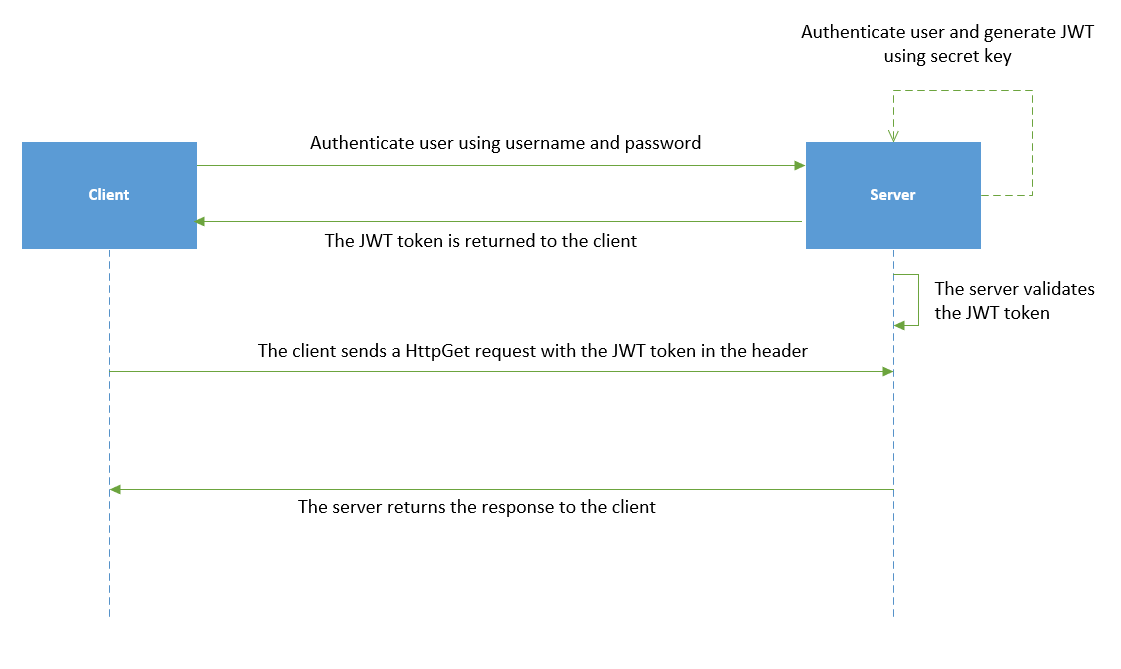
***JWT Token Authentication in .NET Core***

JSON Web Token is an open standard (RFC 7519) that defines a safe, compact, and self-contained, secured way for transmission of information between a sender and a receiver through a URL, a POST parameter, or inside the HTTP Header. It should be noted that the information to be transmitted securely between two parties is represented in JSON format and it is cryptographically signed to verify its authenticity. JWT is typically used for implementing authentication and authorization in Web applications. Because JWT is a standard, all JWTs are tokens but the reverse is not true. You can work with JSON Web Tokens in .NET, Python, Node.js, Java, PHP, Ruby, Go, JavaScript, etc.



**Figure 1:** JWT authentication at work

JWT is represented as a combination of three base64url encoded parts concatenated with period ('.') characters and comprises the following three sections:

* Header
* Payload
* Signature

### Header Section

This section provides metadata about the type of data and the algorithm to be used to encrypt the data that is to be transferred. The JWT header comprises three sections - these include: the metadata for the token, the type of signature, and the encryption algorithm. It comprises two properties, i.e., “alg” and “typ”. Although the former relates to the cryptography algorithm used, i.e., HS256 in this case, the latter is used to specify the type of the token, i.e., JWT in this case.

#### 

{

"typ": "JWT",

"alg": "HS256"

}

### Payload

The payload represents the actual information in JSON format that is to be transmitted over the wire. The code snippet given below illustrates a simple payload.

