LO_3 notes

For 3.1, I covered the testing of the functional/structural/model-based method and analysed them from unit/integrated/system level. Then I set the target level and reported the outcome.

For 3.2, I evaluated the functional testing, structural testing, model-based testing from different perspectives in terms of their properties, defect detection, cost, and whether they are optimistic or pessimistic.

For 3.3, I compared the test carried out and the target in the ***3.1/3.3 testing outcome***. Then I covered the result/achievement of the testing in the ***notes***. I found errors in the code through testing and provided solutions.

For 3.4, I evaluated the results of the testing, and stated what I have learned through the testing process.

LO 3

3.1 Range of techniques

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.

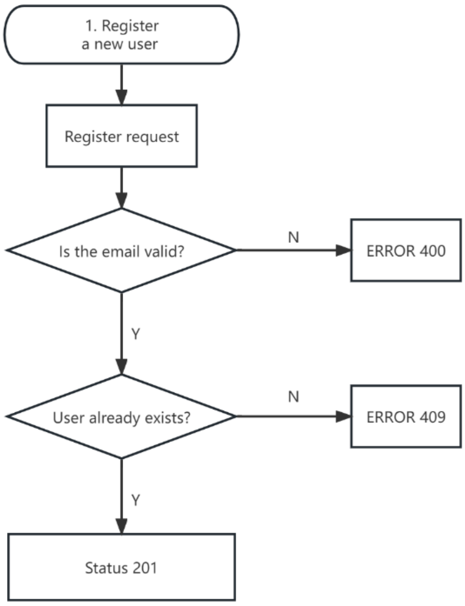
Structural testing

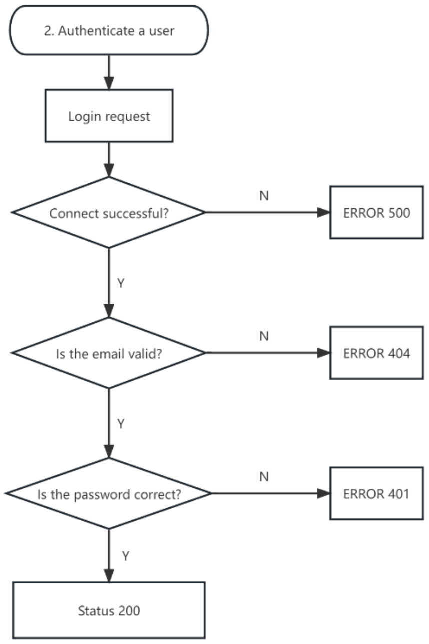
Structural testing, also known as "white box" testing, is a technique that focuses on testing the internal structure of the code and how it is implemented.

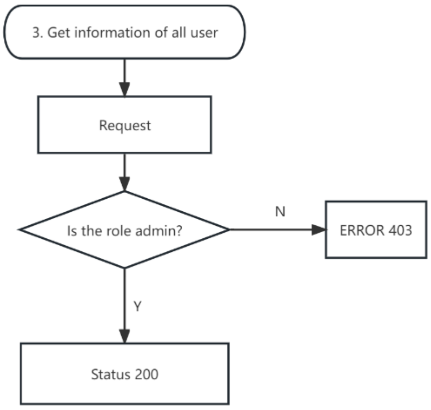
Understand the codebase & identify testable components

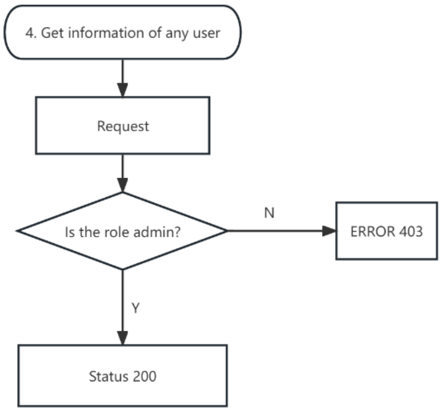
I analysed the code in the endpoints file, mainly the user.js file and order.js file, and drew the flow graph of the functions:

