HSBC – RoboAdvisor Portfolio Rebalancing Service

Test Plan

CPSC 319 – Team Formation

Justin Tao, Yao Liu, Yuntian Wan, Scott Wang

Jingwei Zhang, Lauren Bentley, Sam McConnell

April, 2019

Version 1.1

**Summary**

A mix of manual and automated testing will be employed to ensure quality of our RoboAdvisor API fund rebalancing endpoints. Testing will include manual verification and testing on our RoboAdvisor UI interacting with the deployed backend to ensure main user functionality is nominal and operational. In addition, automated unit and integration tests will be run as part of the deployment pipeline for the API endpoint. These automated tests are more rigorous and test for inputs outside of allowable user-input from simple interaction through the UI. These ensure our API is robust and able to appropriately handle non-complying inputs. Our automated tests are using the TestNG Java testing framework in addition to Mockito to create automated unit and integration tests. For a successful API deployment, a full set of unit and integration tests including any regression tests must be completed. Following successful API deployment, manual verification will be completely on the UI. Once all tests are successfully completed, the API is considered production ready and the build can be promoted to the production servers.

Currently development of tests has been split between 3 developers, with one working through the integration test and the others working through unit tests for the controller and all of our services.

Test data used in our database are provided by HSBC. These sampled customers possess a valid portfolio and funds which can interact with both HSBC’s fund system in addition to our Fund Rebalancing service. Additional test data for local development of our API can be directly created through our APIs or database.

**Functional Test Plan**

The approach of our functional test plan will include a suite of Unit tests, Integration tests, and manual UI tests. In our case, a component or unit will represent the discrete requirements that are identified in our MVP. Due to the nature of our project, there is a clear separation of backend (API endpoints) and frontend (UI) components and thus we will employ a similar test style for each group of components below.

**API Endpoints**

**Components:**

1. getPortfolioPreference
2. createPortfolioPreference
3. setPortfolioAllocation
4. setPortfolioDeviation
5. createRecommendation
6. modifyRecommendation
7. executeRecommendation
8. getRecommendedTransactions

**Test Technique:** All endpoints (1-8 in the list above) are to use black box testing in addition to being automated through the use of the TestNG (v6.8) and Mockito (v2.23.4) frameworks.

**Approach:**

* Unit Testing

**Assumption:**

* We mock the behaviour of the service classes since the controller functions depends on the methods of service class.
* We only test the cases that could happen in real situation based on the user flow. And assumption for each endpoint is also listed.
* The JSON request from UI is correct and valid in terms of formats and values.

**Risks:**

* Test cases may not be comprehensive enough to capture all edge cases

**Number of scripts:**

* At least 7 happy path testing
* At least 21 non-happy path

**Common tests for every endpoints**

* Test of standard input
* Test for a missing header
* Test for an invalid customer ID

**Test Scripts:** Appendix 1.1

**Test Repo Link:** <https://gitlab.com/cpsc319-2018w2/hsbc/formation/roboadvisor-api/tree/master/src/test/java/com/hsbc/roboadvisor/controller>

**Backend Services**

**Components:**

1. PortfolioRepositoryService
2. RecommendationRepositoryService
3. CreateRecommendation

**Test Technique:** All testing under the service class is to use white box testing automated through the use of the TestNG (v6.8) and Mockito (v2.23.4) frameworks

**Approach:**

* Unit Testing

**Assumption:**

* FundSystemService provided by HSBC is functional
* We mock the behaviours of the CRUD repository under the spring data repository library
* We assume the CRUD repository class behaves correctly

**Risks:**

* Changes to the CRUD repository could result in service unit tests that mock repository behaviour incorrectly

**Number of anticipated scripts:**

* At least 6 happy path testing
* At least 9 non-happy path

**Common tests for every endpoints**

* Test of standard input

**Test Scripts:** Appendix 1.2

**Frontend UI**

**Components:**

The UI components and workflows should be tested manually. In principle, the UI’s displayed data should be fetched or calculated using the up-to-date data in the database. Also, all the workflows described in the MVP should be made possible while invalid http posts should be prevented.

**Test Technique:**

* All testings under the frontend UI tests will be done though manual black box testing

**Approach:**

* User Acceptance Testing to ensure functionality of final MVP.

