3rd year
BSc

Distributed Systems ACW Report

STUDENT NUMBER: 201706366

NAME: KAYLUM SNAPE

Table of Contents

1. APIs

An Application Programming Interface Is a layer of software that sits between services, such as a server and a client, to negotiate communication between them. Its purpose is to decouple software and extend the functionality of a service outside of itself, by distributing the execution of tasks to existing services. This decoupled concurrent execution of code across multiple services increases maintainability, performance and security, while decreasing the amount of time it takes to build an application.

Because APIs are middleware, they can also provide a layer of security by authenticating and authorising actors before they can access or modify data. It also provides a layer of abstraction, so that the Intellectual Property of the code is not directly exposed and available to be reverse engineered. User data is also more secure, as it too is never fully exposed to the server.

APIs have a strict set of rules for managing requests between one another, detailed in their API documentation, here is one for Twitters Tweet Lookup ("Tweet Lookup | Twitter API," n.d.). These definitions include the endpoints, functions, classes, arguments, return types and what communication protocol to use, typically carried out over the internet using TCP to establish a connection and HTTP to send requests, though not exclusively, another common design pattern is the message buss.

APIs have functions that are publicly exposed through endpoints, allowing them to be remotely called by requests that conform to the APIs specification. Upon receiving a valid request, the API will route it to the correct endpoint where the actor calling the endpoint is authenticated and authorised via a database look up. Parameters may be extracted from the request call to be passed into the function inside the endpoint. The function will then be carried out by the service and indicate if it was successful in a response.

Also, when managing requests there is the ability to implement middleware design patterns such as pipes and filters to direct and marshal the data in the messages, doing common work here rather than the application allows the application to focus on the business logic.

The ACW follows the stateless REST paradigm. Once the server has completed the requested task, the server can forget about that actor, allowing its resources to be freed up. If that actor makes another request, the server must again look up that actor in the data base using its API key (user ID) and run through authentication and authorisation. In contrast to this, stateful services, such as a gaming server, keep the connection open remembering the client from request to request. Keeping data in memory about what the client is doing takes up more resources and it can be difficult to know when to delete the state and free up resources.

Stateless, while slower, scales well to an arbitrary number of unknown clients, whereas stateful provides faster, more efficient and customisable connections that take up more resources. It is worth noting that there is some in-between, where services can store (cache) a state for a finite amount of time, allowing for fast follow up requests.

2. WebAPI and Route Mapping

Microsoft Web API uses route mapping to ensure that web API requests get to the correct endpoint by defining the structure of expected URIs. The flow is Domain > Controller > Action > Params.

As an example, a request is directed to the web API via the base domain shown in green "https://localhost:44394/api/user/new?username=UserOne", the URI in red is checked against the route template seen in appendix A, we can see that it conforms. Following the API request to the user controller in appendix B, we can see our action method identified by its action name "New", ASP.NET Core routing is case insensitive. This action method has a return type of "ActionResult". Actions allowing us to return a wide range of HTTP status codes, such as 200 OK or Bad Request 400, with a JSON response body.

WebAPI lets us pass in parameters to functions using attributes such as [FromBody], [FromHeader] and [FromQuery]. The latter shown in appendix B allows us to put parameters in the URI, signified with the query notation '?' followed by the parameter name and '=' joining the parameter value. This is the blue section of our example URI.

3. HTTP Requests

HTTP requests use action verbs that map to CRUD database operations.

```
// api/user/new?username=UserOne
[HttpGet]
[ActionName("New")]
0 references | Kaylum Snape, 5 days ago | 1 author, 2 changes
public ActionResult Get([FromQuery] string username)
{
    // If the user exists, respond true, if not and also if string is empty, respond false.
    return Ok(UserDatabaseAccess.UserNameExists(_dbContext, username) ?
        "True - User Does Exist! Did you mean to do a POST to create a new user?" :
        "False - User Does Not Exist! Did you mean to do a POST to create a new user?");
}
```

Figure 1 - HTTP GET request.

