Cargo Connect: Improving Airport Freight Flow

Test Documentation vs 2.0

SE 6387 Advanced Software Engineering Project

R.Z. Wenkstern

Date : 05/09/2025

Group *X*
FNU ANUJA
BRADLEY EVAN STOVER
CHIJIOKE ELISHA-WIGWE
SMIT SONESHBHAI PATEL

Revision History

Version	Date	Decription	Authors
1.0	03/25/25	Initial draft completed	FA, CE, BS, SP
2.0	05/09/25	Final draft completed	FA, CE, BS, SP

Contents

1. Organizational Test Process Documentation

1.1 Test Policy

Our goal is to ensure Cargo Connect delivers reliable, scalable, and user-friendly freight scheduling services with minimal downtime. Cargo Connect adopts a quality-driven testing policy that aims to validate all critical functional and non-functional components of the system before deployment. The goal is to ensure automated freight scheduling, mobile interactions, and third-party integrations operate seamlessly.

1.2 Organizational Test Strategy

1. **Unit Testing**: Done for all backend services (Flask) and mobile app modules (Kotlin).
2. **Integration Testing**: Between backend and PostgreSQL, backend, and third-party APIs (DALI, Airport System).
3. **Acceptance Testing**: Based on user stories and UC1 (Schedule trucks to airport).
   Tools: Postman, Android Studio Unit Testing framework, and manual test execution.

2. Test Management Process Documentation

2.1 Test Plan (including a Test Strategy)

The testing scope for Cargo Connect encompasses end-to-end validation of all critical system functionalities and interactions:

- Scheduling: Ensuring trucks are scheduled to, from, and within airport zones based on cargo availability and dock readiness.
- Data Integration: Validating the flow of data between internal modules and external systems like the Airport System, DALI.
- Mobile Updates: Ensuring the mobile app correctly displays updated truck status, routing changes, and dock reassignments in real time.
- Traffic Rerouting: Simulating various traffic conditions and ensuring DALI integration provides intelligent, adaptive routing suggestions.

Test Types

1. **Unit Testing**

- Conducted on individual modules like scheduling logic, JWT token validation, database access, and mobile app components.
- Tools: Pytest for Flask backend, built-in Android testing tools for Kotlin code.

2. **Integration Testing**

- Tests communication between:
    - Backend ↔ PostgreSQL
    - Backend ↔ Airport APIs
    - Backend ↔ Google APIs
    - Mobile App ↔ Backend API

3. **System Testing**

- Validates end-to-end workflows like UC1:
    - Scheduler triggers scheduling
    - Cargo availability is checked.
    - Dock assignment and rerouting if needed.
    - Mobile app receives updates.

4. **Acceptance Testing**

- Tests whether the system meets business requirements and user expectations.

2.2 Test Status Report

- Summary of test progress at any point during the testing lifecycle.
- Number of tests planned, executed, passed/failed, open defects, etc.

2.3 Test Completion Report

The test completion report will summarize:

- All passed test cases (UC1 functionality verified).
- Integration with DALI, and Airport System successful.
- No high/critical issues pending.
- System ready for deployment.

3. Dynamic Test Process Documentation

3.1 Test Design Specification

Designs will validate:

- Correct scheduling based on flight timing.
- Dock assignment and reassignment logic.
- DALI sending instructions behavior.
- Real-time push notifications via OneSignal.
- Mobile app updates.

3.2 Test Case Specification

Test Case 1 : Schedule truck and send the details to the driver via oneSignal.

Test Case 2 : If no cargo available, log the event and no further actions

Test Case 3 : If there is a dock congestion, assign a parking lot. Dock will be reassigned once available.

Test Case 4 : Send traffic notifications to the truck driver.

3.3 Test Procedure Specification

- Start Flask server.
- Trigger scheduling via API.
- Simulate cargo info fetch from Airport System.
- Observe OneSignal notifications.
- Simulate DALI traffic events.
- Verify real-time updates on mobile.

3.4 Test Data Requirements

Effective testing of Cargo Connect requires carefully prepared and representative test data that simulates real-world scenarios. The data should cover all system interactions and edge cases related to truck scheduling, airport conditions, traffic inputs, and mobile updates.

Below is a detailed description of the necessary test data categories:

1. Truck Data

- **Truck ID**: Unique identifiers for each truck used to schedule, track, and log operations.
- **Location**: GPS coordinates or location zones indicating the current position of the truck (e.g., en route, at terminal, near dock).
- **Availability Status**: Whether the truck is currently available for scheduling or in transit/loading.

2. Airport Dock Status

- **Dock ID**: Each dock at the airport has a unique identifier.
- **Status**: Whether the dock is free, occupied, or reserved.
- **Expected Release Time**: If occupied, this indicates when it will become available.

