "launchUrl": "weatherforecast",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

This is convenient since we are developing a Web API project and we don't 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 HTTPS (`localhost:5001`), and one for HTTP (`localhost:5000`).



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

NOTE: 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. 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.

1.3 Program.cs Class Explanations

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

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Compared to the `Program.cs` class from .NET 5, there are some major changes. Some of the most obvious are:

- Top-level statements
- Implicit using directives



- No Startup class (on the project level)

“Top-level statements” means the compiler generates the namespace, class, and method elements for the main program in our application. We can see that we don’t have the class block in the code nor the **Main** method. All of that is generated for us by the compiler. Of course, we can add other functions to the **Program** class and those will be created as the local functions nested inside the generated **Main** method. Top-level statements are meant to simplify the entry point to the application and remove the extra “fluff” so we can focus on the important stuff instead.

“Implicit using directives” mean the compiler automatically adds a different set of using directives based on a project type, so we don’t have to do that manually. These using directives are stored in the **obj/Debug/net6.0** folder of our project under the name **CompanyEmployees.GlobalUsings.g.cs**:

```
// <auto-generated>
global using global::Microsoft.AspNetCore.Builder;
global using global::Microsoft.AspNetCore.Hosting;
global using global::Microsoft.AspNetCore.Http;
global using global::Microsoft.AspNetCore.Routing;
global using global::Microsoft.Extensions.Configuration;
global using global::Microsoft.Extensions.DependencyInjection;
global using global::Microsoft.Extensions.Hosting;
global using global::Microsoft.Extensions.Logging;
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Net.Http.Json;
global using global::System.Threading;

global using global::System.Threading.Tasks;
```

This means that we can use different classes from these namespaces in our project without adding using directives explicitly in our project files. Of course, if you don’t want this type of behavior, you can turn it off by visiting the project file and disabling the **ImplicitUsings** tag:

```
<ImplicitUsings>disable</ImplicitUsings>
```



By default, this is enabled in the .csproj file, and we are going to keep it like that.

Now, let's take a look at the code inside the **Program** class.

With this line of code:

```
var builder = WebApplication.CreateBuilder(args);
```

The application creates a **builder** variable of the type **WebApplicationBuilder**. The WebApplicationBuilder class is responsible for four main things:

- Adding Configuration to the project by using the **builder.Configuration** property
- Registering services in our app with the **builder.Services** property
- Logging configuration with the **builder.Logging** property
- Other **IHostBuilder** and **IWebHostBuilder** configuration

Compared to .NET 5 where we had a static **CreateDefaultBuilder** class, which returned the **IHostBuilder** type, now we have the static **CreateBuilder** method, which returns **WebApplicationBuilder** type.

Of course, as we see it, we don't have the **Startup** class with two familiar methods: **ConfigureServices** and **Configure**. Now, all this is replaced by the code inside the Program.cs file.

Since we don't have the **ConfigureServices** method to configure our services, we can do that right below the **builder** variable declaration. In the new template, there's even a comment section suggesting where we should start with service registration. A service is a reusable part of the code that adds some functionality to our application, but we'll talk about services more later on.



In .NET 5, we would use the **Configure** method to add different middleware components to the application's request pipeline. But since we don't have that method anymore, we can use the section below the `var app = builder.Build();` part to do that. Again, this is marked with the comment section as well:

```
var builder = WebApplication.CreateBuilder(args);
```

```
// Add services to the container. ①
```

```
| builder.Services.AddControllers();
```

```
| var app = builder.Build();
```

```
// Configure the HTTP request pipeline. ②
```

NOTE: If you still want to create your application using the .NET 5 way, with `Program` and `Startup` classes, you can do that, .NET 6 supports it as well. The easiest way is to create a .NET 5 project, copy the `Startup` and `Program` classes and paste it into the .NET 6 project.

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

1.4 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, 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 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.

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

Now, we mentioned extension methods are great for organizing your code and extending functionalities. Let's go back to our `Program` class and modify it to support CORS and IIS integration now that we've written



extension methods for those functionalities. We are going to remove the first comment and write our code over it:

```
using CompanyEmployees.Extensions;

var builder = WebApplication.CreateBuilder(args);

builder.Services.ConfigureCors();
builder.Services.ConfigureIISIntegration();

builder.Services.AddControllers();

var app = builder.Build();
```

And let's add a few mandatory methods to the second part of the Program class (the one for the request pipeline configuration):

```
var app = builder.Build();

if (app.Environment.IsDevelopment())
    app.UseDeveloperExceptionPage();
else
    app.UseHsts();

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

app.UseCors("CorsPolicy");

app.UseAuthorization();

app.MapControllers();

app.Run();
```

We've added CORS and IIS configuration to the section where we need to configure our services. Furthermore, CORS configuration has been added to the application's pipeline inside the second part of the Program class. 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. Pay attention that we require **Microsoft.AspNetCore.HttpOverrides** using directive to introduce the **ForwardedHeaders** enumeration



- `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.
- `app.UseHsts()` will add middleware for using HSTS, which adds the Strict-Transport-Security header.

1.6 Additional Code in the Program Class

We have to pay attention to the `AddControllers()` method. 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.

Right below the controller registration, we have this line of code:

```
var app = builder.Build();
```

With the `Build` method, we are creating the `app` variable of the type `WebApplication`. This class (`WebApplication`) is very important since it implements multiple interfaces like `IHost` that we can use to start and stop the host, `IApplicationBuilder` that we use to build the middleware pipeline (as you could've seen from our previous custom code), and `IEndpointRouteBuilder` used to add endpoints in our app.

The `UseHttpRedirection` method is used to add the middleware for the redirection from HTTP to HTTPS. Also, we can see the `UseAuthorization` method that adds the authorization middleware to the specified `IApplicationBuilder` to enable authorization capabilities.

Finally, we can see the `MapControllers` method that adds the endpoints from controller actions to the `IEndpointRouteBuilder` and the `Run` method that runs the application and block the calling thread until the host shutdown.



Microsoft advises that the order of adding different middlewares to the application builder is very important, and we are going to talk about that in the middleware section of this book.

1.7 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 **appsettings.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**:



- ◀
 - appsettings.json
 - appsettings.Development.json
 - 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 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**.

Now, let's talk a bit more about the middleware in ASP.NET Core applications.

1.8 ASP.NET Core Middleware

As we already used some middleware code to modify the application's pipeline (CORS, Authorization...), and we are going to use the middleware throughout the rest of the book, we should be more familiar with the ASP.NET Core middleware.

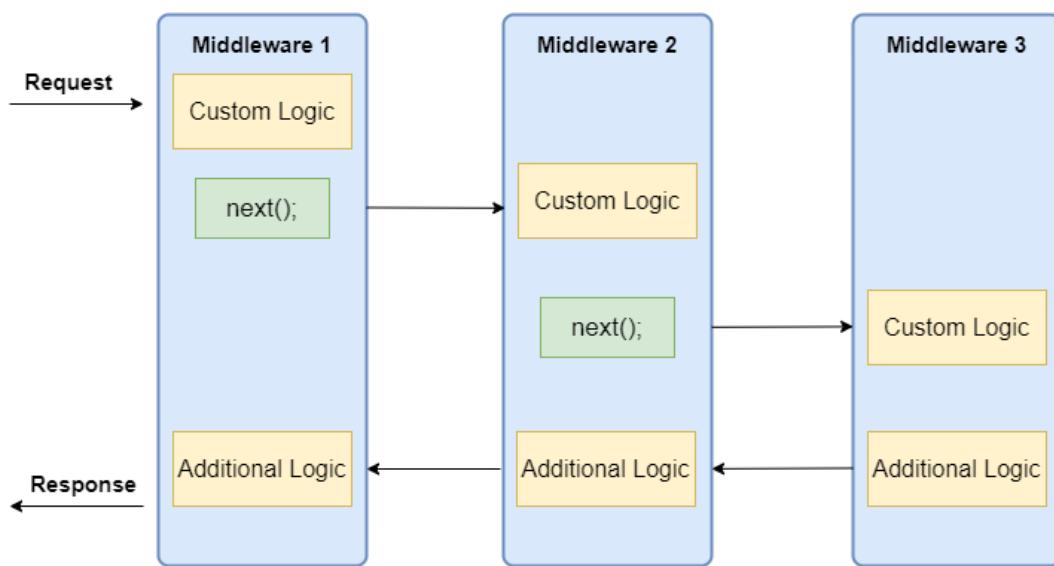
ASP.NET Core middleware is a piece of code integrated inside the application's pipeline that we can use to handle requests and responses. When we talk about the ASP.NET Core middleware, we can think of it as a code section that executes with every request.



Usually, we have more than a single middleware component in our application. Each component can:

- Pass the request to the next middleware component in the pipeline and also
- It can execute some work before and after the next component in the pipeline

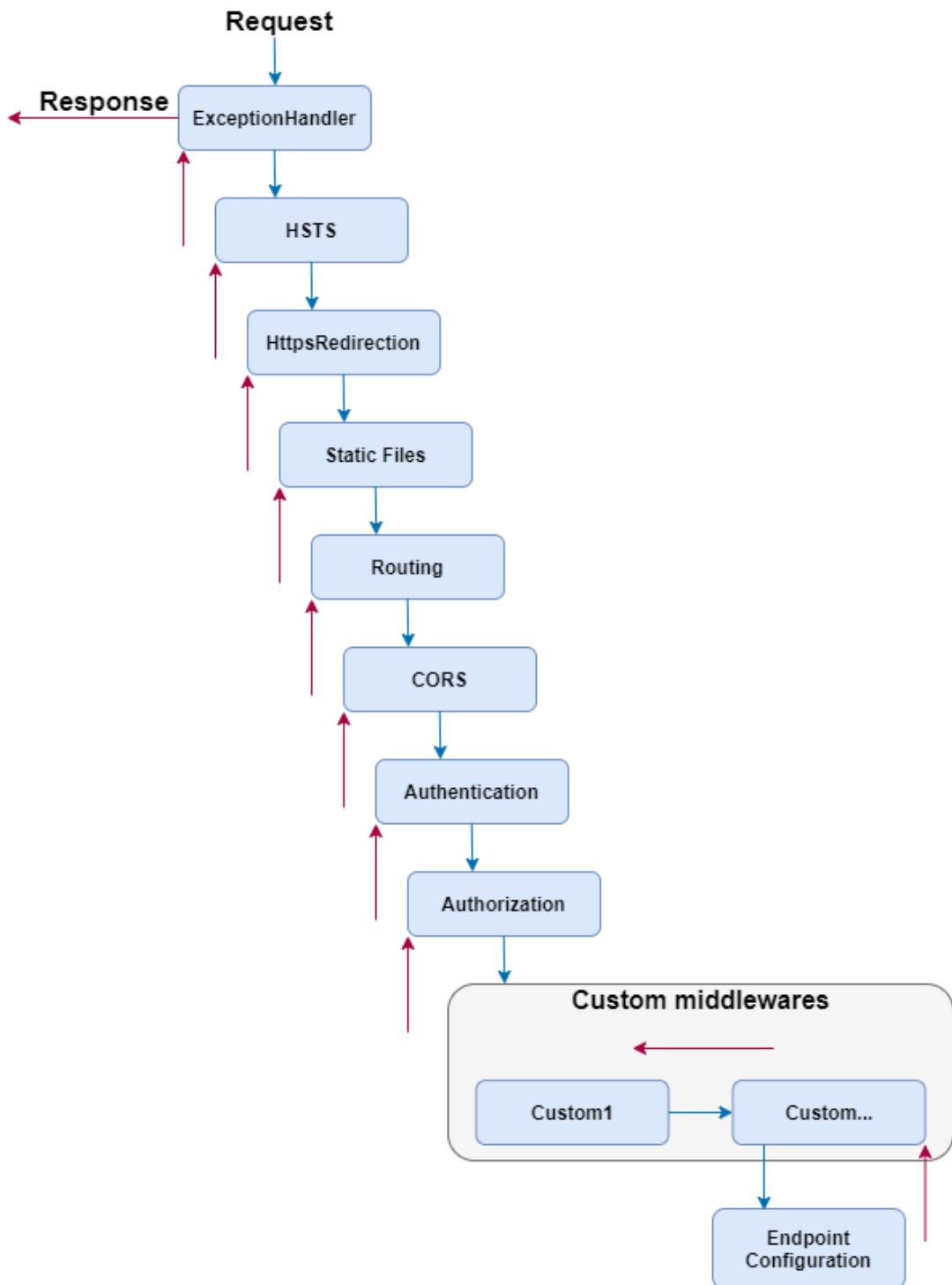
To build a pipeline, we are using request delegates, which handle each HTTP request. To configure request delegates, we use the **Run**, **Map**, and **Use** extension methods. Inside the request pipeline, an application executes each component in the same order they are placed in the code – top to bottom:



Additionally, we can see that each component can execute custom logic before using the **next** delegate to pass the execution to another component. The last middleware component doesn't call the **next** delegate, which means that this component is short-circuiting the pipeline. This is a terminal middleware because it stops further middleware from processing the request. It executes the additional logic and then returns the execution to the previous middleware components.



Before we start with examples, it is quite important to know about the order in which we should register our middleware components. The order is important for the security, performance, and functionality of our applications:





As we can see, we should register the exception handler in the early stage of the pipeline flow so it could catch all the exceptions that can happen in the later stages of the pipeline. When we create a new ASP.NET Core app, many of the middleware components are already registered in the order from the diagram. We have to pay attention when registering additional existing components or the custom ones to fit this recommendation.

For example, when adding CORS to the pipeline, the app in the development environment will work just fine if you don't add it in this order. But we've received several questions from our readers stating that they face the CORS problem once they deploy the app. But once we suggested moving the CORS registration to the required place, the problem disappeared.

Now, we can use some examples to see how we can manipulate the application's pipeline. For this section's purpose, we are going to create a separate application that will be dedicated only to this section of the book. The later sections will continue from the previous project, that we've already created.

1.8.1 Creating a First Middleware Component

Let's start by creating a new ASP.NET Core Web API project, and name it `MiddlewareExample`.

In the `launchSettings.json` file, we are going to add some changes regarding the launch profiles:

```
{  
  "profiles": {  
    "MiddlewareExample": {  
      "commandName": "Project",  
      "dotnetRunMessages": true,  
      "launchBrowser": true,  
      "launchUrl": "weatherforecast",  
      "applicationUrl": "https://localhost:5001;http://localhost:5000",  
      "environmentVariables": {  
        "ASPNETCORE_ENVIRONMENT": "Development"  
      }  
    }  
  }  
}
```



```
}
```

Now, inside the Program class, right below the UseAuthorization part, we are going to use an anonymous method to create a first middleware component:

```
app.UseAuthorization();

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from the middleware component.");
});

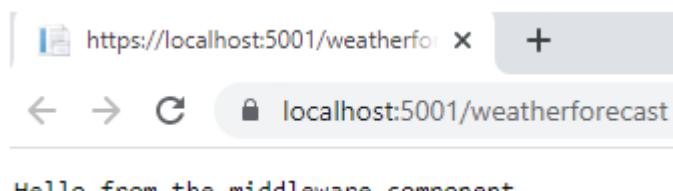
app.MapControllers();
```

We use the **Run** method, which adds a terminal component to the app pipeline. We can see we are not using the **next** delegate because the **Run** method is always terminal and terminates the pipeline. This method accepts a single parameter of the RequestDelegate type. If we inspect this delegate we are going to see that it accepts a single HttpContext parameter:

```
namespace Microsoft.AspNetCore.Http
{
    public delegate Task RequestDelegate(HttpContext context);
}
```

So, we are using that **context** parameter to modify our requests and responses inside the middleware component. In this specific example, we are modifying the response by using the **WriteAsync** method. For this method, we need **Microsoft.AspNetCore.Http** namespace.

Let's start the app, and inspect the result:



There we go. We can see a result from our middleware.



1.8.2 Working with the Use Method

To chain multiple request delegates in our code, we can use the **Use** method. This method accepts a Func delegate as a parameter and returns a Task as a result:

```
public static IApplicationBuilder Use(this IApplicationBuilder app, Func<HttpContext, Func<Task>, Task> middleware);
```

So, this means when we use it, we can make use of two parameters, **context** and **next**:

```
app.UseAuthorization();

app.Use(async (context, next) =>
{
    Console.WriteLine($"Logic before executing the next delegate in the Use method");
    await next.Invoke();
    Console.WriteLine($"Logic after executing the next delegate in the Use method");
});

app.Run(async context =>
{
    Console.WriteLine($"Writing the response to the client in the Run method");
    await context.Response.WriteAsync("Hello from the middleware component.");
});

app.MapControllers();
```

As you can see, we add several logging messages to be sure what the order of executions inside middleware components is. First, we write to a console window, then we invoke the next delegate passing the execution to another component in the pipeline. In the Run method, we write a second message to the console window and write a response to the client. After that, the execution is returned to the Use method and we write the third message (the one below the next delegate invocation) to the console window.

The Run method doesn't accept the next delegate as a parameter, so without it to send the execution to another component, this component short-circuits the request pipeline.

Now, let's start the app and inspect the result, which proves our execution order:



```
Logic before executing the next delegate in the Use method
Writing the response to the client in the Run method
Logic after executing the next delegate in the Use method
```

Maybe you will see two sets of messages but don't worry, that's because the browser sends two sets of requests, one for the `/weatherforecast` and another for the `favicon.ico`. If you, for example, use Postman to test this, you will see only one set of messages.

One more thing to mention. We shouldn't call the `next.Invoke` after we send the response to the client. This can cause exceptions if we try to set the status code or modify the headers of the response.

For example:

```
app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Hello from the middleware component.");
    await next.Invoke();
    Console.WriteLine($"Logic after executing the next delegate in the Use method");
});
app.Run(async context =>
{
    Console.WriteLine($"Writing the response to the client in the Run method");
    context.Response.StatusCode = 200;
    await context.Response.WriteAsync("Hello from the middleware component.");
});
```

Here we write a response to the client and then call `next.Invoke`. Of course, this passes the execution to the next component in the pipeline. There, we try to set the status code of the response and write another one. But let's inspect the result:

```
warn: Microsoft.AspNetCore.Diagnostics.DeveloperExceptionMiddleware[2]
      The response has already started, the error page middleware will not be executed.
fail: Microsoft.AspNetCore.Server.Kestrel[13]
      Connection id "0HMC1NCOT9ODE", Request id "0HMC1NCOT9ODE:00000001": An unhandled exception was thrown by the application.
      System.InvalidOperationException: StatusCode cannot be set because the response has already started.
```

We can see the error message, which is pretty self-explanatory.



1.8.3 Using the Map and MapWhen Methods

To branch the middleware pipeline, we can use both Map and MapWhen methods. The **Map** method is an extension method that accepts a path string as one of the parameters:

```
public static IApplicationBuilder Map(this IApplicationBuilder app, PathString  
pathMatch, Action<IApplicationBuilder> configuration)
```

When we provide the **pathMatch** string, the **Map** method will compare it to the start of the request path. If they match, the app will execute the branch.

So, let's see how we can use this method by modifying the Program class:

```
app.Use(async (context, next) =>  
{  
    Console.WriteLine($"Logic before executing the next delegate in the Use method");  
    await next.Invoke();  
    Console.WriteLine($"Logic after executing the next delegate in the Use method");  
});  
app.Map("/usingmapbranch", builder =>  
{  
    builder.Use(async (context, next) =>  
    {  
        Console.WriteLine("Map branch logic in the Use method before the next  
delegate");  
        await next.Invoke();  
        Console.WriteLine("Map branch logic in the Use method after the next  
delegate");  
    });  
    builder.Run(async context =>  
    {  
        Console.WriteLine($"Map branch response to the client in the Run method");  
        await context.Response.WriteAsync("Hello from the map branch.");  
    });  
});  
app.Run(async context =>  
{  
    Console.WriteLine($"Writing the response to the client in the Run method");  
    await context.Response.WriteAsync("Hello from the middleware component.");  
});
```

By using the Map method, we provide the path match, and then in the delegate, we use our well-known Use and Run methods to execute middleware components.

Now, if we start the app and navigate to **/usingmapbranch**, we are going to see the response in the browser:



← → ⌂ 🔒 localhost:5001/usingmapbranch

Hello from the map branch.

But also, if we inspect console logs, we are going to see our new messages:

```
Logic before executing the next delegate in the Use method
Map branch logic in the Use method before the next delegate
Map branch response to the client in the Run method
Map branch logic in the Use method after the next delegate
Logic after executing the next delegate in the Use method
```

Here, we can see the messages from the **Use** method before the branch, and the messages from the **Use** and **Run** methods inside the **Map** branch. We are not seeing any message from the **Run** method outside the branch. It is important to know that any middleware component that we add after the Map method in the pipeline won't be executed. This is true even if we don't use the Run middleware inside the branch.

1.8.4 Using MapWhen Method

If we inspect the MapWhen method, we are going to see that it accepts two parameters:

```
public static IApplicationBuilder MapWhen(this IApplicationBuilder app,
Func<HttpContext, bool> predicate, Action<IApplicationBuilder> configuration)
```

This method uses the result of the given predicate to branch the request pipeline.

So, let's see it in action:

```
app.Map("/usingmapbranch", builder =>
{
    ...
});
app.MapWhen(context => context.Request.Query.ContainsKey("testquerystring"), builder =>
{
    builder.Run(async context =>
    {
        await context.Response.WriteAsync("Hello from the MapWhen branch."));
```



```
    });
});
app.Run(async context =>
{
    ...
});
```

Here, if our request contains the provided query string, we execute the **Run** method by writing the response to the client. So, as we said, based on the predicate's result the **MapWhen** method branch the request pipeline.

Now, we can start the app and navigate to **https://localhost:5001?testquerystring=test**:

Hello from the MapWhen branch.

And there we go. We can see our expected message. Of course, we can chain multiple middleware components inside this method as well.

So, now we have a good understanding of using middleware and its order of invocation in the ASP.NET Core application. This knowledge is going to be very useful to us once we start working on a custom error handling middleware (a few sections later).

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. We can close this app, and continue with the CompanyEmployees app.



2 CONFIGURING A LOGGING SERVICE

Why do 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 implementation of the logging mechanism, but in all our projects we prefer to create our custom logger service with the external logger library NLog.

We are going to do that because having an abstraction will allow us to have any logger behind our interface. This means that we can start with NLog, and at some point, we can switch to any other logger and our interface will still work because of our abstraction.

2.1 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 (C#) project template:



Add a new project

Recent project templates
A list of your recently accessed templates will be displayed here.

All languages All platforms All project types

Class Library
A project for creating a class library that targets .NET Standard or .NET Core
C# Android Linux macOS Windows Library

VB Class Library
A project for creating a class library that targets .NET Standard or .NET Core
Visual Basic Android Linux macOS Windows Library

Finally, name it **Contracts**, and choose the .NET 6.0 as a version. 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 **Add Project Reference** option. Under Projects, click Solution and check the **Contracts** project.

Now, in the main project right click on **Dependencies** and then click on **Add Project 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.

2.2 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.



Ultimate ASP.NET Core Web API

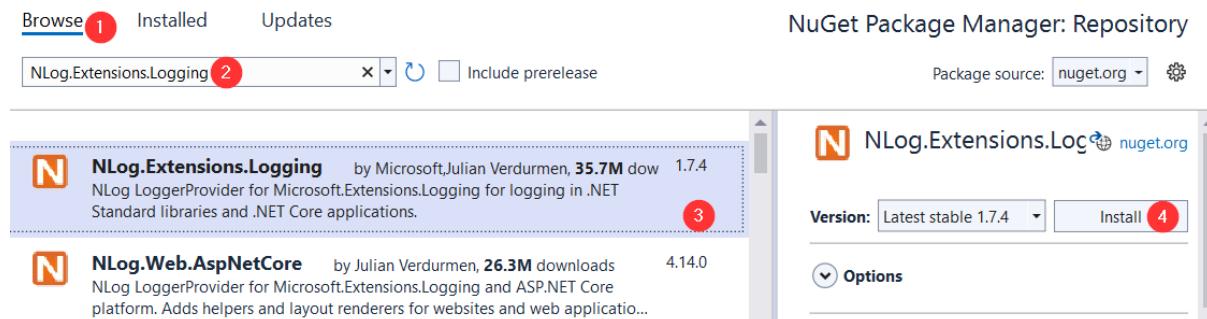
So, let's do that first by right-clicking on the **Contracts** project, choosing the **Add -> New Item** menu, and then selecting the Interface option where we have to specify the name **ILoggerManager** and click the **Add** button. After the file creation, we can add the code:

```
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 1.7.4
```

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



2.3 Implementing the Interface and Nlog.Config File

In the **LoggerService** project, we are going to create a new class: **LoggerManager**. We can do that by repeating the same steps for the interface creation just choosing the class option instead of an interface. 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 **Program** class in the section related to the service configuration.

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**. So, let's right-click on the main project, choose Add -> New Item, and then search for the Text File. Select the Text File, and add the name **nlog.config**.

```
<?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=".\\internal_logs\\internallog.txt">
```



```
<targets>
  <target name="logfile" xsi:type="File"
    fileName=".\\logs\\${shortdate}_logfile.txt"
    layout="${longdate} ${level:uppercase=true} ${message}"/>
</targets>

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

You can find the internal logs at the project root, and the logs folder in the bin\debug folder of the main project once we start the app. Once the application is published both folders will be created at the root of the output folder which is what we want.

NOTE: : If you want to have more control over the log output, we suggest renaming the current file to nlog.development.config and creating another configuration file called nlog.production.config. Then you can do something like this in the code: env.ConfigureNLog(\$"nlog.{env.EnvironmentName}.config"); to get the different configuration files for different environments. From our experience production path is what matters, so this might be a bit redundant.

2.4 Configuring Logger Service for Logging Messages

Setting up the configuration for a logger service is quite easy. First, we need to update the **Program** class and include the path to the configuration file for the NLog configuration:

```
using NLog;

var builder = WebApplication.CreateBuilder(args);

LogManager.LoadConfiguration(string.Concat(Directory.GetCurrentDirectory(),
"/nlog.config"));

builder.Services.ConfigureCors();
builder.Services.ConfigureIISIntegration();
```

We are using NLog's **LogManager** static class with the **LoadConfiguration** method to provide a path to the configuration file.



NOTE: : If VisualStudio asks you to install the NLog package in the main project, don't do it. Just remove the LoggerService reference from the main project and add it again. We have already installed the required package in the LoggerService project and the main project should be able to reference it as well.

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.AddSingleton<ILoggerManager, LoggerManager>();
```

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

```
builder.Services.ConfigureCors();
builder.Services.ConfigureIISIntegration();
builder.Services.ConfigureLoggerService();

builder.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.

2.5 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.LogWarning("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>.

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:

```
2021-09-29 16:31:44.0320 INFO Here is info message from our values controller.
2021-09-29 16:31:44.1068 DEBUG Here is debug message from our values controller.
2021-09-29 16:31:44.1068 WARN Here is warn message from our values controller.
2021-09-29 16:31:44.1068 ERROR Here is an error message from our values 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.



3 ONION ARCHITECTURE IMPLEMENTATION

In this chapter, we are going to talk about the Onion architecture, its layers, and the advantages of using it. We will learn how to create different layers in our application to separate the different application parts and improve the application's maintainability and testability.

That said, 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.

Additionally, we are going to create a Service layer to extract all the business logic from our controllers, thus making the presentation layer and the controllers clean and easy to maintain.

So, let's start with the Onion architecture explanation.



3.1 About Onion Architecture

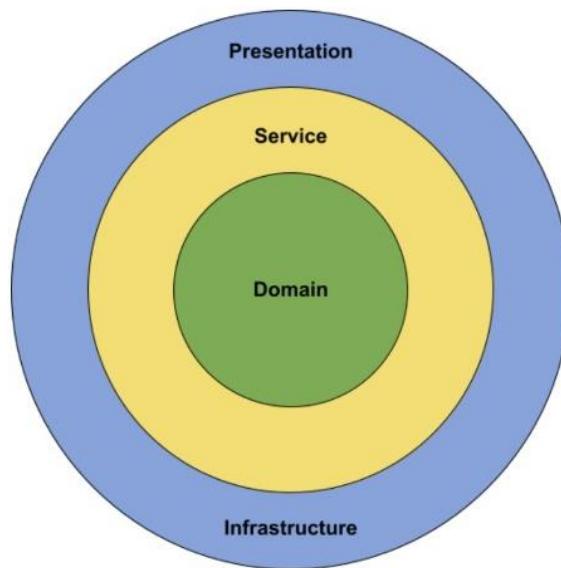
The Onion architecture is a form of layered architecture and we can visualize these layers as concentric circles. Hence the name Onion architecture. The Onion architecture was first introduced by Jeffrey Palermo, to overcome the issues of the traditional N-layered architecture approach.

There are multiple ways that we can split the onion, but we are going to choose the following approach where we are going to split the architecture into 4 layers:

- Domain Layer
- Service Layer
- Infrastructure Layer
- Presentation Layer

Conceptually, we can consider that the Infrastructure and Presentation layers are on the same level of the hierarchy.

Now, let us go ahead and look at each layer with more detail to see why we are introducing it and what we are going to create inside of that layer:



We can see all the different layers that we are going to build in our project.



3.1.1 Advantages of the Onion Architecture

Let us take a look at what are the advantages of Onion architecture, and why we would want to implement it in our projects.

All of the layers interact with each other strictly through the interfaces defined in the layers below. The flow of dependencies is towards the core of the Onion. We will explain why this is important in the next section.

Using dependency inversion throughout the project, depending on abstractions (interfaces) and not the implementations, allows us to switch out the implementation at runtime transparently. We are depending on abstractions at compile-time, which gives us strict contracts to work with, and we are being provided with the implementation at runtime.

Testability is very high with the Onion architecture because everything depends on abstractions. The abstractions can be easily mocked with a mocking library such as Moq. We can write business logic without concern about any of the implementation details. If we need anything from an external system or service, we can just create an interface for it and consume it. We do not have to worry about how it will be implemented.

The higher layers of the Onion will take care of implementing that interface transparently.

3.1.2 Flow of Dependencies

The main idea behind the Onion architecture is the flow of dependencies, or rather how the layers interact with each other. The deeper the layer resides inside the Onion, the fewer dependencies it has.

The Domain layer does not have any direct dependencies on the outside layers. It is isolated, in a way, from the outside world. The outer layers are all allowed to reference the layers that are directly below them in the hierarchy.

We can conclude that all the dependencies in the Onion architecture flow inwards. But we should ask ourselves, why is this important?



The flow of dependencies dictates what a certain layer in the Onion architecture can do. Because it depends on the layers below it in the hierarchy, it can only call the methods that are exposed by the lower layers.

We can use lower layers of the Onion architecture to define contracts or interfaces. The outer layers of the architecture implement these interfaces. This means that in the Domain layer, we are not concerning ourselves with infrastructure details such as the database or external services.

Using this approach, we can encapsulate all of the rich business logic in the Domain and Service layers without ever having to know any implementation details. In the Service layer, we are going to depend only on the interfaces that are defined by the layer below, which is the Domain layer.

So, after all the theory, we can continue with our project implementation. Let's start with the models and the **Entities** project.

3.2 Creating Models

Using the example from the second chapter of this book, we are going to extract a new Class Library project named **Entities**.

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 the Address is 60 characters")]
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: the 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



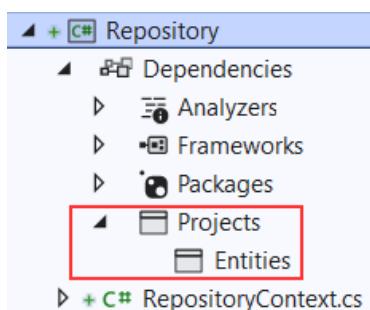
for validation purposes. The first one declares the property as mandatory and the second one defines its maximum length.

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.

3.3 Context Class and the Database Connection

Before we start with the context class creation, we have to create another .NET Class Library and name it **Repository**. We are going to use this project for the database context and repository implementation.

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 **Repository** project. Also, we are going to reference the **Entities** project from the **Repository** project:



Then, let's navigate to the root of the **Repository** 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, in the main project, 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 **appsettings.json** file, and soon you will see why.

But first, we have to add the Repository project's reference into the main project.

Then, let's create a new **ContextFactory** folder **in the main project** and inside it a new **RepositoryContextFactory** class. Since our **RepositoryContext** class is in a **Repository** project and not in the main one, this class will help our application create a derived DbContext instance during the design time which will help us with our migrations:

```
public class RepositoryContextFactory : IDesignTimeDbContextFactory<RepositoryContext>
{
    public RepositoryContext CreateDbContext(string[] args)
    {
        var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json")
            .Build();

        var builder = new DbContextOptionsBuilder<RepositoryContext>()
            .UseSqlServer(configuration.GetConnectionString("sqlConnection"));

        return new RepositoryContext(builder.Options);
    }
}
```



We are using the **IDesignTimeDbContextFactory<out TContext>** interface that allows design-time services to discover implementations of this interface. Of course, the **TContext** parameter is our **RepositoryContext** class.

For this, we need to add two using directives:

```
using Microsoft.EntityFrameworkCore.Design;
using Repository;
```

Then, we have to implement this interface with the **CreateDbContext** method. Inside it, we create the **configuration** variable of the **IConfigurationRoot** type and specify the appsettings file, we want to use. With its help, 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 in the main project and add one more using directive:

```
using Microsoft.EntityFrameworkCore;
```

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);
```

Finally, in the **CreateDbContext** method, we return a new instance of our **RepositoryContext** class with provided options.



3.4 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 **CreateDbContext** method first:

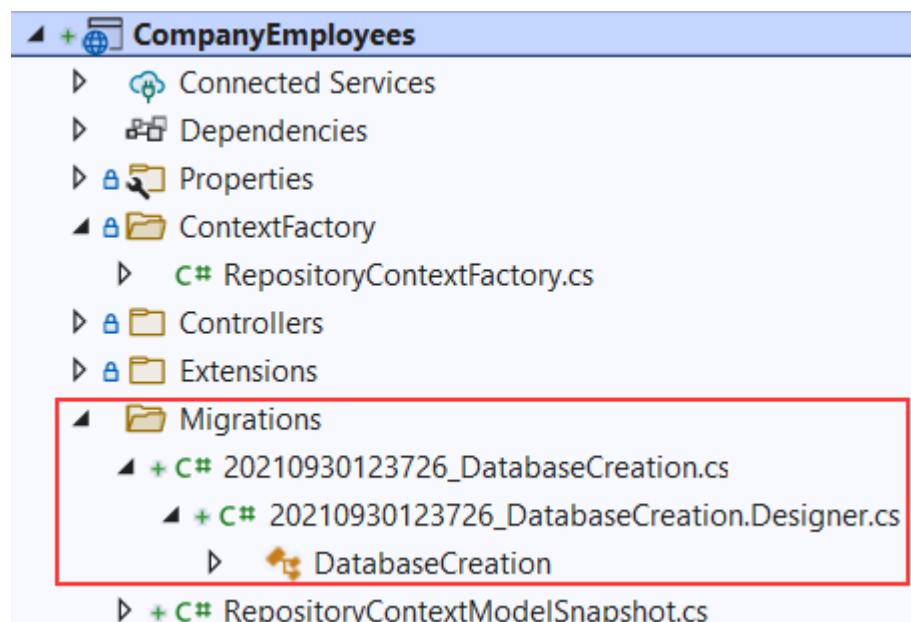
```
var builder = new DbContextOptionsBuilder<RepositoryContext>()
    .UseSqlServer(configuration.GetConnectionString("sqlConnection"),
        b => b.MigrationsAssembly("CompanyEmployees"));
```

We have to make this change because migration assembly is not in our main project, but in the **Repository** project. So, we've just changed 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:

The screenshot shows a database structure for a 'CompanyEmployee' schema. It includes a 'Tables' folder containing 'System Tables', 'FileTables', 'dbo._EFMigrationsHistory', 'dbo.Companies', and 'dbo.Employees'. The 'dbo.Companies' and 'dbo.Employees' tables are specifically highlighted with red boxes. Within each table, there are 'Columns' listed: 'CompanyId' (PK, uniqueidentifier, not null), 'Name' (nvarchar(60), not null), 'Address' (nvarchar(60), not null), 'Country' (nvarchar(max), null) for 'Companies'; and 'EmployeeId' (PK, uniqueidentifier, not null), 'Name' (nvarchar(30), not null), 'Age' (int, not null), 'Position' (nvarchar(20), not null) for 'Employees'. The 'CompanyId' column in both tables is also highlighted with a red box.

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 in the **Repository** project called **Configuration** 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.

3.5 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 reference **Contracts** inside the **Repository** project. Also, in the **Repository** project, we are going to create an abstract class **RepositoryBase** — which is going to implement the **IRepositoryBase** interface:

```
public abstract 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 work 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.



3.6 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 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>, IEmployerRepository
```



```
{  
    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.

3.7 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 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 them inside the dependency injection container. After that, we can inject it inside our services 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 sealed class RepositoryManager : IRepositoryManager  
{  
    private readonly RepositoryContext _repositoryContext;  
    private readonly Lazy<ICompanyRepository> _companyRepository;  
    private readonly Lazy<IEmployeeRepository> _employeeRepository;  
  
    public RepositoryManager(RepositoryContext repositoryContext)  
    {  
        _repositoryContext = repositoryContext;  
        _companyRepository = new Lazy<ICompanyRepository>(() => new  
CompanyRepository(repositoryContext));  
        _employeeRepository = new Lazy<IEmployeeRepository>(() => new  
EmployeeRepository(repositoryContext));  
    }  
  
    public ICompanyRepository Company => _companyRepository.Value;  
    public IEmployeeRepository Employee => _employeeRepository.Value;  
  
    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();
```

The interesting part with the **RepositoryManager** implementation is that we are leveraging the power of the **Lazy** class to ensure the lazy initialization of our repositories. This means that our repository instances are only going to be created when we access them for the first time, and not before that.



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

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

And in the **Program** class above the **AddController()** method, we have to add this code:

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

Excellent.

As soon as we add some methods to the specific repository classes, and add our service layer, we are going to be able to test this logic.

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

Now, we can continue towards creating a service layer in our application.

3.8 Adding a Service Layer

The Service layer sits right above the Domain layer (the Contracts project is the part of the Domain layer), which means that it has a reference to the Domain layer. The Service layer will be split into two projects, `Service.Contracts` and `Service`.

So, let's start with the **Service.Contracts** project creation (.NET Core Class Library) where we will hold the definitions for the service interfaces that are going to encapsulate the main business logic. In the next section, we are going to create a presentation layer and then, we will see the full use of this project.

Once the project is created, we are going to add three interfaces inside it.

ICompanyService:

```
public interface ICompanyService
```



```
{  
}
```

IEmployeeService:

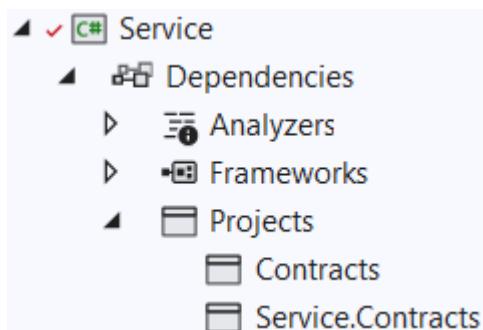
```
public interface IEmployeeService  
{  
}
```

And IServiceManager:

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

As you can see, we are following the same pattern as with the repository contracts implementation.

Now, we can create another project, name it **Service**, and reference the **Service.Contracts** and **Contracts** projects inside it:



After that, we are going to create classes that will inherit from the interfaces that reside in the **Service.Contracts** project.

So, let's start with the **CompanyService** class:

```
using Contracts;  
using Service.Contracts;  
  
namespace Service  
{  
    internal sealed class CompanyService : ICompanyService  
    {  
        private readonly IRepositoryManager _repository;  
        private readonly ILoggerManager _logger;  
  
        public CompanyService(IRepositoryManager repository, ILoggerManager  
logger)
```



```
        {
            _repository = repository;
            _logger = logger;
        }
    }
```

As you can see, our class inherits from the **ICompanyService** interface, and we are injecting the **IRepositoryManager** and **ILoggerManager** interfaces. We are going to use **IRepositoryManager** to access the repository methods from each user repository class (CompanyRepository or EmployeeRepository), and **ILoggerManager** to access the logging methods we've created in the second section of this book.

To continue, let's create a new **EmployeeService** class:

```
using Contracts;
using Service.Contracts;

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

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

Finally, we are going to create the **ServiceManager** class:

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

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

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



```
}
```

Here, as we did with the `RepositoryManager` class, we are utilizing the `Lazy` class to ensure the lazy initialization of our services.

Now, with all these in place, we have to add the reference from the **Service** project inside the main project. Since **Service** is already referencing **Service.Contracts**, our main project will have the same reference as well.

Now, we have to modify the **ServiceExtensions** class:

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

And we have to add using directives:

```
using Service;
using Service.Contracts;
```

Then, all we have to do is to modify the **Program** class to call this extension method:

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

3.9 Registering RepositoryContext at a Runtime

With the **RepositoryContextFactory** class, which implements the **IDesignTimeDbContextFactory** interface, we have registered our **RepositoryContext** class at design time. This helps us find the **RepositoryContext** class in another project while executing migrations.

But, as you could see, we have the **RepositoryManager** service registration, which happens at runtime, and during that registration, we must have **RepositoryContext** registered as well in the runtime, so we could inject it into other services (like `RepositoryManager` service). This might be a bit confusing, so let's see what that means for us.

Let's modify the **ServiceExtensions** class:



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

We are not specifying the **MigrationAssembly** inside the **UseSqlServer** method. We don't need it in this case.

As the final step, we have to call this method in the **Program** class:

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

With this, **we have completed our implementation**, and our service layer is ready to be used in our next chapter where we are going to learn about handling GET requests in ASP.NET Core Web API.

One additional thing. From .NET 6 RC2, there is a shortcut method **AddSqlServer**, which can be used like this:

```
public static void ConfigureDbContext(this IServiceCollection services,
IConfiguration configuration) =>
    services.AddSqlServer<RepositoryContext>((configuration.GetConnectionString("sqlConnection")));
```

This method replaces both **AddDbContext** and **UseSqlServer** methods and allows an easier configuration. But it doesn't provide all of the features the **AddDbContext** method provides. So for more advanced options, it is recommended to use **AddDbContext**. We will use it throughout the rest of the project.



