LO2 Supporting document

Test plan

Here we define the two requirements we will focus on:

- R1: The system should allow users to get view their orders stored in the database
- R2: The system should be able to handle a high number of requests and process them in a timely manner.

Note that we picked these two requirements as they differ a lot (R1 is a functional requirement, whereas R2 is a measurable quality attribute; R1 has higher A&T needs than R2; R1 requires using/integration testing, whereas R2 requires performance testing). However, what we describe in the test plan for a given requirement can be extended to other requirements. For example, the approaches to testing the requirement “user can place order”, or “user can delete order” will be very similar to R1.

Priority and Pre-requisites

R1 (view orders):

- **High A&T needs** as it is one of the core functionalities of the system and is necessary for the system’s usability and functionality. Thus we should use a reasonably high level of resource to ensure we meet the requirement.
- The appropriate level of quality for the software would be high. This is a **critical feature for the system**, and it is essential that it is reliable and accurate to ensure customer satisfaction and to avoid any potential financial losses.
- **Early detection of issues** will lead to **reduced requirement for A&T efforts** so we should think of **early approaches to V&V**:
    - **Design review:** reviewing the system's design documents such as functional requirements, use case diagrams, and class diagrams to ensure that the system's design meets the requirement and that it is complete, consistent, and traceable. We could do this early in the development process **to identify any issues or inconsistencies in the design before they become a problem** in the later stages of development.

    - **Inspection:** formally inspecting the system's source code, test cases, and documentation to ensure that they are complete, consistent, and traceable to the requirement. We could do this **early in the development process**, before the system is fully functional. By performing **inspections on individual components or partitions of the system**, it can be easier to identify and **isolate any issues** that may arise, and make necessary adjustments before they become a problem in later stages of development.

- We should consider **at least two different T&A approaches** as the requirement has high priority:
    - One approach would be to perform **unit testing** on the individual components that handle the logic for an existing user to view orders, such as user authentication, storage logic (using the Order schema), along with testing that making requests to the API behaves as expected.
    - **Integration testing**: test that these additional components work together correctly. This will include testing authenticating the user and then making a request to the API to retrieve these orders, or for example testing that the User Auth module works correctly with the User schema (i.e. that the User Auth module extracts correctly the user id from the access token).
- The **Sensitivity principle** can be used to test the requirement by **testing the system's response to different inputs** and changes in the environment. This can include testing the system's ability to **handle invalid inputs such as incorrect order details**.
- The **Partition principle** can be used to test the requirement by testing the different components or partitions of the system separately and then testing the interactions between them. This can include testing the authentication logic, testing the database logic, and testing the interactions between the two (as mentioned earlier). By testing each partition separately, it is easier to **identify and isolate any issues that may arise**, and by testing the interactions between the partitions, it is possible to ensure that the system behaves correctly.
- Thus, **here are the tasks we would need to schedule in the plan for R1**:
    1. **Design review:** Review the system's design documents such as functional requirements, use case diagrams, and class diagrams to ensure that the

system's design meets the requirement and that it is complete, consistent, and traceable.

2. **Inspection**: Formally inspect the system's source code, test cases, and documentation to ensure that they are complete, consistent, and traceable to the requirement.
3. **Generate synthetic data** (users and orders)
4. **Unit testing:**
    a. Perform **unit testing on the User auth module** (as in order to view orders, the user needs to be authenticated). This can include:
        i. Testing that the function correctly verifies the token and returns a 410 status code when the token is null or undefined.
        ii. Testing that the function correctly verifies the token and returns a 403 status code when the token is invalid.
        iii. Test that the function correctly extracts the user ID from the token and finds the corresponding user (we would need to create a mock function for “User.findOne()” as we want to isolate the components, so that while doing unit testing for the auth module, we don’t actually use the User model, instead we mock it.
    b. Perform **unit testing on the Order Schema**, which is used to communicate with the database when viewing orders. This can include:
        i. Testing that the Order model can be successfully created and saved to the database
        ii. Testing that each field is set correctly and returns the correct value when queried
        iii. Test that the type field of the Order model is set to the default value of 'Box1' when no value is provided.
        iv. Test that the type field of the Order model only accepts the specified enum values of 'Box1' and 'Box2’
        v. Test that orders omitting non-required fields doesn’t cause any error.

vi. Test that the user field of the Order model is correctly referenced to the User model.

vii. Error handling to test that performing operations (such as getting, updating or deleting) on a non-existing Order model returns null.

viii. Error handling to test that trying to save an Order model with missing required fields returns an error.

ix. Test that there is correct error handling with appropriate messages when providing illegal inputs.

