**Authentication with JSON Web Tokens in .NET Core 3.1**

# Authentication (continued)

## **Token Authentication with JSON Web Tokens**

The idea behind token authentication is simple.

At this stage of our application, the user can log in with her username and password. We verify the credentials and tell the user that the password is correct.

But if the user wants to call a function, where she needs to be authenticated, she would have to send the credentials again. That’s because the web service is stateless.

This means, we never know who sent a request. Unless we get some credentials with the request.

Instead of entering the credentials every single time with every request, we could store the username and password on the local or session storage of the browser and grab the information from there. But that’s highly insecure because everybody who has access to your computer could have a glance at your password.

That’s where tokens come in.

A token is nothing more than a long string that stores information - or claims - of the user. These claims do not consist of the password, but it could tell the server who the user is and what kind of rights the user might have.

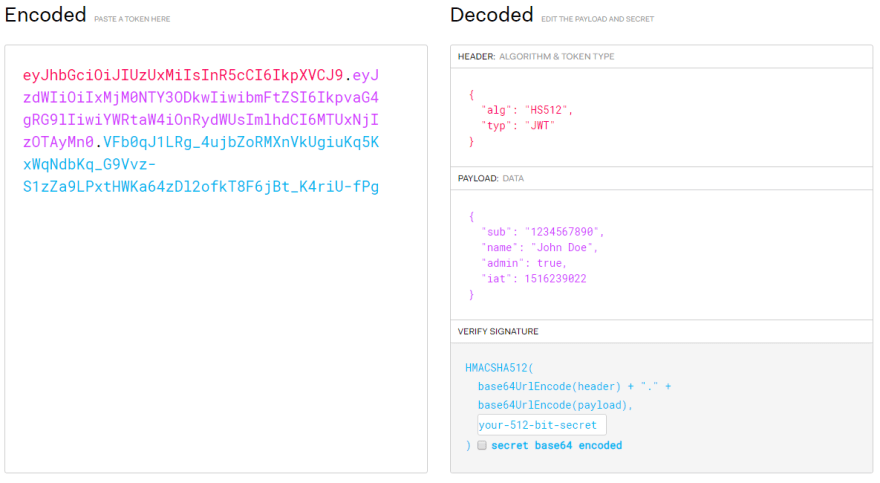
The token is generated with a private key only the server knows. So it’s hard to fake such a token. And we can give that token an expiration date. So, even if someone would be able to steal your token, chances are that the token is invalid as soon as this certain someone tries to use it.

Since this token doesn’t consist of critical information in plain text, we can store it in the browser and automatically send it to the web service with every request.

The service then knows who the user is and may even send a new token back for your next request.

On the website [jwt.io](https://jwt.io/), we’re able to have a look at a JSON web token and even how the information is stored in it.

You see the header with the used algorithm and the payload with claims like the name of the user, for instance.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--z71jyMFV--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/9wlu1f3jetc92k8zsrlv.png)

Okay, let’s use JSON web tokens now for our Web API.

## **JSON Web Tokens (JWT) preparations**

There are some things we have to do first before we write the actual code.

We start with the appsettings.json. Here we enter the security key for the JWT authentication. Below the ConnectionStrings section, we can create a new section AppSettings and just enter a new key called Token and set any kind of string as value. For instance, my top secret key would be totally sufficient here. Just make sure that it has at least 16 characters.

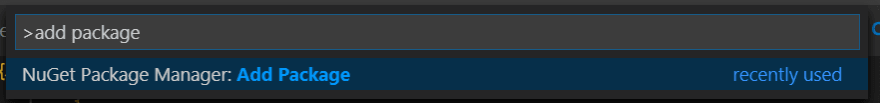
"AppSettings": {

"Token": "my top secret key"

},

After that, we have to add some package reference to our application.

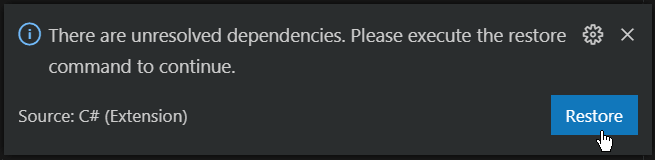
In Visual Studio Code you can do that by pressing Ctrl + Shift + P, enter Add Package to get the entry NuGet Package Manager: Add Package and then you can enter the desired package.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--vyJ8R1L9--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/kuwyyvh9h39oziw08etu.png)

We need a total of three package references.