4 HANDLING GET REQUESTS

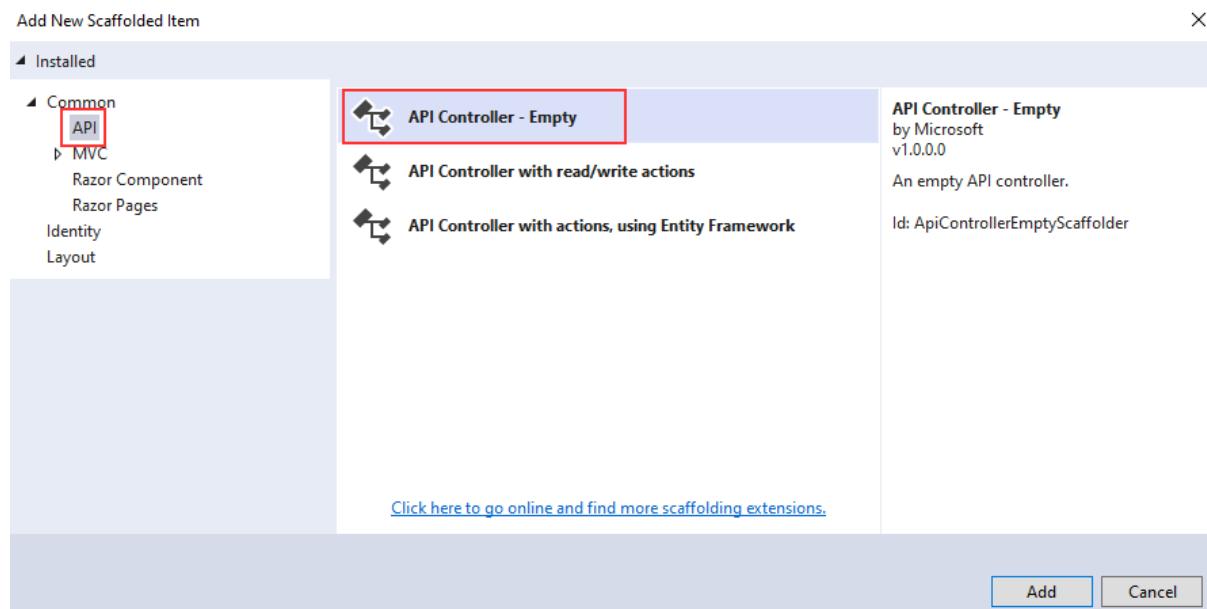
We're all set to add some business logic to our application. But before we do that, let's talk a bit about controller classes and routing because they play an important part while working with HTTP requests.

4.1 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.

If you want to create the controller in the main project, you would right-click on the Controllers folder and then Add=>Controller. Then from the menu, you would choose API Controller Class and give it a name:



But, that's not the thing we are going to do. We don't want to create our controllers in the main project.

What we are going to do instead is create a presentation layer in our application.



The purpose of the presentation layer is to provide the entry point to our system so that consumers can interact with the data. We can implement this layer in many ways, for example creating a REST API, gRPC, etc.

However, we are going to do something different from what you are normally used to when creating Web APIs. By convention, controllers are defined in the **Controllers** folder inside the main project.

Why is this a problem?

Because ASP.NET Core uses Dependency Injection everywhere, we need to have a reference to all of the projects in the solution from the main project. This allows us to configure our services inside the **Program** class.

While this is exactly what we want to do, it introduces a big design flaw. What's preventing our controllers from injecting anything they want inside the constructor?

So how can we impose some more strict rules about what controllers can do?

Do you remember how we split the **Service** layer into the Service.Contracts and Service projects? That was one piece of the puzzle.

Another part of the puzzle is the creation of a new class library project, **CompanyEmployees.Presentation**.

Inside that new project, we are going to install **Microsoft.AspNetCore.Mvc.Core** package so it has access to the **ControllerBase** class for our future controllers. Additionally, let's create a single class inside the **Presentation** project:

```
public static class AssemblyReference
{}
```

It's an empty static class that we are going to use for the assembly reference inside the main project, you will see that in a minute.



The one more thing, we have to do is to reference the **Service.Contracts** project inside the **Presentation** project.

Now, we are going to delete the **Controllers** folder and the **WeatherForecast.cs** file from the main project because we are not going to need them anymore.

Next, we have to reference the Presentation project inside the main one. As you can see, our presentation layer depends only on the service contracts, thus imposing more strict rules on our controllers.

Then, we have to modify the **Program.cs** file:

```
builder.Services.AddControllers()
    .AddApplicationPart(typeof(CompanyEmployees.Presentation.AssemblyReference).Assembly);
```

Without this code, our API wouldn't work, and wouldn't know where to route incoming requests. But now, our app will find all of the controllers inside of the **Presentation** project and configure them with the framework. They are going to be treated the same as if they were defined conventionally.

But, we don't have our controllers yet. So, let's navigate to the **Presentation** project, create a new folder named **Controllers**, and then a new class named **CompaniesController**. Since this is a class library project, we don't have an option to create a controller as we had in the main project. Therefore, we have to create a regular class and then modify it:

```
using Microsoft.AspNetCore.Mvc;

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



We've created this controller in the same way the main project would.

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 **Program** class:

```
app.MapControllerRoute(  
    name: "default",  
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

1

2

3

Our Web API project doesn't configure routes this way, but if you create an MVC project this will be the default route configuration. Of course, if you are using this type of route configuration, you have to use the



`app.UseRouting` method to add the routing middleware in the application's pipeline.

If you inspect the `Program` class in our main project, you won't find the `UseRouting` method because the routes are configured with the `app.MapControllers` method, which adds endpoints for controller actions without specifying any routes.

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.



4.2 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.

4.3 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.

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

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

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

As you can see, we are calling the **FindAll** method from the **RepositoryBase** class, ordering the result with the **OrderBy** method, and then executing the query with the **ToList** method.

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

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

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

Since the **Company** model resides in the **Entities** project, we have to add the **Entities** reference to the **Service.Contracts** project. At least, we have for now.



Let's be clear right away before we proceed. **Getting all the entities from the database is a bad idea.** We're going to start with the simplest method and change it later on.

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

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

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

    public IEnumerable<Company> GetAllCompanies(bool trackChanges)
    {
        try
        {
            var companies =
                _repository.Company.GetAllCompanies(trackChanges);

            return companies;
        }
        catch (Exception ex)
        {
            _logger.LogError($"Something went wrong in the
{nameof(GetAllCompanies)} service method {ex}");
            throw;
        }
    }
}
```

We are using our repository manager to call the **GetAllCompanies** method from the **CompanyRepository** class and return all the companies from the database.

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 IServiceManager _service;

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

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

            return Ok(companies);
        }
        catch
        {
            return StatusCode(500, "Internal server error");
        }
    }
}
```

Let's explain this code a bit.

First of all, we inject the **IServiceManager** interface 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 an injected service to call the service method that gets the data from the repository class.

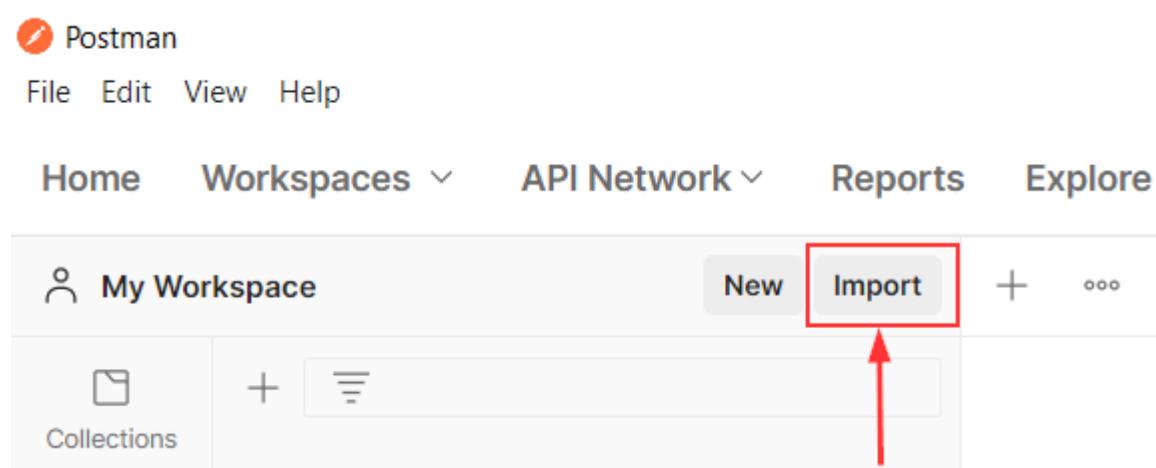
The **IActionResult** interface supports using a variety of methods, which return not only the result but also the status codes. In this situation, the **OK** method returns all the companies and also the status code 200 — which stands for **OK**. If an exception occurs, we are going to return the internal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



4.4 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 **Bonus 2 - CompanyEmployeesRequests.postman_collection.json**, 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:

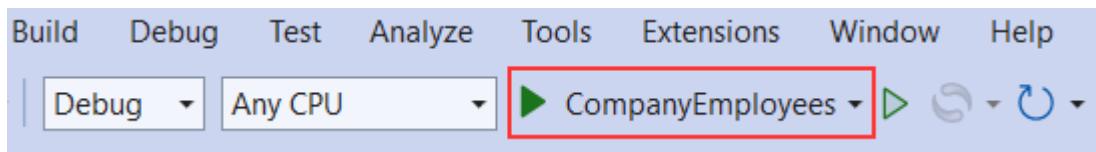


NOTE: Please note that some GUID values will be different for your project, so you have to change them according to those values.

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

```
D:\Projects\codemaze-books\Source Code\V2\04-Handling GET Requests\CompanyEmployees\CompanyEmployees\bin\Debug\net... - □ ×  
info: Microsoft.Hosting.Lifetime[14]  
      Now listening on: https://localhost:5001  
info: Microsoft.Hosting.Lifetime[14]  
      Now listening on: http://localhost:5000  
info: Microsoft.Hosting.Lifetime[0]  
      Application started. Press Ctrl+C to shut down.  
info: Microsoft.Hosting.Lifetime[0]  
      Hosting environment: Development  
info: Microsoft.Hosting.Lifetime[0]  
      Content root path: D:\Projects\codemaze-books\Source Code\V2\04-Handling GET Requests\CompanyEmployees\CompanyEmployees
```

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:

<https://localhost:5001/api/companies>

The screenshot shows the Postman interface with a successful GET request to <https://localhost:5001/api/companies>. The response status is 200 OK, 71 ms, 445 B. The response body is a JSON array containing two company objects:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
1
{
  "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
  "name": "Admin_Solutions Ltd",
  "address": "312 Forest Avenue, BF 923",
  "country": "USA",
  "employees": null
},
{
  "id": "c9d4c053-49b6-410c-bc78-2d54a9991870",
  "name": "IT_Solutions Ltd",
  "address": "583 Wall Dr. Gwynn Oak, MD 21207",
  "country": "USA",
  "employees": null
}
```

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.

4.5 DTO Classes vs. Entity Model Classes

A 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 a new project named Shared and then a new folder **DataTransferObjects** with the **CompanyDto** record inside:

```
namespace Shared.DataTransferObjects
{
    public record CompanyDto(Guid Id, string Name, string FullAddress);
}
```

Instead of a regular class, we are using a record for DTO. This specific record type is known as a Positional record.

A *Record* type provides us an easier way to create an immutable reference type in .NET. This means that the *Record*'s instance property values cannot change after its initialization. The data are passed by value and the equality between two *Records* is verified by comparing the value of their properties.

Records can be a valid alternative to classes when we have to send or receive data. The very purpose of a DTO is to transfer data from one part of the code to another, and immutability in many cases is useful. We use



them to return data from a Web API or to represent events in our application.

This is the exact reason why we are using records for our DTOs.

In our DTO, 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 record, because we are going to use this record only to return a response to the client. Therefore, validation attributes are not required.

So, the first thing we have to do is to add the reference from the **Shared** project to the **Service.Contracts** project, and remove the **Entities** reference. At this moment the **Service.Contracts** project is only referencing the **Shared** project.

Then, we have to modify the **ICompanyService** interface:

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

And the **CompanyService** class:

```
public IEnumerable<CompanyDto> GetAllCompanies(bool trackChanges)
{
    try
    {
        var companies = _repository.Company.GetAllCompanies(trackChanges);

        var companiesDto = companies.Select(c =>
            new CompanyDto(c.Id, c.Name ?? "", string.Join(' ', c.Address, c.Country)))
            .ToList();

        return companiesDto;
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong in the {nameof(GetAllCompanies)} service method {ex}");
        throw;
    }
}
```



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

The screenshot shows a Postman request for `https://localhost:5001/api/companies`. The response is a JSON object with two items, each representing a company with properties `id`, `name`, and `fullAddress`.

```
1
2   [
3     {
4       "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
5       "name": "Admin_Solutions Ltd",
6       "fullAddress": "312 Forest Avenue, BF 923 USA"
7     },
8     {
9       "id": "c9d4c053-49b6-410c-bc78-2d54a9991870",
10      "name": "IT_Solutions Ltd",
11      "fullAddress": "583 Wall Dr. Gwynn Oak, MD 21207 USA"
12    }
]
```

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 a 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.

4.6 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, choose the Service project as a default project from the drop-down list, and run the following command:

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



After installation, we are going to register this library in the **Program** class:

```
builder.Services.AddAutoMapper(typeof(Program));
```

As soon as our library is registered, we are going to create a profile class, also in the main project, 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 record, 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 have to modify the **ServiceManager** class to enable DI in our service classes:

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

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

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



Of course, now we have two errors regarding our service constructors. So we need to fix that in both **CompanyService** and **EmployeeService** classes:

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

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

    ...
}
```

We should do the same in the **EmployeeService** class:

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

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

Finally, we can modify the **GetAllCompanies** method in the **CompanyService** class:

```
public IEnumerable<CompanyDto> GetAllCompanies(bool trackChanges)
{
    try
    {
        var companies = _repository.Company.GetAllCompanies(trackChanges);

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

        return companiesDto;
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong in the {nameof(GetAllCompanies)} service method {ex}");
    }
}
```

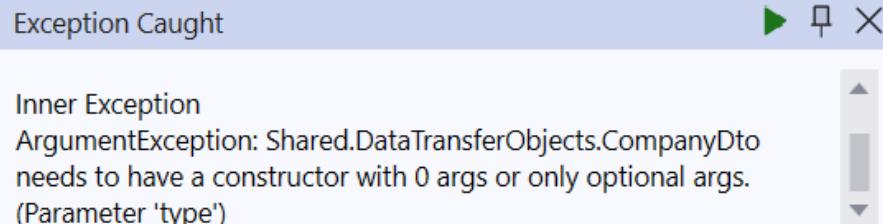


```
        throw;  
    }  
}
```

We are using the Map method and specify the destination and then the source object.

Excellent.

Now if we start our app and send the same request from Postman, we are going to get an error message:



This happens because AutoMapper is not able to find the specific **FullAddress** property as we specified in the **MappingProfile** class. We are intentionally showing this error for you to know what to do if it happens to you in your projects.

So to solve this, all we have to do is to modify the MappingProfile class:

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

This time, we are not using the **ForMember** method but the **ForCtorParam** method to specify the name of the parameter in the constructor that AutoMapper needs to map to.

Now, let's use Postman again to send the request to test our app:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies>

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

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

Body Save Response

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
4     "name": "Admin_Solutions Ltd",
5     "fullAddress": "312 Forest Avenue, BF 923 USA"
6   },
7   {
8     "id": "c9d4c053-49b6-410c-bc78-2d54a9991870",
9     "name": "IT_Solutions Ltd",
10    "fullAddress": "583 Wall Dr. Gwynn Oak, MD 21207 USA"
11  }
12 ]
```

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



5 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 afterward.

Even though there is nothing wrong with the try-catch blocks in our Actions and methods 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 for global error handling to demonstrate the benefits of this approach. Since we already talked about the middleware in ASP.NET Core (in section 1.8), this section should be easier to understand.

5.1 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:

```
using System.Text.Json;

namespace Entities.ErrorModel
{
    public class ErrorDetails
    {
        public int StatusCode { get; set; }
        public string? Message { get; set; }

        public override string ToString() => JsonSerializer.Serialize(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 WebApplication 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 create an extension method, on top of the **WebApplication** type, and we call the **UseExceptionHandler** method. That method adds a middleware to the pipeline that will catch exceptions, log them, and re-execute the request in an alternate pipeline.

Inside the **UseExceptionHandler** method, we use the **appError** variable of the **IApplicationBuilder** type. With that variable, we call the **Run** method, which adds a terminal middleware delegate to the application's pipeline. This is something we already know from section 1.8.

Then, we populate the status code and the content type of our response, log the error message and finally return the response with the custom-



created object. Later on, we are going to modify this middleware even more to support our business logic in a service layer.

Of course, there are several namespaces we should add to make this work:

```
using Contracts;
using Entities.ErrorModel;
using Microsoft.AspNetCore.Diagnostics;
using System.Net;
```

5.2 Program Class Modification

To be able to use this extension method, let's modify the **Program** class:

```
var app = builder.Build();

var logger = app.Services.GetRequiredService<ILoggerManager>();
app.ConfigureExceptionHandler(logger);

if (app.Environment.IsProduction())
    app.UseHsts();

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

app.UseCors("CorsPolicy");

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Here, we first extract the **ILoggerManager** service inside the **logger** variable. Then, we just call the **ConfigureExceptionHandler** method and pass that logger service. It is important to know that we have to extract the **ILoggerManager** service after the **var app = builder.Build()** code line because the **Build** method builds the **WebApplication** and registers all the services added with IOC.



Additionally, we remove the call to the **UseDeveloperExceptionPage** method in the development environment since we don't need it now and it also interferes with our error handler middleware.

Finally, let's remove the **try-catch** block from the **GetAllCompanies** service method:

```
public IEnumerable<CompanyDto> GetAllCompanies(bool trackChanges)
{
    var companies = _repository.Company.GetAllCompanies(trackChanges);

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

    return companiesDto;
}
```

And from our **GetCompanies** action:

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

    return Ok(companies);
}
```

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

5.3 Testing the Result

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

```
[HttpGet]
public IActionResult GetCompanies()
{
    throw new Exception("Exception");
    var companies = _service.CompanyService.GetAllCompanies(trackChanges: false);

    return Ok(companies);
}
```



NOTE: Once you send the request, Visual Studio will stop the execution inside the GetCompanies action on the line where we throw an exception. This is normal behavior and all you have to do is to click the continue button to finish the request flow. Additionally, you can start your app with CTRL+F5, which will prevent Visual Studio from stopping the execution. Also, if you want to start your app with F5 but still to avoid VS execution stoppages, you can open the Tools->Options->Debugging->General option and uncheck the Enable Just My Code checkbox.

And send a request from Postman:

The screenshot shows a Postman request to `https://localhost:5001/api/companies` using the GET method. The 'Headers' tab is selected, showing 6 headers. The 'Body' tab is selected, showing a JSON response with a red box highlighting the error message. The response body is:

```
1 "StatusCode": 500,
2 "Message": "Internal Server Error."
3
4
```

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



6 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.

6.1 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();
```

Then, we have to modify the **ICompanyService** interface:

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

And of course, we have to implement this interface in the **CompanyService** class:

```
public CompanyDto GetCompany(Guid id, bool trackChanges)
{
    var company = _repository.Company.GetCompany(id, trackChanges);
    //Check if the company is null

    var companyDto = _mapper.Map<CompanyDto>(company);
    return companyDto;
}
```

So, we are calling the repository method that fetches a single company from the database, maps the result to **companyDto**, and returns it. You



can also see the comment about the null checks, which we are going to solve just in a minute.

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

```
[HttpGet("{id:guid}")]
public IActionResult GetCompany(Guid id)
{
    var company = _service.CompanyService.GetCompany(id, trackChanges: false);
    return Ok(company);
}
```

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:guid}")]`. You can also see that we are using a route constraint (`:guid` part) where we explicitly state that our id parameter is of the **GUID** type. We can use many different constraints like int, double, long, float, datetime, bool, length, minlength, maxlength, and many others.

Let's use Postman to send a valid request towards our API:

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

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

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

Body Pretty Raw Preview Visualize JSON 200 OK 41 ms 268 B Save Response

Pretty

```
1 {  
2     "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",  
3     "name": "Admin_Solutions Ltd",  
4     "fullAddress": "312 Forest Avenue, BF 923 USA"  
5 }
```

Great. This works as expected. But, what if someone uses an invalid id parameter?



6.1.1 Handling Invalid Requests in a Service Layer

As you can see, in our service method, we have a comment stating that the result returned from the repository could be null, and this is something we have to handle. We want to return the **NotFound** response to the client but without involving our controller's actions. We are going to keep them nice and clean as they already are.

So, what we are going to do is to create custom exceptions that we can call from the service methods and interrupt the flow. Then our error handling middleware can catch the exception, process the response, and return it to the client. This is a great way of handling invalid requests inside a service layer without having additional checks in our controllers.

That said, let's start with a new **Exceptions** folder creation inside the **Entities** project. Since, in this case, we are going to create a not found response, let's create a new **NotFoundException** class inside that folder:

```
public abstract class NotFoundException : Exception
{
    protected NotFoundException(string message)
        : base(message)
    { }
}
```

This is an abstract class, which will be a base class for all the individual not found exception classes. It inherits from the **Exception** class to represent the errors that happen during application execution. Since in our current case, we are handling the situation where we can't find the company in the database, we are going to create a new **CompanyNotFoundException** class in the same **Exceptions** folder:

```
public sealed class CompanyNotFoundException : NotFoundException
{
    public CompanyNotFoundException(Guid companyId)
        : base($"The company with id: {companyId} doesn't exist in the
database.")
    {
    }
}
```



Right after that, we can remove the comment in the GetCompany method and throw this exception:

```
public CompanyDto GetCompany(Guid id, bool trackChanges)
{
    var company = _repository.Company.GetCompany(id, trackChanges);
    if (company is null)
        throw new CompanyNotFoundException(id);

    var companyDto = _mapper.Map<CompanyDto>(company);
    return companyDto;
}
```

Finally, we have to modify our error middleware because we don't want to return the 500 error message to our clients for every custom error we throw from the service layer.

So, let's modify the **ExceptionMiddlewareExtensions** class in the main project:

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

                var contextFeature = context.Features.Get<IExceptionHandlerFeature>();
                if (contextFeature != null)
                {
                    context.Response.StatusCode = contextFeature.Error switch
                    {
                        NotFoundException => StatusCodes.Status404NotFound,
                        _ => StatusCodes.Status500InternalServerError
                    };
                }

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

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



We remove the hardcoded **StatusCode** setup and add the part where we populate it based on the type of exception we throw in our service layer. We are also dynamically populating the **Message** property of the **ErrorDetails** object that we return as the response.

Additionally, you can see the advantage of using the base abstract exception class here (`NotFoundException` in this case). We are not checking for the specific class implementation but the base type. This allows us to have multiple not found classes that inherit from the `NotFoundException` class and this middleware will know that we want to return the `NotFound` response to the client.

Excellent. Now, we can start the app and send the invalid request:

The screenshot shows a Postman request to `https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce2`. The response status is `404 Not Found` with a time of `362 ms` and a size of `329 B`. The JSON response body is:

```
1 "StatusCode": 404,
2 "Message": "The company with id: 3d490a70-94ce-4d15-9494-5248280c2ce2 doesn't exist
3 in the database."
4
```

We can see the status code we require and also the response object with proper **StatusCode** and **Message** properties. Also, if you inspect the log message, you will see that we are logging a correct message.

With this approach, we have perfect control of all the exceptional cases in our app. We have that control due to global error handler implementation. For now, we only handle the invalid id sent from the client, but we will handle more exceptional cases in the rest of the project.



In our tests **for a published app**, the regular request sent from Postman took 7ms and the exceptional one took 14ms. So you can see how fast the response is.

Of course, we are using exceptions only for these exceptional cases (Company not found, Employee not found...) and not throwing them all over the application. So, if you follow the same strategy, you will not face any performance issues.

Lastly, if you have an application where you have to throw custom exceptions more often and maybe impact your performance, we are going to provide an alternative to exceptions in the first bonus chapter of this book (Chapter 32).

6.2 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 in the Presentation project and name it **EmployeesController**:

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

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

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 we are 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).ToList();
```

Then, before we start adding code to the service layer, we are going to create a new DTO. Let's name it EmployeeDto and add it to the **Shared/DataTransferObjects** folder:

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

Since we want to return this DTO to the client, we have to create a mapping rule inside the **MappingProfile** class:

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

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

Now, we can modify the **IEmployeeService** interface:

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

And of course, we have to implement this interface in the **EmployeeService** class:

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



```
    var employeesFromDb = _repository.Employee.GetEmployees(companyId,
trackChanges);
    var employeesDto = _mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

    return employeesDto;
}
```

Here, we first fetch the company entity from the database. If it doesn't exist, we return the **NotFound** response to the client. If it does, we fetch all the employees for that company, map them to the collection of EmployeeDto and return it to the caller.

Finally, let's modify the Employees controller:

```
[HttpGet]
public IActionResult GetEmployeesForCompany(Guid companyId)
{
    var employees = _service.EmployeeService.GetEmployees(companyId, trackChanges:
false);
    return Ok(employees);
}
```

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.

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

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



Ultimate ASP.NET Core Web API

The screenshot shows a Postman request for a GET operation on the URL <https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees>. The response status is 200 OK with a time of 50 ms and a size of 364 B. The response body is a JSON object with two employees:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
{
  "id": "86dba8c0-d178-41e7-938c-ed49778fb52a",
  "name": "Jana McLeaf",
  "age": 30,
  "position": "Software developer"
},
{
  "id": "80abbca8-664d-4b20-b5de-024705497d4a",
  "name": "Sam Raiden",
  "age": 26,
  "position": "Software developer"
}
```

And with an invalid companyId:

The screenshot shows a Postman request for a GET operation on the URL <https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991873/employees>. The response status is 404 Not Found with a time of 367 ms and a size of 329 B. The response body is a JSON object with a message:

```
1
2
3
4
{
  "statusCode": 404,
  "Message": "The company with id: c9d4c053-49b6-410c-bc78-2d54a9991873 doesn't exist in the database."
}
```

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

6.3 Getting a Single Employee for Company

So, as we did in previous sections, let's start with the **IEmployeeRepository** 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();
```

Next, let's add another exception class in the **Entities/Exceptions** folder:

```
public class EmployeeNotFoundException : NotFoundException
{
    public EmployeeNotFoundException(Guid employeeId)
        : base($"Employee with id: {employeeId} doesn't exist in the database.")
    {
    }
}
```

We will soon see why do we need this class.

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

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

And implement this new method in the **EmployeeService** class:

```
public EmployeeDto GetEmployee(Guid companyId, Guid id, bool trackChanges)
{
    var company = _repository.Company.GetCompany(companyId, trackChanges);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

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

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

This is also a pretty clear code and we can see the reason for creating a new exception class.

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

```
[HttpGet("{id:guid}")]
public IActionResult GetEmployeeForCompany(Guid companyId, Guid id)
{
    var employee = _service.EmployeeService.GetEmployee(companyId, id,
trackChanges: false);
    return Ok(employee);
```



```
}
```

Excellent. You can see how clear our action is.

We can test this action by using already created requests from the **Bonus 2-CompanyEmployeesRequests.postman_collection.json** 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>

The screenshot shows a POSTMAN interface with the following details:

- Method: GET
- URL: https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/86dba8c0-d178-41e7-938c-ed49778fb52a
- Headers: (6)
- Body: (Raw JSON response)
- Test Results: Status: 200 OK Time: 82 ms
- Response Body (Pretty JSON):

```
1 {  
2   "id": "86dba8c0-d178-41e7-938c-ed49778fb52a",  
3   "name": "Jana McLeaf",  
4   "age": 30,  
5   "position": "Software developer"  
6 }
```

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>

The screenshot shows a POSTMAN interface with the following details:

- Method: GET
- URL: https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/86dba8c0-d178-41e7-938c-ed49778fb52c
- Headers: (6)
- Body: (Raw JSON response)
- Test Results: Status: 404 Not Found Time: 575 ms Size: 326 B Save Response
- Response Body (Pretty JSON):

```
1 {  
2   "statusCode": 404,  
3   "message": "Employee with id: 86dba8c0-d178-41e7-938c-ed49778fb52c doesn't exist in the database."  
4 }
```



The screenshot shows a Postman request for a GET method at the URL `https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991872/employ`. The response status is 404 Not Found, with a duration of 407 ms and a size of 329 B. The response body is a JSON object:

```
1 "StatusCode": 404,
2 "Message": "The company with id: c9d4c053-49b6-410c-bc78-2d54a9991872 doesn't exist in
3             the database."
4
```

Our responses are pretty self-explanatory, which makes for a good user experience.

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.



7 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 to get in a response to the REST API request.

7.1 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:

The screenshot shows a Postman request to `https://localhost:5001/api/companies`. The `Accept` header is set to `text/xml`, which is highlighted with a red border. The response is a 200 OK status with a JSON payload containing two company objects:

```
1 {
2   "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
3   "name": "Admin_Solutions Ltd",
4   "fullAddress": "312 Forest Avenue, BF 923 USA"
5 },
6 {
7   "id": "c9d4c053-49b6-410c-bc78-2d54a9991870",
8   "name": "IT_Solutions Ltd",
9   "fullAddress": "583 Wall Dr. Gwynn Oak, MD 21207 USA"
10 }
```



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.

7.2 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:

```
builder.Services.ConfigureCors();
builder.Services.ConfigureIISIntegration();
builder.Services.ConfigureLoggerService();
builder.Services.ConfigureRepositoryManager();
builder.Services.ConfigureServiceManager();
builder.Services.ConfigureDbContext(builder.Configuration);
builder.Services.AddAutoMapper(typeof(Program));

builder.Services.AddControllers(config => {
    config.RespectBrowserAcceptHeader = true;
}).AddXmlDataContractSerializerFormatters()
.AddApplicationPart(typeof(CompanyEmployees.Presentation.AssemblyReference).Assembly);
```

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.



7.3 Testing Content Negotiation

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

<https://localhost:5001/api/companies>

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

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Headers 6 hidden

KEY	VALUE	DESCR	...	Bulk Edit	Presets
Accept	text/xml				

Body Cookies Headers (7) Test Results 500 Internal Server Error 3.19 s 661 B Save Response

Pretty Raw Preview Visualize JSON

```
1 "StatusCode": 500,
2 "Message": "Type 'Shared.DataTransferObjects.CompanyDto' cannot be serialized. Consider
3     marking it with the DataContractAttribute attribute, and marking all of its members
4     you want serialized with theDataMemberAttribute attribute. Alternatively, you can
5     ensure that the type is public and has a parameterless constructor - all public
6     members of the type will then be serialized, and no attributes will be required."
```

We get an error because XmlSerializer cannot easily serialize our positional record type. There are two solutions to this. The first one is marking our CompanyDto record with the [Serializable] attribute:

```
[Serializable]
public record CompanyDto(Guid Id, string Name, string FullAddress);
```

Now, we can send the same request again:

```
1 <ArrayOfCompanyDto xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.
2     datacontract.org/2004/07/Shared.DataTransferObjects">
3         <CompanyDto>
4             <_x003C_FullAddress_x003E_k__BackingField>312 Forest Avenue, BF 923 USA</
5                 _x003C_FullAddress_x003E_k__BackingField>
6             <_x003C_Id_x003E_k__BackingField>3d490a70-94ce-4d15-9494-5248280c2ce3</
7                 _x003C_Id_x003E_k__BackingField>
8             <_x003C_Name_x003E_k__BackingField>Admin_Solutions Ltd</
9                 _x003C_Name_x003E_k__BackingField>
10            </CompanyDto>
11        <CompanyDto>
12            <_x003C_FullAddress_x003E_k__BackingField>583 Wall Dr. Gwynn Oak, MD 21207 USA</
13                _x003C_FullAddress_x003E_k__BackingField>
14            <_x003C_Id_x003E_k__BackingField>c9d4c053-49b6-410c-bc78-2d54a9991870</
15                _x003C_Id_x003E_k__BackingField>
16            <_x003C_Name_x003E_k__BackingField>IT_Solutions Ltd</_x003C_Name_x003E_k__BackingField>
17        </CompanyDto>
18    </ArrayOfCompanyDto>
```

This time, we are getting our XML response but, as you can see, properties have some strange names. That's because the compiler behind



the scenes generates the record as a class with fields named like that (name_BackingField) and the XML serializer just serializes those fields with the same names.

If we don't want these property names in our response, but the regular ones, we can implement a second solution. Let's modify our record with the init only property setters:

```
public record CompanyDto
{
    public Guid Id { get; init; }
    public string? Name { get; init; }
    public string? FullAddress { get; init; }
}
```

This object is still immutable and init-only properties protect the state of the object from mutation once initialization is finished.

Additionally, we have to make one more change 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>();
}
```

We are returning this mapping rule to a previous state since now, we do have properties in our object.

Now, we can send the same request again:

```
<ArrayOfCompanyDto xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.
    datacontract.org/2004/07/Shared.DataTransferObjects">
    <CompanyDto>
        <FullAddress>312 Forest Avenue, BF 923 USA</FullAddress>
        <Id>3d490a70-94ce-4d15-9494-5248280c2ce3</Id>
        <Name>Admin_Solutions Ltd</Name>
    </CompanyDto>
    <CompanyDto>
        <FullAddress>583 Wall Dr. Gwynn Oak, MD 21207 USA</FullAddress>
        <Id>c9d4c053-49b6-410c-bc78-2d54a9991870</Id>
        <Name>IT_Solutions Ltd</Name>
    </CompanyDto>
</ArrayOfCompanyDto>
```



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?

7.4 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:

```
builder.Services.AddControllers(config => {
    config.RespectBrowserAcceptHeader = true;
    config.ReturnHttpNotAcceptable = true;
}).AddXmlDataContractSerializerFormatters()
.AddApplicationPart(typeof(CompanyEmployees.Presentation.AssemblyReference).Assembly);
```

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:

<https://localhost:5001/api/companies>



The screenshot shows a Postman interface with a GET request to `https://localhost:5001/api/companies`. The **Headers** tab is active, displaying a table with one row: `Accept` with value `text/css`. The status bar at the bottom right indicates a **406 Not Acceptable** response.

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

So far so good.

7.5 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 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.

7.6 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 **FormatCsv** method 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**:

```
builder.Services.AddControllers(config => {
    config.RespectBrowserAcceptHeader = true;
    config.ReturnHttpNotAcceptable = true;
}).AddXmlDataContractSerializerFormatters()
    .AddCustomCSVFormatter()
.AddApplicationPart(typeof(CompanyEmployees.Presentation.AssemblyReference).Assembly);
```

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



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies>

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

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Headers (6 hidden)

KEY	VALUE	DESCR	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Accept	text/csv				

Body Cookies Headers (4) Test Results

200 OK 2.36 s 326 B Save Response

Pretty Raw Preview Visualize Text

```
1 3d490a70-94ce-4d15-9494-5248280c2ce3,"Admin_Solutions Ltd,"312 Forest Avenue, BF 923 USA"
2 c9d4c053-49b6-410c-bc78-2d54a9991870,"IT_Solutions Ltd,"583 Wall Dr. Gwynn Oak, MD 21207 USA"
```

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.



8 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.



9 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.

9.1 Handling POST Requests

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

```
[HttpGet("{id:guid}", 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 record in the **Shared/DataTransferObjects** folder:

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

We can see that this DTO record is almost the same as the **Company** record 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.

Next, we want to modify the **ICompanyService** interface:

```
public interface ICompanyService
{
    IEnumerable<CompanyDto> GetAllCompanies(bool trackChanges);
    CompanyDto GetCompany(Guid companyId, bool trackChanges);
    CompanyDto CreateCompany(CompanyForCreationDto company);
}
```

And of course, we have to implement this method in the **CompanyService** class:

```
public CompanyDto CreateCompany(CompanyForCreationDto company)
{
    var companyEntity = _mapper.Map<Company>(company);

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

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

    return companyToReturn;
}
```

Here, 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 that, we map the company entity to the company DTO object to return it to the controller.

But we don't have the mapping rule for this so 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**, and as you can see our service method accepts the parameter of the same type as well, but we need the **Company** object to send it to the repository layer 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 is null)  
        return BadRequest("CompanyForCreationDto object is null");  
  
    var createdCompany = _service.CompanyService.CreateCompany(company);  
  
    return CreatedAtRoute("CompanyById", new { id = createdCompany.Id },  
    createdCompany);  
}
```

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

https://localhost:5001/api/companies

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

Params Auth Headers (9) Body **JSON** Pre-req. Tests Settings Cookies Beautify

1 "name": "Marketing Solutions Ltd",
2 "address": "242 Sunny Avenue, K 334",
3 "country": "USA"

Body Cookies Headers (5) Test Results
Pretty Raw Preview Visualize JSON ↻ 201 Created 107 ms 360 B Save Response ↻

1 "id": "14759d51-e9c1-4afc-f9bf-08d98898c9c3",
2 "name": "Marketing Solutions Ltd",
3 "fullAddress": "242 Sunny Avenue, K 334 USA"



9.2 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. We have already explained the code in the service method. 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 `GetCompanies` 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 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 have to validate it against the reference type's default value, which is null.

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:

Headers (5)		Test Results	201 Created	107 ms	360 B	Save Response
KEY	VALUE					
Content-Type ⓘ	application/json; charset=utf-8					
Date ⓘ	Wed, 06 Oct 2021 07:14:56 GMT					
Server ⓘ	Kestrel					
Location ⓘ	https://localhost:5001/api/companies/14759d51-e9c1-4afc-f9bf-08d98898c9c3					
Transfer-Encoding ⓘ	chunked					

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).

Excellent.

There is still one more thing we need to explain.

9.2.1 Validation from the ApiController Attribute

In this section, we are going to talk about the [ApiController] attribute that we can find right below the [Route] attribute in our controller:

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

But, before we start with the explanation, let's place a breakpoint in the **CreateCompany** action, right on the **if (company is null)** check.

Then, let's use Postman to send an invalid POST request:



The screenshot shows a Postman request to `https://localhost:5001/api/companies` using a POST method. The 'Body' tab is selected, showing a JSON payload with a single key 'company':

```
1
2   "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
3   "title": "One or more validation errors occurred.",
4   "status": 400,
5   "traceId": "00-504e16a3d2dcb7f0b3afcc3abe7e8569-fa1b7187e1731b74-00",
6   "errors": {
7     "":
8       [
9         "A non-empty request body is required."
10      ],
11     "company": [
12       "The company field is required."
13     ]
14 }
```

The response status is 400 Bad Request, and the error message indicates that both the request body and the 'company' field are required.

We are going to talk about Validation in chapter 13, but for now, we have to explain a couple of things.

First of all, we have our response - a Bad Request in Postman, and we have error messages that state what's wrong with our request. But, we never hit that breakpoint that we've placed inside the `CreateCompany` action.

Why is that?

Well, the `[ApiController]` attribute is applied to a controller class to enable the following opinionated, API-specific behaviors:

- Attribute routing requirement
- Automatic HTTP 400 responses
- Binding source parameter inference
- Multipart/form-data request inference
- Problem details for error status codes



As you can see, it handles the HTTP 400 responses, and in our case, since the request's body is null, the **[ApiController]** attribute handles that and returns the 400 (BadRequest) response before the request even hits the **CreateCompany** action.

This is useful behavior, but it prevents us from sending our custom responses with different messages and status codes to the client. This will be very important once we get to the Validation.

So to enable our custom responses from the actions, we are going to add this code into the **Program** class right above the **AddControllers** method:

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

With this, we are suppressing a default model state validation that is implemented due to the existence of the **[ApiController]** attribute in all API controllers. So this means that we can solve the same problem differently, by commenting out or removing the **[ApiController]** attribute only, without additional code for suppressing validation. It's all up to you. But we like keeping it in our controllers because, as you could've seen, it provides additional functionalities other than just 400 – Bad Request responses.

Now, once we start the app and send the same request, we will hit that breakpoint and see our response in Postman.

Nicely done.

Now, we can remove that breakpoint and continue with learning about the creation of child resources.



