Authentication And Authorization In ASP.NET Core Web API With JSON Web Tokens

## **Introduction**

Authentication is the process of validating user credentials and authorization is the process of checking privileges for a user to access specific modules in an application. In this article, we will see how to protect an ASP.NET Core Web API application by implementing JWT authentication. We will also see how to use authorization in ASP.NET Core to provide access to various functionality of the application. We will store user credentials in an SQL server database and we will use Entity framework and Identity framework for database operations.

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC**algorithm) or a public/private key pair using **RSA**or **ECDSA**.

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

* Header
* Payload
* Signature

Therefore, a JWT typically looks like the following.

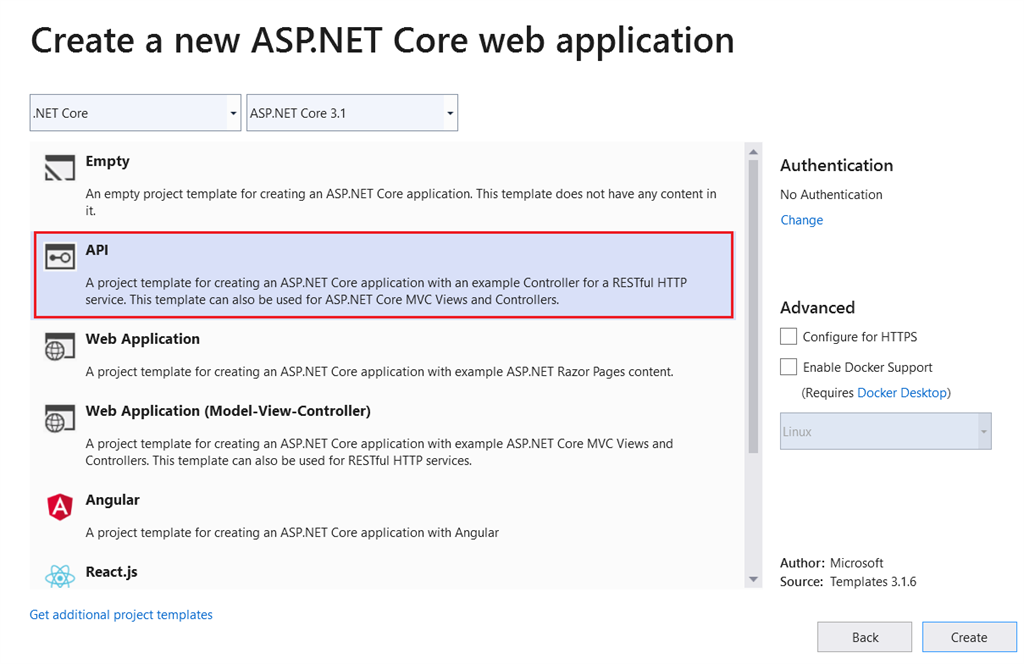
**xxxx.yyyy.zzzz**

Please refer to below link for more details about JSON Web Tokens.

<https://jwt.io/introduction/>

## **Create ASP.NET Core Web API using Visual Studio 2019**

We can create an API application with ASP.NET Core Web API template.



We must install below libraries using NuGet package manager.

* **Microsoft.EntityFrameworkCore.SqlServer**
* **Microsoft.EntityFrameworkCore.Tools**
* **Microsoft.AspNetCore.Identity.EntityFrameworkCore**
* **Microsoft.AspNetCore.Identity**
* **Microsoft.AspNetCore.Authentication.JwtBearer**

We can modify the appsettings.json with below values.

**appsettings.json**

{

  "Logging": {

    "LogLevel": {

      "Default": "Information",

      "Microsoft": "Warning",

      "Microsoft.Hosting.Lifetime": "Information"

    }

  },

  "AllowedHosts": "\*",

  "ConnectionStrings": {

    "ConnStr": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=SarathlalDB;Integrated Security=True;ApplicationIntent=ReadWrite;MultiSubnetFailover=False"

  },

  "JWT": {

    "ValidAudience": "http://localhost:4200",

    "ValidIssuer": "http://localhost:61955",

    "Secret": "ByYM000OLlMQG6VVVp1OH7Xzyr7gHuw1qvUC5dcGt3SNM"

  }

}

We have added a database connection string and also added valid audience, valid issuer and secret key for JWT authentication in above settings file.

