**Project Discussion**

**API – Users**

User authentication is done using JWT tokens. If the username and password match a JWT token is saved in the browser’s session. We can then check this token to let them use the various API endpoints.

During registration, the user password is hashed using SHA512 and a salt. It’s better to avoid using the default salt bundled with the cryptography software as it is “known”. A salt is just a string of text we provide that the algorithm uses to create the final hash. We will only transmit the password hash in the LoginDto so if intercepted the password isn’t compromised.

Https will be used to communicate with the API, so the initial transmission of the user’s password at registration is encrypted.

User management is available by using IUserService.cs it contains all the CRUD commands to manipulate the users. The user controller locks down the user management endpoints to only the admin role. [Authorize(Roles = "admin")]

As an added security measure, the entity linq request also checks if the user role is admin. It again grabs the jwt token from the browser using GetUserRole().

var dbUser =

 GetUserRole().Equals("admin")?

 await \_context.Users.FirstOrDefaultAsync(c => c.Id == id) :

 await \_context.Users.Where(u => u.Id == GetUserId()).FirstOrDefaultAsync();

Alternative approaches:

1) Use newer “bcrypt” hash algorythm rather than SHA512.

2) Use IdentityServer, Keycloak, OpenIddict or some other authentication server. I just went with Identity Core in this project as it will only have a single API and be small scale.

3) Use cloud based authentication services. auth0.com, okta.com, Azure Active Directory B2C. They bill per user but are much easier to set up and can offer the first 1000s of users free.

Challenges:

Learning the security stuff was difficult. It was hard to find a complete guide that used JWT tokes and roles that didn’t also implement other features I didn’t want. Working with Guide A for the JWT tokens and Guide B for the roles meant that somethings A did would break somethings B did.

**Admin Section**

The admin section was built using ASP.NET MVC and razor pages. I have kept it in the same solution, on the same server, as the API.

The reason I chose to use MVC and razor pages:

1) Additional Security. Webpages are rendered on the server side rather than in the client browser.

2) To demonstrate that I can use MVC and razor pages.

I didn’t create a page that allows you to make a user an admin. You can only choose between basic and premium roles when editing a user. I’m happy with this as an additional security measure as this API is for my own personal projects and I can just edit the admin role in the database directly.

Alternative Approaches to building the admin section could be to use “Scaffolding”, where visual studio uses your models to auto generate code that produces a similar admin section to what I’ve created.

Another alternative would be to use a “Starter-Kit” which is basically a pre-built API/Admin Section template you use as a base of your project. Some examples are aspnetboilerplate.com and aspnetzero.com.

Challenges:

1) Table Paging. Originally I just had a GetAll() endpoint in my API that would send out all the users in the database. I quickly realised that the table could be 100s of records long or even worse sending a full database of 1000s of users to the admin section would crash the server. So I wrote an API endpoint that served only 10 users at a time and implemented a CSS table with pages at the bottom, each page making a request for the next batch of 10 users.



2) Username Search. I realised searching by ID wouldn’t be good enough and you can’t expect the admin to look through lots of table pages to find the correct record. I added the username search endpoint to my API.

3) Layout of tables on mobile phones. Making the user table fit into a mobile screen was difficult. I found a guide on how to use css flexbox to re-orientate the table headers into a column and then allow the table to scroll sideways. Otherwise the table would expand off the screen and the user would need to zoom in or out to view the whole table.



**Resetting Passwords**

Users need the ability to reset their password if they forget.

The API has an endpoint of your-url.com/Auth/ForgotPass

This allows anyone to submit an email address/username and request a password reset email.

To stop spam emails to users, this endpoint will only send 1 email per hour. The site’s admin email address has been explicitly denied from this process.

Two fields were added to the user model to support this – bool ForgotPass and DateTime DatePassEmailSent. Using basic logic allows the controller to toggle ForgotPass to true when the email is first sent, then toggle it back to false if the time saved in DatePassEmailSent is greater than 1 hour.

IEmailServices.cs allows the controller to send the password reset email. It gets all the SMTP config information from the appsettings.json file via IOptions<EmailCofnig> dependency injection.

The Auth controller will create a JWT token and pass it to EmailServices.SendPasswordResetEmail(User user, string token)

The email will contain a warning saying only to click the link if the user requested the password reset. The html link will be a url with the user’s ID and Token in the query string.

Your-url.com/ResetPass?id=2&token=233rgths567sdsfg

The Admin Website MVC Controller was reused to route the request to a razor page with a password reset form.

When the user gets to the password reset page they are logged in using the JWT token grabbed from the url. As an added security measure, when the Auth controller creates the token it is saved to the user’s database entry in the ResetToken field. This token is compared against the one provided before resetting the password.



**Logging**

Microsoft’s built in logger interface was used with a 3rd party logging tool called Serilog.

Serilog is the newest of the popular logging software – it is quicker and supports structured logging.

Structured logging involves outputting the logs in json format so that the variables can be accessed programatically. To do that with text files you would need to parse the text and use RegEx to identify what information in the string you are looking for to extract.

Separate logging folders were created for the API and the admin website. Serilog will create both a text file and a json file. A new file will be created everyday.

public class AuthRepository : IAuthRepository

 {

 private readonly ILogger<AuthRepository> \_logger;

 ...

 // Constructor

 public AuthRepository(ILogger<AuthRepository> ...)

 {

 \_logger = logger;

 ...

 }

\_logger.LogInformation("{Time} - Error 1 - Login.AuthRepository.cs - {1} username not found", DateTime.UtcNow, username);



Txt Logs:



Json Logs:



**JWT Token Debate**

This app implements a single JWT access token, that expires after 30 min.

The React client stores this token in memory, not in local storage and not as a cookie. It makes it much harder for a hacker to access. The downside is that if the user refreshes the page, the token is deleted from memory and the user will have to login again. That’s not a great user experience, the user will have to log in every 30 mins or if they refresh the page. However, the aim was to make small apps the user wouldn’t want to stay logged into forever anyway.

Alternative strategies could be to use refresh tokens, so the client can log in once then remain logged in for a long time. The refresh token would be saved to session storage and then used to request new short-lived access tokens whenever the user needs to do something.

Cross site scripting and man in the middle attacks are going to be able to compromise the JWT token wither it’s stored locally or not. If the token is in local storage the hacker can download it and keep it. If the token is only saved in memory, the hacker can’t get to it but they can still use XSS to make requests on your behalf. Javascript code ran in your browser, when the user clicks a link can still make API calls using the JWT token in memory.

The only solution is to harden your app against XSS attacks using validation, or maybe only allowing the user’s IP address to make any requests.

Using an existing Authentication Server product would also be a good idea, as they have more detailed session policies to mitigate these attacks and are easier to update than writing your own auth code. They also have features you could easily use in the future like Oauth, if your app needs to talk to another app.