LO_2

The supporting material includes LO_2 Test Planning document and Supporting Notes.

For 2.1, I wrote a test plan document that covers an introduction, priority of requirements, scaffolding, testing life cycle, example test cases, test tools, potential risks, coverage, and potential improvement. Furthermore, the Detailed test organisation using structural, functional, and model-based testing method document showed the test cases and how and why the test cases are generated.

For 2.2, potential omissions/risks as well as potential improvement are covered in the test planning document; how the plan can ensure adequate testing is covered in 2.2 notes. The Evaluation of the detail of functional, structural, and model-based testing evaluated the detailed test plan in detail.

For 2.3, the scaffolding and instrumentation part of the test planning document shows what the necessary scaffolding/instrumentations are; the 2.3 notes provided some justification that the proposed scaffolding is adequate.

For 2.4, in the 2.4 notes I evaluated the scaffolding/instrumentation, and the potential improvement part in the test plan document showed how the scaffolding/instrumentation could be improved to better the result. The 2.3 and 2.4 each cover some of the other's content.

Test Planning document

Introduction:

This Test Planning document outlines the approach and strategy for testing the API for User Registration, Authentication, and Order Placement. The API will be tested for its functionality, performance, scalability, security, usability, and other non-functional requirements. The test plan will use a combination of manual and automated testing, with the aim of ensuring that all requirements are thoroughly tested, and any defects are identified and addressed early in the development process. This test planning document is intended to provide a clear understanding of the testing scope and goals, and to guide the execution of the tests.

Priority of Requirements:

The functional requirements for the API, as outlined in the Requirement Document, will be prioritized as follows:

User registration: Ensuring that users can successfully register with the system and that all required information is collected and stored correctly.

User authentication: Ensuring that users can successfully authenticate with the system and that the correct level of access is granted based on their role (Admin or User).

Order placement: Ensuring that users can place orders for predefined boxes at the imaginary food shop and that the orders are processed and stored correctly.

User registration and login are seen as the most vital, as both firm employees and administrators require them to access the system. Order placement, viewing, updating, and deleting for corporate employees are also deemed essential since they allow employees to manage their orders.

The non-functional requirements for the API will also be prioritized, with a focus on security, scalability, and reliability.

Scaffolding and Instrumentation:

Some potential scaffolding and instrumentation for this project:

A test environment with the necessary hardware, software, and dependencies installed.

Test data, including user/admin information and their order's information.

A mock database that can simulate the behaviour of MongoDB can help test the front end along without the integration of the backend.

A mock login/authentication system to test the ordering function.

A mock ordering system to test the login/authentication function.

Test tools and frameworks, such as Jest for unit and integration testing, and Artillery for performance testing.

Test reporting tools: they are used to generate reports and metrics about the testing process.

There are several types of other scaffolding and instrumentation can be used:

1.Logging: This involves adding code to the system that generates log messages to record events or data of interest. These log messages can be used to track the flow of execution through the system or to record specific data values.

2.Profiling: This involves adding code to the system that measures the performance of specific sections of the code, such as the execution time or the memory usage. This data can be used to identify bottlenecks or inefficiencies in the system.

3.Monitoring: This involves adding code to the system that tracks specific metrics or events in real-time, such as the system's CPU usage or the number of incoming requests. This data can be used to monitor the system's health and identify any issues that may arise.

Testing Lifecycle:

The testing for this project will be integrated into the development lifecycle as follows:

Unit testing: Unit tests will be written for each component of the system, using the Jest framework. These tests will be conducted during the development phase to ensure that individual components are working as expected.

Integration testing: Integration tests will be written to test the integration between different components of the system. These tests will be run after the unit tests are complete, during the integration phase.

Acceptance testing: User acceptance testing will be conducted during the acceptance phase, to gather feedback from users and ensure that the API meets their needs and expectations. These tests will be run after the integration tests are complete.

Performance testing: Performance tests will be run using the Artillery tool to ensure that the system is scalable and can handle high levels of load. These tests will be conducted during the acceptance phase.

