Michael Edgar

Computing with Games Development  Michael.edgar@students.ittralee.ie

Distributed computing project part 2

# Design Description

I decided to use the REST architecture for the redesign of my twitter messaging service. I chose REST over RPC for a number of reasons, but the main reason I chose it was my previous experience working with REST. While on work placement, I became quite experienced with using REST to send a receive messages. I used it on the command line, through Postman and even in Java through IntelliJ. For that reason, it made sense for me to adopt an approach using REST for this redesign.

REST is mainly known for its use of the four messages: GET, POST, PUT and DELETE, but REST also has other methods: PATCH, HEAD and OPTIONS. It uses HTTP to transport messages, which allows it the same level of scalability as the World Wide Web. Underneath HTTP sits the network layer (TCP/IP) which performs the actual transporting of these messages. There are five main principles to REST: give everything an ID, link things together, use standard methods, have multiple representations of resources for different needs and communicate statelessly

## Message Descriptions

Method: Login

Description: This method receives a String username and String password as parameters. It adds the username to an ArrayList of usernames and returns a String response. If the username or password are invalid, the user will receive a response describing the error. Otherwise, the user will receive a confirmation that they have logged in.

Parameters: username (String), password (String)

Returns: response (String)

Method: Upload

Description: This method accepts a String message and saves the message to an ArrayList of messages. If successful, it returns a success message. If there is an error in saving the message, the method returns a message describing the error.

Parameters: message (String)

Returns: response (String)

Method: Download

Description: This method requires no parameters, and returns a String containing all previously saved messages. If there are no messages saved on the server, this method returns a message saying that there are no saved messages

Parameters: NA

Returns: messages (String)

Method: Log Off

Description: This method takes a username as a parameter. It searches the ArrayList of usernames for the given username. If the username is found, it removes that username and returns a success message. If the username is not found, or if there is an error removing the username, it returns a response describing the error.

Parameters: username (String)

Returns: response (String)

# Implementation

To run the client and server, you will need IntelliJ IDEA. Open the TwitterMessagingProtocol project within IntelliJ and run the Server and Client. To do this, open the run configuration dropdown menu (Figure 2.1) and choose the GlassFish 5.0.1 run configuration. Press the play button to start the server. This will open a browser window with a simple Hello World message once the server is running. You can then start the server by changing the run configuration to TwitterClient. This will start the client that will allow you to login. You will be prompted to enter the username and then the password. Once both have been entered, a GET request will be made which will log in the user with the given username and password. Note: if you enter ‘fail’ in either the username or password section, this will cause the server to return a specified fail message to show an example of a failed login. A sample output can be seen in Figure 2.2. Figure 2.3 shows the project structure with HelloWorld defining a sample GET message that returns a String message (Figure 2.4). This class acts as the homepage when starting the server. The Login class shown in Figure 2.5 controls the server side of the application, in this case the login functionality. This contains two methods, the first is a GET request method that accepts a String username and String password as path parameters and returns a String result. The second method evaluates whether or not the login is valid. For the sake of this project, the login is only invalid if the username or password is equal to ‘fail’. The MyApplication class (Figure 2.6) overrides the getClasses method from the REST Application class. The getClasses method returns a HashSet of classes, in this case the Login and HelloWorld classes. Without this method, a request to the HelloWorld or Login methods would return a 404 Not Found Error. Lastly, the TwitterClient class (Figure 2.7) controls the client side of the application. The client prompts the user for a username and password and then makes a GET request to the server. The client allows the user to exit by entering a single period for the username or password.