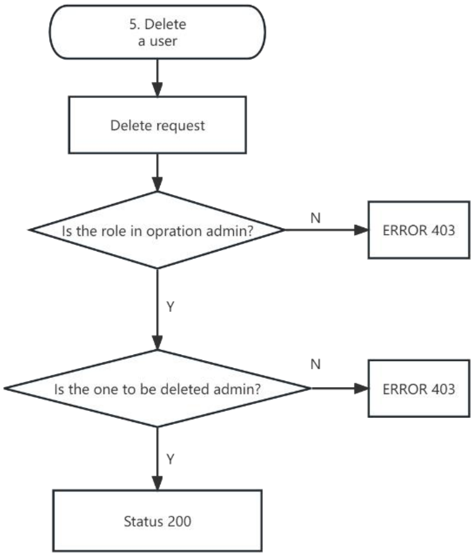
User.js

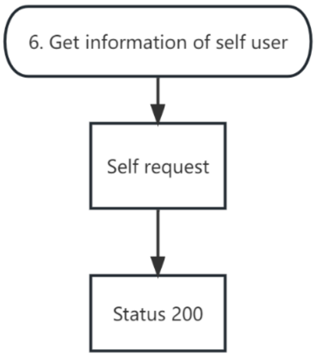


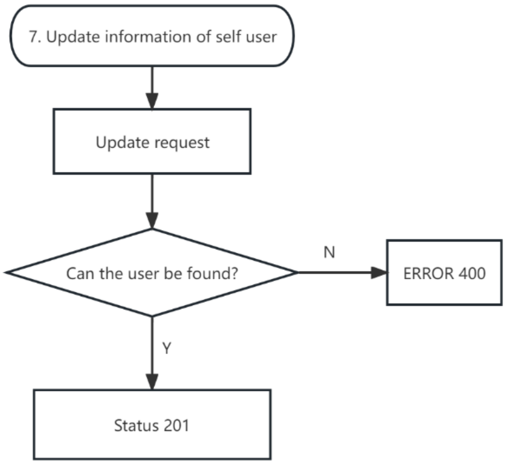


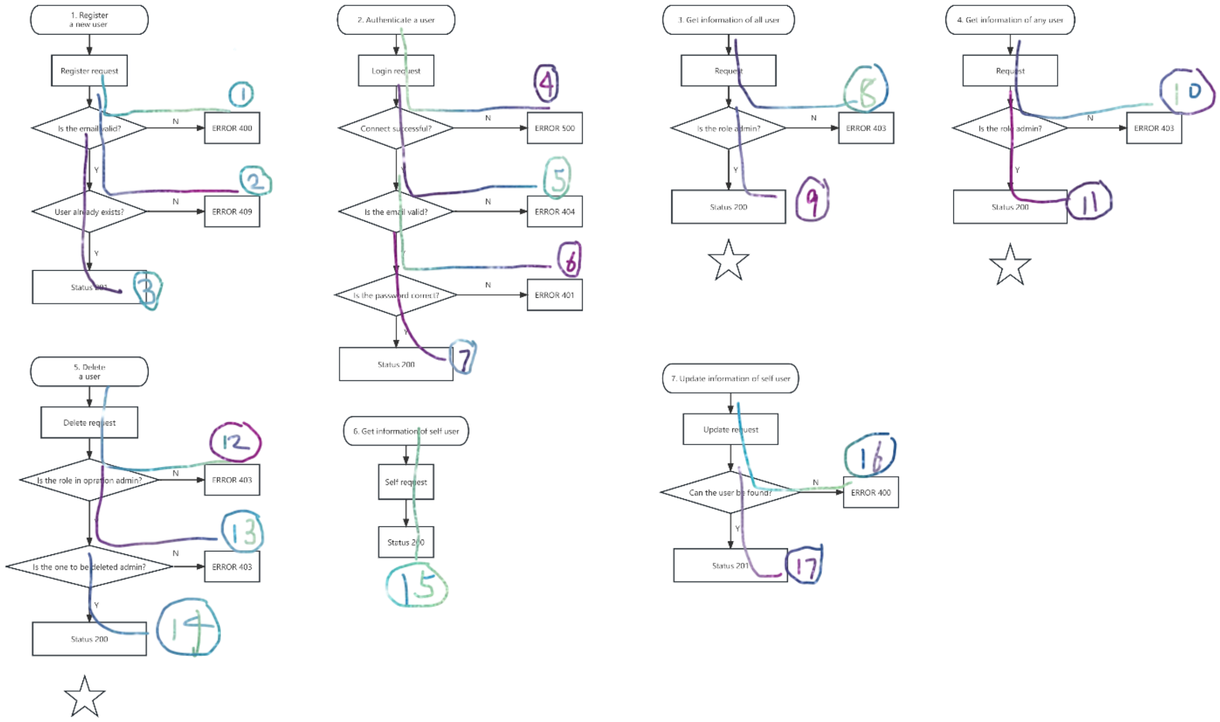




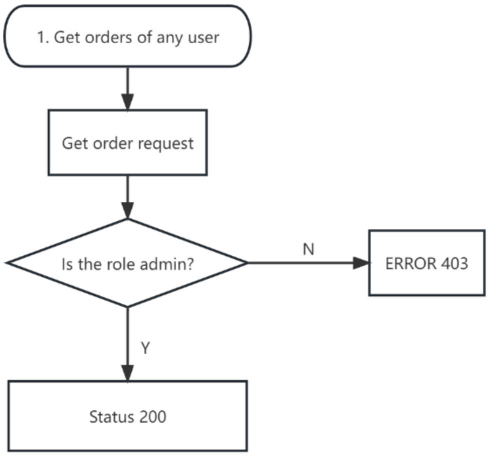

5. Delete a user
Delete request
Is the role in opration admin?
N
ERROR 403
Y
Is the one to be deleted admin?
N
ERROR 403
Y
Status 200

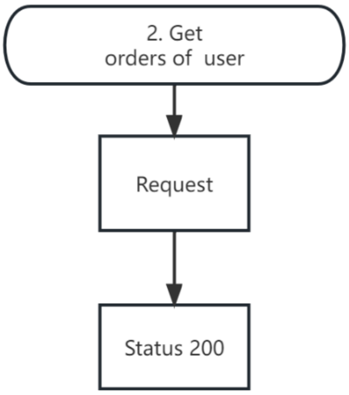

6. Get information of self user
Self request
Status 200

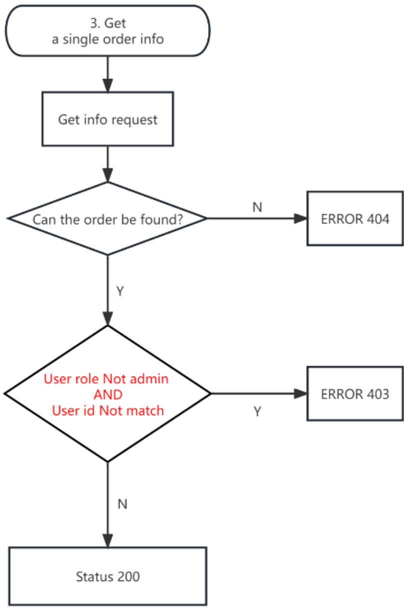

7. Update information of self user
Update request
Can the user be found?
N
ERROR 400
Y
Status 201

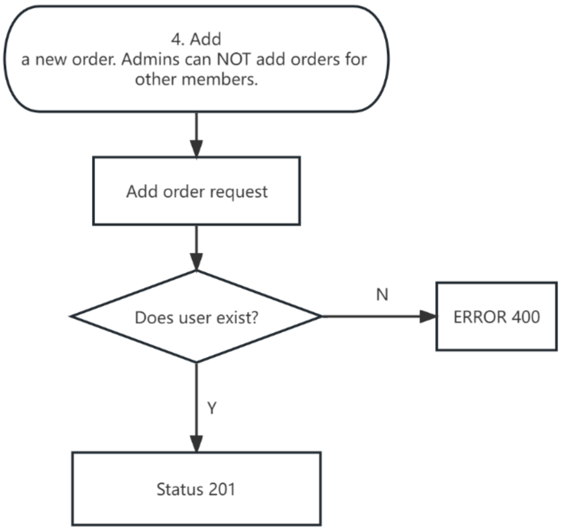

1. Register a new user
Register request
Is the email valid?
N
ERROR 400
Y
User already exists?
N
ERROR 409
Y
Status 201
1
2
3
2. Authenticate a user
Login request
Connect successful?
N
ERROR 500
Is the email valid?
N
ERROR 404
Is the password correct?
N
ERROR 401
Y
Status 200
4
5
6
7
3. Get information of all user
Request
Is the role admin?
N
ERROR 403
Y
Status 200
8
9
☆
4. Get information of any user
Request
Is the role admin?
N
ERROR 403
Y
Status 200
10
11
☆
5. Delete a user
Delete request
Is the role in opration admin?
N
ERROR 403
Is the one to be deleted admin?
N
ERROR 403
Y
Status 200
12
13
14
☆
6. Get information of self user
Self request
Status 200
15
7. Update information of self user
Update request
Can the user be found?
N
ERROR 400
Y
Status 201
16
17

