JWT Authentication in ASP.NET Core Web API

Back to: [ASP.NET Core Web API Tutorials](https://dotnettutorials.net/course/asp-net-core-web-api-tutorials/)

**JWT Authentication in ASP.NET Core Web API**

In this article, I will discuss how to implement **Token-Based Authentication using JWT in ASP.NET Core Web API** Application. Please read our previous article discussing [**Role-Based Basic Authentication in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/role-based-basic-authentication-in-asp-net-core-web-api/). At the end of this article, you will understand the following pointers:

1. **Why Do We Need Token-Based Authentication in ASP.NET Core Web API?**
2. **What is Token-Based Authentication?**
3. **What is JWT Authentication and Its Components?**
4. **How JWT Authentication Works in ASP.NET Core Web API?**
5. **Creating the Authentication Server Application to Generate JWT Token**
6. **Creating API Resource Server Application with JWT Token Validation**
7. **Creating Client Applications to Consume the Resource Server Services with JWT Authentication**
8. **Advantages and Disadvantages of using Token-Based Authentication in ASP.NET Core Web API**

**Why Do We Need Token-Based Authentication in ASP.NET Core Web API?**

The ASP.NET Core Web API is an ideal framework provided by Microsoft to build Web APIs, i.e., HTTP-based services on top of the .NET Core Framework. Once we develop the Web API services, then these services will be consumed by a broad range of clients, such as

* Browsers
* Mobile Applications
* Desktop Applications
* IOTs, etc.

Nowadays, the use of Web APIs is increasing rapidly. So, we should know how to develop Web APIs. However, developing Web APIs alone is not enough if there is no security. Therefore, it is also important for us to implement security for our Web API services, which will be consumed by different clients, such as Browsers, Mobile Devices, Desktop applications, and IoTs.

Nowadays, the most preferred approach to securing Web API resources is to authenticate users in the Web API server using the Signed Token (which contains enough information to identify a particular user), which must be sent to the server by the client with every request. This is called the Token-Based Authentication approach.

**What is Token-Based Authentication?**

In ASP.NET Core Web API, token-based authentication is a popular method for securing Web APIs by validating the identity of users via tokens. This method is preferred in many modern web applications due to its statelessness, which aligns well with the stateless nature of HTTP. The following are the steps or process of how Token-Based Authentication works:

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=39c96553-0d5d-4ecb-6ee1-5703ff7a045e&d_id=118560&imp_id=1073029198925459&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

* **User Login:** The client sends a request to an authentication endpoint with credentials (like username and password). The server validates these credentials against a database or another authentication provider.
* **Token Generation:** Upon successful authentication, the server generates a token. This token contains essential information (claims) about the user, such as the user ID, user name, email, roles, and token expiration time. The most common type of token is the JSON Web Token (JWT).
* **Token Sent to Client:** The server then sends the token back to the client, who must store it securely (usually in local storage or session storage).
* **Client Requests with Token:** For subsequent requests to secure endpoints, the client must include this token, typically in the Authorization header with a Bearer schema. For example, Authorization: Bearer <token>.
* **Server Token Validation:** When the server receives a request with a token, it first validates the token. This validation might involve checking the token’s signature, ensuring it hasn’t expired, and verifying any included claims.
* **Access Granted or Denied:** The server processes the request once the token is validated. If the token is invalid (e.g., expired or tampered with), the server rejects the request, typically with a 401 Unauthorized response, and the client may have to re-authenticate to obtain a new token. The server can also use the claims in the token to perform authorization, such as allowing access to certain methods based on user roles.
* **Token Expiry and Renewal:** Tokens typically have an expiration time. When a token expires, it can either be renewed (i.e., refreshing the token), or the user may need to authenticate again to receive a new one.

**What is JWT Authentication and What Are Its Components in ASP.NET Core Web API?**

JWT stands for JSON Web Token. It enables the secure transmission of information as a JSON object that is digitally signed (using a secret or a public/private key pair) and encrypted. JWTs are commonly used for authentication and information exchange in Restful Web Services. A JWT consists of three main components:

**JWT Header:**

JWT Header contains metadata about the type of token and the cryptographic algorithms used. The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMACSHA256 or RSA. It looks something like this:

**{**

"alg": "HS256"**,**

"typ": "JWT"

**}**

This JSON is then Base64Url encoded to form the first part of the JWT.

**JWT Payload:**

The payload of JWT (JSON Web Token) is the very important part where the bulk of the data is stored. It consists of a set of claims, which are statements about an entity (typically a user) and other data. The payload is represented as a JSON object and can include three types of claims: registered, public, and private claims.

**Registered Claims:** These are a predefined set of claims that are not mandatory but recommended to provide a set of useful claims. They are standardized and provide a consistent way of accessing important information about the token. The following are some of the common registered claims:

Advertisements

* **iss (Issuer):** This claim identifies the principal that issued the JWT. For example, it could be the domain name of the authenticating service.
* **sub (Subject):** This claim identifies the principal that is the subject of the JWT, often a user ID or username that this token represents.
* **aud (Audience):** This claim identifies the recipients the JWT is intended for. It could be the domain name of the client application.
* **exp (Expiration Time):** This numeric date claim defines the time on or after which the JWT will expire. It’s usually represented in Unix time (also known as Epoch time).
* **nbf (Not Before):** Similar to exp, this numeric date claim defines the time before which the JWT must not be accepted for processing.
* **iat (Issued At):** This numeric date claim records the time when the JWT was issued, also in Unix time.
* **jti (JWT ID):** This claim provides a unique identifier for the JWT. It can be used to prevent the JWT from being replayed.

