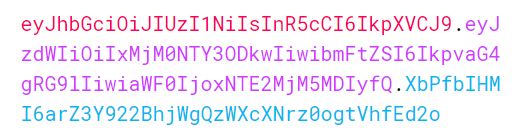
**Introduction to JSON Web Tokens**

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, payload, and the signature.**

One real example of JSON web token:



Every part of all three parts is shown in a different color:

**Header**

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.

Header Sample:

JWT standard header

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4 | {    "alg": "HS256",    "typ": "JWT"  } |

**Payload**

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 user id, user subject, and the 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**.

Payload sample:

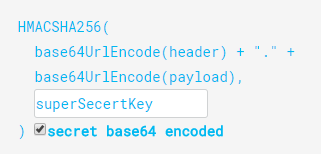
JWT sample payload

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5 | {    "sub": "1234567890",    "name": "John Doe",    "iat": 1516239022  } |

**Signature**

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



So, if malicious users try to modify the values in the payload, they have to recreate the signature and for that purpose, they need the secret key which has present at server. At the server side, we can easily verify if the values are original or not by comparing the original signature with a new signature computed from the values coming from the client.

So, we can easily verify the integrity of our data just by comparing the digital signatures. This is the reason why we use JWT.

## Configuring JWT Authentication

To configure JWT authentication in .NET Core, we need to modify Startup.csfile. It’s a bootstrapper class that runs when our application starts. Inside the file, we have the ConfigureSerivces method that adds services to the IServiceCollection container, thus making them available for the constructor injection.

JWT’s support is built into ASP.NET Core 2.0 and we are going to configure an authentication middleware for JSON web tokens.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public void ConfigureServices(IServiceCollection services)  {      services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)      .AddJwtBearer(options =>      {          options.TokenValidationParameters = new TokenValidationParameters          {              ValidateIssuer = true,              ValidateAudience = true,              ValidateLifetime = true,              ValidateIssuerSigningKey = true,                ValidIssuer = "http://localhost:5000",              ValidAudience = "http://localhost:5000",              IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("superSecretKey@345"))          };      });        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);  } |

**Note - For the sake of simplicity, we are going to add all the code inside the ConfigureServices method. But the better practice is to use Extension methods so we could free our ConfigureServices method from extra code lines.**

**We need to modify the ConfigureServices method to add the JWT support:**

**Code Explanation**

Firstly, we register the JWT authentication middleware by calling the method AddAuthentication on the ISerivceCollectioninterface. Next, we specify the authentication scheme JwtBearerDefaults.AuthenticationScheme. We also provide some parameters that will be used while validating JWT.

**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)**

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)**
* C#

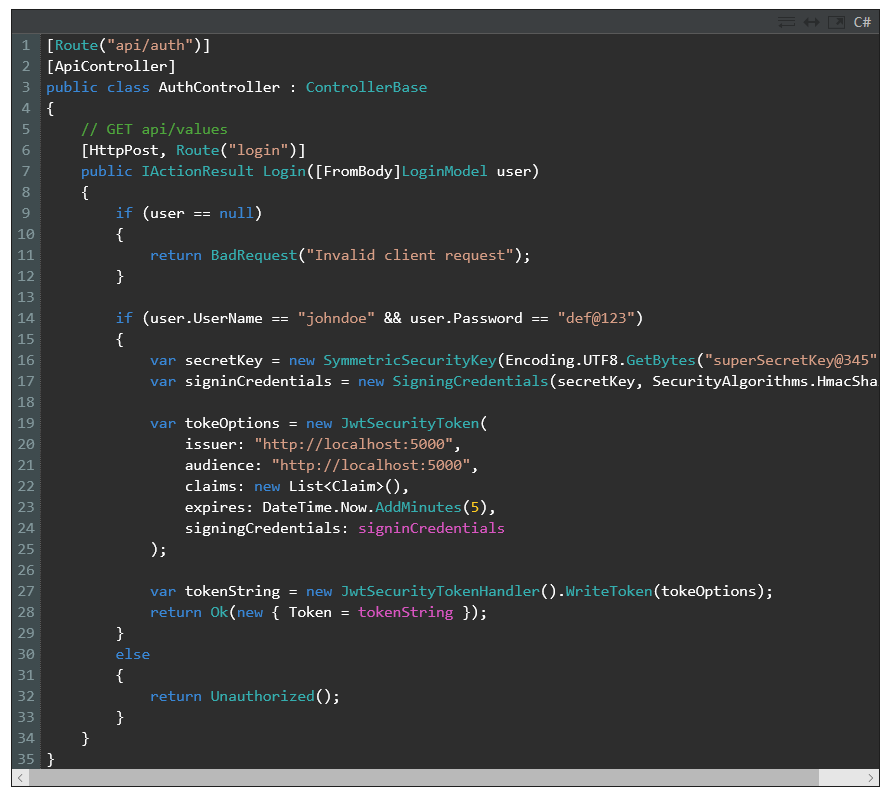
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public void Configure(IApplicationBuilder app, IHostingEnvironment env)  {      if (env.IsDevelopment())      {          app.UseDeveloperExceptionPage();      }      else      {          app.UseHsts();      }        app.UseAuthentication();        app.UseHttpsRedirection();      app.UseMvc();  } |

