LO_4 notes

For 4.1, I covered multiple potential omissions or deficiencies in the testing and how they could be fixed.

For 4.2, I set the target for functional testing, structural testing, model-based testing, and other measurable attributes. For each testing method, I discussed how the target levels motivates the testing process adequately.

For 4.3, I compared the testing carried out and the target. I discussed and explained how well testing targets have been met by the testing motivating variations from target.

For 4.4, I discussed what could be done to achieve or exceed the target levels.

LO 4

4.1 Identifying gaps and omission in the testing process

Insufficient test coverage

Insufficient test coverage is a major issue in the testing process. Due to limited time, I didn't execute all the tests cases outlined in the test planning document. It may lead to defects and bugs going undetected. Certain functionalities or requirements were not thoroughly tested, leading to potential issues in the final product.

To remedy this, it would be necessary to allocate more time and resources for testing, and to prioritize the most important tests to ensure they are executed. Additionally, a review of the test planning document may be necessary to identify any unnecessary tests that could be removed, allowing for more time to focus on essential tests.

Limited test data

Conducting testing with a diverse range of data can help to uncover edge cases and identify any potential issues that may arise with specific data sets. This can include testing with large numbers of users, different types of orders, and various scenarios such as high traffic or peak usage times. However, I did not test the functionalities with many user/admin data.

To do better, more resources such as additional testers or increased testing time may be required. Overall, using more realistic and diverse test data can help to improve the confidence in the system's ability to handle real-world scenarios and identify any potential issues before it is released to production.

Insufficient test environment

I only did the tests on Ubuntu 22.04, if there were more time, I would test on other operating systems like windows 10 and IOS.

Lack of security testing

One limitation of testing is the lack of testing for security vulnerabilities. While the non-functional requirement of security is prioritized, there is not enough test cases or tools dedicated to address security risks in the system.

This could be improved by incorporating security testing tools, such as OWASP ZAP, and incorporating test cases specifically focused on identifying and addressing security vulnerabilities.

Lack of mutation testing

Mutation testing can be carried out manually or by using a mutation testing framework. The process would involve making small, deliberate changes to the code (known as mutants) and running the test suite to see if any of the tests fail. If a test fails, it means that the mutant has been detected and killed, indicating that the original code and test suite have sufficient coverage. If a test does not fail, it means that the mutant has survived, indicating a potential weakness in the test suite. This process can be repeated for different mutants, covering different parts of the code, to identify areas of the

code that may be less well-tested. Mutation testing can find areas of the code that are not covered by the test suite, or tests that are not comprehensive enough to detect faults.

Lack of acceptance testing

Another limitation could be the lack of user acceptance testing. While the test plan mentions conducting acceptance testing to gather feedback from users and ensure the API meets their needs and expectations, it does not specify how this will be done or what specific test cases will be used. This could be improved by incorporating user acceptance testing early in the development process and involving actual end-users in the testing process to gather feedback and ensure the API meets their needs.

Lack of consider for internationalization

there is no mention of testing for internationalization and localization. This could be a limitation as it is important to ensure that the API can be used by users from different regions and cultures. This can be remedied by incorporating internationalization and localization testing and incorporating test cases specifically focused on these aspects.

Human error

The testing process is conducted by humans, and there is always the possibility of human error, such as misinterpreting test results or overlooking a defect.

4.2 Identifying target coverage/performance levels for the different testing procedures

---

Structural testing

The goal of structural testing is to achieve high coverage of the code, meaning that as much of the code as possible is executed during testing. More specifically, achieve 100% coverage in the following aspects:

- Block coverage
- Statement coverage
- Branch coverage
- Path coverage
- Condition coverage

For structural testing, achieving high coverage of the code is crucial. The target level of 100% coverage ensures that all possible code paths are tested, making it more likely that defects will be detected.

Model-based testing (finite state model and flowgraph)

For model-based testing, the target coverage level should focus on ensuring that all relevant states and transitions in the finite state model or branches in flowgraph are covered. This includes identifying all possibilities, as well as testing for edge cases and exceptional states.

- For finite state model, 100% coverage for states and transitions.
- For flowgraph, 100% coverage for branches.

It can ensure that the system's behaviour in all possible states and transitions is thoroughly tested.

Functional testing

For functional testing, the target coverage level would be to test 90% of the functional requirements specified in the requirement documents. This includes all possible scenarios and edge cases for each function, such as valid and invalid inputs, empty fields, and injection attacks, making sure the system have is usable, reliable, and secure.

By testing a large percentage of the functional requirements, the system's ability to meet the requirements can be confidently verified.

Measurable attributes

For performance testing, the target level would be to ensure that the system can handle at least 100 concurrent users and maintain a response time of less than 1 second for all API requests without significant performance degradation.

The target level for scalability testing would be to identify the system's breaking point and ensure that it can handle at least 5 times the expected volume of users and transactions in a production environment. This can be measured by conducting load testing and identifying the maximum number of users and transactions the system can handle before experiencing performance issues.