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

We don't have the **Id** property because we are going to create that Id on the server-side. But additionally, 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 and send it to a service method, 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>();
```

The next thing we have to do is **IEmployeeService** modification:

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



And implement this new method in EmployeeService:

```
public EmployeeDto CreateEmployeeForCompany(Guid companyId, EmployeeForCreationDto employeeForCreation, bool trackChanges)
{
    var company = _repository.Company.GetCompany(companyId, trackChanges);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

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

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

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

    return employeeToReturn;
}
```

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. After that, we map the DTO to an entity, call the repository methods to create a new employee, map back the entity to the DTO, and return it to the caller.

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

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

    var employeeToReturn =
_service.EmployeeService.CreateEmployeeForCompany(companyId, employee, trackChanges: false);

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

As we can see, the main difference between this action and the **CreateCompany** action (if we exclude the fact that we are working with different DTOs) is the return statement, which now has two parameters for the anonymous object.



Ultimate ASP.NET Core Web API

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

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

Let's give this a try:

<https://localhost:5001/api/companies/14759d51-e9c1-4afc-f9bf-08d98898c9c3/employees>

The screenshot shows a POST request in Postman. The URL is <https://localhost:5001/api/companies/14759d51-e9c1-4afc-f9bf-08d98898c9c3/employees>. The request body is a JSON object:

```
1 {  
2   "name": "Martin Geil",  
3   "age": 29,  
4   "position": "Marketing expert"  
5 }
```

The response status is 201 Created, with a response time of 382 ms and a response size of 390 B. The response body is identical to the request body:

```
1 {  
2   "id": "e06cfcc6-e353-4bd8-0870-08d988af0956",  
3   "name": "Martin Geil",  
4   "age": 29,  
5   "position": "Marketing expert"  
6 }
```

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:

The screenshot shows a GET request in Postman. The URL is <https://localhost:5001/api/companies/14759d51-e9c1-4afc-f9bf-08d98898c9c3/employees/e06cfcc6-e353-4bd8-0870-08d988af0956>. The response status is 200 OK, with a response time of 68 ms and a response size of 253 B. The response body is identical to the previous one:

```
1 {  
2   "id": "e06cfcc6-e353-4bd8-0870-08d988af0956",  
3   "name": "Martin Geil",  
4   "age": 29,  
5   "position": "Marketing expert"  
6 }
```



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

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

The screenshot shows a Postman request to `https://localhost:5001/api/companies` using the `POST` method. The `Body` tab is selected, showing a JSON payload:

```
1 {  
2   "name": "Electronics Solutions Ltd",  
3   "address": "312 Deliver Street, F 234",  
4   "country": "USA",  
5   "employees": [  
6     {  
7       "name": "Joan Dane",  
8       "age": 29,  
9       "position": "Manager"  
10      },  
11      {  
12        "name": "Martin Geil",  
13        "age": 29,  
14        "position": "Administrative"  
15      }  
16    ]  
17 }
```

The response status is `201 Created` with a response time of `416 ms` and a size of `364 B`. The response body is:

```
1 {  
2   "id": "9b453d25-3c20-4494-9c4c-08d988b17109",  
3   "name": "Electronics Solutions Ltd",  
4   "fullAddress": "312 Deliver Street, F 234 USA"  
5 }
```



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:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

```
[{"id": "7162979e-cc1c-4d63-47b2-08d988b17111", "name": "Joan Dane", "age": 29, "position": "Manager"}, {"id": "3dd2bbcc-f17b-49dc-47b3-08d988b17111", "name": "Martin Geil", "age": 29, "position": "Administrative"}]
```

We have confirmed that the employees were created as well.

9.5 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 `CompanyId` 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 are going to modify **ICompanyService**:

```
public interface ICompanyService
{
    IEnumerable<CompanyDto> GetAllCompanies(bool trackChanges);
    CompanyDto GetCompany(Guid companyId, bool trackChanges);
    CompanyDto CreateCompany(CompanyForCreationDto company);
    IEnumerable<CompanyDto> GetByIds(IEnumerable<Guid> ids, bool trackChanges);
}
```

And implement this in **CompanyService**:

```
public IEnumerable<CompanyDto> GetByIds(IEnumerable<Guid> ids, bool trackChanges)
{
    if (ids is null)
        throw new IdParametersBadRequestException();

    var companyEntities = _repository.Company.GetByIds(ids, trackChanges);
    if (ids.Count() != companyEntities.Count())
        throw new CollectionByIdsBadRequestException();

    var companiesToReturn = _mapper.Map<IEnumerable<CompanyDto>>(companyEntities);

    return companiesToReturn;
}
```

Here, we check if **ids** parameter is null and if it is we stop the execution flow and return a bad request response to the client. If it's not null, we fetch all the companies for each id in the **ids** collection. If the count of ids and companies mismatch, we return another bad request response to the client. Finally, we are executing the mapping action and returning the result to the caller.

Of course, we don't have these two exception classes yet, so let's create them.

Since we are returning a bad request result, we are going to create a new abstract class in the **Entities/Exceptions** folder:



```
public abstract class BadRequestException : Exception
{
    protected BadRequestException(string message)
        :base(message)
    {
    }
}
```

Then, in the same folder, let's create two new specific exception classes:

```
public sealed class IdParametersBadRequestException : BadRequestException
{
    public IdParametersBadRequestException()
        :base("Parameter ids is null")
    {
    }
}

public sealed class CollectionByIdsBadRequestException : BadRequestException
{
    public CollectionByIdsBadRequestException()
        :base("Collection count mismatch comparing to ids.")
    {
    }
}
```

At this point, we've removed two errors from the GetByIds method. But, to show the correct response to the client, we have to modify the **ConfigureExceptionHandler** class – the part where we populate the StatusCode property:

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

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

```
[HttpGet("collection/({ids})", Name = "CompanyCollection")]
public IActionResult GetCompanyCollection(IEnumerable<Guid> ids)
{
    var companies = _service.CompanyService.GetByIds(ids, trackChanges: false);

    return Ok(companies);
}
```

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

Let's modify the **ICompanyService** interface first:



```
public interface ICompanyService
{
    IEnumerable<CompanyDto> GetAllCompanies(bool trackChanges);
    CompanyDto GetCompany(Guid companyId, bool trackChanges);
    CompanyDto CreateCompany(CompanyForCreationDto company);
    IEnumerable<CompanyDto> GetByIds(IEnumerable<Guid> ids, bool trackChanges);
    (IEnumerable<CompanyDto> companies, string ids) CreateCompanyCollection
        (IEnumerable<CompanyForCreationDto> companyCollection);
}
```

So, this new method will accept a collection of the **CompanyForCreationDto** type as a parameter, and return a Tuple with two fields (companies and ids) as a result.

That said, let's implement it in the **CompanyService** class:

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

    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 (companies: companyCollectionToReturn, ids: ids);
}
```

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 map the company collection back, take all the ids as a comma-separated string, and return the Tuple with these two fields as a result to the caller.

Again, we can see that we don't have the exception class, so let's just create it:

```
public sealed class CompanyCollectionBadRequest : BadRequestException
{
    public CompanyCollectionBadRequest()
        :base("Company collection sent from a client is null.")
```



```
{  
}  
}
```

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

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

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

We receive the **companyCollection** parameter from the client, send it to the service method, and return a result with a comma-separated string and our newly 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:

```
access-control-allow-origin → *
content-type → application/json; charset=utf-8
date → Wed, 25 Nov 2020 10:15:17 GMT
location → https://localhost:5001/api/companies/collection/(System.Linq.Enumerable%2BSelectListIterator%602%5BEntities.DataTransferObjects.CompanyDto, System.Guid%5D)
server → Kestrel
```

We can test our create action now with a bad request:



Ultimate ASP.NET Core Web API

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

The screenshot shows a Postman request to <https://localhost:5001/api/companies/collection> using a POST method. The 'Body' tab is selected, showing a JSON payload with a single field 'Message': "Company collection sent from a client is null.". The response status is 400 Bad Request, with a message indicating the error: "StatusCode": 400, "Message": "Company collection sent from a client is null.".

We can see that the request is handled properly and we have a correct response.

Now, let's send a valid request:

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

The screenshot shows a Postman request to <https://localhost:5001/api/companies/collection> using a POST method. The 'Body' tab is selected, showing a JSON array with two company objects. The response status is 201 Created, with a message indicating success: "id": "582ea192-6fb7-44ff-a2a1-08d988ca3ca9", "name": "Sales all over the world Ltd", "fullAddress": "355 Open Street, B 784 USA".

Excellent. Let's check the header tab:



KEY	VALUE
Content-Type ⓘ	application/json; charset=utf-8
Date ⓘ	Wed, 06 Oct 2021 13:07:22 GMT
Server ⓘ	Kestrel
Location ⓘ	https://localhost:5001/api/companies/collection/(582ea192-6fb7-44ff-a2a1-08d988ca3ca9,a216fbbbe-ebbd-4e09-a2a2-08d988ca3ca9)
Transfer-Encoding ⓘ	chunked

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

The screenshot shows a Postman request configuration. The method is set to GET, and the URL is https://localhost:5001/api/companies/collection/(582ea192-6fb7-44ff-a2a1-08d988ca3ca9,a216fbbbe-ebbd-4e09-a2a2-08d988ca3ca9). The Headers tab shows a Content-Type header of application/json. The response status is 415 Unsupported Media Type, with a detailed message: "type": "https://tools.ietf.org/html/rfc7231#section-6.5.13", "title": "Unsupported Media Type", "status": 415, "traceId": "00-215c2e5b0e9f0dc2b120ada53a00a788-07470cf7e200ac0c-00".

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 in the **GetCompanyCollection** action.

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

9.6 Model Binding in API

Let's create the new folder **ModelBinders** in the **Presentation** 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 a 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**.

These are the required using directives:

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

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)
```

This is the required namespace:

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

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

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:



Ultimate ASP.NET Core Web API

[https://localhost:5001/api/companies/collection/\(582ea192-6fb7-44ff-a2a1-08d988ca3ca9,a216fbbe-ebbd-4e09-a2a2-08d988ca3ca9\)](https://localhost:5001/api/companies/collection/(582ea192-6fb7-44ff-a2a1-08d988ca3ca9,a216fbbe-ebbd-4e09-a2a2-08d988ca3ca9))

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** https://localhost:5001/api/companies/collection/(582ea192-6fb7-44ff-a2a1-08d988ca3ca9,a216fbbe-ebbd-4e09-a2a2-08d988ca3ca9)
- Status:** 200 OK (66 ms, 387 B)
- Headers:** (4)
- Body:** (Pretty) JSON response showing two company objects:

```
1
2 {
3     "id": "582ea192-6fb7-44ff-a2a1-08d988ca3ca9",
4     "name": "Sales all over the world Ltd",
5     "fullAddress": "355 Open Street, B 784 USA"
6 },
7 {
8     "id": "a216fbbe-ebbd-4e09-a2a2-08d988ca3ca9",
9     "name": "Branding Ltd",
10    "fullAddress": "255 Main Street, K 334 USA"
11 }
12 ]
```

Well done.

We are ready to continue towards DELETE actions.



10 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);
```

After that, we have to modify the **IEmployeeService** interface:

```
public interface IEmployeeService
{
    IEnumerable<EmployeeDto> GetEmployees(Guid companyId, bool trackChanges);
    EmployeeDto GetEmployee(Guid companyId, Guid id, bool trackChanges);
    EmployeeDto CreateEmployeeForCompany(Guid companyId, EmployeeForCreationDto
employeeForCreation, bool trackChanges);
    void DeleteEmployeeForCompany(Guid companyId, Guid id, bool trackChanges);
}
```

And of course, the **EmployeeService** class:

```
public void DeleteEmployeeForCompany(Guid companyId, Guid id, bool trackChanges)
{
    var company = _repository.Company.GetCompany(companyId, trackChanges);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

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

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

Pretty straightforward method implementation where we fetch the company and if it doesn't exist, we return the Not Found response. If it exists, we fetch the employee for that company and execute the same check, where if it's true, we return another not found response. Lastly, we delete the employee from the database.

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



```
[HttpDelete("{id:guid}")]
public IActionResult DeleteEmployeeForCompany(Guid companyId, Guid id)
{
    _service.EmployeeService.DeleteEmployeeForCompany(companyId, id, trackChanges:
false);

    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. Call the service method and return the **NoContent()** method, which returns the status code **204 No Content**.

Let's test this:

<https://localhost:5001/api/companies/14759d51-e9c1-4afc-f9bf-08d98898c9c3/employees/e06cfcc6-e353-4bd8-0870-08d988af0956>

The screenshot shows a Postman request for a DELETE operation. The URL is <https://localhost:5001/api/companies/14759d51-e9c1-4afc-f9bf-08d98898c9c3/employees/e06cfcc6-e353-4bd8-0870-08d988af0956>. The response status is 204 No Content, with a response time of 355 ms and a body size of 81 B.

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/14759d51-e9c1-4afc-f9bf-08d98898c9c3/employees/e06cfcc6-e353-4bd8-0870-08d988af0956>

The screenshot shows a Postman request for a GET operation. The URL is <https://localhost:5001/api/companies/14759d51-e9c1-4afc-f9bf-08d98898c9c3/employees/e06cfcc6-e353-4bd8-0870-08d988af0956>. The response status is 404 Not Found, with a response time of 354 ms and a body size of 326 B. The response body is a JSON object with StatusCode: 404 and Message: "Employee with id: e06cfcc6-e353-4bd8-0870-08d988af0956 doesn't exist in the database."

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.

10.1 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);
```

Then we have to modify the service interface:

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



And the service class:

```
public void DeleteCompany(Guid companyId, bool trackChanges)
{
    var company = _repository.Company.GetCompany(companyId, trackChanges);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

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

Finally, let's modify our controller:

```
[HttpDelete("{id:guid}")]
public IActionResult DeleteCompany(Guid id)
{
    _service.CompanyService.DeleteCompany(id, trackChanges: false);

    return NoContent();
}
```

And let's test our action:

<https://localhost:5001/api/companies/0AD5B971-FF51-414D-AF01-34187E407557>

The screenshot shows a Postman request for a DELETE operation. The URL is <https://localhost:5001/api/companies/9b453d25-3c20-4494-9c4c-08d988b17109>. The response status is 204 No Content, with a duration of 3.25 s and a size of 81 B.

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.



11 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.

11.1 Updating Employee

In the previous sections, we first changed our interface, then the repository/service classes, 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 record for update purposes:

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

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 an additional DTO record, we require an additional mapping rule:

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

After adding the mapping rule, we can modify the **IEmployeeService** interface:

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



```
EmployeeDto CreateEmployeeForCompany(Guid companyId, EmployeeForCreationDto employeeForCreation, bool trackChanges);
void DeleteEmployeeForCompany(Guid companyId, Guid id, bool trackChanges);
void UpdateEmployeeForCompany(Guid companyId, Guid id,
    EmployeeForUpdateDto employeeForUpdate, bool compTrackChanges, bool empTrackChanges);
}
```

We are declaring a method that contains both id parameters – one for the company and one for employee, the **employeeForUpdate** object sent from the client, and two track changes parameters, again, one for the company and one for the employee. We are doing that because we won't track changes while fetching the company entity, but we will track changes while fetching the employee.

That said, let's modify the **EmployeeService** class:

```
public void UpdateEmployeeForCompany(Guid companyId, Guid id, EmployeeForUpdateDto employeeForUpdate,
    bool compTrackChanges, bool empTrackChanges)
{
    var company = _repository.Company.GetCompany(companyId, compTrackChanges);
    if (company is null)
        throw new CompanyNotFoundException(companyId);

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

    _mapper.Map(employeeForUpdate, employeeEntity);
    _repository.Save();
}
```

So first, we fetch the company from the database. If it doesn't exist, we interrupt the flow and send the response to the client. After that, we do the same thing for the employee. But there is one difference here. Pay attention to the way we fetch the **company** and the way we fetch the **employeeEntity**. Do you see the difference?

As we've already said: the **trackChanges** parameter will be 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 **employeeForUpdate** 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.

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

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

    _service.EmployeeService.UpdateEmployeeForCompany(companyId, id, employee,
        compTrackChanges: false, empTrackChanges: true);

    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}.

Then, we check if the employee object is null, and if it is, we return a **BadRequest** response.

After that, we just call the update method from the service layer and pass **false** for the company track changes and **true** for the employee track changes.

Finally, we return the 204 **NoContent** status.

We can test our action:



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

The screenshot shows a POSTMAN interface. The URL is <https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBCA8-664D-4B20-B5DE-024705497D4A>. The method is set to PUT. The body contains the following JSON:

```
1 "name": "Sam Raiden",
2 "age": 25, ← Age changed from 26 to 25
3 "position": "Software developer"
4
5
```

The response status is 204 No Content, with a time of 327 ms and a size of 81 B. The response body is empty.

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:

```
SET NOCOUNT ON;
UPDATE [Employees] SET [Age] = @p0
WHERE [EmployeeId] = @p1;
SELECT @@ROWCOUNT;
```

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 to this kind of situation.

NOTE: We've 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, 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.



11.1.1 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.

11.2 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 record for update:

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

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

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

Then, let's move on to the interface modification:

```
public interface ICompanyService  
{
```



```
IEnumerable<CompanyDto> GetAllCompanies(bool trackChanges);
CompanyDto GetCompany(Guid companyId, bool trackChanges);
CompanyDto CreateCompany(CompanyForCreationDto company);
IEnumerable<CompanyDto> GetByIds(IEnumerable<Guid> ids, bool trackChanges);
(IEnumerable<CompanyDto> companies, string ids) CreateCompanyCollection
    (IEnumerable<CompanyForCreationDto> companyCollection);
void DeleteCompany(Guid companyId, bool trackChanges);
void UpdateCompany(Guid companyId, CompanyForUpdateDto companyForUpdate, bool
trackChanges);
}
```

And of course, the service class modification:

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

    _mapper.Map(companyForUpdate, companyEntity);
    _repository.Save();
}
```

So again, we fetch our company entity from the database, and if it is null, we just return the NotFound response. But if it's not null, we map the **companyForUpdate** DTO to **companyEntity** and call the **Save** method.

Right now, we can modify our controller:

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

    _service.CompanyService.UpdateCompany(id, company, trackChanges: true);

    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:



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

PUT

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

Send

Params Auth Headers (9) Body ● Pre-req. Tests Settings

Cookies

raw

JSON

Beautify

```
1  "name": "Admin_Solutions_Ltd_Upd",
2  "address": "312 Forest Avenue, BF 923",
3  "country": "USA",
4  "employees": [
5    {
6      "name": "Geil Metain",
7      "age": 23,
8      "position": "Admin"
9    }
10   ]
11 ]
12 ]
```

Body Cookies Headers (2) Test Results

204 No Content 576 ms 81 B Save Response

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:

```
SET NOCOUNT ON;
UPDATE [Companies] SET [Name] = @p0
WHERE [CompanyId] = @p1;
SELECT @@ROWCOUNT; ①

info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (13ms) [Parameters=@p0='?' (DbType = Guid), @p1='?' (DbType = Int32), @p2='?' (DbType = Guid),
      @p3='?' (Size = 30), @p4='?' (Size = 20)], CommandType='Text', CommandTimeout='30'
      SET NOCOUNT ON;
      INSERT INTO [Employees] ([EmployeeId], [Age], [CompanyId], [Name], [Position])
      VALUES (@p0, @p1, @p2, @p3, @p4); ②
```

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.



12 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 with 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.

12.1 Applying PATCH to the Employee Entity

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

- The **Microsoft.AspNetCore.JsonPatch** library, in the **Presentation** project, to support the usage of **JsonPatchDocument** in our controller and
- The **Microsoft.AspNetCore.Mvc.NewtonsoftJson** library, in the main project, to support request body conversion to a **PatchDocument** once we send our request.



As you can see, we are still using the NewtonsoftJson library to support the PatchDocument conversion. The official statement from Microsoft is that they are not going to replace it with System.Text.Json: "The main reason is that this will require a huge investment from us, with not a very high value-add for the majority of our customers.".

By using **AddNewtonsoftJson**, we are replacing the **System.Text.Json** formatters for **all** JSON content. We don't want to do that so, we are going to add a simple workaround in the **Program** class:

```
NewtonsoftJsonPatchInputFormatter GetJsonPatchInputFormatter() =>
    new ServiceCollection().AddLogging().AddMvc().AddNewtonsoftJson()
        .Services.BuildServiceProvider()
        .GetRequiredService<IOptions<MvcOptions>>().Value.InputFormatters
            .OfType<NewtonsoftJsonPatchInputFormatter>().First();
```

By adding a method like this in the Program class, we are creating a local function. This function configures support for JSON Patch using Newtonsoft.Json while leaving the other formatters unchanged.

For this to work, we have to include two more namespaces in the class:

```
using Microsoft.AspNetCore.Mvc.Formatters;
using Microsoft.Extensions.Options;
```

After that, we have to modify the **AddControllers** method:

```
builder.Services.AddControllers(config => {
    config.RespectBrowserAcceptHeader = true;
    config.ReturnHttpNotAcceptable = true;
    config.InputFormatters.Insert(0, GetJsonPatchInputFormatter());
}).AddXmlDataContractSerializerFormatters()
```

We are placing our **JsonPatchInputFormatter** at the index 0 in the **InputFormatters** list.

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.

After that, we are going to add two new method contracts to the **IEmployeeService** interface:

```
(EmployeeForUpdateDto employeeToPatch, Employee employeeEntity) GetEmployeeForPatch(  
    Guid companyId, Guid id, bool compTrackChanges, bool empTrackChanges);  
  
void SaveChangesForPatch(EmployeeForUpdateDto employeeToPatch, Employee  
employeeEntity);
```

Of course, for this to work, we have to add the reference to the **Entities** project.

Then, we have to implement these two methods in the **EmployeeService** class:

```
public (EmployeeForUpdateDto employeeToPatch, Employee employeeEntity)  
GetEmployeeForPatch  
    (Guid companyId, Guid id, bool compTrackChanges, bool empTrackChanges)  
{  
    var company = _repository.Company.GetCompany(companyId, compTrackChanges);  
    if (company is null)  
        throw new CompanyNotFoundException(companyId);  
  
    var employeeEntity = _repository.Employee.GetEmployee(companyId, id,  
empTrackChanges);  
    if (employeeEntity is null)  
        throw new EmployeeNotFoundException(companyId);  
  
    var employeeToPatch = _mapper.Map<EmployeeForUpdateDto>(employeeEntity);  
  
    return (employeeToPatch, employeeEntity);  
}  
  
public void SaveChangesForPatch(EmployeeForUpdateDto employeeToPatch, Employee  
employeeEntity)  
{  
    _mapper.Map(employeeToPatch, employeeEntity);  
    _repository.Save();  
}
```



In the first method, we are trying to fetch both the company and employee from the database and if we can't find either of them, we stop the execution flow and return the `NotFound` response to the client. Then, we map the employee entity to the `EmployeeForUpdateDto` type and return both objects (`employeeToPatch` and `employeeEntity`) inside the Tuple to the controller.

The second method just maps from `employeeToPatch` to `employeeEntity` and calls the repository's `Save` method.

Now, we can modify our controller:

```
[HttpPatch("{id:guid}")]
public IActionResult PartiallyUpdateEmployeeForCompany(Guid companyId, Guid id,
    [FromBody] JsonPatchDocument<EmployeeForUpdateDto> patchDoc)
{
    if (patchDoc is null)
        return BadRequest("patchDoc object sent from client is null.");

    var result = _service.EmployeeService.GetEmployeeForPatch(companyId, id,
compTrackChanges: false,
empTrackChanges: true);

    patchDoc.ApplyTo(result.employeeToPatch);

    _service.EmployeeService.SaveChangesForPatch(result.employeeToPatch,
result.employeeEntity);

    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 it is, we return a `BadRequest`. Then we call the service method where we map from the `Employee` type to the `EmployeeForUpdateDto` type; we need to do that because the `patchDoc` variable can apply only to the `EmployeeForUpdateDto` type. After apply is executed, we call another service method to map again to the `Employee` type (from `employeeToPatch` to `employeeEntity`) and save changes in the database. In the end, we return `NoContent`.



Ultimate ASP.NET Core Web API

Don't forget to include an additional namespace:

```
using Microsoft.AspNetCore.JsonPatch;
```

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>

The screenshot shows a POSTMAN interface. The URL is <https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBCA8-664D-4B20-B5DE-024705497D4A>. The method is PATCH. The body contains a JSON patch document:

```
1 [ { 2   "op": "replace", 3   "path": "/age", 4   "value": "28"}]
```

A red box highlights the value "28" with the annotation "From 25 to 28". Below the body, the response status is 204 No Content.

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>

The screenshot shows a POSTMAN interface. The URL is <https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBCA8-664D-4B20-B5DE-024705497D4A>. The method is GET. The response status is 200 OK. The response body is:

```
1 { 2   "id": "80abbc8-664d-4b20-b5de-024705497d4a", 3   "name": "Sam Raiden", 4   "age": 28, 5   "position": "Software developer"}
```

And we see the **Age** property has been changed.

Let's send a remove operation in a request:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBKA8-664D-4B20-B5DE-024705497D4A>

The screenshot shows a POSTMAN interface. The URL is <https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBKA8-664D-4B20-B5DE-024705497D4A>. The method is PATCH. The body is a JSON patch document:

```
1 {  
2   "op": "remove",  
3   "path": "/age",  
4   "value": "28"  
5 }
```

The response status is 204 No Content, with a time of 74 ms and a size of 81 B. There are 2 headers listed.

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):

```
1 {  
2   "id": "80abbka8-664d-4b20-b5de-024705497d4a",  
3   "name": "Sam Raiden",  
4   "age": 0,  
5   "position": "Software developer"  
6 }
```

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

<https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBKA8-664D-4B20-B5DE-024705497D4A>

The screenshot shows a POSTMAN interface. The URL is <https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBKA8-664D-4B20-B5DE-024705497D4A>. The method is PATCH. The body is a JSON patch document:

```
1 {  
2   "op": "add",  
3   "path": "/age",  
4   "value": "28"  
5 }
```

The response status is 204 No Content, with a time of 26 ms and a size of 81 B. There are 2 headers listed.

Let's check the employee now:



```
1 <v {  
2     "id": "80abbca8-664d-4b20-b5de-024705497d4a",  
3     "name": "Sam Raiden",  
4     "age": 28,  
5     "position": "Software developer"  
6 }
```

Excellent.

Everything works as expected.



13 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.

13.1 ModelState, Rerun Validation, and Built-in Attributes

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.



It is important to know that model validation occurs after model binding and reports errors where the data, sent from the client, doesn't meet our validation criteria. Both model validation and data binding occur before our request reaches an action inside a controller. We are going to use the **ModelState.IsValid** expression to check for those validation rules.

By default, we don't have to use the **ModelState.IsValid** expression in Web API projects since, as we explained in section 9.2.1, controllers are decorated with the **[ApiController]** attribute. But, as we could've seen, it defaults all the model state errors to **400 - BadRequest** and doesn't allow us to return our custom error messages with a different status code. So, we suppressed it in the **Program** class.

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. If we didn't suppress the model validation from the **[ApiController]** attribute, we wouldn't be able to return this status code (422) since, as we said, it would default to 400.

13.1.1 Rerun Validation

In some cases, we want to repeat our validation. This can happen if, after the initial validation, we compute a value in our code, and assign it to the property of an already validated object.

If this is the case, and we want to run the validation again, we can use the **ModelStateDictionary.ClearValidationState** method to clear the validation specific to the model that we've already validated, and then use the **TryValidateModel** method:

```
[HttpPost]
public IActionResult POST([FromBody] Book book)
{
    if (!ModelState.IsValid)
        return UnprocessableEntity(ModelState);
```



```
var newPrice = book.Price - 10;
book.Price = newPrice;

ModelState.ClearValidationState(nameof(Book));
if (!TryValidateModel(book, nameof(Book)))
    return UnprocessableEntity(ModelState);

_service.CreateBook(book);

return CreatedAtRoute("BookById", new { id = book.Id }, book);
}
```

This is just a simple example but it explains how we can revalidate our model object.

13.1.2 Built-in Attributes

Validation attributes let us specify validation rules for model properties. At the beginning of this chapter, we have marked some validation attributes. Those attributes (Required and MaxLength) are part of built-in attributes. And of course, there are more than two built-in attributes. These are the most used ones:

ATTRIBUTE	USAGE
[ValidateNever]	Indicates that property or parameter should be excluded from validation.
[Compare]	We use it for the properties comparison.
[EmailAddress]	Validates the email format of the property.
[Phone]	Validates the phone format of the property.
[Range]	Validates that the property falls within a specified range.
[RegularExpression]	Validates that the property value matches a specified regular expression.
[Required]	We use it to prevent a null value for the property.
[StringLength]	Validates that a string property value doesn't exceed a specified length limit.

If you want to see a complete list of built-in attributes, you can visit [this page](#).



13.2 Custom Attributes and IValidableObject

There are scenarios where built-in attributes are not enough and we have to provide some custom logic. For that, we can create a custom attribute by using the **ValidationAttribute** class, or we can use the **IValidableObject** interface.

So, let's see an example of how we can create a custom attribute:

```
public class ScienceBookAttribute : ValidationAttribute
{
    public BookGenre Genre { get; set; }
    public string Error => $"The genre of the book must be {BookGenre.Science}";

    public ScienceBookAttribute(BookGenre genre)
    {
        Genre= genre;
    }

    protected override ValidationResult? IsValid(object? value, ValidationContext
validationContext)
    {
        var book = (Book)validationContext.ObjectInstance;

        if (!book.Genre.Equals(Genre.ToString()))
            return new ValidationResult(Error);

        return ValidationResult.Success;
    }
}
```

Once this attribute is called, we are going to pass the genre parameter inside the constructor. Then, we have to override the **IsValid** method. There we extract the object we want to validate and inspect if the **Genre** property matches our value sent through the constructor. If it's not we return the **Error** property as a validation result. Otherwise, we return success.

To call this custom attribute, we can do something like this:

```
public class Book
{
    public int Id { get; set; }

    [Required]
    public string? Name { get; set; }

    [Range(10, int.MaxValue)]
    public int Price { get; set; }
```



```
[ScienceBook(BookGenre.Science)]
public string? Genre { get; set; }
}
```

Now we can use the **IValidatableObject** interface:

```
public class Book : IValidatableObject
{
    public int Id { get; set; }

    [Required]
    public string? Name { get; set; }

    [Range(10, int.MaxValue)]
    public int Price { get; set; }

    public string? Genre { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
    {
        var errorMessage = $"The genre of the book must be {BookGenre.Science}";
        if (!Genre.Equals(BookGenre.Science.ToString()))
            yield return new ValidationResult(errorMessage, new[] {
nameof(Genre) });
    }
}
```

This validation happens in the model class, where we have to implement the **Validate** method. The code inside that method is pretty straightforward. Also, pay attention that we don't have to apply any validation attribute on top of the **Genre** property.

As we've seen from the previous examples, we can create a custom attribute in a separate class and even make it generic so it could be reused for other model objects. This is not the case with the **IValidatableObject** interface. It is used inside the model class and of course, the validation logic can't be reused.

So, this could be something you can think about when deciding which one to use.

After all of this theory and code samples, we are ready to implement model validation in our code.



13.3 Validation while Creating Resource

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

<https://localhost:5001/api/companies/582ea192-6fb7-44ff-a2a1-08d988ca3ca9/employees>

The screenshot shows a Postman request configuration for a POST request to the specified URL. The 'Body' tab is selected, containing the following JSON:

```
1 "name": null,
2 "age": 29,
3 "position": null
```

The response status is 500 Internal Server Error, with a duration of 1386 ms and a size of 330 B. The response body is:

```
1 "StatusCode": 500,
2 "Message": "An error occurred while updating the entries. See the inner exception for details."
```

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 the wrong model to the API — thus the error message should be different.

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

```
public record EmployeeForCreationDto(
    [Required(ErrorMessage = "Employee name is a required field.")]
    [MaxLength(30, ErrorMessage = "Maximum length for the Name is 30 characters.")]
    string Name,
    [Required(ErrorMessage = "Age is a required field.")]
    int Age,
    [Required(ErrorMessage = "Position is a required field.")]
    [MaxLength(20, ErrorMessage = "Maximum length for the Position is 20 characters.")]
    string Position
);
```

This is how we can apply validation attributes in our positional records. But, in our opinion, positional records start losing readability once the



attributes are applied, and for that reason, we like using init setters if we have to apply validation attributes. So, we are going to do exactly that and modify this position record:

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

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

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

Now, we have to modify our action:

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

    if (!ModelState.IsValid)
        return UnprocessableEntity(ModelState);

    var employeeToReturn =
_service.EmployeeService.CreateEmployeeForCompany(companyId, employee,
trackChanges: false);

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

As mentioned before in the part about the **ModelState** dictionary, all we have to do is to call the **IsValid** method and return the **UnprocessableEntity** response by providing our **ModelState**.

And that is all.

Let's send our request one more time:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies/582ea192-6fb7-44ff-a2a1-08d988ca3ca9/employees>

POST https://localhost:5001/api/companies/582ea192-6fb7-44ff-a2a1-08d988ca3ca9/employees

Params Authorization Headers (10) Body **JSON** Pre-request Script Tests Settings Cookies Beautify

```
1 "name": null,
2 "age": 29,
3 "position": null
```

Body Cookies Headers (4) Test Results 422 Unprocessable Entity 19 ms 258 B Save Response

Pretty Raw Preview Visualize JSON

```
1
2 "Name": [
3     "Employee name is a required field."
4 ],
5 "Position": [
6     "Position is a required field."
7 ]
```

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

<https://localhost:5001/api/companies/582ea192-6fb7-44ff-a2a1-08d988ca3ca9/employees>

POST https://localhost:5001/api/companies/582ea192-6fb7-44ff-a2a1-08d988ca3ca9/employees

Params Authorization Headers (10) Body **JSON** Pre-request Script Tests Settings Cookies Beautify

```
1 "name": "Michael Patel",
2 "age": 29,
3 "position": "Some position with invalid length"
```

Body Cookies Headers (4) Test Results 422 Unprocessable Entity 38 ms 232 B Save Response

Pretty Raw Preview Visualize JSON

```
1
2 "Position": [
3     "Maximum length for the Position is 20 characters."
4 ]
```

Excellent. It works 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.



13.3.1 Validating Int Type

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

<https://localhost:5001/api/companies/582ea192-6fb7-44ff-a2a1-08d988ca3ca9/employees>

The screenshot shows a Postman request to `https://localhost:5001/api/companies/582ea192-6fb7-44ff-a2a1-08d988ca3ca9/employees`. The request method is POST. The request body is JSON with the following content:

```
1 {  
2   "name": null,  
3   "position": "Some position with invalid length"  
4 }
```

The response status is 422 Unprocessable Entity, with a time of 48 ms and a size of 278 B. The response body shows validation errors:

```
1 {  
2   "Name": [  
3     "Employee name is a required field."  
4   ],  
5   "Position": [  
6     "Maximum length for the Position is 20 characters."  
7   ]  
8 }
```

We can 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.

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:



<https://localhost:5001/api/companies/582ea192-6fb7-44ff-a2a1-08d988ca3ca9/employees>

The screenshot shows a POST request to <https://localhost:5001/api/companies/582ea192-6fb7-44ff-a2a1-08d988ca3ca9/employees>. The request body is:

```
1 {  
2   "name": null,  
3   "position": "Some position with invalid length"  
4 }
```

The response status is 422 Unprocessable Entity with a time of 819 ms and a size of 334 B. The response body is:

```
1 {"Age": [  
2     "Age is required and it can't be lower than 18"  
3 ],  
4   "Name": [  
5     "Employee name is a required field."  
6 ],  
7   "Position": [  
8     "Maximum length for the Position is 20 characters."  
9   ]  
10 }  
11 }
```

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.

13.4 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** record:

```
public record EmployeeForUpdateDto  
{  
    [Required(ErrorMessage = "Employee name is a required field.")]
```



```
[MaxLength(30, ErrorMessage = "Maximum length for the Name is 30 characters.")]
public string Name? { get; init; }

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

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

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 record in the **DataTransferObjects** folder:

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

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

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

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

```
public record EmployeeForCreationDto : EmployeeForManipulationDto;
public record EmployeeForUpdateDto : EmployeeForManipulationDto;
```

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

```
if (employee is null)
    return BadRequest("EmployeeForUpdateDto object is null");

if (!ModelState.IsValid)
    return UnprocessableEntity(ModelState);
```



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

Let's test this:

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

The screenshot shows a POST request in Postman to the URL `https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBCA8-664D-4B20-B5DE-024705497D4A`. The request body is a JSON object with three fields: "name": null, "age": 29, and "position": null. The response status is 422 Unprocessable Entity, indicating validation errors. The response body shows validation errors for both "Name" and "Position" fields, stating they are required.

Great.

Everything works well.

13.5 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);
```

But once we do this, we are going to get an error. That's because the current **ApplyTo** method comes from the `JsonPatch` namespace, and we need the method with the same name but from the `NewtonsoftJson` namespace.



Since we have the **Microsoft.AspNetCore.Mvc.NewtonsoftJson** package installed in the main project, we are going to remove it from there and install it in the **Presentation** project.

If we navigate to the **ApplyTo** method declaration we can find two extension methods:

```
public static class JsonPatchExtensions
{
    public static void ApplyTo<T>(this JsonPatchDocument<T> patchDoc, T
objectToApplyTo, ModelStateDictionary ModelState) where T : class...
    public static void ApplyTo<T>(this JsonPatchDocument<T> patchDoc, T
objectToApplyTo, ModelStateDictionary ModelState, string prefix) where T : class...
}
```

We are using the first one.

After the package installation, the error in the action will disappear.

Now, right below the **ApplyTo** method, we can add our familiar validation logic:

```
patchDoc.ApplyTo(result.employeeToPatch, ModelState);
if (!ModelState.IsValid)
    return UnprocessableEntity(ModelState);
_service.EmployeeService.SaveChangesForPatch(...);
```

Let's test this now:



Ultimate ASP.NET Core Web API

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

The screenshot shows a Postman request configuration. The method is PATCH, the URL is <https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBCA8-664D-4B20-B5DE-024705497D4A>, and the body is:

```
1
2 → {
3   →   "op": "remove",
4   →   "path": "/ageeeeeee"
5 }
```

The response status is 422 Unprocessable Entity with the message: "The target location specified by path segment 'ageeeeeee' was not found."

You can see that it works as it is supposed to.

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

The screenshot shows a Postman request configuration. The method is PATCH, the URL is <https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBCA8-664D-4B20-B5DE-024705497D4A>, and the body is:

```
1 [
2   → [
3     → {
4       →   "op": "remove",
5       →   "path": "/age"
6     }
7   ]
8 ]
```

The response status is 204 No Content.

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 that doesn't allow that value to be below 18. So, where is the problem?

Let's illustrate this for you:



```
patchDoc.ApplyTo(result.employeeToPatch, ModelState);
if (!ModelState.IsValid) // We validate patchDoc
    return UnprocessableEntity(ModelState);
_service.EmployeeService.SaveChangesForPatch(result.employeeToPatch, result.employeeEntity); // In this service method, we map to employeeEntity and save it to the db
return NoContent();
```

As you can see, we are validating **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:

```
patchDoc.ApplyTo(result.employeeToPatch, ModelState);
TryValidateModel(result.employeeToPatch);
if (!ModelState.IsValid)
    return UnprocessableEntity(ModelState);
```

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

Now, we can test this again:



Ultimate ASP.NET Core Web API

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

The screenshot shows the Postman application interface. A PATCH request is being sent to the URL <https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees/80ABBCA8-664D-4B20-B5DE-024705497D4A>. The request body is a JSON object with the following structure:

```
1 [  
2   ...  
3   ... "op": "remove",  
4   ... "path": "/age"  
5 ]
```

The response status is 422 Unprocessable Entity. The response body is:

```
1 {"Age": [  
2     "Age is required and it can't be lower than 18"  
3 ]}
```

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



14 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.

14.1 What is Asynchronous Programming?

Async programming is a parallel programming technique that allows the working process to run separately from the main application thread.

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

How so?

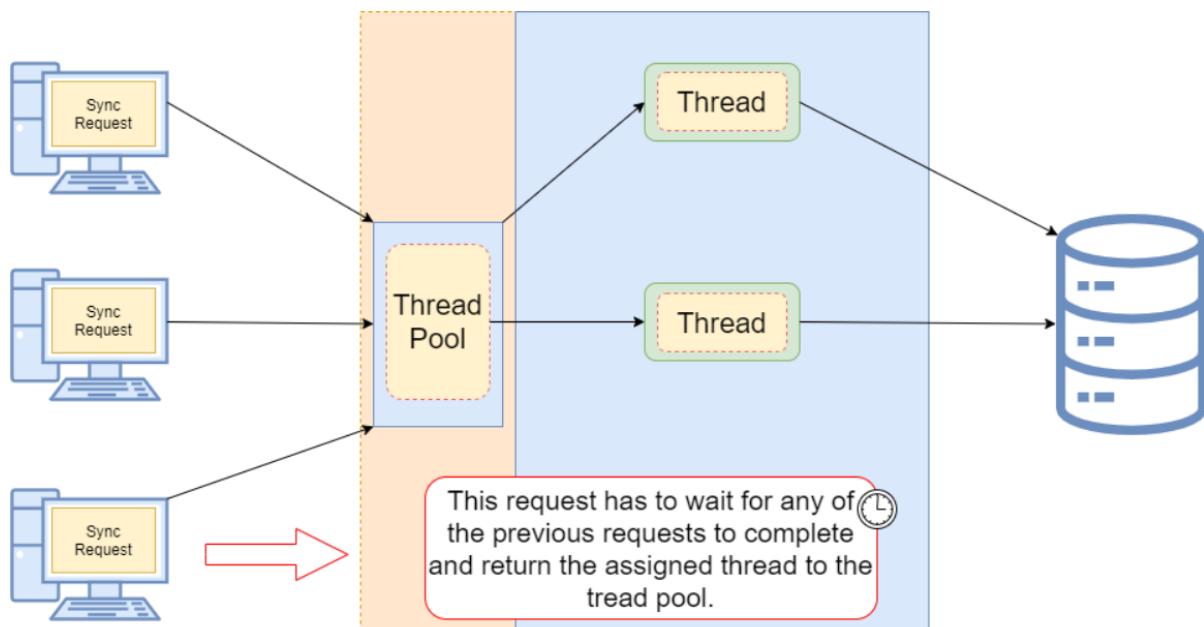
Because we are not sending requests to the server and blocking it while waiting for the responses anymore (as long as it takes). Now, when we send a request to the server, the thread pool delegates a thread to that request. Eventually, that thread finishes its job and returns to the thread pool freeing itself for the next request. At some point, the data will be fetched from the database and the result needs to be sent to the requester. At that time, the thread pool provides another thread to handle that work. Once the work is done, a thread is going back to the thread pool.

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.

Let's imagine that our thread pool has two threads and we have used one thread with a first request. Now, the second request arrives and we have to use the second thread from a thread pool. At this point, our thread pool is out of threads. If a third request arrives now it has to wait for any of the first two requests to complete and return assigned threads to a thread pool. Only then the thread pool can assign that returned thread to a new request:



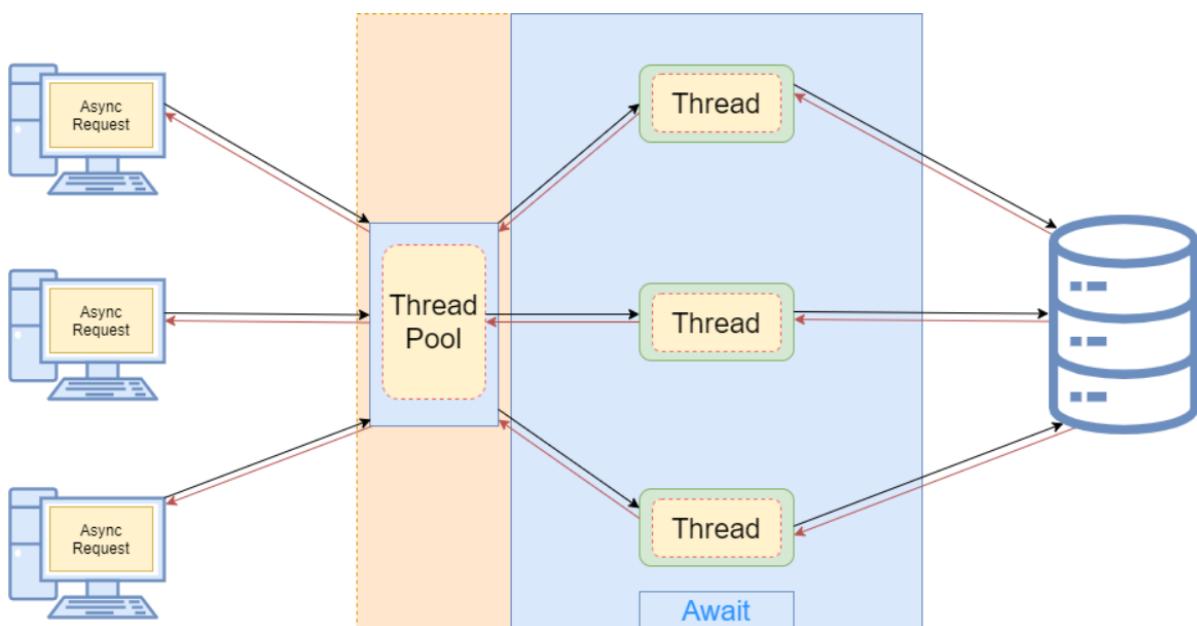
As a result of a request waiting for an available thread, our client experiences a slow down for sure. Additionally, if the client has to wait too long, they will receive an error response usually the service is unavailable (503). But this is not the only problem. Since the client expects the list of entities from the database, we know that it is an I/O operation. So, if we have a lot of records in the database and it takes three seconds for the database to return a result to the API, our thread is doing nothing except waiting for the task to complete. So basically, we are blocking that thread



and making it three seconds unavailable for any additional requests that arrive at our API.

