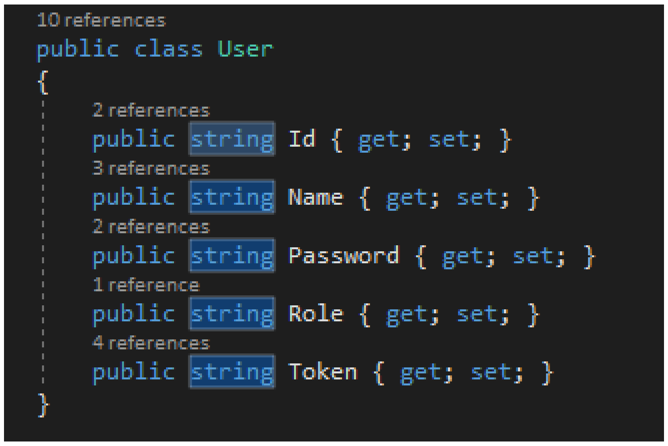
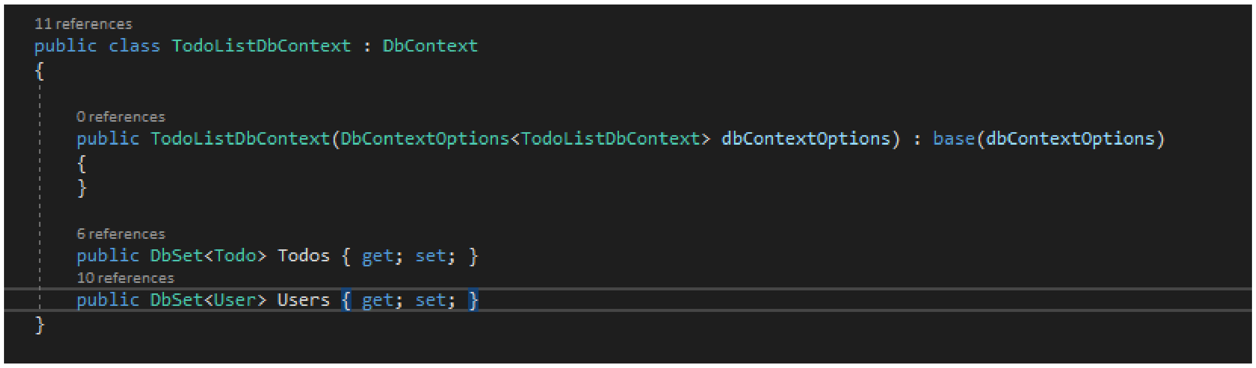
Securing an API with Json Web Tokens and Microsoft.Authorization

In a previous project I set up a simple API that connects to a postgres database running on a docker container that supports CRUD operations. I'm going to extend that project and add role authorization to the endpoints so they can only be accessed by users with the proper roles. To do this I need to start by adding a User table to my database. I need to add a User class object to my models with the following attributes:



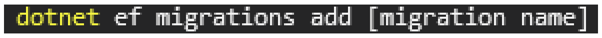
```
10 references
public class User
{
    2 references
    public string Id { get; set; }
    3 references
    public string Name { get; set; }
    2 references
    public string Password { get; set; }
    1 reference
    public string Role { get; set; }
    4 references
    public string Token { get; set; }
}
```

Now I'm going to add a DbSet<User> declaration in my database context class. That code will look like the following:



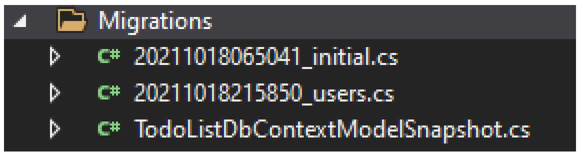
```
11 references
public class TodoListDbContext : DbContext
{
    0 references
    public TodoListDbContext(DbContextOptions<TodoListDbContext> dbContextOptions) : base(dbContextOptions)
    {
    }

    6 references
    public DbSet<Todo> Todos { get; set; }
    10 references
    public DbSet<User> Users { get; set; }
}
```

To finish adding this table to my database I just need to make a new migration. In my terminal I'll go to my projects directory and run the following script:



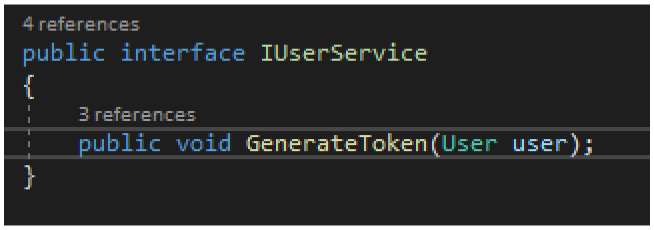
```
dotnet ef migrations add [migration name]
```