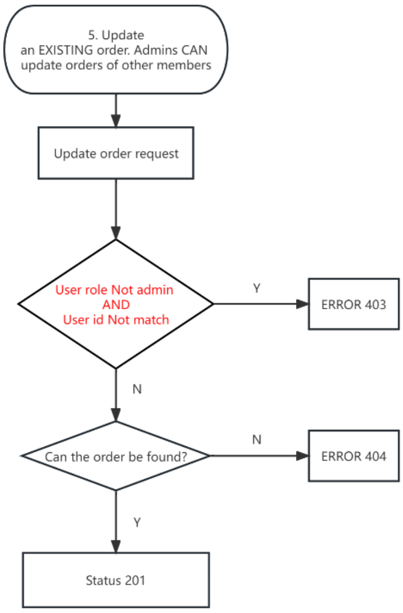
Order.js

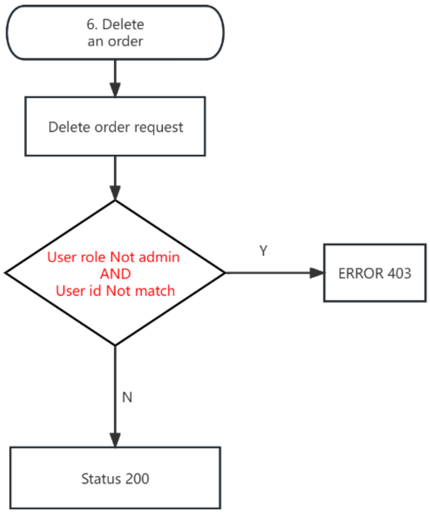

1. Get orders of any user
Get order request
Is the role admin?
N
ERROR 403
Y
Status 200

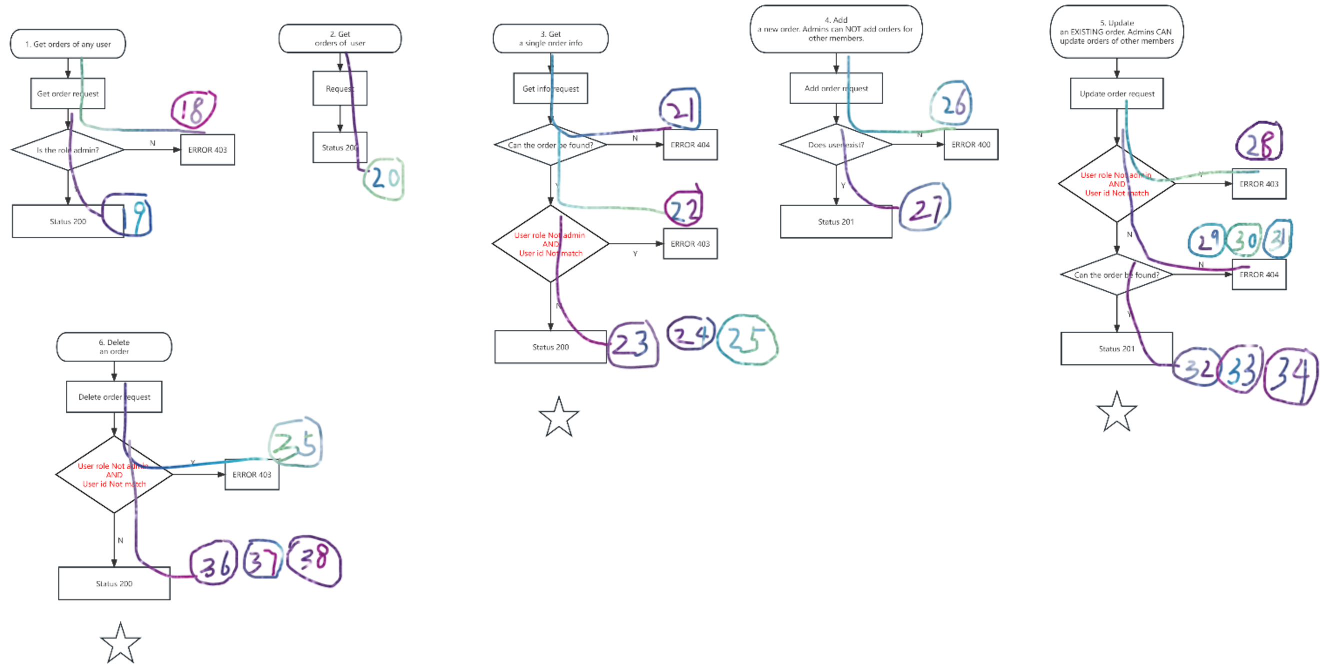

2. Get
orders of user
Request
Status 200


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

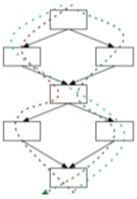


Coverage Analysation:

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.

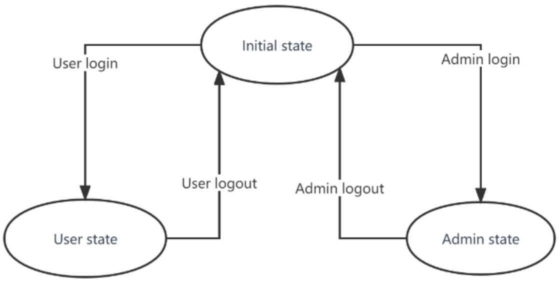
Testing the scalability and performance of the system: Ensuring that the system can handle expected traffic without crashing or slowing down.

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.

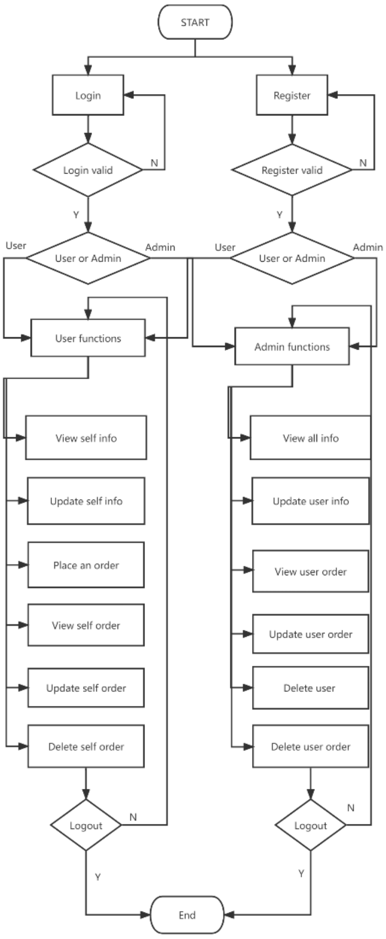
Testing the system's reliability: Ensuring that the system is reliable and that it will perform consistently over time.

Model-based approaches