The first one is Microsoft.IdentityModel.Tokens. This package includes types that provide support for security tokens and cryptographic operations like signing and verifying signatures. You can always choose the latest version.

After adding the package reference, Visual Studio Code wants to execute the restore command to resolve the new dependencies. Of course, we can do that.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--sq8O67mo--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/qqqxwtit81vx945tfrmi.png)

By the way, instead of using the NuGet Package Manager, you could have entered dotnet add package Microsoft.IdentityModel.Tokens into the terminal to add the latest version. Both ways work. It’s up to you what you prefer.

The second package is System.IdentityModel.Tokens.Jwt. This one provides support for creating, serializing and validating JSON web tokens. Exactly what we need.

And the last one is Microsoft.AspNetCore.Authentication.JwtBearer. This is a .NET Core middleware that enables our application to receive bearer tokens.

A bearer token is just a general name for the token we have already discussed. Some servers use short strings as tokens, we will utilize structured JSON web tokens. Both can be called bearer tokens.

Alright. After adding these packages, your .csproj file should be a bit bigger now and include these new references.

<PackageReference Include="Microsoft.IdentityModel.Tokens" Version="5.6.0"/>

<PackageReference Include="System.IdentityModel.Tokens.Jwt" Version="5.6.0"/>

<PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="3.1.2"/>

Now we’re ready to write some code.

## **JSON Web Tokens (JWT) implementations**

In the AuthRepository we start by adding a new private method called CreateToken() that returns a string and takes a User object as an argument.

private string CreateToken(User user)

{

return string.Empty; //token;

}

Let’s return an empty string for now, so that we can already call this method in the Login() method and set the response.Data accordingly if the user has entered the correct password.

public async Task<ServiceResponse<string>> Login(string username, string password)

{

ServiceResponse<string> response = new ServiceResponse<string>();

User user = await \_context.Users.FirstOrDefaultAsync(x => x.Username.ToLower().Equals(username.ToLower()));

if (user == null)

{

response.Success = false;

response.Message = "User not found.";

}

else if (!VerifyPasswordHash(password, user.PasswordHash, user.PasswordSalt))

{

response.Success = false;

response.Message = "Wrong password.";

}

else

{

response.Data = CreateToken(user);

}

return response;

}

Back to the CreateToken() method, we declare a List of Claims.

We will have to add some using directives while implementing this method.

The first claim type we add is the ClaimTypes.NameIdentifier. This will be the Id of the given user. The second one is ClaimTypes.Name, which will simply be the Username.

List<Claim> claims = new List<Claim>

{

new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),

new Claim(ClaimTypes.Name, user.Username)

};

Next, we need a SymmetricSecurityKey. This is the secret key from our appsettings.json file. We need to be able to access the file, so we jump to the constructor of the AuthRepository and inject the IConfiguration. Make sure to add the Microsoft.Extensions.Configuration using directive.

private readonly IConfiguration \_configuration;

public AuthRepository(DataContext context, IConfiguration configuration)

{

\_configuration = configuration;

\_context = context;

}

Back to the CreateToken() method, we create an instance of the SymmetricSecurityKey class and give it a byte array. We do that with Encoding.UTF8.GetBytes() and then access the \_configuration to get the proper section with our secret key as value.

SymmetricSecurityKey key = new SymmetricSecurityKey(Encoding.UTF8

.GetBytes(\_configuration.GetSection("AppSettings:Token").Value));

With that key, we create new SigningCredentials and use the HmacSha512Signature algorithm for that.

SigningCredentials creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signature);

Next is the SecurityTokenDescriptor. This object gets the information used to create the final token. We’ll give it the claims and an expiration date, for instance.

To set the claims, we set the Subject with a new ClaimsIdentity and give it the claims we created before.

Expires can be set to any date. How about the next day? So DateTime.Now.AddDays(1).

And finally, the SigningCredentials, which is our creds object.

