1- Problem Definition

We would like you to create a java based backend application using REST. It should contain the following endpoints:

- GET /api/stocks (get a list of stocks)
- GET /api/stocks/1 (get one stock from the list)
- PUT /api/stocks/1 (update the price of a single stock)
    - Note: this Endpoint is implemented on “PUT /api/stocks/1/price”, since we are only updating the price.
- POST /api/stocks (create a stock)

2- Execution

This program is written using spring boot and can be run on any JVM (1.8+). It exposes some REST APIs to read and write Stocks data. It also provides a web page in which latest information of all stocks can be seen in a grid. All the changes made to stocks are pushed to this page.

For the services which manipulate the data (CREATE/UPDATE), a token based authentication is employed. Therefore, in order to call those services, you must login before and get a token. Then the token must be sent as a header parameter in every request. It is assumed that the GET requests (and also the UI) do not require any authentication.

2-1- REST API

The API documentation is available online here on postman.

As an example the Login API is describes as following:

Endpoint: host/api/v1/login

Credentials must be sent as a JSON body. Please don’t forget to set the media type to application/json. you can login using the default account:

```
{
    "username": "admin",
    "password": "admin"
}
```

After successful authentication it returns a token like this:

```
{
    "token": "e26450c1-5673-4c81-8e61-50c8d7d772f2"
}
```

2-2- User Interface

The user interface can be found on the root address of the web application. Here is a screenshot:

