Authentication handler in ASP.Net Core (JWT and Custom)

Authentication is the process that helps identify who is the users. On the other hand, authorization is the process of determining what a user can do. For authorization to work, the user will be authenticated first. We need the user’s identity to identify the role of a user and act accordingly.

Authentication middleware is responsible for authentication in ASP.Net Core applications. The authentication middleware uses the registered authentication handlers to authenticate a user. The registered handlers and their associated configurations are called schemes.

In ASP.Net Core, the authentication middleware is added in the Startup class, inside the Configure method. It is done by calling UseAuthentication method on the IApplicationBuilder instance passed to the method.

Authentication schemes are registered in the Startup class inside of the ConfigureServices method. It is done by calling AddAuthentication method on the IServiceCollection instance passed to the method. We can register multiple authentication schemes, whereas only one of them will be a default scheme.

Authentication Scheme and Handler

As I mentioned above, the scheme is nothing but the index of a handler and its configuration. A scheme is a mechanism for referring an authentication, the challenge (how to challenge a request if authentication fails) and forbid behavior (how to react when authorization of a user fails).

The configuration method for AddAuthentication provides a way to configure the default scheme. We will show it in detail later.

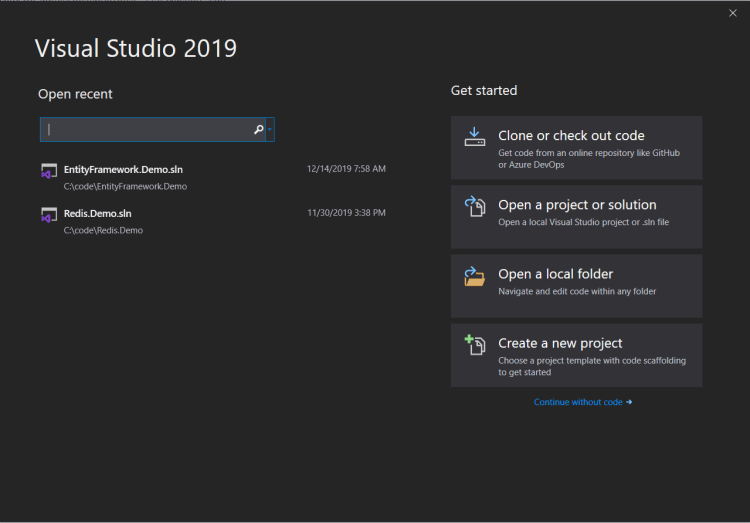
An authentication handler is a class, where we will define how to react to a specific scheme. To implement a handler, we will either have to implement the interface IAuthenticationHandler or derive from class AuthenticationHandler<TOptions>.

Inside the handler, we can use our own logic for authenticating a user.

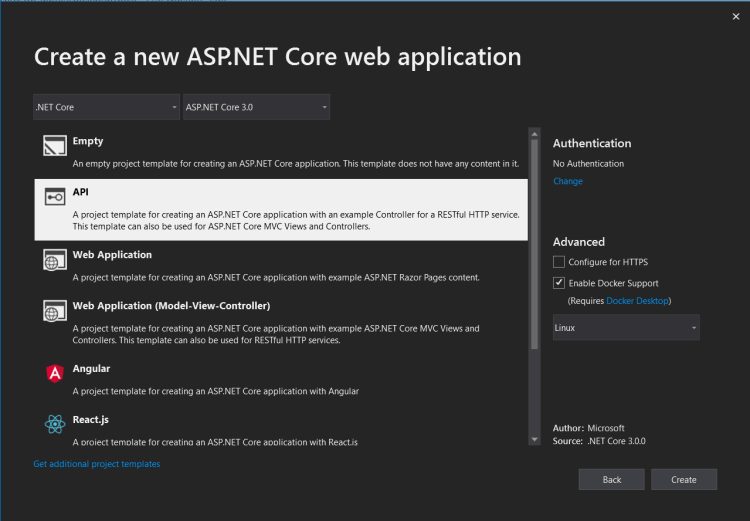
Creating a Web API Application

To demonstrate the feature, I will create a ASP.Net Core Web API application. The application will have a simple name API, which will return names of few states. But to access the API the caller will first authenticate using a /name/authenticate API endpoint.

To create a new ASP.Net Core Web API application, I will use Visual Studio 2019. After opening Visual Studio 2019, I will click on **“Create a new project”** option.



From the list of project templates, I will select **“ASP.NET Core Web Application”**. And create a new project named **“Auth.Demo”**. After that, I will select “API” as the type of project and will keep Docker enable. And, I will use ASP.NET Core 3.0.