The migration name can be anything but I'm going to call it users. Once I have done this I should have the initial migration, my users migration, and the model snapshot. It will look like this:



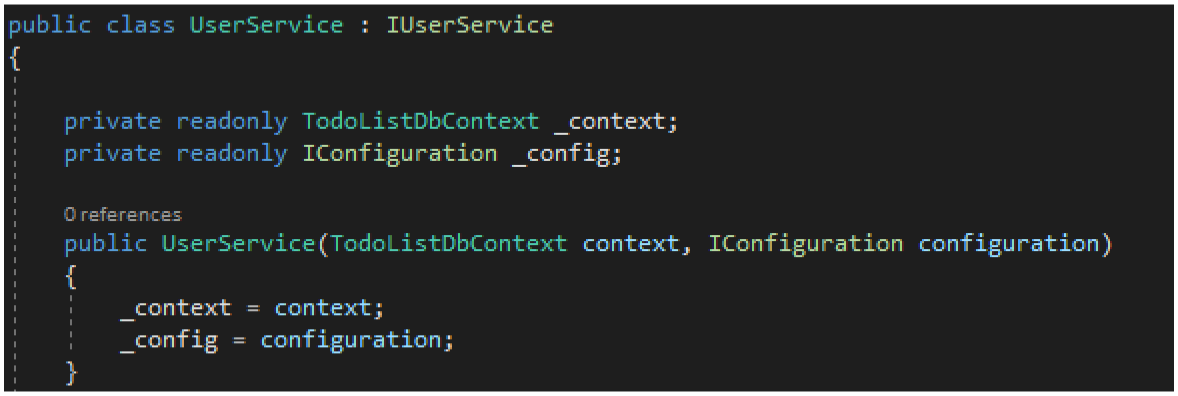
```
Migrations
    20211018065041_initial.cs
    20211018215850_users.cs
    TodoListDbContextModelSnapshot.cs
```

Now that my database is all set up with a Users table, I need to now be able to generate a Json Web Token. To accomplish this I'm going to make a Service folder and add two files, an IUserService.cs, and an UserService.cs. These will provide my Controller with the method for generating a token and can be easily extended in the future as needed without crowding any of my code. For this my IUserService.cs will declare only one function:



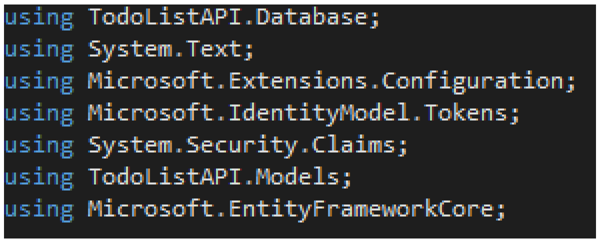
```
4 references
public interface IUserService
{
    3 references
    public void GenerateToken(User user);
}
```

In my UserService.cs file I will inherit from this interface and set up the constructor to take in an IConfiguration object and a TodoListDbContext object so it will have access to my database and be able to access my appsettings.json attributes. It will look like this:



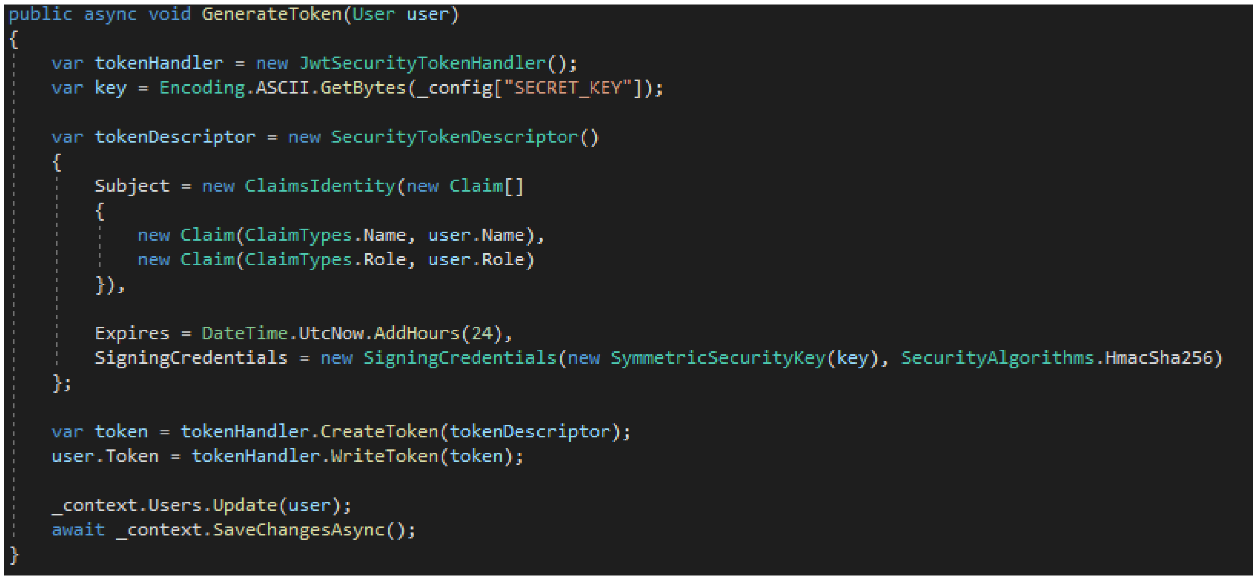
```
public class UserService : IUserService
{

    private readonly TodoListDbContext _context;
    private readonly IConfiguration _config;

    0 references
    public UserService(TodoListDbContext context, IConfiguration configuration)
    {
        _context = context;
        _config = configuration;
    }
```

Now, I'm going to create my GenerateToken() method. First, I need to import some NuGet packages. I will need Microsoft.AspNetCore.Authentication.JwtBearer, and System.IdentityModel.Tokens.Jwt. I will then need to include the following using statements:



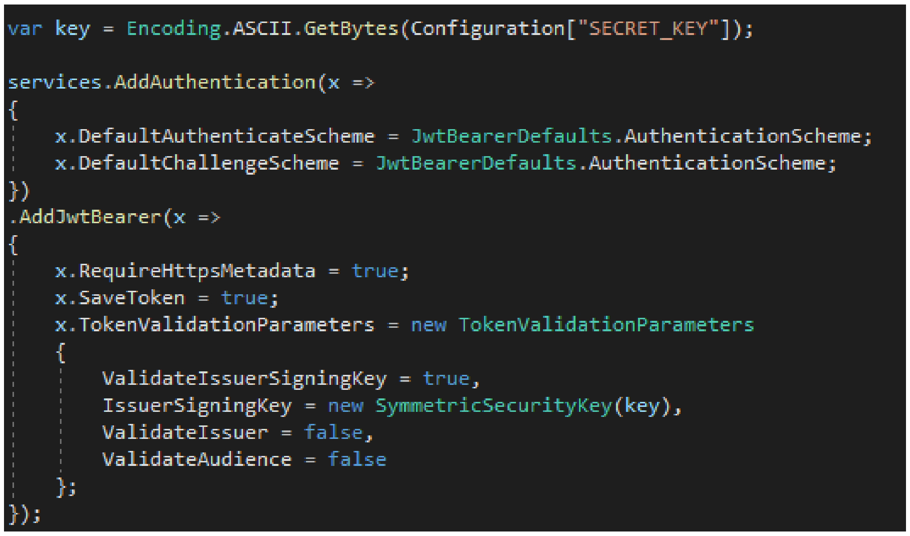
```
using TodoListAPI.Database;
using System.Text;
using Microsoft.Extensions.Configuration;
using Microsoft.IdentityModel.Tokens;
using System.Security.Claims;
using TodoListAPI.Models;
using Microsoft.EntityFrameworkCore;
```

Now I'm ready to write my method. This method will create a token based on a key that I will access from my appsettings.json file. This key can be anything. It will also encode a role and a name that it will get from the User and then after generating the token it will save it to the User's token attribute. The code will look like this:



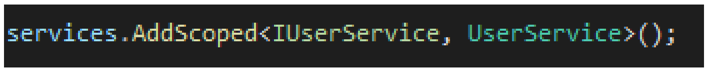
```
public async void GenerateToken(User user)
{
    var tokenHandler = new JwtSecurityTokenHandler();
    var key = Encoding.ASCII.GetBytes(_config["SECRET_KEY"]);

    var tokenDescriptor = new SecurityTokenDescriptor()
    {
        Subject = new ClaimsIdentity(new Claim[]
        {
            new Claim(ClaimTypes.Name, user.Name),
            new Claim(ClaimTypes.Role, user.Role)
        }),

        Expires = DateTime.UtcNow.AddHours(24),
        SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256)
    };

    var token = tokenHandler.CreateToken(tokenDescriptor);
    user.Token = tokenHandler.WriteToken(token);

    _context.Users.Update(user);
    await _context.SaveChangesAsync();
}
```