The target level for security testing would be identifying and addressing at least 80% of potential vulnerabilities using security testing tools such as OWASP ZAP.

Memory usage is another measurable attribute that should be considered. The target level for memory usage should be set to ensure that the system uses memory efficiently and does not consume excessive resources. This can be measured by monitoring the system's memory usage during testing and setting a target level within acceptable limits.

4.3 Discussing how the testing carried out compares with the target levels

---

In general, the testing targets were met with some variations.

Functional testing

The target is 90% of the functional requirement’s coverage. The test plan achieved the target. But in the execution part I was only able to test about 80% of all the required functions. This variation is likely due to limitations in time and resources for testing, as well as potential issues with the implementation of certain functions in the code.

Structural testing

Structural testing also had a variation from the target. The target is 100% code coverage. During the testing process, I was only able to carry out 37 out of the 38 planned tests. This resulted in a code coverage of 97%.

While the coverage rate is still high, it falls short of the target level. This variation is likely due to the failure to simulate a connection failure with MongoDB, which resulted in one planned test not being executed.

Model-based testing

For finite state model, the goal is to achieve 100% state and transition coverage. I achieved 100% state coverage and 50% transition coverage. This is because the logout functionality is not developed in the code.

For flowgraph model, the goal is to achieve 100% branch coverage. I achieved 83% (20/24) branch coverage. This is also because the logout functionality is not developed in the code.

Testing for measurable attributes

Condition 1: 10 users per second for 10s

According to the report, the performance metrics for the system is tested for 9.466 seconds. For the http part, the system was able to handle 27 requests per second. The median response time was 10.1ms, with the 95th percentile response time being 80.6ms and the 99th percentile response time being 98.5ms. And for plugins and vusers part, the response time were also low. Overall, the performance test result meets the target level.

Metric	Value
errors.Failed capture or match	1
http.codes.200	140
http.codes.201	57
http.codes.404	1
http.codes.409	42
http.requests	240
http.responses	240
plugins.metrics-by-endpoint./.codes.200	28
plugins.metrics-by-endpoint./order.codes.201	56
plugins.metrics-by-endpoint./order/63c5add88057a1e5dd34acc3.codes.200	1
plugins.metrics-by-endpoint./order/63c5add88057a1e5dd34acd5.codes.200	1
plugins.metrics-by-endpoint./order/63c5add98057a1e5dd34ace4.codes.200	1
plugins.metrics-by-endpoint./order/63c5add98057a1e5dd34acf6.codes.200	1
plugins.metrics-by-endpoint./order/63c5add98057a1e5dd34acfc.codes.200	1
plugins.metrics-by-endpoint./order/63c5add98057a1e5dd34ad02.codes.200	1
plugins.metrics-by-endpoint./order/63c5adda8057a1e5dd34ad25.codes.200	1

I also tested the following conditions:
Condition 2: 100 users per second for 10s
Condition 3: 1000 users per second for 10s
Condition 4: 10000 users per second for 10s
In condition 2, the system is still useable, but the error rate and response time has increased slightly.

Metric	Value
errors.Failed capture or match	2
http.codes.200	1648
http.codes.201	655
http.codes.404	2
http.codes.409	330
http.requests	2635
http.responses	2635
plugins.metrics-by-endpoint./.codes.200	340
plugins.metrics-by-endpoint./order.codes.201	654
plugins.metrics-by-endpoint./order/63c5b6778057a1e5dd34ae9d.codes.200	1
plugins.metrics-by-endpoint./order/63c5b6778057a1e5dd34aea6.codes.200	1
plugins.metrics-by-endpoint./order/63c5b6778057a1e5dd34aeaf.codes.200	1
plugins.metrics-by-endpoint./order/63c5b6778057a1e5dd34aeb7.codes.200	1
plugins.metrics-by-endpoint./order/63c5b6778057a1e5dd34aeb9.codes.200	1
plugins.metrics-by-endpoint./order/63c5b6778057a1e5dd34aebb.codes.200	1
plugins.metrics-by-endpoint./order/63c5b6778057a1e5dd34aebd.codes.200	1

In condition 3, the system is barely useable, with a high error rate and long response time.

Metric	Value
errors.ETIMEDOUT	6372
http.codes.200	3602
http.codes.201	5
http.codes.400	1
http.codes.409	280
http.requests	10260
http.responses	3888
plugins.metrics-by-endpoint./.codes.200	3346
plugins.metrics-by-endpoint./order.codes.201	4
plugins.metrics-by-endpoint./users/login.codes.200	256
plugins.metrics-by-endpoint./users/register.codes.201	1
plugins.metrics-by-endpoint./users/register.codes.400	1
plugins.metrics-by-endpoint./users/register.codes.409	280
vusers.completed	3628
vusers.created	10000
vusers.created_by_name.Check API	3346

In condition 4, the system is not able to function at all.