**Assumption:**

* Backend API endpoints are reachable
* Backend Services are functional

**Risks:**

* Performing of manual test script may not hit every edge case

**Number of anticipated scripts:**

* 6 happy paths testing
* 3 non-happy paths
  + Login page, Set Allocation, and Deviation

**Test Scripts:** Appendix 1.3

**Integration Tests**

**Components:**

Integration testing represents a step where single modules programs are combined and tested together. For the HSBC portfolio rebalancing API, it is essential for our team to test how the portfolio controller interacts with the database and the funds system controller. Therefore, the goal is to test each endpoint in the portfolio that persists changes to either the MySQL database backend or the funds system controller.

**Test Technique:**

* A functional black box testing technique will be used. One automated testing script will be used to test the API.

**Approach:**

* Integration testing will involve executing the API for every use case with the intent of exposing errors

**Risks:**

* The biggest potential risks with SIT is lack of access to HSBC production systems and mocking out the funds system controller
* Another potential risk includes being able to localizing defects to the correct part if a test is failing and unit tests are passing.

**Number of anticipated scripts:**

* 1 happy path testing including integration of multiple systems

**Test Scripts:** Appendix 1.4

**Non Functional Test Plan**

As identified in the Business Requirement Documentation, our focus for non-functional testing is primarily in availability, reliability, and security.

**Availability Testing**

**Approach:** To ensure constant availability, a GitLab cron job will be created to exploit GitLab’s built in CI/CD. The job will complete a simple get request. If the request succeeds, the job will succeed. We are able to determine if availability is disrupted by examining the job for failures.

**Assumption:** GitLab CI/CD is functional and sufficient pipeline minutes remain.

**Number of anticipated scripts:** 1

**Test Script:** Appendix 1.5.1

**Reliability Testing**

**Approach:** To ensure reliability, manual testing will be employed to simulate multi-user experience. Multiple testers will each emulate a user and interact with the system, creating portfolio preferences and executing rebalancing recommendations. Testers will follow a typical user flow.

**Assumption:** Testers are available for testing application.

**Number of anticipated scripts:** 1

**Test Script:** Appendix 1.3 UI Test Script

**Security Testing, Code Quality and Testing Coverage**

**Approach:** Security testing is accomplished through the use of SonarQube Community Edition. SonarQube will scan our source code and determine for us potential bugs, vulnerabilities, code smells, coverage, and duplications. A report will be generated outlining all the discovered issues. SonarQube periodically to ensure that the source code is sufficiently secure and up-to-standards.

**Assumption:** SonarQube is able to sufficiently capture potential weaknesses and issues in the source code.

**Number of anticipated scripts:** 1

**Test Script:** Appendix 1.5.2

**Test Scripts**

Our approach to for formulating the test scripts for the above functional and non-functional testing is with a principle of comprehensiveness and robustness. Regarding functional testing, test scripts were generated with the aim to expose every end-point and test for all major edge-cases in both the backend and frontend. For non-functional testing, the main goal is to ensure availability, reliability, and security with our test scripts.

Automated unit and integration tests are run as part of the deployment pipeline. The pipeline is configured as part of GitLab’s continuous integration and deployment. The stages are described in the pipeline.gitlab-ci.yml file which gitlab picks up and executes on their kubernetes runners.

The pipeline.gitlab-ci.yml file can be found in the Appendix 2.1.

**Security Testing and Test Data**

HSBC RoboAdvisor security will be represented by the used of the x-custid header required for all API calls. Authentication and authorization will be implemented in full by HSBC upon source code handover. No tests will be required for verifying the the authorization. However, unit tests do assert for the valid existence of the x-custid header.

Test data has been provided by HSBC and there contains no sensitive data to mask. Any additional simulated customers, portfolios, and funds also do not require any data masking in development and testing purposes. Test data for portfolio preferences can be created manually through interactions with our API or directly through our database. Test data for customers, portfolios, and funds will need to be created and provided along with the Fund System by HSBC. The fund system is assumed to be functional and will not require any testing provided by us.

**Regression Testing**

Regression tests will be composed of unit, integration, as well as UI user acceptance testing. With subsequent major releases, regression testing will first be conducted through UI user acceptance tests to ensure backwards compatibility and identify potential regressions. Unit tests and integration tests will also be run automatically to ensure new code is non-breaking for any new commits. Regression testing for portfolio rebalancing will ensure both categorical and fund-based rebalancing executes properly.