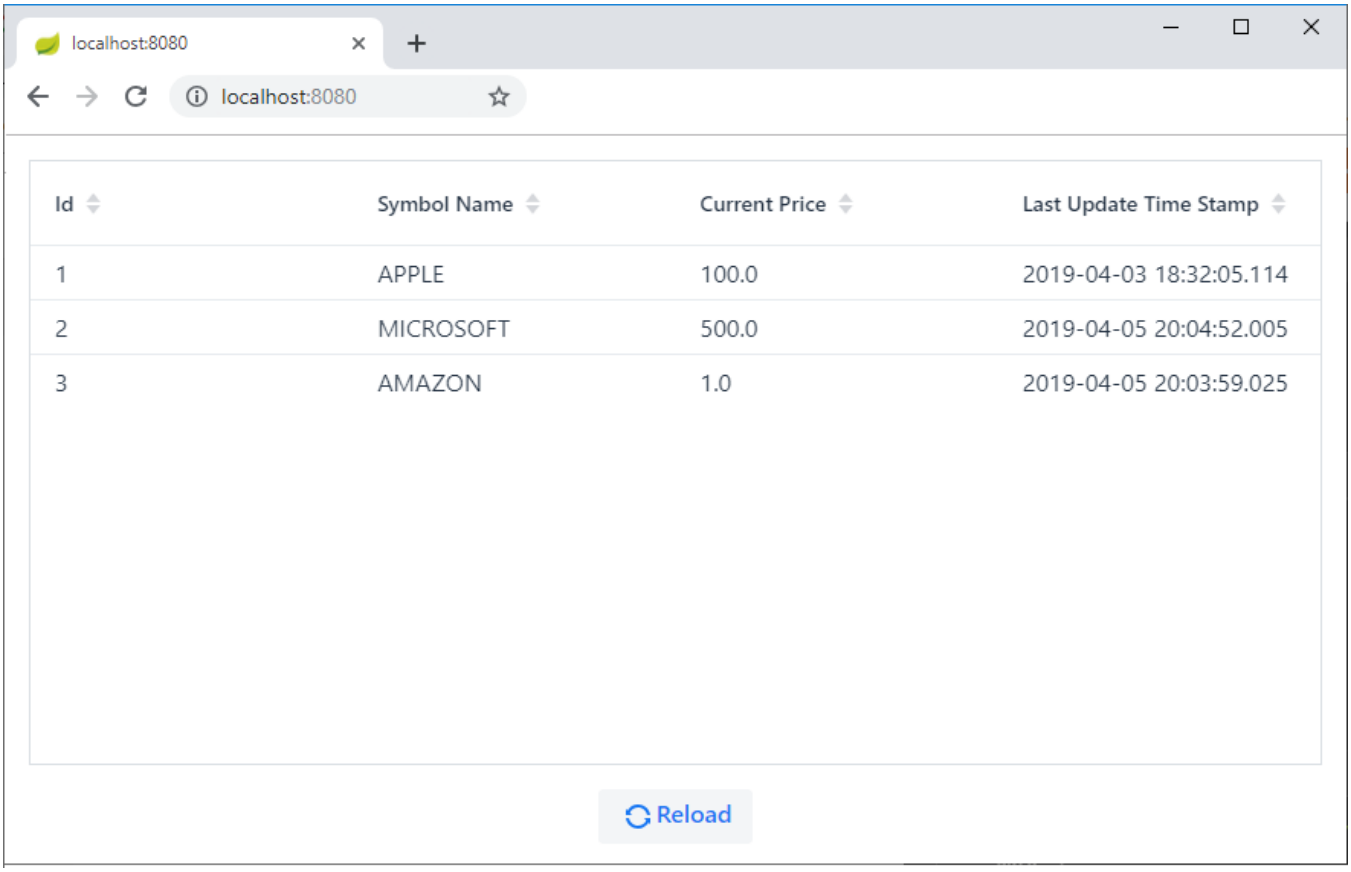

Figure 1 - User Interface

3- Architecture

The below figure demonstrates the overall architecture of this application. Users send their request via REST API to the system. They also can view the list of Stocks on a Web UI. The Stock Service application talks to an in memory database for persisting the data.

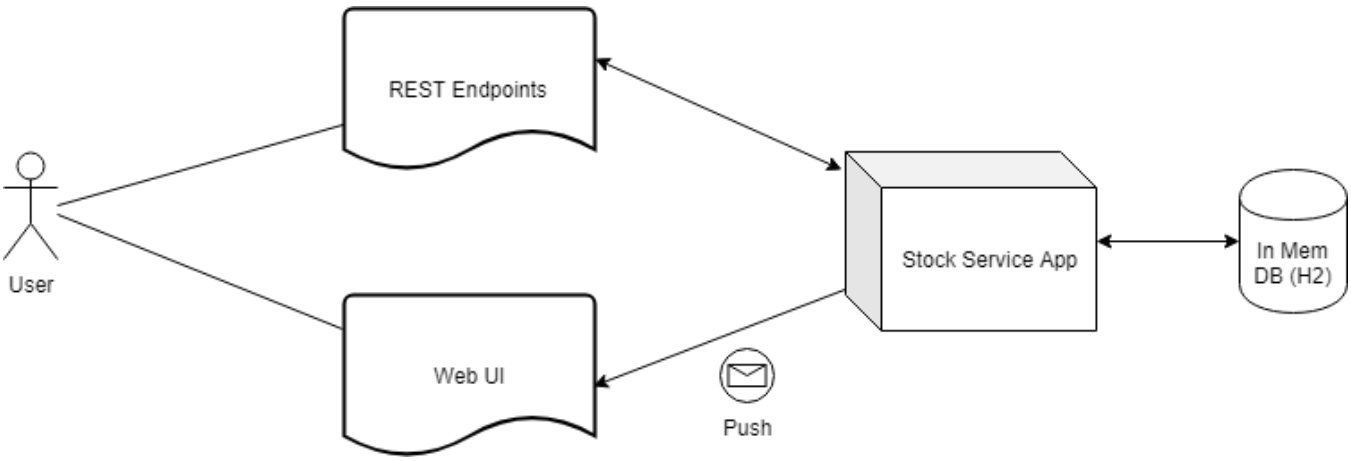

Figure 2 – Architecture

4- Design

In this section design decisions and the main purpose of core objects is explained.

4-1- Domain Objects

Domain objects are described in this section.

