2020

Distributed System Report

DISTRIBUTED SYSTEMS API – TASK 15

BENJAMIN STANLEY – 201603759

1. An Application Programming Interface (API) is essentially a messenger that takes requests and delivers them to a system which performs functions, and then returns this response back to the user.

   This API is stateless as the server forgets about the client after responding to the request. When the client makes a new request, the server can then find the ID and validate the client. Each request for a stateless interaction must include all the required information. This differs from Stateful Server which remembers the client after the request.

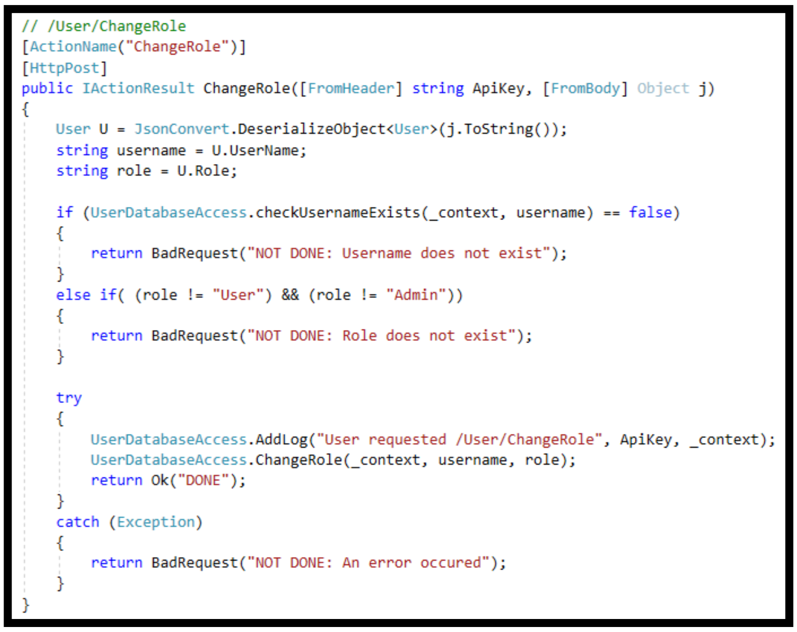
2. Route mapping involves the directing to a specific controller depending on the request path. At the top of a controller will be *[Route("api/[controller]/[action]")]* so that various named actions can be added and called with parameters in the query, header, and body of the request.

   The use of the ID parameter in WebAPI can allow an integer to be passed into a controller action as a parameter, i.e. passing in an ID in the query, and routing to an action that will return a book in a database which has correlating ID.

3. The various HTTP methods of requests determine what action the WebAPI will select. The request type can be explicitly stated by including an attribute like [HttpGet] above the method.

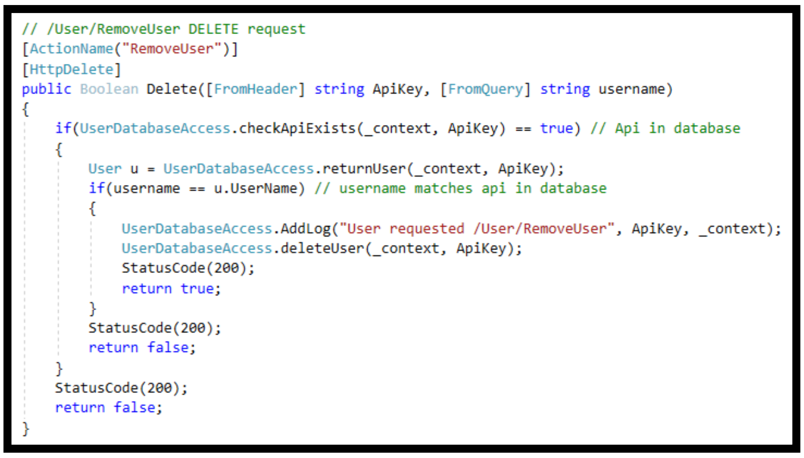
   The Get, Post and Delete requests can essentially be mapped to the CRUD operations (on server data). A Get request can be considered a retrieve action, Post as a create action, and Delete self explanatorily a delete action.