**Test Scripts to Include:** Appendix 1.1, 1.3, 1.4

**Appendix**

**1. Test Scripts**

**1.1 API Unit Tests**

Below are the unit tests for every endpoints:

1. getPortfolioPreference endpoint
2. Test for a preference ID that doesn’t exist
   1. Test for an alpha preference ID
3. createPortfolioPreference endpoint
   1. Test with an invalid allocation list
   2. Test with a preference ID that doesn’t exist
4. setPortfolioAllocation endpoint
   1. Test with an invalid allocation list
   2. Test with a preference ID that doesn’t exist
5. setPortfolioDeviation endpoint
   1. Test with an invalid deviation
   2. Test with a preference ID that doesn’t exist
6. createRecommendation endpoint
   1. Test with a preference ID that doesn’t exist
   2. Test with an existing preference that doesn’t have corresponding portfolio
   3. Test with an invalid recommendation ID
7. modifyRecommendation endpoint
   1. Test with a preference ID that doesn’t exist.
   2. Test with a recommendation ID that doesn’t exist.
   3. Test with some requests that is impossible to realize (e.g. user sells more units than what he owns)
8. executeRecommendation endpoint
   1. Test with a recommendation ID that doesn’t exist.
   2. Test with an existing preference that doesn’t have corresponding portfolio

**1.2 Service Unit Tests**

1. PortfolioRepositoryService

This service will provide tests to validate our ability to persist preferences, in addition to modifying preferences given their portfolio ID

* 1. Test saving a standard portfolio preference
  2. Test to confirm that the service does not save portfolio preferences with allocations that do not sum to 100%
  3. Test updating a portfolio preference with a valid deviation
  4. Test updating a portfolio preference with a valid allocation
  5. Test updating a portfolio preference when the portfolioID doesn’t have an associated portfolio in the repository

1. RecommendationRepositoryService

We only test the saveRecommendation method in this class since other methods only implement methods from the CRUDrepository class.

The purpose of the saveRecommendation class is to return a recommendation based on the user’s portfolio preference.

* 1. Test the cases where the calculated recommendation is invalid (e.g. The balance after executing the recommendation is huge)
  2. Test the cases where the portfolio preference doesn’t have enough information to create the recommendation.
  3. Test the cases where the fund list of the user is different with the funds of the portfolio
  4. Test the cases where the inputs are invalid.

**1.3 UI Test Scripts**

*The test conditions and expected outputs are outlined below.*

1. Login (only valid user ids can login, otherwise be prevented)
   1. Login using valid ids. It should navigate to the dashboard page displaying “Welcome back, [id]!”
   2. Login using invalid ids. Should loading incorrect loading page with logout button.
2. Dashboard data should be displayed and calculated correctly
   1. The list of portfolio ids should match the corresponding portfolios stored in the database for the given user.
   2. The total assets should match the sum of the values of the given user’s funds.
3. Portfolio page data should be displayed and calculated correctly
   1. Manually calculate each fund’s proportion. They should match the proportions in the UI.
   2. Compare the deviation and funds’ target values in the browser and the values in the database. They should be equal.
4. Set deviation should work properly
   1. Select a portfolio, click “Update Allocation” button, change the allowed deviation, click “SAVE” button. The modified result should be preserved if between 0 and 5.
   2. Repeat 4.a, and then refresh the page, the modified result should be preserved.
   3. Repeat 4.a, and then click “Back” button and select the same portfolio, the modified result should be preserved.
   4. Select the same portfolio again, “Update Allocation” button, change the allowed deviation to 6, click “SAVE” button. An error message should pop-up at the bottom indicating deviation must be between 0 and 5.
5. Set each fund’s target proportion should work properly
   1. Select a portfolio, click “Update Allocation” button, change the target value for each funds (Add up to 100%), click “SAVE”. The modified result should be preserved, and the donut chart should be updated correspondingly.
   2. A notification should pop up at the Top indicating that a portfolio is due for rebalancing.
   3. Select a portfolio, click “Update Allocation”, change the target value for each funds (Not add up to 100%), click “SAVE” button. A warning should pop up saying “Target does not add up to 100”, and the save action should be prevented.
   4. Repeat 5.b, then click “Cancel” button, the target values should change back to the previous ones.
   5. Repeat 5.a, then refresh the page. The modified result should be preserved.
6. Rebalance should work properly
   1. Select a portfolio, click “Rebalance” button. Buy/sell Recommendations should appear, as well as the “Modify” and “Execute” buttons.
   2. Repeat 6.a, then check all the funds’ balance, target and actual proportion should match the values stored in the database.
   3. Repeat 6.a, then click “Execute” button. Each funds’ current value should be updated according to the recommendation.
   4. Repeat 6.c, then refresh the page, the modified values should be preserved.
7. Modify rebalance should work properly
   1. Repeat step 5a)
   2. Click “Rebalance” button. Buy/sell Recommendations should appear, as well as the “Modify” and “Execute” buttons
   3. Click the “Modify” button. Each fund should allow you to set whether you wish to buy or sell. In addition, the numbers of units should be allowed to be set by the user.
   4. Toggle a “Buy” button to a “Sell” button and vice versa. Click “Save Modification”. The modified recommendation action should be persisted.
   5. Refresh the page, the modified recommendation action should be persisted.
   6. Repeat 7b)
   7. Per a fund input a number into units greater than the number of units owned of that fund. Make sure recommendation button is set to sell. An error message should pop up indicating you cannot sell more funds than you own.
   8. Change the Fund’s recommendation to ‘Buy’ and click “Save Modification”. A pop-up message indicating the recommendation modify has been successfully saved. The modified recommendation values should be persisted for units.
   9. Refresh the page, the modified recommendation action should be persisted.