* And that’s all we need to configure the JWT authentication in ASP.NET Core.

Now let’s create the AuthController inside the Controllers folder. Inside the AuthControllerwe are going to validate the user’s credentials. If the credentials are valid, we are going to issue a JSON web token. For this demo, we are going to hardcode the username and password to implement a fake user. After validating the user’s credentials we are going to generate a JWT with a secret key. JWT uses the secret key to generate the signature.

Let’s implement the AuthController:

C#



Code Explanation

First of all, notice the use of the [HttpPost] attribute. After applying this attribute to action methods as a result the API endpoint only responds to HTTP POST requests. Inside the login method, we are creating the SymmetricSecretKey with the secret key value **superSecretKey@345.**Then, we are creating the objec SigningCredentials and as arguments, we provide a secret key and a name of the algorithm that we are going to use to encode the token.

Here comes the interesting part.

The first two steps are the standard steps that you don’t need to worry about. The third step is the one that we are interested in. In the third step we are creating the JwtSecurityToken object with some important parameters:

* **Issuer:** The first parameter is a simple string representing the name of the web server that issues the token
* **Audience:** The second parameter is a string value representing valid recipients
* **Claims:** The third argument is a list of user roles, for example, the user can be an admin, manager or author (we are going to add roles in the next post)
* **Expires:** The fifth argument is DateTime object that represents the date and time after which the token expires

Finally, we create a string representation of JWT by calling the WriteToken method on JwtSecurityTokenHandler**.** Finally, we are returning JWT in a response. As a response, we have created an anonymous object that contains only the Token property

**Simple authorization in ASP.NET Core**

Simple Authorization is controlled through the AuthorizeAttribute attribute and its various parameters. At its simplest, applying the AuthorizeAttribute attribute to a controller or action limits access to the controller or action to any authenticated user.

For example, the following code limits access to the AccountController to any authenticated user.

[Authorize]

public class AccountController : Controller

{

public ActionResult Login()

{

}

public ActionResult Logout()

{

}

}

If you want to apply authorization to an action rather than the controller, apply the AuthorizeAttribute attribute to the action itself:

public class AccountController : Controller

{

public ActionResult Login()

{

}

[Authorize]

public ActionResult Logout()

{

}

}

Now only authenticated users can access the Logout function.

You can also use the AllowAnonymous attribute to allow access by non-authenticated users to individual actions. For example:

[Authorize]

public class AccountController : Controller

{

[AllowAnonymous]

public ActionResult Login()

{

}

public ActionResult Logout()

{

}

}

This would allow only authenticated users to the AccountController, except for the Login action, which is accessible by everyone, regardless of their authenticated or unauthenticated / anonymous status.

**Role Based authorization in ASP.NET Core**

When an identity is created it may belong to one or more roles. For example, Tracy may belong to the Administrator and User roles whilst Scott may only belong to the User role. How these roles are created and managed depends on the backing store of the authorization process.

Roles are exposed to the developer through the [IsInRole](https://docs.microsoft.com/en-us/dotnet/api/system.security.principal.genericprincipal.isinrole) method on the [ClaimsPrincipal](https://docs.microsoft.com/en-us/dotnet/api/system.security.claims.claimsprincipal) class.