Create an “ApplicationUser” class inside a new folder “Authentication” which will inherit the IdentityUser class. IdentityUser class is a part of Microsoft Identity framework. We will create all the authentication related files inside the “Authentication” folder.

**ApplicationUser.cs**

**using** Microsoft.AspNetCore.Identity;

**namespace** JWTAuthentication.Authentication

{

**public** **class** ApplicationUser: IdentityUser

    {

    }

}

We can create the “ApplicationDbContext” class and add below code.

**ApplicationDbContext.cs**

1. **using** Microsoft.AspNetCore.Identity.EntityFrameworkCore;
2. **using** Microsoft.EntityFrameworkCore;
4. **namespace** JWTAuthentication.Authentication
5. {
6. **public** **class** ApplicationDbContext : IdentityDbContext<ApplicationUser>
7. {
8. **public** ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : **base**(options)
9. {
11. }
12. **protected** **override** **void** OnModelCreating(ModelBuilder builder)
13. {
14. **base**.OnModelCreating(builder);
15. }
16. }
17. }

Create a static class “UserRoles” and add below values.

**UserRoles.cs**

1. **namespace** JWTAuthentication.Authentication
2. {
3. **public** **static** **class** UserRoles
4. {
5. **public** **const** **string** Admin = "Admin";
6. **public** **const** **string** User = "User";
7. }
8. }

We have added two constant values “Admin” and “User” as roles. You can add many roles as you wish.

Create class “RegisterModel” for new user registration.

**RegisterModel.cs**

**using** System.ComponentModel.DataAnnotations;

**namespace** JWTAuthentication.Authentication

{

**public** **class** RegisterModel

    {

        [Required(ErrorMessage = "User Name is required")]

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

        [EmailAddress]

        [Required(ErrorMessage = "Email is required")]

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

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

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

    }

}

Create class “LoginModel” for user login.

**LoginModel.cs**

**using** System.ComponentModel.DataAnnotations;

**namespace** JWTAuthentication.Authentication

{

**public** **class** LoginModel

    {

        [Required(ErrorMessage = "User Name is required")]

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

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

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

    }

}

We can create a class “Response” for returning the response value after user registration and user login. It will also return error messages, if the request fails.

**Response.cs**

**namespace** JWTAuthentication.Authentication

{

**public** **class** Response

    {

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

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

    }

}

We can create an API controller “AuthenticateController” inside the “Controllers” folder and add below code.

**AuthenticateController.cs**

**using** JWTAuthentication.Authentication;

**using** Microsoft.AspNetCore.Http;

**using** Microsoft.AspNetCore.Identity;

**using** Microsoft.AspNetCore.Mvc;

**using** Microsoft.Extensions.Configuration;

**using** Microsoft.IdentityModel.Tokens;

**using** System;

**using** System.Collections.Generic;

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

**using** System.Security.Claims;

**using** System.Text;

**using** System.Threading.Tasks;

**namespace** JWTAuthentication.Controllers

{

    [Route("api/[controller]")]

    [ApiController]

**public** **class** AuthenticateController : ControllerBase

    {

**private** **readonly** UserManager<ApplicationUser> userManager;

**private** **readonly** RoleManager<IdentityRole> roleManager;

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

**public** AuthenticateController(UserManager<ApplicationUser> userManager, RoleManager<IdentityRole> roleManager, IConfiguration configuration)

        {

**this**.userManager = userManager;

**this**.roleManager = roleManager;

            \_configuration = configuration;

        }

        [HttpPost]

        [Route("login")]

**public** async Task<IActionResult> Login([FromBody] LoginModel model)

        {

            var user = await userManager.FindByNameAsync(model.Username);

**if** (user != **null** && await userManager.CheckPasswordAsync(user, model.Password))

            {

                var userRoles = await userManager.GetRolesAsync(user);

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

                {

**new** Claim(ClaimTypes.Name, user.UserName),

**new** Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),

                };