With asynchronous requests, the situation is completely different.

When a request arrives at our API, we still need a thread from a thread pool. So, that leaves us with only one thread left. But because this action is now asynchronous, as soon as our request reaches the I/O point where the database has to process the result for three seconds, the thread is returned to a thread pool. Now we again have two available threads and we can use them for any additional request. After the three seconds when the database returns the result to the API, the thread pool assigns the thread again to handle that response:



Now that we've cleared that out, we can learn how to implement asynchronous code in .NET Core and .NET 5+.

14.2 Async, Await Keywords, and Return Types

The **async** and **await** keywords play a crucial part in asynchronous programming. We use the **async** keyword in the method declaration and its purpose is to enable the **await** keyword within that method. So yes,



we can't use the **await** keyword without previously adding the **async** keyword in the method declaration. Also, using only the **async** keyword doesn't make your method asynchronous, just the opposite, that method is still synchronous.

The **await** keyword performs an asynchronous wait on its argument. It does that in several steps. The first thing it does is to check whether the operation is already complete. If it is, it will continue the method execution synchronously. Otherwise, the **await** keyword is going to pause the **async** method execution and return an incomplete task. Once the operation completes, a few seconds later, the **async** method can continue with the execution.

Let's see this with a simple example:

```
public async Task<IEnumerable<Company>> GetCompanies()
{
    _logger.LogInfo("Inside the GetCompanies method.");
    var companies = await _repoContext.Companies.ToListAsync();
    return companies;
}
```

So, even though our method is marked with the **async** keyword, it will start its execution synchronously. Once we log the required information synchronously, we continue to the next code line. We extract all the companies from the database and to do that, we use the **await** keyword. If our database requires some time to process the result and return it, the **await** keyword is going to pause the **GetCompanies** method execution and return an incomplete task. During that time the thread will be returned to a thread pool making itself available for another request. After the database operation completes the **async** method will resume executing and will return the list of companies.

From this example, we see the **async** method execution flow. But the question is how the **await** keyword knows if the operation is completed or not. Well, this is where **Task** comes into play.



14.2.1 Return Types of the Asynchronous Methods

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.

It is very important to understand that the `Task` represents an execution of the asynchronous method and not the result. The `Task` has several properties that indicate whether the operation was completed successfully or not (`Status`, `IsCompleted`, `IsCanceled`, `IsFaulted`). With these properties, we can track the flow of our `async` operations. So, this is the answer to our question. With `Task`, we can track whether the operation is completed or not. This is also called TAP (Task-based Asynchronous Pattern).



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

14.2.2 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 runs slower, we should switch back to the sync one.

14.3 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 void CreateCompany(Company company) => Create(company);

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

public void DeleteCompany(Company company) => Delete(company);
```

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

14.4 IRepositoryManager and RepositoryManager Changes

If we inspect the mentioned interface and the class, we will see the **Save** method, which 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 the **RepositoryManager** class modification:

```
public async Task SaveAsync() => await _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.

14.5 Updating the Service layer

Again, we have to start with the interface modification:

```
public interface ICompanyService
{
    Task<IEnumerable<CompanyDto>> GetAllCompaniesAsync(bool trackChanges);
    Task<CompanyDto> GetCompanyAsync(Guid companyId, bool trackChanges);
    Task<CompanyDto> CreateCompanyAsync(CompanyForCreationDto company);
    Task<IEnumerable<CompanyDto>> GetByIdsAsync(IEnumerable<Guid> ids, bool
trackChanges);
    Task<(IEnumerable<CompanyDto> companies, string ids)>
CreateCompanyCollectionAsync
        (IEnumerable<CompanyForCreationDto> companyCollection);
    Task DeleteCompanyAsync(Guid companyId, bool trackChanges);
    Task UpdateCompanyAsync(Guid companyid, CompanyForUpdateDto companyForUpdate,
bool trackChanges);
}
```

And then, let's modify the class methods one by one.

GetAllCompanies:

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

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

    return companiesDto;
}
```

GetCompany:

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

    var companyDto = _mapper.Map<CompanyDto>(company);
    return companyDto;
}
```

CreateCompany:

```
public async Task<CompanyDto> CreateCompanyAsync(CompanyForCreationDto company)
{
```



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

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

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

        return companyToReturn;
    }
```

GetByIds:

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

    var companyEntities = await _repository.Company.GetByIdsAsync(ids, trackChanges);
    if (ids.Count() != companyEntities.Count())
        throw new CollectionByIdsBadRequestException();

    var companiesToReturn = _mapper.Map<IEnumerable<CompanyDto>>(companyEntities);

    return companiesToReturn;
}
```

CreateCompanyCollection:

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

    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 (companies: companyCollectionToReturn, ids: ids);
}
```

DeleteCompany:

```
public async Task DeleteCompanyAsync(Guid companyId, bool trackChanges)
{
```



```
var company = await _repository.Company.GetCompanyAsync(companyId,
trackChanges);
if (company is null)
    throw new CompanyNotFoundException(companyId);

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

UpdateCompany:

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

    _mapper.Map(companyForUpdate, companyEntity);
    await _repository.SaveAsync();
}
```

That's all the changes we have to make in the **CompanyService** class.

Now we can move on to the controller modification.

14.6 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
_service.CompanyService.GetAllCompaniesAsync(trackChanges: false);

    return Ok(companies);
}
```



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.

NOTE: We've changed all the method names in the repository and service layers by adding the `Async` suffix. But, we didn't do that in the controller's action. The main reason for that is when a user calls a method from your service or repository layers they can see right-away from the method name whether the method is synchronous or asynchronous. Also, your layers are not limited only to sync or async methods, you can have two methods that do the same thing but one in a sync manner and another in an async manner. In that case, you want to have a name distinction between those methods. For the controller's actions this is not the case. We are not targeting our actions by their names but by their routes. So, the name of the action doesn't really add any value as it does for the method names.

So to continue, let's modify all the other actions.

GetCompany:

```
[HttpGet("{id:guid}", Name = "CompanyById")]
public async Task<IActionResult> GetCompany(Guid id)
{
    var company = await _service.CompanyService.GetCompanyAsync(id, trackChanges:
false);
    return Ok(company);
}
```

GetCompanyCollection:

```
[HttpGet("collection/({ids})", Name = "CompanyCollection")]
public async Task<IActionResult> GetCompanyCollection(
    [ModelBinder(BinderType = typeof(ArrayModelBinder))] IEnumerable<Guid> ids)
{
    var companies = await _service.CompanyService.GetByIdsAsync(ids, trackChanges:
false);

    return Ok(companies);
}
```

CreateCompany:

```
[HttpPost]
```



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

    if (!ModelState.IsValid)
        return UnprocessableEntity(ModelState);

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

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

CreateCompanyCollection:

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

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

DeleteCompany:

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

    return NoContent();
}
```

UpdateCompany:

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

    await _service.CompanyService.UpdateCompanyAsync(id, company, trackChanges:
true);

    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 the 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.

14.7 Continuation in Asynchronous Programming

The `await` keyword does three things:

- It helps us extract the result from the async operation – we already learned about that
- Validates the success of the operation
- Provides the Continuation for executing the rest of the code in the async method

So, in our `GetCompanyAsync` service method, all the code after awaiting an async operation is executed inside the continuation if the async operation was successful.

When we talk about continuation, it can be confusing because you can read in multiple resources about the `SynchronizationContext` and capturing the current context to enable this continuation. When we await a task, a request context is captured when `await` decides to pause the method execution. Once the method is ready to resume its execution, the application takes a thread from a thread pool, assigns it to the context (`SynchronizationContext`), and resumes the execution. But this is the case for ASP.NET applications.



We don't have the `SynchronizationContext` in ASP.NET Core applications. ASP.NET Core avoids capturing and queuing the context, all it does is take the thread from a thread pool and assign it to the request. So, a lot less background works for the application to do.

One more thing. We are not limited to a single continuation. This means that in a single method, we can use multiple `await` keywords.

14.8 Common Pitfalls

In our `GetAllCompaniesAsync` repository method if we didn't know any better, we could've been tempted to use the `Result` property instead of the `await` keyword:

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

We can see that the `Result` property returns the result we require:

```
// Summary:
//     Gets the result value of this System.Threading.Tasks.Task`1.
//
// Returns:
//     The result value of this System.Threading.Tasks.Task`1, which
//     is of the same type as the task's type parameter.

public TResult Result
{
    get...
}
```

But don't use the `Result` property.

With this code, we are going to block the thread and potentially cause a deadlock in the application, which is the exact thing we are trying to avoid using the `async` and `await` keywords. It applies the same to the `Wait` method that we can call on a `Task`.



So, that's it regarding the asynchronous implementation in our project. We've learned a lot of useful things from this section and we can move on to the next one – Action filters.



15 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 that 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 a cleaner and reusable code in our Web API.

15.1 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
        }
    }
}
```

15.2 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 **Program** class:

```
builder.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, but as a service in the IoC container:

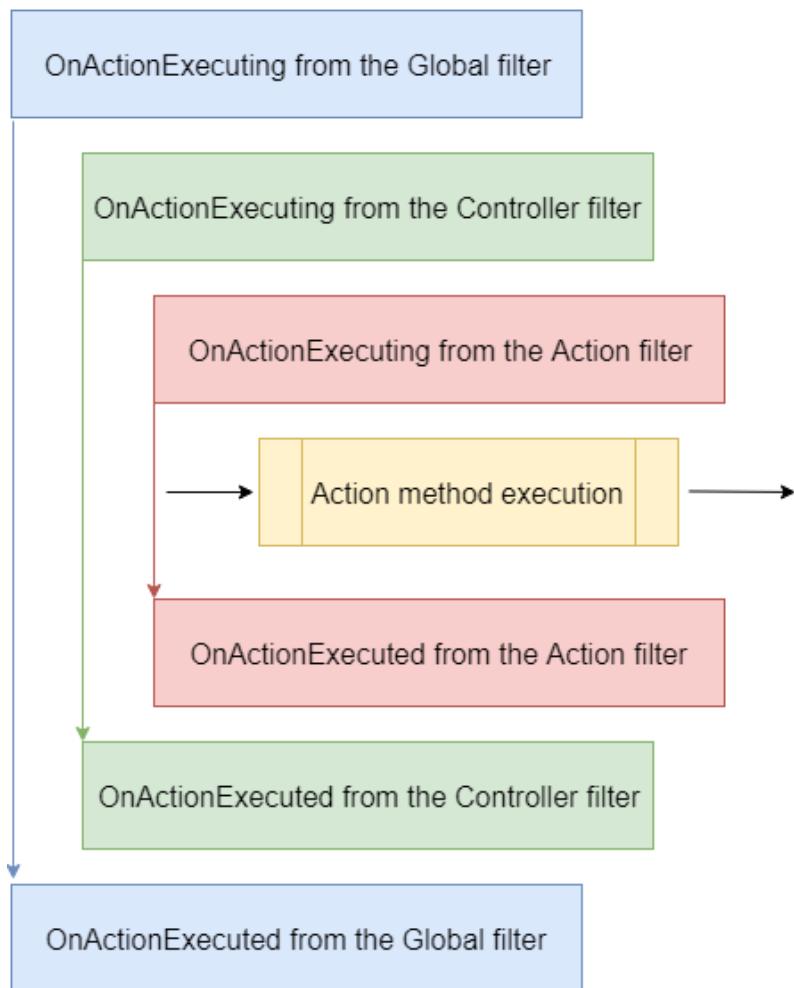
```
builder.Services.AddScoped<ActionFilterExample>();
builder.Services.AddScoped<ControllerFilterExample>();
```

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" };
        }
    }
}
```

15.3 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" };
}
```

15.4 Improving the Code with Action Filters

Our actions are clean and readable without **try-catch** blocks due to global exception handling and a service layer implementation, but we can improve them even further.

So, let's start with the validation code from the POST and PUT actions.

15.5 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 is null)
    return BadRequest("CompanyForUpdateDto object is null");

if (!ModelState.IsValid)
    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
{
    public ValidationFilterAttribute()
    {}

    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 is null)
    {
        context.Result = new BadRequestObjectResult($"Object is null. Controller: {controller}, action: {action}");
        return;
    }

    if (!context.ModelState.IsValid)
        context.Result = new UnprocessableEntityObjectResult(context.ModelState);
}
```

We are using the `context` parameter to retrieve different values that we need inside this method. With the `RouteData.Values` dictionary, we can get the values produced by routes on the current routing path. Since we need the name of the action and the controller, we extract them from the `Values` dictionary.

Additionally, we use the `ActionArguments` dictionary to extract the DTO parameter that we send to the POST and PUT actions. If that parameter is null, we set the `Result` property of the `context` object to a new instance of the `BadRequestObjectReturnResult` class. If the model is invalid, we create a new instance of the `UnprocessableEntityObjectResult` class and pass `ModelState`.

Next, let's register this action filter in the `Program` class above the `AddControllers` method:

```
builder.Services.AddScoped<ValidationFilterAttribute>();
```

Finally, let's remove the mentioned validation code from our actions and call this action filter as a service.

POST:

```
[HttpPost]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> CreateCompany([FromBody] CompanyForCreationDto company)
```



Ultimate ASP.NET Core Web API

```
var createdCompany = await _service.CompanyService.CreateCompanyAsync(company);

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

PUT:

```
[HttpPut("{id:guid}")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> UpdateCompany(Guid id, [FromBody] CompanyForUpdateDto
company)
{
    await _service.CompanyService.UpdateCompanyAsync(id, company, trackChanges:
true);

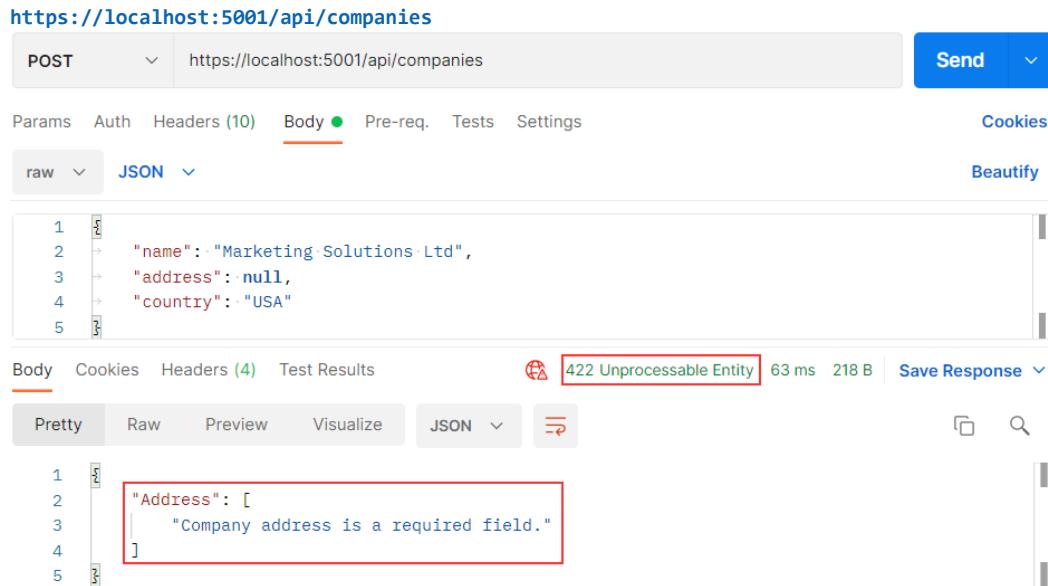
    return NoContent();
}
```

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:

<https://localhost:5001/api/companies>



The screenshot shows a Postman request to `https://localhost:5001/api/companies` via POST. The body is set to JSON and contains the following invalid data:

```
1 "name": "Marketing Solutions Ltd",
2 "address": null,
3 "country": "USA"
```

The response status is `422 Unprocessable Entity`. The error message in the response body is:

```
1 "Address": [
2     "Company address is a required field."
3 ]
```

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:



The screenshot shows a POST request to `https://localhost:5001/api/companies/53a1237a-3ed3-4462-b9f0-5a7bb1056a33/employees`. The request body is JSON:

```
1 {  
2   "name": "Martin Geil",  
3   "age": 0,  
4   "position": "Marketing expert"  
5 }
```

The response status is **422 Unprocessable Entity**, with a message: `"Age": ["Age is required and it can't be lower than 18"]`.

15.6 Refactoring the Service Layer

Because we are already working on making our code reusable in our actions, we can review our classes from the service layer.

Let's inspect the **CompanyService** class first.

Inside the class, we can find three methods (`GetCompanyAsync`, `DeleteCompanyAsync`, and `UpdateCompanyAsync`) where we repeat the same code:

```
var company = await _repository.Company.GetCompanyAsync(id, trackChanges);  
if (company is null)  
    throw new CompanyNotFoundException(id);
```

This is something we can extract in a private method in the same class:

```
private async Task<Company> GetCompanyAndCheckIfItExists(Guid id, bool trackChanges)  
{  
    var company = await _repository.Company.GetCompanyAsync(id, trackChanges);  
    if (company is null)  
        throw new CompanyNotFoundException(id);  
    return company;  
}
```

And then we can modify these methods.

GetCompanyAsync:



```
public async Task<CompanyDto> GetCompanyAsync(Guid id, bool trackChanges)
{
    var company = await GetCompanyAndCheckIfExists(id, trackChanges);

    var companyDto = _mapper.Map<CompanyDto>(company);
    return companyDto;
}
```

DeleteCompanyAsync:

```
public async Task DeleteCompanyAsync(Guid companyId, bool trackChanges)
{
    var company = await GetCompanyAndCheckIfExists(companyId, trackChanges);

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

UpdateCompanyAsync:

```
public async Task UpdateCompanyAsync(Guid companyId,
                                      CompanyForUpdateDto companyForUpdate, bool trackChanges)
{
    var company = await GetCompanyAndCheckIfExists(companyId, trackChanges);

    _mapper.Map(companyForUpdate, company);
    await _repository.SaveAsync();
}
```

Now, this looks much better without code repetition.

Furthermore, we can find code repetition in almost all the methods inside the **EmployeeService** class:

```
var company = await _repository.Company.GetCompanyAsync(companyId, trackChanges);
if (company is null)
    throw new CompanyNotFoundException(companyId);

var employeeDb = await _repository.Employee.GetEmployeeAsync(companyId, id,
trackChanges);
if (employeeDb is null)
    throw new EmployeeNotFoundException(id);
```

In some methods, we can find just the first check and in several others, we can find both of them.

So, let's extract these checks into two separate methods:

```
private async Task CheckIfCompanyExists(Guid companyId, bool trackChanges)
{
    var company = await _repository.Company.GetCompanyAsync(companyId,
trackChanges);
    if (company is null)
```



```
        throw new CompanyNotFoundException(companyId);
    }

private async Task<Employee> GetEmployeeForCompanyAndCheckIfExists
    (Guid companyId, Guid id, bool trackChanges)
{
    var employeeDb = await _repository.Employee.GetEmployeeAsync(companyId, id,
trackChanges);
    if (employeeDb is null)
        throw new EmployeeNotFoundException(id);

    return employeeDb;
}
```

With these two extracted methods in place, we can refactor all the other methods in the class.

GetEmployeesAsync:

```
public async Task<IEnumerable<EmployeeDto>> GetEmployeesAsync(Guid companyId, bool
trackChanges)
{
    await CheckIfCompanyExists(companyId, trackChanges);

    var employeesFromDb = await _repository.Employee.GetEmployeesAsync(companyId,
trackChanges);
    var employeesDto = _mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

    return employeesDto;
}
```

GetEmployeeAsync:

```
public async Task<EmployeeDto> GetEmployeeAsync(Guid companyId, Guid id, bool
trackChanges)
{
    await CheckIfCompanyExists(companyId, trackChanges);

    var employeeDb = await GetEmployeeForCompanyAndCheckIfExists(companyId, id,
trackChanges);

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

CreateEmployeeForCompanyAsync:

```
public async Task<EmployeeDto> CreateEmployeeForCompanyAsync(Guid companyId,
EmployeeForCreationDto employeeForCreation, bool trackChanges)
{
    await CheckIfCompanyExists(companyId, trackChanges);

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

    _repository.Employee.CreateEmployeeForCompany(companyId, employeeEntity);
    await _repository.SaveAsync();
```



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

        return employeeToReturn;
    }
```

DeleteEmployeeForCompanyAsync:

```
public async Task DeleteEmployeeForCompanyAsync(Guid companyId, Guid id, bool
trackChanges)
{
    await CheckIfCompanyExists(companyId, trackChanges);

    var employeeDb = await GetEmployeeForCompanyAndCheckIfItExists(companyId, id,
trackChanges);

    _repository.Employee.DeleteEmployee(employeeDb);
    await _repository.SaveChangesAsync();
}
```

UpdateEmployeeForCompanyAsync:

```
public async Task UpdateEmployeeForCompanyAsync(Guid companyId, Guid id,
EmployeeForUpdateDto employeeForUpdate,
bool compTrackChanges, bool empTrackChanges)
{
    await CheckIfCompanyExists(companyId, compTrackChanges);

    var employeeDb = await GetEmployeeForCompanyAndCheckIfItExists(companyId, id,
empTrackChanges);

    _mapper.Map(employeeForUpdate, employeeDb);
    await _repository.SaveChangesAsync();
}
```

GetEmployeeForPatchAsync:

```
public async Task<(EmployeeForUpdateDto employeeToPatch, Employee employeeEntity)>
GetEmployeeForPatchAsync
    (Guid companyId, Guid id, bool compTrackChanges, bool empTrackChanges)
{
    await CheckIfCompanyExists(companyId, compTrackChanges);

    var employeeDb = await GetEmployeeForCompanyAndCheckIfItExists(companyId, id,
empTrackChanges);

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

    return (employeeToPatch: employeeToPatch, employeeEntity: employeeDb);
}
```

Now, all of the methods are cleaner and easier to maintain since our validation code is in a single place, and if we need to modify these validations, there's only one place we need to change.



Additionally, if you want you can create a new class and extract these methods, register that class as a service, inject it into our service classes and use the validation methods. It is up to you how you want to do it.

So, we have seen how to use action filters to clear our action methods and also how to extract methods to make our service cleaner and easier to maintain.

With that out of the way, we can continue to Paging.



16 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.

16.1 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 would that 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.



16.2 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 employees = await _service.EmployeeService.GetEmployeesAsync(companyId,
trackChanges: false);
    return Ok(employees);
}
```

A few things to take note of here:

- 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 **Shared** project and then inside, create the required classes.

First the **RequestParameters** class:

```
public abstract class RequestParameters
```



```
{  
    const int maxPageSize = 50;  
    public int PageNumber { get; set; } = 1;  
  
    private int _pageSize = 10;  
    public int PageSize  
    {  
        get  
        {  
            return _pageSize;  
        }  
        set  
        {  
            _pageSize = (value > maxPageSize) ? maxPageSize : value;  
        }  
    }  
}
```

And then the **EmployeeParameters** class:

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

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 Shared.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);
}
```

As Visual Studio suggests, we have to add the reference to the **Shared** project.

After that, let's modify 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?

Since we call this repository method in our service layer, we have to modify it as well.

So, let's start with the **IEmployeeService** modification:

```
public interface IEmployeeService
{
    Task<IEnumerable<EmployeeDto>> GetEmployeesAsync(Guid companyId,
        EmployeeParameters employeeParameters, bool trackChanges);
    ...
}
```

In this interface, we only have to modify the **GetEmployeesAsync** method by adding a new parameter.

After that, let's modify the **EmployeeService** class:

```
public async Task<IEnumerable<EmployeeDto>> GetEmployeesAsync(Guid companyId,
    EmployeeParameters employeeParameters, bool trackChanges)
{
```



```
await CheckIfCompanyExists(companyId, trackChanges);

var employeesFromDb = await _repository.Employee
    .GetEmployeesAsync(companyId, employeeParameters, trackChanges);
var employeesDto = _mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

return employeesDto;
}
```

Nothing too complicated here. We just accept an additional parameter and pass it to the repository method.

Finally, we have to modify the **GetEmployeesForCompany** action and fix that error by adding another argument to the **GetEmployeesAsync** method call:

```
[HttpGet]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId,
    [FromQuery] EmployeeParameters employeeParameters)
{
    var employees = await _service.EmployeeService.GetEmployeesAsync(companyId,
        employeeParameters, trackChanges: false);

    return Ok(employees);
}
```

16.3 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": 32, "position": "Marketing expert II" }	{ "name": "Nina Hawk", "age": 26, "position": "Marketing expert II" }
{ "name": "Mihael Fins", "age": 30, "position": "Marketing expert" }	{ "name": "Martha Grown", "age": 35, "position": "Marketing expert" }	{ "name": "Kirk Metha", "age": 30, "position": "Marketing expert" }



Ultimate ASP.NET Core Web API

}	"position": "Marketing expert II" }	}
---	--	---

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:

Params ● Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (5) Test Results 200 OK 32 ms 474 B Save Response

Pretty Raw Preview Visualize JSON

```
1
2 {
3     "id": "af491bb5-b657-4431-f2c3-08d98e193cd3",
4     "name": "Kirk Metha",
5     "age": 30,
6     "position": "Marketing expert"
7 },
8 {
9     "id": "931bc1a1-ddea-4f09-f2c2-08d98e193cd3",
10    "name": "Martha Grown",
11    "age": 35,
12    "position": "Marketing expert II"
13 }
```

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	1
2	E36882C5-DDBF-4748-F2BF-08D98E193CD3	John Spike	32	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870	2
3	AF491BB5-B657-4431-F2C3-08D98E193CD3	Kirk Metha	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870	3
4	931BC1A1-DDEA-4F09-F2C2-08D98E193CD3	Martha Grown	35	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870	4
5	DB6D7CDC-9251-4E0A-F2C1-08D98E193CD3	Mihael Fins	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870	
6	ED8B6253-DE5F-48BC-F2BE-08D98E193CD3	Mihael Worth	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870	
7	5C8277BF-4A28-4ACC-F2C0-08D98E193CD3	Nina Hawk	26	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870	
8	80ABBCA8-664D-4B20-B5DE-024705497D4A	Sam Raiden	28	Software developer	C9D4C053-49B6-410C-BC78-2D54A9991870	

And we can see that we have the correct data returned.



Now, what can we do to improve this solution?

16.4 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 **Shared/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);
}
```

After that, we are going to modify the **IEmployeeService** interface:

```
Task<(IEnumerable<EmployeeDto> employees, MetaData metaData)> GetEmployeesAsync(Guid
companyId, EmployeeParameters employeeParameters, bool trackChanges);
```

Now our method returns a Tuple containing two fields – employees and metadata.



So, let's implement that in the **EmployeeService** class:

```
public async Task<IEnumerable<EmployeeDto> employees, MetaData metaData>
GetEmployeesAsync
    (Guid companyId, EmployeeParameters employeeParameters, bool trackChanges)
{
    await CheckIfCompanyExists(companyId, trackChanges);

    var employeesWithMetaData = await _repository.Employee
        .GetEmployeesAsync(companyId, employeeParameters, trackChanges);
    var employeesDto =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesWithMetaData);

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

We change the method signature and the name of the **employeesFromDb** variable to **employeesWithMetaData** since this name is now more suitable. After the mapping action, we construct a Tuple and return it to the caller.

Finally, let's modify the controller:

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

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

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

The new thing in this action is that we modify the response header and add our metadata as the X-Pagination header. For this, we need the **System.Text.Json** namespace.

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>

Body Cookies Headers (5) Test Results

🌐 200 OK

Pretty

Raw

Preview

Visualize

JSON



```
1
2   {
3     "id": "af491bb5-b657-4431-f2c3-08d98e193cd3",
4     "name": "Kirk Metha",
5     "age": 30,
6     "position": "Marketing expert"
7   },
8   {
9     "id": "931bc1a1-ddea-4f09-f2c2-08d98e193cd3",
10    "name": "Martha Grown",
11    "age": 35,
12    "position": "Marketing expert II"
13  }
14 ]
```

But now we have some additional useful information in the X-Pagination response header:

Body	Cookies	Headers (5)	Test Results	🌐 200 OK 32 ms 474 B Save Response
KEY	VALUE			
Content-Type ⓘ	application/json; charset=utf-8			
Date ⓘ	Thu, 14 Oct 2021 07:22:57 GMT			
Server ⓘ	Kestrel			
Transfer-Encoding ⓘ	chunked			
X-Pagination ⓘ	{"CurrentPage":2,"TotalPages":4,"PageSize":2,"TotalCount":8,"HasPrevious":true,"HasNext":true}			

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.

16.4.1 Additional Advice

This solution works great with a small amount of data, but with bigger tables with millions of rows, we can improve it by modifying the `GetEmployeesAsync` repository method:



```
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)
        .Skip((employeeParameters.PageNumber - 1) * employeeParameters.PageSize)
        .Take(employeeParameters.PageSize)
        .ToListAsync();

    var count = await FindByCondition(e => e.CompanyId.Equals(companyId), trackChanges)
        .CountAsync();

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

Even though we have an additional call to the database with the **CountAsync** method, this solution was tested upon millions of rows and was much faster than the previous one. Because our table has few rows, we will continue using the previous solution, but feel free to switch to this one if you want.

Also, to enable the client application to read the new **X-Pagination** header that we've added in our action, we have to modify the CORS configuration:

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



17 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.

17.1 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.

17.2 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.

17.3 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?mi  
nAge=26&maxAge=29
```

But, we want to be able to do this too:

```
https://localhost:5001/api/companies/companyId/employees?mi  
nAge=26
```

Or like this:

```
https://localhost:5001/api/companies/companyId/employees?ma  
xAge=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 than the 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 **GetEmployeesAsync** service method by adding a validation check as a first statement:

```
public async Task<IEnumerable<EmployeeDto>> GetEmployeesAsync(
    Guid companyId, EmployeeParameters employeeParameters, bool trackChanges)
{
    if (!employeeParameters.ValidAgeRange)
        throw new MaxAgeRangeBadRequestException();

    await CheckIfCompanyExists(companyId, trackChanges);

    var employeesWithMetaData = await _repository.Employee
        .GetEmployeesAsync(companyId, employeeParameters, trackChanges);
    var employeesDto =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesWithMetaData);

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



We've added our validation check and a BadRequest response if the validation fails.

But we don't have this custom exception class so, we have to create it in the **Entities/Exceptions** class:

```
public sealed class MaxAgeRangeBadRequestException : BadRequestException
{
    public MaxAgeRangeBadRequestException()
        :base("Max age can't be less than min age.")
    {
    }
}
```

That should do it.

After the service class modification and creation of our custom exception class, 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.

17.4 Sending and Testing a Query

Let's send a first request with only a MinAge parameter:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?minAge=32>

GET https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?minAge=32

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (5) Test Results 200 OK 129 ms 480 B Save Response

Pretty Raw Preview Visualize JSON

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
[{"id": "e36882c5-dbbf-4748-f2bf-08d98e193cd3", "name": "John Spike", "age": 32, "position": "Marketing expert II"}, {"id": "931bc1a1-ddea-4f09-f2c2-08d98e193cd3", "name": "Martha Grown", "age": 35, "position": "Marketing expert II"}]
```

Next, let's send one with only a MaxAge parameter:

<https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?maxAge=26>

GET https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?maxAge=26

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (5) Test Results 200 OK 95 ms 369 B Save Response

Pretty Raw Preview Visualize JSON

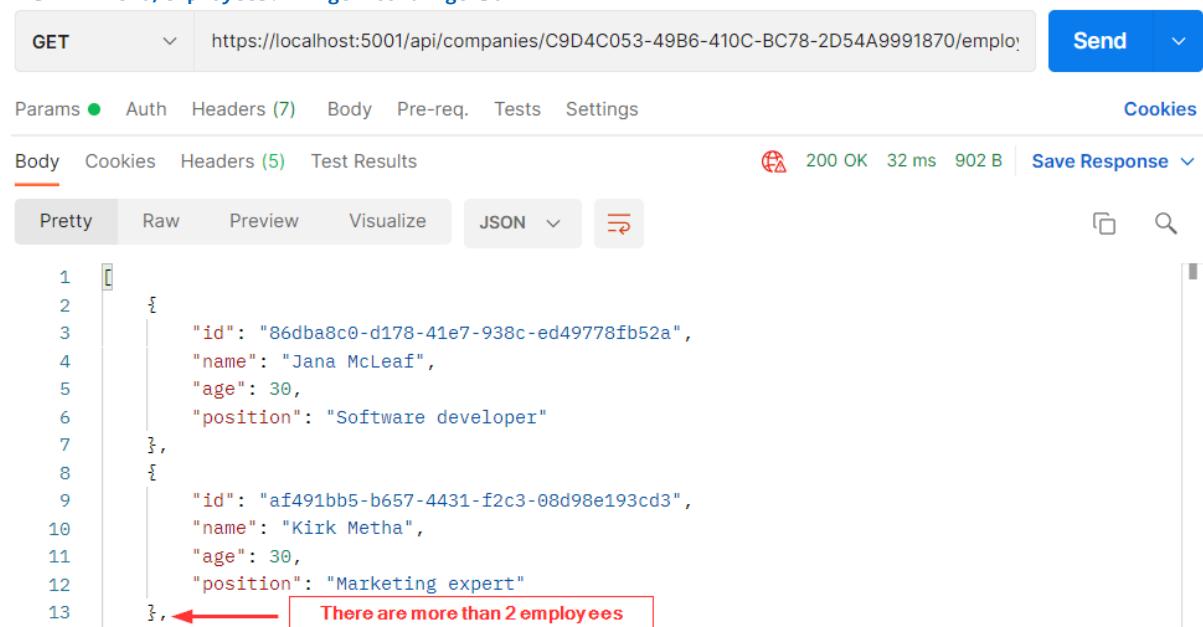
```
1
2
3
4
5
6
7
8
[{"id": "5c8277bf-4a28-4acc-f2c0-08d98e193cd3", "name": "Nina Hawk", "age": 26, "position": "Marketing expert II"}]
```

After that, we can combine those two:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?minAge=26&maxAge=30>



GET https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?minAge=26&maxAge=30

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (5) Test Results

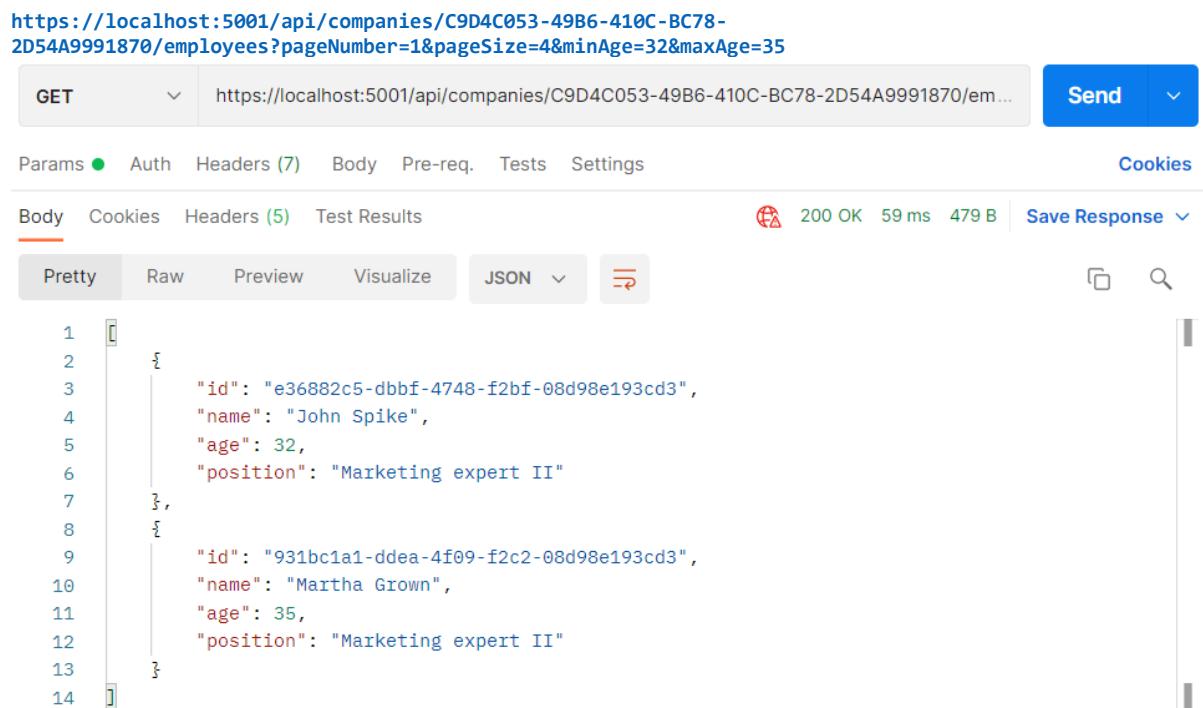
200 OK 32 ms 902 B Save Response

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": "86dba8c0-d178-41e7-938c-ed49778fb52a",
4     "name": "Jana McLeaf",
5     "age": 30,
6     "position": "Software developer"
7   },
8   {
9     "id": "af491bb5-b657-4431-f2c3-08d98e193cd3",
10    "name": "Kirk Metha",
11    "age": 30,
12    "position": "Marketing expert"
13  }
]
```

There are more than 2 employees

And finally, we can test the filter with the paging:



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

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (5) Test Results

200 OK 59 ms 479 B Save Response

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": "e36882c5-dbbf-4748-f2bf-08d98e193cd3",
4     "name": "John Spike",
5     "age": 32,
6     "position": "Marketing expert II"
7   },
8   {
9     "id": "931bc1a1-ddea-4f09-f2c2-08d98e193cd3",
10    "name": "Martha Grown",
11    "age": 35,
12    "position": "Marketing expert II"
13  }
]
```

Excellent. The filter is implemented and we can move on to the searching part.



18 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.

18.1 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.

18.2 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 `searchTerm="name"` 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);
}
```

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.

18.3 Testing Our Implementation

Let's send a first request with the value Mihael Fins for the search term:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?searchTerm=Mihael Fins>

GET https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?searchTerm=Mihael Fins Send

Params ● Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (5) Test Results 200 OK 21 ms 368 B Save Response

Pretty Raw Preview Visualize JSON

```
1
2 {
3     "id": "db6d7cdc-9251-4e0a-f2c1-08d98e193cd3",
4     "name": "Mihael Fins",
5     "age": 30,
6     "position": "Marketing expert"
7 }
```

This is working great.

Now, let's find all employees that contain the letters "ae":

GET https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?searchTerm=ae Send

Params ● Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (5) Test Results 200 OK 64 ms 475 B Save Response

Pretty Raw Preview Visualize JSON

```
1
2 [
3     {
4         "id": "db6d7cdc-9251-4e0a-f2c1-08d98e193cd3",
5         "name": "Mihael Fins",
6         "age": 30,
7         "position": "Marketing expert"
8     },
9     {
10         "id": "ed8b6253-de5f-48bc-f2be-08d98e193cd3",
11         "name": "Mihael Worth",
12         "age": 30,
13         "position": "Marketing expert"
14     }
]
```

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>

The screenshot shows the Postman interface with the following details:

- Method: GET
- URL: https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?pageNumber=1&pageSize=4&minAge=32&maxAge=35&searchTerm=MA
- Status: 200 OK (86 ms, 371 B)
- Body (Pretty):

```
1
2   {
3     "id": "931bc1a1-ddea-4f09-f2c2-08d98e193cd3",
4     "name": "Martha Grown",
5     "age": 35,
6     "position": "Marketing expert II"
7 }
```

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.



19 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.

19.1 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 consider all the parameters 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	28	Software developer	C9D4C053-49B6-410C-BC78-2D54A9991870
2	ED8B6253-DE5F-48BC-F2BE-08D98E193CD3	Mihael Worth	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870
3	E36882C5-DBBF-4748-F2BF-08D98E193CD3	John Spike	32	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
4	5C8277BF-4A28-4ACC-F2C0-08D98E193CD3	Nina Hawk	26	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
5	DB6D7CDC-9251-4E0A-F2C1-08D98E193CD3	Mihael Fins	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870
6	931BC1A1-DDEA-4F09-F2C2-08D98E193CD3	Martha Grown	35	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
7	AF491BB5-B657-4431-F2C3-08D98E193CD3	Kirk Metha	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870
8	86DBA8C0-D178-41E7-938C-ED4977FB52A	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>

The screenshot shows a Postman request to add a new employee. The URL is <https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees>. The request method is POST. The body contains the following JSON data:

```
1  {
2    "name": "Jana McLeaf",
3    "age": 27,
4    "position": "Marketing expert II"
5 }
```

The response status is 201 Created, with a response time of 334 ms and a response size of 393 B. The response body is identical to the request body.

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.



19.2 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.



19.3 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 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** repository 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.