Metric	Value
errors.ECONNREFUSED	100007
http.requests	100007
vusers.created	100007
vusers.created_by_name.Check API	33214
vusers.created_by_name.Login user and apply multiple order actions	33572
vusers.created_by_name.Register User	33221
vusers.failed	100007

The goal for performance testing is to have the system handle 100 concurrent users, I think the system's performance meets the target.

Overall, the testing targets were met to a certain extent, but variations were present due to limitations in time and resources, as well as issues with the implementation of certain functions and lack of logout functionality in the code. These variations highlight the importance of thorough testing and the need to consider potential limitations and issues when setting testing targets.

Result of testing

Functional testing:

Issues in the code:

When developers developed the code, they missed some functions that is required by the specification. Including:

1. When admin need to get user information or delete a user, the code doesn't check if the required user exists or not.
2. When getting an order's information and deleting an order, the code doesn't check if the specified order exists or not.

Solution:

The developers need to write code to check these conditions and return proper error message.

Structural testing:

Issues in the code:

The fifth function of order.js has a structural error, the function first tries to find the user, and then used the user's parameter to check the user role and user's order, and then check if the user is

found after it. This would lead to that the error code 403 for checking the user role and user's order will never be executed, and always use exception to return the error, which is 400.

This is detected by structural testing the code with the flow graph. The returned information in tests didn't meet the expectation of the flow graph.

Solution:

I modified the code and put checking if the user exists before checking the user role and user's order, and the error message returned in the tests meet the expectation.

Model-based testing:

Issues in the code:

The model-based testing found that the logout function is missing.

Solution:

The developers need to add the logout function.

4.4 Discussion of what would be necessary to achieve the target levels

---

Functional testing

For functional testing, it would be necessary to allocate more time and resources for testing, and to prioritize the most important tests to ensure they are executed. Additionally, a review of the test planning document may be necessary to identify any unnecessary tests that could be removed, allowing for more time to focus on essential tests. To exceed the target level coverage, we involve stakeholders in testing to ensure that the system meets the expectations.

Structural testing

To achieve the target levels for structural testing, I need to execute all tests, including simulating connection failures with MongoDB. This could be accomplished by setting up a test environment that mimics such environment and incorporating test cases specifically focused on this issue.

Model-based testing

To achieve the target levels for model-based testing, it would be necessary to develop the logout functionality in the code write test cases for it.

Performance and scalability testing

It would be necessary to invest in more powerful hardware and infrastructure to handle the expected volume of users and transactions in a production environment. This could include upgrading servers, optimizing database queries, and implementing caching mechanisms to improve the system's response time. To find the bottle neck, we can perform load tests on different parts of the system separately.

Stress testing can be used to identify the system's breaking point and identify bottlenecks that need to be addressed. Furthermore, conducting load testing on different parts of the system such as the database and API could also help identify scalability issues. These tests should be conducted on the same hardware and infrastructure as the production environment to ensure accurate results.

Improvement for scaffolding and instrumentation

The instrumentation could be improved in certain areas to test the requirements more adequately. For example, additional test data and edge cases could be added to test the system's behaviour under different scenarios. Additionally, more advanced performance testing tools and frameworks could be used to further stress test the system and identify any scalability issues.

Furthermore, as the testing process progresses, it may be necessary to revise the instrumentation based on any issues or challenges that arise.

The test plan needs to be continuously updated as the development and testing procedure proceeds to ensure that the testing is of high quality.

Supporting material for 4.2

Functional testing

Functional tests check that the system is working correctly and that it meets the requirements specified.

These functions below can be tested individually as unit tests. To make the tests, the scaffolding needed includes mock user information, mock admin information and IO sockets for each unit functions. And performance testing tools such as Artillery, that can be used to measure the system's performance and scalability under different loads.

Step 1: Identify functions from requirement documents.

1. Login (user & admin)
2. Register (user & admin)
3. Get information of self-user (user)
4. Get information of any user (admin)
5. Get information of all users (admin)
6. Delete self-user (user)
7. Delete any user (admin)
8. Update information of self-user (user)
9. Get self-order (user)
10. Get order of any user (admin)
11. Get a single order info (user & admin)
12. Add self-order (user & admin)
13. Update self-order (user & admin)
14. Update other’s order (admin)
15. Delete self-order (user & admin)
16. Delete another user (admin)

Step 2: Write test cases for each

Function1: Login (user & admin)

Case 1: Correct user login (email & password)
Case 2: Invalid user Email
Case 3: Empty user Email
Case 4: Incorrect user password
Case 5: Empty user password
Case 6: Correct admin login (email & password)
Case 7: Invalid admin Email
Case 8: Empty admin Email
Case 9: Incorrect admin password
Case 10: Empty admin password

Function 2: Register (user & admin)

Case 1: Correct user register (name & role & email & password & address)
Case 2: Invalid username (injection attack)
Case 3: Empty username
Case 3,4,5: Invalid (not admin or user) / injection attack / empty role
Case 6,7,8: Invalid (missing "@"; missing ".com") / empty email
Case 9: empty password
Case 10: Invalid / empty address
Case 11: Duplicate email