**foreach** (var userRole **in** userRoles)

                {

                    authClaims.Add(**new** Claim(ClaimTypes.Role, userRole));

                }

                var authSigningKey = **new** SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_configuration["JWT:Secret"]));

                var token = **new** JwtSecurityToken(

                    issuer: \_configuration["JWT:ValidIssuer"],

                    audience: \_configuration["JWT:ValidAudience"],

                    expires: DateTime.Now.AddHours(3),

                    claims: authClaims,

                    signingCredentials: **new** SigningCredentials(authSigningKey, SecurityAlgorithms.HmacSha256)

                    );

**return** Ok(**new**

                {

                    token = **new** JwtSecurityTokenHandler().WriteToken(token),

                    expiration = token.ValidTo

                });

            }

**return** Unauthorized();

        }

        [HttpPost]

        [Route("register")]

**public** async Task<IActionResult> Register([FromBody] RegisterModel model)

        {

            var userExists = await userManager.FindByNameAsync(model.Username);

**if** (userExists != **null**)

**return** StatusCode(StatusCodes.Status500InternalServerError, **new** Response { Status = "Error", Message = "User already exists!" });

            ApplicationUser user = **new** ApplicationUser()

            {

                Email = model.Email,

                SecurityStamp = Guid.NewGuid().ToString(),

                UserName = model.Username

            };

            var result = await userManager.CreateAsync(user, model.Password);

**if** (!result.Succeeded)

**return** StatusCode(StatusCodes.Status500InternalServerError, **new** Response { Status = "Error", Message = "User creation failed! Please check user details and try again." });

**return** Ok(**new** Response { Status = "Success", Message = "User created successfully!" });

        }

        [HttpPost]

        [Route("register-admin")]

**public** async Task<IActionResult> RegisterAdmin([FromBody] RegisterModel model)

        {

            var userExists = await userManager.FindByNameAsync(model.Username);

**if** (userExists != **null**)

**return** StatusCode(StatusCodes.Status500InternalServerError, **new** Response { Status = "Error", Message = "User already exists!" });

            ApplicationUser user = **new** ApplicationUser()

            {

                Email = model.Email,

                SecurityStamp = Guid.NewGuid().ToString(),

                UserName = model.Username

            };

            var result = await userManager.CreateAsync(user, model.Password);

**if** (!result.Succeeded)

**return** StatusCode(StatusCodes.Status500InternalServerError, **new** Response { Status = "Error", Message = "User creation failed! Please check user details and try again." });

**if** (!await roleManager.RoleExistsAsync(UserRoles.Admin))

                await roleManager.CreateAsync(**new** IdentityRole(UserRoles.Admin));

**if** (!await roleManager.RoleExistsAsync(UserRoles.User))

                await roleManager.CreateAsync(**new** IdentityRole(UserRoles.User));

**if** (await roleManager.RoleExistsAsync(UserRoles.Admin))

            {

                await userManager.AddToRoleAsync(user, UserRoles.Admin);

            }

**return** Ok(**new** Response { Status = "Success", Message = "User created successfully!" });

        }

    }

}

We have added three methods “login”, “register”, and “register-admin” inside the controller class. Register and register-admin are almost same but the register-admin method will be used to create a user with admin role. In login method, we have returned a JWT token after successful login.

We can make below changes in “ConfigureServices” and “Configure” methods in “Startup” class as well.

**Startup.cs**

**using** JWTAuthentication.Authentication;

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

**using** Microsoft.AspNetCore.Builder;

**using** Microsoft.AspNetCore.Hosting;

**using** Microsoft.AspNetCore.Identity;

**using** Microsoft.EntityFrameworkCore;

**using** Microsoft.Extensions.Configuration;

**using** Microsoft.Extensions.DependencyInjection;

**using** Microsoft.Extensions.Hosting;

**using** Microsoft.IdentityModel.Tokens;

**using** System.Text;

**namespace** JWTAuthentication

{

**public** **class** Startup

    {

**public** Startup(IConfiguration configuration)

        {

            Configuration = configuration;

        }

**public** IConfiguration Configuration { **get**; }

        // This method gets called by the runtime. Use this method to add services to the container.

