# Distributed Systems Programming 600089

## Distributed Systems API: Report

Student Number: 201601628

Submitted April 2020

Words: 1100

# 1. APIs

APIs expose some functionality of one system (server) to others (clients) without needing to know specific implementation details. Instead, the developer need only know what is available (e.g. endpoints) and what the API will return for a given call. They enable systems to be decoupled and distributed, facilitate code reuse, and can increase application security.

RESTful APIs rely on the HTTP protocol and define endpoints linked to server actions. Requests to an endpoint invoke the mapped function and return a HTTP response. This is the architecture used in this project.

The implemented API can further be described as stateless. Servers/APIs are either stateful or stateless. The former dictates that the server stores client information in memory between requests. This can monitor a client's 'state' and enable faster response times and more complex requests comprised of multiple communications. On the contrary, stateless servers do not store information about the clients and each request is a single, self-contained message; outside of a given request thread, the server has no knowledge its clients.

# 2. WebAPI Route Mapping

Route mapping describes the WebAPI mechanism for linking URI requests to server actions defined in *controllers*. Figure 1 illustrates how URIs are formed based on the controllers present in the WebAPI project. For example: a controller named, "*ProtectedController*", might be accessed via the URI, "*<host base-url>/api/Protected*".

```
[Route("api/[Controller]")]
```

*Figure 1: Simple form of route mapping in WebAPI.*

Using the routing format in Figure 1, method names are limited to Get(), Post() etc. Adding the "[ActionName]" attribute to the route mapping enables more flexibility in naming, as shown in Figure 2. In this case, the developer can continue to use conventional method names, like Get and Post, and then use the Action decorator to 'name' the method as demonstrated in Figure 3. Alternatively, HTTP decorators can specify the HTTP operation associated with the method and the method name can be anything the developer wishes; as shown in Figure 4.

```
[Route("api/[Controller]/[Action]")]
```

*Figure 2: Routing with specified action (method) names.*

```
[ActionName("Hello")]
0 references | Zak (Lenovo), 24 days ago | 1
public IActionResult Get()
```

*Figure 3: Using the [ActionName] decorator with conventional method names.*

```
[HttpDelete]
[Authorize(Roles = "Admin,User")]
0 references | Zak (Lenovo), 21 days ago | 1 author, 5 c
public bool RemoveUser([FromHeader
```

*Figure 4: Using HTTP decorators with custom method names.*

Similarly, there are several means of parameterising endpoints. One way is by convention, using an integer parameter, “id”. If the route is configured with an id component, as in Figure 5, the request URI can integrate an index. For example, a call to, “*<host-url>/api/examplecontroller/user/0”*, might return the 0th user stored in the database.

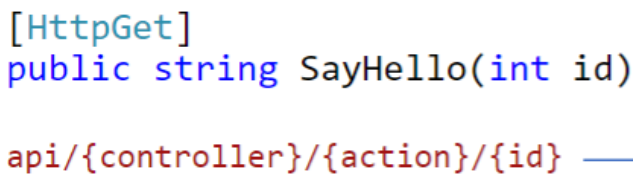


*Figure 5: Using the id parameter convention in WebAPI.*

## 3. HTTP Requests

GET, POST and DELETE describe fundamental HTTP requests. It is possible to create a comprehensive API with them and they are essential in any CRUD (Create Read Update Delete) API. GET requests handle *Read*, POST requests handle *Create* and *Update*, and DELETE requests handle *Delete*. Figures 6, 7 and 8 illustrate these requests in the implemented API.


*Figure 6: GET request in the User Controller, which checks if a username (provided in the URI) exists in the database.*


*Figure 7: POST request in the User Controller of the implemented API, which creates a new user in the database with the username present in the request body.*


*Figure 8: DELETE request in the User Controller, which deletes the user with a username specified in the URI and verifies the request with the API key present in the request header.*

## 4. API Keys

The implemented API authenticates requests by searching the database for the key presented in the request header. The server then searches the database for a record with the given key. If found, the API middleware verifies the user’s role if appropriate and either allows or disallows the endpoint to be reached.

Identification via API key is common in many APIs, including those provided by Microsoft Azure. It is a credible method of authentication but relies on the consuming API/developer to handle their keys securely. Often, keys must be present in the codebase.

Ultimately, API keys and their storage present attack vectors. A request can easily impersonate a user if the key is compromised. Drawing parallels to a user password; the implemented API is

particularly negligent in that the API keys are stored as plain text. Changing the implementation to instead store a (salt & peppered) hash of the API  key, using SHA256 for example, could increase security.

## 5. RSA Algorithm

RSA is an asymmetric encryption method comprised of public & private key-pairs. The private key is kept secret, server-side, whilst the public key can be distributed to clients. Data encrypted with one key is decrypted by the other. The steps are:

1. Calculate n as the product of 2 large prime numbers p and q.
2. Calculate $\phi(n)$ as (p-1)(q-1).
3. Choose e to be >1 and coprime to $\phi(n)$.
4. Public key is comprised of n and e (modulus and exponent).
5. Private key is calculated as 1mod $\phi(n)$.

## 6. AES Algorithm

AES is a symmetric encryption method, where the same key is used by both parties for encryption and decryption. It is suited to larger amounts of data than RSA and is often used in tandem. It is an iterative algorithm which forms chained blocks. The following are the steps are repeated each 'round':

1. Substitute plaintext bytes with values from a lookup table.
2. Rotate each row by n bytes, with carry (e.g. row n >> n bytes).
3. Mix columns via matrix multiplication.
4. XOR the 'round' key with the state array.

## 7. Entity Framework

Entity Framework abstracts database development and management to allow developers to focus on application development. It manages many intricacies of database access, such as lazy loading and concurrency.

There are several approaches to developing with EF: code-first, model-first and database-first.

- Code-first allows the developer to focus on business logic and the code generation with little concern for database model/development. EF then generates the database by mapping domain classes specified in the *context* class to tables.
- Model-first enables the developer to utilise the designer integrated with Visual Studio to model domain classes and their relationships. From the model, these classes are generated mapped by EF to produce the database.
- Database-first is a wizard-based approach that is well suited to scenarios where a database already exists. Unlike the other approaches, EF maps the existing database to generate the context and domain classes.

EF core uses *migrations* to create and update databases. Each migration defines an Up() and Down() method, describing the database objects to add or remove according to the context and domain classes. EF also manages migration versioning and warns of changes that may cause data loss.

## 8. Reflections

All tasks were completed fully. Logging was implemented as a new middleware to provide a more thorough and maintainable solution. The singleton pattern was used to create an internal RSA service,

ensuring the RSA key-pair was consistent between clients. The façade pattern was used to simplify calls that modified the database, whilst adding functionality for concurrency exceptions and retries.

Difficulties in interpreting the finer details of the specification, e.g. the exact behaviour of log archiving, were resolved by communicating with the module leader. The main stalling point was the implementation of the *Protected/AddFifty* method. Because all requests are received as a string, it was easy to neglect the underlying types and necessary conversions. In this case, some time was wasted before realising the API wasn't converting the decrypted message from a string to integer.