**Public claims:** These are the claims that are not registered but are also not private. They can be used to share information between parties that agree to use them and are usually defined in a namespace that prevents collision with other claims. An example might be a claim detailing user permissions or roles, like “role”: “admin.”

**Private claims:** These are custom claims created to share information between parties that agree to use them and are neither registered nor public claims. For example, an application might include claims related to user access levels or profile data, like “nickname”: “johnny123.”

**Example of a JWT Payload:**

**{**

"iss": "http://example.com"**,**

"sub": "user12345"**,**

"aud": "http://exampleapp.com"**,**

"exp": 1700001234**,**

"nbf": 1699990000**,**

"iat": 1699990200**,**

"jti": "uniqueid123"**,**

"role": "admin"**,**

"nickname": "johnny123"

**}**

This JSON is also Base64Url encoded to form the second part of the JWT.

**JWT Signature:**

The signature of JWT is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn’t changed along the way. To create the signature part, you have to take the encoded header, the encoded payload, a secret, and the algorithm specified in the header. For example, if you are using the HMAC SHA256 algorithm, the signature will be created like this:

HMACSHA256**(**

base64UrlEncode**(**header**)** + "." +

base64UrlEncode**(**payload**)**,

secret**)**

The signature is the third part of the JWT.

**Putting it all together:**

The output is three Base64-URL strings separated by dots. The structure looks like this:

**eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c**

1. The first part is the Base64Url encoded Header.
2. The second part is the Base64Url encoded Payload.
3. The third part is the Base64Url encoded Signature.

This compact form allows JWTs to be sent through URL parameters, HTTP headers, or within an HTTP body. The recipient decodes the JWT to claim the token and verify the signature for security purposes.

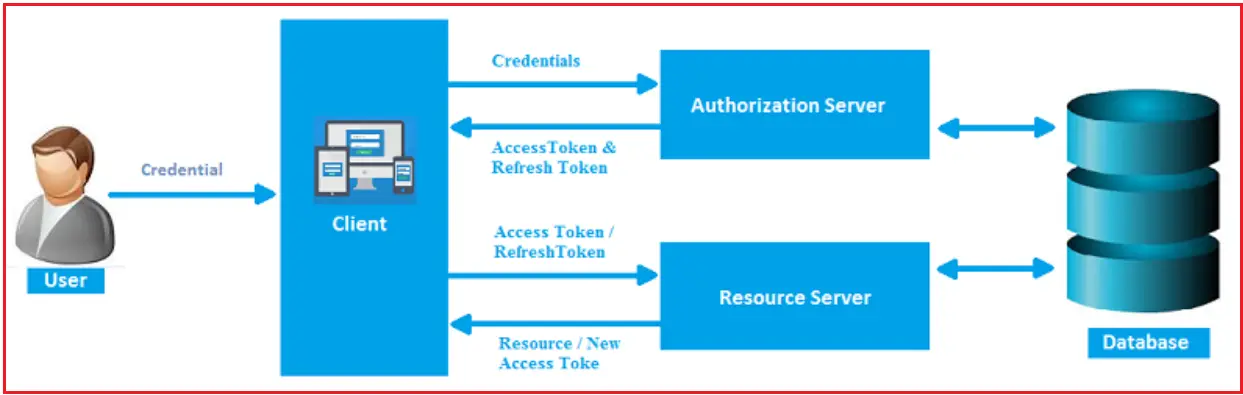
**How JWT Authentication Works in ASP.NET Core Web API?**

JWT (JSON Web Tokens) authentication in ASP.NET Core Web API involves using tokens to secure communication between a client, an authorization server, and a resource server. This authentication mechanism is useful in scenarios where we need to implement stateless authentication across a distributed system. The following Components Involved in JWT Based Token Authentication in ASP.NET Core Web API:

* **Client:** The entity that requests access to resources.
* **Authorization Server:** Validates the credentials provided by the client and issues tokens.
* **Resource Server:** This server hosts the protected resources and uses the token sent by the client to authenticate and authorize requests.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=39c96553-0d5d-4ecb-6ee1-5703ff7a045e&d_id=118560&imp_id=7045761566951637&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

For a better understanding of how it works in ASP.NET Core Web API, please have a look at the following diagram:

**Authentication at the Authorization Server**

The client sends a login request with credentials (username and password) to the authorization server. If the credentials are valid, the authorization server generates a JWT. This token includes claims about the user and additional data needed for authorization, such as user roles or permissions. The authorization server signs the JWT using a secret key or a public/private key pair. The client receives this token.

**Token Use at the Resource Server**

The client makes a request to the resource server for resources and includes the JWT in the HTTP Authorization header. The resource server validates the token’s signature and checks the token’s validity (expiry, issuer, intended audience, etc.). If the token is valid, the server checks the claims in the token to determine if the client has the necessary permissions to access the requested resources. Let us proceed and implement the same step-by-step:

**Creating the Authentication Server Application**

The Authentication server will handle user authentication and token generation. So, create a new ASP.NET Core Web API Project named **JWTAuthServer**. Once you create the Project, then please install the following Package:

**Microsoft.AspNetCore.Authentication.JwtBearer**

You can install the Package using NuGet Package Manager for the Solution or by executing the following command in the Package Manager Console:

**Install-Package Microsoft.AspNetCore.Authentication.JwtBearer**

**Creating User Model:**

Now, we need to create the User model, which will hold the user information. So, create a class file named **User.cs** within the Models folder and then copy and paste the following code:

**namespace** *JWTAuthServer.Models*

**{**

**public** **class** User

**{**

// This property represents the unique identifier for the user in the database.