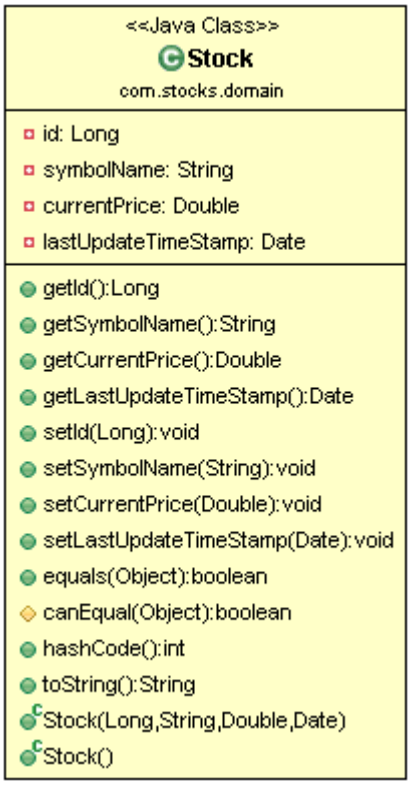

Figure 3- Domain objects

4-1-1- Stock

This is a knowing class to keep the desired information of a stock.

4-2- Services

The services and repositories can be seen in the below figure:

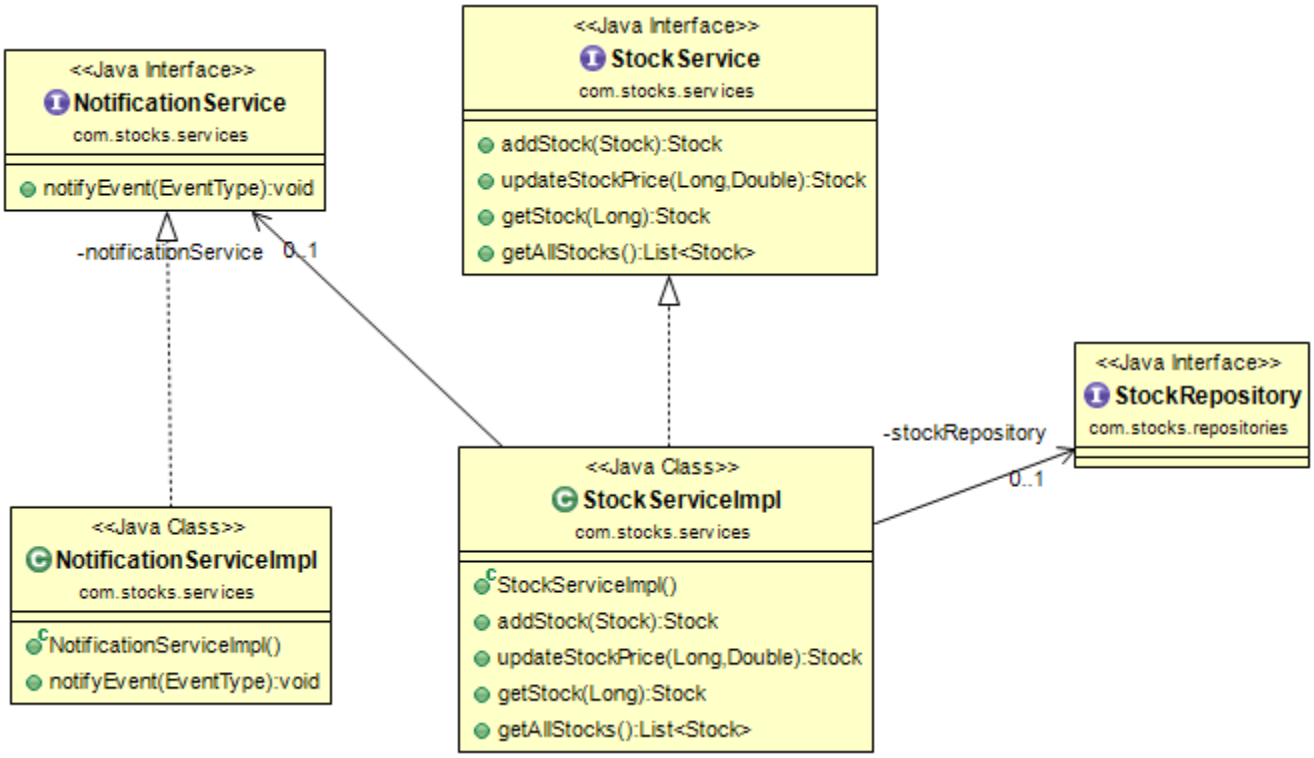

Figure 4 - Services and Repositories

4-2-1- StockRepository

Provides the CRUD operation on DB for Stock.

4-2-2- StockService

Provides a definition of required services which must be available for Stocks.

4-2-3- StockServiceImpl

Provides an implementation of services related to Stocks.

4-2-4- NotificationService

Provides a definition of required services for notification actions. Whenever any type of notification should be sent in the app, this class is responsible of that part.

4-2-5- NotificationServiceImpl

Provides an implementation of NotificationService.

4-3- Controller Layer

Controller Layer which is responsible for handling REST requests is explained in this section.