Function 3: Get information of self-user (user)

Case 1: Successful case

Function 4: Get information of any user (admin)

Case 1: Successful case: admin get info of user
Case 2: Successful case: admin get info of another admin
Case 3: User get info of admin
Case 4: User get info of another user
Case 5: Admin get user info, but user doesn't exist

Function 5: Get information of all users (admin)

Case 1: Successful case: admin get info of all users
Case 2: User get info of all users
Case 3: Admin get user info, but there is no user

Function 6: Delete self-user (user)

Case 1: User delete self-user
Case 2,3: User delete another user / admin

Function 7: Delete any user (admin)

Case 1: Admin delete user
Case 2: Admin delete admin
Case 3: Admin delete a user that doesn't exist

Function 8: Update information of self-user (user)

Case 1: User update info of self-user
Case 2: User update info of another user
Case 3: User update info of admin

Function 9: Get self-order (user)

Case 1: User get self-user's order
Case 2: User get other user's order
Case 3: User get admin's order

Function 10: Get order of any user (admin)

Case 1: Admin get user's order
Case 2: Admin get self-admin's order
Case 3: Admin get another admin's order
Case 4: Admin get a user's order that doesn't exist

Function 11: Get a single order info (user & admin)

Case 1: User get self-order
Case 2: Admin get self-order
Case 3: Admin get user's order
Case 4: User get other user's order
Case 5: Admin get an order that doesn't exist

Function 12: Add self-order (user & admin)

Case 1,2,3: User add self-order / other-user-order / admin-order
Case 4,5,6: Admin add self-order / other-admin-order / admin-order

Function 13&14: Update self-order & Update other's order

Case 1,2,3: User update self-order / other-user-order / admin-order
Case 4,5,6: Admin update self-order / other-admin-order / admin-order
Case 7: Admin update an order, but order doesn't exist

Function 15&16: Delete self-order & Delete another user

Case 1,2,3: User delete self-order / other-user-order / admin-order
Case 4,5,6: Admin delete self-order / other-admin-order / admin-order
Case 7: Admin delete a user, but user doesn't exist

Integration level

Integration tests are used to test how different parts of the system work together. For this project, integration tests could include:

1. Testing the integration between the user registration process and the database, by checking that user information is correctly stored in the MongoDB after registration.
2. Testing the integration between the login process and the JWT authentication, by checking that the correct JWT token is returned after a successful login and that the token can be used to authenticate subsequent requests.
3. Testing the integration between the order placement process and the inventory system, by checking that the correct items are removed from the inventory after an order is placed.
4. Testing the integration between the login process and the adding order function, by checking that an order can be found after user login and adding an order.
5. Testing the integration between updating order and deleting order, by checking that an order cannot be found after updated and deleted.
6. Testing the integration between updating user info and deleting a user, by checking that an user cannot be found after updated and deleted.
7. Testing the integration between the password encryption process and the login process, by checking that the system correctly compares the hashed and salted password stored in the database with the one provided by the user during login.
8. Testing the integration between the API endpoints and the database queries, by checking that the correct data is being retrieved from the database and returned in the API response.
9. Testing the integration between the system and the performance testing tool, Artillery, to ensure it can handle expected traffic without crashing or slowing down.

System level

System tests are used to test the entire system as a whole, including all of its components and external interfaces. For example:

Testing the entire API, including making various requests to different endpoints and verifying that the correct HTTP status codes and data are returned.

Testing the entire system's performance, including testing how the system behaves when it's under a heavy load, and verifying that it can handle expected traffic without crashing or slowing down.

Testing the entire system's security, including testing for common vulnerabilities like SQL injection, cross-site scripting, or cross-site request forgery.

Testing the entire system's scalability and reliability, including testing how the system behaves when it's under a heavy load, and verifying that it can handle expected traffic without crashing or slowing down.

