StocksPlayground

Elena Bianchini 1796906 - Andrea De Murtas 1883216 - Tiziano Finizzi 1915610

July 4, 2023

Contents

1 Introduction

StocksPlayground is a web application whose aim is to provide a tool for users to view and analyze the stock and cryptocurrency market, and it also provides the possibility to make an account to use for saving favorite stocks and for keeping them all in one place. It is mainly built like a dashboard, composed of a series of views that group together relevant information, like the hottest sector of the market or the biggest gainer of the moment. It is also possible to view a treemap of the market, that is, a structure composed of block where each block represents a ticker, and whose size depends on the stock's market capitalization and whose color varies based on how much that company is gaining or losing at that moment in time. In addition, for every ticker there is a dedicated page where a candlestick graph is displayed, together with some relevant information about the symbol. These are just some examples of the features of StocksPlayground, as we will present them all in the next pages.

2 Infrastructure

The application is built on top of an infrastructure that was implemented using Docker and Docker Compose, as we can see in the following figure:

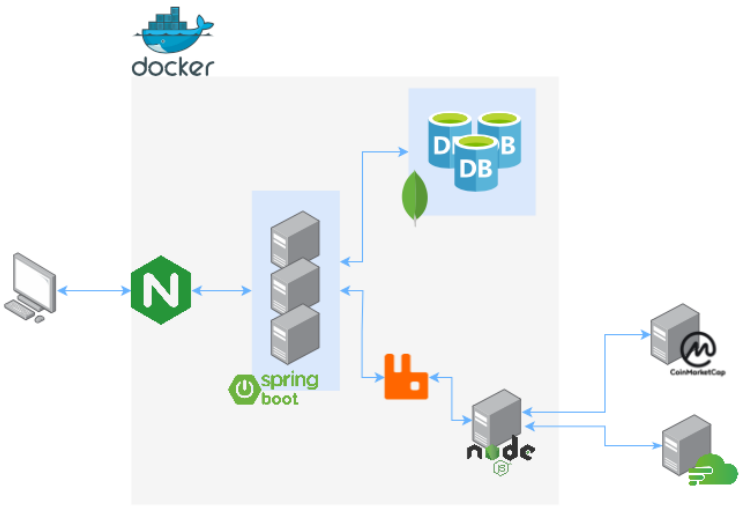

Figure 1: Infrastructure

- The `NGINX` is used as a reverse proxy for the application, as it permits to implement useful feature like caching and load balancing. We decided to use a round-robin load balancing strategy for distributing requests among the different instances of the server.

- The server itself is implemented through `Spring Boot`, a framework for building applications in Java. It makes it simple to create stand-alone, production-grade Spring based Applications that you can "just run". We decided to use this framework because it is very simple to use and it provides a lot of useful features, like the possibility to use the `Spring Security` module for managing the authentication and authorization of the users.
- The `NodeJS` server is used for performing API requests, as Javascript makes it very simple to perform HTTP requests and handle JSON objects.
- The `RabbitMQ` is used to build an RPC system, so that it behaves as a middleware that allows the communication between the instances of Spring Boot and the one of NodeJS.
- The `MongoDB` database is used for storing the data of the users, like their username, password and favorite stocks. We decided to use this database because it is production ready, it is very simple to use and it is very flexible, as it is a NoSQL database. It is also composed of JSON documents, a format that is easy parsable and not too verbose. Moreover, we deployed it as a replica set to ensure dependability.
- For the API to use for retrieving the data about the stocks, we decided to use `Finnhub`, a free API that provides real-time information about the stock market, and `CoinMarketCap`, since the latter had some more information we needed.

3 Deployement

The use of Docker Compose makes it straightforward to deploy the application, since we rely on a `docker-compose.yml` file that specifies all the services that compose the application, and how they should be dispatched. In particular, in the file we specify things like container names, ports that should be exposed, subnetworks and so on. Notably, in this file we also make a part of the configuration needed to make `MongoDB` work in replicaset mode.
The deployement itself is very simple. We just need to run `docker-compose build` to build the images, and then `docker-compose up` to start the containers. At a certain point, we created a script to automatize this process, and also to perform some steps needed for the set up of the application, which we will dive into later.

