Distributed Systems Coursework Report

Kai Tindall

April 2021

Contents

List of Figures

Chapter 1

APIs

1.1 What is an API?

An API (Application Programming Interface) is a way for programs to interact with other programs in a defined manner. It allows people to create their own client programs for services like Twitter, or Facebook, or most other web services.

APIs typically have well defined structures documented by the people who created them, because they are not self documenting. It is very hard to guess the structure of an API just by using it alone without documentation.

Most are also stateless mechanisms. This means that the API will not remember who you are after the request is dealt with unless you provide identification. In the coursework solution (which was stateless) this is the API key header variable.

Being stateless is good for the server resources, as it means that once the server has completed the request, it can release the resources back for another request that comes in. This means that a lot of requests can be served with relatively little hardware, improving the scalability of your solution. This is very important if you're planning on serving a lot of clients.

Some APIs are stateful however, This means that if you send multiple requests very close together then the server will remember you. This can be useful as it means less information needs to be included in each request, however it means that the server needs to keep more resources allocated for the potential of follow-up requests being sent. This is fine for solutions expecting small amounts of clients but it will not scale well.

1.2 Route Mapping and WebAPI

WebAPI is an API framework built by Microsoft using ASP.NET core, it makes writing your own API very easy and provides route mapping. Route mapping is a tool that can be used in WebAPI to map actions to the URL the request has been sent to. For example, a standard route mapping is "{base_URL}/{controller_name}/{action_name}", this means that if a request is sent to "{base_URL}/Hello/Talkback" then the method called Talkback in the Hello controller will be called in order to serve that request.

An action in WebAPI is a method within a controller that handles a request. It can access the request through properties of the controller, or WebAPI can inject some parameters of the request into the arguments of the action for you. This can be done by using tags such as [FromHeader], [FromBody], or [FromQuery] before the arguments in the action signature to pull them in from the request. This makes it easier to access parts of the request and means not having to write code to do the same thing.

1.3 HTTP Verbs

HTTP Verbs are the different kind of requests that can be sent, some examples are GET, POST, and DELETE. Two requests can go to the exact same URL but are served by different actions because the requests use different methods.

1.3.1 GET Requests

GET requests are used when a request wants to get data from the API. This might be something like getting what the weather is like in London. GET requests shouldn't be used to send data that needs to be stored in a database.

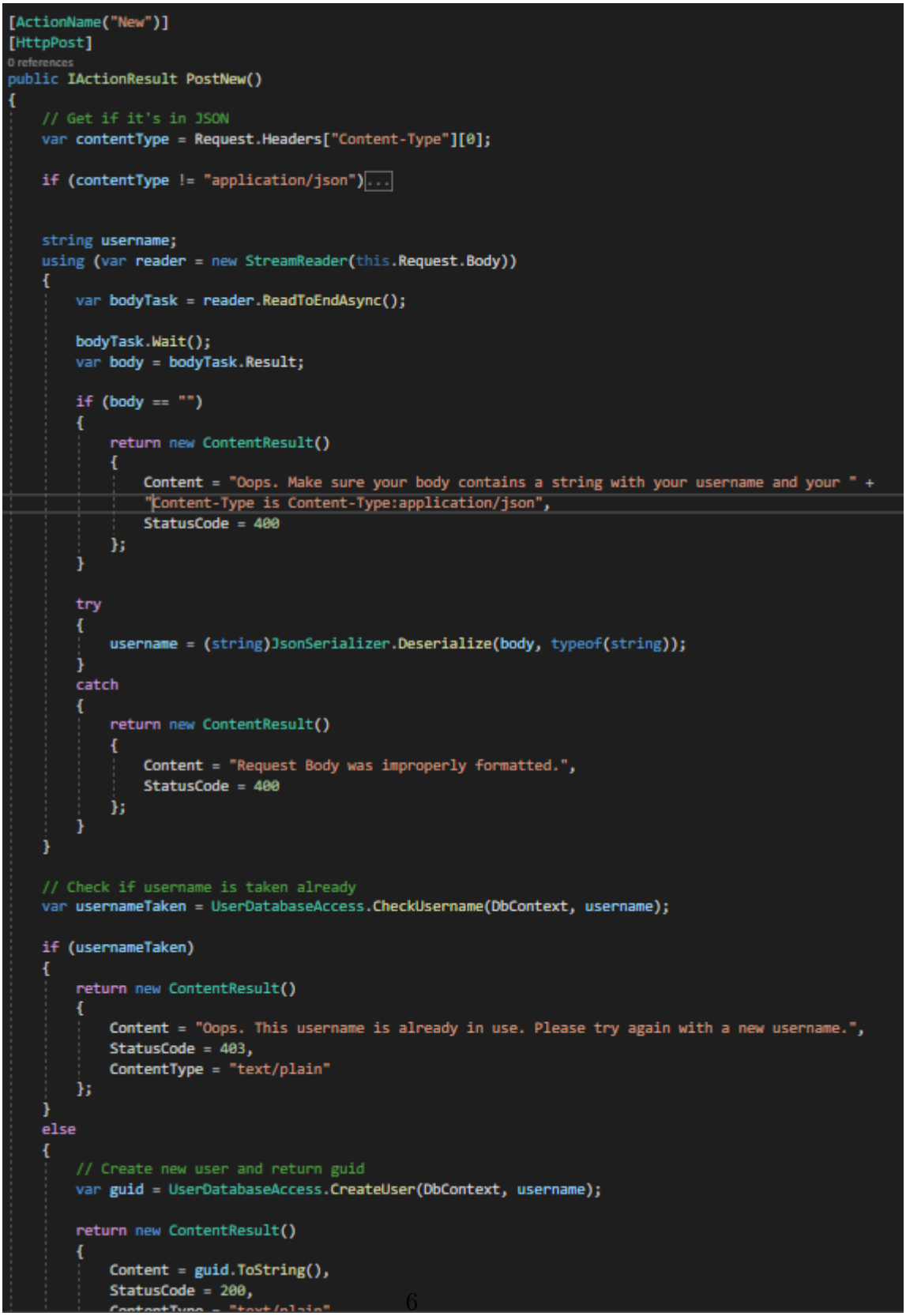
In the coursework a GET request is used in the Talkback/Hello action. This action doesn't require any additional information, and all it sends the client back is a simple hello world message.

```
[ActionName("Hello")]
[HttpGet]
0 references
public string HelloWorld()
{
    return "Hello world";
}
```