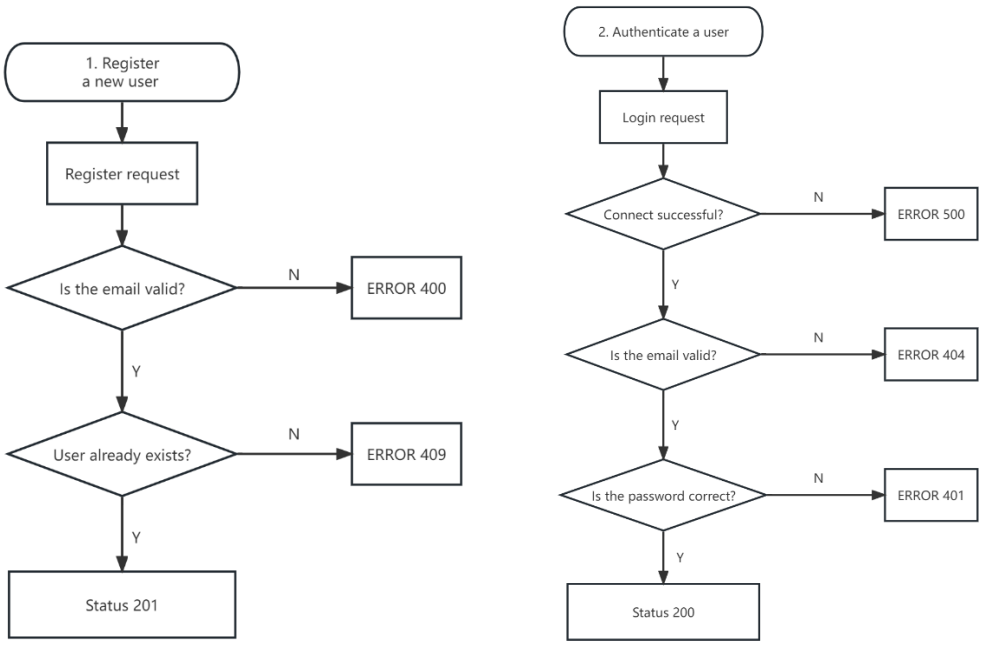
Structural testing

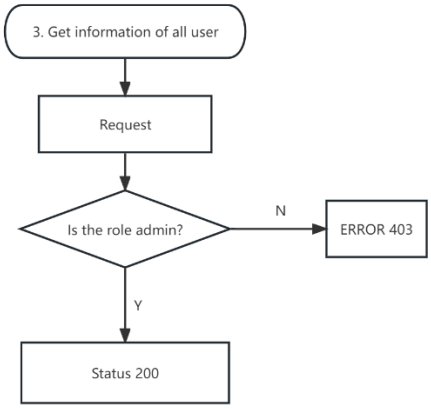
Structural testing, also known as "white box" testing, is a technique that focuses on testing the internal structure of the code and how it is implemented.

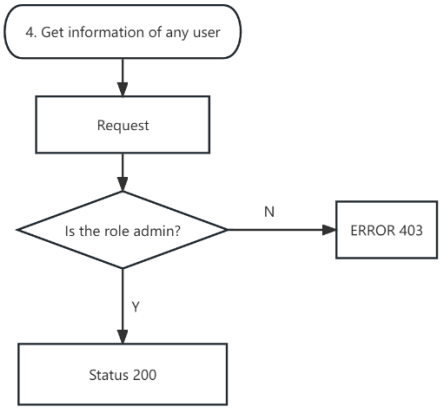
Understand the codebase & identify testable components

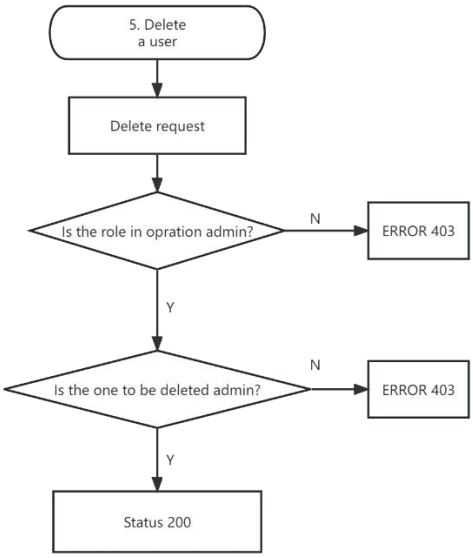
I analysed the code in the endpoints file, mainly the user.js file and order.js file, and drew the flow graph of the functions:

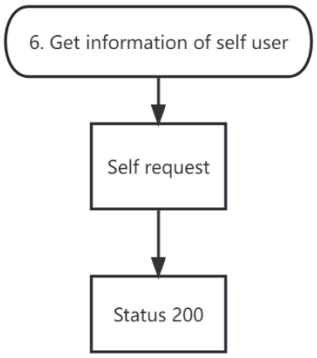
User.js

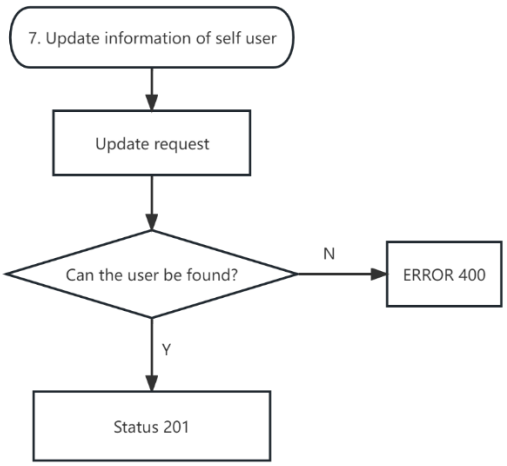


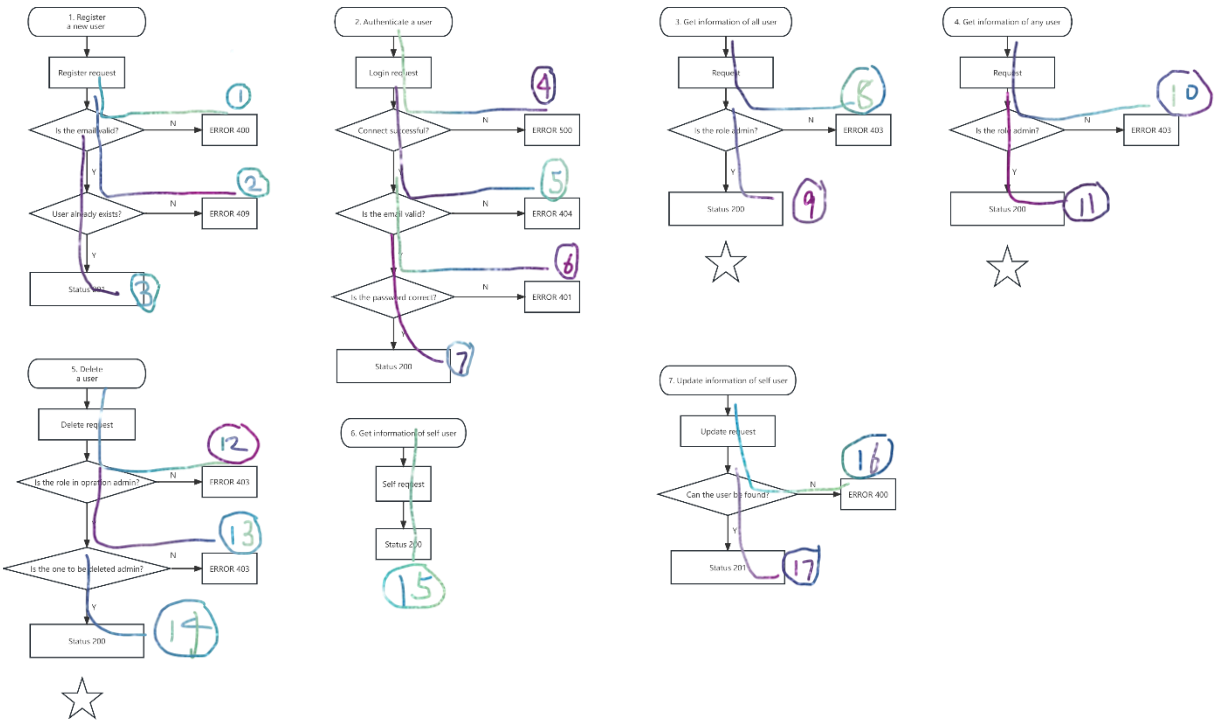

3. Get information of all user
Request
Is the role admin?
N
ERROR 403
Y
Status 200


4. Get information of any user
Request
Is the role admin?
N
ERROR 403
Y
Status 200

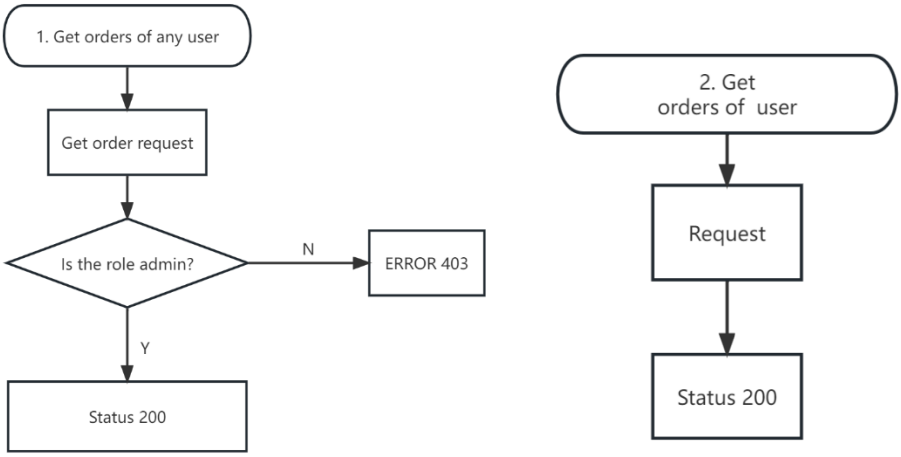

5. Delete a user
Delete request
Is the role in opration admin?
N
ERROR 403
Y
Is the one to be deleted admin?
N
ERROR 403
Y
Status 200