#### 

{

"sub": "1234567890",

"name": "Joydip Kanjilal",

"admin": true,

"jti": "cdafc246-109d-4ac9-9aa1-eb689fad9357",

"iat": 1611497332,

"exp": 1611500932

}

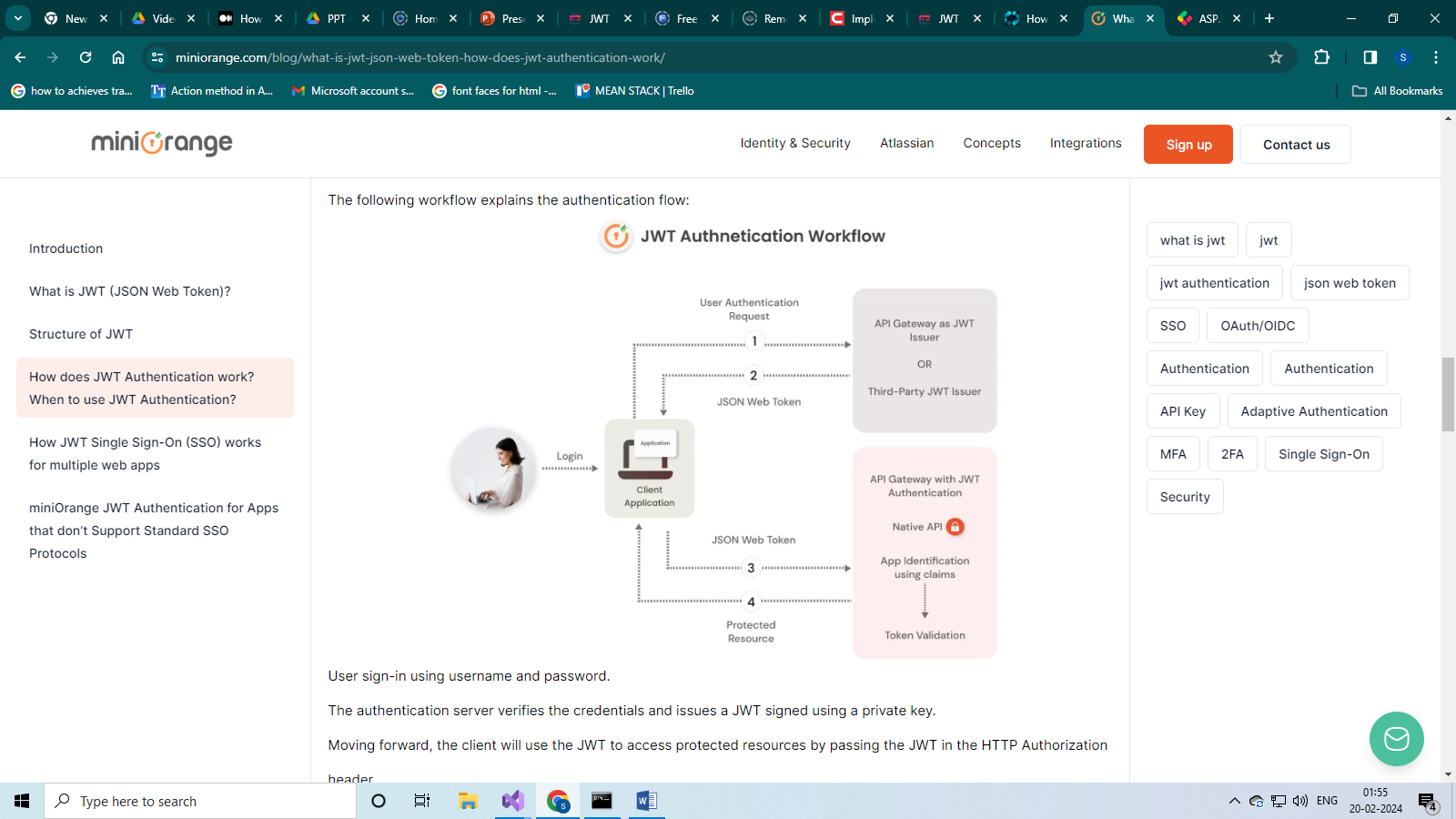
The payload typically contains claims, the identity information of the user, the allowed permissions, etc. You can use claims to transmit additional information. These are also called JWT claims and are of two types: Reserved and Custom. Here's a list of the reserved claims:

* iss: This represents the issuer of the token.
* sub: This is the subject of the token.
* aud: This represents the audience of the token.
* exp: This is used to define token expiration.
* nbf: This is used to specify the time before which the token must not be processed.
* iat: This represents the time when the token was issued.
* jti: This represents a unique identifier for the token.

You can also use custom claims, which can be added to the token using a rule.

### Signature

The signature adheres to the JSON Web Signature (JWS) specification and is used to verify the integrity of the data transferred over the wire. It comprises a hash of the header, the payload, and the secret, and is used to ensure that the message was not changed while being transmitted. The final signed token is created by adhering to the JSON Web Signature (JWS) specification. The encoded JWT header and as well as the encoded JWT payload is combined and then it's signed using a strong encryption algorithm such as HMAC SHA 256.



# Step 1: Create a .NET Core 8 Web API Project

Start by creating a new .NET Core 8 Web API project in your preferred development environment. You can use Visual Studio

 Once the project is created, you can proceed with the following steps.