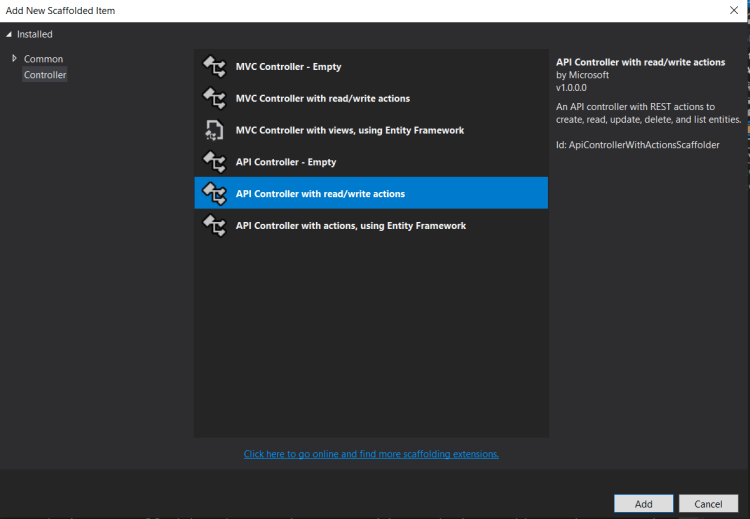


Once the project is created, I will create a new API to authenticate. But firstly, I will delete the auto-generated classes WeatherForecast and WeatherForecastController.

Creating Name API

Firstly, I will create a new API, by right-clicking the **“Controllers”** folder, then selecting **“Add -> Controller”** menu option.

Secondly, when the Add New item popup appears, I will select the “API Controller with read/write actions” option.



Finally, I will name the controller as **“NameController”**. From the auto-generated controller class, I will delete the code for Post, Put and Delete methods, as they are not relevant for this example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **using** System.Collections.Generic;  **using** Microsoft.AspNetCore.Mvc;    **namespace** Auth.Demo.Controllers  {      [Route("api/[controller]")]      [ApiController]  **public** **class** NameController : ControllerBase      {          // GET: api/Name          [HttpGet]  **public** IEnumerable<**string**> Get()          {  **return** **new** **string**[] { "value1", "value2" };          }            // GET: api/Name/5          [HttpGet("{id}", Name = "Get")]  **public** **string** Get(**int** id)          {  **return** "value";          }      }  } |

**Create the Authenticate method**

Inside of the NameController class, I will create a new HTTP POST method named Authenticate. This method will accept the UserCred object parameter from the body of the POST request.

The UserCred class contain two properties Username and Password.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [HttpPost("authenticate")]  **public** IActionResult Authenticate([FromBody] UserCred userCred)  {  **return** Ok();  }    **public** **class** UserCred  {  **public** **string** Username { **get**; **set**; }  **public** **string** Password { **get**; **set**; }  } |

But before we proceed further to implement the Authenticate method, let us first configure Startup class to wire up the authentication middleware.

**Wiring up Startup for authentication middleware**

Firstly, inside the Startup classes Configure method, I will call UseAuthentication extension method on the IApplicationBuilder instance. I will call this method just above the call of UseAuthorization.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **public** **void** Configure(IApplicationBuilder app, IWebHostEnvironment env)  {  **if** (env.IsDevelopment())      {          app.UseDeveloperExceptionPage();      }        app.UseRouting();        app.UseAuthentication();      app.UseAuthorization();        app.UseEndpoints(endpoints =>      {          endpoints.MapControllers();      });  } |

At this point, I have asked the .Net Core to use authentication, but have not provided any authentication schemes yet. But before we do that, we have to consider how everything will play out.

Once a user is authenticated, we will let the authorizer determine if the Identity created is allowed to access a particular resource or not. For this example, we will not have any authorization, we will let all authenticated users have access to all resources.

Post authentication, we will send a token back to the caller, using which the caller will make subsequent resource calls. This token generation and lifetime management process can be custom. Or we can use something like JWT.

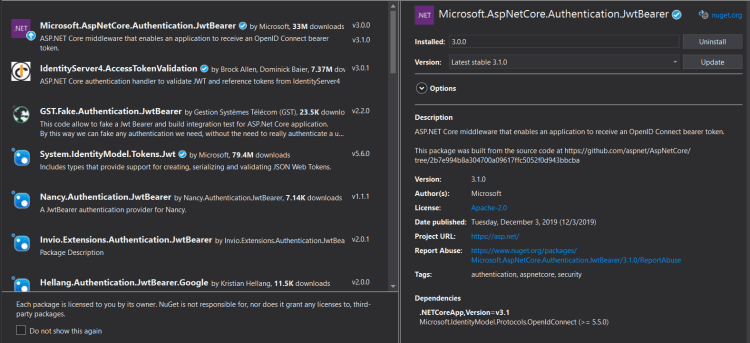
**What is JWT**

JWT stands for [JSON Web Token](https://en.wikipedia.org/wiki/JSON_Web_Token). JWT is JSON based access token created for claims. It is a self-contained and compact standard for an access token to securely transfer claims.