Figure 1.1: Hello World GET Request

1.3.2 POST Requests

POST requests should be used when you'd like to create data in the database. This method was used in the coursework when the user would like to create a new ApiKey for themselves. The user sends in the username they'd like to use in the POST request and if it's successful the server will create that new user in the database and return the associated ApiKey for the user to use.



```
[ActionName("New")]
[HttpPost]
0 references
public IActionResult PostNew()
{
    // Get if it's in JSON
    var contentType = Request.Headers["Content-Type"][0];

    if (contentType != "application/json")...


    string username;
    using (var reader = new StreamReader(this.Request.Body))
    {
        var bodyTask = reader.ReadToEndAsync();

        bodyTask.Wait();
        var body = bodyTask.Result;

        if (body == "")
        {
            return new ContentResult()
            {
                Content = "Oops. Make sure your body contains a string with your username and your " +
                "Content-Type is Content-Type:application/json",
                StatusCode = 400
            };
        }

        try
        {
            username = (string)JsonSerializer.Deserialize(body, typeof(string));
        }
        catch
        {
            return new ContentResult()
            {
                Content = "Request Body was improperly formatted.",
                StatusCode = 400
            };
        }
    }

    // Check if username is taken already
    var usernameTaken = UserDatabaseAccess.CheckUsername(DbContext, username);

    if (usernameTaken)
    {
        return new ContentResult()
        {
            Content = "Oops. This username is already in use. Please try again with a new username.",
            StatusCode = 403,
            ContentType = "text/plain"
        };
    }
    else
    {
        // Create new user and return guid
        var guid = UserDatabaseAccess.CreateUser(DbContext, username);

        return new ContentResult()
        {
            Content = guid.ToString(),
            StatusCode = 200,
            ContentType = "text/plain"
```