As part of the MVC pattern, the Rest Controller is designed to receive user requests, talk to the core and return a response. It currently offers the required methods to read and write the stocks. In all services if the operation is successful a HttpStatus of OK/CREATED is returned along with a JSON response.

In case of exception, a separate ExceptionHandlerAdvice is designed to keep the REST Controller codes clean (thanks to SpringBoot). ExceptionHandlerAdvice returns a proper JSON response and HttpStatus according to the occurred incident.

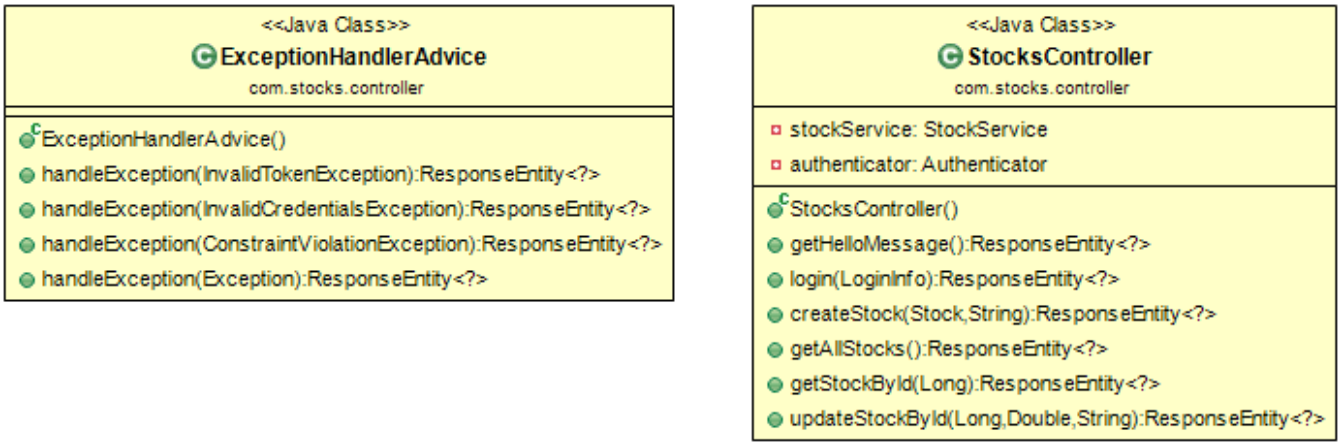

Figure 5- Controller Layer

4-4- Authentication

In this application a simple, from scratch token based authentication approach is taken. AuthenticatorImpl class maintains a hash map of logged in users and their corresponding tokens. When a new user logs in with correct credentials the maps get updated. In all the REST calls isTokenValid is called to check user validity.