19.4 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:

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

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (5) Test Results 200 OK 40 ms 1.2 KB Save Response

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": "86dba8c0-d178-41e7-938c-ed49778fb52a",
4     "name": "Jana McLeaf",
5     "age": 30,
6     "position": "Software developer"
7   ,
8   {
9     "id": "87537319-be5b-4c70-c6e8-08d98ee515e0",
10    "name": "Jana McLeaf",
11    "age": 27,
12    "position": "Marketing expert II"
13   ,
14   {
15     "id": "e36882c5-dbbf-4748-f2bf-08d98e193cd3",
16     "name": "John Spike",
17     "age": 32,
18     "position": "Marketing expert II"
19   ...
]
```

We can see that this 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.



19.5 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);

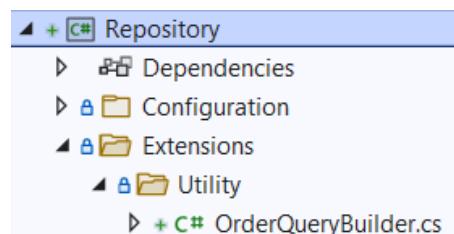
    var orderQuery = OrderQueryBuilder.CreateOrderQuery<Employee>(orderByQueryString);

    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:

The screenshot shows a Postman collection titled '19-Sorting in ASP.NET Core ...'. It contains the following requests:

- POST POST Employee for Compa...
- GET GET Employees by compan...

But now, this functionality is reusable.



20 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.

20.1 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.

20.2 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** from **System.Dynamic** namespace as a return type. We need to do that to shape our data the way we want it.

To implement this interface, we are going to create a new **DataShaping** folder in the **Service** project and add a new **DataShaper** class:

```
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)
    {
```



Ultimate ASP.NET Core Web API

```
        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;
}
```

We need these namespaces to be included as well:

```
using Contracts;
using System.Dynamic;
using System.Reflection;
```

There is quite a lot of code in our class, so let's break it down.

20.3 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;
}
```

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;

}
```

Here, we loop through the **requiredProperties** parameter. 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 **Program** class:

```
builder.Services.AddScoped<IDataShaper<EmployeeDto>, DataShaper<EmployeeDto>>();
```

During the service registration, we provide the type to work with.

Because we want to use the **DataShaper** class inside the service classes, we have to modify the constructor of the **ServiceManager** class first:



```
public ServiceManager(IRepositoryManager repositoryManager, ILoggerManager logger,
    IMapper mapper, IDataShaper<EmployeeDto> dataShaper)
{
    _companyService = new Lazy<ICompanyService>(() =>
        new CompanyService(repositoryManager, logger, mapper));
    _employeeService = new Lazy<IEmployeeService>(() =>
        new EmployeeService(repositoryManager, logger, mapper, dataShaper));
}
```

We are going to use it only in the **EmployeeService** class.

Next, let's add one more field and modify the constructor in the **EmployeeService** class:

```
...
private readonly IDataShaper<EmployeeDto> _dataShaper;

public EmployeeService(IRepositoryManager repository, ILoggerManager logger,
    IMapper mapper, IDataShaper<EmployeeDto> dataShaper)
{
    _repository = repository;
    _logger = logger;
    _mapper = mapper;
    _dataShaper = dataShaper;
}
```

Let's also modify the **GetEmployeesAsync** method of the same class:

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

    await CheckIfCompanyExists(companyId, trackChanges);

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

    var employeesDto =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesWithMetaData);
    var shapedData = _dataShaper.ShapeData(employeesDto,
employeeParameters.Fields);

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

We have changed the method signature so, we have to modify the interface as well:

```
Task<(IEnumerable<ExpandoObject> employees, MetaData metaData)> GetEmployeesAsync(Guid
companyId,
    EmployeeParameters employeeParameters, bool trackChanges);
```



Ultimate ASP.NET Core Web API

Now, we can test our solution:

<https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?fields=name,age>

The screenshot shows the Postman interface with the following details:

- Method: GET
- URL: https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?fields=name,age
- Status: 200 OK (green icon)
- Time: 28 ms
- Size: 547 B
- Save Response button
- Params, Auth, Headers (8), Body, Pre-req., Tests, Settings tabs
- Body tab selected
- JSON dropdown: JSON
- Pretty, Raw, Preview, Visualize buttons
- Copy, Save, Search icons
- Code block (numbered 1-17) showing the JSON response:

```
1 {
2   "Name": "Jana McLeaf",
3   "Age": 27
4 },
5 {
6   "Name": "Jana McLeaf",
7   "Age": 30
8 },
9 {
10  "Name": "John Spike",
11  "Age": 32
12 },
13 {
14  "Name": "Kirk Metha",
15  "Age": 30
16 },
17 }
```

It works great.

Let's also test this solution by combining all the functionalities that we've implemented in the previous chapters:

<https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?pageNumber=1&pageSize=4&minAge=26&maxAge=32&searchTerm=A&orderBy=nameDesc&fields=name,age>



The screenshot shows a Postman request for a GET operation. The URL is <https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees>. The response status is 200 OK with a 79 ms duration and 386 B size. The response body is displayed in Pretty JSON format:

```
1
2
3     {
4         "Name": "Sam Raiden",
5         "Age": 28
6     },
7     {
8         "Name": "Nina Hawk",
9         "Age": 26
10    },
11    {
12        "Name": "Mihael Worth",
13        "Age": 30
14    },
15    {
16        "Name": "Mihael Fins",
17        "Age": 30
18    }
```

Excellent. Everything is working like a charm.

20.4 Resolving XML Serialization Problems

Let's send the same request one more time, but this time with the different accept header (text/xml):

```
<ArrayOfKeyValuePairOfstringanyType xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
<KeyValuePairOfstringanyType>
  <Key>Name</Key>
  <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:string">Sam Raiden</Value>
</KeyValuePairOfstringanyType>
<KeyValuePairOfstringanyType>
  <Key>Age</Key>
  <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:int">28</Value>
</KeyValuePairOfstringanyType>
</ArrayOfKeyValuePairOfstringanyType>
<ArrayOfKeyValuePairOfstringanyType>
  <KeyValuePairOfstringanyType>
    <Key>Name</Key>
    <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:string">Nina Hawk</Value>
  </KeyValuePairOfstringanyType>
  <KeyValuePairOfstringanyType>
    <Key>Age</Key>
    <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:int">26</Value>
  </KeyValuePairOfstringanyType>
</ArrayOfKeyValuePairOfstringanyType>
```



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. Also, you have to do the same thing for the **IEmployeeService** interface and the **EmployeeService** class. 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>

```
1 <ArrayOfEntity xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/Entities.Models">
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.

To sum up, data shaping is an exciting and neat little feature that can 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.



21 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.

21.1 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.



21.1.1 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": 28,
        "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.

21.1.2 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.

21.1.3 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 the developer's 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.



21.2 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 project.

21.3 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.

21.4 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.

21.4.1 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 systemTextJsonOutputFormatter = config.OutputFormatters
            .OfType<SystemTextJsonOutputFormatter>()?.FirstOrDefault();

        if (systemTextJsonOutputFormatter != null)
        {
            systemTextJsonOutputFormatter.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.

Now, we have to add that to the **Program** class, just after the **AddControllers** method:

```
builder.Services.AddCustomMediaTypes();
```

Excellent. The registration process is done.

21.4.2 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 in the **Presentation** project inside the **ActionFilters** folder, 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 **Program** class:

```
builder.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.



21.5 Implementing HATEOAS

We are going to start by creating a new interface in the **Contracts** project:

```
public interface IEmployeeLinks
{
    LinkResponse TryGenerateLinks(IEnumerable<EmployeeDto> employeesDto,
                                  string fields, Guid companyId, HttpContext httpContext);
}
```

Currently, you will get the error about **HttpContext**, but we will solve that a bit later.

Let's continue 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 : IEmployeeLinks
{
    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 **EmployeeService** class, which we will modify a bit later.

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 ReturnLinkedEmployees(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. The `ReturnShapedEmployees` method 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,
employeesDtoList[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 };
}
```

In this method, 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 them 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 **Program** class:

```
builder.Services.AddScoped<IEmployeeLinks, EmployeeLinks>();
```

After the service registration, we are going to create a new record inside the **Entities/LinkModels** folder:

```
public record LinkParameters(EmployeeParameters EmployeeParameters, HttpContext
Context);
```

We are going to use this record to transfer required parameters from our controller to the service layer and avoid the installation of an additional NuGet package inside the **Service** and **Service.Contracts** projects.

Also for this to work, we have to add the reference to the **Shared** project, install the **Microsoft.AspNetCore.Mvc.Abstractions** package needed for **HttpContext**, and add required using directives:

```
using Microsoft.AspNetCore.Http;
using Shared.RequestFeatures;
```



Now, we can return to the **IEmployeeLinks** interface and fix that error by importing the required namespace. As you can see, we didn't have to install the Abstractions NuGet package since **Contracts** references **Entities**. If Visual Studio keeps asking for the package installation, just remove the **Entities** reference from the **Contracts** project and add it again.

Once that is done, we can modify the **EmployeesController**:

```
[HttpGet]
[ServiceFilter(typeof(ValidateMediaTypeAttribute))]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId,
    [FromQuery] EmployeeParameters employeeParameters)
{
    var linkParams = new LinkParameters(employeeParameters, HttpContext);

    var pagedResult = await _service.EmployeeService.GetEmployeesAsync(companyId,
        linkParams, trackChanges: false);

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

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

So, we create the **linkParams** variable and send it instead of **employeeParameters** to the service method.

Of course, this means we have to modify the **IEmployeeService** interface:

```
Task<(LinkResponse linkResponse, MetaData metaData)> GetEmployeesAsync(Guid companyId,
    LinkParameters linkParameters, bool trackChanges);
```

Now the Tuple return type has the **LinkResponse** as the first field and also we have **LinkParameters** as the second parameter.

After we modified our interface, let's modify the **EmployeeService** class:

```
private readonly IRepositoryManager _repository;
private readonly ILoggerManager _logger;
private readonly IMapper _mapper;
private readonly IEmployeeLinks _employeeLinks;

public EmployeeService(IRepositoryManager repository, ILoggerManager logger,
    IMapper mapper, IEmployeeLinks employeeLinks)
{
```



```
        _repository = repository;
        _logger = logger;
        _mapper = mapper;
        _employeeLinks = employeeLinks;
    }

    public async Task<(LinkResponse linkResponse, MetaData metaData)> GetEmployeesAsync
        (Guid companyId, LinkParameters linkParameters, bool trackChanges)
    {
        if (!linkParameters.EmployeeParameters.ValidAgeRange)
            throw new MaxAgeRangeBadRequestException();

        await CheckIfCompanyExists(companyId, trackChanges);

        var employeesWithMetaData = await _repository.Employee
            .GetEmployeesAsync(companyId, linkParameters.EmployeeParameters,
trackChanges);

        var employeesDto =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesWithMetaData);
        var links = _employeeLinks.TryGenerateLinks(employeesDto,
linkParameters.EmployeeParameters.Fields,
companyId, linkParameters.Context);

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

First, we don't have the DataShaper injected anymore since this logic is now inside the EmployeeLinks class. Then, we change the method signature, fix a couple of errors since now we have **linkParameters** and not **employeeParameters** as a parameter, and we call the TryGenerateLinks method, which will return **LinkResponse** as a result.

Finally, we construct our Tuple and return it to the caller.

Now we can return to our controller and modify the **GetEmployeesForCompany** action:

```
[HttpGet]
[ServiceFilter(typeof(ValidateMediaTypeAttribute))]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId,
    [FromQuery] EmployeeParameters employeeParameters)
{
    var linkParams = new LinkParameters(employeeParameters, HttpContext);

    var result = await _service.EmployeeService.GetEmployeesAsync(companyId,
        linkParams, trackChanges: false);

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



```
        return result.linkResponse.HasLinks ? Ok(result.linkResponse.LinkedEntities) :  
            Ok(result.linkResponse.ShanedEntities);  
    }
```

We change the **pageResult** variable name to **result** and use it to return the proper response to the client. If our result has links, we return linked entities, otherwise, we return shaped ones.

Before we test this, we shouldn't forget to modify the ServiceManager's constructor:

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

Excellent. We can test this now:



Ultimate ASP.NET Core Web API

<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>

The screenshot shows the Postman interface with a GET request to the specified URL. The 'Headers' tab is selected, showing a table with one row: 'Accept' set to 'application/vnd.codemaze.hateoas+json'. The 'Body' tab is also visible. The response pane displays a JSON object representing an employee with navigation links for self, delete, update, and partial update.

```
3  {
4      "Name": "Sam Raiden",
5      "Age": 28,
6      "Links": [
7          {
8              "href": "https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/
9                  employees/80abbca8-664d-4b20-b5de-024705497d4a?fields=name,age",
10             "rel": "self",
11             "method": "GET"
12         },
13         {
14             "href": "https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/
15                 employees/80abbca8-664d-4b20-b5de-024705497d4a",
16             "rel": "delete_employee",
17             "method": "DELETE"
18         },
19         {
20             "href": "https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/
21                 employees/80abbca8-664d-4b20-b5de-024705497d4a",
22             "rel": "update_employee",
23             "method": "PUT"
24         },
25         {
26             "href": "https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/
27                 employees/80abbca8-664d-4b20-b5de-024705497d4a",
28             "rel": "partially_update_employee",
29             "method": "PATCH"
30         }
31     ]
32 }
```

You can test this with the xml media type as well (we have prepared the request in Postman for you).



22 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.

22.1 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.

22.2 OPTIONS Implementation

We are going to implement this request in the **CompaniesController** — so, let's open it and add a new 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:

The screenshot shows a Postman request configuration. The method is set to **OPTIONS**, the URL is <https://localhost:5001/api/companies>, and the **Send** button is visible. Below the URL, there are tabs for **Params**, **Auth**, **Headers (6)**, **Body**, **Pre-req.**, **Tests**, and **Settings**. The **Headers** tab is selected. A table titled "Query Params" is present, with one row showing "Key" and "Value". At the bottom, the response summary shows a globe icon, **200 OK**, 20 ms, 132 B, and a "Save Response" button.

As you can see, we are getting a 200 OK response. Let's inspect the Headers tab:

The screenshot shows the Headers tab in Postman with four entries: Content-Length, Date, Server, and Allow. The Content-Length entry has its value "0" highlighted with a red box. The Date value is "Sat, 16 Oct 2021 08:17:29 GMT", the Server value is "Kestrel", and the Allow value is "GET, OPTIONS, POST, PUT, DELETE". Above the table, there are tabs for Body, Cookies, Headers (4), and Test Results. To the right, the response summary shows a globe icon, **200 OK**, 20 ms, 132 B.

Everything works as expected.

Let's move on.



22.3 Head HTTP Request

The 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.

22.4 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 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:

The screenshot shows a Postman interface. The URL is `https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?pageNumber=2&pageSize=2`. The method dropdown is set to **HEAD**. The response status is **200 OK** with a response time of **126 ms** and a size of **230 B**. The response body is empty, containing only the number **1**.

As you can see, we receive a 200 OK status code with the empty body. Let's check the Headers part:



Body Cookies Headers (4) Test Results

🌐 200 OK 126 ms 230 B Save Response ▾

KEY	VALUE
Content-Type ⓘ	application/json; charset=utf-8
Date ⓘ	Sat, 16 Oct 2021 08:23:04 GMT
Server ⓘ	Kestrel
X-Pagination ⓘ	{"CurrentPage":2,"TotalPages":5,"PageSize":2,"TotalCount":9,"HasPrevious":true,"HasNext":true}

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.



23 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.

23.1 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();
}
```

In this action, 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 systemTextJsonOutputFormatter = config.OutputFormatters
            .OfType<SystemTextJsonOutputFormatter>()?.FirstOrDefault();

        if (systemTextJsonOutputFormatter != null)
        {
            systemTextJsonOutputFormatter.SupportedMediaTypes
                .Add("application/vnd.codemaze.hateoas+json");
            systemTextJsonOutputFormatter.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:



Ultimate ASP.NET Core Web API

https://localhost:5001/api

GET https://localhost:5001/api

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Headers 6 hidden

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Accept	application/vnd.codemaze.apiroot+json	

Body Cookies Headers (4) Test Results 200 OK 20 ms

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "href": "https://localhost:5001/api",
4     "rel": "self",
5     "method": "GET"
6   },
7   {
8     "href": "https://localhost:5001/api/companies",
9     "rel": "companies",
10    "method": "GET"
11  },
12  {
13    "href": "https://localhost:5001/api/companies",
14    "rel": "create_company",
15    "method": "POST"
16  }
17 ]
```

This works great.

Let's test what is going to happen if we don't provide the custom media type:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api>

GET https://localhost:5001/api

Params Auth Headers (7) Body Pre-req. Tests Settings

Body Cookies Headers (2) Test Results 204 No Content

Pretty Raw Preview Visualize Text XML

1

Well, we get the **204 No Content** message as expected.

Of course, you can test the xml request as well:

https://localhost:5001/api

GET https://localhost:5001/api Send

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (4) Test Results 200 OK 8 ms 605 B Save Response

Pretty Raw Preview Visualize XML XML

1 <ArrayOfLink xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/Entities.LinkModels">

2 <Link>

3 <Href><https://localhost:5001/api></Href>

4 <Method>GET</Method>

5 <Rel>self</Rel>

6 </Link>

7 <Link>

8 <Href><https://localhost:5001/api/companies></Href>

9 <Method>GET</Method>

10 <Rel>companies</Rel>

11 </Link>

12 <Link>

13 <Href><https://localhost:5001/api/companies></Href>

14 <Method>POST</Method>

15 <Rel>create_company</Rel>

16 </Link>

17 </ArrayOfLink>

Great.

Now we can move on to the versioning chapter.



24 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.

24.1 Required Package Installation and Configuration

In order to start, we have to install the

[Microsoft.AspNetCore.Mvc.Versioning](#) library in the Presentation project:



Microsoft.AspNetCore.Mvc.Versioning by Microsoft
A service API versioning library for Microsoft ASP.NET Core.

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 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 **Program** class:

```
builder.Services.ConfigureVersioning();
```

API versioning is installed and configured, and we can move on.

24.2 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 IServiceProvider _service;

    public CompaniesV2Controller(IServiceProvider service) => _service = service;

    [HttpGet]
```



```
public async Task<IActionResult> GetCompanies()
{
    var companies = await _service.CompanyService
        .GetAllCompaniesAsync(trackChanges: false);

    return Ok(companies);
}
```

By using the `[ApiVersion("2.0")]` attribute, we are stating that this controller is version 2.0.

After that, 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>

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

Params Auth Headers (7) Body Pre-req. Tests Settings

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON ↻

```
1 [
2   {
3     "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
4     "name": "Admin_Solutions Ltd Upd",
5     "fullAddress": "312 Forest Avenue, BF 923 USA"
6   },

```

If we inspect the Headers tab of the response, we are going to find that the controller V1 was assigned for this request:



Body Cookies Headers (5) Test Results

🌐 200 OK 105 ms

KEY	VALUE
Content-Type ⓘ	application/json; charset=utf-8
Date ⓘ	Sat, 16 Oct 2021 18:48:46 GMT
Server ⓘ	Kestrel
Transfer-Encoding ⓘ	chunked
api-supported-versions ⓘ	1.0

Of course, you can place a breakpoint in **GetCompanies** actions in both controllers and confirm which endpoint was hit.

Now, let's see how we can provide a version inside the request.

24.2.1 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>

GET https://localhost:5001/api/companies?api-version=2.0 Send

Params ● Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (5) Test Results 200 OK 2.53 s 666 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
3   "name": "Admin_Solutions Ltd Upd",
4   "fullAddress": "312 Forest Avenue, BF 923 USA"
5 }
```

So, we get the same response body.

But, we can inspect the response headers to make sure that version 2.0 is used:



Body	Cookies	Headers (5)	Test Results
			🌐 200 OK 21 ms
KEY			VALUE
Content-Type	application/json; charset=utf-8	①	
Date	Sat, 16 Oct 2021 19:00:03 GMT	①	
Server	Kestrel	①	
Transfer-Encoding	chunked	①	
api-supported-versions	2.0	①	

24.2.2 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
```

Also, let's just slightly modify the **GetCompanies** action in this controller, so we could see the difference in Postman by just inspecting the response body:

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

    var companiesV2 = companies.Select(x => $"{x.Name} V2");

    return Ok(companiesV2);
}
```

We are creating a projection from our companies collection by iterating through each element, modifying the **Name** property to contain the **V2** suffix, and extracting it to a new collection **companiesV2**.

Now, we can test it:



<https://localhost:5001/api/2.0/companies>

The screenshot shows a Postman interface with a red box highlighting the URL field: `https://localhost:5001/api/2.0/companies`. Below the URL, there are tabs for Params, Auth, Headers (7), Body, Pre-req., Tests, and Settings. The Body tab is selected, showing a JSON response with four items:

```
1 [  
2     "Admin_Solutions Ltd Upd V2",  
3     "Branding Ltd V2",  
4     "IT_Solutions Ltd V2",  
5     "Sales all over the world Ltd V2"  
6 ]
```

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.

24.2.3 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:



<https://localhost:5001/api/companies>

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

Params Auth Headers (8) Body Pre-req. Tests Settings Cookies

KEY	VALUE	DESCRIF	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Accept	application/json				
<input checked="" type="checkbox"/> api-version	2.0				

Body Cookies Headers (5) Test Results

200 OK 39 ms 281 B Save Response

Pretty Raw Preview Visualize JSON

```
1 "Admin_Solutions Ltd Upd V2",
2 "Branding Ltd V2",
3 "IT_Solutions Ltd V2",
4 "Sales all over the world Ltd V2"
```

If we want to support query string versioning, we should use a **new `QueryStringApiVersionReader`** class instead:

```
opt.ApiVersionReader = new QueryStringApiVersionReader("api-version");
```

24.2.4 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:

Body Cookies Headers (5) Test Results

200 OK 95 ms

KEY	VALUE
Content-Type ⓘ	application/json; charset=utf-8
Date ⓘ	Sat, 16 Oct 2021 19:20:53 GMT
Server ⓘ	Kestrel
Transfer-Encoding ⓘ	chunked
api-deprecated-versions ⓘ	2.0



24.2.5 Using Conventions

If we have a lot of versions of a single controller, we can assign these versions in the configuration instead:

```
services.AddApiVersioning(opt =>
{
    opt.ReportApiVersions = true;
    opt.AssumeDefaultVersionWhenUnspecified = true;
    opt.DefaultApiVersion = new ApiVersion(1, 0);
    opt.ApiVersionReader = new HeaderApiVersionReader("api-version");
    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.



25 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.

25.1 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.

25.1.1 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 the client side. It lives on the network.

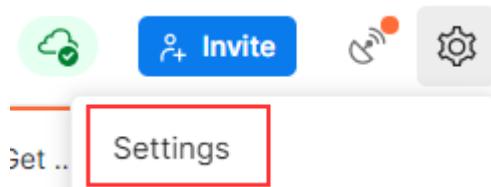
With the private cache, if five clients request the same response for the first time, every response will be served from the API and not from the cache. But if they request the same response again, that response should come from the cache (if it's not expired). This is not the case with the shared cache. 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.

25.1.2 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.

25.2 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:

Headers

Send no-cache header OFF

Great. We can move on.

Let's assume we want to use the `ResponseCache` attribute to cache the result from the `GetCompany` action:

```
public class ResponseCacheAttribute : Attribute, IFilterFactory, IFilterMetadata, IOrderedFilter
{
    public int Duration { ... }
    public ResponseCacheLocation Location { ... }
    public bool NoStore { ... }
    public string? VaryByHeader { ... }
    public string[]? VaryByQueryKeys { ... }
    public string? CacheProfileName { ... }
    public int Order { ... }
    public bool IsReusable { ... }

    public CacheProfile GetCacheProfile(MvcOptions options) { ... }
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider) { ... }

    public ResponseCacheAttribute() { ... }
}
```

It is obvious that 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:



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

GET		https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3	Send	v							
Params	Auth	Headers (7)	Body	Pre-req.	Tests	Settings	Cookies				
Body	Cookies	Headers (6)	Test Results				Save Response v				
KEY							VALUE				
Content-Type ⓘ							application/json; charset=utf-8				
Date ⓘ							Sun, 17 Oct 2021 07:40:03 GMT				
Server ⓘ							Kestrel				
Cache-Control ⓘ							public,max-age=60				
Transfer-Encoding ⓘ							chunked				
api-supported-versions ⓘ							1.0				

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.

25.3 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 **Program** class:

```
builder.Services.ConfigureResponseCaching();
```

Additionally, we have to add caching to the application middleware right below **UseCors()** because Microsoft recommends having **UseCors** before **UseResponseCaching**, and as we learned in the section 1.8, order is very important for the middleware execution:

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

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>

GET	https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3	Send					
Params	Auth	Headers (7)	Body	Pre-req.	Tests	Settings	Cookies
Body	Cookies	Headers (6)	Test Results	🕒 200 OK 20 ms 314 B Save Response ▾			
KEY	VALUE						
Content-Type ⓘ	application/json; charset=utf-8						
Date ⓘ	Sun, 17 Oct 2021 07:50:09 GMT						
Server ⓘ	Kestrel						
Age ⓘ ←	9						
Cache-Control ⓘ	public,max-age=60						
Transfer-Encoding ⓘ	chunked						

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.

Another way to confirm that is to wait 60 seconds to pass. After that, you can send the request and inspect the console. You will see the SQL query generated. But if you send a second request, you will find no new logs for the SQL query. That's because we are receiving our response from the cache.

Additionally, with every subsequent request within 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. You will also see new logs in the console.

Furthermore, 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 the **AddControllers** method:

```
builder.Services.AddControllers(config =>
{
    config.RespectBrowserAcceptHeader = true;
    config.ReturnHttpNotAcceptable = true;
    config.InputFormatters.Insert(0, GetJsonPatchInputFormatter());
    config.CacheProfiles.Add("120SecondsDuration", new CacheProfile { Duration =
120 });
})...
```

We only set up 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 attribute 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**:

<https://localhost:5001/api/companies>

Headers (6)		Test Results			
Content-Type	application/json; charset=utf-8	200	OK	143 ms	701 B
Date	Sun, 17 Oct 2021 08:04:54 GMT				
Server	Kestrel				
Cache-Control	public,max-age=120				
Transfer-Encoding	chunked				
api-supported-versions	1.0				

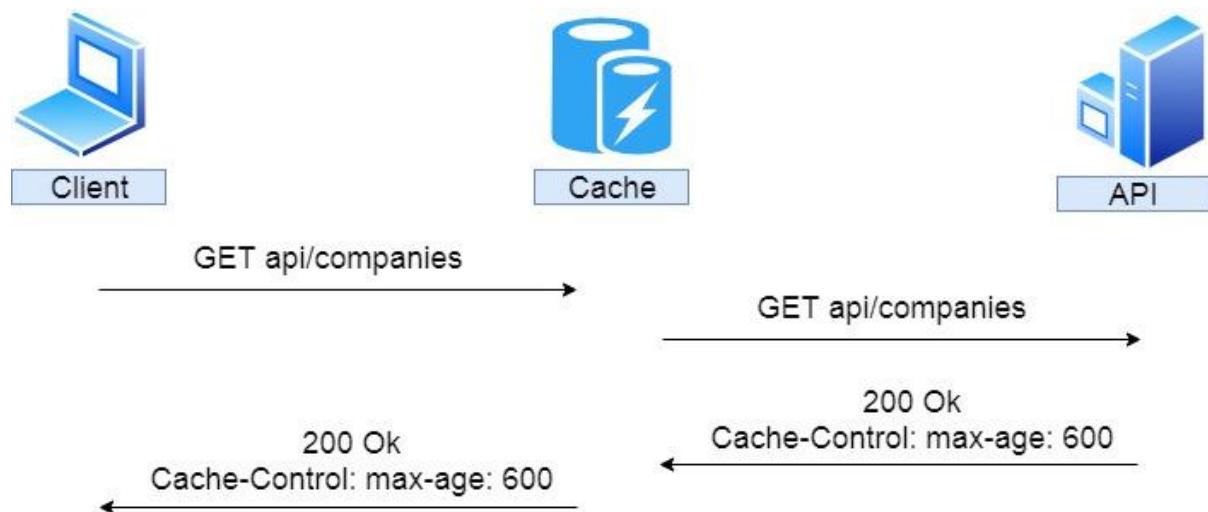
There you go. Now, let's talk some more about the Expiration and Validation models.



25.4 Expiration Model

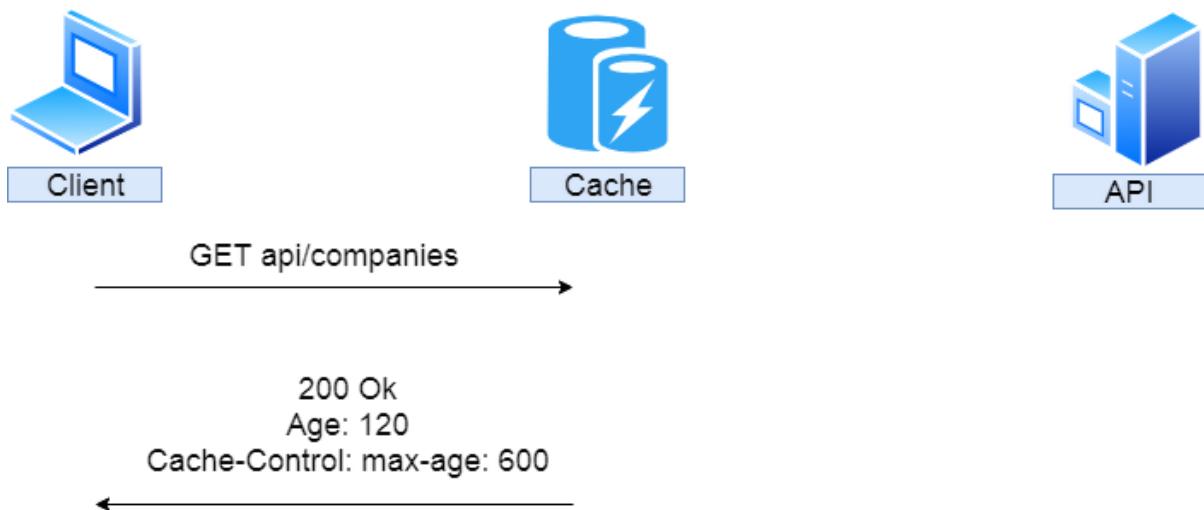
The expiration model allows the server to recognize whether or not the response has expired. As long as the response is fresh, it will be served from the cache. To achieve that, the Cache-Control header is used. We have seen this in the previous example.

Let's look at the diagram to see how caching works:

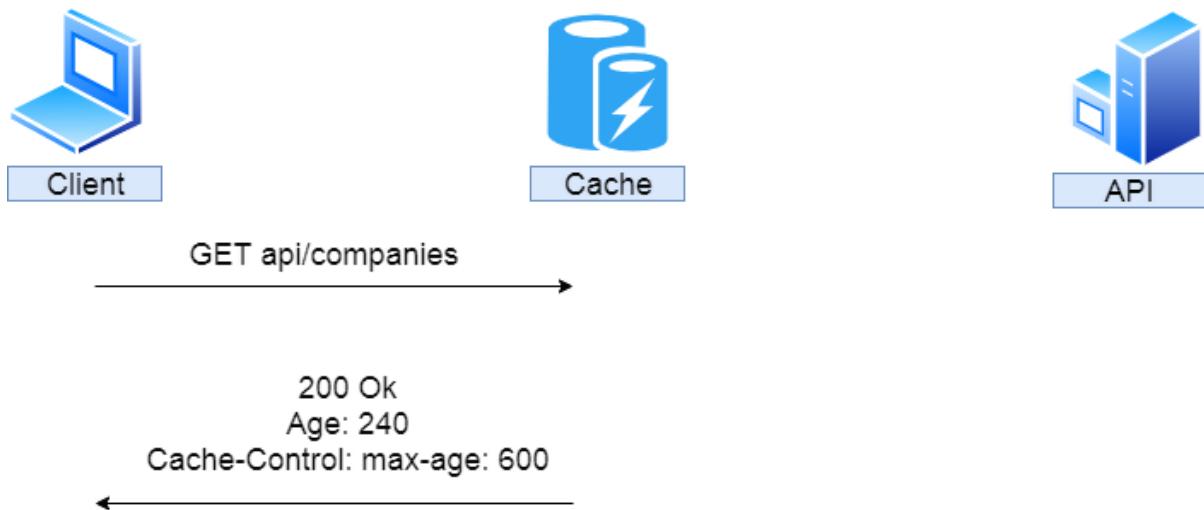


So, the client sends a request to get companies. There is no cached version of that response; therefore, the request is forwarded to the API. The API returns the response with the Cache-Control header with a 10-minute expiration period; it is being stored in the cache and forwarded to the client.

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 a value of 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.

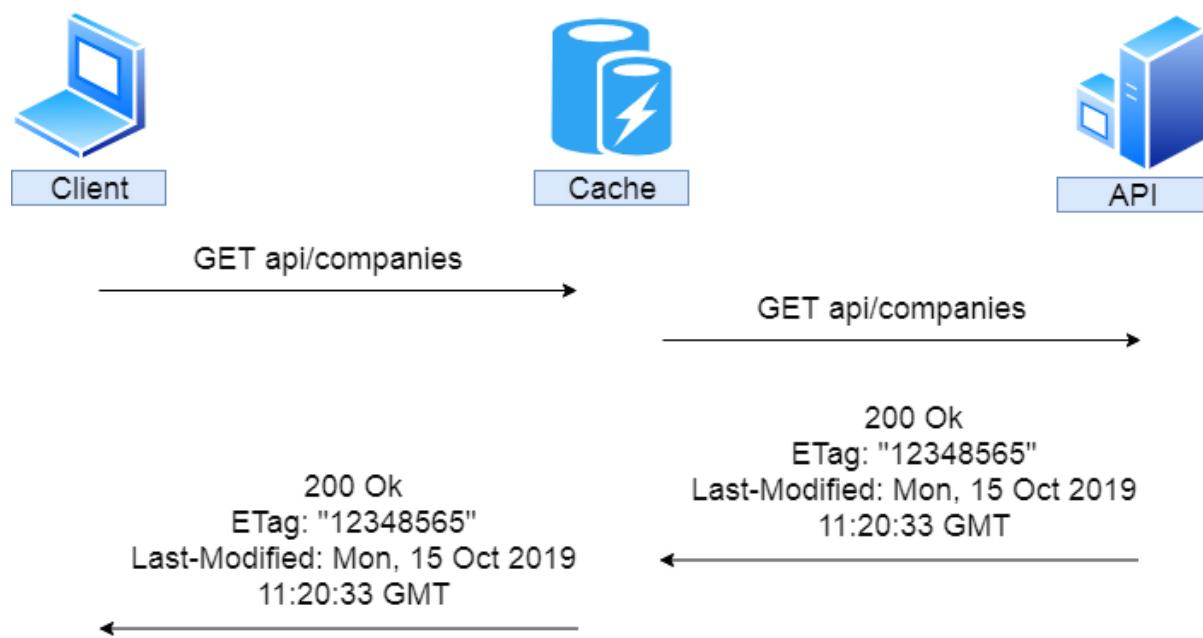


25.5 Validation Model

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.

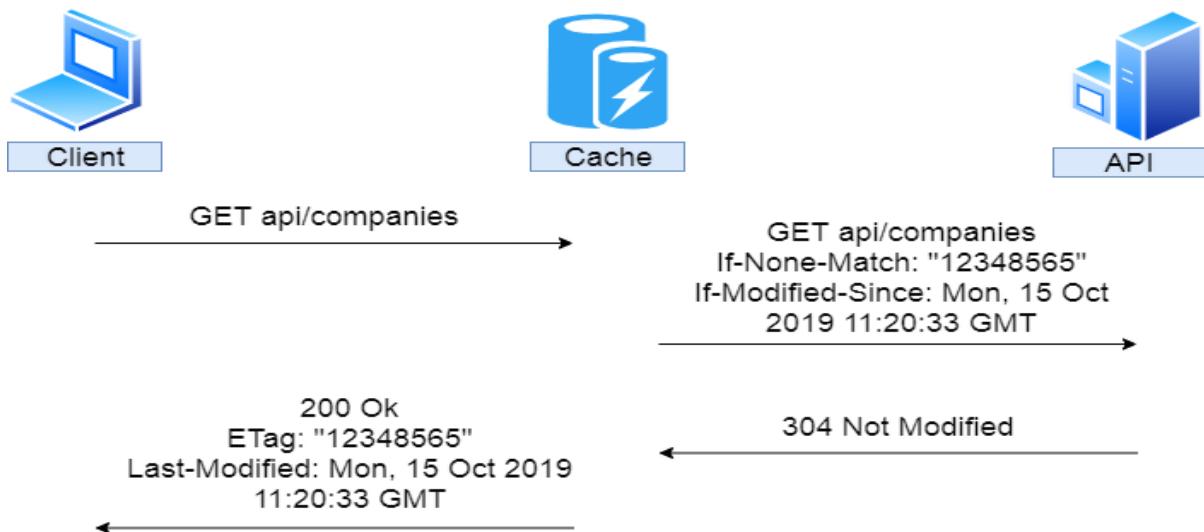
To prevent that, we use validators. The HTTP standard advises using Last-Modified and ETag validators in combination if possible.

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, a 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.

25.6 Supporting Validation

To support validation, we are going to use the **Marvin.Cache.Headers** library. This library supports HTTP cache headers like **Cache-Control**, **Expires**, **Etag**, and **Last-Modified** and also implements validation and expiration models.

So, let's install the **Marvin.Cache.Headers** library in the Presentation project, which will enable the reference for the main project as well. We are going to need it in both projects.



Now, let's modify the **ServiceExtensions** class:

```
public static void ConfigureHttpCacheHeaders(this IServiceProvider services) =>
    services.AddHttpCacheHeaders();
```

We are going to add additional configuration later.

Then, let's modify the **Program** class:

```
builder.Services.ConfigureResponseCaching();
builder.Services.ConfigureHttpCacheHeaders();
```

And finally, let's add HttpCacheHeaders to the request pipeline:

```
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>

Cache-Control ⓘ	public,max-age=60
ETag ⓘ	"B0AE1EB1A7F9A1C19F124B5C9AD666D8"
Expires ⓘ	Sun, 17 Oct 2021 09:25:44 GMT
Last-Modified ⓘ	Sun, 17 Oct 2021 09:24:44 GMT
Transfer-Encoding ⓘ	chunked
Vary ⓘ	Accept, Accept-Language, Accept-Encoding

We can see that 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.

25.6.1 Configuration

We can globally configure our expiration and validation headers. To do that, let's modify the **ConfigureHttpCacheHeaders** method:

```
public static void ConfigureHttpCacheHeaders(this IServiceProvider 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 a single company:

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

KEY	VALUE
Content-Type ⓘ	application/json; charset=utf-8
Date ⓘ	Sun, 17 Oct 2021 09:28:16 GMT
Server ⓘ	Kestrel
Cache-Control ⓘ	private,max-age=65,must-revalidate
ETag ⓘ	"B0AE1EB1A7F9A1C19F124B5C9AD666D8"
Expires ⓘ	Sun, 17 Oct 2021 09:29:21 GMT
Last-Modified ⓘ	Sun, 17 Oct 2021 09:28:16 GMT
Transfer-Encoding ⓘ	chunked
Vary ⓘ	Accept, Accept-Language, Accept-Encoding
api-supported-versions ⓘ	1.0

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. You can check the console again and see the SQL logs for each request you send.

Other than 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:

Cache-Control ⓘ	private,max-age=65,must-revalidate
-----------------	------------------------------------

But if we send the **GetCompany** request:

Cache-Control ⓘ	public,max-age=60
-----------------	-------------------

You can see that it is public and you can send the same request again to see the Age header for the cached response.

25.7 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:

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

The screenshot shows the Postman interface with a GET request to the specified URL. In the Headers section, there are two entries: 'Accept' with the value 'application/json' and 'If-None-Match' with the value '\"BOAE1EB1A7F9A1C19F124B5C9AD666D8\"'. The response status is shown as '304 Not Modified'.



Ultimate ASP.NET Core Web API

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:

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

The screenshot shows a Postman request for a PUT operation. The URL is <https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3>. The Headers tab shows 10 items. The Body tab is selected and contains the following JSON:

```
1
2   ...
3     "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
4     "name": "Admin_Solutions Ltd Upd2",
5     "address": "312 Forest Avenue, BF 923",
6     "country": "USA"
```

The response status is 204 No Content, with a response time of 398 ms and 337 B. The response body is empty.

And then send the same request with the same If-None-Match value:

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

The screenshot shows a Postman request for a GET operation. The URL is <https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3>. The Headers tab is selected and shows two entries: Accept (application/json) and If-None-Match ("B0AE1EB1A7F9A1C19F124B5C9AD666D8"). The response status is 200 OK, with a response time of 97 ms and 512 B. The response body is empty.

You can see that we get 200 OK and if we inspect Headers, we will find that ETag is different because the resource changed:



Body	Cookies	Headers (10)	Test Results	200 OK	97 ms	512 B	Save Response
KEY				VALUE			
		Content-Type	application/json; charset=utf-8				
		Date	Sun, 17 Oct 2021 09:49:03 GMT				
		Server	Kestrel				
		Cache-Control	public,max-age=60				
		ETag	"00FF933A810BAFBF3F47975A3457E4C0"				

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.



26 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 of 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.

26.1 Implementing Rate Limiting

To start, we have to install the [AspNetCoreRateLimit](#) library in the main project:



[AspNetCoreRateLimit](#) by Stefan Prodan, Cristi Pufu, 3.13M 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:

```
builder.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>();
    services.AddSingleton<IProcessingStrategy, AsyncKeyLockProcessingStrategy>();
}
```

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, configuration, and processing strategy as a singleton. They serve the purpose of storing rate limit counters and policies as well as adding configuration.

Now, we have to modify the **Program** class again:

```
builder.Services.AddMemoryCache();
builder.Services.ConfigureRateLimitingOptions();
builder.Services.AddHttpContextAccessor();
```

Finally, we have to add it to the request pipeline:

```
app.UseIpRateLimiting();
```



Ultimate ASP.NET Core Web API

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

And that is it. We can test this now:

<https://localhost:5001/api/companies>

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

Send

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body Cookies Headers (13) Test Results

ETag "0B5E738EA14164FE0BA61BF09EBC94B9"

Expires Sun, 17 Oct 2021 18:49:37 GMT

Last-Modified Sun, 17 Oct 2021 18:48:32 GMT

Transfer-Encoding chunked

Vary Accept, Accept-Language, Accept-Encoding

api-supported-versions 1.0

X-Rate-Limit-Limit 5m

X-Rate-Limit-Remaining 2

X-Rate-Limit-Reset 2021-10-17T18:53:29.7138638Z

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:

<https://localhost:5001/api/companies>

Body Cookies Headers (5) Test Results

429 Too Many Requests 18 ms 212 B

KEY VALUE

Content-Type text/plain

Date Sun, 17 Oct 2021 18:49:53 GMT

Server Kestrel

Retry-After 215

Transfer-Encoding chunked

The status code is 429 Too Many Requests and we have the Retry-After header.

We can inspect the body as well:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies>

The screenshot shows the Postman interface with a request to `https://localhost:5001/api/companies` using the `GET` method. The response status is `429 Too Many Requests`, with a message indicating a quota exceeded: `1 API calls quota exceeded! maximum admitted 3 per 5m.`. The response body is empty.

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](#).



27 JWT, IDENTITY, AND REFRESH TOKEN

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.

Finally, we are going to learn more about the refresh token flow and implement it in our Web API project.

So, let's start with Identity integration.

27.1 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:



.NET

Microsoft.AspNetCore.Identity.EntityFrameworkCore by Microsoft, 24.5M downloads
ASP.NET Core Identity provider that uses Entity Framework Core.

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. For this, we have to include the **Identity.EntityFrameworkCore** namespace:

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
```



We don't have to install the library in the **Repository** project since we already did that in the **Entities** project, and **Repository** has the reference to **Entities**.

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.AddIdentity<User, IdentityRole>(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;
    })
    .AddEntityFrameworkStores<RepositoryContext>()
    .AddDefaultTokenProviders();
}
```

With the **AddIdentity** method, we are adding and configuring Identity for the specific type; in this case, the **User** and the **IdentityRole** type. 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 add **EntityFrameworkStores** implementation with the default token providers.

Now, let's modify the **Program** class:

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

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



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

That's it. We have prepared everything we need.