**1.4 Integration Test Script**

1. Create a portfolio preference
2. Get an existing portfolio preference
3. Not able to get an invalid portfolio preference id
4. Change the allocation for an existing fund in a portfolio preference
5. Ensure change in allocation persisted to portfolio preference
6. Not able to put an allocation where the portfolio preference total does not sum to 100% for all funds
7. Update a deviation for a portfolio
8. Ensure change in allocation persisted to portfolio preference
9. Create a rebalance recommendation for a portfolio
10. Try to create a rebalance recommendation for a portfolio that does not exist in the funds system, but that a portfolio preference exists
11. Execute a rebalance recommendation
    1. Ensure it updates the portfolio in fund-system-controller for specific funds affected.
12. Update the transactions in the rebalance recommendation
    1. Ensure valid modification

**1.5 Non Functional Testing**

1. Availability Test Script: <https://gitlab.com/cpsc319-2018w2/hsbc/formation/roboadvisor-cron/blob/ef0c57e1d9af0fa57910b468ae13e02cad5fb48a/test.sh>
2. Security Test Script: mvn sonar:sonar -Dsonar.host.url=http://localhost:9000 -Dsonar.login=<sonar token>

**2. Other**

**2.1 The pipeline.gitlab-ci.yml file contents**

|  |
| --- |
| image: jamesdbloom/docker-java8-maven:latest  stages:  - compile  - unit\_test  - integration\_test  - deploy  build:  stage: compile  script:  - mvn clean compile  only:  - merge\_requests  - master   unit\_test:  stage: unit\_test  script:  - mvn clean test  only:  - merge\_requests  - master   integration\_test:  stage: integration\_test  script:   - mvn clean -DskipSurefire integration-test  only:  - merge\_requests  - master  deployment:  stage: deploy  script:  - wget https://dl.google.com/dl/cloudsdk/channels/rapid/downloads/google-cloud-sdk-218.0.0-linux-x86.tar.gz  - tar -xzf google-cloud-sdk-218.0.0-linux-x86.tar.gz  - ./google-cloud-sdk/install.sh  - echo $DEPLOY\_KEY\_FILE\_INTEGRATION > /tmp/$CI\_PIPELINE\_ID.json  - ./google-cloud-sdk/bin/gcloud auth activate-service-account --key-file /tmp/$CI\_PIPELINE\_ID.json  - source /builds/cpsc319-2018w2/hsbc/formation/roboadvisor-api/google-cloud-sdk/path.bash.inc  - ./google-cloud-sdk/bin/gcloud config set project $PROJECT\_ID  - mvn appengine:deploy -Pcloud  only:  - master |