Figure 1 shows use of a HTTP GET request that maps to a READ against the servers DB.

```
// api/user/new
[HttpPost]
[ActionName("New")]
0 references | Kaylum Snape, 3 days ago | 1 author, 3 changes
public ActionResult Post([FromBody] string jsonUsername)
{
    // If there is no string submitted in the body, from body fails. Otherwise, If the string is empty.
    if (string.IsNullOrWhiteSpace(jsonUsername))
    {
        return BadRequest(error: "Oops. Make sure your body contains a string with your username and your Content-Type is Content-Type:application/json");
    }

    // If the username is already taken.
    if (UserDatabaseAccess.UserNameExists(_dbContext, jsonUsername))
    {
        return StatusCode(403, value: "Oops. This username is already in use. Please try again with a new username.");
    }

    // Create a new user and return the ApiKey as a string to the client.
    var newUser = UserDatabaseAccess.PostUser(_dbContext, jsonUsername);
    return Ok(newUser.ApiKey);
}
```

Figure 2 - HTTP POST request.

Figure 2 shows use of a HTTP POST request that maps to a CREATE against the servers DB.

```
// api/user/removeuser?username=UserOne
[HttpDelete]
[ActionName("RemoveUser")]
// Authenticate who we are talking to in CustomAuthenticationHandler.
// Verify the authentic user is authorised to carry out this action in CustomAuthorizationHandler.
[Authorize(Roles = "Admin, User")]
0 references | Kaylum Snape, 3 days ago | 1 author, 2 changes
public ActionResult Delete([FromHeader] string apiKey, [FromQuery] string username)
{
    var user = UserDatabaseAccess.GetUser(_dbContext, apiKey, username:null);
    UserDatabaseAccess.LogAction(_dbContext, user,
        logString:$"/user/removeuser called for {user.UserName}.");

    return Ok(UserDatabaseAccess.DeleteUser(_dbContext, apiKey, username));
}
```

Figure 3 - HTTP DELETE request.

Figure 3 shows use of a HTTP DELETE request that maps to a REMOVE against the servers DB.

4. API Key, Authentication, Authorisation, and Middleware

In the ACW, the HTTP message header is used to send an ApiKey, that is the user ID stored in the DB.

```
var apiKey:StringValues = Context.Request.Headers["ApiKey"];

// If the ApiKey is not valid, the user does not exists in the DB.
var user = UserDatabaseAccess.GetUser(DbContext, apiKey, username:null);
if (user is null)
{
    // Fail authentication, causing HandleChallengeAsync to be called.
    return Task.FromResult(AuthenticateResult.Fail("ApiKey not valid."));
}
```

Figure 4 - Get ApiKey from header.

This allows us to check if the user exists in the database, figure 4, performed when we pass through our firs middleware class that authenticates the user.

```
// Create Claims.
var claims = new[]
{
    new Claim(type:ClaimTypes.Name, value:user.UserName),
    new Claim(type:ClaimTypes.Role, value:user.Role.ToString())
};

// Add claims to ClaimsIdentity.
var identity = new ClaimsIdentity(claims, authenticationType:"ApiKey");

// ClaimsPrincipal created from the identity.
var claimsPrincipal = new ClaimsPrincipal(identity);

// Generate a new AuthenticationTicket from claimsPrincipal.
var ticket = new AuthenticationTicket(claimsPrincipal, this.Scheme.Name);

// Return a Success AuthenticateResult.
return Task.FromResult(AuthenticateResult.Success(ticket));
```

Figure 5 - Build Authentication Ticket.

Authentication class then builds claims against the user, this gets their username and role. Figure 5.

```
// If the user is not null and it's identity has been authenticated (from CustomAuthenticationHandler).
if (context.User != null && context.User.Identity.IsAuthenticated)
{
    // If the user is in the required role to perform the action it's asking for.
    if (requirement.AllowedRoles.Any(role:string => context.User.IsInRole(role)))
    {
        // Mark requirement as succeeded, they have been authorised.
        context.Succeed(requirement);
        return Task.CompletedTask;
    }

    // If the user is not in the required role, they are not authorised.
    context.Fail();
    HttpContextAccessor.HttpContext.Response.StatusCode = 403;
    HttpContextAccessor.HttpContext.Response.WriteAsync(text:"Forbidden. Admin access only.");
}
```

Figure 6 - Check authorisation.

```
[ActionName("ChangeRole")]
[Authorize(Roles = "Admin")]
0 references | Kaylum Snape, 3 days ago | 1 author, 1 change
public ActionResult ChangeRole([FromHeader] string apiKey, [FromBody] ChangeRole jsonRequest)
{
```

Figure 7 - [Authorize] attribute.