## Adding role checks

Role-based authorization checks are declarative—the developer embeds them within their code, against a controller or an action within a controller, specifying roles which the current user must be a member of to access the requested resource.

For example, the following code limits access to any actions on the AdministrationController to users who are a member of the Administrator role:

[Authorize(Roles = "Administrator")]

public class AdministrationController : Controller

{

}

You can specify multiple roles as a comma separated list:

[Authorize(Roles = "HRManager,Finance")]

public class SalaryController : Controller

{

}

This controller would be only accessible by users who are members of the HRManager role or the Finance role.

If you apply multiple attributes then an accessing user must be a member of all the roles specified; the following sample requires that a user must be a member of both the PowerUser and ControlPanelUser role.

[Authorize(Roles = "PowerUser")]

[Authorize(Roles = "ControlPanelUser")]

public class ControlPanelController : Controller

{

}

You can further limit access by applying additional role authorization attributes at the action level:

[Authorize(Roles = "Administrator, PowerUser")]

public class ControlPanelController : Controller

{

public ActionResult SetTime()

{

}

[Authorize(Roles = "Administrator")]

public ActionResult ShutDown()

{

}

}

In the previous code snippet members of the Administrator role or the PowerUser role can access the controller and the SetTime action, but only members of the Administrator role can access the ShutDown action.

You can also lock down a controller but allow anonymous, unauthenticated access to individual actions.

[Authorize]

public class ControlPanelController : Controller

{

public ActionResult SetTime()

{

}

[AllowAnonymous]

public ActionResult Login()

{

}

}

**With JWT Bearer token authentication**

You have to add the role in claim to validate the role.

**Ex**

**Add below part in configure services**

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters()

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

ValidIssuer = token.Issuer,

ValidAudience =token.Audience,

IssuerSigningKey = new SymmetricSecurityKey(secret),

**RoleClaimType = "Role"**

};

});

**And add the Role type calim when authenticating the user**

token = string.Empty;

var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_tokenManagement.Secret));

var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

List<Claim> claims = new List<Claim>();

claims.Add(new Claim(JwtRegisteredClaimNames.GivenName, request.Username));

**claims.Add(new Claim("Role", request.Role));**

claims.Add(new Claim("access", "report"));

var jwtToken = new JwtSecurityToken(

\_tokenManagement.Issuer,

\_tokenManagement.Audience,

claims,

expires: DateTime.Now.AddMinutes(\_tokenManagement.AccessExpiration),

signingCredentials: credentials

);

token = new JwtSecurityTokenHandler().WriteToken(jwtToken);

return true;

**Claim Based authorization in ASP.NET Core**

When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is a name value pair that represents what the subject is, not what the subject can do. For example, you may have a driver's license, issued by a local driving license authority. Your driver's license has your date of birth on it. In this case the claim name would be DateOfBirth, the claim value would be your date of birth, for example 8th June 1970 and the issuer would be the driving license authority. Claims based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value. For example if you want access to a night club the authorization process might be:

The door security officer would evaluate the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.

An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

## Adding claims checks

Claim based authorization checks are declarative - the developer embeds them within their code, against a controller or an action within a controller, specifying claims which the current user must possess, and optionally the value the claim must hold to access the requested resource. Claims requirements are policy based, the developer must build and register a policy expressing the claims requirements.

The simplest type of claim policy looks for the presence of a claim and doesn't check the value.

First you need to build and register the policy. This takes place as part of the Authorization service configuration, which normally takes part in ConfigureServices() in your Startup.cs file.

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.AddAuthorization(options =>

{

options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("**access** "));

});

}

Most claims come with a value. You can specify a list of allowed values when creating the policy. The following example would only succeed for employees whose employee number was 1, 2, 3, 4 or 5.