This is what our basic API controller appears like. Its sole purpose is to provide the response “Hello World”

namespace JwtInDotnetCore.Controllers  
{  
 [Route("api/[controller]")]  
 [ApiController]  
 public class HelloWorldController : ControllerBase  
 {  
 [HttpGet]  
 public IActionResult Get()   
 {  
 return Ok("Hello World");  
 }  
 }  
}

## Install NuGet Package

## You’ll need some NuGet packages to handle JWT authentication. Open your project’s .csproj file and add the following package references:

We need to install NuGet Packages

* Microsoft.AspNet Core.Authentication.JwtBearer
* System.IdentityModel.Tokens.Jwt.
* Microsoft Entity Framework Core Sql Server.
* Microsoft Entity Framework Core Tools

## Configure appsetting.json (Asp.Net Core JWT )

Now after the NuGet packages are installed, we need to modify our appsetting.json file and add the secret key and other values and the database connection string.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | {    "Logging": {      "LogLevel": {        "Default": "Information",        "Microsoft.AspNetCore": "Warning"      }    },    "ConnectionStrings": {      "connection": "Server=(local)\\MSSQL;Database=Inventory;Trusted\_Connection=True;MultipleActiveResultSets=true"    },    "AllowedHosts": "\*",  **"JWTToken": {**  **"Key": "This is Secret key",**  **"Issuer": "**[**https://labpys.com**](https://labpys.com/)**",**  **"Audience" : "labpys.com"**    } |

## Configure Program.cs (Asp.Net Core)

Here we need to configure and register dependencies services and set the ordering of the request pipeline We also, need to add **UseAuthentication()** and **UseAuthorization()** middleware and configure addAuthentication with addJwtBearer.

using Microsoft.AspNetCore.Authentication.JwtBearer;  
using Microsoft.IdentityModel.Tokens;  
using System.Text;  
  
var builder = WebApplication.CreateBuilder(args);  
  
//Jwt configuration starts here  
var jwtIssuer = builder.Configuration.GetSection("Jwt:Issuer").Get<string>();  
var jwtKey = builder.Configuration.GetSection("Jwt:Key").Get<string>();  
  
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)  
 .AddJwtBearer(options =>  
 {  
 options.TokenValidationParameters = new TokenValidationParameters  
 {  
 ValidateIssuer = true,  
 ValidateAudience = true,  
 ValidateLifetime = true,  
 ValidateIssuerSigningKey = true,  
 ValidIssuer = jwtIssuer,  
 ValidAudience = jwtIssuer,  
 IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey))  
 };  
 });  
//Jwt configuration ends here  
  
builder.Services.AddControllers();  
builder.Services.AddEndpointsApiExplorer();  
builder.Services.AddSwaggerGen();  
  
var app = builder.Build();  
  
if (app.Environment.IsDevelopment())  
{  
 app.UseSwagger();  
 app.UseSwaggerUI();  
}  
  
app.UseHttpsRedirection();  
  
app.UseAuthentication();  
app.UseAuthorization();  
  
  
app.MapControllers();  
  
app.Run();

# Step 4: Generate JWT Tokens

using Microsoft.AspNetCore.Mvc;  
using Microsoft.IdentityModel.Tokens;  
using System.IdentityModel.Tokens.Jwt;  
using System.Text;  
  
namespace JwtInDotnetCore.Controllers  
{  
 [Route("api/[controller]")]  
 [ApiController]  
 public class LoginController : ControllerBase  
 {  
 private IConfiguration \_config;  
 public LoginController(IConfiguration config)   
 {  
 \_config = config;  
 }  
  
 [HttpPost]  
 public IActionResult Post([FromBody] LoginRequest loginRequest)  
 {  
 //your logic for login process  
 //If login usrename and password are correct then proceed to generate token  
  
 var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_config["Jwt:Key"]));  
 var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);  
  
 var Sectoken = new JwtSecurityToken(\_config["Jwt:Issuer"],  
 \_config["Jwt:Issuer"],  
 null,  
 expires: DateTime.Now.AddMinutes(120),  
 signingCredentials: credentials);  
  
 var token = new JwtSecurityTokenHandler().WriteToken(Sectoken);  
  
 return Ok(token);  
 }  
 }  
}

# Step 5: Protect API Endpoints

Now that you have configured JWT authentication, you can protect your API endpoints by applying the [Authorize] attribute to controllers or actions that require authentication.

[Authorize]  
[Route("api/[controller]")]  
[ApiController]  
public class HelloWorldController : ControllerBase  
{  
 [HttpGet]  
 public IActionResult Get()   
 {  
 return Ok("Hello World");  
 }  
}

Since now we have added [Authorize] in the controller, when we call HelloWorld endpoint we get an 401 response.