**public** **int** Id **{** **get**; **set**; **}**

// This property is typically used as a unique identifier for login purposes.

**public** **string** Username **{** **get**; **set**; **}**

// This property stores the user's password, which should be securely hashed before storage.

**public** **string** Password **{** **get**; **set**; **}**

// This stores the user's email address, used for communication or password recovery.

**public** **string** Email **{** **get**; **set**; **}**

// This list holds the roles assigned to the user, used for authorization.

// It is initialized as a new list of strings, meaning it starts empty but ready to be added to.

**public** List**<string>** Roles **{** **get**; **set**; **}** = new List**<string>()**;

**}**

**}**

**Creating User Store**

Whenever a new user registers with our application, we need to store the user details in the database. But, for simplicity, let us create a list to store all users of our application. So, create a class file named **UserStore.cs** within the Models folder and copy and paste the following code.

**namespace** *JWTAuthServer.Models*

**{**

**public** **class** UserStore

**{**

// Declare a public static list of User objects named 'Users'.

// Being static, this list is shared across all instances of the UserStore class and accessible without creating an instance of the class.

// This list is initialized with predefined user data.

**public** **static** List**<**User**>** Users = new List**<**User**>**

**{**

new User **{** Id=1, Username = "admin", Password = "password", Email="admin@Example.com", Roles = new List**<string>** **{** "Admin", "User" **}** **}**,

new User **{** Id=2, Username = "user", Password = "password", Email="user@Example.com", Roles = new List**<string>** **{** "User" **}** **}**,

new User **{** Id=3, Username = "test", Password = "password", Email="test@Example.com", Roles = new List**<string>** **{** "Admin" **}** **}**

**}**;

**}**

**}**

**Creating Login Model:**

Now, we need to create a model that should accept the user details (Username and Password), then validate the user and generate the token if the provided username and password are valid. So, create a class file named **Login.cs**within the Models folder and then copy and paste the following code.

**using** *System.ComponentModel.DataAnnotations;*

**namespace** *JWTAuthServer.Models*

**{**

**public** **class** Login

**{**

// This property represents the username input from a user during the login process.

// Usernames are essential for identifying the user in the system.

// The Required attribute makes sure that the username field is not empty.

// The StringLength attribute can be used to specify the maximum length of the username.

**[**Required**(**ErrorMessage = "Username is required."**)]**

**[**StringLength**(**100, ErrorMessage = "Username must be less than 100 characters."**)]**

**public** **string** Username **{** **get**; **set**; **}**

// This property is used to store the password input from a user during the login process.

// Passwords need secure handling to prevent unauthorized access.

// The Required attribute ensures the password field is not left empty.

// The StringLength attribute can specify minimum and maximum password lengths, enhancing security.

**[**Required**(**ErrorMessage = "Password is required."**)]**

**[**StringLength**(**100, MinimumLength = 6, ErrorMessage = "Password must be between 6 and 100 characters."**)]**

**public** **string** Password **{** **get**; **set**; **}**

**}**

**}**

**Creating API Controller:**

Create a controller that will authenticate a user and issue a JWT taken. So, create an API Empty controller named **UsersController** within the Controllers folder and copy and paste the following code into it.

**using** *JWTAuthServer.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.IdentityModel.Tokens;*

**using** *System.IdentityModel.Tokens.Jwt;*

**using** *System.Security.Claims;*

**using** *System.Text;*

**namespace** *JWTAuthServer.Controllers*

**{**

**[**Route**(**"api/[controller]"**)]**

**[**ApiController**]**

**public** **class** AuthController : ControllerBase

**{**

// Private field to hold the configuration settings injected through the constructor.

**private** **readonly** IConfiguration \_configuration;

// Constructor that accepts IConfiguration and initializes the \_configuration field.

**public** AuthController**(**IConfiguration configuration**)**

**{**

\_configuration = configuration;

**}**

// Action method responding to POST requests on "api/Users/Login".

// Expects a Login model in the request body.

**[**HttpPost**(**"Login"**)]**

**public** IActionResult Login**([**FromBody**]** Login request**)**

**{**

// Checks if the provided model (request) is valid based on data annotations in the Login model.

**if** **(**ModelState.IsValid**)**

**{**

// Searches for a user in a predefined user store that matches both username and password.

var user = UserStore.Users.FirstOrDefault**(**u =**>** u.Username == request.Username && u.Password == request.Password**)**;

// Checks if the user object is null, which means no matching user was found.

**if** **(**user == **null)**

**{**

// Returns a 401 Unauthorized response with a custom message.

**return** Unauthorized**(**"Invalid user credentials."**)**;

**}**

// Calls a method to generate a JWT token for the authenticated user.

var token = IssueToken**(**user**)**;

// Returns a 200 OK response, encapsulating the JWT token in an anonymous object.

**return** Ok**(**new **{** Token = token **})**;

**}**

// If the model state is not valid, returns a 400 Bad Request response with a custom message.

**return** BadRequest**(**"Invalid Request Body"**)**;

**}**

// Private method to generate a JWT token using the user's data.

