Application Test Results

FDM06

Contents

1. Functional Tests

Below are a set of tables that represent the pass or fail status of various test cases in relation to their associated functional requirement / user story. Refer to FDM05 “Test Plan” documentation for functional test case definitions and FDM05 “FRS” for a list of functional requirements.

1.1. Authentication

Test Case	Related Functional Requirement(s)	Status
1.01. Account Creation	UA01, UA11	PASS
1.02. Account Activation	UA02	PASS
1.03. Log In – Non-Activated Account	UA03	PASS
1.04. Log In – Activated Account	UA03	PASS
1.05. Log Out	UA04	PASS
1.06. Change Email Address	UA09	PASS
1.07. Log In – Super User Account	UA10	PASS
1.08. Request Data Removal	UA16	PASS
1.09. Request Data Removal	UA17	PASS

1.2. Input Validation

Test Case	Related Functional Requirement(s)	Status
2.01. Registration – Invalid Name Format	IV01	PASS
2.02. Registration – Invalid Email Format	IV01	PASS
2.03. Registration – Invalid Password Format	IV01	PASS
2.04. Registration – Invalid Username Format	IV01	PASS
2.05. Registration – Reserved Email	IV01	PASS
2.06. Registration – Reserved Username	IV01	PASS
2.07. Log In – Invalid Username	IV02	PASS
2.08. Log In – Incorrect Password	IV02	PASS
2.09. Create / Edit Company – Invalid Company Name Format	IV03	PASS
2.10. Create / Edit Company – Invalid Company Symbol Format	IV03	PASS
2.11. Create / Edit Company – Reserved Company Name	IV03	PASS
2.12. Create / Edit Company – Reserved Company Symbol	IV03	PASS

1.3. Broker Account Verification

Test Case	Related Functional Requirement(s)	Status
3.01. Not Verified – Account Restrictions	UA06	PASS
3.02. Not Verified – Credit Card Details	UA06, UA07	PASS
3.03. Verification Process	UA06, UA08	PASS

1.4. Stock Trading

Test Case	Related Functional Requirement(s)	Status
4.01. View Share Pricing	ST01	PASS
4.02. Purchase Shares	ST02	PASS
4.03. Sell Shares	ST03	PASS
4.04. Share Sales Disabled	ST03	PASS
4.05. Transaction History	ST07	PASS
4.06. Admin - View Share Pricing	ST08	PASS
4.07. Edit Company – Company Name	ST09	PASS
4.08. Edit Company – Company Symbol	ST09	PASS
4.09. Create Company	ST10, ST11	PASS
4.10. Delete Company	ST12	PASS
4.11. Search Company Name (Full) - Existing	ST04	PASS
4.12. Search Company Name (Partial) - Existing	ST04	PASS
4.13. Search Company Name - Non-Existing	ST04	PASS
4.14. Search Company Symbol (Full) - Existing	ST04	PASS
4.15. Search Company Symbol (Partial) - Existing	ST04	PASS
4.16. Search Company Symbol - Non-Existing	ST04	PASS
4.17. Filter Company List – Gains	ST05	PASS
4.18. Filter Company List – Losses	ST05	PASS
4.19. Sort Company List – Symbol (Ascending)	ST06	PASS
4.20. Sort Company List – Symbol (Descending	ST06	PASS
4.21. Sort Company List – Price (Ascending)	ST06	PASS
4.22. Sort Company List – Price (Descending)	ST06	PASS
4.23. Sort Company List – Gains (Ascending)	ST06	PASS
4.24. Sort Company List – Gains (Descending)	ST06	PASS

1.5. User Account Activation / De-Activation

Test Case	Related Functional Requirement(s)	Status
5.01. De-Activate User Account (Admin)	UA12	PASS
5.02. De-Activate User Account (No Funds)	UA13	PASS
5.03. Request User Account Re-Activation	UA14	PASS
5.04. Re-Activate User Account	UA15	PASS

1.6. Reports

Test Case	Related Functional Requirement(s)	Status
6.01. Held Shares Report (.xml)	RT01	PASS
6.02. Held Shares Report (.csv)	RT01	PASS
6.03. Custom Stock Report (.xml)	RT02	PASS
6.04. Custom Stock Report (.csv)	RT02	PASS
6.05. Full Stock Report (.xml)	RT03	PASS
6.06. Full Stock Report (.csv)	RT03	PASS

2. Unit Tests

Below are a set of tables representing the unit tests for each corresponding service.

The raw test output can be found in files alongside this document in the format: “tests/testReport-{service}.json” or within the codebase of each service in the format “testReport.json”.