Please note that this approach is chosen here only for simplicity. More sophisticated schemes such as JWT, Basic Auth (not really a suitable approach for production) can be employed for this purpose. Spring security has a lot to offer on this topic.

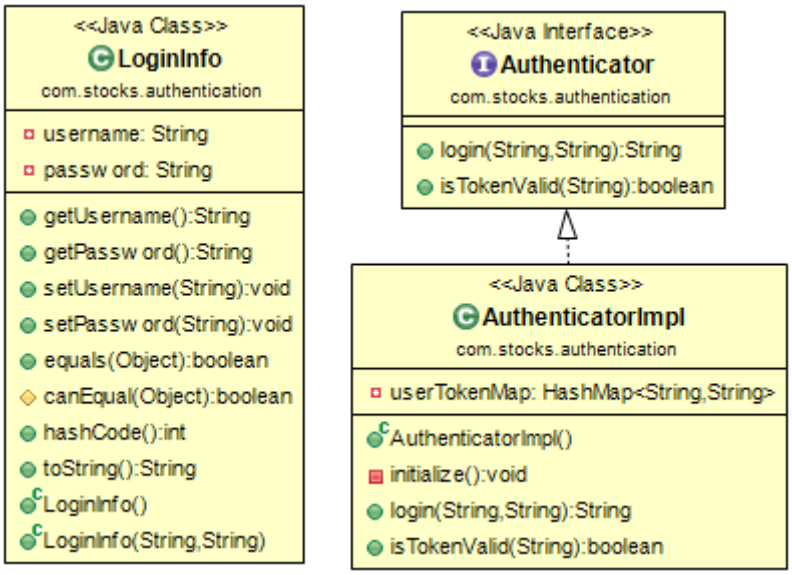

Figure 6- Authentication

4-5- Exceptions

The custom exception classes keep only a message regarding the incident.

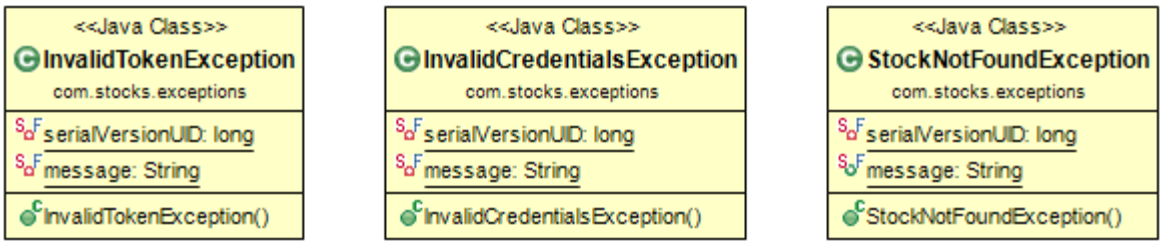

Figure 7-Custom Exception Classes

4-6- User Interface

The user interface layer is implemented in the web platform using Vaadin framework. It follows the MVP pattern which every View is separated from the logic by talking to a Presenter. As the name says, PushNotficationBroadcaster class is used to take care of pushing the events to the UI.

Here the presenter classes talk to the core logic and update the UI. As an alternative solution we could connect the UI layer to our existing controllers (Employing MVC instead of MVP).