SecurityTokenDescriptor tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(claims),

Expires = DateTime.Now.AddDays(1),

SigningCredentials = creds

};

We are almost done.

Now we need a new JwtSecurityTokenHandler and use this tokenHandler and the tokenDescriptor to create the SecurityToken.

JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler();

SecurityToken token = tokenHandler.CreateToken(tokenDescriptor);

And finally, with tokenHandler.WriteToken(token); we return the JSON web token as a string.

private string CreateToken(User user)

{

List<Claim> claims = new List<Claim>

{

new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),

new Claim(ClaimTypes.Name, user.Username)

};

SymmetricSecurityKey key = new SymmetricSecurityKey(Encoding.UTF8

.GetBytes(\_configuration.GetSection("AppSettings:Token").Value));

SigningCredentials creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signature);

SecurityTokenDescriptor tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(claims),

Expires = DateTime.Now.AddDays(1),

SigningCredentials = creds

};

JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler();

SecurityToken token = tokenHandler.CreateToken(tokenDescriptor);

return tokenHandler.WriteToken(token);

}

Alright, take a deep breath and let’s test this in Postman. HTTP Method is POST, we use the login URL, the correct credentials and hit ‘Send’.

{

"data": "eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJuYW1laWQiOiIxIiwidW5pcXVlX25hbWUiOiJwYXRyaWNrIiwibmJmIjoxNTgyMTA1MDEyLCJleHAiOjE1ODIxOTE0MTIsImlhdCI6MTU4MjEwNTAxMn0.FaxvO8pqLLkBjplEW815-DzekgBwW94gBx--\_3n4X5UAs6kRuM2zflpwQ0H2PnCgIuupJKq7EED5c\_mC\_DI8FQ",

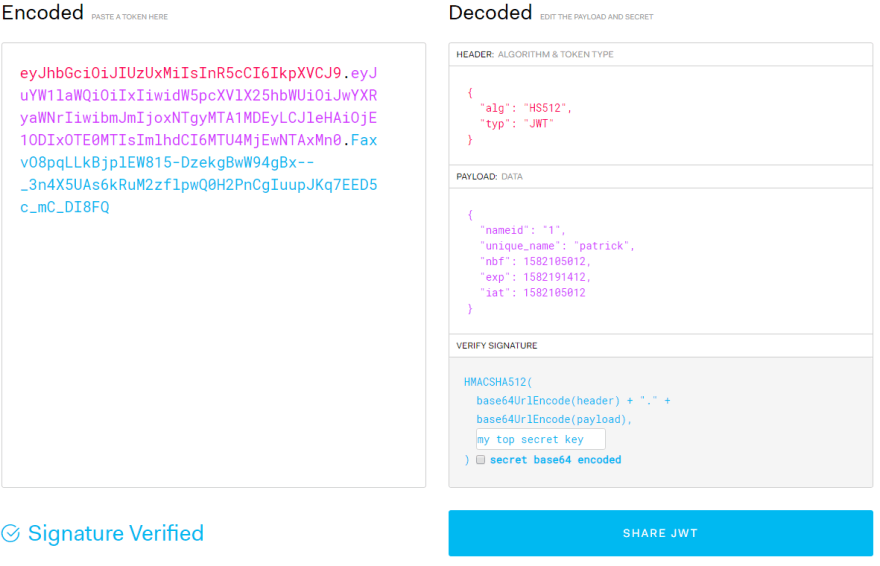
"success": true,

"message": null

}

There is our token!

We can grab this token now and have a deeper look at the JWT debugger on jwt.io.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--r6vQEt4h--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/jy4t0qaodfwc05u1bo0e.png)

When you paste the token, you’re able to see the claims we have entered in the code.

And if you enter the correct key, the signature can be verified.

Great! So that’s how we get our JSON Web Token.

## **Authorize Attribute**

To secure web service calls or even a complete controller, we can use the [Authorize] attribute on top of the controller class or on top of any method you want to secure. But before we can use this attribute, we have to add an authentication scheme to the web service. We do that in the Startup class.