Finite State Model



Flowgraph Model



Testing target

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

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.

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.

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.

Testing outcome

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 is due to several factors, such as limited time and resources for testing.

Structural testing

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 is because I failed to simulate the connection failure with MongoDB.

Model-based testing

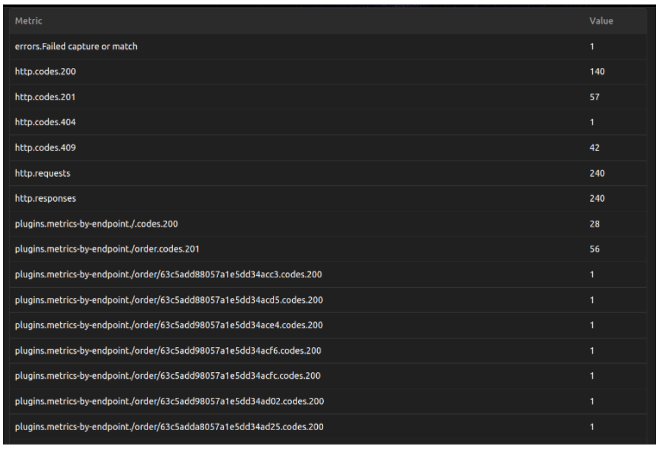
For finite state model, the goal is to achieve 100% state and transition coverage. I achieved 100% state coverage and 50% transition coverage. This is because the logout functionality is not developed in the code.