**public** **void** ConfigureServices(IServiceCollection services)

        {

            services.AddControllers();

            // For Entity Framework

            services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(Configuration.GetConnectionString("ConnStr")));

            // For Identity

            services.AddIdentity<ApplicationUser, IdentityRole>()

                .AddEntityFrameworkStores<ApplicationDbContext>()

                .AddDefaultTokenProviders();

            // Adding Authentication

            services.AddAuthentication(options =>

            {

                options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

                options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

                options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;

            })

            // Adding Jwt Bearer

            .AddJwtBearer(options =>

            {

                options.SaveToken = **true**;

                options.RequireHttpsMetadata = **false**;

                options.TokenValidationParameters = **new** TokenValidationParameters()

                {

                    ValidateIssuer = **true**,

                    ValidateAudience = **true**,

                    ValidAudience = Configuration["JWT:ValidAudience"],

                    ValidIssuer = Configuration["JWT:ValidIssuer"],

                    IssuerSigningKey = **new** SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configuration["JWT:Secret"]))

                };

            });

        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

**public** **void** Configure(IApplicationBuilder app, IWebHostEnvironment env)

        {

**if** (env.IsDevelopment())

            {

                app.UseDeveloperExceptionPage();

            }

            app.UseRouting();

            app.UseAuthentication();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>

            {

                endpoints.MapControllers();

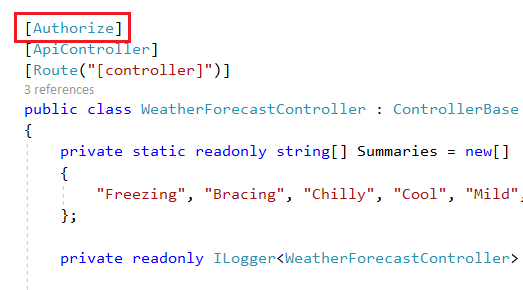
            });

        }

    }

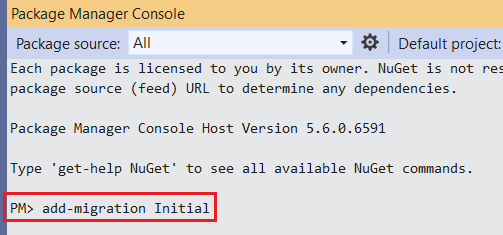
}

We can add “Authorize” attribute inside the “WeatherForecast” controller.



We must create a database and required tables before running the application. As we are using entity framework, we can use below database migration command with package manger console to create a migration script.

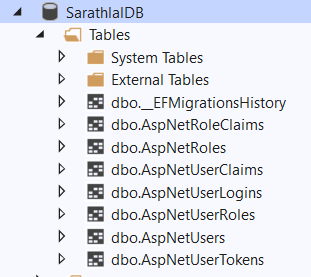
**“add-migration Initial”**



Use below command to create database and tables.

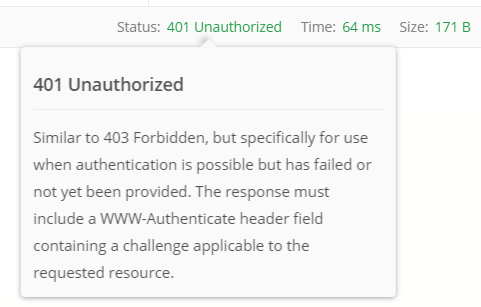
**“update-database”**

If you check the database using SQL server object explorer, you can see that below tables are created inside the database.