In the ConfigureServices() method, we use AddAuthentication() with the JwtBearerDefaults.AuthenticationScheme and add some configuration options with AddJwtBearer().

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

});

Regarding the options we initialize a new instance of TokenValidationParameters and set these parameters.

We want to validate the signing key, so we set ValidateIssuerSigningKey to true.

The IssuerSigningKey is again the one from our appsettings.json file. So a new SymmetricSecurityKey that gets the encoded AppSettings:Token value.

We don’t need to validate the issuer or the audience, hence we set ValidateIssuer and ValidateAudience to false.

So far the authentication scheme.

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuerSigningKey = true,

IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII

.GetBytes(Configuration.GetSection("AppSettings:Token").Value)),

ValidateIssuer = false,

ValidateAudience = false

};

});

Additionally, we have to add the .NET Core AuthenticationMiddleware to the IApplicationBuilder to enable authentication capabilities.

To do that, we add the line app.UseAuthentication(); **above** app.UseAuthorization();. It’s important to add the line there.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

//app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

Alright. With that in place, we can make use of the authentication. To do that we add the [Authorize] attribute on top of the CharacterController class.

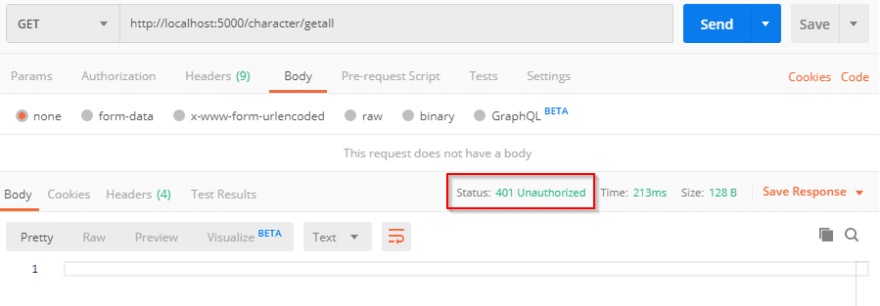
[Authorize]

[ApiController]

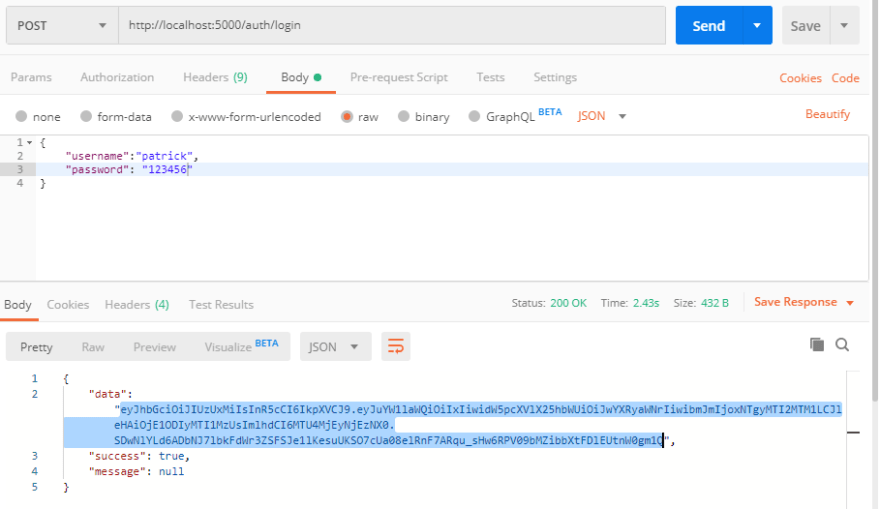
[Route("[controller]")]

public class CharacterController : ControllerBase

If we now want to get all RPG characters with Postman using GET as HTTP method and the URL http://localhost:5000/character/getall, we get a 401 Unauthorized back.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--rLTbl-WU--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/5e9blygm02hruj7qzz0v.png)

