Optional 16c : JWT Authentication in Web API

Objective

We have secured access to our MVC controllers. In this exercise you will learn how to add authentication to Web API controllers.

This exercise will take around 30 minutes.

Referenced material

This exercise is based on material from the chapter "Security”.

Adding JWT Authentication and Authorisation Web API

- Open the 'Begin' solution in Visual Studio and compile (Shift Ctrl+B). The ‘Begin’ solution is identical to the ‘End’ solution from the previous lab.

- Open /ApiControllers/ForumController. Add an [Authorise] attribute to the class:

  ```
      [Route("api/[controller]", Order = 10)]
      [ApiController]
      [Authorise]
      public class ForumController : ControllerBase
      {
  ```

- Run the program, and, without signing in, change the URL to end with /client_demo.html

  Click on the Load Forums button. You should see an Error 401 – Unauthorised message.

  Go back to the web site’s home page, and log on. Then repeat the test on the client demo page. This time it should work.

- You could simply leave things as they are. We now have security on our Web API controller. The authorisation is currently done using a cookie, in a format that is specific to ASP.Net Core.

  However, it’s very common to use an open, industry-standard method called JWT – JSON Web Tokens.

  In the rest of this lab, we are going to see how to implement JWT authorisation in our application.

- JWT tokens encrypt the user’s claims, as well as some data about their issuer, using a private key.

  We are going to store this key and its associated details in appsettings.json, for simplicity. So, add the following to appsettings.json now:

  ```
    "Logging": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft": "Warning",
  ```

```
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "JwtKey": "QAForumKey:Shh!ItsAnApplicationSecret!",
  "JwtIssuer": "http://qaforum.com",
  "JwtExpireDays": 30,
  "AllowedHosts": "*"
}
```

Important note: appsettings.json is not secure, and is not a suitable place to store your key in real life! You must find a secure location in which to store the key. See this website for more details:

https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-3.1&tabs=windows

- Next, we need to install a Nuget packaget. Use the Nuget Package Manager to find out what version of the Microsoft.AspNetCore projects are already installed. Then, be sure to install the same version number of the following package:

    - Microsoft.AspNetCore.ApiAuthorisation.IdentityServer

- Add the following to ConfigureServices in Startup.cs:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication()
        .AddJwtBearer(options => {
            options.RequireHttpsMetadata = false;
            options.SaveToken = true;
            options.TokenValidationParameters = new TokenValidationParameters
            {
                ValidIssuer = Configuration["JwtIssuer"],
                ValidAudience = Configuration["JwtIssuer"],
                IssuerSigningKey = new SymmetricSecurityKey
                           (Encoding.UTF8.GetBytes(Configuration["JwtKey"]))
            };
        });
```

When the authorisation system attempts to authorise Web API clients, it will use the details configured here to attempt to decrypt the JWT token. If the decryption is successful, that proves that the certificate must have been encrypted using the secret key stored in appsettings.json.

Note that we do *not* change the default authentication scheme to use JWT. If we did that, it would break all the MVC controllers, which don't used JWT. If your application consists *only* of Web API controllers, you may want to change the default. You can do this as follows:

```
services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
    .AddJwtBearer(options => {....
```

- Add a controller to the ApiControllers folder. The controller is an API Controller – Empty, called UserController. This controller will contain endpoints that the clients can use when their users log on.

The constructor for the class will set up some fields that will be injected, as shown below. When resolving namespaces, take extra care with the IConfiguration data type, because there are multiple data types with this same name in different namespace. The namespace you need is Microsoft.Extensions.Configuration, and this might not be the first one shown.

```
    [Route("api/[controller]")]
    [ApiController]
    public class UserController : ControllerBase
    {
        private readonly UserManager<ApplicationUser> userManager;
        private readonly SignInManager<ApplicationUser> signinManager;
        private readonly IConfiguration config;

        public UserController(UserManager<ApplicationUser> userManager,
                        SignInManager<ApplicationUser> signinManager,
                        IConfiguration config)
        {
            this.userManager = userManager;
            this.signinManager = signinManager;
            this.config = config;
        }

    }
```

- Now, add the following action and helper method. Once again, there is a data type which exists in two namespaces. This time, the type is JwtRegisteredClaimNames, and the namespace you need is System.IdentityModel.Tokens.Jwt