For our project, we will use JWT. For creating a JWT, we can use different hash algorithms. We will use HS256 algorithm for this project.

**Adding Authentication**

Now, I will update the Startup class to call the extension method AddAuthentication of IServiceCollection instance inside of the ConfigureServices method. I will add NuGet package **“Microsoft.AspNetCore.Authentication”** to enable JWT.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **var** tokenKey = Configuration.GetValue<**string**>("TokenKey");  **var** key = Encoding.ASCII.GetBytes(tokenKey);    services.AddAuthentication(x =>  {      x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;      x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;  })  .AddJwtBearer(x =>  {      x.RequireHttpsMetadata = **false**;      x.SaveToken = **true**;      x.TokenValidationParameters = **new** TokenValidationParameters      {          ValidateIssuerSigningKey = **true**,          IssuerSigningKey = **new** SymmetricSecurityKey(key),          ValidateIssuer = **false**,          ValidateAudience = **false**      };  }); |

Configuration file below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {    "TokenKey": "This is my test private key",    "Logging": {      "LogLevel": {        "Default": "Information",        "Microsoft": "Warning",        "Microsoft.Hosting.Lifetime": "Information"      }    },    "AllowedHosts": "\*"  } |

In the above code, first, inside of the AddAuthentication method, I am setting the default authentication and challenge scheme as JwtBearerDefaults.AuthenticationScheme.

Secondly, I am calling the AddJwtBearer extension method. And inside of the code I am setting the IssuerSigningKey using the string key **“This is my test private key”** from the configuration file. The AddJwtBearer will handle all requests and will check for a valid JWT Token in the header. If it is not passed, or the token is expired, it will generate a 401 Unauthorized HTTP response.

**Authentication Manager**

Next, I am going to implement the authentication manager which will handle authentication of users.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | **using** Microsoft.IdentityModel.Tokens;  **using** System;  **using** System.Collections.Generic;  **using** System.IdentityModel.Tokens.Jwt;  **using** System.Linq;  **using** System.Security.Claims;  **using** System.Text;    **namespace** Auth.Demo  {  **public** **interface** IJWTAuthenticationManager      {  **string** Authenticate(**string** username, **string** password);      }    **public** **class** JWTAuthenticationManager : IJWTAuthenticationManager      {          IDictionary<**string**, **string**> users = **new** Dictionary<**string**, **string**>          {              { "test1", "password1" },              { "test2", "password2" }          };    **private** **readonly** **string** tokenKey;    **public** JWTAuthenticationManager(**string** tokenKey)          {  **this**.tokenKey = tokenKey;          }    **public** **string** Authenticate(**string** username, **string** password)          {  **if** (!users.Any(u => u.Key == username && u.Value == password))              {  **return** **null**;              }    **var** tokenHandler = **new** JwtSecurityTokenHandler();  **var** key = Encoding.ASCII.GetBytes(tokenKey);  **var** tokenDescriptor = **new** SecurityTokenDescriptor              {                  Subject = **new** ClaimsIdentity(**new** Claim[]                  {  **new** Claim(ClaimTypes.Name, username)                  }),                  Expires = DateTime.UtcNow.AddHours(1),                  SigningCredentials = **new** SigningCredentials(  **new** SymmetricSecurityKey(key),                      SecurityAlgorithms.HmacSha256Signature)              };  **var** token = tokenHandler.CreateToken(tokenDescriptor);  **return** tokenHandler.WriteToken(token);          }      }  } |

In JWTAuthenticationManager class, I am keeping a constant dictionary of username and password for demo purposes. In a real-life scenario, this information will be saved encrypted in data storage.

Inside the Authenticate method, I am checking against the dictionary if the username and password are available. If they are, then I will create a JWT token using the JWT API which will expire in an hour. And I am using HS256 algorithm for encryption of the token. Also, I am using the same key from the configuration, as I used in the Startup class for configuring JWT handler.

**Updating NameController**

Finally, I will update NameController class’s Authenticate method to call JWTAuthenticationManager and authenticate the incoming username and password. And send back the generated JWT token.