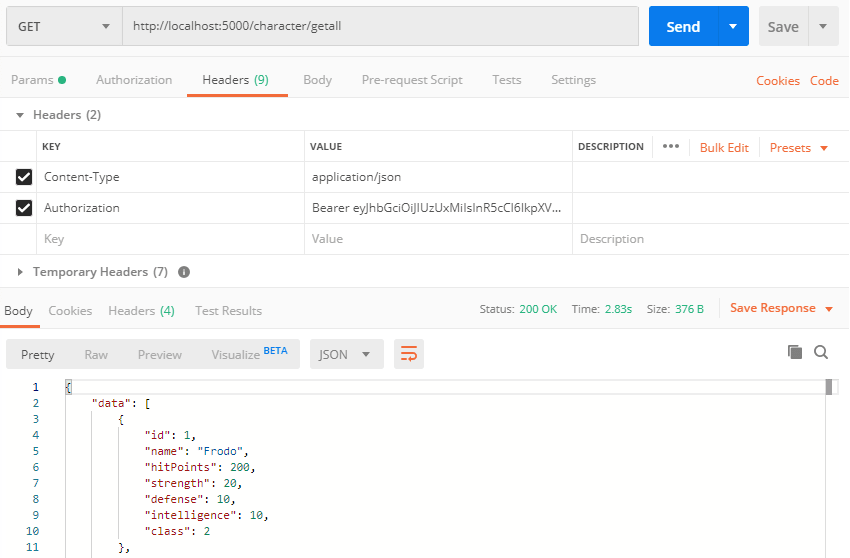
That’s exactly what we want! So now we have to add a token to the header of our request. Let’s login first to get the JSON web token.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--MaHwmu4T--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/hcpiijsk6obax34gpu5w.png)

Just copy the token and then go back to the GetAll call.

Here we have to add a header. We add the key Authorization and add the token as value. The token itself is not enough. We have to add the term Bearer with a space in front of the token.

Now we’re finally able to get all RPG characters again.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--V8yp7iua--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/l41fpj3h0p3n5ne8vmqq.png)

There’s one little thing I’d like to add here.

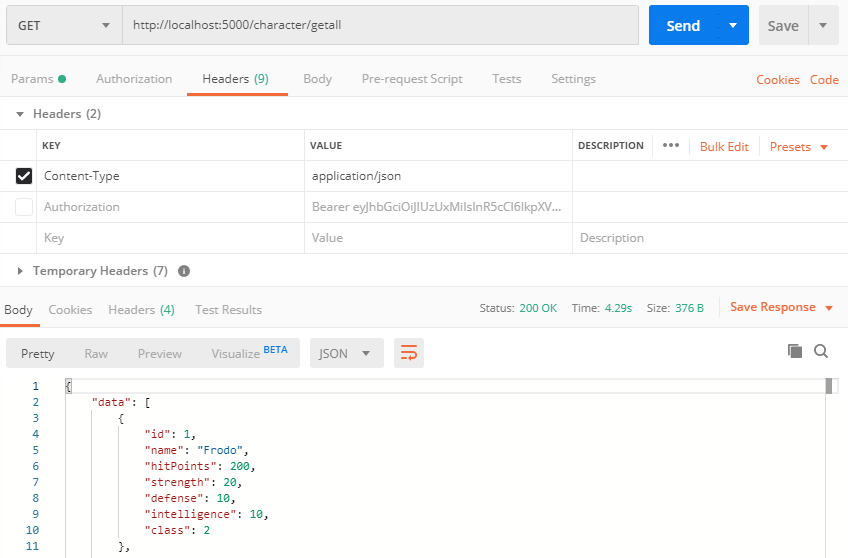
To call any method of the CharacterController, we have to be authenticated. But we can make exceptions to the rule. For instance, if we add the attribute [AllowAnonymous] on top of the GetAll() method, we can call this method again without being authenticated.

[AllowAnonymous]

[HttpGet("GetAll")]

public async Task<IActionResult> Get()

In Postman you can see that we can remove the Authorization key in the header and still get the RPG characters from the database.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--ps8v7qlE--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/o7rj9v0qjezi8hwol7m3.png)

Let’s remove the attribute again, because we only want authenticated users to get characters.

Actually, we want users to get their created characters. At the moment they can see all RPG characters.

So we have to make use of the relation between users and characters and we also have to get the claims from the authenticated user.

## **Read Claims & Get the User’s RPG Characters**

One of the beauties of the ControllerBase class is that it provides a User object of type ClaimsPrincipal.