27.2 Creating Tables and Inserting Roles

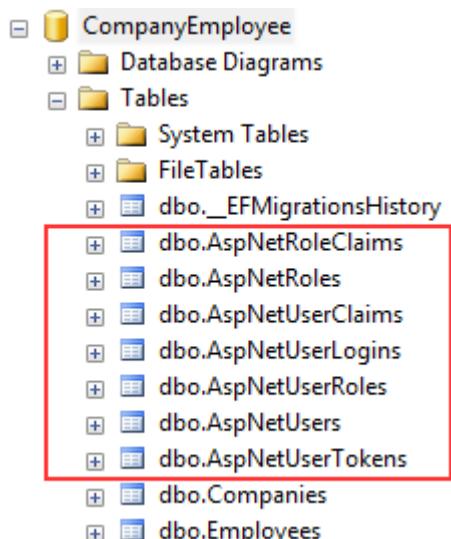
Creating tables is quite an easy process. All we have to do is to create and apply migration. So, let's create a migration:

```
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 **Repository/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"
    }
);
```

For this to work, we need the following namespaces included:

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
```

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.

27.3 User Creation

To create/register a new user, we have to create a new controller:

```
[Route("api/authentication")]
[ApiController]
public class AuthenticationController : ControllerBase
{
    private readonly IServiceProvider _service;

    public AuthenticationController(IServiceProvider service) => _service = service;
}
```



So, nothing new here. We have the basic setup for our controller with **IServiceManager** injected.

The next thing we have to do is to create a **UserForRegistrationDto** record in the **Shared/DataTransferObjects** folder:

```
public record UserForRegistrationDto
{
    public string? FirstName { get; init; }
    public string? LastName { get; init; }
    [Required(ErrorMessage = "Username is required")]
    public string? UserName { get; init; }
    [Required(ErrorMessage = "Password is required")]
    public string? Password { get; init; }
    public string? Email { get; init; }
    public string? PhoneNumber { get; init; }
    public ICollection<string>? Roles { get; init; }
}
```

Then, let's create a mapping rule in the **MappingProfile** class:

```
CreateMap<UserForRegistrationDto, User>();
```

Since we want to extract all the registration/authentication logic to the service layer, we are going to create a new **IAuthenticationService** interface inside the **Service.Contracts** project:

```
public interface IAuthenticationService
{
    Task<IdentityResult> RegisterUser(UserForRegistrationDto userForRegistration);
}
```

This method will execute the registration logic and return the identity result to the caller.

Now that we have the interface, we need to create an implementation service class inside the **Service** project:

```
internal sealed class AuthenticationService : IAuthenticationService
{
    private readonly ILoggerManager _logger;
    private readonly IMapper _mapper;
    private readonly UserManager<User> _userManager;
    private readonly IConfiguration _configuration;

    public AuthenticationService(ILoggerManager logger, IMapper mapper,
        UserManager<User> userManager, IConfiguration configuration)
    {
        _logger = logger;
    }
}
```



```
        _mapper = mapper;
        _userManager = userManager;
        _configuration = configuration;
    }
}
```

This code is pretty familiar from the previous service classes except for the **UserManager** class. This class is used to provide the APIs for managing users in a persistence store. It is not concerned with how user information is stored. For this, it relies on a UserStore (which in our case uses Entity Framework Core).

Of course, we have to add some additional namespaces:

```
using AutoMapper;
using Contracts;
using Entities.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Configuration;
using Service.Contracts;
```

Great. Now, we can implement the **RegisterUser** method:

```
public async Task<IdentityResult> RegisterUser(UserForRegistrationDto
userForRegistration)
{
    var user = _mapper.Map<User>(userForRegistration);

    var result = await _userManager.CreateAsync(user,
userForRegistration.Password);

    if (result.Succeeded)
        await _userManager.AddToRolesAsync(user, userForRegistration.Roles);

    return result;
}
```

So 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 as a result.

After that, if a user is created, we add that user to the named roles — the ones sent from the client side — and return the result.



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.

We want to provide this service to the caller through `ServiceManager` and for that, we have to modify the `IServiceManager` interface first:

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

And then the `ServiceManager` class:

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

    public ServiceManager(IRepositoryManager repositoryManager,
        ILoggerManager logger,
        IMapper mapper, IEmployeeLinks employeeLinks,
        UserManager<User> userManager,
        IConfiguration configuration)
    {
        _companyService = new Lazy<ICompanyService>(() =>
            new CompanyService(repositoryManager, logger, mapper));
        _employeeService = new Lazy<IEmployeeService>(() =>
            new EmployeeService(repositoryManager, logger, mapper,
employeeLinks));
        _authenticationService = new Lazy<IAuthenticationService>(() =>
            new AuthenticationService(logger, mapper, userManager,
configuration));
    }

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

Finally, it is time to create the `RegisterUser` action:

```
[HttpPost]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> RegisterUser([FromBody] UserForRegistrationDto
userForRegistration)
{
    var result = await
_service.AuthenticationService.RegisterUser(userForRegistration);
    if (!result.Succeeded)
```



```
        {
            foreach (var error in result.Errors)
            {
                ModelState.TryAddModelError(error.Code, error.Description);
            }
            return BadRequest(ModelState);
        }

        return StatusCode(201);
    }
}
```

We are implementing our existing action filter for the entity and model validation on top of our action. Then, we call the **RegisterUser** method and accept the result. If the registration fails, we iterate through each error add it to the **ModelState** and return the **BadRequest** response. Otherwise, we return the 201 created status code.

Before we continue with testing, 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.

Now we can start with testing.

Let's send a valid request first:

<https://localhost:5001/api/authentication>

The screenshot shows the Postman interface with the following details:

- Method: POST
- URL: https://localhost:5001/api/authentication
- Headers: (10)
- Body (JSON):

```
1 {
2     "firstname": "Jonh",
3     "lastname": "Doe",
4     "username": "JDoe",
5     "password": "Password1000",
6     "email": "johndoe@mail.com",
7     "phonenumber": "589-654",
8     "roles": [
9         "Manager"
10    ]
11 }
```
- Tests: (12)
- Test Results: 201 Created, 849 ms, 455 B, Save Response



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
```

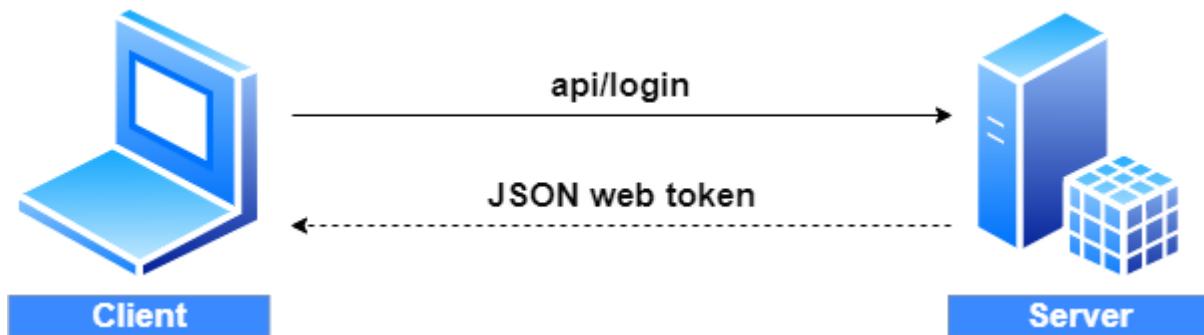
```
{  
    "DuplicateEmail": [  
        "Email 'johndoe@mail.com' is already taken."  
    ],  
    "DuplicateUserName": [  
        "Username 'JDoe' is already taken."  
    ]  
}
```

Excellent. Everything is working as planned. We can move on to the JWT implementation.



27.4 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 their 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.

27.5 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.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiaWF0IjoxNTE2MjM5MDIyfQ.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. On 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.

27.6 JWT Configuration

Let's start by modifying the appsettings.json file:

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft.AspNetCore": "Warning"  
    }  
  },  
  "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["validIssuer"],
            ValidAudience = jwtSettings["validAudience"],
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey))
        };
    });
}
```

First, we extract the **JwtSettings** from the **appsettings.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.

For this to work, we require the following namespaces:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Identity;
using Microsoft.IdentityModel.Tokens;
using System.Text;
```

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 **Program** class:

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

And that is it. We can now protect our endpoints.

27.7 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()
```

The [Authorize] attribute specifies that the action or controller that it is applied to requires authorization. For it to be available we need an additional namespace:

```
using Microsoft.AspNetCore.Authorization;
```

Now to test this, let's send a request to get all companies:

<https://localhost:5001/api/companies>

The screenshot shows the Postman application interface. A GET request is made to the URL <https://localhost:5001/api/companies>. The 'Headers' tab is active, showing 7 items. The 'Body' tab is also visible. At the bottom right, there is a status indicator showing a globe icon and the text '401 Unauthorized'.

We see the protection works. We get a 401 Unauthorized response, which is expected because an unauthorized user tried to access the protected endpoint. So, what we need is for our users to be authenticated and to have a valid token.

27.8 Implementing Authentication

Let's begin with the `UserForAuthenticationDto` record:

```
public record UserForAuthenticationDto
{
    [Required(ErrorMessage = "User name is required")]
    public string? UserName { get; init; }
    [Required(ErrorMessage = "Password name is required")]
    public string? Password { get; init; }
}
```

To continue, let's modify the `IAuthenticationService` interface:

```
public interface IAuthenticationService
{
    Task<IdentityResult> RegisterUser(UserForRegistrationDto userForRegistration);
    Task<bool> ValidateUser(UserForAuthenticationDto userForAuth);
    Task<string> CreateToken();
}
```



Next, let's add a private variable in the **AuthenticationService** class:

```
private readonly UserManager<User> _userManager;
private readonly IConfiguration _configuration;

private User? _user;
```

Before we continue to the interface implementation, we have to install **System.IdentityModel.Tokens.Jwt** library in the **Service** project.

Then, we can implement the required methods:

```
public async Task<bool> ValidateUser(UserForAuthenticationDto userForAuth)
{
    _user = await _userManager.FindByNameAsync(userForAuth.UserName);

    var result = (_user != null && await _userManager.CheckPasswordAsync(_user,
userForAuth.Password));
    if (!result)
        _logger.LogWarning($"{nameof(ValidateUser)}: Authentication failed. Wrong user
name or password.");

    return result;
}

public async Task<string> CreateToken()
{
    var signingCredentials = GetSigningCredentials();
    var claims = await GetClaims();
    var tokenOptions = GenerateTokenOptions(signingCredentials, claims);

    return new JwtSecurityTokenHandler().WriteToken(tokenOptions);
}

private SigningCredentials GetSigningCredentials()
{
    var key = Encoding.UTF8.GetBytes(Environment.GetEnvironmentVariable("SECRET"));
    var secret = new SymmetricSecurityKey(key);

    return new SigningCredentials(secret, SecurityAlgorithms.HmacSha256);
}

private async Task<List<Claim>> GetClaims()
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.Name, _user.UserName)
    };

    var roles = await _userManager.GetRolesAsync(_user);
    foreach (var role in roles)
    {
        claims.Add(new Claim(ClaimTypes.Role, role));
    }

    return claims;
}
```



```
private JwtSecurityToken GenerateTokenOptions(SigningCredentials signingCredentials,
List<Claim> claims)
{
    var jwtSettings = _configuration.GetSection("JwtSettings");

    var tokenOptions = new JwtSecurityToken
    (
        issuer: jwtSettings["validIssuer"],
        audience: jwtSettings["validAudience"],
        claims: claims,
        expires: DateTime.Now.AddMinutes(Convert.ToDouble(jwtSettings["expires"])),
        signingCredentials: signingCredentials
    );

    return tokenOptions;
}
```

For this to work, we require a few more namespaces:

```
using System.IdentityModel.Tokens.Jwt;
using Microsoft.IdentityModel.Tokens;
using System.Text;
using System.Security.Claims;
```

Now we can explain the code.

In the **ValidateUser** method, we fetch the user from the database and check whether they exist 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. If the check result is false, we log a message about failed authentication. Lastly, we return the result.

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  
}
```

Finally, we have to add a new action in the **AuthenticationController**:

```
[HttpPost("login")]  
[ServiceFilter(typeof(ValidationFilterAttribute))]  
public async Task<IActionResult> Authenticate([FromBody] UserForAuthenticationDto  
user)  
{  
    if (!await _service.AuthenticationService.ValidateUser(user))  
        return Unauthorized();  
  
    return Ok(new { Token = await _service  
        .AuthenticationService.CreateToken() });  
}
```

There is nothing special in this controller. If validation fails, we return the 401 Unauthorized response; otherwise, we return our created token:

The screenshot shows a Postman request to `https://localhost:5001/api/authentication/login`. The request method is POST. The JSON body is:

```
1  {  
2   "username": "JDoe",  
3   "password": "Password1000"  
4 }
```

The response status is 200 OK with a response time of 218 ms and a size of 879 B. The response body is a JSON object containing a token:

```
1  {  
2   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXCVJ9.  
eyJodHRwOi8vc2NoZW1hc54bWxzb2FwLm9yZy93cy8yMDA1LzA1L21kZW50aXR5L2NsYWltcy9uYW1lIjoiSkRvZ  
SIisImh0dHA6Ly9zY2hlbWFzLm1pY3Jvc29mdC5jb20vd3MvMjAwOC8wNi9pZGVudGl0eS9jbGFpbXMvcm9sZSI6Ik  
1hbhFnZXIiLCJleHAiOjE2MzQ1NDQ3NzUsImlzcyI6IkNvZGVNYXplQVBIIwiYXVkJjoiaHR0cHM6Ly9sb2NhGh  
vc3Q6NTAwMSJ9.SKHexayg0vzo6GA2_Tba_JtdMhQXLwRz09o0p47WVng"  
3 }
```

Excellent. We can see our token generated.



Now, let's send invalid credentials:

<https://localhost:5001/api/authentication/login>

Body Cookies Headers (11) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "type": "https://tools.ietf.org/html/rfc7235#section-3.1",  
3   "title": "Unauthorized",  
4   "status": 401,  
5   "traceId": "00-fa297853051bc1849b469efbf86458fe-a3558ea45a876af3-00"  
6 }
```



401 Unauthorized 452 ms

And we get a 401 Unauthorized response.

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:

<https://localhost:5001/api/companies>

Params Auth **Headers (8)** Body Pre-req. Tests Settings Cookies

Type
Bearer Token

Token
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ!...

The authorization header will be automatically generated when you send the request.
[Learn more about authorization](#)

Now, we can send the request again:



https://localhost:5001/api/companies

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

Params Auth Headers (8) Body Pre-req. Tests Settings Cookies

Headers Hide auto-generated headers

KEY	VALUE	DESC	...	Bulk Edit	Presets
Authorization	Bearer eyJhbGciOiJIUzI1NilsInR5cCl...				

Body Cookies Headers (13) Test Results 200 OK 188 ms 996 B Save Response

Pretty Raw Preview Visualize JSON

```
1 [ { 2   "id": "3d490a70-94ce-4d15-9494-5248280c2ce3", 3   "name": "Admin_Solutions Ltd Upd2", 4   "fullAddress": "312 Forest Avenue, BF 923 USA" 5 }, ]
```

Excellent. It works like a charm.

27.9 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):



Ultimate ASP.NET Core Web API

POST <https://localhost:5001/api/authentication>

Params Auth Headers (10) **Body** Pre-req. Tests Settings

raw JSON

```
1
2   "firstname": "Jane",
3   "lastname": "Doe",
4   "username": "JaneDoe",
5   "password": "Password2000",
6   "email": "janedoe@mail.com",
7   "phonenumber": "583-653",
8   "roles": [
9     "Administrator"
10  ]
11
```

Body Cookies Headers (12) Test Results 201 Created

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:

<https://localhost:5001/api/companies>

GET <https://localhost:5001/api/companies>

Params Auth Headers (8) Body Pre-req. Tests Settings

Body Cookies Headers (9) Test Results 403 Forbidden

We get a 403 Forbidden response because this user is not allowed to access the required endpoint. If we log in 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.

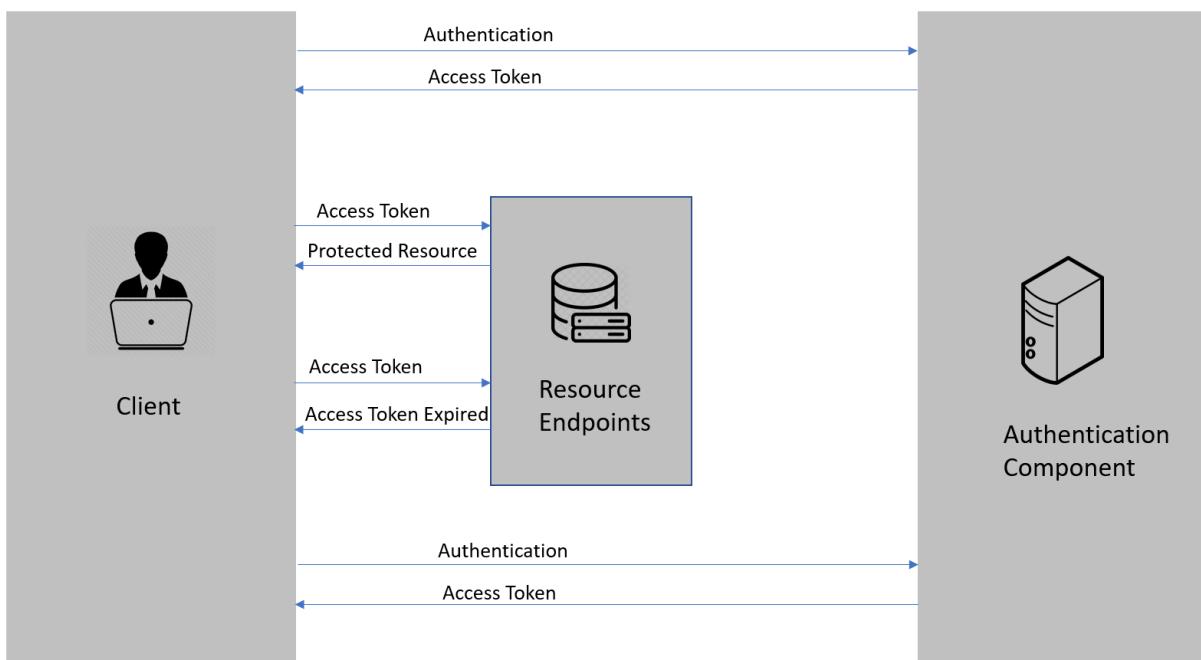
One more thing. Our token expires after five minutes after the creation point. So, if we try to send another request after that period (we probably have to wait 5 more minutes due to the time difference between servers, which is embedded inside the token – this can be overridden with the ClockSkew property in the TokenValidationParameters object), we are going to get the 401 Unauthorized status for sure. Feel free to try.



28 REFRESH TOKEN

In this chapter, we are going to learn about refresh tokens and their use in modern web application development.

In the previous chapter, we have created a flow where a user logs in, gets an access token to be able to access protected resources, and after the token expires, the user has to log in again to obtain a new valid token:

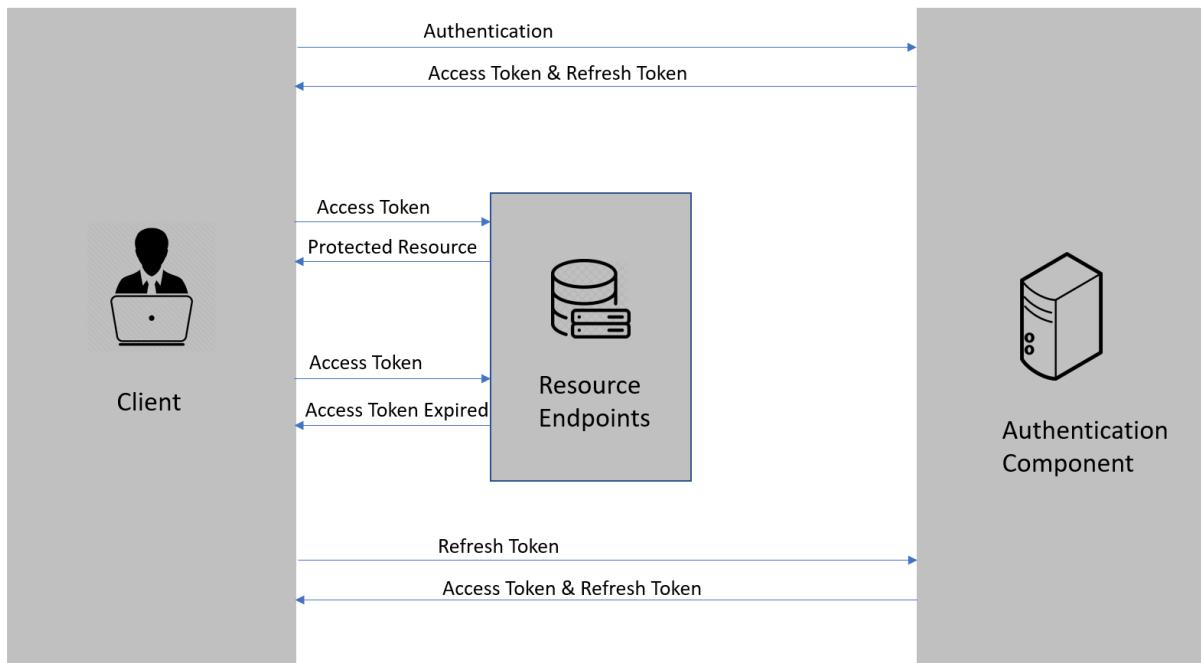


This flow is great and is used by many enterprise applications.

But sometimes we have a requirement not to force our users to log in every single time the token expires. For that, we can use a refresh token.

Refresh tokens are credentials that can be used to acquire new access tokens. When an access token expires, we can use a refresh token to get a new access token from the authentication component. The lifetime of a refresh token is usually set much longer compared to the lifetime of an access token.

Let's introduce the refresh token to our authentication workflow:



1. First, the client authenticates with the authentication component by providing the credentials.
2. Then, the authentication component issues the access token and the refresh token.
3. After that, the client requests the resource endpoints for a protected resource by providing the access token.
4. The resource endpoint validates the access token and provides a protected resource.
5. Steps 3 & 4 keep on repeating until the access token expires.
6. Once the access token expires, the client requests a new access token by providing the refresh token.
7. The authentication component issues a new access token and refresh token.
8. Steps 3 through 7 keep on repeating until the refresh token expires.



9. Once the refresh token expires, the client needs to authenticate with the authentication server once again and the flow repeats from step 1.

28.1 Why Do We Need a Refresh Token

So, why do we need both access tokens and refresh tokens? Why don't we just set a long expiration date, like a month or a year for the access tokens? Because, if we do that and someone manages to get hold of our access token they can use it for a long period, even if we change our password!

The idea of refresh tokens is that we can make the access token short-lived so that, even if it is compromised, the attacker gets access only for a shorter period. With refresh token-based flow, the authentication server issues a one-time use refresh token along with the access token. The app stores the refresh token safely.

Every time the app sends a request to the server it sends the access token in the Authorization header and the server can identify the app using it. Once the access token expires, the server will send a token expired response. Once the app receives the token expired response, it sends the expired access token and the refresh token to obtain a new access token and a refresh token.

If something goes wrong, the refresh token can be revoked which means that when the app tries to use it to get a new access token, that request will be rejected and the user will have to enter credentials once again and authenticate.

Thus, refresh tokens help in a smooth authentication workflow without the need for users to submit their credentials frequently, and at the same time, without compromising the security of the app.



28.2 Refresh Token Implementation

So far we have learned the concept of refresh tokens. Now, let's dig into the implementation part.

The first thing we have to do is to modify the User class:

```
public class User : IdentityUser
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public string? RefreshToken { get; set; }
    public DateTime RefreshTokenExpiryTime { get; set; }
}
```

Here we add two additional properties, which we are going to add to the AspNetUsers table.

To do that, we have to create and execute another migration:

```
Add-Migration AdditionalUserFieldsForRefreshToken
```

If for some reason you get the message that you need to review your migration due to possible data loss, you should inspect the migration file and leave only the code that adds and removes our additional columns:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AddColumn<string>(
        name: "RefreshToken",
        table: "AspNetUsers",
        type: "nvarchar(max)",
        nullable: true);

    migrationBuilder.AddColumn<DateTime>(
        name: "RefreshTokenExpiryTime",
        table: "AspNetUsers",
        type: "datetime2",
        nullable: false,
        defaultValue: new DateTime(1, 1, 1, 0, 0, 0, 0, DateTimeKind.Unspecified));
}

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropColumn(
        name: "RefreshToken",
        table: "AspNetUsers");

    migrationBuilder.DropColumn(
        name: "RefreshTokenExpiryTime",
        table: "AspNetUsers");
}
```



Also, you should open the **RepositoryContextModelSnapshot** file, find the **AspNetRoles** part and revert the Ids of both roles to the previous values:

```
b.ToTable("AspNetRoles", (string)null);

b.HasData(
    new
    {
        Id = "4ac8240a-8498-4869-bc86-60e5dc982d27",
        ConcurrencyStamp = "ec511bd4-4853-426a-a2fc-751886560c9a",
        Name = "Manager",
        NormalizedName = "MANAGER"
    },
    new
    {
        Id = "562419f5-eed1-473b-bcc1-9f2dbab182b4",
        ConcurrencyStamp = "937e9988-9f49-4bab-a545-b422dde85016",
        Name = "Administrator",
        NormalizedName = "ADMINISTRATOR"
    });
});
```

After that is done, we can execute our migration with the **Update-Database** command. This will add two additional columns in the **AspNetUsers** table.

To continue, let's create a new record in the **Shared/DataTransferObjects** folder:

```
public record TokenDto(string AccessToken, string RefreshToken);
```

Next, we are going to modify the **IAuthenticationService** interface:

```
public interface IAuthenticationService
{
    Task<IdentityResult> RegisterUser(UserForRegistrationDto userForRegistration);
    Task<bool> ValidateUser(UserForAuthenticationDto userForAuth);
    Task<TokenDto> CreateToken(bool populateExp);
}
```

Then, we have to implement two new methods in the **AuthenticationService** class:

```
private string GenerateRefreshToken()
{
    var randomNumber = new byte[32];
    using (var rng = RandomNumberGenerator.Create())
    {
        rng.GetBytes(randomNumber);
        return Convert.ToString(randomNumber);
```



```
    }

private ClaimsPrincipal GetPrincipalFromExpiredToken(string token)
{
    var jwtSettings = _configuration.GetSection("JwtSettings");

    var tokenValidationParameters = new TokenValidationParameters
    {
        ValidateAudience = true,
        ValidateIssuer = true,
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(Environment.GetEnvironmentVariable("SECRET"))),
        ValidateLifetime = true,
        ValidIssuer = jwtSettings["validIssuer"],
        ValidAudience = jwtSettings["validAudience"]
    };

    var tokenHandler = new JwtSecurityTokenHandler();
    SecurityToken securityToken;
    var principal = tokenHandler.ValidateToken(token, tokenValidationParameters, out
securityToken);

    var jwtSecurityToken = securityToken as JwtSecurityToken;
    if (jwtSecurityToken == null ||
!jwtSecurityToken.Header.Alg.Equals(SecurityAlgorithms.HmacSha256,
    StringComparison.InvariantCultureIgnoreCase))
    {
        throw new SecurityTokenException("Invalid token");
    }

    return principal;
}
```

GenerateRefreshToken contains the logic to generate the refresh token. We use the **RandomNumberGenerator** class to generate a cryptographic random number for this purpose.

GetPrincipalFromExpiredToken is used to get the user principal from the expired access token. We make use of the **ValidateToken** method from the **JwtSecurityTokenHandler** class for this purpose. This method validates the token and returns the **ClaimsPrincipal** object.

After that, to generate a refresh token and the expiry date for the logged-in user, and to return both the access token and refresh token to the caller, we have to modify the **CreateToken** method in the same class:

```
public async Task<TokenDto> CreateToken(bool populateExp)
{
    var signingCredentials = GetSigningCredentials();
```



```
var claims = await GetClaims();
var tokenOptions = GenerateTokenOptions(signingCredentials, claims);

var refreshToken = GenerateRefreshToken();

_user.RefreshToken = refreshToken;

if(populateExp)
    _user.RefreshTokenExpiryTime = DateTime.Now.AddDays(7);

await _userManager.UpdateAsync(_user);

var accessToken = new JwtSecurityTokenHandler().WriteToken(tokenOptions);

return new TokenDto(accessToken, refreshToken);
}
```

Finally, we have to modify the **Authenticate** action:

```
[HttpPost("login")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> Authenticate([FromBody] UserForAuthenticationDto
user)
{
    if (!await _service.AuthenticationService.ValidateUser(user))
        return Unauthorized();

    var tokenDto = await _service.AuthenticationService
        .CreateToken(populateExp: true);

    return Ok(tokenDto);
}
```

That's it regarding the action modification.

Now, we can test this by sending the POST request from Postman:



<https://localhost:5001/api/authentication/login>

POST https://localhost:5001/api/authentication/login Send

Params Auth Headers (10) Body **JSON** Pre-req. Tests Settings Cookies Beautify

```
1 {  
2   "username": "JDoe",  
3   "password": "Password1000"  
4 }
```

Body Cookies Headers (13) Test Results 200 OK 849 ms 947 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {  
2   "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJodHRwOi8vc2NoZW1hcyc54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjois  
kRvZSISImh0dHA6Ly9zY2h1bWFzLm1pY3Jvc29mdC5jb20vd3MvMjAwOC8wNi9pZGVudGl0eS9jbGFpbXMvcm  
9sZSI6Ik1hbmcFnZXIiLCJleHAiOjE2MzQ1NTk0OTYsImlzcyI6IkNvZGVNYXplQVBJIiwiYXVKIjoiaHR0cHM  
6Ly9sb2Nhbgvhc3Q6NTAwMSJ9.OPoQNCNdazc6vLYdqNBN71TjGI3zs0APY_1byl_7Qlg",  
3   "refreshToken": "LjmKhKtIr16WQ37pRyeEc9x2ve9+AT5KGslnDQSsgVg="  
4 }
```

We can see the successful authentication and both our tokens.

Additionally, if we inspect the database, we are going to find populated RefreshToken and Expiry columns for JDoe:

RefreshToken	RefreshTokenExpiryTime
LjmKhKtIr16WQ37pRyeEc9x2ve9+AT5KGslnDQSsgVg=	2021-10-25 14:13:16.3943712

28.3 Token Controller Implementation

It is a good practice to have a separate endpoint for the refresh token action, and that's exactly what we are going to do now.

Let's start by creating a new **TokenController** in the **Presentation** project:

```
[Route("api/token")]
[ApiController]
public class TokenController : ControllerBase
{
    private readonly IServiceProvider _service;

    public TokenController(IServiceProvider service) => _service = service;
}
```



Before we continue with the controller modification, we are going to modify the **IAuthenticationService** interface:

```
public interface IAuthenticationService
{
    Task<IdentityResult> RegisterUser(UserForRegistrationDto userForRegistration);
    Task<bool> ValidateUser(UserForAuthenticationDto userForAuth);
    Task<TokenDto> CreateToken(bool populateExp);
    Task<TokenDto> RefreshToken(TokenDto tokenDto);
}
```

And to implement this method:

```
public async Task<TokenDto> RefreshToken(TokenDto tokenDto)
{
    var principal = GetPrincipalFromExpiredToken(tokenDto.AccessToken);

    var user = await _userManager.FindByNameAsync(principal.Identity.Name);
    if (user == null || user.RefreshToken != tokenDto.RefreshToken ||
        user.RefreshTokenExpiryTime <= DateTime.Now)
        throw new RefreshTokenBadRequest();

    _user = user;

    return await CreateToken(populateExp: false);
}
```

We first extract the principal from the expired token and use the **Identity.Name** property, which is the username of the user, to fetch that user from the database. If the user doesn't exist, or the refresh tokens are not equal, or the refresh token has expired, we stop the flow returning the **BadRequest** response to the user. Then we just populate the **_user** variable and call the **CreateToken** method to generate new Access and Refresh tokens. This time, we don't want to update the expiry time of the refresh token thus sending **false** as a parameter.

Since we don't have the **RefreshTokenBadRequest** class, let's create it in the **Entities\Exceptions** folder:

```
public sealed class RefreshTokenBadRequest : BadRequestException
{
    public RefreshTokenBadRequest()
        : base("Invalid client request. The tokenDto has some invalid values.")
    {
    }
}
```



And add a required using directive in the **AuthenticationService** class to remove the present error.

Finally, let's add one more action in the TokenController:

```
[HttpPost("refresh")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> Refresh([FromBody]TokenDto tokenDto)
{
    var tokenDtoToReturn = await
_service.AuthenticationService.RefreshToken(tokenDto);

    return Ok(tokenDtoToReturn);
}
```

That's it.

Our refresh token logic is prepared and ready for testing.

Let's first send the POST authentication request:

The screenshot shows a Postman request to `https://localhost:5001/api/authentication/login`. The Body tab is selected, showing the following JSON payload:

```
1 → "username": "JDoe",
2 → "password": "Password1000"
```

The response status is 200 OK, with a response time of 102 ms and a size of 947 B. The response body contains two tokens:

```
1 ↴ {
  "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJodHRwOi8vc2NoZW1hcyc54bWzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjoi
kRVZSIsImh0dHA6Ly9zY2hlbWFzLm1pY3Jvc29mdC5jb20vd3MvMjAwOC8wNi9pZGVudGl0eS9jbGFpbXMcm
9sZSI6Ik1hbmcnZXIiLCJleHAiOjE2MzQ1NjQ5NzMsImlzcyI6IkNvZGVNYXplQVBJIiwiYXVKIjoiaHR0cHM
6Ly9sb2Nhbgvhc3Q6NTAwMSJ9.e34Wacc7IdTaSA6uAS95wkos8LP3qnyLvoMBazeCOEc",
  "refreshToken": "YA9KbaVKU+KuDj7QsUbKfPWFYUVa1e74gHCVf50WLWY="
```

As before, we have both tokens in the response body.

Now, let's send the POST refresh request with these tokens as the request body:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/token/refresh>

POST

https://localhost:5001/api/token/refresh

Send

Params Auth Headers (10) Body ● Pre-req. Tests Settings

Cookies

raw

JSON

Beautify

```
1 {"accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
2 eyJodHRwOi8vc2NoZW1hcyc54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjois  
3 kRvZSIIsImh0dHA6Ly9zY2hlbWFzLm1pY3Jvc29mdC5jb20vd3MvMjAwOC8wNi9pZGVudGl0eS9jbGFpbXMvcm  
4 9sZSI6Ik1hbmFnZXIiLCJleHAiOjE2MzQ1NjQ5NzMzMlzcI6IkNvZGVNYXplQVBJIiwiYXVKIjoiaHR0cHM  
6Ly9sb2Nhbgvhc3Q6NTAwMSJ9.e34Wacc7IdTaSA6uAS5wkos8LP3qnyLvoMBazeCOEc",  
3 "refreshToken": "YA9KbaVKU+KuDj7qsUbKfPWFYUVa1e74gHCVf50WLwY="
```

Body Cookies Headers (13) Test Results

200 OK 100 ms 947 B

Save Response

Pretty

Raw

Preview

Visualize

JSON



```
1 {"accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
2 eyJodHRwOi8vc2NoZW1hcyc54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjois  
3 kRvZSIIsImh0dHA6Ly9zY2hlbWFzLm1pY3Jvc29mdC5jb20vd3MvMjAwOC8wNi9pZGVudGl0eS9jbGFpbXMvcm  
4 9sZSI6Ik1hbmFnZXIiLCJleHAiOjE2MzQ1NjUwNDIsImlzcI6IkNvZGVNYXplQVBJIiwiYXVKIjoiaHR0cHM  
6Ly9sb2Nhbgvhc3Q6NTAwMSJ9.gTHG8FtZMo8YokRIA1o9ok3SebAgNPQ7S3TgeJQmpvw",  
3 "refreshToken": "x7/dHKgGxliios34LF3Kqlq5n33aUh+zGXhtZ3SrS2w="
```

And we can see new tokens in the response body. Additionally, if we inspect the database, we will find the same refresh token value:

RefreshToken	RefreshTokenExpiryTime
x7/dHKgGxliios34LF3Kqlq5n33aUh+zGXhtZ3SrS2w=	2021-10-25 15:44:33.5052966

Usually, in your client application, you would inspect the **exp** claim of the access token and if it is about to expire, your client app would send the request to the **api/token** endpoint and get a new set of valid tokens.



29 BINDING CONFIGURATION AND OPTIONS PATTERN

In the previous chapter, we had to use our appsettings file to store some important values for our JWT configuration and read those values from it:

```
"JwtSettings": {  
    "validIssuer": "CodeMazeAPI",  
    "validAudience": "https://localhost:5001",  
    "expires": 5  
},
```

To access these values, we've used the **GetSection** method from the **IConfiguration** interface:

```
var jwtSettings = configuration.GetSection("JwtSettings");
```

The **GetSection** method gets a sub-section from the appsettings file based on the provided key.

Once we extracted the sub-section, we've accessed the specific values by using the **jwtSettings** variable of type **IConfigurationSection**, with the key provided inside the square brackets:

```
ValidIssuer = jwtSettings["validIssuer"],
```

This works great but it does have its flaws.

Having to type sections and keys to get the values can be repetitive and error-prone. We risk introducing errors to our code, and these kinds of errors can cost us a lot of time until we discover them since someone else can introduce them, and we won't notice them since a null result is returned when values are missing.

To overcome this problem, we can bind the configuration data to strongly typed objects. To do that, we can use the **Bind** method.



29.1 Binding Configuration

To start with the binding process, we are going to create a new **ConfigurationModels** folder inside the **Entities** project, and a new **JwtConfiguration** class inside that folder:

```
public class JwtConfiguration
{
    public string Section { get; set; } = "JwtSettings";

    public string? ValidIssuer { get; set; }
    public string? ValidAudience { get; set; }
    public string? Expires { get; set; }
}
```

Then in the **ServiceExtensions** class, we are going to modify the **ConfigureJWT** method:

```
public static void ConfigureJWT(this IServiceCollection services, IConfiguration
configuration)
{
    var jwtConfiguration = new JwtConfiguration();
    configuration.Bind(jwtConfiguration.Section, jwtConfiguration);

    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 = jwtConfiguration.ValidIssuer,
            ValidAudience = jwtConfiguration.ValidAudience,
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey))
        };
    });
}
```

We create a new instance of the **JwtConfiguration** class and use the **Bind** method that accepts the section name and the instance object as parameters, to bind to the **JwtSettings** section directly and map configuration values to respective properties inside the



JwtConfiguration class. Then, we just use those properties instead of string keys inside square brackets, to access required values.

There are two things to note here though. The first is that the names of the configuration data keys and class properties must match. The other is that if you extend the configuration, you need to extend the class as well, which can be a bit cumbersome, but it beats getting values by typing strings.

Now, we can continue with the **AuthenticationService** class modification since we extract configuration values in two methods from this class:

```
...
private readonly JwtConfiguration _jwtConfiguration;

private User? _user;

public AuthenticationService	ILoggerManager logger, IMapper mapper,
                           UserManager<User> userManager, IConfiguration configuration)
{
    _logger = logger;
    _mapper = mapper;
    _userManager = userManager;
    _configuration = configuration;
    _jwtConfiguration = new JwtConfiguration();
    _configuration.Bind(_jwtConfiguration.Section, _jwtConfiguration);
}
```

So, we add a readonly variable, and create an instance and execute binding inside the constructor.

And since we're using the **Bind()** method we need to install the **Microsoft.Extensions.Configuration.Binder** NuGet package.

After that, we can modify the **GetPrincipalFromExpiredToken** method by removing the **GetSection** part and modifying the **TokenValidationParameters** object creation:

```
private ClaimsPrincipal GetPrincipalFromExpiredToken(string token)
{
    var tokenValidationParameters = new TokenValidationParameters
    {
        ValidateAudience = true,
        ValidateIssuer = true,
```



```
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(Environment.GetEnvironmentVariable("SECRET"))),
        ValidateLifetime = true,
        ValidIssuer = _jwtConfiguration.ValidIssuer,
        ValidAudience = _jwtConfiguration.ValidAudience
    };
    ...
    return principal;
}
```

And let's do a similar thing for the **GenerateTokenOptions** method:

```
private JwtSecurityToken GenerateTokenOptions(SigningCredentials signingCredentials,
List<Claim> claims)
{
    var tokenOptions = new JwtSecurityToken(
    {
        issuer: _jwtConfiguration.ValidIssuer,
        audience: _jwtConfiguration.ValidAudience,
        claims: claims,
        expires: DateTime.Now.AddMinutes(Convert.ToDouble(_jwtConfiguration.Expires)),
        signingCredentials: signingCredentials
    });

    return tokenOptions;
}
```

Excellent.

At this point, we can start our application and use both requests from Postman's collection - **28-Refresh Token** - to test our configuration.

We should get the same responses as we did in a previous chapter, which proves that our configuration works as intended but now with a better code and less error-prone.

29.2 Options Pattern

In the previous section, we've seen how we can bind configuration data to strongly typed objects. The options pattern gives us similar possibilities, but it offers a more structured approach and more features like validation, live reloading, and easier testing.



Once we configure the class containing our configuration we can inject it via dependency injection with **IOptions<T>** and thus injecting only part of our configuration or rather only the part that we need.

If we need to reload the configuration without stopping the application, we can use the **IOptionsSnapshot<T>** interface or the **IOptionsMonitor<T>** interface depending on the situation. We'll see when these interfaces should be used and why.

The options pattern also provides a good validation mechanism that uses the widely used DataAnnotations attributes to check if the configuration abides by the logical rules of our application.

The testing of options is also easy because of the helper methods and easy to mock options classes.

29.2.1 Using IOptions

We have already written a lot of code in the previous section that can be used with the **IOptions** interface, but we still have some more actions to do.

The first thing we are going to do is to register and configure the JwtConfiguration class in the **ServiceExtensions** class:

```
public static void AddJwtConfiguration(this IServiceCollection services,  
IConfiguration configuration) =>  
    services.Configure<JwtConfiguration>(configuration.GetSection("JwtSettings"));
```

And call this method in the **Program** class:

```
builder.Services.ConfigureJWT(builder.Configuration);  
builder.Services.AddJwtConfiguration(builder.Configuration);
```

Since we can use **IOptions** with DI, we are going to modify the **ServiceManager** class to support that:

```
public ServiceManager(IRepositoryManager repositoryManager,  
ILoggerManager logger,  
IMapper mapper, IEmployeeLinks employeeLinks,  
UserManager<User> userManager,  
IOptions<JwtConfiguration> configuration)
```



We just replace the **IConfiguration** type with the **IOptions** type in the constructor.

For this, we need two additional namespaces:

```
using Entities.ConfigurationModels;
using Microsoft.Extensions.Options;
```

Then, we can modify the AuthenticationService's constructor:

```
private readonly ILoggerManager _logger;
private readonly IMapper _mapper;
private readonly UserManager<User> _userManager;
private readonly IOptions<JwtConfiguration> _configuration;
private readonly JwtConfiguration _jwtConfiguration;

private User? _user;

public AuthenticationService(ILoggerManager logger, IMapper mapper,
    UserManager<User> userManager, IOptions<JwtConfiguration> configuration)
{
    _logger = logger;
    _mapper = mapper;
    _userManager = userManager;
    _configuration = configuration;
    _jwtConfiguration = _configuration.Value;
}
```

And that's it.

We inject **IOptions** inside the constructor and use the **Value** property to extract the **JwtConfiguration** object with all the populated properties. Nothing else has to change in this class.

If we start the application again and send the same requests, we will still get valid results meaning that we've successfully implemented **IOptions** in our project.

One more thing. We didn't modify anything inside the **ServiceExtensions/ConfigureJWT** method. That's because this configuration happens during the service registration and not after services are built. This means that we can't resolve our required service here.



Well, to be precise, we can use the **BuildServiceProvider** method to build a service provider containing all the services from the provided **IServiceCollection**, and thus being able to access the required service. But if you do that, you will create one more list of singleton services, which can be quite expensive depending on the size of your application. So, you should be careful with this method.

That said, using Binding to access configuration values is perfectly safe and cheap in this stage of the application's lifetime.

29.2.2 IOptionsSnapshot and IOptionsMonitor

The previous code looks great but if we want to change the value of Expires to 10 instead of 5 for example, we need to restart the application to do it. You can imagine how useful would be to have a published application and all you need to do is to modify the value in the configuration file without restarting the whole app.

Well, there is a way to do it by using **IOptionsSnapshot** or **IOptionsMonitor**.

All we would have to do is to replace the **IOptions<JwtConfiguration>** type with the **IOptionsSnapshot<JwtConfiguration>** or **IOptionsMonitor<JwtConfiguration>** types inside the **ServiceManager** and **AuthenticationService** classes. Also if we use **IOptionsMonitor**, we can't use the **Value** property but the **CurrentValue**.

So the main difference between these two interfaces is that the **IOptionsSnapshot** service is registered as a scoped service and thus can't be injected inside the singleton service. On the other hand, **IOptionsMonitor** is registered as a singleton service and can be injected into any service lifetime.



To make the comparison even clearer, we have prepared the following list for you:

IOptions<T>:

- Is the original Options interface and it's better than binding the whole Configuration
- Does not support configuration reloading
- Is registered as a singleton service and can be injected anywhere
- Binds the configuration values only once at the registration, and returns the same values every time
- Does not support named options

IOptionsSnapshot<T>:

- Registered as a scoped service
- Supports configuration reloading
- Cannot be injected into singleton services
- Values reload per request
- Supports named options

IOptionsMonitor<T>:

- Registered as a singleton service
- Supports configuration reloading
- Can be injected into any service lifetime
- Values are cached and reloaded immediately
- Supports named options

Having said that, we can see that if we don't want to enable live reloading or we don't need named options, we can simply use `IOptions<T>`. If we do, we can use either `IOptionsSnapshot<T>` or `IOptionsMonitor<T>`, but `IOptionsMonitor<T>` can be injected into other singleton services while `IOptionsSnapshot<T>` cannot.

We have mentioned Named Options a couple of times so let's explain what that is.



Let's assume, just for example sake, that we have a configuration like this one:

```
"JwtSettings": {  
    "validIssuer": "CodeMazeAPI",  
    "validAudience": "https://localhost:5001",  
    "expires": 5  
},  
"JwtAPI2Settings": {  
    "validIssuer": "CodeMazeAPI2",  
    "validAudience": "https://localhost:5002",  
    "expires": 10  
},
```

Instead of creating a new `JwtConfiguration2` class that has the same properties as our existing `JwtConfiguration` class, we can add another configuration:

```
services.Configure<JwtConfiguration>("JwtSettings",  
configuration.GetSection("JwtSettings"));  
services.Configure<JwtConfiguration>("JwtAPI2Settings",  
configuration.GetSection("JwtAPI2Settings"));
```

Now both sections are mapped to the same configuration class, which makes sense. We don't want to create multiple classes with the same properties and just name them differently. This is a much better way of doing it.

Calling the specific option is now done using the `Get` method with a section name as a parameter instead of the `Value` or `CurrentValue` properties:

```
_jwtConfiguration = _configuration.Get("JwtSettings");
```

That's it. All the rest is the same.



30 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.

30.1 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.



30.2 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 in the main project. Let's open the Package Manager Console window and type the following command:

```
PM> Install-Package Swashbuckle.AspNetCore
```

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.

Also, we need an additional namespace:

```
using Microsoft.OpenApi.Models;
```

The next step is to call this method in the **Program** class:

```
builder.Services.ConfigureSwagger();
```

And in the middleware part of the class, 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 slightly 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>:

The screenshot shows the Swagger UI interface for the 'Code Maze API'. At the top, there's a navigation bar with the 'Swagger' logo and a dropdown menu showing 'Select a definition' set to 'Code Maze API v1'. Below the header, the main title 'Code Maze API' is displayed, followed by 'v1' and 'OAS3' badges. A sidebar on the left lists several sections: 'Authentication', 'Companies', 'Employees', 'Root', 'Token', and 'Schemas'. Each section has a small downward arrow to its right, indicating it can be expanded. The 'Schemas' section is currently expanded, showing a list of DTOs.

Also if we expand the Schemas part, we are going to find the DTOs that we used in our project.

If we click on a specific controller to expand its details, we are going to see all the actions inside:



Companies

GET	/api/companies
POST	/api/companies
OPTIONS	/api/companies
GET	/api/companies/{id}
DELETE	/api/companies/{id}
PUT	/api/companies/{id}
GET	/api/companies/collection/({ids})
POST	/api/companies/collection

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:

Companies

GET /api/companies

Parameters

No parameters

Execute

Cancel

Once we click the **Execute** button, we are going to see that we get our response:



Responses

Curl

```
curl -X 'GET' \
  'https://localhost:5001/api/companies' \
  -H 'accept: */*'
```



Request URL

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

Server response

Code Details

401

Undocumented Error: response status is 401

And this is an expected response. We are not authorized. To enable authorization, we have to add some modifications.

30.3 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:

Authentication

POST /api/authentication/login

Companies

GET /api/companies

POST /api/companies

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:



Authentication

POST /api/authentication/login

Parameters

No parameters

Request body

application/json-patch+json

```
{  
    "userName": "JDoe",  
    "password": "Password1000"  
}
```

Execute Clear

200 Response body

```
{  
    "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcyc54bWzbz2FwLm9  
yZy93cy8yMDA1LzA1L21kZW50aXR5L2NsYWItcy9uYm11IjoisKrvZSIstmh0dHAGLy9zY2h1bWFzLm1pY3Jvc29md  
C5jb20vd3MvMjAwOC8wNi9pZGVudGl0eS9jbGFpbXMvcm9sZSI6Ik1hbmfNzXiILC1eHAI0jE2MzQ3OTk20DgsIm1  
zcI6IkNvZGVNYXpIQVB3IiwiYXVkJioiaHR0cHM6Ly9sb2NhbGhvC3Q6NTAwMSJ9.xJInShw2czcm9Kckt5cpoZ_P  
KYNotWaLM4can010Yng",  
    "refreshToken": "19d08WnfGxOtsNt2Kd/Zdov0HhFNJLDNjg28fipjvWY="  
}
```

Download

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:

Available authorizations

X

Bearer (apiKey)

Place to add JWT with Bearer

Name: Authorization

In: header

Value:

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcyc54bWzbz2FwLm9yZy93cy8yMDA1LzA1L21kZW50aXR5L2NsYWItcy9uYm11IjoisKrvZSIstmh0dHAGLy9zY2h1bWFzLm1pY3Jvc29mdC5jb20vd3MvMjAwOC8wNi9pZGVudGl0eS9jbGFpbXMvcm9sZSI6Ik1hbmfNzXiILC1eHAI0jE2MzQ3OTk20DgsIm1zcI6IkNvZGVNYXpIQVB3IiwiYXVkJioiaHR0cHM6Ly9sb2NhbGhvC3Q6NTAwMSJ9.xJInShw2czcm9Kckt5cpoZ_PKYNotWaLM4can010Yng

Authorize

Close



After authorization, we are going to click on the Close button and try our request:

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

Server response

Code	Details
200	Response body

```
[  
  {  
    "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",  
    "name": "Admin_Solutions Ltd Upd2",  
    "fullAddress": "312 Forest Avenue, BF 923 USA"  
},
```

And we get our response. Excellent job.

30.4 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:

Code Maze API v1 OAS3

</swagger/v1/swagger.json>

CompanyEmployees API by CodeMaze

[Terms of service](#)

[John Doe - Website](#)

[Send email to John Doe](#)

[CompanyEmployees API LICX](#)

For enabling XML comments, we need to suppress warning 1591, which will now give warnings about any method, class, or field that doesn't have triple-slash comments. We need to do this in the **Presentation** project.

Additionally, we have to add the documentation path for the same project, since our controllers are in the Presentation project:

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFramework>net6.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
    </PropertyGroup>

    <PropertyGroup Condition="'$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
        <DocumentationFile>CompanyEmployees.Presentation.xml</DocumentationFile>
        <OutputPath></OutputPath>
        <NoWarn>1701;1702;1591</NoWarn>
    </PropertyGroup>

    <PropertyGroup Condition="'$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
        <NoWarn>1701;1702;1591</NoWarn>
    </PropertyGroup>
```

Now, let's modify our configuration:

```
s.SwaggerDoc("v2", new OpenApiInfo { Title = "Code Maze API", Version = "v2" });

var xmlFile = $"{typeof(Presentation.AssemblyReference).Assembly.GetName().Name}.xml";
var xmlPath = Path.Combine(ApplicationContext.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 screenshot shows the 'Companies' section of the Swagger UI. It features a 'GET' button, a URL field containing '/api/companies', and a description box that reads 'Gets the list of all companies'. There is also a lock icon indicating security information.

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)]
```

Here, we are using both XML comments and data annotation attributes.

Now, we can see the result:

The screenshot shows the 'POST /api/companies' endpoint in the Swagger UI. It includes a description 'Creates a newly created company'.

And, if we inspect the response part, we will find our mentioned responses:



Responses

Code	Description	Links
201	Returns the newly created item	<i>No links</i>
400	If the item is null	<i>No links</i>
	<p>Media type</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">text/plain</div> <p>Example Value Schema</p> <pre>{ "type": "string", "title": "string", "status": 0, "detail": "string", "instance": "string", "additionalProp1": "string", "additionalProp2": "string", "additionalProp3": "string" }</pre>	

422

If the model is invalid

No links

Media type

text/plain

▼

Excellent.

We can continue to the deployment part.



31 DEPLOYMENT TO IIS

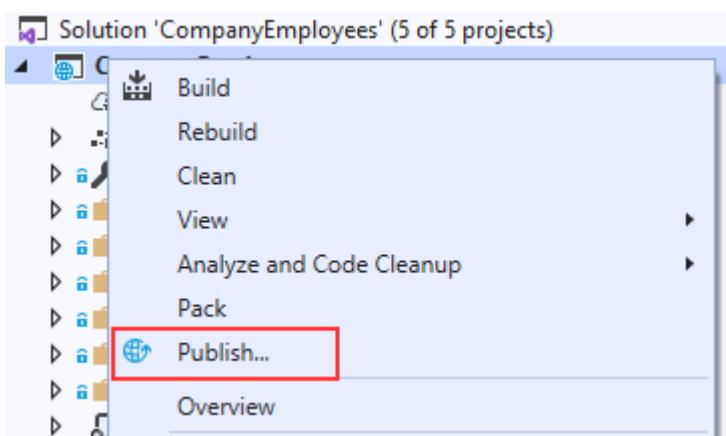
Before we start the deployment process, we would like to point out one important thing. We should always try to deploy an application on at least a local machine to somehow simulate the production environment as soon as we start with development. That way, we can observe how the application behaves in 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.

31.1 Creating Publish Files

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 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 click Next:



Ultimate ASP.NET Core Web API

Publish

Where are you publishing today?

The screenshot shows a list of publishing targets:

- Azure**: Publish your application to the Microsoft cloud.
- Docker Container Registry**: Publish your application to any supported Container Registry that works with Docker images.
- Folder**: Publish your application to a local folder or file share. This option is highlighted with a blue background.
- FTP/FTPS Server**: Publish your application to an FTP/FTPS server.
- Web Server (IIS)**: Publish your application to IIS using Web Deploy or Web Deploy Package.
- Import Profile**: Import your publish settings to deploy your app.

At the bottom of the dialog are buttons for Back, Next, Finish, and Cancel.

And point to the location of the Publish folder we just created and click Finish:

Publish

Provide the path to a local or network folder

The screenshot shows the "Location" step of the publish process. The "Target" is set to "Folder location". The "Folder location" field contains "D:\Publish". A "Browse..." button is available to change the path. Below the input field, there is a note and a list:

For local folders you can provide either a full path or a relative path to the project, for example:

- publish\ (relative path)
- C:\Users\Username\Documents (full path)

Publish windows can be different depending on the Visual Studio version.

After that, we have to click the Publish button:



The screenshot shows the 'FolderProfile1.pubxml' publish profile in Visual Studio. At the top, there's a folder icon labeled 'FolderProfile1.pubxml' with a dropdown arrow, and a blue 'Publish' button. Below that is a toolbar with a '+' icon for 'New' and a 'More actions' dropdown. A prominent message box says 'Ready to publish.' Below the message are several configuration settings:

Setting	Value	Action
Target location	D:\Publish	
Delete existing files	False	
Configuration	Release	
Target Runtime	Portable	

[Show all settings](#)

Visual Studio is going to do its job and publish the required files in the specified folder.

3.1.2 Windows Server Hosting Bundle

Before 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.

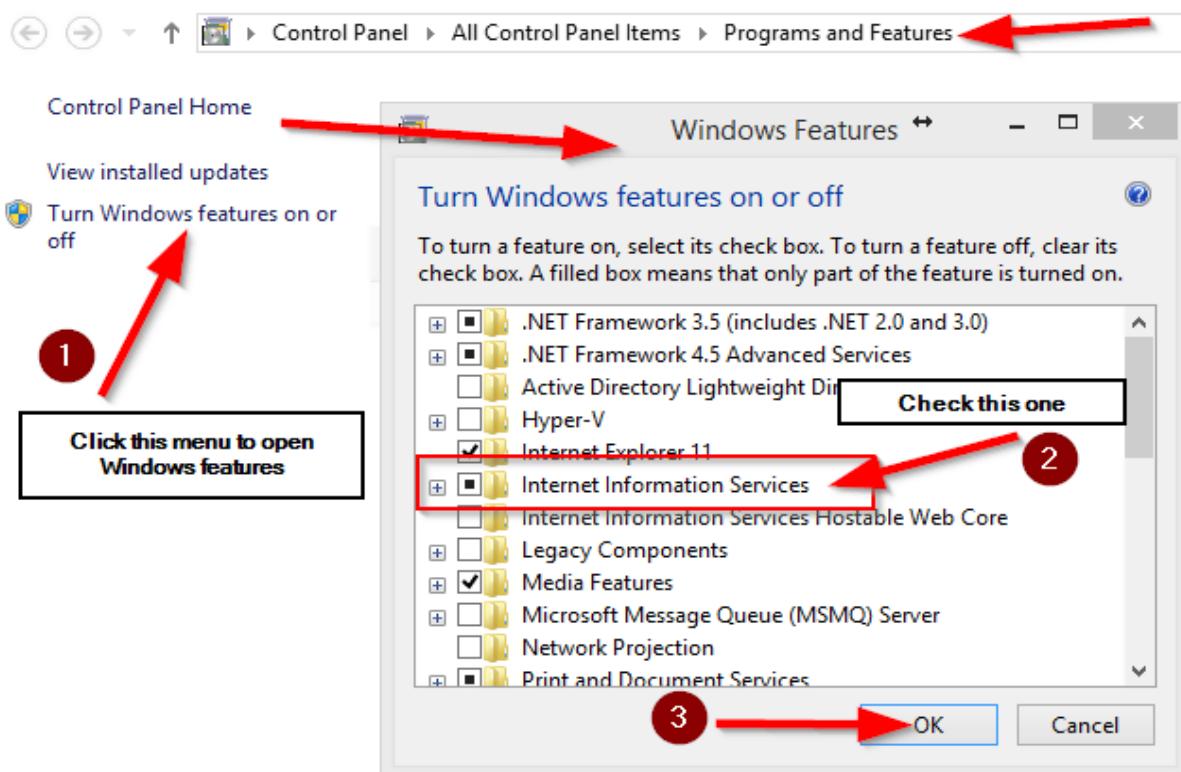
After the installation, we are going to locate the Windows hosts file on C:\Windows\System32\drivers\etc and add the following record at the end of the file:

127.0.0.1 www.companyemployees.codemaze

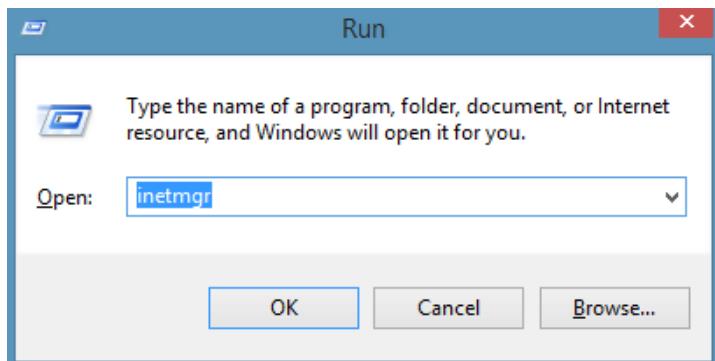
After that, we are going to save the file.

3.1.3 Installing IIS

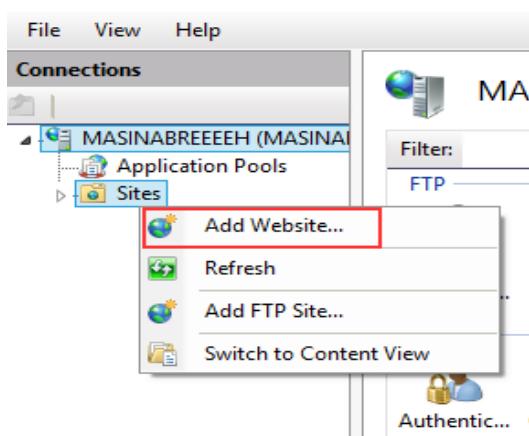
If you don't have IIS installed on your machine, you need to install it by opening ControlPanel and then Programs and Features:



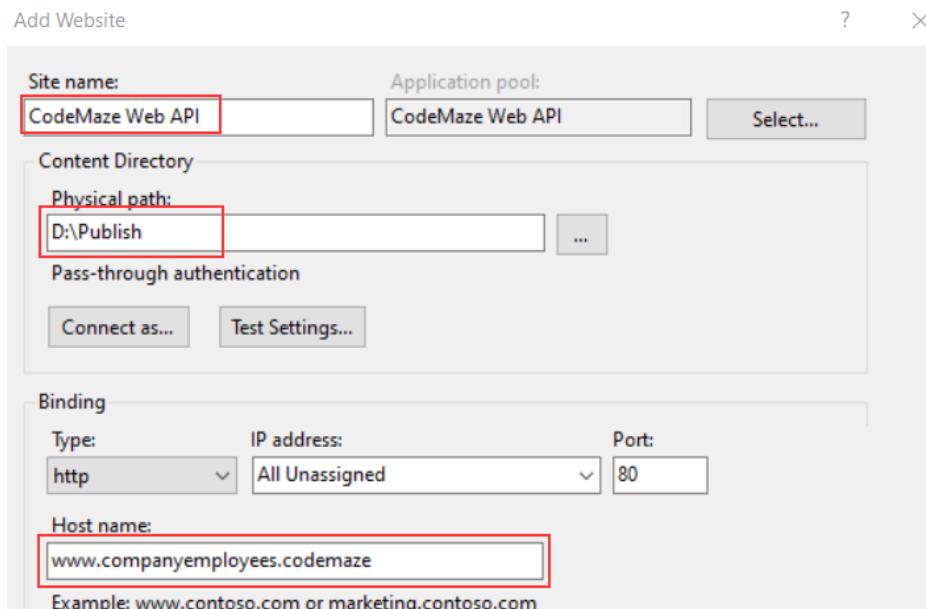
After the IIS installation finishes, let's open the Run window (windows key + R) and type: **inetmgr** to open the IIS manager:



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:



And click the OK button.



Ultimate ASP.NET Core Web API

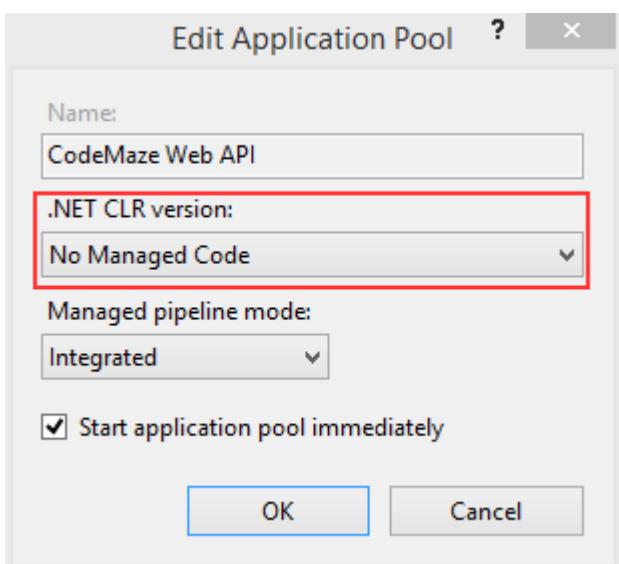
After this step, we are going to have our site inside the “sites” folder in the IIS Manager. Additionally, we need to set up some basic settings for our application pool:

The screenshot shows the IIS Manager's Application Pools section. On the left, under 'Connections', there's a tree view with 'MASINABREEEH (MASINA)' expanded, showing 'Application Pools' which is also highlighted with a red box. The main area is titled 'Application Pools' and contains a table with the following data:

Name	Status	.NET CLR Version	Managed Pipe...
CodeMaze Web API	Started	v4.0	Integrated
[redacted]	Started	No Managed Code	Integrated
[redacted]	Started	No Managed Code	Integrated

On the right, the 'Actions' pane includes links for 'Add Application Pool...', 'Set Application Pool Defaults...', 'Start', 'Stop', 'Recycle...', 'Edit Application Pool' (which is selected), 'Basic Settings...', 'Recycling...', and 'Advanced Settings...'. The 'Edit Application Pool' and 'Basic Settings...' links are highlighted with a red box.

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.

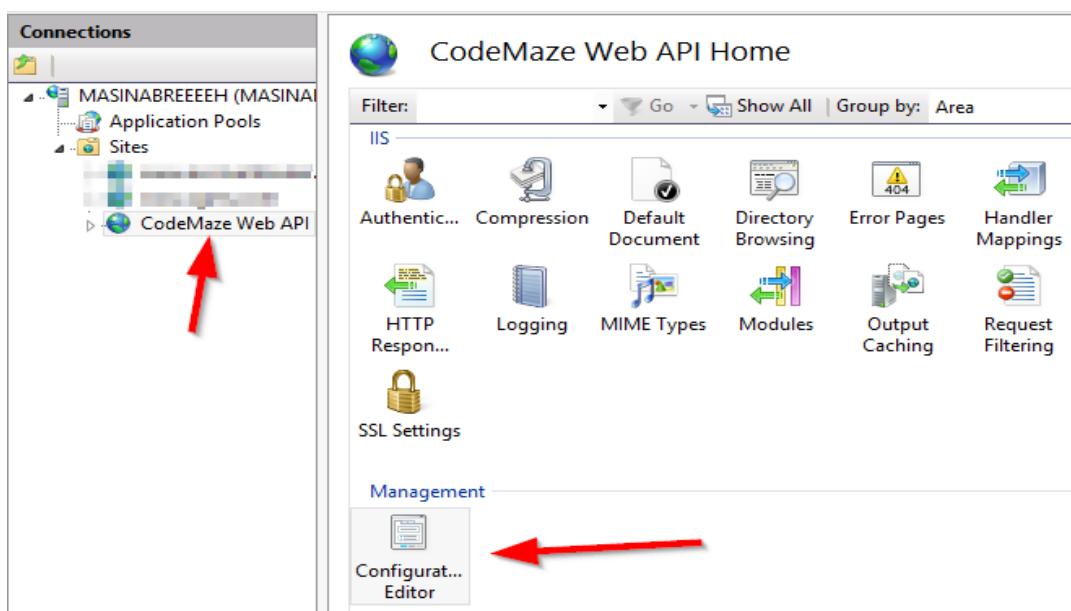
Our website and the application pool should be started automatically.



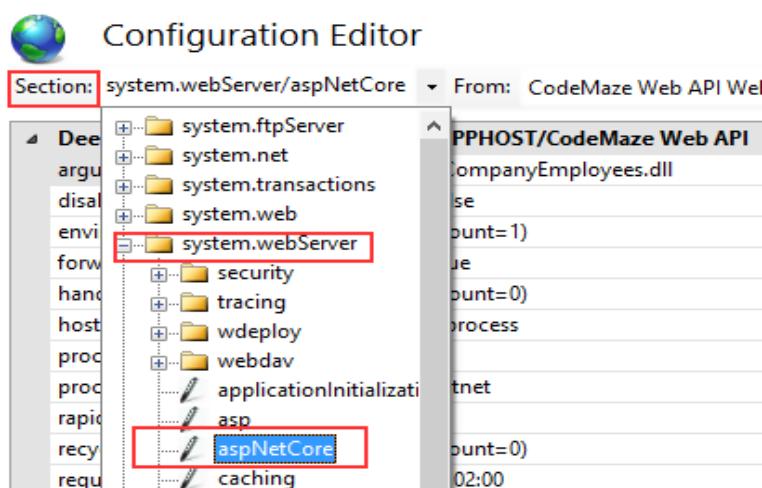
31.4 Configuring Environment File

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**:

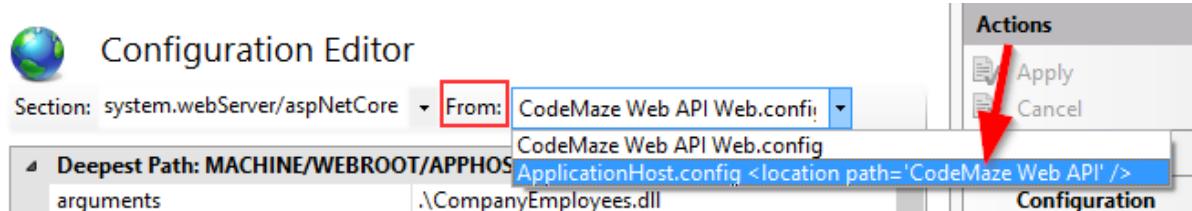


Then, in the section box, we are going to choose **system.webServer/aspNetcore**:

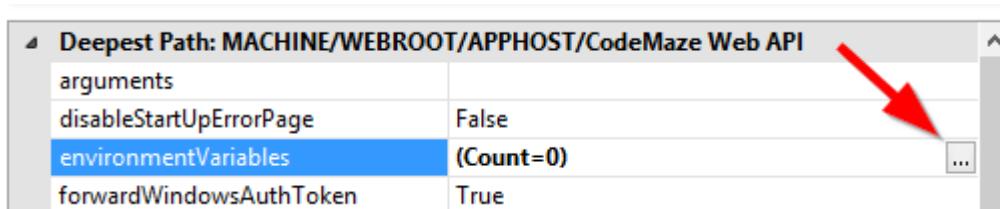




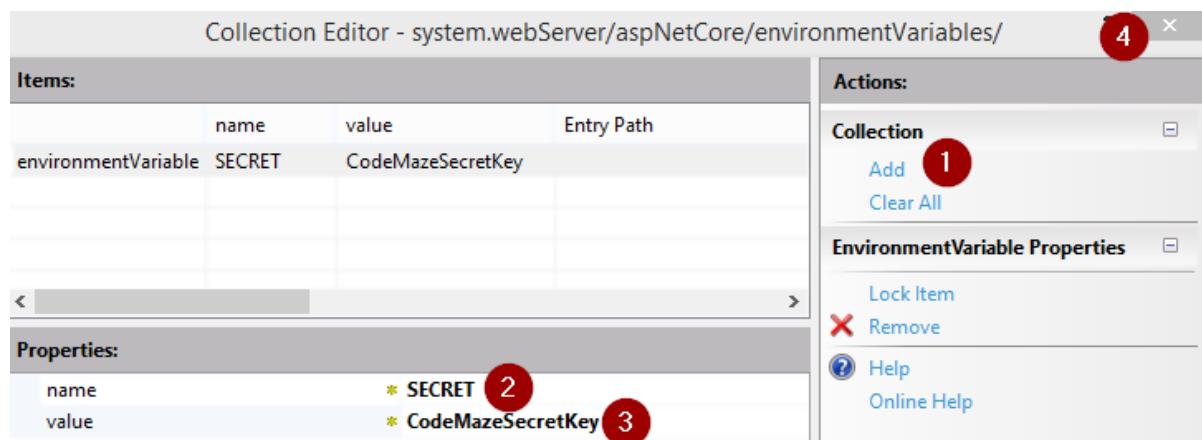
From the “From” combo box, we are going to choose **ApplicationHost.config**:



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.

31.5 Testing Deployed Application

Let's open Postman and send a request for the Root document:



Ultimate ASP.NET Core Web API

The screenshot shows a Postman request configuration for the URL `http://www.companyemployees.codemaze/api`. The method is set to `GET`. The `Headers (6)` tab is selected, showing a single header `Accept` with the value `application/vnd.codemaze.apiroot+json`. The `Temporary Headers (5)` section is collapsed. Below the headers, the `Body` tab is selected, showing the response body in JSON format:

```
[{"rel": "self", "method": "GET", "href": "http://www.companyemployees.codemaze/api"}, {"rel": "companies", "method": "GET", "href": "http://www.companyemployees.codemaze/api/companies"}, {"rel": "create_company", "method": "POST", "href": "http://www.companyemployees.codemaze/api/companies"}]
```

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.

If you get an error message that the `Presentation.xml` file is missing, you can copy it from the project and paste it into the Publish folder. Also, in the Properties window for that file, you can set it to always copy during the publish.

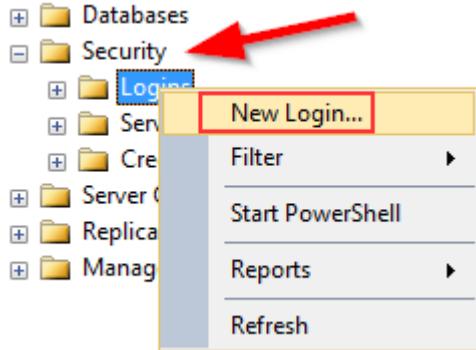
Now, let's continue.

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

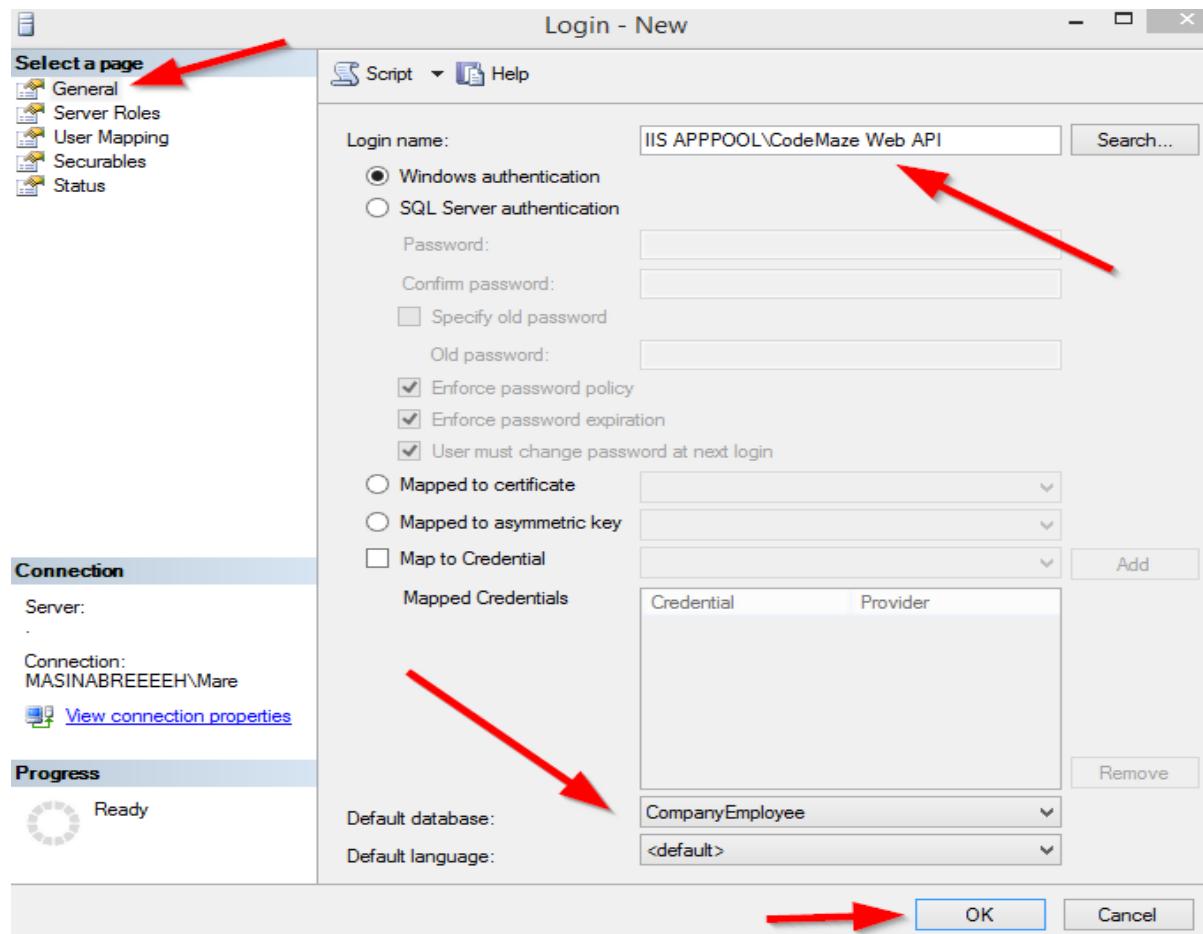


Ultimate ASP.NET Core Web API

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



Ultimate ASP.NET Core Web API

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/login>

POST http://www.companyemployees.codemaze/api/authentication/login Send

Params Auth Headers (10) Body **Pre-req.** Tests Settings Cookies Beautify

raw JSON

```
1 "username": "JDoe",
2 "password": "Password1000"
```

Body Cookies Headers (13) Test Results 200 OK 40 ms 958 B Save Response

Pretty Raw Preview Visualize JSON

```
1 "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcyc54bWxb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjoISK
RvZSIisImh0dHA6Ly9zY2h1bWFzLm1pY3Jvc29mdC5jb20vd3MvMjAwOC8wNi9pZGVudGl0eS9jbGFpbXMvcm9s
ZSI6Ik1hbmFnZXIiLCJleHaiOjE2MzQ4MDkyMDksImlzcyI6IkNvZGVNYXplQVBJIiwiYXVkJioiaHR0cHM6Ly
9sb2Nhbgvhc3Q6NTAwMSJ9.67PRAsuCzY6JYTzvTk--p6422No6vRk1MH3ll6Tvt4",
3 "refreshToken": "ezqruiEtHndgCALgmCv6EeRFpEWdzDTCaH3nH3u4SZ/s="
4 "
```

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>

GET http://www.companyemployees.codemaze/api/companies Send

Params Auth **Headers (8)** Body Pre-req. Tests Settings Cookies

Type Token Bearer Token eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Body Cookies Headers (13) Test Results 200 OK 23 ms 1007 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
3   "name": "Admin_Solutions Ltd Upd2",
4   "fullAddress": "312 Forest Avenue, BF 923 USA"
5 }
```

And there we go. Our API is published and working as expected.



32 BONUS1 - RESPONSE PERFORMANCE IMPROVEMENTS

As mentioned in section 6.1.1, we will show you an alternative way of handling error responses. To repeat, with custom exceptions, we have great control of returning error responses to the client due to the global error handler, which is pretty fast if we use it correctly. Also, the code is pretty clean and straightforward since we don't have to care about the return types and additional validation in the service methods.

Even though some libraries enable us to write custom responses, for example, `OneOf`, we still like to create our abstraction logic, which is tested by us and fast. Additionally, we want to show you the whole creation process for such a flow.

For this example, we will use an existing project from part 6 and modify it to implement our API Response flow.

32.1 Adding Response Classes to the Project

Let's start with the API response model classes.

The first thing we are going to do is create a new **Responses** folder in the **Entities** project. Inside that folder, we are going to add our first class:

```
public abstract class ApiBaseResponse
{
    public bool Success { get; set; }

    protected ApiBaseResponse(bool success) => Success = success;
}
```

This is an abstract class, which will be the main return type for all of our methods where we have to return a successful result or an error result. It also contains a single **Success** property stating whether the action was successful or not.

Now, if our result is successful, we are going to create only one class in the same folder:



```
public sealed class ApiOkResponse<TResult> : ApiBaseResponse
{
    public TResult Result { get; set; }

    public ApiOkResponse(TResult result)
        :base(true)
    {
        Result = result;
    }
}
```

We are going to use this class as a return type for a successful result. It inherits from the **ApiBaseResponse** and populates the **Success** property to true through the constructor. It also contains a single **Result** property of type **TResult**. We will store our concrete result in this property, and since we can have different result types in different methods, this property is a generic one.

That's all regarding the successful responses. Let's move one to the error classes.

For the error responses, we will follow the same structure as we have for the exception classes. So, we will have base abstract classes for `NotFound` or `BadRequest` or any other error responses, and then concrete implementations for these classes like `CompanyNotFound` or `CompanyBadRequest`, etc.

That said, let's use the same folder to create an abstract error class:

```
public abstract class ApiErrorResponse : ApiBaseResponse
{
    public string Message { get; set; }

    public ApiErrorResponse(string message)
        : base(false)
    {
        Message = message;
    }
}
```

This class also inherits from the **ApiBaseResponse**, populates the **Success** property to false, and has a single **Message** property for the error message.



In the same manner, we can create the **ApiBadRequestResponse** class:

```
public abstract class ApiBadRequestResponse : ApiBaseResponse
{
    public string Message { get; set; }

    public ApiBadRequestResponse(string message)
        : base(false)
    {
        Message = message;
    }
}
```

This is the same implementation as the previous one. The important thing to notice is that both of these classes are abstract.

To continue, let's create a concrete error response:

```
public sealed class CompanyNotFoundResponse : ApiNotFoundResponse
{
    public CompanyNotFoundResponse(Guid id)
        : base($"Company with id: {id} is not found in db.")
    {
    }
}
```

The class inherits from the **ApiNotFoundResponse** abstract class, which again inherits from the **ApiBaseResponse** class. It accepts an id parameter and creates a message that sends to the base class.

We are not going to create the **CompanyBadRequestResponse** class because we are not going to need it in our example. But the principle is the same.

32.2 Service Layer Modification

Now that we have the response model classes, we can start with the service layer modification.

Let's start with the **ICompanyService** interface:

```
public interface ICompanyService
{
    ApiBaseResponse GetAllCompanies(bool trackChanges);
    ApiBaseResponse GetCompany(Guid companyId, bool trackChanges);
}
```



We don't return concrete types in our methods anymore. Instead of the `IEnumerable<CompanyDto>` or `CompanyDto` return types, we return the `ApiBaseResponse` type. This will enable us to return either the success result or to return any of the error response results.

After the interface modification, we can modify the `CompanyService` class:

```
public ApiBaseResponse GetAllCompanies(bool trackChanges)
{
    var companies = _repository.Company.GetAllCompanies(trackChanges);

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

    return new ApiOkResponse<IEnumerable<CompanyDto>>(companiesDto);
}

public ApiBaseResponse GetCompany(Guid id, bool trackChanges)
{
    var company = _repository.Company.GetCompany(id, trackChanges);
    if (company is null)
        return new CompanyNotFoundResponse(id);

    var companyDto = _mapper.Map<CompanyDto>(company);
    return new ApiOkResponse<CompanyDto>(companyDto);
}
```

Both method signatures are modified to use `APIBaseResponse`, and also the return types are modified accordingly. Additionally, in the `GetCompany` method, we are not using an exception class to return an error result but the `CompanyNotFoundResponse` class. With the `ApiBaseResponse` abstraction, we are safe to return multiple types from our method as long as they inherit from the `ApiBaseResponse` abstract class. Here you could also log some messages with `_logger`.

One more thing to notice here.

In the `GetAllCompanies` method, we don't have an error response just a successful one. That means we didn't have to implement our API response flow, and we could've left the method unchanged (in the interface and this class). If you want that kind of implementation it is perfectly fine. We



just like consistency in our projects, and due to that fact, we've changed both methods.

32.3 Controller Modification

Before we start changing the actions in the **CompaniesController**, we have to create a way to handle error responses and return them to the client – similar to what we have with the global error handler middleware.

We are not going to create any additional middleware but another controller base class inside the **Presentation/Controllers** folder:

```
public class ApiControllerBase : ControllerBase
{
    public IActionResult ProcessError(ApiBaseResponse baseResponse)
    {
        return baseResponse switch
        {
            ApiNotFoundResponse => NotFound(new ErrorDetails
            {
                Message = ((ApiNotFoundResponse)baseResponse).Message,
                StatusCode = StatusCodes.Status404NotFound
            }),
            ApiBadRequestResponse => BadRequest(new ErrorDetails
            {
                Message = ((ApiBadRequestResponse)baseResponse).Message,
                StatusCode = StatusCodes.Status400BadRequest
            }),
            _ => throw new NotImplementedException()
        };
    }
}
```

This class inherits from the **ControllerBase** class and implements a single **ProcessError** action accepting an **ApiBaseResponse** parameter. Inside the action, we are inspecting the type of the sent parameter, and based on that type we return an appropriate message to the client. A similar thing we did in the exception middleware class.

If you add additional error response classes to the Response folder, you only have to add them here to process the response for the client. Additionally, this is where we can see the advantage of our abstraction approach.



Now, we can modify our **CompaniesController**:

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

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

    [HttpGet]
    public IActionResult GetCompanies()
    {
        var baseResult = _service.CompanyService.GetAllCompanies(trackChanges:
false);

        var companies =
((ApiOkResponse<IEnumerable<CompanyDto>>)baseResult).Result;

        return Ok(companies);
    }

    [HttpGet("{id:guid}")]
    public IActionResult GetCompany(Guid id)
    {
        var baseResult = _service.CompanyService.GetCompany(id, trackChanges:
false);
        if (!baseResult.Success)
            return ProcessError(baseResult);

        var company = ((ApiOkResponse<CompanyDto>)baseResult).Result;

        return Ok(company);
    }
}
```

Now our controller inherits from the **ApiControllerBase**, which inherits from the **ControllerBase** class. In the **GetCompanies** action, we extract the result from the service layer and cast the **baseResult** variable to the concrete **ApiOkResponse** type, and use the **Result** property to extract our required result of type **IEnumerable<CompanyDto>**.