Figure 1.2: New User POST Request

1.3.3 DELETE Requests

DELETE requests are exactly what they sound like, they're requests that can be used to instruct the server to delete some data. DELETE requests have been used in the coursework to allow clients to delete users from the database.

```
[Authorize(Roles = "Admin, User")]
[HttpDelete]
0 references
public bool RemoveUser([FromQuery]string username, [FromHeader]string ApiKey)
{
    Response.StatusCode = 200; // Set status code to OK 200

    // Check if ApiKey is in db
    if (!UserDatabaseAccess.CheckGuid(DbContext, ApiKey))
    {
        UserDatabaseAccess.WriteLog(DbContext, ApiKey, $"Tried to remove user {user
        return false;
    }

    // Check ApiKey and username line up
    var apiKeyUser = UserDatabaseAccess.GetUser(DbContext, ApiKey);

    if (apiKeyUser.UserName != username)
    {
        UserDatabaseAccess.WriteLog(DbContext, ApiKey, $"Tried to remove user {user
        return false;
    }

    // Delete user from db
    UserDatabaseAccess.DeleteUser(DbContext, ApiKey);

    UserDatabaseAccess.WriteLog(DbContext, ApiKey, $"Deleted user {username}");

    return true;
}
```

Figure 1.3: DELETE User Request

1.4 Use of the API key

The coursework uses an API key to allow the user to authenticate themselves and authorise actions. Usually this is done by attaching the API key into a

header value. There is middle-ware utilised in the server to check the API key before it gets to the executing the action.

The server's middle-ware first checks the request's authentication. Authentication is the process of checking if the API key is even linked to a real user. If the request is linked with a real user then authentication succeeds and moves on to authorisation. Some actions require heightened privileges to be accessed, if an action requires a certain user role then authorisation middleware kicks in to check if the authenticated user has the correct credentials. If not, they don't get access to the action.

Of course, there are some flaws in the current system. Having the API key unencrypted in a HTTP request header leaves it vulnerable to a malicious actor watching the network traffic and stealing the API key to use for themselves. This could be improved upon by using HTTPS. HTTPS encrypts all traffic between the client and server. However, a more systematic approach might be more useful.

OAuth is a system that allows you to use other peoples systems to authenticate users. By making users sign into their Microsoft, Google, Facebook, or even GitHub accounts, it allows you to rely on that companies security to keep malicious actors from gaining unwanted access.

Another option could be encrypting your API key. This would be relatively simple and doesn't rely on others not making mistakes. However, it does mean that there's more chance you could introduce a mistake yourself.

Chapter 2

Cryptographic Functions

2.1 RSA

The RSA algorithm is an asymmetric encryption algorithm. This means that there are two keys, a public and a private. The public key can encrypt data that can only be decrypted by the private key, and visa versa for the private key. This means that Client and Server can have an encrypted conversation by first exchanging entirely public keys that anyone can know.

You can also use RSA in the other way to sign messages. Signing messages means that you are proving the authenticity of the message. This is done by signing the message with the private key that can only be decrypted by the public key.

2.2 AES

AES is a symmetric encryption algorithm. This means there is only one key that both encrypts and decrypts. There is also an initialisation vector that is used to change where you start in the algorithm. This means that with the same key and different IVs, you can end up with different cipher text. With there only being one key and the nature of the API being stateless, it means that the key needs to be sent over with the request. But obviously you don't want to send the key in plain text, so to do this you need to first encrypt the key using the server's public RSA key.

Chapter 3

Database Access

A lot of the time APIs aren't very useful unless they are backed up by a database for permenent data storage. Because in the coursework Microsoft's WebAPI was used, it meant that Entity Framework could also be used.

3.1 Entity Framework

Entity Framework is a database access framework built by Microsoft that allows easy database access in C#. It allows databases to be represented in C# classes which are very easy to work with. There are a few different ways in which you can work with Entity Framework.

3.1.1 Code First

Code first means writing C# classes that represent the data structures first before using tools to convert these classes into a migration. A migration is another C# class that Entity Framework uses to make changes to the database using whatever flavour of SQL the database uses under the hood. These migrations have an Up and a Down function, which are the inverse of each other. This means that these migrations are reversible, if something goes wrong.

Migrations aren't perfect all of the time, it is automatically generated code after all. Therefore it is always wise to check migrations first before using them to update the database.

3.1.2 Model First

Building a database model first is great for people who like to plan out visually what the data will look like. The user will design graphically an entity relationship database using tools provided. Then, the same tool can be used to implement or change the structure of the real database. Similar to using migrations in code first.

3.1.3 Database First

Database first is the approach where you don't actually use Entity Framework to create the database, but rather you point Entity Framework to an existing database and it uses the structure to create all the matching C# classes. This approach is good if you're working with a database that you've already created and just want to use.

Chapter 4

Tasks Completed

I have completed all tasks laid out to the best of my abilities. My main problem was working with JSON data in the actions, however due to Microsoft providing outstanding documentation for all of their products, I was able to overcome this.