```
[HttpPost("signin")]
public async Task<IActionResult> SignInAsync
                       ([FromForm]string username, [FromForm]string password)
{
    var user = await userManager.FindByNameAsync(username);
    var result = await signinManager.PasswordSignInAsync
               (user, password, false, false);

    if (result == Microsoft.AspNetCore.Identity.SignInResult.Success)
    {
        // Signing in successfully places a cookie in the response.
        // Remove the cookie from the response, since JWT will replace
        // the cookie
        Response.Cookies.Delete(".AspNetCore.Identity.Application");
        var token = await GenerateJwtToken(user);
        return Ok(token);
    }

    return Unauthorised();
}

private async Task<string> GenerateJwtToken(ApplicationUser user)
{
    var claims = new List<Claim>
    {
        // Add some standard claims to the new token
        new Claim(JwtRegisteredClaimNames.Sub, user.Email),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
        new Claim(ClaimTypes.NameIdentifier, user.Id),
        new Claim(ClaimTypes.Name, user.UserName)
    };
    // Copy all the user's claims from the Identity system's claims
    // list into the JWT token
    foreach (var claim in await userManager.GetClaimsAsync(user))
    {
        claims.Add(claim);
    }

    var key = new SymmetricSecurityKey
                (Encoding.UTF8.GetBytes(config["JwtKey"]));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
    var expires = DateTime.Now.AddDays(int.Parse(config["JwtExpireDays"]));

    var token = new JwtSecurityToken(
        config["JwtIssuer"],
        config["JwtIssuer"],
        claims,
        expires: expires,
        signingCredentials: creds
    );

    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Take a moment to look through this code and see how it works. It uses the Identity system to sign in the user (i.e. check their credentials). It then generates a JWT token, with all the same claims as the Identity user, and returns that token to the client. When it creates this token, it uses the signing credentials from the configuration system to encrypt it – the same credentials that the authorisation system was configured to use to decrypt the tokens a few steps earlier.

- Finally, you will remember that we didn't change the default authorisation scheme. So, in /ApiControllers/ForumController we will need to specify the non-default scheme we want to use:

```
    [Route("api/[controller]", Order = 10)]
    [ApiController]
    [Authorise(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
    public class ForumController : ControllerBase
    {
```

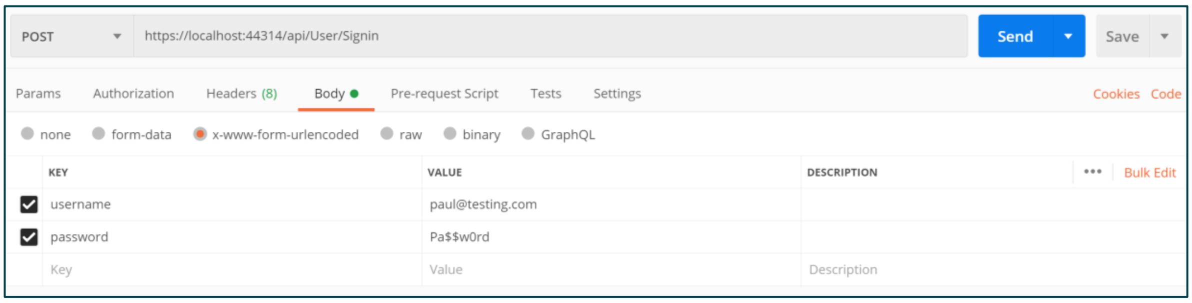
- Let's test what we've done using Postman.

  Run the application, make a note of the port number on which it's running, then start Postman.

  In Postman, create a new request:

  - Set the request type to Post
  - In the URL, enter https://localhost:<port>/api/User/SignIn
  - Click on the Body of the request, then choose x-www-form-urlencoded. In that section, you can enter key-value pairs, which will be encoded by Postman in the standard URL format and then added to the body of the request. Enter the following two keys, with values that represent a user which you know is valid in the application:

  - username
  - password

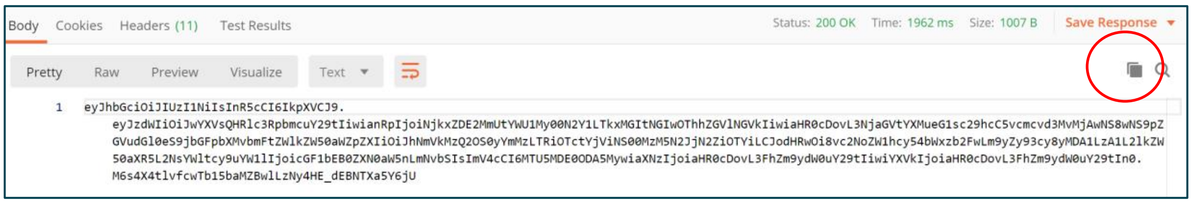


POST https://localhost:44314/api/User/Signin

Body: x-www-form-urlencoded

KEY	VALUE	DESCRIPTION
username	paul@testing.com	
password	Pa$$w0rd	

- Send the request. The response should contain a JWT token, similar to the following:



Status: 200 OK Time: 1962 ms Size: 1007 B

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJzdWIiOiJwYXVsQHRlc3RpbmcuY29tIiwianRpIjoiNjkxZDE2MmUtYWU1My00N2Y1LTkxMGItNGIwOThhZGVlNGVkIiwiaHR0cDovL3NjaGVtYXMueG1sc29hcC5vcmcvd3MvMjAwNS8wNS9pZ
    GVudGl0eS9jbGFpbXMvbmFtZWlkZW50aWZpZXIiOiJhNmVkMzQ2OS0yYmMzLTRiOTctYjViNS00MzM5N2JjN2ZiOTYiLCJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW
    50aXR5L2NsYWltcy9uYW1lIjoicGF1bEB0ZXN0aW5nLmNvbSIsImV4cCI6MTU5MDE0ODA5MywiaXNzIjoiaHR0cDovL3FhZm9ydW0uY29tIiwiYXVkIjoiaHR0cDovL3FhZm9ydW0uY29tIn0.
    M6s4X4tlvfcwTb15baMZBwlLzNy4HE_dEBNTXa5Y6jU
```

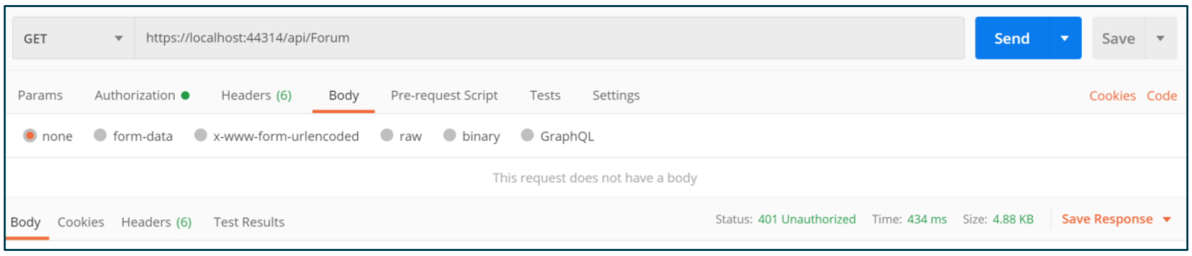
Click the Copy button (circled in red above)

- Create a new request. Configure it as follows:

  Set the request type to Get

  In the URL, enter https://localhost:<port>/api/Forum

  Send the request. You should see a response code of 401 – Unauthorised. Of course, that's what we expected, because we didn't send the JWT token with the request!



GET https://localhost:44314/api/Forum

Body: none

This request does not have a body

Status: 401 Unauthorized Time: 434 ms Size: 4.88 KB

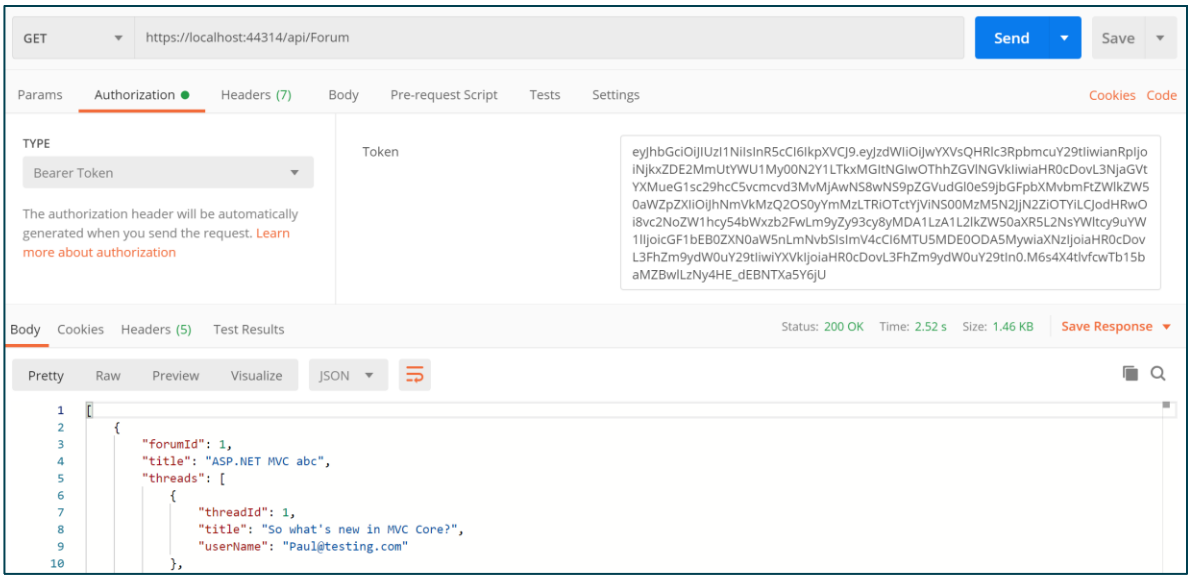
- In the Request section, click on Authorisation. Then:

  Change the "Type" dropdown to "Bearer Token".

  To the right of the dropdown is a text box for you to enter the token. Paste your token in here

  Resubmit the request

  This time, the request should succeed with code 200 – Ok, and you should see the forum details!



GET https://localhost:44314/api/Forum

Authorization: TYPE Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

Token

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJwYXVsQHRlc3RpbmcuY29tIiwianRpIjo
iNjkxZDE2MmUtYWU1My00N2Y1LTkxMGItNGIwOThhZGVlNGVkIiwiaHR0cDovL3NjaGVt
YXMueG1sc29hcC5vcmcvd3MvMjAwNS8wNS9pZGVudGl0eS9jbGFpbXMvbmFtZWlkZW5
0aWZpZXIiOiJhNmVkMzQ2OS0yYmMzLTRiOTctYjViNS00MzM5N2JjN2ZiOTYiLCJodHRwO
i8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW
1lIjoicGF1bEB0ZXN0aW5nLmNvbSIsImV4cCI6MTU5MDE0ODA5MywiaXNzIjoiaHR0cDov
L3FhZm9ydW0uY29tIiwiYXVkIjoiaHR0cDovL3FhZm9ydW0uY29tIn0.M6s4X4tlvfcwTb15b
aMZBwlLzNy4HE_dEBNTXa5Y6jU
```