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.AddAuthorization(options =>

{

options.AddPolicy("Founders", policy =>

policy.RequireClaim("EmployeeNumber", "1", "2", "3", "4", "5"));

});

}

**With JWT Bearer token authentication**

You have to add the **access** in claim to validate the role. **(Example case of access claim)**

**Ex**

**Add below part in configure services**

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters()

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

ValidIssuer = token.Issuer,

ValidAudience =token.Audience,

IssuerSigningKey = new SymmetricSecurityKey(secret),

RoleClaimType = "Role"

};

});

services.AddAuthorization(opt => {

opt.AddPolicy("hasReportAccess", policy => policy.RequireClaim("**access**", "report").RequireRole("user"));

});

**And add the claim when authenticating the user**

token = string.Empty;

var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_tokenManagement.Secret));

var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

List<Claim> claims = new List<Claim>();

claims.Add(new Claim(JwtRegisteredClaimNames.GivenName, request.Username));

claims.Add(new Claim("Role", request.Role));

**claims.Add(new Claim("access", "report"));**

var jwtToken = new JwtSecurityToken(

\_tokenManagement.Issuer,

\_tokenManagement.Audience,

claims,

expires: DateTime.Now.AddMinutes(\_tokenManagement.AccessExpiration),

signingCredentials: credentials

);

token = new JwtSecurityTokenHandler().WriteToken(jwtToken);

return true;

Underneath the covers, [role-based authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles?view=aspnetcore-2.2) and [claims-based authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/claims?view=aspnetcore-2.2) use a requirement, a requirement handler, and a pre-configured policy. These building blocks support the expression of authorization evaluations in code. The result is a richer, reusable, testable authorization structure.

An authorization policy consists of one or more requirements. It's registered as part of the authorization service configuration, in the Startup.ConfigureServices method:

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

services.AddAuthorization(options =>

{

options.AddPolicy("officeHour ", policy =>

policy.Requirements.Add(new officeHourRequirment (8,23)).RequireRole("user")

});

}

To implement the requirement inherit the class from **IAuthorizationRequirement interface** check the **officeHourRequirment** example

**public class officeHourRequirment : IAuthorizationRequirement**

{

public officeHourRequirment(int start, int end)

{

Start = start;

End = end;

}

public int Start { get; set; }

public int End { get; set; }

}

**public class officeHourRequirmentHandler : AuthorizationHandler<officeHourRequirment>**

{

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, officeHourRequirment requirement)

{

var Now = DateTime.Now;

if (Now.Hour > requirement.Start && Now.Hour < requirement.End)

{

context.Succeed(requirement);

}

return Task.CompletedTask;

}

}

Configure the policy in **ConfigureServices** Method

**services.AddAuthorization(opt => {**

**opt.AddPolicy("officeHour", policy => policy.AddRequirements(new officeHourRequirment(8,23)).RequireRole("user"));**

**});**

**services.AddSingleton<IAuthorizationHandler, officeHourRequirmentHandler>();**

**Controller Method**

//Policy based authorization

[HttpGet("financereport")]

[Authorize(Policy = "officeHour")]

public string GetFinanceReport()

{

Code……

}

**Resource Based Authorization**

It is similar to policy-based authorization, but the one small difference is that here we authorize the source from external resource.

For example – we are checking the author authorization. So, we are getting the author info from external resource at runtime and authoring the source.

First, we are adding the dependency of requirement in ConfigureService method of startup class.

**services.AddSingleton<IAuthorizationHandler, AuthorRequirementHandler>();**

After that we need to implement the requirement.

**public class AuthorRequirement : IAuthorizationRequirement { }**

**public class AuthorRequirementHandler : AuthorizationHandler<AuthorRequirement, Report>**

{

protected override Task HandleRequirementAsync(

AuthorizationHandlerContext context,

AuthorRequirement requirement,

Report resource)

{

if (context.User.Identity.Name == resource.Author)

{

context.Succeed(requirement);

}

return Task.CompletedTask;

}

}

**Controller Code**

[HttpGet]

public async Task<IActionResult> PutReport(string id)

{

var report = new Report { Author = "alice", Content = "" }; // Here we would get the **resource from somewhere**

var result = await \_authorizationService.AuthorizeAsync(HttpContext.User, report, new AuthorRequirement());

if (result.Succeeded)

return Ok();

else

return Unauthorized();

}

In controller code we are getting the required information for authorization from external resource and calling the \_authorizationService.AuthorizeAsync method and passing the required params.

**private IAuthorizationService \_authorizationService;**

**purohitankit7995@gmail.com**