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.

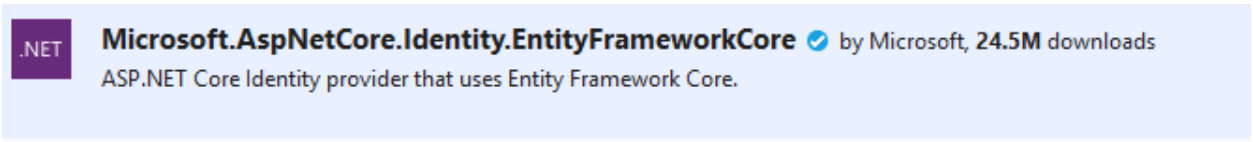
So, let's start with Identity integration.

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:



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

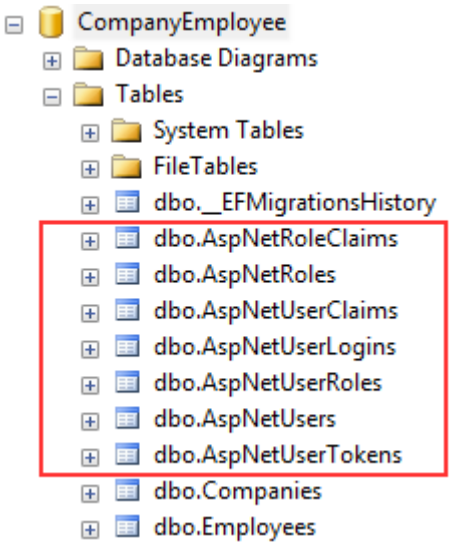
Creating tables is quite an easy process. All we have to do is to create and apply migration. So, let's create a migration:

```
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 IServiceManager _service;

       public AuthenticationController(IServiceManager 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)
```

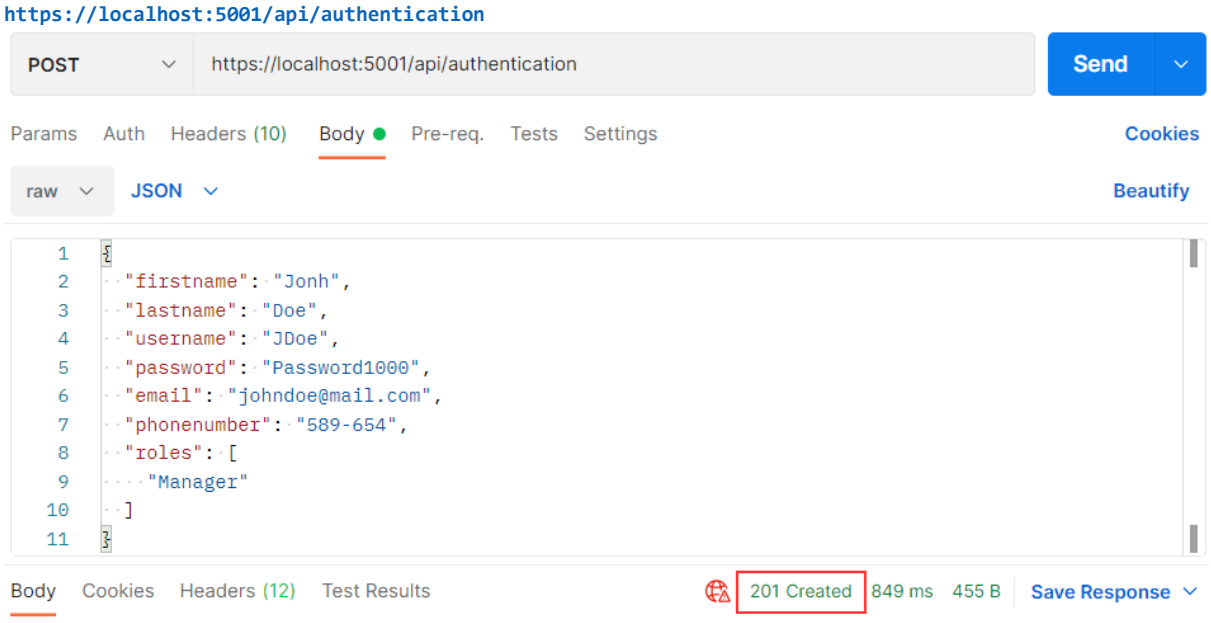
```
        {
                foreach (var error in result.Errors)
                {
                        ModelState.TryAddModelError(error.Code, error.Description);
                }
                return BadRequest(ModelState);
        }

        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:



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.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHM
I6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

Every part of all three parts is shown in a different color. The first part of JWT is the header, which is a JSON object encoded in the base64 format. The header is a standard part of JWT and we don't have to worry about it. It contains information like the type of token and the name of the algorithm:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

After the header, we have a payload which is also a JavaScript object encoded in the base64 format. The payload contains some attributes about the logged-in user. For example, it can contain the user id, the user subject, and information about whether a user is an admin user or not. **JSON web tokens are not encrypted** and can be decoded with any base64 decoder, so please **never include sensitive information in the Payload:**

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Finally, we have the signature part. Usually, the server uses the signature part to verify whether the token contains valid information, the information which the server is issuing. It is a digital signature that gets generated by combining the header and the payload. Moreover, it's based on a secret key that only the server knows:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  superSecertKey
) ☑ secret base64 encoded
```

So, if malicious users try to modify the values in the payload, they have to recreate the signature; for that purpose, they need the secret key only known to the server. 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.