Summary

In my development of Contact Service, I focused on testing key functionalities like adding, deleting, and updating contacts. This approach helped me keep the requirements by making sure that all edge cases, like handling null inputs or duplicate contacts, were tested. For example, in ‘ContactServiceTest.java’, the test case ‘testAddContact()’ checks for duplicate entries by asserting that the system throws an exception when a duplicate contact is added.

For Task Service, my testing strategy involved both positive and negative tests. This helped make sure that the service handled typical user interactions and potential errors. In the ‘TaskServiceTest.java’ file, the ‘testUpdateTask()’ method validates that tasks are updated correctly, and any attempt to update a non-existent task result in an appropriate exception.

In developing Appointment Service, my testing focused on appointment overlaps and time zone conversions, ensuring the service met the complex scheduling requirements. For example, the ‘testAppointmentOverlap()’ method in ‘AppointmentServiceTest.java’ ensures that no two appointments overlap, meeting the required specifications.

The effectiveness of my JUnit tests is proven by the high coverage percentage, showing that most of the code paths were tested. For example, the overall code coverage was 90%, demonstrating that critical paths and edge cases were sufficiently covered. The code coverage showed 100% method coverage and 85% branch coverage, which shows that the tests are comprehensive and reliable.

I did my best to stick to practices like using meaningful assertions and mocking dependencies to isolate the unit under test. For example, in ‘testUpdateTask()’ , I used ‘assertEquals(expectedTask, actualTask)’ to make sure that the task was updated as expected. This method ensures that the test only passes when the task’s state matches the expected result. Strategically, I aimed to write clear and reusable test methods to reduce redundancy and improve maintainability. For example, the method ‘setupMockTasks()’ is a helper function that initializes common test data, which is reused across multiple test cases, making my code more efficient.

Reflection

For my testing techniques, I used Unit testing and Mocking. Unit testing focuses on individual components by themselves, making sure that each unit functions correctly by itself. This method is great for identifying bugs early in the coding process. Unit testing was crucial for the project because it allowed me to check the core functionalities of each service independently. Mocking lets you simulate complex dependencies by replacing them with controlled and predictable behavior. This technique helps to make sure that tests remain focused on the unit that is being tested. I used mocking throughout the Task Service tests to isolate the task operations from external dependencies.

A couple of testing techniques that I did not use Integration testing and Exploratory testing. Integration testing focuses on the interactions between integrated units or components to ensure that they work together as expected. It is especially useful in identifying issues related to the interfaces between components. In future projects that involve complex interactions between multiple services, integration testing will be essential. Exploratory testing involves testing without predefined test cases, relying on the tester’s intuition and experience to identify defects. It is useful for uncovering unexpected issues that automated tests could miss. This technique would be great for scenarios where the software’s behavior is not entirely predictable, like in a new or fast evolving application.

Throughout this project, I feel I used a considerably cautious mindset, understanding the importance of thoroughly testing each unit before integration. Given the complexity of the task service, this was very important to avoid failures. For example, I carefully reviewed the task creation logic, making sure that even rare edge cases, like tasks scheduled for leap years, were handled correctly. To limit bias, I used code reviews with paired testing sessions. This helped me gain a different perspective and catch issues I might have overlooked. During the review of the Task Service tests, I asked a friend of mine, who works as a professional software developer, to challenge my approach about task priority handling, which lead to identifying a small bug that I had initially missed. Maintaining discipline in testing is crucial to keeping up long term code quality and maintainability. Cutting corners in testing can lead to technical debt, making future maintenance more difficult and prone to errors. I plan to avoid technical debt by sticking to a strict code review process and continually refactoring test to improve readability and effectiveness. For example, during this project, I refactored the ‘TaskServiceTest.java’ file multiple times to make sure it was comprehensive and maintainable.