Next I need to add some startup code to my startup.cs file. I'm going to add some default authentication structure:



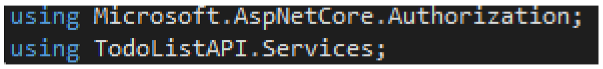
```
var key = Encoding.ASCII.GetBytes(Configuration["SECRET_KEY"]);

services.AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(x =>
{
    x.RequireHttpsMetadata = true;
    x.SaveToken = true;
    x.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
```

I'll also need to add an AddScoped() to my startup to map my interface to my class so it can be implemented at runtime:



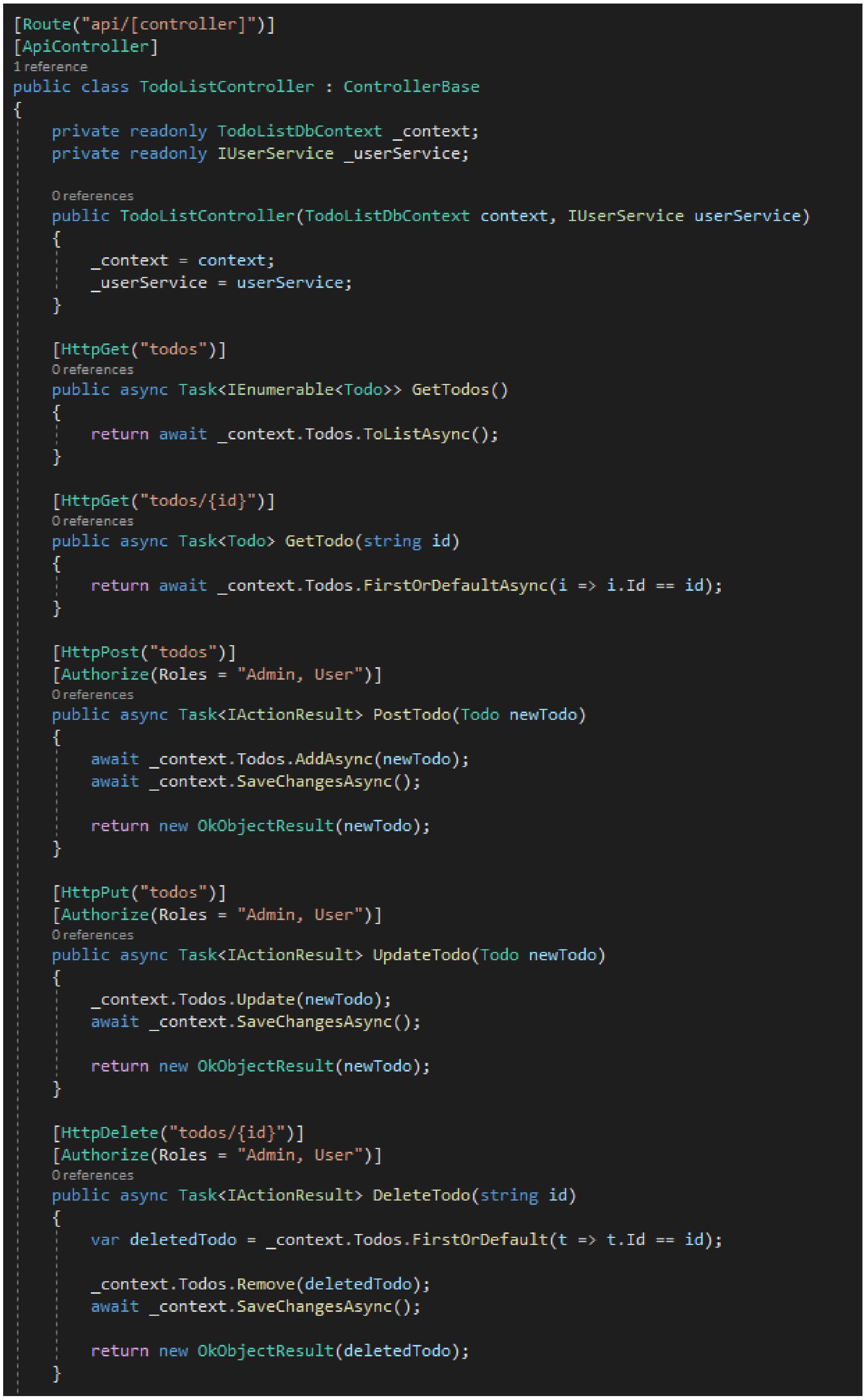
```
services.AddScoped<IUserService, UserService>();
```