We use their role in the Authorisation class, figure 6, to see if they are authorised to execute the code they are requesting by checking against requirements.AllowedRoles. This value is set by the endpoint attribute [Authorize], as seen in figure 7.

An API Key is a good way to identify users, as it is a guid, a unique arbitrary value. It is not safe in this project as it is not encrypted, or even encoded, making it accessible to anyone who intercepts our packets, allowing them to impersonate us. If I were using this API key in practice, I would encrypt it using RSA where the only way to decrypt it is with the servers' private key.

5. Asymmetric Cryptography: RSA Algorithm

RSA keys are generated using complex computations from number theory to find three values that satisfy the equation $(\boldsymbol{me})\boldsymbol{d} \equiv \boldsymbol{m} \pmod{\boldsymbol{n}}$ for all $\boldsymbol{m}$ in the range $[0...\boldsymbol{n})$, where $\boldsymbol{n}$ is the modulus that defines the key length, $\boldsymbol{e}$ is the public key exponent and $\boldsymbol{d}$ is the private key exponent. The public and private keys are made up of $\boldsymbol{n}$ and their respective exponents, $\boldsymbol{e}$, $\boldsymbol{d}$. ("The RSA Cryptosystem - Concepts," n.d.)

6. Symmetric Cryptography: AES Algorithm

The AES algorithm is a block cipher that splits plain text into blocks of 16 bytes and then for each block shifts the block rows, mixes the columns using XOR, adds diffusion and finally adds a round key.

7. Databases and .Net Core Entity Framework

Entity Framework is a Microsoft framework for SQL Database that converts between Objects and tables, removing the necessity to write SQL. It handles basic CRUD operations allowing you to perform queries using LINQ, as if you were searching through C# Objects.

EF can be used in three major ways. In the code first approach, your code defines the database and mapping that EF creates for you. This gives you full control of your code but little over your DB, and so is typically just used for storage. Model first, used in the ACW, allows you to define the structure of your objects and how they relate to each other, leaving EF to create the database from them. You lose a little control in code and the DB but is efficient and productive for small projects. Database first takes an existing database and generates the code entities and maps for you, this gives you control over the DB, but depending on the changes, you may have to modify how you code works. Information expands upon ("entity framework - Code-first vs Model/Database-first," n.d.).

EF uses the concept of migrations to update the DB. It is comparable to version control, where you first make changes and create a commit, an intent. EF then generates the changes that you would like to make, allowing you to review them before using the command Update-Database to apply the changes to the database.

8. Reflective Statement about the code project

All tasks were completed in their entirety. I used Microsofts documentation, stack overflow posts and my peers to achieve the desired functionality. Postman also enabled me to easily send and receive requests against my server and the test server, saving the responses and comparing functionality, appendix C.

```
protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, RolesAuthorizationRequirement requirement)
{
    #region Task6
    // If the user is not null and it's identity has been authenticated (from CustomAuthenticationHandler).
    if (context.User != null && context.User.Identity.IsAuthenticated)
    {
        // If the user is in the required role to perform the action it's asking for.
        if (requirement.AllowedRoles.Any(role :string => context.User.IsInRole(role)))
        {
            // Mark requirement as succeeded, they have been authorised.
            context.Succeed(requirement);
            return Task.CompletedTask;
        }

        // If the user is not in the required role, they are not authorised.
        context.Fail();
        HttpContextAccessor.HttpContext.Response.StatusCode = 403;
        HttpContextAccessor.HttpContext.Response.WriteAsync( text: "Forbidden. Admin access only.");
    }
    // TODO: Mention in report that I was setting 403 response outside of authentication check,
    // so I just moved it inside so it only sets the 403 response when the user is not in the
    // correct role (fails authorisation), leaving the response free to be set to 401 when
    // failing authentication.

    // Also mention that I was returning a JsonSerializer.Serialize("Forbidden. Admin access only.")
    // Meaning that the response included the "", I just returned the text instead.

    return Task.CompletedTask;
    #endregion
}
```

Figure 8 - Set response in wrong place.

```
// Deal with 401 challenge concerns, when user fails authentication.
// We don't challenge the user, just deny access.
// https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authentication.authenticationhandler-1.
0 references | Kaylum Snape, 4 days ago | 1 author, 3 changes
protected override async Task HandleChallengeAsync(AuthenticationProperties properties)
{
    var messagebytes:byte[] = Encoding.ASCII.GetBytes(s:"Unauthorized. Check ApiKey in Header is correct.");
    Context.Response.StatusCode = 401;
    Context.Response.ContentType = "application/json";
    await Context.Response.Body.WriteAsync(messagebytes, offset:0, count:messagebytes.Length);
}
```

Figure 9 - HandelChallengeAsync

I had an issue where I was failing authentication but receiving a 403 response, this turned out to be because I was setting the response outside of the user authenticated check, figure 8. Once moving it inside the response was left free to be set by Handle Challenge, returning 401, figure 9.