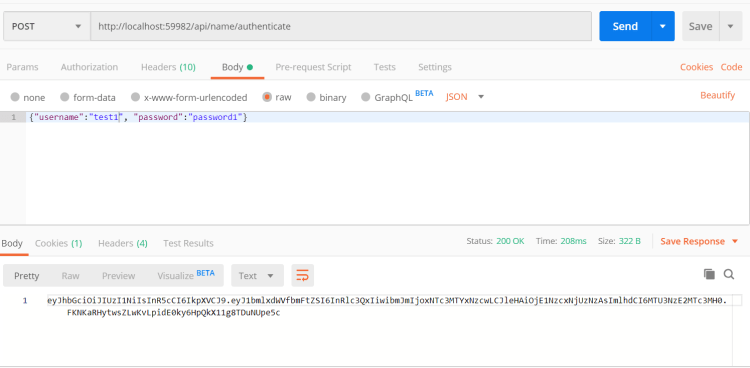
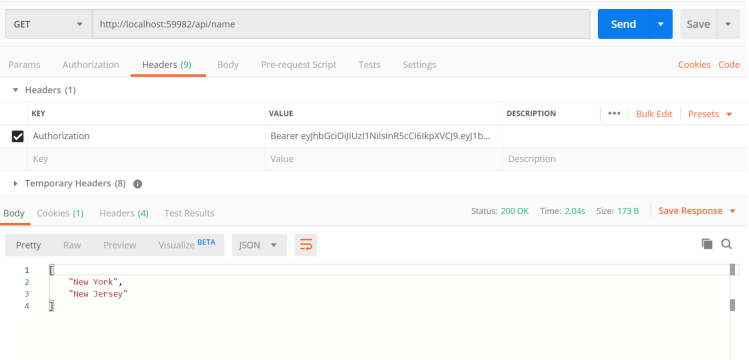
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | **using** System.Collections.Generic;  **using** Microsoft.AspNetCore.Authorization;  **using** Microsoft.AspNetCore.Mvc;    **namespace** Auth.Demo.Controllers  {      [Authorize]      [Route("api/[controller]")]      [ApiController]  **public** **class** NameController : ControllerBase      {  **private** **readonly** IJWTAuthenticationManager jWTAuthenticationManager;    **public** NameController(IJWTAuthenticationManager jWTAuthenticationManager)          {  **this**.jWTAuthenticationManager = jWTAuthenticationManager;          }            // GET: api/Name          [HttpGet]  **public** IEnumerable<**string**> Get()          {  **return** **new** **string**[] { "New York", "New Jersey" };          }            // GET: api/Name/5          [HttpGet("{id}", Name = "Get")]  **public** **string** Get(**int** id)          {  **return** "New Jersey";          }            [AllowAnonymous]          [HttpPost("authenticate")]  **public** IActionResult Authenticate([FromBody] UserCred userCred)          {  **var** token = jWTAuthenticationManager.Authenticate(userCred.Username, userCred.Password);    **if** (token == **null**)  **return** Unauthorized();    **return** Ok(token);          }      }  } |

In the above code if the token returned is null, I am sending an Unauthorized response back. Else sending a Ok response with the JWT token generated.

Next, I will update the Startup class to register JWTAuthenticationManager.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | **using** System.Text;  **using** Microsoft.AspNetCore.Authentication.JwtBearer;  **using** Microsoft.AspNetCore.Builder;  **using** Microsoft.AspNetCore.Hosting;  **using** Microsoft.Extensions.Configuration;  **using** Microsoft.Extensions.DependencyInjection;  **using** Microsoft.Extensions.Hosting;  **using** Microsoft.IdentityModel.Tokens;    **namespace** Auth.Demo  {  **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();  **var** tokenKey = Configuration.GetValue<**string**>("TokenKey");  **var** key = Encoding.ASCII.GetBytes(tokenKey);                services.AddAuthentication(x =>              {                  x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;                  x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;              })              .AddJwtBearer(x =>              {                  x.RequireHttpsMetadata = **false**;                  x.SaveToken = **true**;                  x.TokenValidationParameters = **new** TokenValidationParameters                  {                      ValidateIssuerSigningKey = **true**,                      IssuerSigningKey = **new** SymmetricSecurityKey(key),                      ValidateIssuer = **false**,                      ValidateAudience = **false**                  };              });                services.AddSingleton<IJWTAuthenticationManager>(**new** JWTAuthenticationManager(tokenKey));          }            // 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();              });          }      }  } |

Finally, I will make an HTTP POST to create a JWT token and use this token to call api/name.

HTTP POST to get tokenHTTP GET to get names