Test Cases:

The following test cases will be developed to test the system:

User registration/authentication test cases: verify administrators/users can login or register to the system using correct information, invalid input and edge cases.

Order placement test cases: successful order placement and invalid input.

Other more detailed test cases can also be carried out, like can user/administrators view/delete orders correctly or not.

Performance testing: test the reaction time of the system.

Test Tools:

The following test tools can be used to test the system:

Jest: This tool will be used for automated unit and integration testing.

Artillery: This tool will be used for automated performance testing.

Postman: This tool will be used for manual testing of the API routes.

Risks:

Security risks: The system should be tested for potential vulnerabilities, such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF) attacks; and ensure that data is stored and handled securely and appropriately.

Strategies to mitigate these risks include input validation and sanitization, and proper use of encryption and secure communications.

Technology Risks: Error may occur when engaging with an unknown commercial off-the-shelf (COTS) component. There may be insufficient test and analysis automation tools and non-proprietary components do not fulfil quality criteria.

To manage the risks, we can anticipate and allocate time for testing novel interfaces; train COTS components and new tools; monitor, document, and disseminate common mistakes and idioms; and introduce new tools through pilot projects or prototyping exercises with reduced risk. Introduce new tools through low-risk pilot projects and prototyping, plan training time.

Schedule Risks: Integration testing may incur unanticipated expenses and delays due to insufficient unit testing. And due of the difficulties of arranging meetings, inspection can be a bottleneck in development.

Solution: Track and reward quality unit testing as shown by low fault density in integration. Set aside time in the weekly calendar for inspections to take precedence over other meetings and tasks.

Development Risks: Inadequate unit testing and analysis prior to committing to the code base, or the delivery of low-quality software to the testing group.

Solution: Provide early warning and feedback; schedule inspection of design, code, and test suites; link development and inspection to the incentive system; enhance training through inspection; and mandate coverage or other unit test-level criteria.

Executions Risks: Costs of execution are more than planned, e.g., testing requires expensive or complex machines or systems that are not readily available.

We can reduce the number of components that necessitate the execution of the full system. Examine the architecture to determine and enhance testability. Increase intermediate feedback. Invest in scaffolding.

Test Coverage:

The test planning document aim to achieve high test coverage, including all functional and non-functional requirements, with the goal of testing all requirements thoroughly and ensuring that all defects are identified and addressed.

Potential improvement:

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.

Learning outcome 2 notes

2.1 Construction of the test plan

This part of the LO 2 is in the Test Planning Document.

2.2 Evaluation of the quality of the test plan

The risk part is shown in the Test Planning Document.

To assess how well this test plan will ensure adequate testing:

This test plan outlines the strategy for testing the API, including functional, structural, and performance testing. It also prioritizes the functional requirements and non-functional requirements to ensure that the most critical aspects of the system are thoroughly tested. The plan also includes scaffolding and instrumentation, such as test data, mock database etc., which are important for conducting effective testing.

The risk section of the test plan is also well-developed, identifying potential risks to the system and outlining strategies for mitigating them. This includes security risks, technology risks, schedule risks, development risks, and execution risks.

Overall, the test plan appears to be well-structured and should ensure adequate testing of the API.

2.3 Instrumentation of the code

The scaffolding part is shown in the Test Planning Document.

Justify that the scaffolding is adequate:

The test plan includes a section on scaffolding and instrumentation that describes the tools and resources required to run the tests. The plan mentions a test environment with the necessary hardware, software, and dependencies installed, test data including user/admin information and their order's information, a mock database that simulates the behaviour of MongoDB, test tools and frameworks such as Jest for unit and integration testing and Artillery for performance testing, and test reporting tools. These resources provide a solid foundation for testing and should ensure that the tests are carried out accurately and efficiently.

Furthermore, the plan mentions various types of code scaffolding that can be used, such as logging, profiling, and monitoring. These scaffoldings enable developers to track the execution flow and measure performance, which can aid in identifying any issues in the system and improving the overall performance of the API. The inclusion of this type of scaffolding in the plan is sufficient and will aid in ensuring that the tests are thorough and effective.