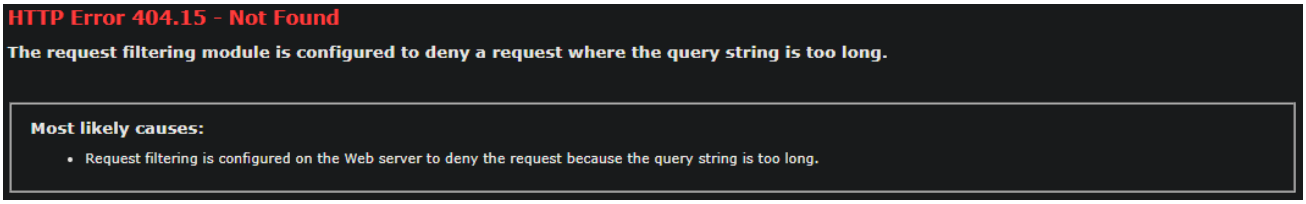

Figure 10 - Query string is too long.

Task 14 gave me trouble, I believed the error lied in the length of my hashed values, as they exceed the accepted HTTP URL length, shown in the error, figure 10. After some guidance from John, it was realised that my RSA encrypted bytes where 256, not 128. This was because I had originally set the key length to 2048 and that value was stored in the Machine Key Store, after creating a fresh store, my key length was set to 1024 bits, 128bytes and the request sent.

References

Tweet Lookup | Twitter API [WWW Document], n.d. Available online: https://developer.twitter.com/en/docs/twitter-api/tweets/lookup/introduction [accessed 15/4/2021].

The RSA Cryptosystem - Concepts [WWW Document], n.d. Available online: https://cryptobook.nakov.com/asymmetric-key-ciphers/the-rsa-cryptosystem-concepts [accessed 19/4/2021].

entity framework - Code-first vs Model/Database-first [WWW Document], n.d. . Stack Overflow. Available online: https://stackoverflow.com/questions/5446316/code-first-vs-model-database-first [accessed 16/4/2021].

Appendices

Appendix A – BaseController, Route template

```
[ApiController] // Tells the framework this is a controller containing endpoints.
// Route is the path to these endpoints.
// [controller] by convention is the controller class name minus the "Controller" suffix.
// [action] attribute tells route to pay attention to action names.
[Route(template: "api/[Controller]/[Action]")]
7 references | Kaylum Snape, 6 days ago | 1 author, 1 change
public abstract class BaseController : ControllerBase
{
    /// <summary>
    /// This DbContext contains the database context defined in UserContext.cs
    /// You can use it inside your controllers to perform database CRUD functionality
    /// </summary>
    1 reference | Kaylum Snape, 6 days ago | 1 author, 1 change
    protected Models.UserContext DbContext { get; set; }
    3 references | Kaylum Snape, 6 days ago | 1 author, 1 change
    public BaseController(Models.UserContext dbcontext)
    {
        DbContext = dbcontext;
    }
}
```

Appendix B – UserController, New action method

```
1 reference | Kaylum Snape, 3 days ago | 1 author, 5 changes
public class UserController : BaseController
{
    private readonly UserContext _dbContext; // Not to be changed by the controller.

    // Pass in UserContext through dependency injection.
    0 references | Kaylum Snape, 6 days ago | 1 author, 1 change
    public UserController(Models.UserContext dbcontext) : base(dbcontext)
    {
        _dbContext = dbcontext;
    }

    #region TASK4
    // api/user/new?username=UserOne
    [HttpGet]
    [ActionName("New")]
    0 references | Kaylum Snape, 4 days ago | 1 author, 2 changes
    public ActionResult Get([FromQuery] string username)
    {
        // If the user exists, respond true, if not and also if string is empty, respond false.
        return Ok(UserDatabaseAccess.UserNameExists(_dbContext, username) ? "True - User Does Exist!
    }
```

Appendix C – Postman.