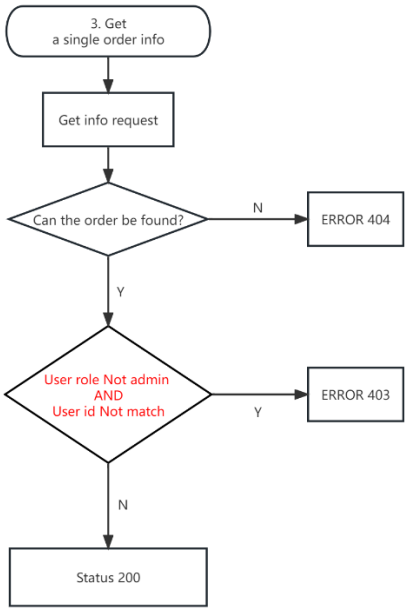

6. Get information of self user
Self request
Status 200

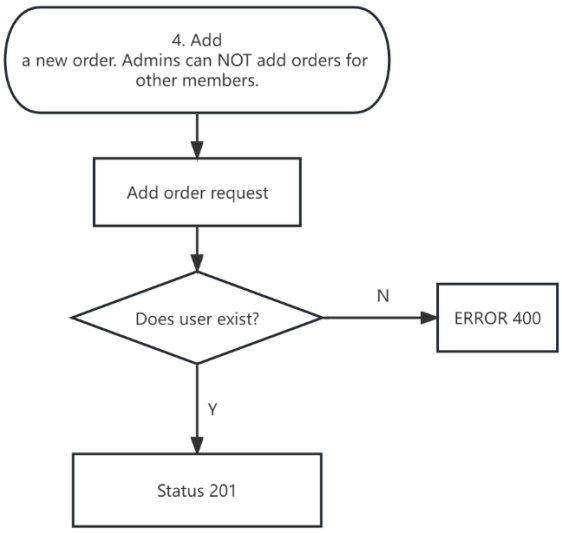

7. Update information of self user
Update request
Can the user be found?
N
ERROR 400
Y
Status 201

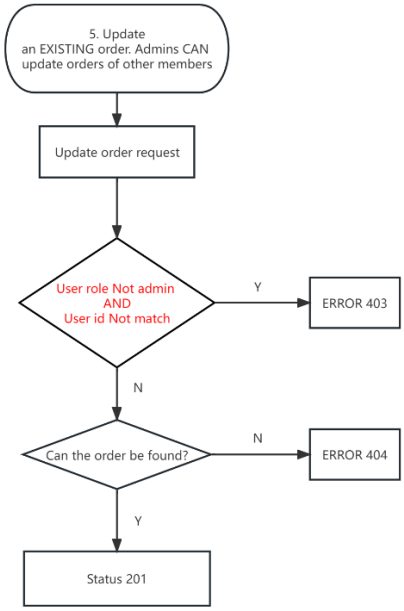

1. Register a new user
Register request
Is the email valid?
ERROR 400
User already exists?
ERROR 409
2. Authenticate a user
Login request
Connect successful?
ERROR 500
Is the email valid?
ERROR 404
Is the password correct?
ERROR 401
Status 200
3. Get information of all user
Request
Is the role admin?
ERROR 403
Status 200
4. Get information of any user
Request
Is the role admin?
ERROR 403
Status 200
5. Delete a user
Delete request
Is the role in opration admin?
ERROR 403
Is the one to be deleted admin?
ERROR 403
Status 200
6. Get information of self user
Self request
Status 200
7. Update information of self user
Update request
Can the user be found?
ERROR 400
Status 201

