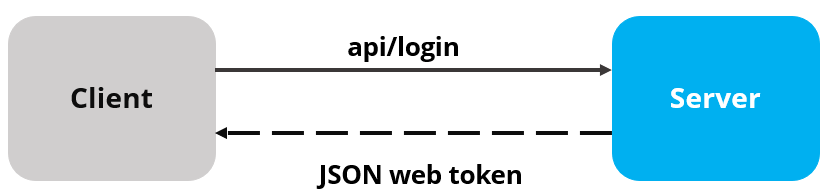
**Understanding the Overall Concept of JWT Authentication**



Once the server verifies the user’s credentials and confirms their validity, it sends an encoded JWT to the client. A JSON web token is essentially a JavaScript object that can include various attributes of the logged-in user, such as a username, user subject, user roles, or other relevant information.

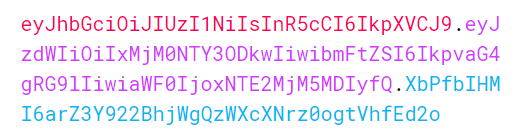
On the client side, we store the JWT in the browser’s storage to maintain the user’s login session. Additionally, we can utilize the information from the JWT to enhance the security of our application.What is JWT (JSON Web Token)

**What is JWT (JSON Web Token)**

JSON web tokens provide a secure method for exchanging data between two parties using a JSON object. This open standard is widely used for web authentication. In our scenario, we’ll use JSON web tokens to safely transfer a user’s data between the client and the server.

JSON web tokens consist of three basic parts: the **header, payload**, and **signature**.

One real example of a JSON web token:



Different token parts are shown with different colors:

**Header :**

The initial section of a JWT is the **Header**, represented as a JSON object encoded in **base64**format. This part is standardized in JWT, and typically, we don’t need to concern ourselves with it. The header includes details such as the token type and the algorithm name.

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

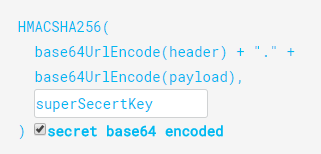
**Payload :**

Following the Header, there’s the Payload, another JavaScript object encoded in **base64**format. The payload holds various attributes about the logged-in user, such as user id, user subject, and details indicating whether a user is an admin. It’s important to note that JSON web tokens are not encrypted and can be decoded with any **base64 decoder**. Therefore, sensitive information should never be included in the Payload.

{   
 "sub": "1234567890",   
 "name": "Chandrashekhar Singh",   
 "iat": 1516239022   
}

**Signature**

Lastly, there’s the **Signature**part. Typically, the server employs the signature to verify if the token holds valid information — the data issued by the server. The signature is a digital signature generated by combining the header and the payload. Importantly, it’s based on a secret key known only to the server.



Exactly, if malicious users attempt to modify the payload values, they must recreate the signature, which requires the secret key known only to the server. On the server side, we can verify the authenticity of the values by comparing the original signature with a new signature computed from the client’s provided values.

By comparing digital signatures, we can effortlessly confirm the integrity of our data. This is the fundamental reason for employing JWT — it provides a secure means of ensuring data integrity through signature verification.

**Creating ASP.NET Core Web API Project**

Now, let’s make a new ASP.NET Core Web API project. We can do this using the .NET Core Command-Line Interface (CLI) or by using Visual Studio. In this article, we’ll use Visual Studio.

We can open the **launchSettings.json** file and modify the applicationUrl property:

"applicationUrl": "https://localhost:5001;http://localhost:5000"

Now, we can start our project.

As a result, we will see our application hosted at [**https://localhost:5001**](https://localhost:5001/)and the browser automatically sends a GET request to **/weatherforecast.**

So far so good.

In the next step, we’ll set up JWT authentication in our application.

**Setting up JWT Authentication :**

To set up JWT authentication in .NET Core, we need to make changes to the **Program.cs** file. If you’re using .NET Core version 8, these modifications should be added to the **ConfigureServices**method inside the **Startup.cs** file.

For simplicity, we’ll place all the code inside the **Program**class. However, it’s considered better practice to use **extension methods** to keep our class clean from additional code lines.

Firstly, let’s install the **Microsoft.AspNetCore.Authentication.JwtBearer** **NuGet package**, which is necessary for working with JWT in the ASP.NET Core app.

Next, let’s insert the code to configure JWT authentication just above the line where we add controllers using **builder.Services.AddControllers()**.

builder.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 = "https://localhost:5001",  
 ValidAudience = "https://localhost:5001",  
 IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("superSecretKey@830"))  
 };  
 });

To make this work, we need to include a few using directives:

using Microsoft.AspNetCore.Authentication.JwtBearer;  
using Microsoft.IdentityModel.Tokens;  
using System.Text;

**Explanation of JWT Configuration:**

First, we set up the JWT authentication middleware by using the **AddAuthentication**method. Then, we define the default authentication scheme as **JwtBearerDefaults.AuthenticationScheme** and **DefaultChallengeScheme**.

Afterward, we enable JWT authentication with the **AddJwtBearer**method, specifying parameters to configure JWT bearer options:

* The issuer checks if the server that created the token is valid **(ValidateIssuer=true).**
* The audience ensures that the token is intended for a valid recipient **(ValidateAudience=true).**
* ValidateLifetime verifies that the token has not expired **(ValidateLifetime=true).**
* ValidateIssuerSigningKey checks if the signing key is valid and trusted by the server **(ValidateIssuerSigningKey=true).**

Additionally, we provide values for the issuer, audience, and the secret key used by the server to create the JWT signature.

For simplicity, we hardcode both the username and password, although it’s recommended to store credentials securely, such as in a database, configuration file, or environment variable.