Finally, I'm going to set up my TodoListController.cs file. I'm going to add the necessary CRUD operations for my Users table modeled off of the same logic I used for my Todo CRUD operations so I won't go into much detail here. Then I'll decorate the ones I want to have role authorization with the [Authorize(Roles = "")] decorator and add either Admin, User or both to the required roles. First, I'll need to include some using statements at the top. I will add the following using statements:

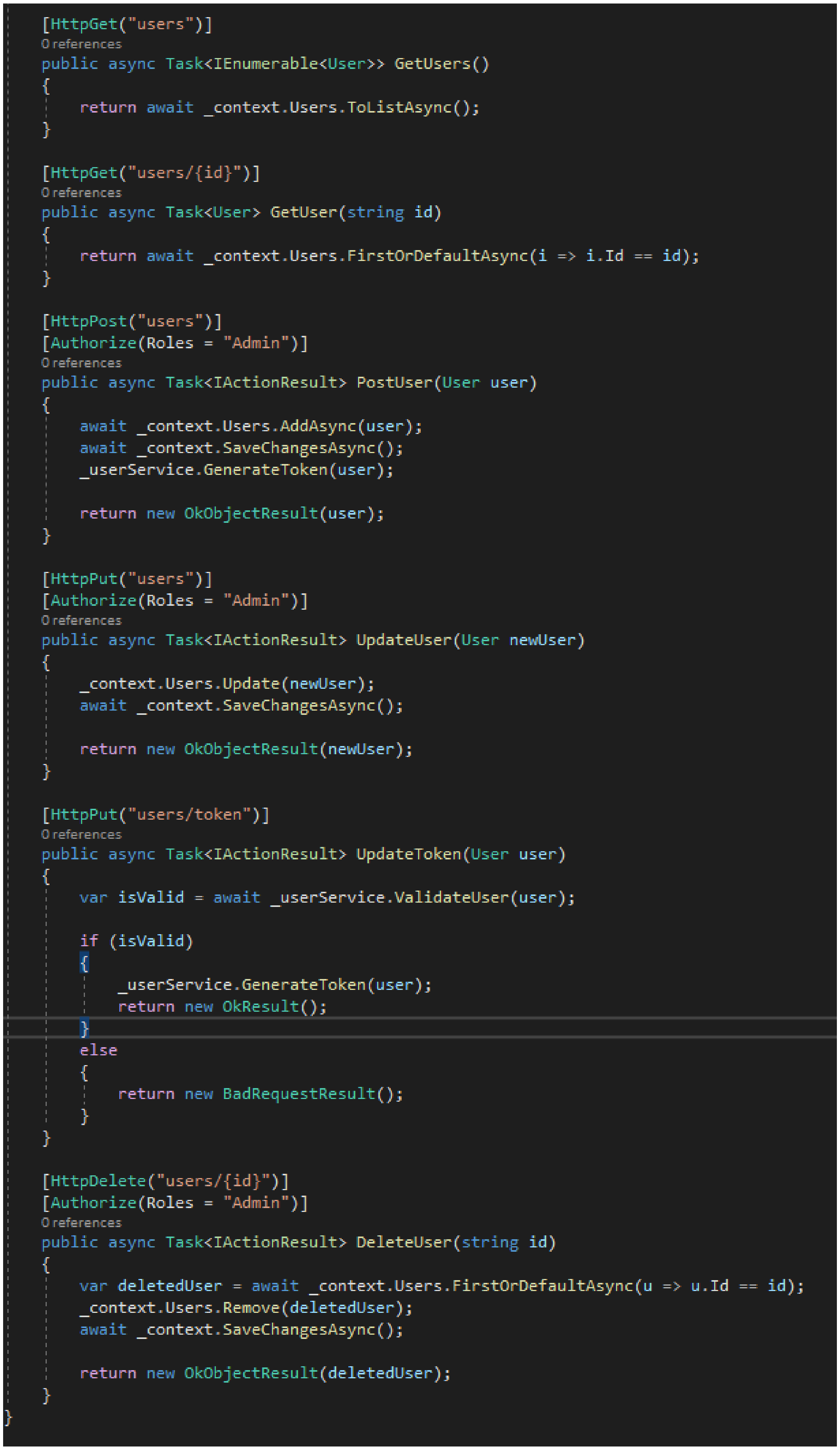


```
using Microsoft.AspNetCore.Authorization;
using TodoListAPI.Services;
```