We do a similar thing for the **GetCompany** action. Of course, here we check if our result is successful and if it's not, we return the result of the **ProcessError** method.

And that's it.



We can leave the solution as is, but we mind having these castings inside our actions – they can be moved somewhere else making them reusable and our actions cleaner. So, let's do that.

In the same project, we are going to create a new Extensions folder and a new **ApiBaseResponseExtensions** class:

```
public static class ApiBaseResponseExtensions
{
    public static TResultType GetResult<TResultType>(this ApiBaseResponse
apiBaseResponse) =>
        ((ApiOkResponse<TResultType>)apiBaseResponse).Result;
}
```

The **GetResult** method will extend the **ApiBaseResponse** type and return the result of the required type.

Now, we can modify actions inside the controller:

```
[HttpGet]
public IActionResult GetCompanies()
{
    var baseResult = _service.CompanyService.GetAllCompanies(trackChanges: false);

    var companies = baseResult.GetResult<IEnumerable<CompanyDto>>();

    return Ok(companies);
}

[HttpGet("{id:guid}")]
public IActionResult GetCompany(Guid id)
{
    var baseResult = _service.CompanyService.GetCompany(id, trackChanges: false);
    if (!baseResult.Success)
        return ProcessError(baseResult);

    var company = baseResult.GetResult<CompanyDto>();

    return Ok(company);
}
```

This is much cleaner and easier to read and understand.

32.4 Testing the API Response Flow

Now we can start our application, open Postman, and send some requests.

Let's try to get all the companies:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies>

```
1 [
2   {
3     "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
4     "name": "Admin_Solutions Ltd Upd2",
5     "fullAddress": "312 Forest Avenue, BF 923 USA"
6   },
7   {
8     "id": "a216fbbe-ebbd-4e09-a2a2-08d988ca3ca9",
9     "name": "Branding Ltd",
10    "fullAddress": "255 Main Street, K 334 USA"
11  }
]
```

Then, we can try to get a single company:

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

```
1 [
2   {
3     "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
4     "name": "Admin_Solutions Ltd Upd2",
5     "fullAddress": "312 Forest Avenue, BF 923 USA"
6   }
]
```

And finally, let's try to get a company that does not exist:

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

```
1 [
2   {
3     "statusCode": 404,
4     "message": "Company with id: 3d490a70-94ce-4d15-9494-5248280c2ce2 is not found in db."
5   }
]
```

And we have our response with a proper status code and response body.

Excellent.



We have a solution that is easy to implement, fast, and extendable.

Our suggestion is to go with custom exceptions since they are easier to implement and fast as well. But if you have an app flow where you have to return error responses at a much higher rate and thus maybe impact the app's performance, the API Response flow is the way to go.



33 BONUS 2 - INTRODUCTION TO CQRS AND MEDIATOR WITH ASP.NET CORE WEB API

In this chapter, we will provide an introduction to the CQRS pattern and how the .NET library MediatR helps us build software with this architecture.

In the **Source Code** folder, you will find the folder for this chapter with two folders inside – **start** and **end**. In the **start** folder, you will find a prepared project for this section. We are going to use it to explain the implementation of CQRS and MediatR. We have used the existing project from one of the previous chapters and removed the things we don't need or want to replace - like the service layer.

In the **end** folder, you will find a finished project for this chapter.

33.1 About CQRS and Mediator Pattern

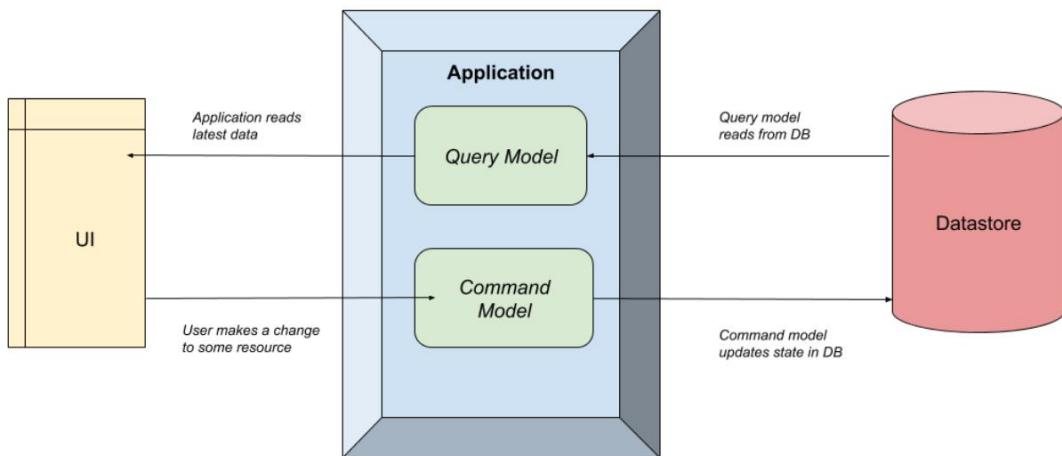
The MediatR library was built to facilitate two primary software architecture patterns: CQRS and the Mediator pattern. Whilst similar, let's spend a moment understanding the principles behind each pattern.

33.1.1 CQRS

CQRS stands for “Command Query Responsibility Segregation”. As the acronym suggests, it’s all about splitting the responsibility of commands (saves) and queries (reads) into different models.

If we think about the commonly used CRUD pattern (Create-Read-Update-Delete), we usually have the user interface interacting with a datastore responsible for all four operations. CQRS would instead have us split these operations into two models, one for the queries (aka “R”), and another for the commands (aka “CUD”).

The following image illustrates how this works:



The Application simply separates the query and command models.

The **CQRS pattern makes no formal requirements of how this separation occurs**. It could be as simple as a separate class in the same application (as we'll see shortly with MediatR), all the way up to separate physical applications on different servers. That decision would be based on factors such as scaling requirements and infrastructure, so we won't go into that decision path here.

The key point being is that to create a CQRS system, we just need to **split the reads from the writes**.

What problem is this trying to solve?

Well, a common reason is when we design a system, we start with data storage. We perform database normalization, add primary and foreign keys to enforce referential integrity, add indexes, and generally ensure the "write system" is optimized. This is a common setup for a relational database such as SQL Server or MySQL. Other times, we think about the read use cases first, then try and add that into a database, worrying less about duplication or other relational DB concerns (often "document databases" are used for these patterns).

Neither approach is wrong. But the issue is that it's a constant balancing act between reads and writes, and eventually one side will "win out". All



further development means both sides need to be analyzed, and often one is compromised.

CQRS allows us to “break free” from these considerations and give each system the equal design and consideration it deserves without worrying about the impact of the other system. This has tremendous benefits on both performance and agility, especially if separate teams are working on these systems.

33.1.2 Advantages and Disadvantages of CQRS

The benefits of CQRS are:

- Single Responsibility – Commands and Queries have only one job. It is either to change the state of the application or retrieve it. Therefore, they are very easy to reason about and understand.
- Decoupling – The Command or Query is completely decoupled from its handler, giving you a lot of flexibility on the handler side to implement it the best way you see fit.
- Scalability – The CQRS pattern is very flexible in terms of how you can organize your data storage, giving you options for great scalability. You can use one database for both Commands and Queries. You can use separate Read/Write databases, for improved performance, with messaging or replication between the databases for synchronization.
- Testability – It is very easy to test Command or Query handlers since they will be very simple by design, and perform only a single job.

Of course, it can't all be good. Here are some of the disadvantages of CQRS:

- Complexity – CQRS is an advanced design pattern, and it will take you time to fully understand it. It introduces a lot of complexity that

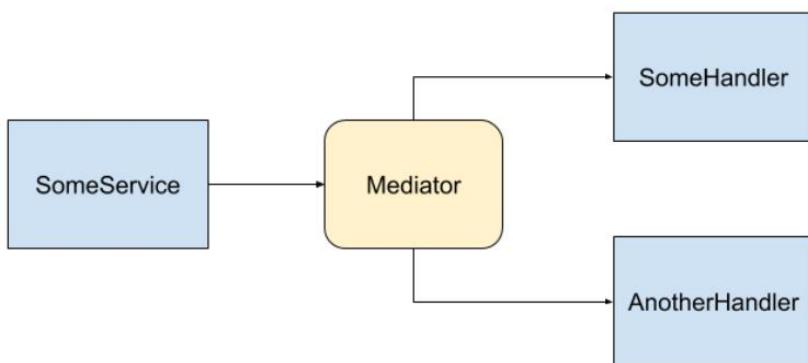


will create friction and potential problems in your project. Be sure to consider everything, before deciding to use it in your project.

- Learning Curve – Although it seems like a straightforward design pattern, there is still a learning curve with CQRS. Most developers are used to the procedural (imperative) style of writing code, and CQRS is a big shift away from that.
- Hard to Debug – Since Commands and Queries are decoupled from their handler, there isn't a natural imperative flow of the application. This makes it harder to debug than traditional applications.

33.1.3 Mediator Pattern

The Mediator pattern is simply defining an object that encapsulates how objects interact with each other. Instead of having two or more objects take a direct dependency on each other, they instead interact with a “mediator”, who is in charge of sending those interactions to the other party:



In this image, **SomeService** sends a message to the Mediator, and the Mediator then invokes multiple services to handle the message. There is no direct dependency between any of the blue components.

The reason the Mediator pattern is useful is the same reason patterns like Inversion of Control are useful. It enables “loose coupling”, as the dependency graph is minimized and therefore code is simpler and easier



to test. In other words, the fewer considerations a component has, the easier it is to develop and evolve.

We saw in the previous image how the services have no direct dependency, and the producer of the messages doesn't know who or how many things are going to handle it. This is very similar to how a message broker works in the "publish/subscribe" pattern. If we wanted to add another handler we could, and the producer wouldn't have to be modified.

Now that we've been over some theory, let's talk about how MediatR makes all these things possible.

33.2 How MediatR facilitates CQRS and Mediator Patterns

You can think of MediatR as an "in-process" Mediator implementation, that helps us build CQRS systems. All communication between the user interface and the data store happens via MediatR.

The term "in process" is an important limitation here. Since it's a .NET library that manages interactions within classes on the same process, it's not an appropriate library to use if we want to separate the commands and queries across two systems. A better approach would be to use a message broker such as Kafka or Azure Service Bus.

However, for this chapter, we are going to stick with a simple single-process CQRS system, so MediatR fits the bill perfectly.

33.3 Adding Application Project and Initial Configuration

Let's start by opening the starter project from the **start** folder. You will see that we don't have the Service nor the Service.Contracts projects. Well, we don't need them. We are going to use CQRS with MediatR to replace that part of our solution.

But, we do need an additional project for our business logic so, let's create a new class library (.NET Core) and name it **Application**.



Additionally, we are going to add a new class named **AssemblyReference**. We will use it for the same purpose as we used the class with the same name in the **Presentation** project:

```
public static class AssemblyReference
{
}
```

Now let's install a couple of packages.

The first package we are going to install is the **MediatR** in the **Application** project:

```
PM> install-package MediatR
```

Then in the main project, we are going to install another package that wires up MediatR with the ASP.NET dependency injection container:

```
PM> install-package MediatR.Extensions.Microsoft.DependencyInjection
```

After the installations, we are going to configure **MediatR** in the **Program** class:

```
builder.Services.AddMediatR(typeof(Application.AssemblyReference).Assembly);
```

For this, we have to reference the **Application** project, and add a using directive:

```
using MediatR;
```

The **AddMediatR** method will scan the project assembly that contains the handlers that we are going to use to handle our business logic. Since we are going to place those handlers in the **Application** project, we are using the Application's assembly as a parameter.

Before we continue, we have to reference the **Application** project from the **Presentation** project.

Now MediatR is configured, and we can use it inside our controller.



In the Controllers folder of the **Presentation** project, we are going to find a single controller class. It contains only a base code, and we are going to modify it by adding a sender through the constructor injection:

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

    public CompaniesController(ISender sender) => _sender = sender;
}
```

Here we inject the **ISender** interface from the MediatR namespace. We are going to use this interface to send requests to our handlers.

We have to mention one thing about using **ISender** and not the **IMediator** interface. From the MediatR version 9.0, the **IMediator** interface is split into two interfaces:

```
public interface ISender
{
    Task<TResponse> Send<TResponse>(IRequest<TResponse> request, CancellationToken cancellationToken = default);
    Task<object?> Send(object request, CancellationToken cancellationToken = default);
}
public interface IPublisher
{
    Task Publish(object notification, CancellationToken cancellationToken = default);
    Task Publish<TNotification>(TNotification notification, CancellationToken cancellationToken = default)
        where TNotification : INotification;
}
public interface IMediator : ISender, IPublisher
{}
```

So, by looking at the code, it is clear that you can continue using the **IMediator** interface to send requests and publish notifications. But it is recommended to split that by using **ISender** and **IPublisher** interfaces.

With that said, we can continue with the Application's logic implementation.

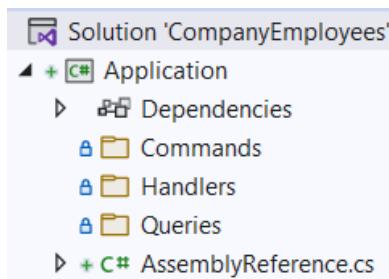


33.4 Requests with MediatR

MediatR Requests are simple request-response style messages where a single request is synchronously handled by a single handler (synchronous from the request point of view, not C# internal `async/await`). Good use cases here would be returning something from a database or updating a database.

There are two types of requests in MediatR. One that returns a value, and one that doesn't. Often this corresponds to reads/queries (returning a value) and writes/commands (usually doesn't return a value).

So, before we start sending requests, we are going to create several folders in the **Application** project to separate queries, commands, and handlers:



Since we are going to work only with the company entity, we are going to place our queries, commands, and handlers directly into these folders. But in larger projects with multiple entities, we can create additional folders for each entity inside each of these folders for better organization.

Also, as we already know, we are not going to send our entities as a result to the client but DTOs, so we have to reference the **Shared** project.

That said, let's start with our first query. Let's create it in the **Queries** folder:

```
public sealed record GetCompaniesQuery(bool TrackChanges) :  
    IRequest<IEnumerable<CompanyDto>>;
```



Here, we create the **GetCompaniesQuery** record, which implements **IRequest<IEnumerable<CompanyDto>>**. This simply means our request will return a list of companies.

Here we need two additional namespaces:

```
using MediatR;
using Shared.DataTransferObjects;
```

Once we send the request from our controller's action, we are going to see the usage of this query.

After the query, we need a handler. This handler in simple words will be our replacement for the service layer method that we had in our project. In our previous project, all the service classes were using the repository to access the database – we will make no difference here. For that, we have to reference the **Contracts** project so we can access the **IRepositoryManager** interface.

After adding the reference, we can create a new **GetCompaniesHandler** class in the **Handlers** folder:

```
internal sealed class GetCompaniesHandler : IRequestHandler<GetCompaniesQuery,
IEnumerable<CompanyDto>>
{
    private readonly IRepositoryManager _repository;

    public GetCompaniesHandler(IRepositoryManager repository) => _repository =
repository;

    public Task<IEnumerable<CompanyDto>> Handle(GetCompaniesQuery request,
CancellationToken cancellationToken)
    {
        throw new NotImplementedException();
    }
}
```

Our handler inherits from **IRequestHandler<GetCompaniesQuery, IEnumerable<Product>>**. This means this class will handle **GetCompaniesQuery**, in this case, returning the list of companies.



We also inject the repository through the constructor and add a default implementation of the **Handle** method, required by the **IRequestHandler** interface.

These are the required namespaces:

```
using Application.Queries;
using Contracts;
using MediatR;
using Shared.DataTransferObjects;
```

Of course, we are not going to leave this method to throw an exception. But before we add business logic, we have to install **AutoMapper** in the Application project:

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

Register the package in the Program class:

```
builder.Services.AddMediatR(typeof(Application.AssemblyReference).Assembly);
builder.Services.AddAutoMapper(typeof(Program));
```

And create the **MappingProfile** class, also in the main project, with a single mapping rule:

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

Everything with these actions is familiar since we've already used AutoMapper in our project.

Now, we can modify the handler class:

```
internal sealed class GetCompaniesHandler : IRequestHandler<GetCompaniesQuery,
IEnumerable<CompanyDto>>
{
    private readonly IRepositoryManager _repository;
    private readonly IMapper _mapper;

    public GetCompaniesHandler(IRepositoryManager repository, IMapper mapper)
    {
```



```
        _repository = repository;
        _mapper = mapper;
    }

    public async Task<IEnumerable<CompanyDto>> Handle(GetCompaniesQuery request,
        CancellationToken cancellationToken)
    {
        var companies = await
_repository.Company.GetAllCompaniesAsync(request.TrackChanges);

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

        return companiesDto;
    }
}
```

This logic is also familiar since we had almost the same one in our **GetAllCompaniesAsync** service method. One difference is that we are passing the track changes parameter through the request object.

Now, we can modify **CompaniesController**:

```
[HttpGet]
public async Task<IActionResult> GetCompanies()
{
    var companies = await _sender.Send(new GetCompaniesQuery(TrackChanges: false));

    return Ok(companies);
}
```

We use the **Send** method to send a request to our handler and pass the **GetCompaniesQuery** as a parameter. Nothing more than that.

We also need an additional namespace:

```
using Application.Queries;
```

Our controller is clean as it was with the service layer implemented. But this time, we don't have a single service class to handle all the methods but a single handler to take care of only one thing.

Now, we can test this:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies>

The screenshot shows a Postman request to `https://localhost:5001/api/companies`. The response is a 200 OK status with 17 ms latency and 638 B size. The response body is a JSON array containing two company objects:

```
1
2 {
3     "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
4     "name": "Admin_Solutions Ltd Upd2",
5     "fullAddress": "312 Forest Avenue, BF 923 USA"
6 },
7 {
8     "id": "a216fbbe-ebbd-4e09-a2a2-08d988ca3ca9",
9     "name": "Branding Ltd",
10    "fullAddress": "255 Main Street, K 334 USA"
11 }
```

Everything works great.

With this in mind, we can continue and implement the logic for fetching a single company.

So, let's start with the query in the **Queries** folder:

```
public sealed record GetCompanyQuery(Guid Id, bool TrackChanges) :  
IRequest<CompanyDto>;
```

Then, let's implement a new handler:

```
internal sealed class GetCompanyHandler : IRequestHandler<GetCompanyQuery, CompanyDto>  
{  
    private readonly IRepositoryManager _repository;  
    private readonly IMapper _mapper;  
  
    public GetCompanyHandler(IRepositoryManager repository, IMapper mapper)  
    {  
        _repository = repository;  
        _mapper = mapper;  
    }  
  
    public async Task<CompanyDto> Handle(GetCompanyQuery request, CancellationToken cancellationToken)  
    {  
        var company = await _repository.Company.GetCompanyAsync(request.Id,  
request.TrackChanges);  
        if (company is null)  
            throw new CompanyNotFoundException(request.Id);  
  
        var companyDto = _mapper.Map<CompanyDto>(company);  
  
        return companyDto;  
    }  
}
```



```
}
```

So again, our handler inherits from the **IRequestHandler** interface accepting the query as the first parameter and the result as the second. Then, we inject the required services and familiarly implement the **Handle** method.

We need these namespaces here:

```
using Application.Queries;
using AutoMapper;
using Contracts;
using Entities.Exceptions;
using MediatR;
using Shared.DataTransferObjects;
```

Lastly, we have to add another action in **CompaniesController**:

```
[HttpGet("{id:guid}", Name = "CompanyById")]
public async Task<IActionResult> GetCompany(Guid id)
{
    var company = await _sender.Send(new GetCompanyQuery(id, TrackChanges: false));
    return Ok(company);
}
```

Awesome, let's test it:

The screenshot shows a Postman request to `https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3`. The response status is 200 OK with a response time of 13 ms and a body size of 273 B. The response body is a JSON object:

```
1  {
2      "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
3      "name": "Admin_Solutions Ltd Upd2",
4      "fullAddress": "312 Forest Avenue, BF 923 USA"
5 }
```

Excellent, we can see the company DTO in the response body.

Additionally, we can try an invalid request:



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

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

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

Body Cookies Headers (7) Test Results 404 Not Found 1088 ms 329 B Save Response

Pretty Raw Preview Visualize JSON

```
1 "StatusCode": 404,
2 "Message": "The company with id: 3d490a70-94ce-4d15-9494-5248280c2ce2 doesn't exist in the
3   database."
4
```

And, we can see this works as well.

33.5 Commands with MediatR

As with both queries, we are going to start with a command record creation inside the **Commands** folder:

```
public sealed record CreateCompanyCommand(CompanyForCreationDto Company) :  
IRequest<CompanyDto>;
```

Our command has a single parameter sent from the client, and it inherits from **IRequest<CompanyDto>**. Our request has to return **CompanyDto** because we will need it, in our action, to create a valid route in the return statement.

After the query, we are going to create another handler:

```
internal sealed class CreateCompanyHandler : IRequestHandler<CreateCompanyCommand,  
CompanyDto>  
{  
    private readonly IRepositoryManager _repository;  
    private readonly IMapper _mapper;  
  
    public CreateCompanyHandler(IRepositoryManager repository, IMapper mapper)  
    {  
        _repository = repository;  
        _mapper = mapper;  
    }  
  
    public async Task<CompanyDto> Handle(CreateCompanyCommand request,  
CancellationToken cancellationToken)  
    {  
        var companyEntity = _mapper.Map<Company>(request.Company);  
  
        _repository.Company.CreateCompany(companyEntity);  
        await _repository.SaveAsync();  
    }  
}
```



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

        return companyToReturn;
    }
}
```

So, we inject our services and implement the Handle method as we did with the service method. We map from the creation DTO to the entity, save it to the database, and map it to the company DTO object.

Then, before we add a new mapping rule in the **MappingProfile** class:

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

Now, we can add a new action in a controller:

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

    var company = await _sender.Send(new
CreateCompanyCommand(companyForCreationDto));

    return CreatedAtRoute("CompanyId", new { id = company.Id }, company);
}
```

That's all it takes. Now we can test this:

The screenshot shows a Postman request to `https://localhost:5001/api/companies` using the `POST` method. The `Body` tab is selected, showing the following JSON payload:

```
1 "name": "CQRS Test Company",
2 "address": "CQRS Test Address",
3 "country": "CQRS Test Country"
```

The response tab shows a `201 Created` status with a response body containing the newly created company's details:

```
1 {
2   "id": "7aea16e2-74b9-4fd9-c22a-08d9961aa2d5",
3   "name": "CQRS Test Company",
4   "fullAddress": "CQRS Test Address CQRS Test Country"
5 }
```

A new company is created, and if we inspect the Headers tab, we are going to find the link to fetch this new company:



Body	Cookies	Headers (5)	Test Results	Save Response
			201 Created 54 ms 362 B	

KEY	VALUE
Content-Type ⓘ	application/json; charset=utf-8
Date ⓘ	Sat, 23 Oct 2021 11:46:22 GMT
Server ⓘ	Kestrel
Location ⓘ	https://localhost:5001/api/companies/7aea16e2-74b9-4fd9-c22a-08d9961aa2d5
Transfer-Encoding ⓘ	chunked

There is one important thing we have to understand here. We are communicating to a datastore via simple message constructs without having any idea on how it's being implemented. The commands and queries could be pointing to different data stores. They don't *know* how their request will be handled, and they don't *care*.

33.5.1 Update Command

Following the same principle from the previous example, we can implement the update request.

Let's start with the command:

```
public sealed record UpdateCompanyCommand  
    (Guid Id, CompanyForUpdateDto Company, bool TrackChanges) : IRequest;
```

This time our command inherits from **IRequest** without any generic parameter. That's because we are not going to return any value with this request.

Let's continue with the handler implementation:

```
internal sealed class UpdateCompanyHandler : IRequestHandler<UpdateCompanyCommand, Unit>  
{  
    private readonly IRepositoryManager _repository;  
    private readonly IMapper _mapper;  
  
    public UpdateCompanyHandler(IRepositoryManager repository, IMapper mapper)  
    {  
        _repository = repository;  
        _mapper = mapper;  
    }  
  
    public async Task<Unit> Handle(UpdateCompanyCommand request, CancellationToken cancellationToken)  
    {
```



```
        var companyEntity = await  
_repository.Company.GetCompanyAsync(request.Id, request.TrackChanges);  
        if (companyEntity is null)  
            throw new CompanyNotFoundException(request.Id);  
  
        _mapper.Map(request.Company, companyEntity);  
        await _repository.SaveAsync();  
  
        return Unit.Value;  
    }  
}
```

This handler inherits from **IRequestHandler<UpdateCompanyCommand, Unit>**. This is new for us because the first time our command is not returning any value. But **IRequestHandler** always accepts two parameters (**TRequest** and **TResponse**). So, we provide the **Unit** structure for the **TResponse** parameter since it represents the **void** type.

Then the **Handle** implementation is familiar to us except for the return part. We have to return something from the **Handle** method and we use **Unit.Value**.

Before we modify the controller, we have to add another mapping rule:

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

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

```
[HttpPut("{id:guid}")]  
public async Task<IActionResult> UpdateCompany(Guid id, CompanyForUpdateDto  
companyForUpdateDto)  
{  
    if (companyForUpdateDto is null)  
        return BadRequest("CompanyForUpdateDto object is null");  
  
    await _sender.Send(new UpdateCompanyCommand(id, companyForUpdateDto,  
TrackChanges: true));  
  
    return NoContent();  
}
```

At this point, we can send a PUT request from Postman:



<https://localhost:5001/api/companies/7aea16e2-74b9-4fd9-c22a-08d9961aa2d5>

PUT

https://localhost:5001/api/companies/7aea16e2-74b9-4fd9-c22a-08d9961aa2d5

Send

Params Auth Headers (9) Body ● Pre-req. Tests Settings

Cookies

raw ▼ JSON ▼

Beautify

```
1  ↴ "name": "Updated CQRS Test Company",
2  ↴ "address": "CQRS Test Address",
3  ↴ "country": "CQRS Test Country"
4
5
```

Body Cookies Headers (2) Test Results

🌐 204 No Content 283 ms 81 B Save Response ▼

There is the 204 status code.

If you fetch this company, you will find the name updated for sure.

33.5.2 Delete Command

After all of this implementation, this one should be pretty straightforward.

Let's start with the command:

```
public record DeleteCompanyCommand(Guid Id, bool TrackChanges) : IRequest;
```

Then, let's continue with a handler:

```
internal sealed class DeleteCompanyHandler : IRequestHandler<DeleteCompanyCommand, Unit>
{
    private readonly IRepositoryManager _repository;

    public DeleteCompanyHandler(IRepositoryManager repository) => _repository = repository;

    public async Task<Unit> Handle(DeleteCompanyCommand request, CancellationToken cancellationToken)
    {
        var company = await _repository.Company.GetCompanyAsync(request.Id, request.TrackChanges);
        if (company is null)
            throw new CompanyNotFoundException(request.Id);

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

        return Unit.Value;
    }
}
```

Finally, let's add one more action inside the controller:

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



```
public async Task<IActionResult> DeleteCompany(Guid id)
{
    await _sender.Send(new DeleteCompanyCommand(id, TrackChanges: false));

    return NoContent();
}
```

That's it. Pretty easy.

We can test this now:

<https://localhost:5001/api/companies/7aea16e2-74b9-4fd9-c22a-08d9961aa2d5>

The screenshot shows the Postman interface. A DELETE request is being sent to the URL <https://localhost:5001/api/companies/7aea16e2-74b9-4fd9-c22a-08d9961aa2d5>. In the 'Params' tab, there is a single entry: 'Key' with a value of 'Value'. The response status is 204 No Content, with a duration of 476 ms and a size of 81 B. The 'Save Response' button is visible.

It works great.

Now that we know how to work with requests using MediatR, let's see how to use notifications.

33.6 MediatR Notifications

So far we've only seen a single request being handled by a single handler. However, what if we want to handle a single request by multiple handlers?

That's where notifications come in. In these situations, we usually have multiple independent operations that need to occur after some event.

Examples might be:

- Sending an email
- Invalidating a cache
- ...



To demonstrate this, we will update the delete company flow we created previously to publish a notification and have it handled by two handlers.

Sending an email is out of the scope of this book (you can learn more about that in our Bonus 6 Security book). But to demonstrate the behavior of notifications, we will use our logger service and log a message as if the email was sent.

So, the flow will be - once we delete the Company, we want to inform our administrators with an email message that the delete has action occurred.

That said, let's start by creating a new **Notifications** folder inside the **Application** project and add a new notification in that folder:

```
public sealed record CompanyDeletedNotification(Guid Id, bool TrackChanges) :  
INotification;
```

The notification has to inherit from the **INotification** interface. This is the equivalent of the **IRequest** we saw earlier, but for Notifications.

As we can conclude, notifications don't return a value. They work on the fire and forget principle, like publishers.

Next, we are going to create a new Emailhandler class:

```
internal sealed class EmailHandler : INotificationHandler<CompanyDeletedNotification>  
{  
    private readonly ILoggerManager _logger;  
  
    public EmailHandler(ILoggerManager logger) => _logger = logger;  
  
    public async Task Handle(CompanyDeletedNotification notification,  
CancellationToken cancellationToken)  
    {  
        _logger.LogWarning($"Delete action for the company with id:  
{notification.Id} has occurred.");  
  
        await Task.CompletedTask;  
    }  
}
```

Here, we just simulate sending our email message in an async manner. Without too many complications, we use our logger service to process the message.



Let's continue by modifying the `DeleteCompanyHandler` class:

```
internal sealed class DeleteCompanyHandler :  
INotificationHandler<CompanyDeletedNotification>  
{  
    private readonly IRepositoryManager _repository;  
  
    public DeleteCompanyHandler(IRepositoryManager repository) => _repository =  
repository;  
  
    public async Task Handle(CompanyDeletedNotification notification,  
CancellationToken cancellationToken)  
    {  
        var company = await _repository.Company.GetCompanyAsync(notification.Id,  
notification.TrackChanges);  
        if (company is null)  
            throw new CompanyNotFoundException(notification.Id);  
  
        _repository.Company.DeleteCompany(company);  
        await _repository.SaveAsync();  
    }  
}
```

This time, our handler inherits from the **INotificationHandler** interface, and it doesn't return any value – we've modified the method signature and removed the return statement.

Finally, we have to modify the controller's constructor:

```
private readonly ISender _sender;  
private readonly IPublisher _publisher;  
  
public CompaniesController(ISender sender, IPublisher publisher)  
{  
    _sender = sender;  
    _publisher = publisher;  
}
```

We inject another interface, which we are going to use to publish notifications.

And, we have to modify the `DeleteCompany` action:

```
[HttpDelete("{id:guid}")]  
public async Task<IActionResult> DeleteCompany(Guid id)  
{  
    await _publisher.Publish(new CompanyDeletedNotification(id, TrackChanges:  
false));  
  
    return NoContent();  
}
```

To test this, let's create a new company first:

[Pretty](#)[Raw](#)[Preview](#)[Visualize](#)[JSON](#) ▾

```
1  {
2      "id": "e06089af-baeb-44ef-1fdf-08d99630e212",
3      "name": "CQRS Test Company",
4      "fullAddress": "CQRS Test Address CQRS Test Country"
5  }
```

Now, if we send the Delete request, we are going to receive the 204 NoContent response:

<https://localhost:5001/api/companies/e06089af-baeb-44ef-1fdf-08d99630e212>

[DELETE](#) ▾<https://localhost:5001/api/companies/e06089af-baeb-44ef-1fdf-08d99630e212>[Send](#) ▾[Params](#) [Auth](#) [Headers \(7\)](#) [Body](#) [Pre-req.](#) [Tests](#) [Settings](#)[Cookies](#)[Body](#) [Cookies](#) [Headers \(2\)](#) [Test Results](#)🌐 204 No Content 469 ms 81 B [Save Response](#) ▾

And also, if we inspect the logs, we will find a new logged message stating that the delete action has occurred:

2021-10-23 16:31:07.9674 WARN Delete action for the company with id: e06089af-baeb-44ef-1fdf-08d99630e212 has occurred.

33.7 MediatR Behaviors

Often when we build applications, we have many cross-cutting concerns. These include authorization, validating, and logging.

Instead of repeating this logic throughout our handlers, we can make use of Behaviors. Behaviors are very similar to ASP.NET Core middleware in that they accept a request, perform some action, then (optionally) pass along the request.

In this section, we are going to use behaviors to perform validation on the DTOs that come from the client.

As we have already learned in chapter 13, we can perform the validation by using data annotations attributes and the ModelState dictionary. Then



we can extract the validation logic into action filters to clear our actions. Well, we can apply all of that to our current solution as well.

But, some developers have a preference for using fluent validation over data annotation attributes. In that case, behaviors are the perfect place to execute that validation logic.

So, let's go step by step and add the fluent validation in our project first and then use behavior to extract validation errors if any, and return them to the client.

33.7.1 Adding Fluent Validation

The FluentValidation library allows us to easily define very rich custom validation for our classes. Since we are implementing CQRS, it makes the most sense to define validation for our Commands. We should not bother ourselves with defining validators for Queries, since they don't contain any behavior. We use Queries only for fetching data from the application.

So, let's start by installing the FluentValidation package in the Application project:

```
PM> install-package FluentValidation.AspNetCore
```

The FluentValidation.AspNetCore package installs both FluentValidation and FluentValidation.DependencyInjectionExtensions packages.

After the installation, we are going to register all the validators inside the service collection by modifying the Program class:

```
builder.Services.AddMediatR(typeof(Application.AssemblyReference).Assembly);
builder.Services.AddAutoMapper(typeof(Program));
builder.Services.AddValidatorsFromAssembly(typeof(Application.AssemblyReference).Assembly);
```

Then, let's create a new **Validators** folder inside the **Application** project and add a new class inside:

```
public sealed class CreateCompanyCommandValidator : AbstractValidator<CreateCompanyCommand>
{
```



```
public CreateCompanyCommandValidator()
{
    RuleFor(c => c.Company.Name).NotEmpty().MaximumLength(60);

    RuleFor(c => c.Company.Address).NotEmpty().MaximumLength(60);
}

}
```

The following using directives are necessary for this class:

```
using Application.Commands;
using FluentValidation;
```

We create the **CreateCompanyCommandValidator** class that inherits from the **AbstractValidator<T>** class, specifying the type **CreateCompanyCommand**. This lets FluentValidation know that this validation is for the **CreateCompanyCommand** record. Since this record contains a parameter of type **CompanyForCreationDto**, which is the object that we have to validate since it comes from the client, we specify the rules for properties from that DTO.

The **NotEmpty** method specifies that the property can't be null or empty, and the **MaximumLength** method specifies the maximum string length of the property.

33.7.2 Creating Decorators with MediatR PipelineBehavior

The CQRS pattern uses Commands and Queries to convey information, and receive a response. In essence, it represents a request-response pipeline. This gives us the ability to easily introduce additional behavior around each request that is going through the pipeline, without actually modifying the original request.

You may be familiar with this technique under the name Decorator pattern. Another example of using the Decorator pattern is the ASP.NET Core Middleware concept, which we talked about in section 1.8.

MediatR has a similar concept to middleware, and it is called IPipelineBehavior:



```
public interface IPipelineBehavior<in TRequest, TResponse> where TRequest : notnull
{
    Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken,
    RequestHandlerDelegate<TResponse> next);
}
```

The pipeline behavior is a wrapper around a request instance and gives us a lot of flexibility with the implementation. Pipeline behaviors are a good fit for cross-cutting concerns in your application. Good examples of cross-cutting concerns are logging, caching, and of course, validation!

Before we use this interface, let's create a new exception class in the **Entities/Exceptions** folder:

```
public sealed class ValidationAppException : Exception
{
    public IReadOnlyDictionary<string, string[]> Errors { get; }

    public ValidationAppException(IReadOnlyDictionary<string, string[]> errors)
        :base("One or more validation errors occurred")
        => Errors = errors;
}
```

Next, to implement the **IPipelineBehavior** interface, we are going to create another folder named **Behaviors** in the **Application** project, and add a single class inside it:

```
public sealed class ValidationBehavior<TRequest, TResponse> :
IPipelineBehavior<TRequest, TResponse>
    where TRequest : IRequest<TResponse>
{
    private readonly IEnumerable<IValidator<TRequest>> _validators;

    public ValidationBehavior(IEnumerable<IValidator<TRequest>> validators) =>
    _validators = validators;

    public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken,
    RequestHandlerDelegate<TResponse> next)
    {
        if (!_validators.Any())
            return await next();

        var context = new ValidationContext<TRequest>(request);

        var errorsDictionary = _validators
            .Select(x => x.Validate(context))
            .SelectMany(x => x.Errors)
            .Where(x => x != null)
            .GroupBy(
                x => x.PropertyName.Substring(x.PropertyName.IndexOf('.') + 1),
                x => x.ErrorMessage,
```



```
(propertyName, errorMessages) => new
{
    Key = propertyName,
    Values = errorMessages.Distinct().ToArray()
})
.ToDictionary(x => x.Key, x => x.Values);

if (errorsDictionary.Any())
    throw new ValidationAppException(errorsDictionary);

return await next();
}
}
```

This class has to inherit from the `IPipelineBehavior` interface and implement the `Handler` method. We also inject a collection of **IValidator** implementations in the constructor. The FluentValidation library will scan our project for all **AbstractValidator** implementations for a given type and then provide us with the instance at runtime. It is how we can apply the actual validators that we implemented in our project.

Then, if there are no validation errors, we just call the `next` delegate to allow the execution of the next component in the middleware.

But if there are any errors, we extract them from the **_validators** collection and group them inside the dictionary. If there are entries in our dictionary, we throw the **ValidationAppException** and pass the dictionary with errors. This exception will be caught inside our global error handler, which we will modify in a minute.

But before we do that, we have to register this behavior in the **Program** class:

```
builder.Services.AddMediatR(typeof(Application.AssemblyReference).Assembly);
builder.Services.AddAutoMapper(typeof(Program));
builder.Services.AddTransient(typeof(IPipelineBehavior<,>),
    typeof(ValidationBehavior<,>));
builder.Services.AddValidatorsFromAssembly(typeof(Application.AssemblyReference).Assem-
bly);
```

After that, we can modify the **ExceptionMiddlewareExtensions** class:

```
public static class ExceptionMiddlewareExtensions
```



```
{  
    public static void ConfigureExceptionHandler(this WebApplication app,  
ILoggerManager logger)  
    {  
        app.UseExceptionHandler(appError =>  
        {  
            appError.Run(async context =>  
            {  
                context.Response.ContentType = "application/json";  
  
                var contextFeature = context.Features.Get<IExceptionHandlerFeature>();  
                if (contextFeature != null)  
                {  
                    context.Response.StatusCode = contextFeature.Error switch  
                    {  
                        NotFoundException => StatusCodes.Status404NotFound,  
                        BadRequestException => StatusCodes.Status400BadRequest,  
                        ValidationAppException =>  
StatusCodes.Status422UnprocessableEntity,  
                        _ => StatusCodes.Status500InternalServerError  
                    };  
  
                    logger.LogError($"Something went wrong: {contextFeature.Error}");  
  
                    if (contextFeature.Error is ValidationAppException exception)  
                    {  
                        await context.Response  
                            .WriteAsync(JsonSerializer.Serialize(new { exception.Errors  
}));  
                    }  
                    else  
                    {  
                        await context.Response.WriteAsync(new ErrorDetails()  
                        {  
                            StatusCode = context.Response.StatusCode,  
                            Message = contextFeature.Error.Message,  
                            }.ToString());  
                    }  
                }  
            });  
        });  
    }  
}
```

So we modify the switch statement to check for the **ValidationAppException** type and to assign a proper status code 422.

Then, we use the declaration pattern to test the type of the variable and assign it to a new variable named **exception**. If the type is **ValidationAppException** we just write our response to the client providing our errors dictionary as a parameter. Otherwise, we do the same thing we did up until now.



Now, we can test this by sending an invalid request:

The screenshot shows a POST request to `https://localhost:5001/api/companies`. The Body tab is selected, showing a JSON payload with a missing 'address' field:

```
1 "name": "Marketing Solutions Ltd",
2 "address": null,
3 "country": "USA"
```

The response status is `422 Unprocessable Entity`, indicating validation errors. The response body shows:

```
1 "Errors": {
2     "Address": [
3         "'Company Address' must not be empty."
4     ]
5 }
```

Excellent, this works great.

Additionally, if the **Address** property has too many characters, we will see a different message:

The response body for a long address shows a more detailed validation message:

```
1 "Errors": {
2     "Address": [
3         "The length of 'Company Address' must be 60 characters or fewer. You entered 244
4             characters."
5     ]
6 }
```

Great.

33.7.3 Validating null Object

Now, if we send a request with an empty request body, we are going to get the result produced from our action:



<https://localhost:5001/api/companies>

POST



<https://localhost:5001/api/companies>

Params Auth Headers (9) Body Pre-req. Tests Settings

Body

Cookies

Headers (4)

Test Results



400 Bad Request

Pretty

Raw

Preview

Visualize

JSON



1 "CompanyForCreationDto object is null"

We can see the 400 status code and the error message. It is perfectly fine since we want to have a Bad Request response if the object sent from the client is null. But if for any reason you want to remove that validation from the action, and handle it with fluent validation rules, you can do that by modifying the **CreateCompanyCommandValidator** class and overriding the **Validate** method:

```
public sealed class CreateCompanyCommandValidator :  
AbstractValidator<CreateCompanyCommand>  
{  
    public CreateCompanyCommandValidator()  
    {  
        RuleFor(c => c.Company.Name).NotEmpty().MaximumLength(60);  
  
        RuleFor(c => c.Company.Address).NotEmpty().MaximumLength(60);  
    }  
  
    public override ValidationResult  
Validate(ValidationContext<CreateCompanyCommand> context)  
    {  
        return context.InstanceToValidate.Company is null  
        ? new ValidationResult(new[] { new  
ValidationFailure("CompanyForCreationDto",  
                    "CompanyForCreationDto object is null") })  
        : base.Validate(context);  
    }  
}
```

Now, you can remove the validation check inside the action and send a null body request:



Ultimate ASP.NET Core Web API

<https://localhost:5001/api/companies>

POST

https://localhost:5001/api/companies

Send

Params Auth Headers (9) Body Pre-req. Tests Settings

Cookies

raw

JSON

Beautify

1

Body Cookies Headers (7) Test Results



422 Unprocessable Entity

761 ms

293 B

Save Response

Pretty

Raw

Preview

Visualize

JSON



```
1
2   "Errors": {
3     "CompanyForCreationDto": [
4       "CompanyForCreationDto object is null"
5     ]
6   }
7 }
```

Pay attention that now the status code is 422 and not 400. But this validation is now part of the fluent validation.

If this solution fits your project, feel free to use it. Our recommendation is to use 422 only for the validation errors, and 400 if the request body is null.