   Below illustrates the implementation of these requests in my Server Project.



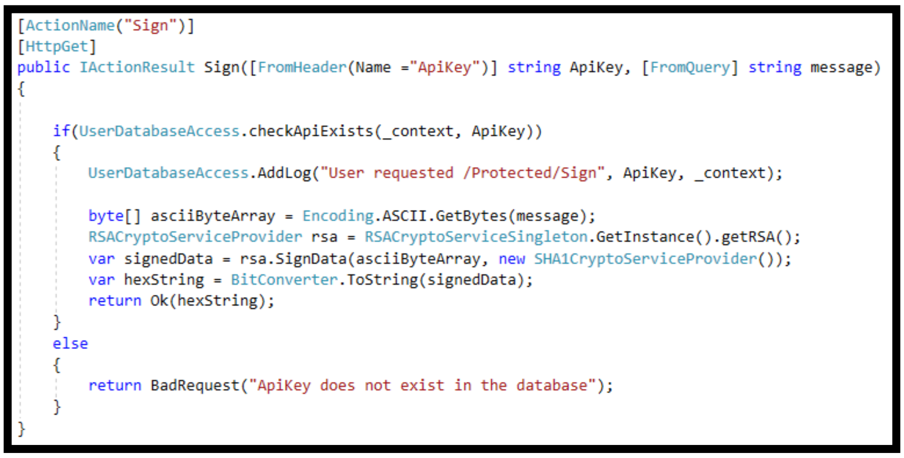
```
// /User/ChangeRole
[ActionName("ChangeRole")]
[HttpPost]
public IActionResult ChangeRole([FromHeader] string ApiKey, [FromBody] Object j)
{
    User U = JsonConvert.DeserializeObject<User>(j.ToString());
    string username = U.UserName;
    string role = U.Role;

    if (UserDatabaseAccess.checkUsernameExists(_context, username) == false)
    {
        return BadRequest("NOT DONE: Username does not exist");
    }
    else if( (role != "User") && (role != "Admin"))
    {
        return BadRequest("NOT DONE: Role does not exist");
    }

    try
    {
        UserDatabaseAccess.AddLog("User requested /User/ChangeRole", ApiKey, _context);
        UserDatabaseAccess.ChangeRole(_context, username, role);
        return Ok("DONE");
    }
    catch (Exception)
    {
        return BadRequest("NOT DONE: An error occured");
    }
}
```

Figure 1: Post Request (ChangeRole)



```
// /User/RemoveUser DELETE request
[ActionName("RemoveUser")]
[HttpDelete]
public Boolean Delete([FromHeader] string ApiKey, [FromQuery] string username)
{
    if(UserDatabaseAccess.checkApiExists(_context, ApiKey) == true) // Api in database
    {
        User u = UserDatabaseAccess.returnUser(_context, ApiKey);
        if(username == u.UserName) // username matches api in database
        {
            UserDatabaseAccess.AddLog("User requested /User/RemoveUser", ApiKey, _context);
            UserDatabaseAccess.deleteUser(_context, ApiKey);
            StatusCode(200);
            return true;
        }
        StatusCode(200);
        return false;
    }
    StatusCode(200);
    return false;
}
```

Figure 2: Delete Request (RemoveUser)



```
[ActionName("Sign")]
[HttpGet]
public IActionResult Sign([FromHeader(Name ="ApiKey")] string ApiKey, [FromQuery] string message)
{

    if(UserDatabaseAccess.checkApiExists(_context, ApiKey))
    {
        UserDatabaseAccess.AddLog("User requested /Protected/Sign", ApiKey, _context);

        byte[] asciiByteArray = Encoding.ASCII.GetBytes(message);
        RSACryptoServiceProvider rsa = RSACryptoServiceSingleton.GetInstance().getRSA();
        var signedData = rsa.SignData(asciiByteArray, new SHA1CryptoServiceProvider());
        var hexString = BitConverter.ToString(signedData);
        return Ok(hexString);
    }
    else
    {
        return BadRequest("ApiKey does not exist in the database");
    }
}
```