Overall, these tools provide a solid foundation for developing, testing, and deploying the API.

2.4 Evaluation of the instrumentation

This plan for scaffolding and instrumentation appears to be well thought out and comprehensive. It covers the necessary hardware, software, and dependencies for the test environment, test data for various scenarios, mock systems for testing specific functions, and test tools and frameworks for various types of testing. Additionally, it includes logging, profiling, and monitoring for tracking and measuring the performance and behaviour of the system during testing.

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. It would also be beneficial to also include a plan for managing and storing the generated logs and metrics, and to also add a plan for testing the mock systems to ensure their accuracy.

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

Detailed test organisation using structural, functional, and model-based testing method.

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.

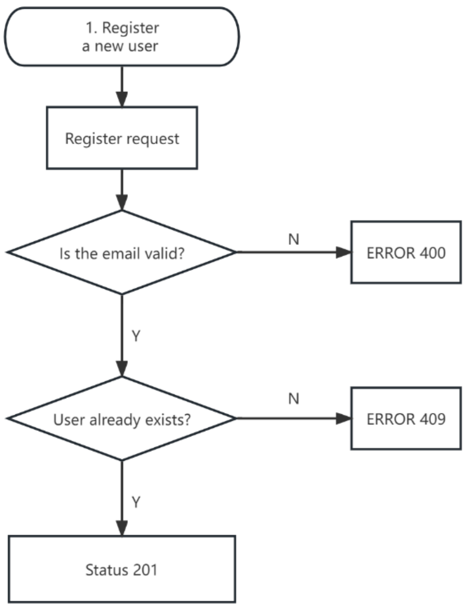
Structural testing

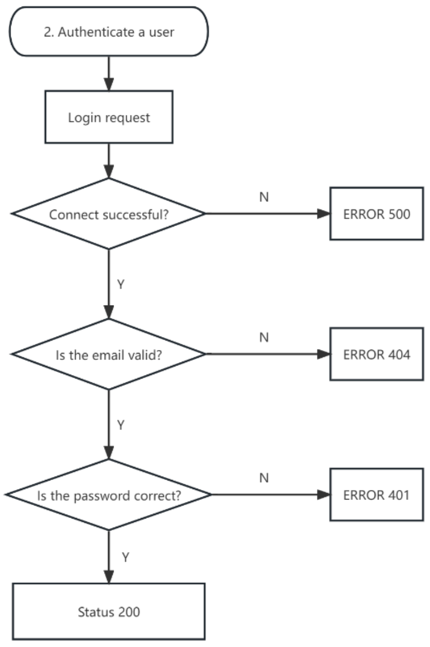
Structural testing, also known as "white box" testing, is a technique that focuses on testing the internal structure of the code and how it is implemented.

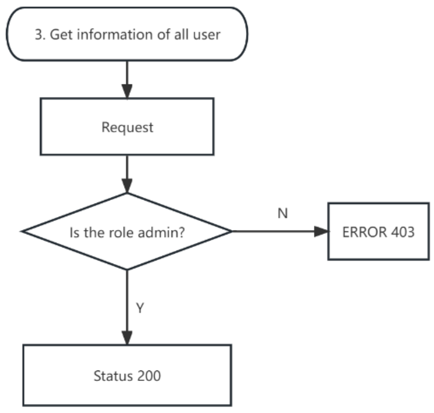
Understand the codebase & identify testable components

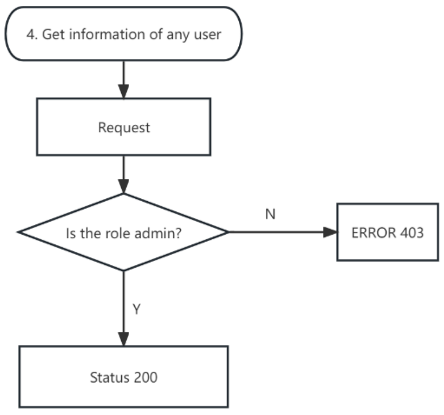
I analysed the code in the endpoints file, mainly the user.js file and order.js file, and drew the flow graph of the functions:

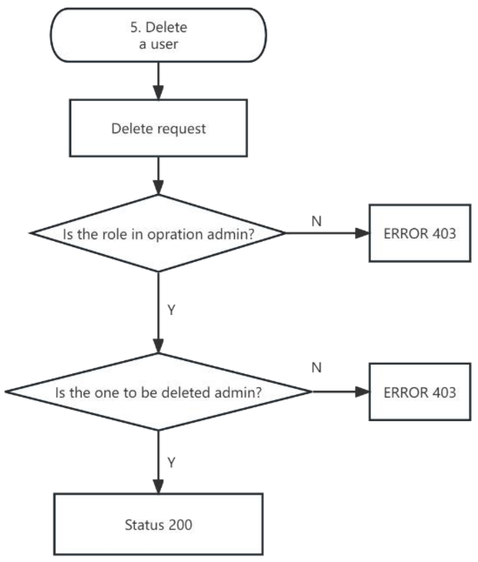
User.js

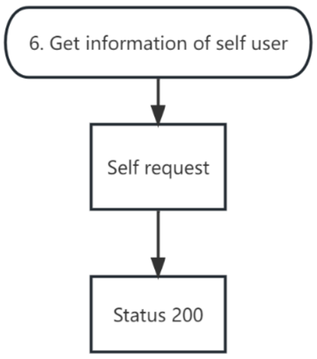


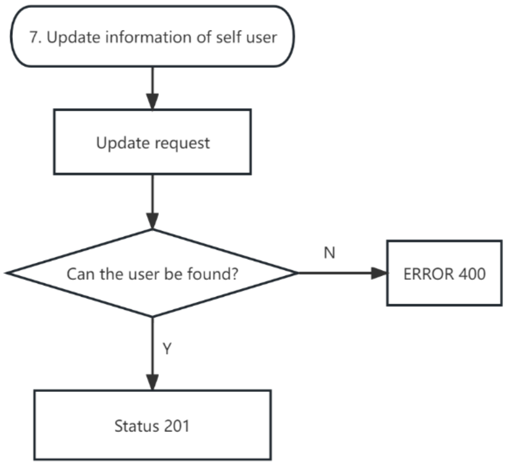


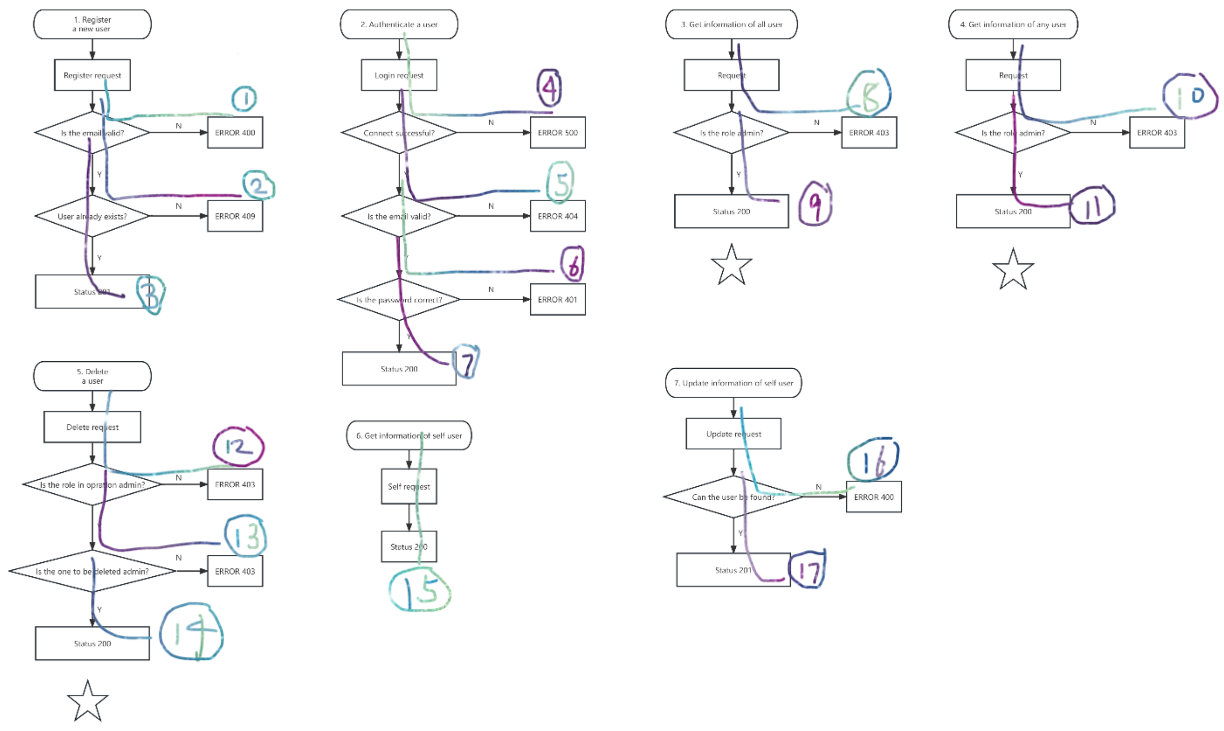




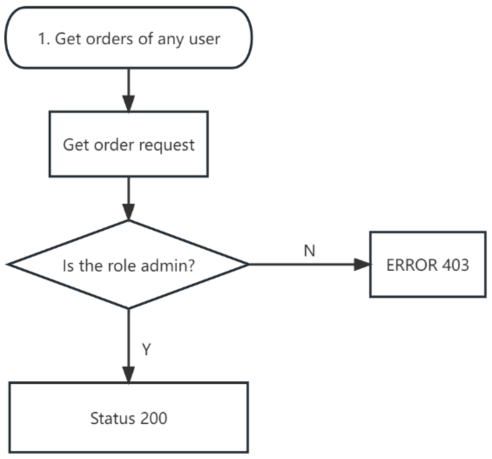

5. Delete a user
Delete request
Is the role in opration admin?
N
ERROR 403
Y
Is the one to be deleted admin?
N
ERROR 403
Y
Status 200


6. Get information of self user
Self request
Status 200

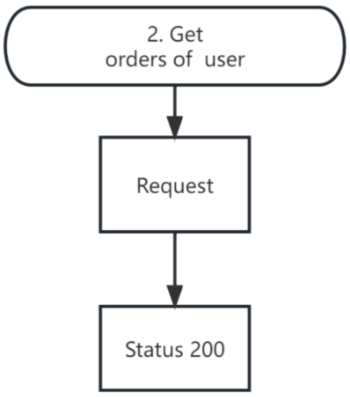

7. Update information of self user
Update request
Can the user be found?
N
ERROR 400
Y
Status 201


1. Register a new user
Register request
Is the email valid?
ERROR 400
User already exists?
ERROR 409
Status 201
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
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