**private** **string** IssueToken**(**User user**)**

**{**

// Creates a new symmetric security key from the JWT key specified in the app configuration.

var securityKey = new SymmetricSecurityKey**(**Encoding.UTF8.GetBytes**(**\_configuration**[**"Jwt:Key"**]))**;

// Sets up the signing credentials using the above security key and specifying the HMAC SHA256 algorithm.

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

// Defines a set of claims to be included in the token.

var claims = new List**<**Claim**>**

**{**

// Custom claim using the user's ID.

new Claim**(**"Myapp\_User\_Id", user.Id.ToString**())**,

// Standard claim for user identifier, using username.

new Claim**(**ClaimTypes.NameIdentifier, user.Username**)**,

// Standard claim for user's email.

new Claim**(**ClaimTypes.Email, user.Email**)**,

// Standard JWT claim for subject, using user ID.

new Claim**(**JwtRegisteredClaimNames.Sub, user.Id.ToString**())**

**}**;

// Adds a role claim for each role associated with the user.

user.Roles.ForEach**(**role =**>** claims.Add**(**new Claim**(**ClaimTypes.Role, role**)))**;

// Creates a new JWT token with specified parameters including issuer, audience, claims, expiration time, and signing credentials.

var token = new JwtSecurityToken**(**

issuer: \_configuration**[**"Jwt:Issuer"**]**,

audience: \_configuration**[**"Jwt:Audience"**]**,

claims: claims,

expires: DateTime.Now.AddHours**(**1**)**, // Token expiration set to 1 hour from the current time.

signingCredentials: credentials**)**;

// Serializes the JWT token to a string and returns it.

**return** new JwtSecurityTokenHandler**()**.WriteToken**(**token**)**;

**}**

**}**

**}**

**Understanding Login Method**

The primary objective of the Login method is to authenticate users based on their credentials (username and password). Upon successful authentication, it generates and returns a JWT token that the client can use for subsequent API requests to prove their authenticated status. The following things are done in the IssueToken method to generate a JWT Token:

* **Receive Credentials:** The method accepts a Login model containing the user’s credentials (username and password) via a POST request.
* **Validate Credentials:** It validates the provided credentials against a user store (e.g., a database or in-memory store).
* **Token Generation:** If the credentials are valid, the method uses the IssueToken method to generate a JWT for the authenticated user.
* **Response:** For authenticated users, it returns the JWT token in the response body. If the credentials are invalid or validation fails, it returns an error response (either 401 Unauthorized for invalid credentials or 400 Bad Request for invalid input data).

**Understanding IssueToken Method**

The purpose of the private IssueToken method is to generate a JWT for a user who has successfully authenticated. This token serves as an access token for the user to make authorized API calls. The following things are done in the IssueToken method to generate a JWT Token:

* **Security Key Configuration:** It configures a symmetric security key from a secret key defined in the application’s configuration.
* **Signing Credentials:** Establishes the signing credentials using the symmetric key and the HMAC SHA256 algorithm.
* **Claims Definition:** Assembles a set of claims to be included in the JWT. These claims typically contain user identification information such as user ID, username, and roles, which are essential for authorizing user actions on the server.
* **Token Creation:** This function constructs the JWT using the defined parameters, such as issuer, audience, claims, expiration time, and signing credentials.
* **Token Serialization:** Serializes the JWT into a string form and returns it.

**Modifying appsettings.json File:**

Please make sure to have the issuer, audience, and key settings configured in your appsettings.json file, which is used by the JWT Authentication service in the Program class. So, please modify the **appsettings.json** file as follows:

**{**

"Logging": **{**

"LogLevel": **{**

"Default": "Information"**,**

"Microsoft.AspNetCore": "Warning"

**}**

**},**

"AllowedHosts": "\*"**,**

"Jwt": **{**

"Key": "KHPK6Ucf/zjvU4qW8/vkuuGLHeIo0l9ACJiTaAPLKbk="**,** //Secret Key

"Issuer": "https://localhost:7035"**,** //Authentication Server Domain URL Base Address

"Audience": "http://localhost:5001" //Client Application Domain URL Base Address

**}**

**}**

**Ignoring Camel Case Naming Convention:**

We want the JSON property name to be displayed as it is. So, modify the AddControllers service as follows in the Program.cs class file:

builder.Services.AddControllers**()**

.AddJsonOptions**(**options =**>**

**{**

// This will use the property names as defined in the C# model

options.JsonSerializerOptions.PropertyNamingPolicy = **null**;

**})**;

**Creating a Console Application for Generating Secret Key:**

Please create a Dot Net Console application and then modify the Program class as follows to generate the Secret Key. The following code is self-explained, so please read the comment lines for a better understanding.

**using** *System.Security.Cryptography;*

**namespace** *KeyGeneratorApp*

**{**

**class** Program

**{**

**static** **void** Main**(string[]** args**)**

**{**

// Allocate an array for storing random bytes.

// This array has 32 bytes, equivalent to 256 bits.

var randomBytes = new **byte[**32**]**;

// Create an instance of a cryptographic random number generator.

**using** **(**var rng = RandomNumberGenerator.Create**())**

**{**

// Fill the previously defined array with securely generated random bytes.

rng.GetBytes**(**randomBytes**)**;

**}** // The 'using' block ensures that the RandomNumberGenerator object is properly disposed of after use.

// Convert the random bytes array into a Base64 string to make it readable and easy to handle as text.