Once, I've finished implementing my new CRUD operations and decorators my code will look like this:



```
[Route("api/[controller]")]
[ApiController]
1 reference
public class TodoListController : ControllerBase
{
    private readonly TodoListDbContext _context;
    private readonly IUserService _userService;

    0 references
    public TodoListController(TodoListDbContext context, IUserService userService)
    {
        _context = context;
        _userService = userService;
    }

    [HttpGet("todos")]
    0 references
    public async Task<IEnumerable<Todo>> GetTodos()
    {
        return await _context.Todos.ToListAsync();
    }

    [HttpGet("todos/{id}")]
    0 references
    public async Task<Todo> GetTodo(string id)
    {
        return await _context.Todos.FirstOrDefaultAsync(i => i.Id == id);
    }

    [HttpPost("todos")]
    [Authorize(Roles = "Admin, User")]
    0 references
    public async Task<IActionResult> PostTodo(Todo newTodo)
    {
        await _context.Todos.AddAsync(newTodo);
        await _context.SaveChangesAsync();

        return new OkObjectResult(newTodo);
    }

    [HttpPut("todos")]
    [Authorize(Roles = "Admin, User")]
    0 references
    public async Task<IActionResult> UpdateTodo(Todo newTodo)
    {
        _context.Todos.Update(newTodo);
        await _context.SaveChangesAsync();

        return new OkObjectResult(newTodo);
    }

    [HttpDelete("todos/{id}")]
    [Authorize(Roles = "Admin, User")]
    0 references
    public async Task<IActionResult> DeleteTodo(string id)
    {
        var deletedTodo = _context.Todos.FirstOrDefault(t => t.Id == id);

        _context.Todos.Remove(deletedTodo);
        await _context.SaveChangesAsync();

        return new OkObjectResult(deletedTodo);
    }
```



```
    [HttpGet("users")]
    0 references
    public async Task<IEnumerable<User>> GetUsers()
    {
        return await _context.Users.ToListAsync();
    }

    [HttpGet("users/{id}")]
    0 references
    public async Task<User> GetUser(string id)
    {
        return await _context.Users.FirstOrDefaultAsync(i => i.Id == id);
    }

    [HttpPost("users")]
    [Authorize(Roles = "Admin")]
    0 references
    public async Task<IActionResult> PostUser(User user)
    {
        await _context.Users.AddAsync(user);
        await _context.SaveChangesAsync();
        _userService.GenerateToken(user);

        return new OkObjectResult(user);
    }

    [HttpPut("users")]
    [Authorize(Roles = "Admin")]
    0 references
    public async Task<IActionResult> UpdateUser(User newUser)
    {
        _context.Users.Update(newUser);
        await _context.SaveChangesAsync();

        return new OkObjectResult(newUser);
    }

    [HttpPut("users/token")]
    0 references
    public async Task<IActionResult> UpdateToken(User user)
    {
        var isValid = await _userService.ValidateUser(user);

        if (isValid)
        {
            _userService.GenerateToken(user);
            return new OkResult();
        }
        else
        {
            return new BadRequestResult();
        }
    }

    [HttpDelete("users/{id}")]
    [Authorize(Roles = "Admin")]
    0 references
    public async Task<IActionResult> DeleteUser(string id)
    {
        var deletedUser = await _context.Users.FirstOrDefaultAsync(u => u.Id == id);
        _context.Users.Remove(deletedUser);
        await _context.SaveChangesAsync();

        return new OkObjectResult(deletedUser);
    }
}
```

The endpoints with the [Authorize] decorator will now require to be passed a token that has the corresponding role encoded on the token or it will send a 401 Unauthorized Request back.

I now have an API that allows for CRUD operations on both my Users table and my Todos table and requires an Authorized user token in order to access certain functions. These tokens are generated when the User is created. It still has a hiccup with regenerating the user token. This can be bypassed easily by simply removing the expiration date or extending it so you don’t have to worry about regenerating it. This isn’t great for security but for the simplicity of this API it isn’t the worst thing in the world.