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

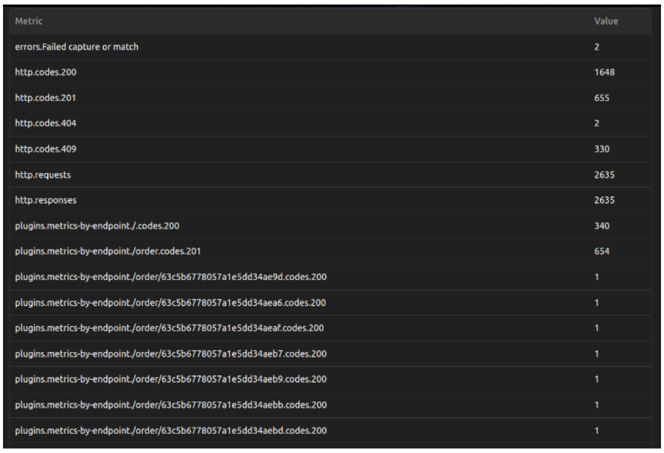
Condition 1: 10 users per second for 10s

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.



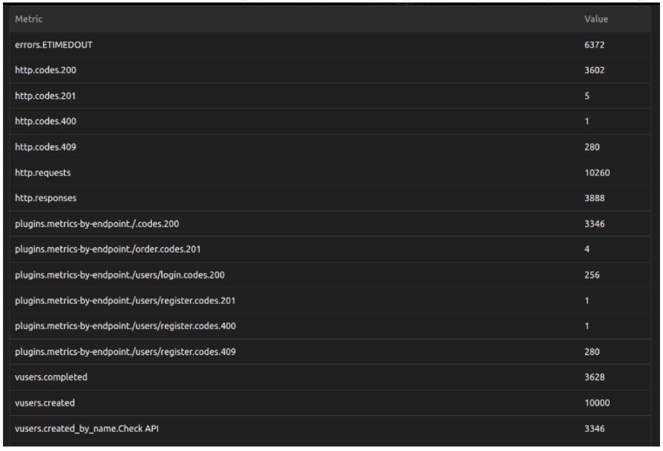
Metric	Value
errors.Failed capture or match	1
http.codes.200	140
http.codes.201	57
http.codes.404	1
http.codes.409	42
http.requests	240
http.responses	240
plugins.metrics-by-endpoint./.codes.200	28
plugins.metrics-by-endpoint./order.codes.201	56
plugins.metrics-by-endpoint./order/63c5add88057a1e5dd34acc3.codes.200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



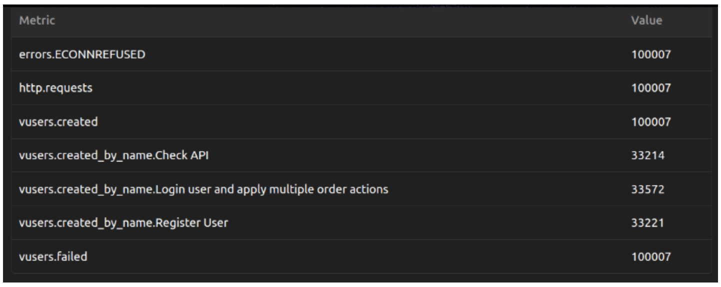
Metric	Value
errors.Failed capture or match	2
http.codes.200	1648
http.codes.201	655
http.codes.404	2
http.codes.409	330
http.requests	2635
http.responses	2635
plugins.metrics-by-endpoint./.codes.200	340
plugins.metrics-by-endpoint./order.codes.201	654
plugins.metrics-by-endpoint./order/63c5b6778057a1e5dd34ae9d.codes.200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

3.2 Evaluation criteria for the adequacy of the testing

Functional testing:

Most of the functional requirements can be tested by the functional tests. The functional test cases are generated based on the requirement specification, which describes in detail what the project need to do. It cannot check code that doesn't execute the function in the requirement. Overall, I believe there is a lot of confidence can be placed in the system's ability to meet the requirements.

Defect detection: The test cases designed can detect defects like injection attacks in the system.

Time and resource: The time for generating the tests and executing the tests are both relatively long.

Structural testing:

All functions in the code can be covered, but if the developer failed to write the functions for the requirements, then we cannot ensure this part of function is usable. The white box testing cannot test the functions that are not implemented, but it can check the code that doesn't directly execute a function. Structural testing can provide some confidence, but it would be better to combine functional testing and structural testing.

Defect detection: This kind of tests can verify then defects the developer noticed, but if the developer failed to detect some defects, then we cannot test them.