5. **Integration testing:**

   a. Testing the User Auth module with the Order model when querying the “/orders”, along with making sure the correct response status is returned, with the actual relevant data (the right orders for the given user).

   b. Test the User Auth module with the Order model to test that the function correctly extracts the user ID from the token and finds the corresponding user in the database.

   c. Testing that querying the “/orders” route should return an empty array if the user has no orders.

   d. Testing that appropriate errors are returned when not providing an authentication token, or when providing an invalid authentication token.

R2: Performance

- **Moderate A&T needs** as performance is critical for the system's usability and functionality only at a later stage. Currently, **we are developing our MVP so performance is not as critical**. It is however still important to understand the potential bottlenecks of the system so we can think of a better solution as soon as possible before developing too much a part of the system that can cause in the future bad performance.
- **Early detection of issues** will lead to reduced requirement for A&T efforts so we should think of **early approaches to V&V**:
  - **Performance modeling:** This approach involves creating models of the system's performance characteristics and using them to identify potential bottlenecks or scalability issues. We could do this early in the development

process and can help to identify any issues or inconsistencies in the design before they become a problem in later stages of development.

    - **Profiling:** This approach involves measuring the system's performance characteristics during development to identify any issues or inefficiencies in the system's design or implementation. We could do this early in the development process, before the system is fully functional. By performing profiling on individual components or partitions of the system, it can be easier to identify and isolate any performance issues that may arise, and make necessary adjustments before they become a problem in later stages of development.

- One approach to test this requirement would be to **perform load testing on the system to measure its performance characteristics under different levels of load and concurrency**.
- **The Restriction principle** can be used to test the requirement by testing the system's response to different inputs and changes in the environment. This can include testing the system's ability to **handle a high number of concurrent requests**, testing the system's ability to **handle invalid inputs such as malformed requests or unexpected data**, testing the system's **ability to handle exceptional cases such as network failures or hardware failures**, and testing the system's ability to **handle a high number of requests over a long period of time**.
- **The Feedback principle** can be used to test the requirement by testing the system's **ability to provide feedback to the user** and the system administrator in case of any performance issues. This can include testing the system's ability to **provide real-time performance metrics**, **testing the system's ability to provide alerts or notifications** in case of any performance bottlenecks or issues, and testing the system's ability to provide detailed performance reports for system administrators.
- Here are the **tasks we would need to include in our plan:**

    1. **Modeling the system's performance characteristics** and identifying potential bottlenecks or scalability issues
    2. **Measuring the system's performance characteristics** during development to identify any issues or inefficiencies in the system's design or implementation
    3. **Performing load testing** on the system:

- Defining the test environment and test conditions
- Identifying the different types of load that the system needs to be able to handle (e.g. number of concurrent users, number of requests per second, etc.)
- Setting up the test environment to mimic the expected production environment as closely as possible. In our case we use Artillery so we need to configure performance.yml.
- **Run tests:**
    - Running the performance tests and view the system's performance characteristics (e.g. response time, throughput, error rate, etc.)
    - Running the tests and measuring the system's ability to provide detailed performance reports for system administrators
- **Analyzing the results:**
    - of the load tests and identifying any issues or bottlenecks in the system's performance
    - of the tests and identifying any issues or weaknesses in the system's ability to provide detailed performance reports for system administrators

Scaffolding and Instrumentation

R1 (view orders):

- Scaffolding will be required in order to generate synthetic data. Indeed, we need to populate the database with users, some of them having orders, and some having no orders at all. This will allow us to perform various unit and integration tests by setting up an environment.
- Furthermore, mocking is required to mimic some components that are used, in order to isolate the components we are interested to test. For example, the User Auth module queries the User model. But when we are performing unit tests on the User Auth module, we create a mock to simulate the query it makes to the User model, in order to isolate it.
- Additionally, when testing on the User Auth module, where the “next()” function will need to be mocked in order check whether the module correctly extracts the user ID

from the token and finds the corresponding user in the database.