Status: 200 OK Time: 2.52 s Size: 1.46 KB

```
[
    {
        "forumId": 1,
        "title": "ASP.NET MVC abc",
        "threads": [
            {
                "threadId": 1,
                "title": "So what's new in MVC Core?",
                "userName": "Paul@testing.com"
            },
```

- Finally, go back to your web browser, and check that all the other functionality works as expected, and that our inclusion of JWT authentication for Web API has not affected our MVC controllers.

Using JWT Authentication in JavaScript Clients

Now that we know our JWT authentication works correctly, let's have a look at how to use it from within a JavaScript application.

If you're not familiar with JavaScript and JQuery, don't worry, we're giving you all the code you need. But you should take a moment to study the code and see if you can work out what it does.

- In Visual Studio, open wwwroot/client_demo.html

- In the file are 3 paragraphs, wrapped in <p> </p> tags.

  Immediately after those three paragraphs, insert the following HTML, which shows fields and buttons for the user to log on and off:

```
<div class="container pb-4">
    <div id="logon" class="row">
        <div class="form-group d-flex flex-column justify-content-
end">
            <label for="username" class="control-
label">Username:</label>
            <input type="text" name="username" id="username"
                         class="form-control" />
        </div>
        <div class="form-group d-flex flex-column justify-content-
end">
            <label for="password" class="control-
label">Password:</label>
            <input type="password" name="password" id="password"
```

```
                            class="form-control" />
        </div>

        <div class="form-group d-flex flex-column justify-content-
end">
            <button id="logon-btn" class="btn btn-secondary btn-sm">
                Log On
            </button>
        </div>
    </div>
    <div id="logon-message" class="row">
        Logon message
    </div>
    <div id="logoff" class="row">
       <button id="logoff-btn" class="btn btn-secondary btn-sm">Log
Off</button>
    </div>
</div>
```

- Lower down on the page, the JavaScript starts. (Note that JavaScript should normally be in a different file to HTML. We've put them together here only to make it easier for you to find all the related code in this simple example.)