The following codebases of the architecture were out of scope for unit testing:

- api-gateway
- client
- microservice-util
- pricing-service
- service-registry

2.1. Test Suites

Service	Test Suite	Test Count	Pass Count
auth-service	tokenController	6	6
auth-service	authController	20	20
auth-service	activationController	32	30
auth-service	userController	24	24
broker-service	accountController	14	14
broker-service	transcationController	10	10
notification-service	notificationController	7	7
reporting-service	reportingController	21	21
stock-service	stockController	26	26
stock-service	validation	24	24
verification-service	verificationController	11	11

2.2. Service Test Count

Service	Test Count	Pass count
auth-service	82	82
broker-service	24	24
notification-service	7	7
reporting-service	21	21
stock-service	45	45
verification-service	11	11
Total	**190**	**190**

3. Component Tests

A set of component tests were created to ensure the validity of the architecture and connectivity in the server-side code. Therefore, the base set of tests were built to ensure each endpoint could receive all valid successful responses (HTTP 200 - 299), and error responses were out of scope.

Raw output from the component testing can be found alongside this document, within the file "tests/componentResults.csv". The detail within this output is slightly lacking based on the limited support to automatically write the results to a file from Postman. Custom scripts had to be written to create the file, so the format was kept barebones and could have contained much more detail (e.g., request and response headers and body).

A summary of the output has been outlined in a table below.

3.1. Component Test Count

Endpoint	**Test Count**	**Pass count**
/login	1	1
/register	1	1
/admin	1	1
/user	4	4
/activate	1	1
/deactivate	1	1
/reactivate	2	2
/requests	1	1
/account	2	2
/transactions	1	1
/buy	1	1
/sell	1	1
/report	4	4
/stock	1	1
/companies	4	4
/verify	2	2
Total	**28**	**28**

4. Performance Tests

Below is a table containing various metrics received from the performance testing of the final software solution. Refer to FDM06 “Performance Assessment Scheme” for specific test cases and the overall approach.

‘Mean response time’ is determined by performing many isolated calls to the correlated endpoint and calculating the average (mean) of these outcomes. It is worth noting that many of these calls are cached client-side, so the values shown are likely to be significantly reduced under some circumstances.

Request capacity was determined through sending a fluctuated number of requests up to a given count over a set period. The value shown is the number of successful responses received out of 1 million requests over a 30-minute period.

4.1 Performance Test Results

Endpoint	**Request Count Capacity (spanning 30 min period)**	**Mean Response Time (milliseconds)**
/login	376,650	17.96
/register	29,280	330.28
/admin	993,420	13.33
/user	272,280	24.33
/requests	12,240	50.00
/account	24,120	17.63
/transactions	532,290	26.40
/buy	11,340	83.14
/sell	14,580	83.80
/report	13,500	1359.41
/stock	41,500	78.68
/companies	515,280	13.68
/verify	656,310	18.77
Avg. (Mean)	**268,676**	**163**

5. Outcomes

All functionality-based tests (functional, unit, and component) passed, which was to be expected as meeting these requirements was paramount to project success. In addition, the nature of test-driven development ensures that unit and component tests are passed.

However, there are still various improvements that could have been made to these areas.

- Documentation and tracking of the functional tests are not verbose and could be improved by using spreadsheets, timestamps, and screen captures.
- Various services throughout the architecture were out of scope for unit testing in the hopes they would be covered by component tests. For a more robust testing process, it would be wise to unit test all code within the software solution.
- The component tests ensured a basic level of architectural connectivity with very sparse testing, and there were many more scenarios which could have been tested.
- Due to use of RESTful services, there was more code contained within the client than anticipated, including caching and state management. This functionality is integral to the user experience and should have been unit tested as the component tests also only covered the API.

The response time metrics were much better than anticipated, however they were performed in isolation and not under any significant load. The process could have been improved through targeting all endpoints simultaneously, combined with stress testing to simulate thousands of users, which would have provided a more reliable scenario and accurate results. Nevertheless, these outcomes are useful in detecting serious bottlenecks within the solution, such as user registration or report generation.

The metrics gathered for user capacity were also surprisingly high in some areas, whereby over half a million requests were handled successfully. However, some areas were significantly lower in which further investigation would be needed into the service calls, database transactions, and algorithms that are in use for these problem areas.

Overall, the bottleneck for many of these tests is the host system used at the time. There was no plan to distribute the microservice architecture across many machines, and a wise next step would be to do this using a cloud provider such as AWS or Azure DevOps. These platforms would allow the use of containerization through Docker and Kubernetes, which would have provided extra features in stability such as load balancing and auto scaling. This should have been a serious consideration to play to the strengths of the chosen design and its architecture.

The chosen design was also picked due to its benefits in providing a secure solution, by focusing on features such as NoSQL to limit database query injection and an API gateway pattern to reduce attack vectors onto the backend services. To better evaluate the effectiveness of these, it would have been wise to include a security / penetration testing phase into the test plan.