Figure 3: Get Request (Sign)

4. Upon the client requesting the creation of a new user, the server adds this user to the database, giving them a unique API key and role (by default ‘user’, otherwise ‘admin’ if no admin present in the system). The API key is then returned to the client and displayed on the console, after which it will never be seen again. The API key can be passed in the header for requests, e.g. removing a user and for actions requiring admin status. This is good for identifying users as it is a unique identifier, however as it is a randomly generated GUID, the string value for this is often long and complicated, rendering it difficult to remember.

   In this project however, the API key would not be considered safe as it is essentially a user password, whilst only being saved in the database as the original plaintext. This could be prevented by allowing choice of a unique username and memorable secure password (or still use the GUID API key), which would be hashed when saving to the database.

5. RSA is an example of an asymmetric cryptography function, used for encrypting, decrypting, signing and key exchange. This process involves use of a public and a private key. The message is encrypted initially using the public key, which is shared openly. After this, the message can only be decrypted by the private key due to its distinct mathematical properties. This key is as expected kept a secret, only the RSA user decrypting the message having access.

The four steps involved in the RSA algorithm are as follows:

1) Key generation
    a. Calculate $n = pq$, with p and q being two distinct prime numbers.
    b. Calculate $\lambda(n)$ = lowest common multiple of $(p\text{-}1, q\text{-}1)$.
    c. Choose and integer e between 1 and $\lambda(n)$.
    d. Calculate $d = e^{-1} \pmod{\lambda(n)}$.
    e. The public key consists of mod n and e, whilst the private key consists of d.
2) Key distribution – distribution of the public key.
3) Encryption – encryption of the message using public key.
4) Decryption – decryption of the message using the secret private key.

6. AES is an example of a symmetric cryptography function, being considered far stronger than other examples of this. With symmetric-key cryptography, the private key is involved in both the encryption and the decryption. This algorithm works by converting plain text into cypher text, in the form of random characters.

This process involves 10-14 rounds of the following encryption:

1) SubBytes – mathematically translates the inverse of each byte via a lookup table.
2) ShiftRows – shifts last three rows in a direction.
3) MixColumns – creation of output column bytes using XOR (combination of the four bytes in each column).
4) AddRoundKey – each byte is combined with a round key byte using XOR.

The key used is kept private, and due to the high-level encryption would be far beyond the capabilities of modern computers to decrypt this by brute force.

7. Entity Framework (EF) is a tool for linking objects to table entries in SQL Databases and removes the need for writing SQL queries. It allows performing of CRUD operations within a database as if they were C# objects.

Model first includes the creation of visual diagrams followed by EF performing operations on it accordingly. This technique can be suitable for large data structures and is very incorporating of database change. Disadvantages include the source code being generated for us meaning limited control, and potential for loss of data when updating.

Database first includes building the database first, then allowing EF to complete the rest. This is suitable for pre-existing databases. Downsides include manual updates of the database and even less control on the autogenerated classes.

Code first is the reverse of this, creating the classes first, followed by EF generating the Database for us. This approach doesn't require visual diagrams, being more appropriate for smaller projects. However, this does require good C# knowledge for EF.

A migration is an EF tool which automatically will update the database when changes are made to the model e.g. addition/removal of a class variable, or addition of a separate table. This migration tool will help to prevent loss of data where applicable (unless explicitly being removed).

8. All tasks within this assignment were completed. Many of the tasks required referral to lecture notes or previously completed lab work. Tasks 5 and 6 confused me due to limited knowledge on claim types. Task 12 caused issues with the encryption/decryption however I realised I was overcomplicating the situation and resolved this by simplifying code. Task 13 caused trouble attaching log instances to the user instance however this was fixed via the use of lazy loading. With task 14, I had a problem with the AES encryption/decryption, solved by using MemoryStream, CryptoStream and StreamWriter.