Console.WriteLine**(**Convert.ToBase64String**(**randomBytes**))**;

// Pause the program and wait for the user to press a key so they can see the output before the console window closes.

Console.ReadKey**()**;

**}**

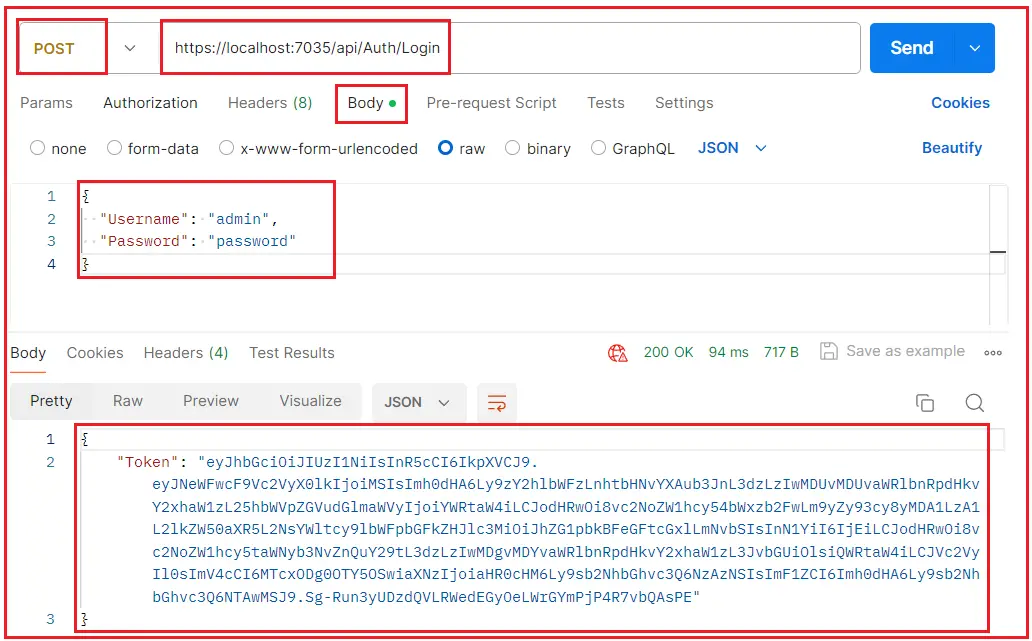
**}**

**}**

Now, run the above code for each client and get the Secret key.

**Testing the JWT Token Generation:**

With this, our Authentication Server implementation is done, which will validate the user details and generate the JWT token. So, to test the functionality, run the application and access the Login endpoint using any client tool like Postman, Swagger, or Fiddler by providing the User Name and Password, and it should generate the JWT token as shown

in the below image:

**Understanding the Generated Token:**

As you can see, the token is generated in three parts. To understand the different parts of the generated token, please visit the [**https://jwt.io/**](https://jwt.io/)website. Replace your generated token in the Encoded section, and you should see the different parts in different colors. In the Decoded section, you will see the details of the token.

**Creating API Resource Server Application:**

Next, we need to create an ASP.NET Core Web API Project named **APIResourceServer.**This will be our Resource Server, exposing the API endpoints to be consumed by clients with JWT authentication.

Once you create the Resource Server Project, please install the Package: **Microsoft.AspNetCore.Authentication.JwtBearer**. You can install the Package using NuGet Package Manager for Solution or by executing the following command in the Package Manager Console:

**Install-Package Microsoft.AspNetCore.Authentication.JwtBearer**

**Configure JWT Authentication to Validate the Token in the Resource Server:**

We need to ensure that the Resource Server validates the JWT tokens coming from the client. We need to configure the JWT Bearer token authentication within the **Program.cs** class file. So, modify the Program class as follows:

**using** *Microsoft.AspNetCore.Authentication.JwtBearer;*

**using** *Microsoft.IdentityModel.Tokens;*

**using** *System.Text;*

**namespace** *APIResourceServer*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

var builder = WebApplication.CreateBuilder**(**args**)**;

// Add services to the container.

builder.Services.AddControllers**()**

.AddJsonOptions**(**options =**>**

**{**

// This will use the property names as defined in the C# model

options.JsonSerializerOptions.PropertyNamingPolicy = **null**;

**})**;

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer**()**;

builder.Services.AddSwaggerGen**()**;

// Configure JWT authentication

builder.Services.AddAuthentication**(**JwtBearerDefaults.AuthenticationScheme**)**

.AddJwtBearer**(**options =**>**

**{**

options.TokenValidationParameters = new TokenValidationParameters

**{**

ValidateIssuer = **true**,

ValidateAudience = **true**,

ValidateLifetime = **true**,

ValidateIssuerSigningKey = **true**,

ValidIssuer = builder.Configuration**[**"Jwt:Issuer"**]**,

ValidAudience = builder.Configuration**[**"Jwt:Audience"**]**,

IssuerSigningKey = new SymmetricSecurityKey**(**Encoding.UTF8.GetBytes**(**builder.Configuration**[**"Jwt:Key"**]))**

**}**;

**})**;

var app = builder.Build**()**;

// Configure the HTTP request pipeline.