Time and resource: The time for generating the test cases and executing the test cases are relatively short.

Model-based testing: Finite State Model:

The model is an abstraction of a part of the functions, so I cannot guarantee all or most of the functions and requirements are detected.

For finite state model, I am only able to generate the model of login for this project. Even though I have 100% state coverage and 100% transition coverage, only limited functions can be tested.

For flowgraph, there are blocks with multiple conditions cannot be thoroughly tested.

So, there is not a lot of confidence if we only use model-based testing to test the system.

Defect detection: The ability of the test cases to detect defects in the system.

Time and resource: The time for generating the model and tests is relatively long but the time for executing the tests is short.

From the optimistic/pessimistic perspective:

Functional testing can be seen as optimistic, as it tests the system based on the requirements specified and assumes that the system is working correctly. It cannot check for code that doesn't execute the functions in the requirements.

Structural testing is considered more pessimistic because it tests the internal structure of the code, including the parts of the code that may not be directly related to the requirements. It cannot test the functions that are not implemented, but it can check the code that doesn't directly execute a function.

Model-based testing can be seen as a combination of optimistic and pessimistic. On one hand, it generates test cases based on a model of the system's behaviour, which can test the system's behaviour in various states and transitions, as well as its performance in edge cases and exceptional states. On the other hand, the model is an abstraction of a part of the functions, so it cannot guarantee that all or most of the functions and requirements are detected.

In general, testing methods that aim to find as many defects as possible are pessimistic. On the other hand, testing methods that aim to prove that the system is working correctly are optimistic.

3.3 Results 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.

3.4 Evaluation of the results

The structural testing is quite useful, and I found that it is the easiest one to implement. Reading the code base can help me to understand the input and the output of the functions, helping me to write the testing code more quickly.

Functional testing can find the required functions that the developers forgot to develop, which is essential to delivering a good product.

For model-based testing, I initially thought that it may not be so useful for this project because this method will only keep the functions that can fit in the model and omit everything else. But now I found that this method can find some missing functions that is useful but not included in the code or the specification. for example, in the FSM, the state transferred from idle to user through login, then the state needs a way to transfer back, and the logout function, which is missing in structural and functional testing, will be needed. So, one of the model-based testing's unique effect is that it can find deficiency in the specification.

Test Log

Test Log			
Test Description	Expected Result	Actual Result	Pass/Fail
Get all users	200	200	Pass
Get one user	200	200	Pass
Get posted orders	200	200	Pass
Get orders of another user	200	200	Pass
Hit generic admin user error	400	400	Pass
Add a new order	201	201	Pass
Delete a simple user	200	200	Pass
Fail to delete an admin user	403	403	Pass
Get a specific order3	200	200	Pass
Get a specific order4	200	200	Pass
Add, then update an order3	"{Test Order Updated}"	"{Test Order Updated}"	Pass
Add, then update an order3.2	404	404	Pass
Add, then update an order4	"{Test Order Updated}"	"{Test Order Updated}"	Pass
add, then update an order3.3	404	404	Pass
add, then delete an order3	404	404	Pass
Fail to get all users	403	403	Pass
Fail to get one user	403	403	Pass
Update credentials	"Updated User"	"Updated User"	Pass
Get user orders	200	200	Pass
Get a specific order1	200	200	Pass
Get a specific order2	403	403	Pass
Add, then update an order1	"{Test Order Updated}"	"{Test Order Updated}"	Pass
Add, then update an order1.2	404	404	Pass
Add, then update an order2	403	403	Pass
Add, then delete an order2.2	404	404	Pass
Add, then delete an order3	403	403	Pass
Fail to access orders of another user if the role is not admin	403	403	Pass
Fail deleting a user if role is not admin	403	403	Pass
Get info of self user	200	200	Pass
Check system is on	200	200	Pass
Login user	200	200	Pass
Fail to login user (wrong password)	401	401	Pass
Fail to login user (invalid email)	404	404	Pass
Hit random endpoint	404	404	Pass
Fail unauthorized access to /users	401	401	Pass
Fail unauthorized access to /orders	401	401	Pass
Fail unauthorized access to /orders/user/{id}	401	401	Pass
Fail unauthorized access to post /order	401	401	Pass

Fail unauthorized access to put /order	401	401	Pass
Fail unauthorized access to delete /order/{id}	401	401	Pass