4 Feature Overview

After starting the application, we can navigate to the homepage:

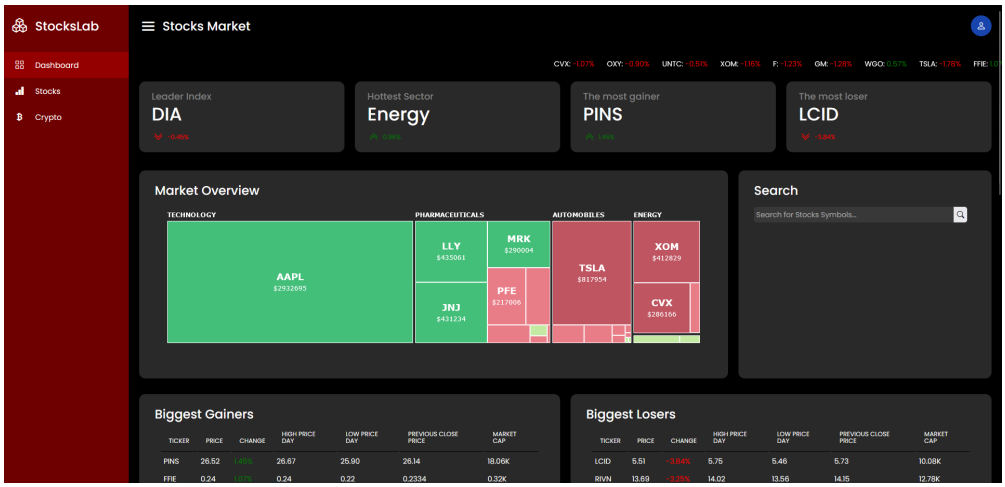

Figure 2: Homepage

Here we can see the various subviews that compose the homepage. All the data needed to build them are retrieved through API calls, and then presented in a nice way. In particular, the treemap is built by retrieving relevant data and then displaying them using the library `AG Charts`. What is happening under the hood here is basically: through javascript code, we perform a request to different endpoints of our application, that are configured to in turn perform an RPC call through `RabbitMQ`. The server then gets the result back from `NodeJS` and returns it to the frontend, so that the content can be dynamically loaded. Since this can take some time, we also added a loading animation, that is displayed while the data is being retrieved, as we can see from the following picture:

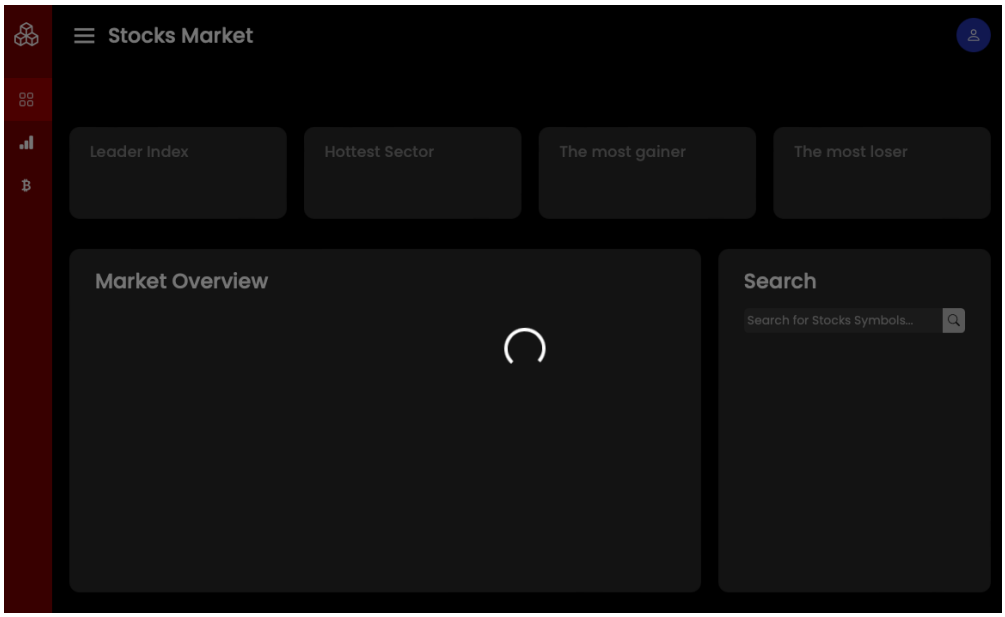

Figure 3: Homepage Loading

Let's now try to search for a specific stock through the search bar:

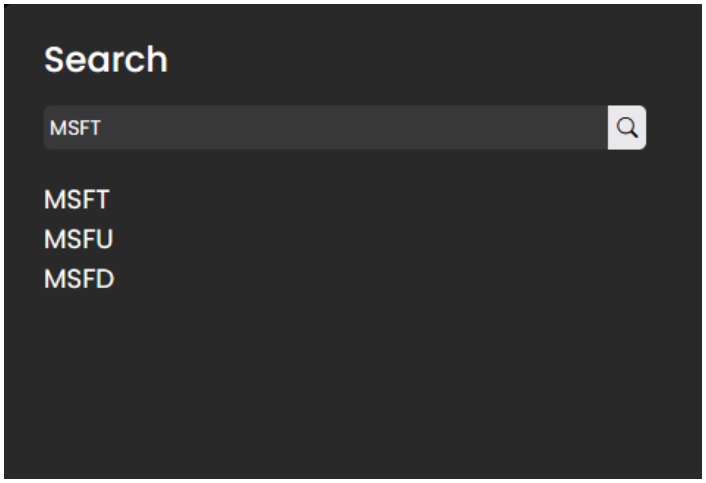

Figure 4: Search

As we can see, when typing a ticker and pressing the search button, we get a list of results. We can then press one of them to be redirected to this page:

Figure 5: Stock Page

In this page we have an overview of the symbol we searched for. The url is of the form `stock.html?symbol=<ticker>&time_window=<t_window>`. These query parameters are used by the server for deciding which stock it should display information for, and what time window to take for the candlestick graph. This graph is implemented using `apexcharts`, and can also display additional information when we pass over it with the mouse:

Figure 6: Candlestick Graph

We can also navigate to the Cryptocurrencies page:

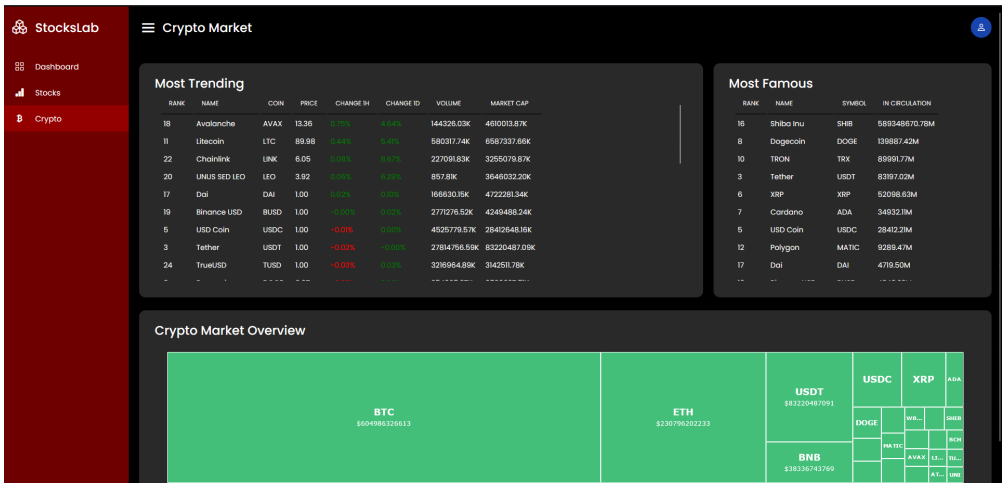

Figure 7: Cryptocurrencies Page

This page works similarly to the Stocks page, and the subviews are still constructed using API calls. Let's now explore the profile functionality. First of all, we need to register a new user. We do this by selecting "Signup" in the menu that opens when passing with the mouse over the user icon. We are then redirected to this page:

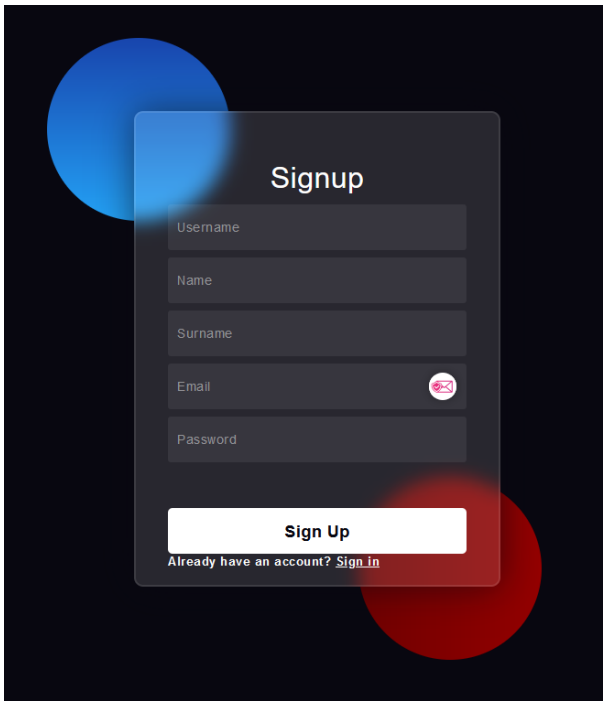

Figure 8: Signup Page

After filling the form, we can press the "Sign Up" button, and we are redirected to the login page:

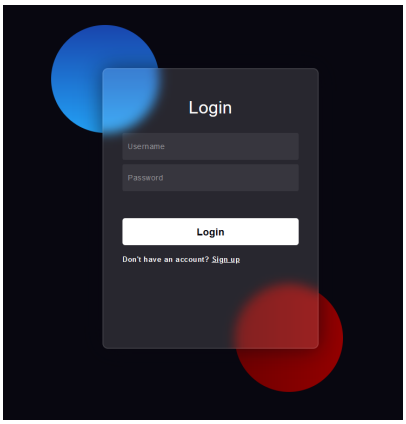

Figure 9: Login Page

After the login, the profile page becomes accessible:

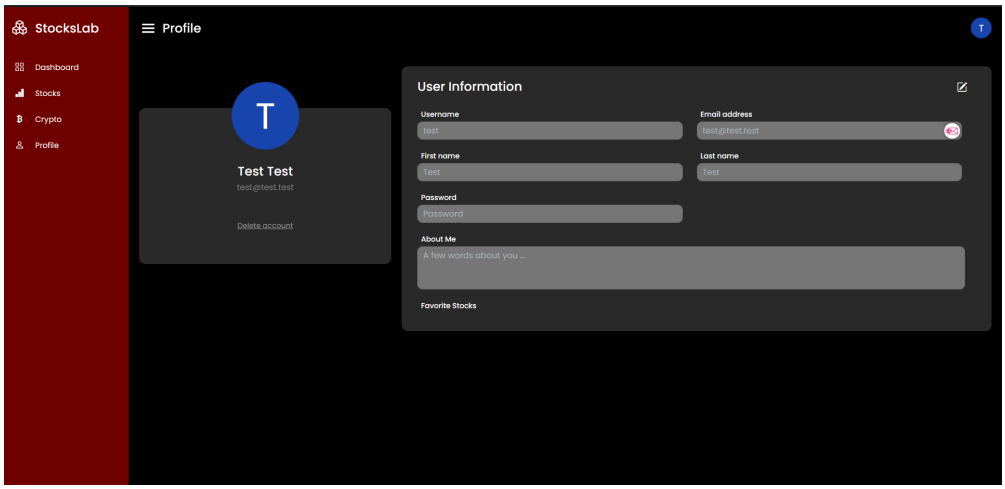

Figure 10: Profile Page

Here we can see the user information, and also the list of saved stocks. First, let's test the edit functionality. By default, the user information is not editable. We can change this by pressing the "Edit" button in the top right. We can then try to edit a field, for example the first name, and then press the button "Send". As we can see from the following picture, the update is successful:

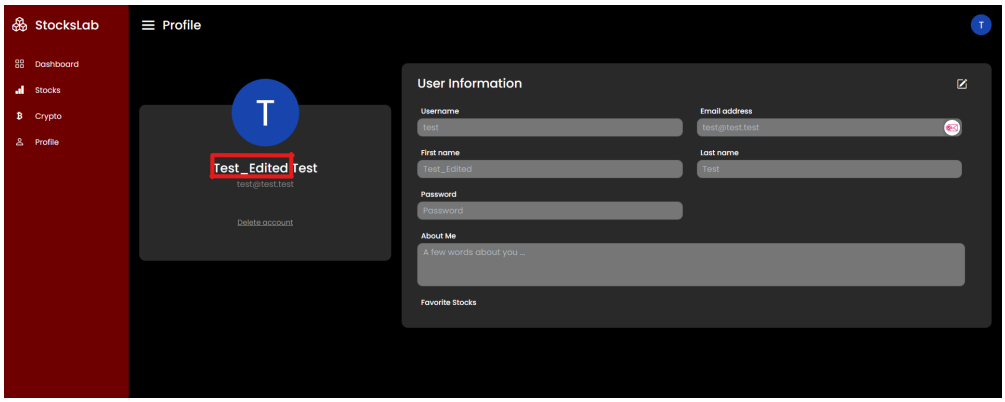

Figure 11: Profile Edit

There is also the possibility to save stocks as favorites. For doing this, we can navigate to the "Stock" page for a specific symbol while being logged in, and we click the newly created button "Add to favorites":

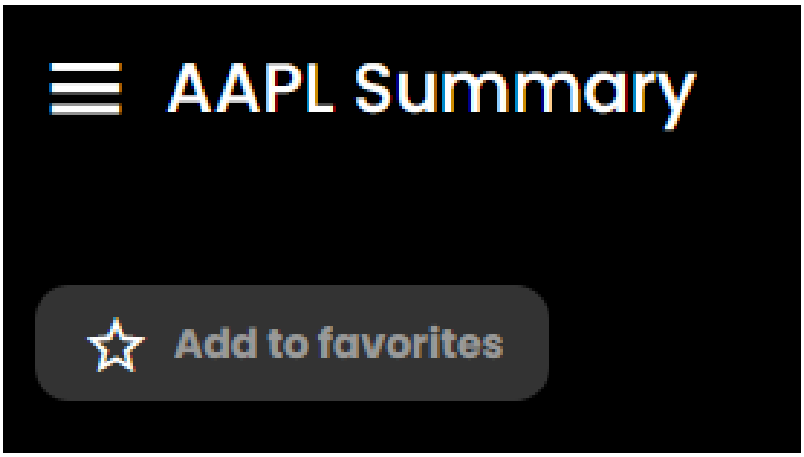

Figure 12: Add Stock to Favorites

We can then navigate to the profile page, and we can see that the stock has been added to the list of favorites:

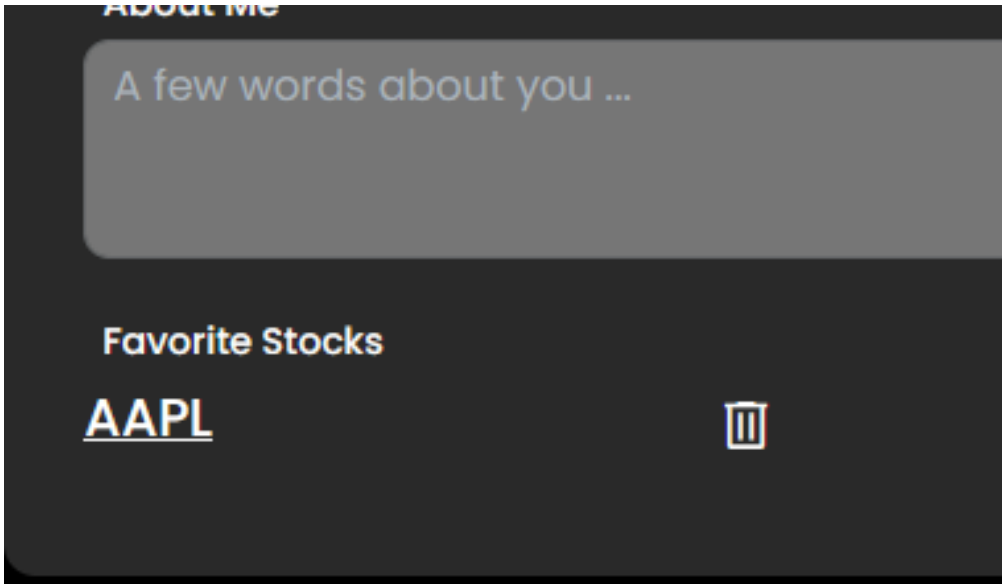

Figure 13: Newly Added Favorite

Finally, we can also delete the account. We can do this by pressing the "Delete Account" button. We are then asked to confirm the decision, and upon pressing "Ok" we get redirected to the homepage.

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
```