5 Technical Overview

Here we present a concise overview of the main technical aspects of how the application is implemented. Our istances of Spring Boot are configured in such a way to communicate with all the components needed. The configuration parameters are mainly stored in the `application.properties` file:

```
spring.data.mongodb.uri=mongodb://admin:password@mongo1:27017/test

spring.rabbitmq.host=rabbitmq
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
spring.rabbitmq.publisher-confirm-type=correlated
spring.rabbitmq.publisher-returns=true

lap.app.jwtSecret=[omitted]
lap.app.jwtExpirationMs=86400000
```

These parameters are then used in the corresponding Java classes to configure the connection to the database and the message broker. A generic request is handled similarly to this one:

```
@GetMapping(value = "/graph")
public ResponseEntity<Void> redirectGraphPage(HttpServletRequest request, @RequestParam String symbol)
    throws Exception {
  String queryString = request.getQueryString();
  String redirectUrl = "/stock.html";
  if (queryString != null) {
    redirectUrl += "?" + queryString;
  }
  return ResponseEntity.status(302).header("Location", redirectUrl).build();
}
```

The `@GetMapping` annotation is used to map the request to the corresponding method, and the function contains the logic to apply to respond to the request. The POST requests to `/signup` are handled in the following way:

```
PostMapping("/signin")
    public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginRequest loginRequest) {
        try {
            Authentication authentication = authenticationManager.authenticate(
```

```
                    new UsernamePasswordAuthenticationToken(
                        loginRequest.getUsername(),
                        loginRequest.getPassword()));
            SecurityContextHolder.getContext().
                setAuthentication(authentication);
            String jwt = jwtUtils.generateJwtToken(
                authentication);

            UserDetailsImpl userDetails = (UserDetailsImpl)
                authentication.getPrincipal();
            List<String> roles = userDetails.getAuthorities().
                stream()
                    .map(GrantedAuthority::getAuthority)
                    .collect(Collectors.toList());

            return ResponseEntity.ok(new JwtResponse(jwt,
                    userDetails.getId(),
                    userDetails.getUsername(),
                    userDetails.getName(),
                    userDetails.getSurname(),
                    userDetails.getEmail(),
                    roles));
        } catch (AuthenticationException e) {
            // Authentication failed
            return ResponseEntity.status(HttpStatus.
                UNAUTHORIZED).build();
        } catch (Exception e) {
            // Other exception occurred
            return ResponseEntity.status(HttpStatus.
                INTERNAL_SERVER_ERROR).build();
        }
    }
```

As we can see, the `AuthenticationManager` is the main Spring Security interface for authenticating a user. It is used to decide if the user credentials are valid or not. If they are valid, an `Authentication` object is returned, and a JWT is set for the user. As an example for the communication with MongoDB, we show the code for the `UserRepository`:

```
@Repository
public interface UserRepository extends MongoRepository<User,
    String> {

    @Query("{'username' :?0}")
    User findUserByUsername(String username);

    @Query(value="{'username' : ?0}", delete = true)
```

```
8       public void deleteByUsername(String username);
9
10      Boolean existsByUsername(String username);
11
12      Boolean existsByEmail(String email);
13
14      @SuppressWarnings("unchecked")
15      User save(User user);
16
17  }
```

The `UserRepository` is an interface that extends the `MongoRepository` interface. The latter is a Spring Data interface for MongoDB. It provides methods for CRUD operations by default, and we can also define custom methods. In this case, we defined the methods to find a user by username, to delete a user by username, and to check if a user exists by username or email.
For RabbitMQ, since it would be too long to show all the code, we refer to the source code attached to this document.