Figure 2.1 Run Configuration Dropdown Menu

A screenshot of a cell phone

Description automatically generated

Figure 2.2 Sample Client Output

A screenshot of a cell phone

Description automatically generated

Figure 2.3 Project Structure

A screenshot of a cell phone

Description automatically generated

Figure 2.4 HelloWorld Class

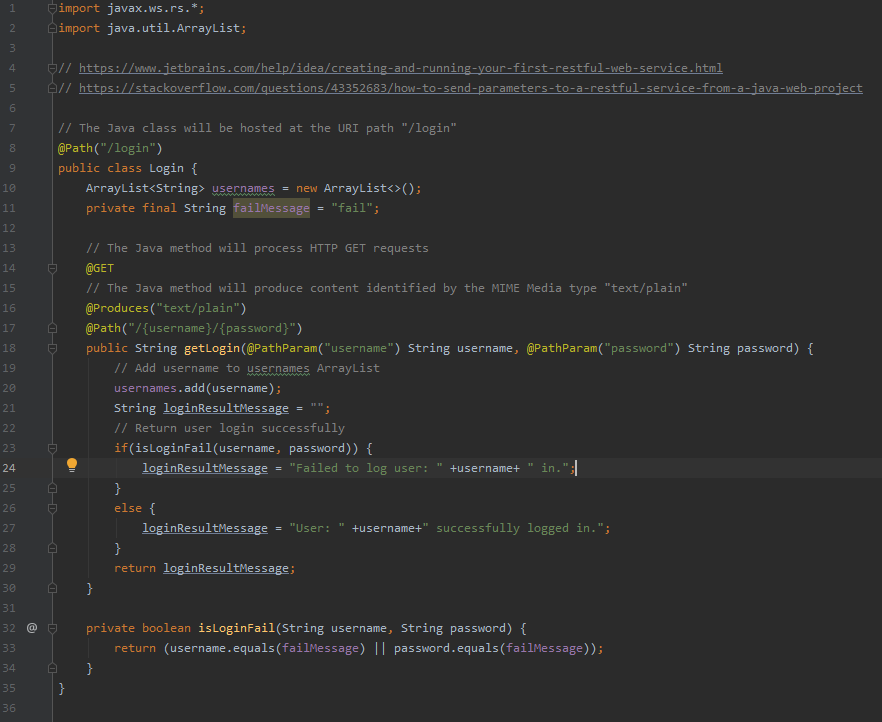


Figure 2.5 Login Class

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Figure 2.6 MyApplication Class

Figure 2.7 TwitterClient Class

# Comparison with Previous Implementation

In part one, I designed a protocol using sockets, threads, and a client/server interface. I built the protocol from the ground up, implementing SSLSockets for security and designing each of the three layers of the architecture (presentation, application, and service). However, in part two of the project, I redesigned this service using the REST architecture. While I still had to use a client/server interface, I no longer needed to worry about sockets as the message transport was handled by REST itself. In other words, REST handles the service layer of the architecture, so that I could focus on the presentation and application layers.

Similarly, for the first implementation, I needed to define each of the four message types, their message numbers, their responses, and their error codes. This contrasts greatly with the REST implementation which did not require any of these definitions. This is because the REST implementation already had defined messages that I could build on top of. This served to greatly reduce the complexity and development time required.

Another difference between the two implementations is the need for synchronisation. Since the first implementation used threads and sockets, I had to create methods that would connect the client to the server before sending messages and close this connection once the last message had been sent. This also required port numbers to connect the client to the server. The REST implementation did not require this connection. Instead, messages were sent directly without any need for connecting or disconnecting or port numbers.

As mentioned above, the first implementation required the use of a defined port number. The server would be run on a particular port and the client would connect using that port number to communicate with the server. This poses two potential problems: if this port isn’t opened to connections outside the network, this prevents users outside the network from connecting to the server and using the service. However, if the opened to connections outside the network, this poses a security risk as users now have access to the network through this port. These issues are completely avoided with the REST implementation. Since REST uses HTTP, it automatically sends messages through port 80. This means users can connect with the server without knowing the port or address of the network.

A big difference between the two approaches is the development time required for each. Part one required several days of development between research, implementation, troubleshooting and improving, whereas using REST reduced this development time to 2-3 hours. This is mainly because REST only needs 3 classes to implement the same service provided by five classes in part one.

One disadvantage to the REST implementation is latency. The original implementation in part one ran very quickly, with the server loading almost immediately and the client connecting within seconds. However, there seemed to be a small degree of latency within the REST implementation. This could be an issue if used in services that rely heavily on speed.

Something that the first implementation lacked is true scalability. Since it relied on threads, this implementation is limited by the RAM of the server and the resources required by each thread. This could cause an issue in a service with greater volume requirements. Contrast this with the REST implementation. Since REST sits on HTTP, it is highly scalable and therefore would not suffer the same restraints as the first implementation.

One thing that both implementations had in common was the use of SSL to secure messages. SSL or SSL/TLS encrypts the messages that are being sent and authenticates clients and servers using certificates to ensure security.