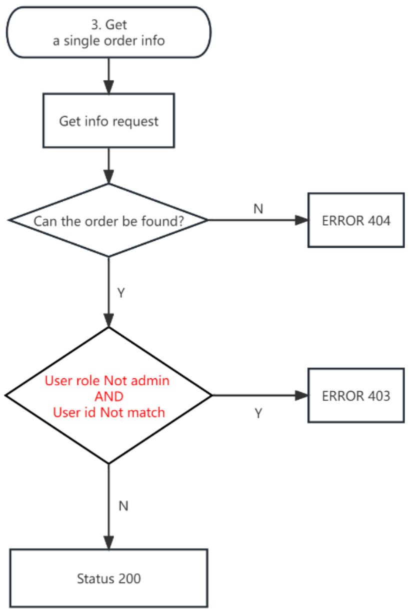
Order.js



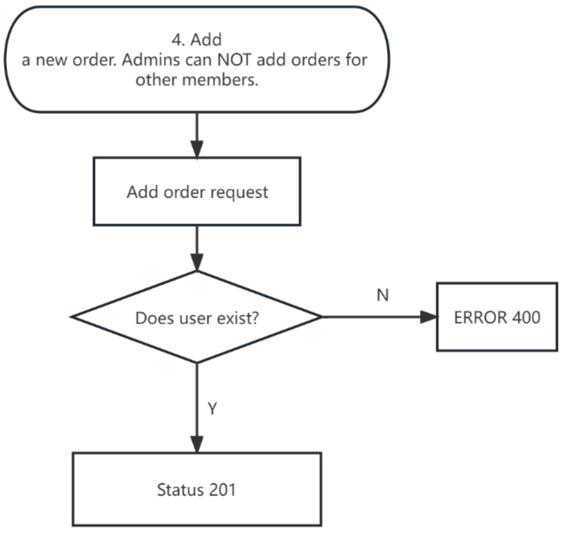
1. Get orders of any user
Get order request
Is the role admin?
N
ERROR 403
Y
Status 200



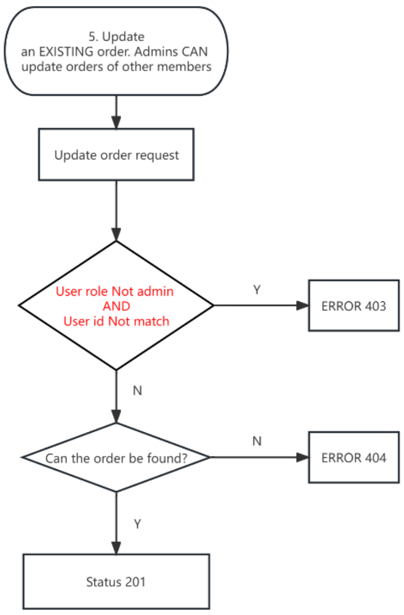
2. Get orders of user
Request
Status 200



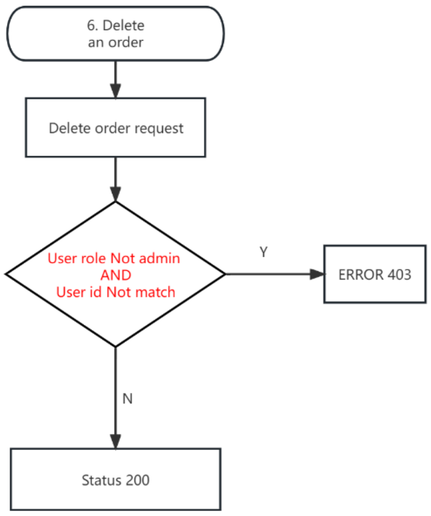
3. Get a single order info
Get info request
Can the order be found?
N
ERROR 404
Y
User role Not admin AND User id Not match
Y
ERROR 403
N
Status 200