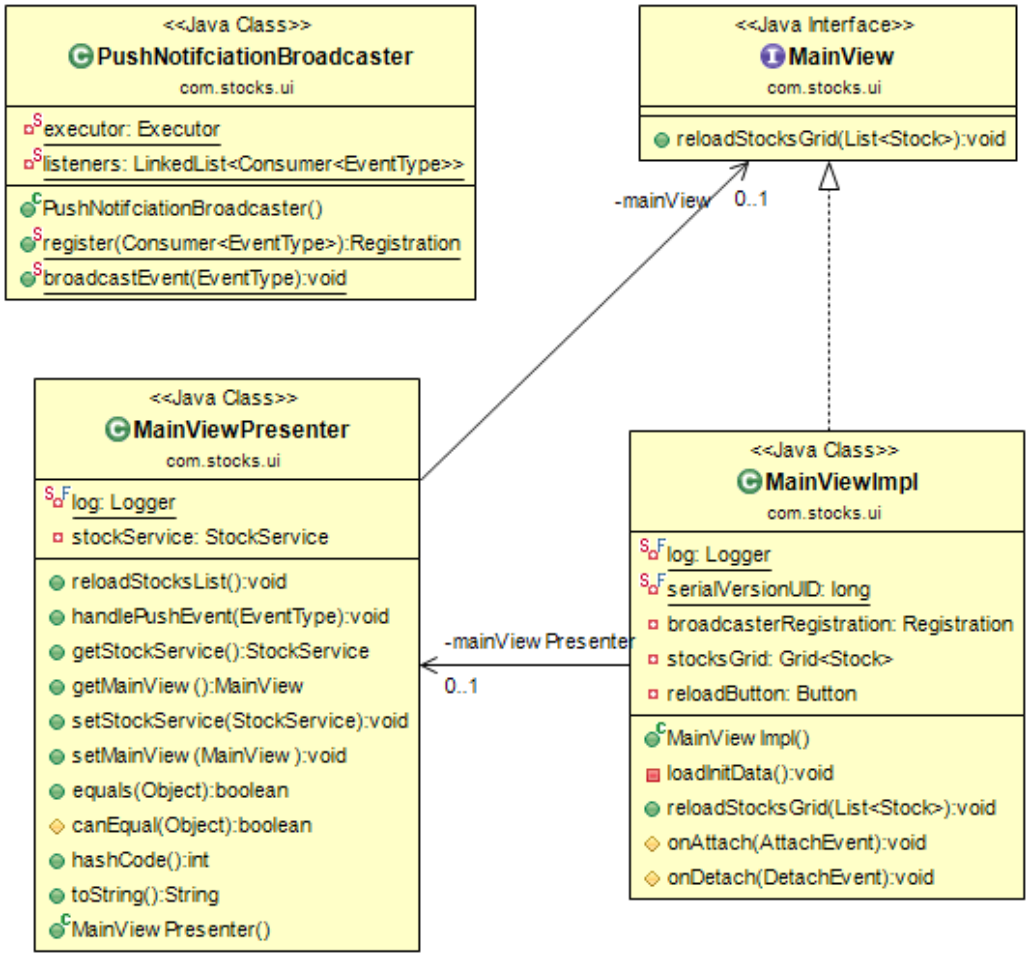

Figure 8 - User Interface Layer

5- Tests

Unit test and integration tests are written in multiple classes. The tests are designed for both core and UI layer. The UI tests make sure about the correctness of Presenters and Pusher. The view layer can be tested with other tools such as Selenium. For simplicity we skipped this part. The test classes can be seen in the below image.