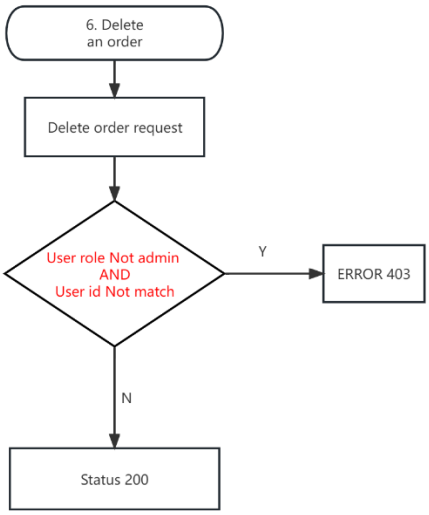
Order.js

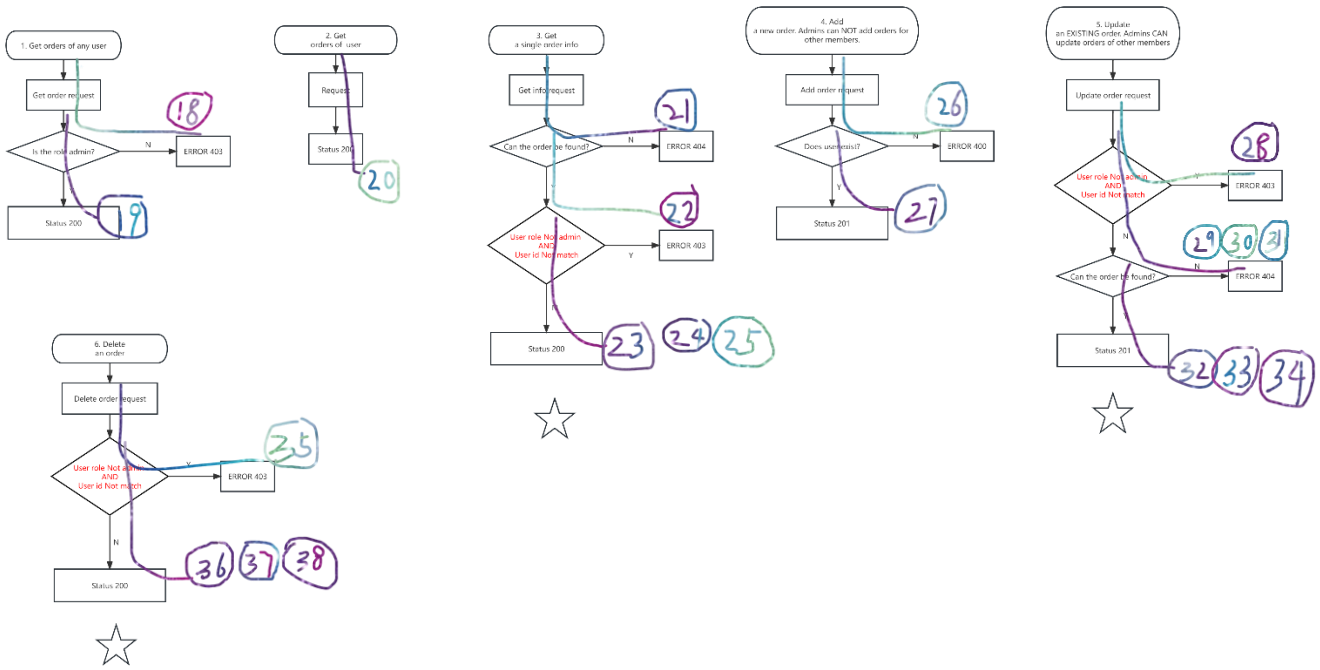

1. Get orders of any user
Get order request
Is the role admin?
N
ERROR 403
Y
Status 200
2. Get orders of user
Request
Status 200


3. Get
a single order info
Get info request
Can the order be found?
N
ERROR 404
Y
User role Not admin
AND
User id Not match
Y
ERROR 403
N
Status 200


4. Add
a new order. Admins can NOT add orders for
other members.
Add order request
Does user exist?
N
ERROR 400
Y
Status 201


5. Update
an EXISTING order. Admins CAN
update orders of other members
Update order request
User role Not admin
AND
User id Not match
Y
ERROR 403
N
Can the order be found?
N
ERROR 404
Y
Status 201


6. Delete
an order
Delete order request
User role Not admin
AND
User id Not match
Y
ERROR 403
N
Status 200

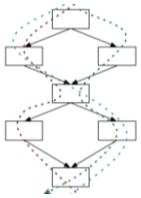


Coverage Analysation:

The first thing to think about is statement coverage and block coverage. To achieve 100% statement/block coverage, we can generate test cases for each Y/N choices.

The next more refined level is branch coverage. Branch coverage is stricter when there are cases where a block can be accessed through multiple paths. But in the code base, there is no such cases, so generating test cases for each Y/N can guarantee 100% branch coverage.

The next level is condition coverage. The difference exists when there are and/or Boolean conditions. A block that contains A and B have 4 possibilities: A=1, B=1; A=1, B=0; A=0, B=1; A=0, B=0. All 4 cases should be tested. In the code of the project, such cases exist in function 3,5,6 of order.js, so cases for all 4 possibilities should be considered to achieve 100% condition coverage.

For path coverage, there is no structure like  . So, no more cases are needed to achieve 100% path coverage.

The scaffolding would include mock user information, mock admin information and IO sockets for each unit functions; and code coverage tools such as Istanbul, that can be used to measure the percentage of the codebase that has been tested and identify any untested areas.

Integration level

We can test each of 2 function's combinations to perform integration tests.

System-level

Testing the integration with external systems: Ensuring that the system is properly integrated with external systems, such as the MongoDB database, and that there are no structural issues.

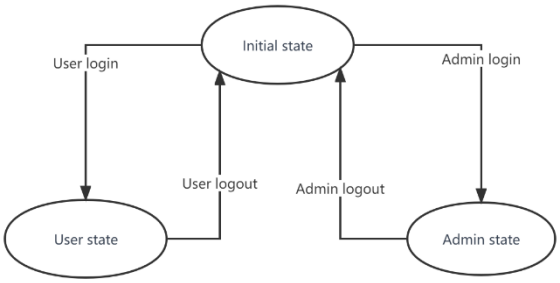
Testing the scalability and performance of the system: Ensuring that the system can handle expected traffic without crashing or slowing down.

Testing the system's robustness and error handling: Ensuring that the system can handle unexpected inputs and events, and that the appropriate error messages and responses are generated.

Testing the system's reliability: Ensuring that the system is reliable and that it will perform consistently over time.

Model-based approaches

Finite State Model



Flowgraph Model