Regarding the logic of the NodeJS istances, we can see that a generic api requests gets retrieved from the queue in the following way:

```
1   amqp.connect('amqp://rabbitmq', function(err, conn) {
2     conn.createChannel(function(err, ch) {
3       var from_queue = 'spring_node';
4       ch.assertQueue(from_queue, { durable: false });
5
6       ch.consume(from_queue, function(msg) {
7
8         var msgId = JSON.parse(msg.properties.messageId);
9         var corrId = msg.properties.correlationId;
10        var id = msgId.id;
11        var replyTo = msg.properties.replyTo;
12
13        switch(msgId.type){
14          case "GRF":
15            var symbol = msgId.ticker;
16            calls.performGraph(symbol, corrId, replyTo, id);
17            break;
18
19          case "STCK":
20            var symbol = msgId.ticker;
21            calls.performStockData(symbol, corrId, replyTo, id);
22            break;
23
24          case "DATI":
25            calls.performGetData(corrId, replyTo, id);
```

```
26            break;
27
28          case "NEWS":
29            calls.performNews(corrId, replyTo, id);
30            break;
31
32          case "BOX":
33            calls.performBoxes(corrId, replyTo, id);
34            break;
35
36          case "SEARCH":
37            var search=msgId.q;
38            calls.performSearchCall(search, corrId, replyTo, id)
                  ;
39            break;
40
41          case "CRYPTO":
42            calls.performCrypto(corrId, replyTo, id);
43            break;
44
45          default:
46            sendToQueue("default", corrId, replyTo, id);
47            break;
48        }
49      }, {noAck: true});
50    });
51  });
```

Based on the label attached to the message, the corresponding function is called. A generic API call is implemented as follows:

```
1  exports.performStockData = function (symbol, corrId, replyTo,
       id) {
2      const opts = {
3          method: 'GET',
4          json: true,
5          url: "https://finnhub.io/api/v1/quote?symbol="+symbol+
               "&token="+api_key2
6      };
7
8      request(opts, function (error, response, body) {
9          if (error) throw new Error(error);
10
11         const obj = {price: (body.c).toFixed(2), change: (body
               .dp).toFixed(2), high: body.h.toFixed(2), low: body
               .l.toFixed(2)};
12         server.sendToQueue(JSON.stringify(obj), corrId,
```

```
            replyTo, id);
    });
}
```

As we can see, the function performs a GET request to the Finnhub API, and then sends the result to the queue.
Lastly, we give a brief overview of the MongoDB replica set configuration. The configuration file is the following:

```
#!/bin/bash

DELAY=15

mongosh -u admin -p password --eval 'var config = {
    "_id": "rs0",
    "version": 1,
    "members": [
        {
            "_id": 1,
            "host": "mongo1:27017",
            "priority": 2
        },
        {
            "_id": 2,
            "host": "mongo2:27017",
            "priority": 1
        },
        {
            "_id": 3,
            "host": "mongo3:27017",
            "priority": 1
        }
    ]
};
rs.initiate(config, { force: true });'

echo "****** Waiting for ${DELAY} seconds for replicaset
    configuration to be applied ******"
sleep $DELAY
mongosh -u admin -p password --eval "$(cat /scripts/mongo-init
    .js)"
```

The script is executed when the MongoDB container is started. It first initializes the replica set, and then executes the `mongo-init.js` script, which creates the admin user and the database for the application. The script is the following:

```
rs.status();
db.createUser(
```

```
                {
                        user: "admin",
                        pwd: "password",
                        roles: [
                                { role: "readWrite", db: "test
                                    " },
                        ]
                }
);

db.users.drop();
db.roles.drop();
db.data.drop();
db.createCollection("users");
db.createCollection("roles");
db.createCollection("data");
db.createCollection("favorites");
db.roles.insertMany([
        { name: "ROLE_USER" },
        { name: "ROLE_ADMIN" }
]);
```

Lastly, the `deploy.sh` script used to deploy the application is the following:

```
#!/bin/bash

if [[ $1 == "--build" ]]; then
  echo "Building Docker images..."
  xterm -e "docker-compose build"
fi

# Start the Docker Compose stack in detached mode
xterm -e "docker-compose up" &

echo "Remember to execute docker-compose down when you are
    done."
sleep 1

# Wait for the containers to start up
echo "Waiting for containers to start up..."
sleep 10

# Get the container ID for the container with 'mongo1' in its
    name
MONGO1_CONTAINER_ID=$(docker container ls --all | grep
    lapproject_mongo1 | awk '{print $1}')
# Execute the rs-init.sh script in the 'mongo1' container
```

```
21 echo "Executing rs-init.sh in mongo1 container..."
22 docker exec -it $MONGO1_CONTAINER_ID /scripts/rs-init.sh
```