- Integration tests would then not use the mock, so our integration test would see if the User Auth module works correctly with the User model.
- We added code instrumentation by logging successes and errors within the system.

R2: Performance

Note that we are going to be using Artillery for performance testing. However, if doing it ourselves for more control, we would require scaffolding and code instrumentation:

- Create scaffolding code to:
  - generate the load and simulate different user scenarios
  - set up the environment
  - manage the user sessions and maintain state if testing the system with a large number of current users
  - manage and tack requests or transactions
- Code instrumentation:
  - measure response time by adding code tu measure the time before and after the request is made and then calculating the difference
  - measure throughput by adding code to increment a counter every time a request is made and then measuring the value of the counter over a certain period of time
  - measure memory usage, CPU usage or network usage by measuring these values before and after a request

Lifecycle & risks and processes

R1 (view orders):

We are choosing to follow the Agile development lifecycle. The testing for R1, including unit and integration tests, would be part of the development phase. it could be part of the “development and testing” sprints, where the focus would be on implementing additional features while ensuring they work correctly. Generating data using scaffolding would be part of the test data preparation phase, done before the actual testing starts.

Of course, this test data has to be updated and maintained throughout the development cycle. To test this requirement, we would need:

- Code: the code for the feature and the corresponding tests
- Data: the synthetic data that will be used to test the feature, as well as the code for the Order and User models to know how the data has to be structured
- Results from earlier tasks: The requirements and design of the feature, as well as any previous test results, can be used to guide the development of the tests.

As for the synthetic data, unrepresentative synthetic data could lead to poor performance of the design in production. To mitigate this risk, it's important to use a test data generator that creates data that is representative of the real-world data that the system will be handling. Additionally, it's important to validate the synthetic data against the real-world data to ensure that it's representative.

Here are the risks and processes for R1:

1.
    a. Personnel Risk: A staff member is lost or is underqualified for task.
    b. Process: Regularly check in with staff members to ensure they have the necessary skills and resources to complete their tasks. Have a plan in place for replacing staff members in case of unexpected absences.
2.
    a. Development Risk: Poor quality software delivered to testing group or inadequate unit test and analysis before committing to the code base.
    b. Process: Implement a code review process before committing code to the codebase. Have a separate testing group to verify the quality of the software before it is released to production.

R2 (performance):

The testing for R2 would be done during the later stages of the Agile development lifecycle, such as during the testing or staging phase. Indeed, to do performance testing, load testing requires a functional and stage system to test against. It is also important to schedule the performance testing early enough in the development process to allow enough time for any performance issues to be identified and addressed before the

release. As for risk related to the project design and scheduling, it is important to consider the possibility that synthetic data may not be representative of actual usage patterns of the system. This could result in the system performing poorly when it is used in the real world. To mitigate this risk, it is important to use real-world data as much as possible and to test the system in a real-world environment, or as close to it as possible. Additionally, it is important to monitor the system in production to detect any performance issues and address them as soon as possible.

Here are the risks and processes for R2:

1.
    a. Technology Risk: Many faults are introduced interfacing to an unfamiliar commercial off-the-shelf (COTS) component.
    b. Process: Perform thorough testing and evaluation of COTS components before integrating them into the system. Have a plan in place for dealing with faults that are discovered during testing.
2.
    a. Schedule Risk: Difficulty of scheduling meetings makes inspection a bottleneck in development.
    b. Process: Use virtual meeting tools to facilitate collaboration and inspection. Have a dedicated team member responsible for scheduling and coordinating meetings.

Evaluation of Test Plan

There are potential omissions or vulnerabilities of the test plan, such as:

- Not mentioning any testing for security vulnerabilities, such as testing for SQL injection or cross-site scripting attacks. This is important especially that we handle sensitive user information when logging in the user.
- Not mentioning testing for accessibility, such as testing for compliance with accessibility guidelines (such as WCAG).
- Not mentioning browser compatibility

Evaluation of Code Instrumentation and scaffolding

- Instrumentation provided seems sufficient for R2. It would be better to specifically determine which libraries to use for measurement but there isn’t too much that could be added at this stage.
- Regarding scaffolding, the current plan is to generate by hand the various users and orders and to add them in the database. A better approach would be to generate them automatically, with a wide variety of inputs to test every possible scenario.