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseSwagger**()**;

app.UseSwaggerUI**()**;

**}**

app.UseHttpsRedirection**()**;

// Registers the authentication middleware that uses the default authentication scheme set by AddAuthentication.

app.UseAuthentication**()**;

// Registers the authorization middleware to enforce authorization policies on the app's routes.

app.UseAuthorization**()**;

// Maps controller routes.

app.MapControllers**()**;

app.Run**()**;

**}**

**}**

**}**

**Understanding JWT Service and Middleware Components:**

**AddAuthentication:** AddAuthentication is used to configure the authentication services in an ASP.NET Core application. When configuring it with **JwtBearerDefaults.AuthenticationScheme**, it sets up the JWT Bearer as the default authentication scheme. This is essential for enabling the application to accept and validate JWT tokens for authentication.

**UseAuthentication:** UseAuthentication is a middleware component that is added to the HTTP request pipeline and is responsible for invoking the authentication schemes configured in AddAuthentication. For JWT, it attempts to validate the JWT token present in the request’s Authorization header, checks if the token is valid, and sets the user’s identity based on the token’s claims.

**UseAuthorization:** UseAuthorization is another middleware component that checks the user’s identity established by UseAuthentication and enforces authorization policies on routes, controllers, or actions. This is important for securing endpoints to ensure that only authenticated and authorized users can access them.

**Modifying appsettings.json File:**

Please make sure to have the issuer, audience, and key settings configured in your appsettings.json file same as the Authentication Server Application. So, please modify the **appsettings.json** file as follows:

**{**

"Logging": **{**

"LogLevel": **{**

"Default": "Information"**,**

"Microsoft.AspNetCore": "Warning"

**}**

**},**

"AllowedHosts": "\*"**,**

"Jwt": **{**

"Key": "KHPK6Ucf/zjvU4qW8/vkuuGLHeIo0l9ACJiTaAPLKbk="**,** //Secret Key

"Issuer": "https://localhost:7035"**,** //Authentication Server Domain URL

"Audience": "http://localhost:5001" //Client Application Domain URL

**}**

**}**

**Creating API Controller:**

Next, we need to Create the API Controller that exposes the endpoints to be consumed by the clients. So, create an Empty API Controller named **UsersController** within the Controllers folder and then copy and paste the following code:

**using** *Microsoft.AspNetCore.Authorization;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *APIResourceServer.Controllers*

**{**

**[**Route**(**"api/[controller]"**)]**

**[**ApiController**]**

**public** **class** UsersController : ControllerBase

**{**

// Static list to simulate database of users.

**private** **static** List**<**User**>** \_users = new List**<**User**>**

**{**

new User **{** Id = 1, Username = "admin", Email = "admin@example.com" **}**,

new User **{** Id = 2, Username = "user", Email = "user@example.com" **}**,

new User **{** Id = 3, Username = "test", Email="test@example.com" **}**

**}**;

// GET method to retrieve all users.

**[**HttpGet**]**

//To secure the endpoints with JWT authentication, we need to apply the [Authorize] attribute

**[**Authorize**]**

**public** ActionResult**<**List**<**User**>>** GetAllUsers**()**

**{**

// Return the static list of users.

**return** \_users;

**}**

// GET method with a parameter to retrieve a specific user by their ID.

**[**HttpGet**(**"{id}"**)]**

**[**Authorize**]**

**public** ActionResult**<**User**>** GetUser**(int** id**)**

**{**

// Searches the list for a user with the specified ID.

var user = \_users.FirstOrDefault**(**u =**>** u.Id == id**)**;

// If no user is found, return a 404 Not Found response.

**if** **(**user == **null)**

**return** NotFound**()**;

// If found, return the user.

**return** user;

**}**

// POST method to create a new user.

**[**HttpPost**]**

**[**Authorize**]**

**public** ActionResult**<**User**>** CreateUser**([**FromBody**]** User user**)**

**{**

// Adds the new user to the list.

\_users.Add**(**user**)**;

// Return a response with the location of the newly created user.

**return** CreatedAtAction**(nameof(**GetUser**)**, new **{** id = user.Id **}**, user**)**;

**}**

// PUT method to update an existing user.

**[**HttpPut**(**"{id}"**)]**

**[**Authorize**]**

**public** ActionResult**<**User**>** UpdateUser**(int** id, **[**FromBody**]** User user**)**

**{**

// Find the index of the user in the list.

var index = \_users.FindIndex**(**u =**>** u.Id == id**)**;

// If no user is found at that index, return a 404 Not Found.

**if** **(**index == -1**)**

**return** NotFound**()**;

// Update the user at the specific index.

\_users**[**index**]** = user;

// Return a 204 No Content response, indicating the update was successful.

**return** NoContent**()**;

**}**

// DELETE method to remove a user by ID.

**[**HttpDelete**(**"{id}"**)]**

**[**Authorize**]**

**public** IActionResult DeleteUser**(int** id**)**

**{**

// Find the index of the user to be deleted.

var index = \_users.FindIndex**(**u =**>** u.Id == id**)**;

// If no user is found at that index, return a 404 Not Found.

**if** **(**index == -1**)**

**return** NotFound**()**;

// Remove the user from the list.

\_users.RemoveAt**(**index**)**;

// Return a 204 No Content response, indicating the deletion was successful.

**return** NoContent**()**;

**}**

**}**

// Definition of the User model used in the controller.