// Summary:

// Gets the System.Security.Claims.ClaimsPrincipal for user associated with the

// executing action.

public ClaimsPrincipal User { get; }

This User object provides all the claims we added to the JSON web token.

Let’s get the NameIdentifier which was the Id of our authenticated user from the database.

In the Get() method of the CharacterController we define a new int variable.

With User.Claims we access the claims and then we iterate through them or find the one we want with FirstOrDefault() followed by a lambda expression where the Type of the claim equals ClaimTypes.NameIdentifier.

From that result, we grab the Value and parse it to an int.

int id = int.Parse(User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.NameIdentifier).Value);

Now we have to pass this id to the CharacterService. So we make a little change to the GetAllCharacters() method and add the id as an argument in the corresponding interface as well as the service and maybe call this argument userId to avoid any confusion.

Task<ServiceResponse<List<GetCharacterDto>>> GetAllCharacters(int userId);

Then we can call the method with the proper userId we got from the claims and make a slight modification to the service method.

public async Task<IActionResult> Get()

{

int userId = int.Parse(User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.NameIdentifier).Value);

return Ok(await \_characterService.GetAllCharacters(userId));

}

The only thing we have to change in the GetAllCharacters() method is how we access the Characters from the \_context. Instead of returning all of the RPG characters, we make use of the Where() function and only return the characters that are related to the given user with the help of the userId.

List<Character> dbCharacters = await \_context.Characters.Where(c => c.User.Id == userId).ToListAsync();

Thanks to Entity Framework we can access the related User object and its Id and actually get the RPG characters that have the proper UserId set in the database table.

When we test that with Postman, we get no characters back. That’s absolutely correct because we haven’t set any relations, yet.

But we can fix that real quick.

In the Characters table, we simply set the UserId of Frodo to 1. Run the test again, and now we’re getting Frodo! Setting the UserId for Sam as well, we’re getting both our heroes back.

Great! This works. Of course, we want to add the relations with our code and not manually in the database every single time. Before we do that, let’s sum up what you have learned in this section.

## **Summary**

Congratulations! You successfully implemented JSON Web Token Authentication in your Web API.

We started by creating the User model and adding this new User as a relation to our RPG characters. This is a one-to-many relation, which means that one user can have several characters.

Then we were already diving into the theory of authentication. You learned how authentication works in general with password hash values and password salts and why you should use hashed values and salts. It’s all about security.

We built the user registration and the user login and used a specific cryptography algorithm to hash and verify the entered passwords.

After that, we covered token authentication with JSON web tokens.

You now know what a bearer token is, what you find in that token - like the used algorithm and the payload - and how you can add any claims you want to add to that token.

So you learned how you can create a JSON web token and how to secure your Web API with the [Authorize] attribute. While doing that, you also learned how to read the claims from the JSON web token and use them to return the correct data to the user.

Great! Now it’s time to add some more relations to our application. What about some skills to use during fights, for instance? And, of course, let’s add the correct relation between users and characters as soon as a character has been generated.

We do all that in the next section.

That's it for the 8th part of this tutorial series. I hope it was useful for you. To get notified for the next part, simply follow me here on [dev.to](https://dev.to/_patrickgod) or [subscribe to my newsletter](https://mailchi.mp/96620175570f/dotnetcore). You'll be the first to know.

See you next time!

Take care.

Next up: Advanced Relationships with Entity Framework Core

*Image created by cornecoba on*[*freepik.com*](https://www.freepik.com/free-vector/flying-rocket-background_1130797.htm)*.*

### But wait, there’s more!

* Let’s connect on [Twitter](https://twitter.com/_PatrickGod), [YouTube](https://www.youtube.com/channel/UCq8LldVrjqe61KQttZlLW8g), [LinkedIn](https://www.linkedin.com/in/patrickgod) or here on [dev.to](https://dev.to/_patrickgod).
* Get the [5 Software Developer’s Career Hacks](http://patrickgod.com/5-software-developers-career-hacks/) for free.
* Enjoy more valuable articles for your developer life and career on [patrickgod.com](http://patrickgod.com/).