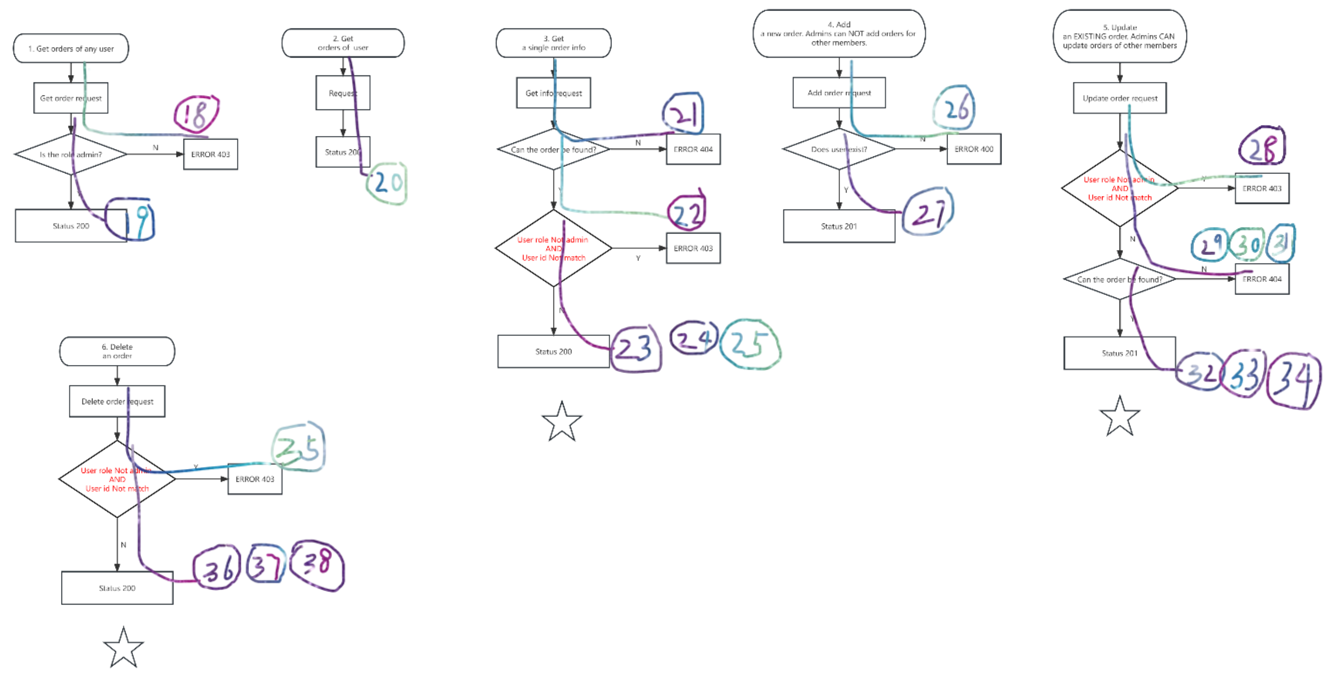
4. Add a new order. Admins can NOT add orders for other members.
Add order request
Does user exist?
N
ERROR 400
Y
Status 201



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

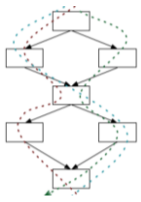


Coverage Analysation:

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.

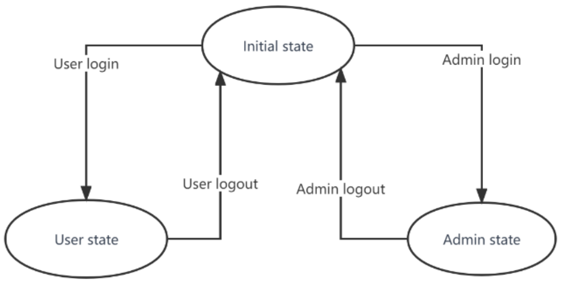
Testing the scalability and performance of the system: Ensuring that the system can handle expected traffic without crashing or slowing down.

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.

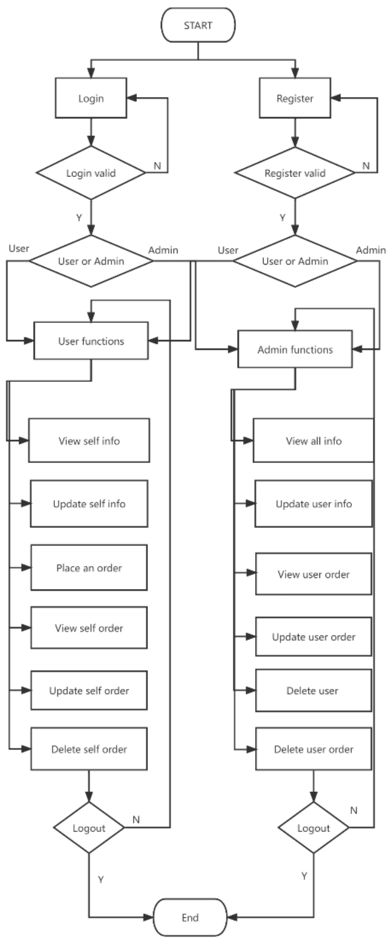
Testing the system's reliability: Ensuring that the system is reliable and that it will perform consistently over time.

Model-based approaches

Finite State Model



Flowgraph Model



Evaluation of the detail of functional, structural, and model-based 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.