**public** **class** User

**{**

**public** **int** Id **{** **get**; **set**; **}**

**public** **string** Username **{** **get**; **set**; **}**

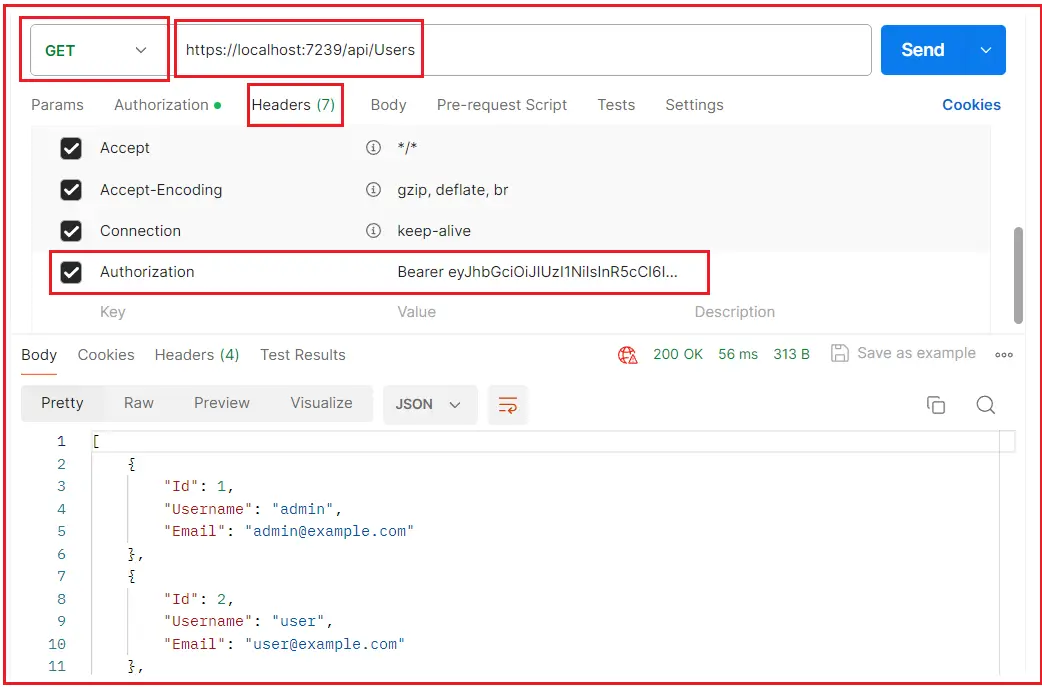
**public** **string** Email **{** **get**; **set**; **}**

**}**

**}**

**Accessing the Resource Server Services with JWT Authentication:**

Now, let us see how to access the Resource Server services with JWT Authentication. We have already discussed how to call the Login API End point of the Authentication Server to get the JWT Token. Now, I assume you have the JWT Token. So, open then Postman and issue a Request to the API Endpoints, which will return all the User data as shown in the below image:



**Creating Client Application for Consuming Resources with JWT Authentication:**

Now, we will create a Dot Net Console Application Consuming the Resource Server API Endpoints with JWT Authentication. That means first, it will call the Authentication Server Login endpoint to get the JWT Token, and then using that JWT Token, it will consume all the Resource Server services. So, create a Dot Net Console Application named **JWTClientApp** and then modify the Program class as follows:

**using** *System.Net;*

**using** *System.Net.Http.Headers;*

**using** *System.Text;*

**using** *System.Text.Json;*

**namespace** *JWTClientApp*

**{**

**public** **class** Program

**{**

// HttpClient instance used to make HTTP requests.

**private** **static** **readonly** HttpClient httpClient = new HttpClient**()**;

// Base URL for the API endpoints on the resource server.

**private** **static** **readonly** **string** baseUrl = "https://localhost:7239/api/users";

// The entry point of the program.

**static** **async** Task Main**(string[]** args**)**

**{**

**try**

**{**

// Authenticate and get a JWT token from the auth server.

var token = **await** AuthenticateAndGetToken**()**;

Console.WriteLine**(**"Token received: " + token**)**;

// Use the token to perform CRUD operations.

Console.WriteLine**(await** GetUser**(**token, 1**))**; // Get the user with ID 1.

Console.WriteLine**(await** CreateUser**(**token, new User **{** Id = 4, Username = "newuser", Email = "newuser@example.com" **}))**;

Console.WriteLine**(await** UpdateUser**(**token, 4, new User **{** Id = 4, Username = "updatedUser", Email = "updateduser@example.com" **}))**;

Console.WriteLine**(await** DeleteUser**(**token, 4**))**; // Delete the user with ID 4.

Console.ReadKey**()**;

**}**

**catch** **(**Exception ex**)**

**{**

Console.WriteLine**(**"Error: " + ex.Message**)**;

**}**

**}**

// Method to authenticate by sending credentials and receiving a JWT token.

**static** **async** Task**<string>** AuthenticateAndGetToken**()**

**{**

// URL for the login endpoint.

var loginUrl = "https://localhost:7035/api/Users/Login";

// Request body containing login credentials.

var loginRequestBody = new **{** Username = "admin", Password = "password" **}**;

var requestContent = new StringContent**(**JsonSerializer.Serialize**(**loginRequestBody**)**, Encoding.UTF8, "application/json"**)**;

// Make a POST request to the login URL.

var response = **await** httpClient.PostAsync**(**loginUrl, requestContent**)**;

**if** **(**!response.IsSuccessStatusCode**)**

**{**

// If login fails, read and return the error message.

var errorContent = **await** response.Content.ReadAsStringAsync**()**;

**return** $"Login failed: {errorContent}";

**}**

// Read the response content and extract the token.

var loginResponseContent = **await** response.Content.ReadAsStringAsync**()**;

var tokenObject = JsonSerializer.Deserialize**<**JsonElement**>(**loginResponseContent**)**;

**return** tokenObject.GetProperty**(**"Token"**)**.GetString**()**;

**}**

// Method to get a user's data.