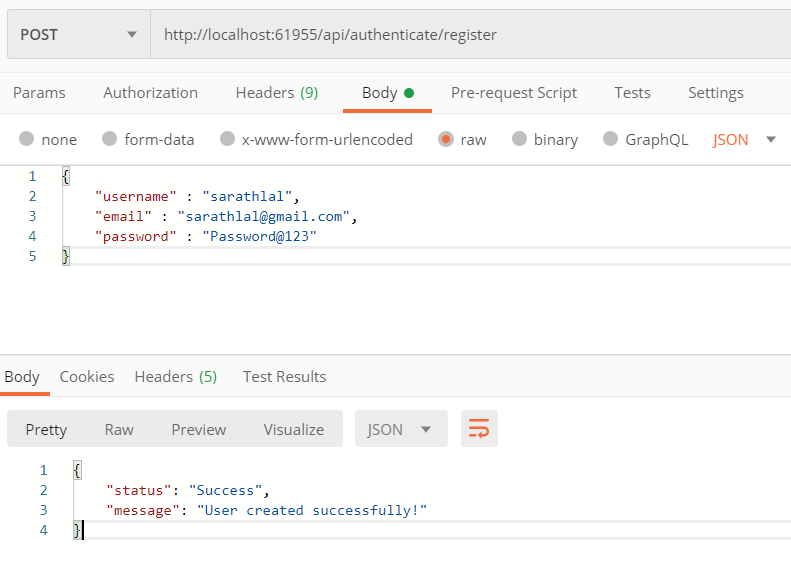
Above seven tables are used by identity framework to manage authentication and authorization.

We can run the application and try to access get method in weatherforecast controller from Postman tool.

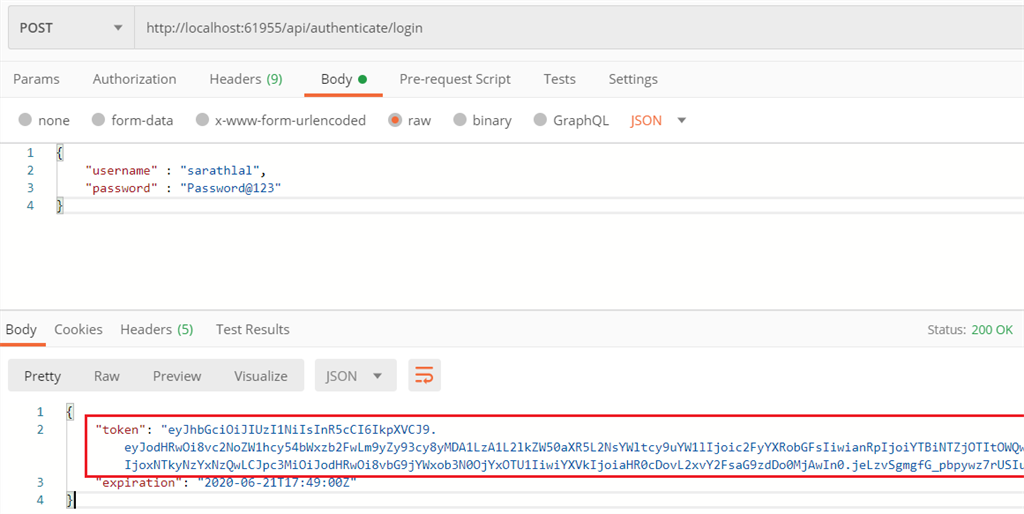


We have received a 401 unauthorized error. Because, we have added Authorize attribute to entire controller. We must provide a valid token via request header to access this controller and methods inside the controller.

We can create a new user using register method in authenticate controller.