Finally, we need to take one more step to make our authentication middleware accessible to the application.

app.UseHttpsRedirection();  
  
app.UseAuthentication();  
app.UseAuthorization();  
  
app.MapControllers();  
  
app.Run();

That’s all we have to do to set up JWT authentication in ASP.NET Core. If we were working with a .NET 8 application, **we would perform these configurations in the Configure method within the Startup class.**

**Securing API Endpoints :**

We have a way to get sample weather information through an insecure API endpoint **(/weatherforecast)** where anyone can access the data easily. To address this, we’re planning to create a new secure endpoint **(/api/customers)** to provide a list of customers. This new endpoint will only be accessible to users who are logged in, adding a layer of security.

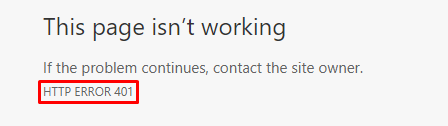
To implement this, we’ll start by creating an empty **CustomersController** in the **Controllers**folder. Inside this controller, we’ll include a Get action method that returns an array of customers. To ensure security, we’ll use the **[Authorize]** attribute to restrict access to only logged-in users.

[Route("api/[controller]")]  
[ApiController]  
public class CustomersController : ControllerBase  
{  
 [HttpGet, Authorize]  
 public IEnumerable<string> Get()  
 {  
 return new string[] { "John Doe", "Chandrashekhar Singh" };  
 }  
}

In order to use the Authorize attribute, we need to include a new using directive at the beginning of the file:

using Microsoft.AspNetCore.Authorization;

Adding the **Authorize**attribute above the **GET**method limits access exclusively to authorized users. This means that only users who are logged in have permission to retrieve the customer list. Consequently, if we attempt to make a request to[**https://localhost:5001/api/customers**](https://localhost:5001/api/customers) directly from the browser’s address bar, instead of receiving a list of customers, we will receive a 401 Not Authorized response.



**Adding Login Endpoint :**

To confirm the identity of users who haven’t revealed their identity, we need to create a login point. This allows users to enter a username and password. If the provided information is correct, we will generate a **JSON web token**for the user.

Before we begin working on the authentication **controller**, we must include a **LoginModel**to store the user’s login details on the server. The **LoginModel**is a basic class with two properties: **UserName and Password**. To organize this, we’ll make a Models folder in the main directory and create a **LoginModel**class inside it.

public class LoginModel  
{  
 public string? UserName { get; set; }  
 public string? Password { get; set; }  
}

Also, let’s create one more class inside the same Models folder:

public class AuthenticatedResponse  
{  
 public string? Token { get; set; }  
}

In a moment, we’ll understand why this class is necessary.

Next, let’s generate the **AuthController**within the **Controllers**folder.

Within the **AuthController**, we’ll include the **Login**action to verify the user’s login details. If the details are correct, we’ll generate a JSON web token. For the purpose of this demonstration, we’ll set the username and password directly in the code to simulate a fake user. Once we’ve confirmed the user’s credentials, we’ll generate a JWT using a secret key. The secret key is crucial for JWT to create the signature.

Now, let’s proceed with implementing the **AuthController**:

[Route("api/[controller]")]  
[ApiController]  
public class AuthController : ControllerBase  
{  
 [HttpPost("login")]  
 public IActionResult Login([FromBody] LoginModel user)  
 {  
 if (user is null)  
 {  
 return BadRequest("Invalid client request");  
 }  
  
 if (user.UserName == "cssingh" && user.Password == "css@123")  
 {  
 var secretKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("superSecretKey@830"));  
 var signinCredentials = new SigningCredentials(secretKey, SecurityAlgorithms.HmacSha256);  
 var tokeOptions = new JwtSecurityToken(  
 issuer: "https://localhost:5001",  
 audience: "https://localhost:5001",  
 claims: new List<Claim>(),  
 expires: DateTime.Now.AddMinutes(5),  
 signingCredentials: signinCredentials  
 );  
  
 var tokenString = new JwtSecurityTokenHandler().WriteToken(tokeOptions);  
  
 return Ok(new AuthenticatedResponse { Token = tokenString });  
 }  
  
 return Unauthorized();  
 }  
}

**Login Action Explanation :**

In the **Login**action, we use the **HttpPost**attribute to specify that it should be invoked through an HTTP POST request. Within this method, we initialize the **SymmetricSecretKey**with the value **superSecretKey@830.** Next, we create a **SigningCredentials**object, specifying the secret key and the algorithm for encoding the token.

The crucial part begins here. The first two steps are standard and require no special attention. The third step is where our focus lies. Here, we instantiate a **JwtSecurityToken**object with key parameters:

1. **Issuer**: A simple string denoting the name of the web server issuing the token.
2. **Audience**: A string representing valid recipients.
3. **Claims**: A list of user roles (e.g., admin, manager, author; roles will be covered in the next post).
4. **Expires**: A DateTime object indicating when the token expires.

Subsequently, we create a string representation of the JWT by invoking the **WriteToken**method on **JwtSecurityTokenHandler**. Finally, we construct a response containing the JWT in an **AuthenticatedResponse** object, which only includes the **Token**property.

**Testing the JWT Authentication :**

Now, launch the application, and we’ll use Postman to send requests.

To begin, send a **POST**request to [**https://localhost:5001/api/auth/login**](https://localhost:5001/api/auth/login)**.** Include the following request body:

{   
 "UserName":"cssingh",   
 "Password": "css@123"   
}

Upon sending the request, you should receive a 200 OK response in the response section, and the JWT string will be present in the response body.