**static** **async** Task**<string>** GetUser**(string** token, **int** userId**)**

**{**

// Set the Authorization header with the received JWT token.

httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue**(**"Bearer", token**)**;

var response = **await** httpClient.GetAsync**(**baseUrl + $"/{userId}"**)**;

**return** **await** ProcessResponse**(**response, "Get User"**)**;

**}**

// Method to create a new user.

**static** **async** Task**<string>** CreateUser**(string** token, User user**)**

**{**

// Set the Authorization header and make a POST request to create a new user.

httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue**(**"Bearer", token**)**;

var response = **await** httpClient.PostAsync**(**baseUrl,

new StringContent**(**JsonSerializer.Serialize**(**user**)**, Encoding.UTF8, "application/json"**))**;

**return** **await** ProcessResponse**(**response, "Create User"**)**;

**}**

// Method to update an existing user.

**static** **async** Task**<string>** UpdateUser**(string** token, **int** userId, User user**)**

**{**

// Set the Authorization header and make a PUT request to update the user.

httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue**(**"Bearer", token**)**;

var response = **await** httpClient.PutAsync**(**baseUrl + $"/{userId}",

new StringContent**(**JsonSerializer.Serialize**(**user**)**, Encoding.UTF8, "application/json"**))**;

**return** **await** ProcessResponse**(**response, "Update User"**)**;

**}**

// Method to delete a user.

**static** **async** Task**<string>** DeleteUser**(string** token, **int** userId**)**

**{**

// Set the Authorization header and make a DELETE request to remove the user.

httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue**(**"Bearer", token**)**;

var response = **await** httpClient.DeleteAsync**(**baseUrl + $"/{userId}"**)**;

**return** **await** ProcessResponse**(**response, "Delete User"**)**;

**}**

// Helper method to process and format the response from HTTP requests.

**private** **static** **async** Task**<string>** ProcessResponse**(**HttpResponseMessage response, **string** action**)**

**{**

var responseBody = **await** response.Content.ReadAsStringAsync**()**;

**if** **(**response.IsSuccessStatusCode**)**

**{**

**return** $"{action} succeeded: {responseBody}";

**}**

// Handle different HTTP status codes and customize the error message.

**switch** **(**response.StatusCode**)**

**{**

**case** HttpStatusCode.Unauthorized:

**return** $"{action} Failed: Unauthorized - Token may be invalid or expired";

**case** HttpStatusCode.Forbidden:

**return** $"{action} Failed: Forbidden - Insufficient permissions";

**default**:

**return** $"{action} Failed: {response.StatusCode} - {responseBody}";

**}**

**}**

**}**

// Definition of the User model.

**public** **class** User

**{**

**public** **int** Id **{** **get**; **set**; **}**

**public** **string** Username **{** **get**; **set**; **}**

**public** **string** Email **{** **get**; **set**; **}**

**}**

**}**

**Output:**

**Advantages and Disadvantages of using Token-Based Authentication in ASP.NET Core Web API**

Token-based authentication in ASP.NET Core Web API offers several advantages but also comes with some disadvantages. They are as follows:

**Advantages of using Token-Based Authentication in ASP.NET Core Web API**

* Being stateless means the Load balancers can send a client request to any of the servers since there’s no session-specific dependency on a single server.
* Tokens can be designed to be self-contained, carrying all the user information necessary to handle a request, which reduces the risk of CSRF (Cross-Site Request Forgery) attacks. Tokens can also be encrypted for additional security.
* Token-based authentication can easily support cross-platform authentication across various types of clients (web, mobile, and desktop applications) since it uses standard HTTP headers to carry tokens.

**Disadvantages of using Token-Based Authentication in ASP.NET Core Web API**

* Tokens must be securely stored on the client side. Improper handling can lead to vulnerabilities, particularly with XSS (Cross-Site Scripting) attacks, where an attacker might steal the token and gain unauthorized access.
* Managing the token’s expiration is important. If the expiration is too short, users may be required to authenticate frequently. If too long, it might increase the risk of misuse of the token if it’s compromised.
* If the data payload includes too many claims, it can increase in size. Encoding, decoding, and validating tokens on every request can also add computational overhead.

In the next article, I will discuss how to implement [**Refresh Token in ASP.NET Core Web API**](https://dotnettutorials.net/lesson/refresh-token-in-asp-net-core-web-api-using-jwt-authentication/) Application. Here, in this article, I explain Token-Based Authentication using JWT in ASP.NET Core Web API Application with an Example. I hope you enjoy this JWT Authentication in ASP.NET Core Web API article.

Tips :

* 1. Execute console application as a separate instance of VS Studio
  2. Whatever the generated **secret key** is there, copy it in appsettings.json

"Jwt": {

"Key": "X8v3HypneHSAU7IzOdoCOXAH5fV9NZgiD5tFbMed0SU=", //**Secret Key**

"Issuer": "https://localhost:**7108**", //Authentication Server Domain URL Base Address

"Audience": "http://localhost:5001" //Client Application Domain URL Base Address

}

* 1. Keep the same port number of your project which is currently running in the browser.