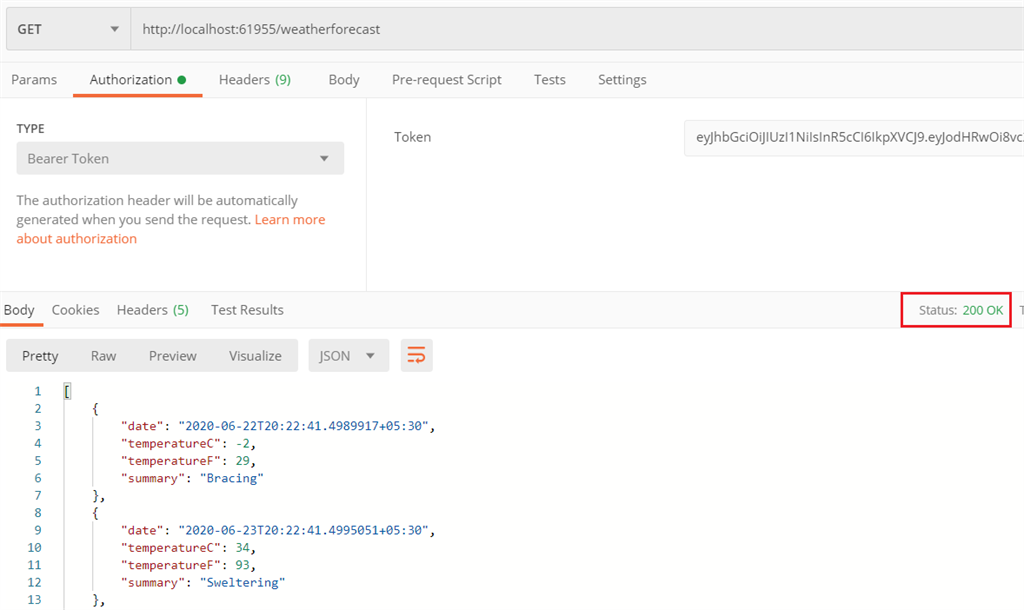


We can use above user credentials to login and get a valid JWT token.



We have received a token after successful login with above credentials.

We can pass above token value as a bearer token inside the authorization tab and call get method of weatherforecast controller again.



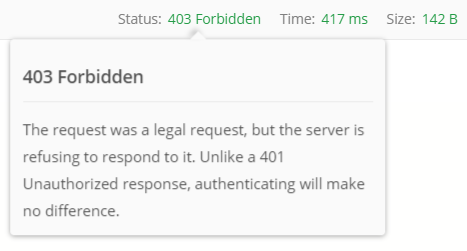
This time, we have successfully received the values from controller.

We can modify the weatherforecast controller with role-based authorization.



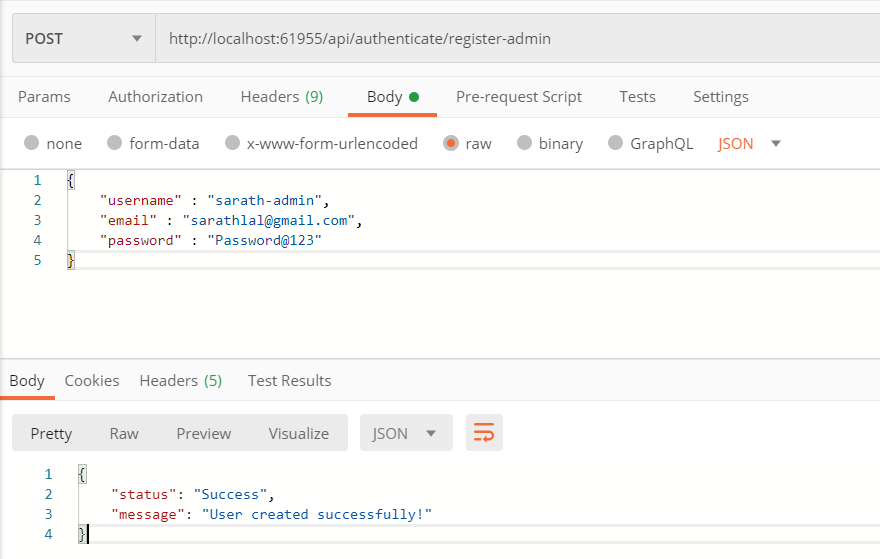
Now, only users with admin role can access this controller and methods.

We can try to access the weatherforecast controller with same token again in Postman tool.