  Add the following to the start of the JavaScript, to hide the logon message and logoff button when the page loads initially. The code also declares a variable to hold the JWT token:

```
    <script>
        $(function () {
            var jwt = "";

            $('#logon-message').hide();
            $('#logoff').hide();
            var error = function (jqXHR) {

                try {
```

- Add some code to log the user on. This code should come after the error handler, and before the “Get” button handler:

```
$('#logon-btn').click(function () {
    var username = $('#username').val();
    $.post('/api/User/Signin',
        {
            "username": username,
            "password": $('#password').val()
        })
        .done(function (data) {
            jwt = data;
            $('#logon-message').text('Logged on as ' +
username).show();
            $('#logoff').show();
            $('#logon').hide();
        })
        .fail(function (jqXHR) {
            if (jqXHR.status === 401) {
                $('#logon-message').text('Logon
unsuccessful').show();
            } else {
                $('#logon-message')
                         .text('Something went wrong while logging
on').show();
            }
        });
});
```

Take a moment to examine this code. See that it sends a Post request to the Signin API. If the request succeeds, it stores the response text (the JWT token) in a variable for later use.

- Immediately below the logon code, add the following code to handle the user clicking the logoff button:

```
            $('#logoff-btn').click(function () {
                jwt = "";
                $('#logon-message').hide();
                $('#logoff').hide();
                $('#logon').show();

            });
```

Notice that the logoff code does not involve making any requests to the server. It simply clears the JWT token from its variable, and shows the logon form.

- Now we can log on and off, so the next step is to send the JWT token with each of the two requests.

Modify the Get request as follows:

```
$('#get').click(function () {
    // This code runs when the GET button is clicked
    var pleaseWait = $('<div>');
    pleaseWait.append($('<h3>').text('Please wait'));
    pleaseWait.append($('<p>').text('Loading the forums...'));
    $('#output').html(pleaseWait);

    $.ajax({      // This line sends a GET request, and parses the
JSON result
        dataType: "json",
        url: "/api/Forum",
        headers: { Authorisation: "Bearer " + jwt }
    })
    ~~$.getJSON("/api/Forum")~~      // Remove this
        .done(function (data) {
            // If the GET request succeeds
            var output = $('<div>');
            output.append($('<h3>').text('Here are the forums:'));
```

We have removed the call to $.getJSON, because $.getJSON is a helper function which is designed to handle only the most basic type of request, and does not allow us to add headers to the request. Instead, $.ajax is used, and you can see where we add the "Authorisation" header which includes the JWT token.

- The Post request needs a similar change, but it's more straightforward because it already uses $.ajax. So, add one line to include the new header:

```
            $('#post').click(function (e) {
                // This code runs when the POST button is clicked
                e.preventDefault();
                var pleaseWait = $('<div>');
                pleaseWait.append($('<h3>').text('Please wait'));
                pleaseWait.append($('<p>').text('Adding the
forum...'));
                $('#output').html(pleaseWait);

                var forum = { title: $('#title').val() };

                // This command sends the POST request to the server
                $.ajax({
                    method: "POST",
                    url: "/api/Forum",
                    data: JSON.stringify(forum),
                    contentType: "application/json; charset=utf-8",
                    headers: { Authorisation: "Bearer " + jwt }
                }).done(function () {

                    // If the POST request succeeds
```

- Run the program, and navigate to /client_demo.html.

  Check that you can't load or add data when you are not logged in. Check that you can't log in with incorrect details, but that after entering the correct details, viewing and adding data works correctly.

Optional Super Challenge – Invalidating Tokens

One downside of JWT is that once a client has a token, there is no built-in mechanism to invalidate that token except to wait for it to expire. (In an emergency, you can change the key that is used for signing tokens – but that invalidates all tokens that were encrypted with the old key.)

There are several solutions to this. One of these is as follows:

Add a new property to the ApplicationUser class, called Version. Set Version to be 0 for new users.

Every time the user changes their password or gets locked out of their account, the Version must be increased by one. Optionally, you can supply clients with a Logout API endpoint, which also increases the Version by one.

When a JWT token is issued, a claim is added to the token. The Type of the claim could be something like “User-Version”, and the value of the claim must be the Version number of the user who is being authenticated.

Every time JWT is used to authorise a request, prior to the request being authorised, the ApplicationUser object for the user must be loaded (the user name and id are both contained within the JWT token in the form of claims). Then, the User-Version claim must be checked against the version in the ApplicationUser object. If they don’t match, the authorisation will fail.

The previous step can be implemented using custom middleware to do the checking and to reject the authorisation if necessary.

If you are feeling really adventurous, have a go at implementing this. Don’t be afraid to use Google to help with things you’re not sure about, especially creating middleware (which has not been covered in any detail on this course).

Congratulations, you now understand how you can generate JWT tokens, and use them to authenticate and authorise users of your Web API controllers.