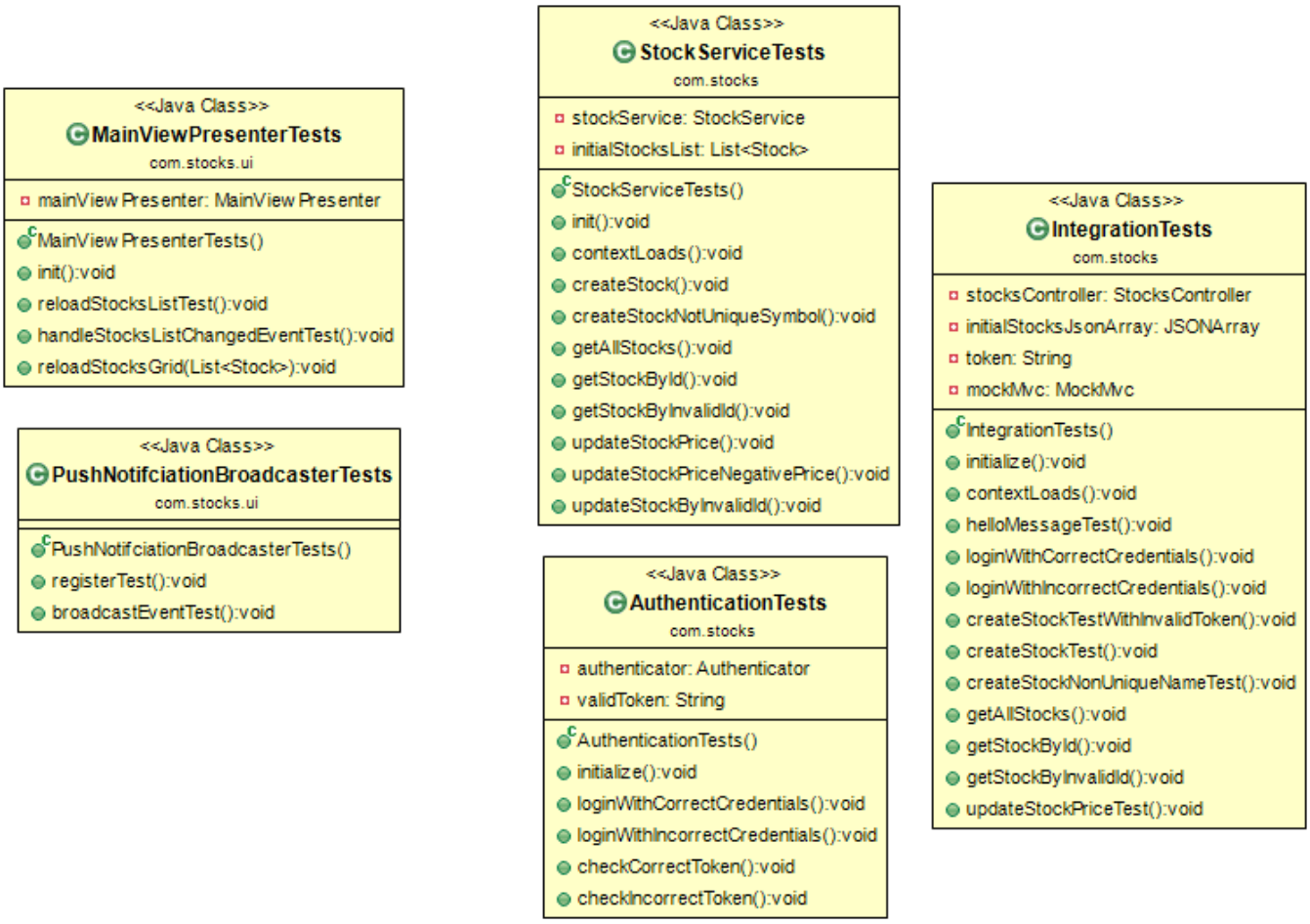

Figure 9- Test Classes

5-1- Coverage

Below figure demonstrates the code coverage for the tests. Note that domain classes indicate low percent because of auto generated getters and setters. Also the views are not tested here, as we described before.

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
StockService	69.2 %	1,343	599	1,942
src/test/java	95.3 %	826	41	867
com.stocks.ui	100.0 %	54	0	54
MainViewPresenterTests.java	100.0 %	27	0	27
PushNotifciationBroadcasterTests.java	100.0 %	27	0	27
com.stocks	95.0 %	772	41	813
IntegrationTests.java	100.0 %	552	0	552
StockServiceTests.java	86.9 %	186	28	214
AuthenticationTests.java	72.3 %	34	13	47
src/main/java	48.1 %	517	558	1,075
com.stocks.controller	100.0 %	191	0	191
com.stocks.exceptions	100.0 %	12	0	12
com.stocks.services	100.0 %	104	0	104
com.stocks.authentication	39.1 %	75	117	192
com.stocks	37.5 %	3	5	8
com.stocks.ui	30.7 %	105	237	342
com.stocks.domain	11.9 %	27	199	226

Figure 10 - Coverage