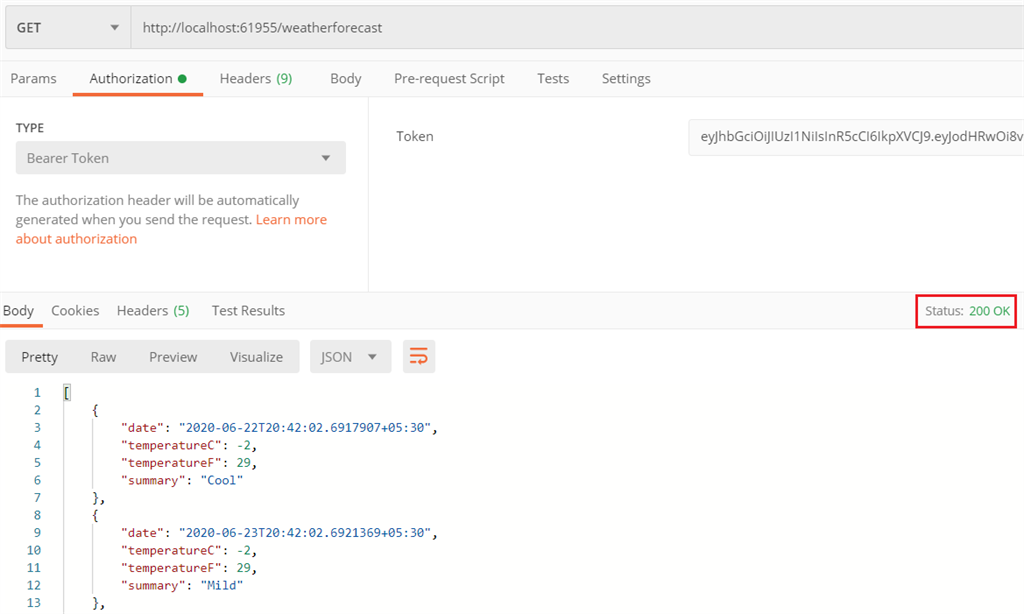


We have received a 403 forbidden error now. Even though, we are passing a valid token we don’t have sufficient privilege to access the controller. To access this controller, user must have an admin role permission. Current user is a normal user and do not have any admin role permission.

We can create a new user with admin role. We already have a method “register-admin” in authenticate controller for the same purpose.

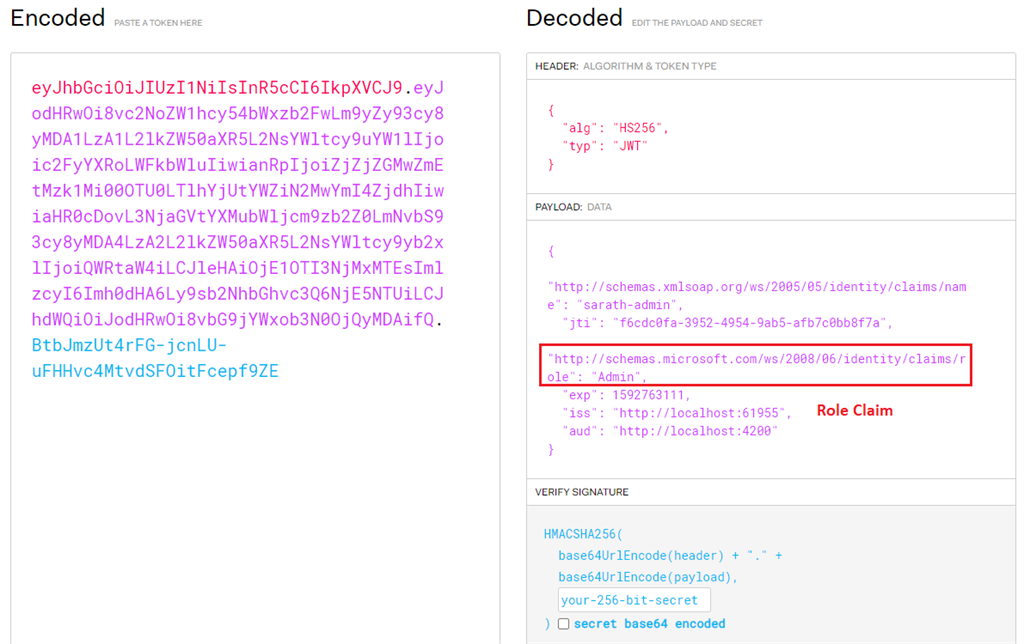


We can login with this new user credentials and get a new token and use this token instead of old token to access the weatherforecast controller.



We have again received the values from weatherforecast controller successfully.

We can see the token payload and other details using **jwt.io** site.



Inside the payload section, you can see the user name, role and other details as claims.