Use Case: This data is used to assign a truck to a dock or reroute it to temporary parking.

3. Flight Schedules

- **Flight Number**: Identifies which flight a truck may be associated with.
- **Arrival/Departure Times**: Required to align freight delivery and pickup accurately.
- **Cargo Ready Time**: Time when freight becomes available for loading/unloading.

Use Case: Flight schedule data is used by the scheduler to coordinate truck timing and avoid delays or congestion.

4. Simulated Traffic Events (DALI Integration)

- **Event Type**: E.g., accident, roadblock, green_light, reroute_needed.
- **Location**: Affected coordinates or area.
- **Severity/Instruction**: Actionable data such as "slow down", "rerouting", or "maintain 40 km/h".

Use Case: This helps validate the DALI system's ability to provide real-time routing instructions to truck drivers.

3.5 Test Data Readiness Report

All test data present in PostgreSQL test schema. Mock services for Airport API and DALI are responsive. All test data including mocks for Airport System APIs are available and yet to be validated. I am using mocked data to run and check my test cases.

3.6 Test Environment Requirements

- Local server for Flask.
- PostgreSQL DB.
- Android emulator/device.
- Internet access for OneSignal and external APIs.
- Postman can be used for manually triggering and testing backend endpoints like scheduling, truck updates, and notifications.
- Mock API Servers - To simulate the behavior of third-party APIs (Airport System) when the actual services are unavailable or unstable.

3.7 Test Environment Readiness Report

Environment setup is complete. Flask app and mobile app are running. All APIs are reachable, and logs configured.

3.8 Actual Results

Actual outputs from test execution were recorded and matched with expected results with few of the test cases. We are still not able to push all kind of notifications from DALI system by tracking real time data from the truck.
Truck schedule details is being sent to the driver and the driver can receive it and start the navigation.

3.9 Test Result

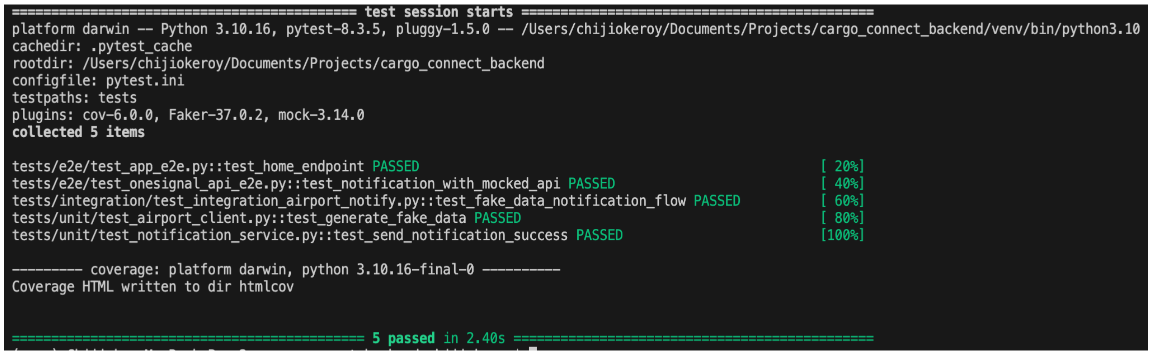
Below are the screenshots of test cases code and results attached.

Few testcases code snippets.

```
@RunWith(MockitoJUnitRunner::class)
class MainActivityTest {

    @Mock
    lateinit var mockContext: Context

    @Mock
    lateinit var mockNotificationClickEvent: INotificationClickEvent

    @Mock
    lateinit var mockPayload: JSONObject

    @Mock
    lateinit var mockIntent: Intent

    @Test
    fun testNotificationClick() {
        // Prepare mock data
        `when`(mockNotificationClickEvent.notification.additionalData).thenReturn(mockPayload)
        `when`(mockPayload.optString("message")).thenReturn("Test Message")
        `when`(mockPayload.optString("latitude")).thenReturn("32.9857")
        `when`(mockPayload.optString("longitude")).thenReturn("96.7502")

        // Initialize MainActivity
        val mainActivity = MainActivity()

        // Simulate the notification click
        mainActivity.onCreate(null)
        mainActivity.openEventHandler.onClick(mockNotificationClickEvent)

        // Verify that the intent is correctly created with the expected parameters
        val expectedUri = "http://maps.google.com/maps?saddr=32.9857,96.7502&daddr=32.9857,96.7502"
        verify(mockContext, times(1)).startActivity(
            argThat { it.action == Intent.ACTION_VIEW && it.data.toString() == expectedUri }
        )
    }
}
```

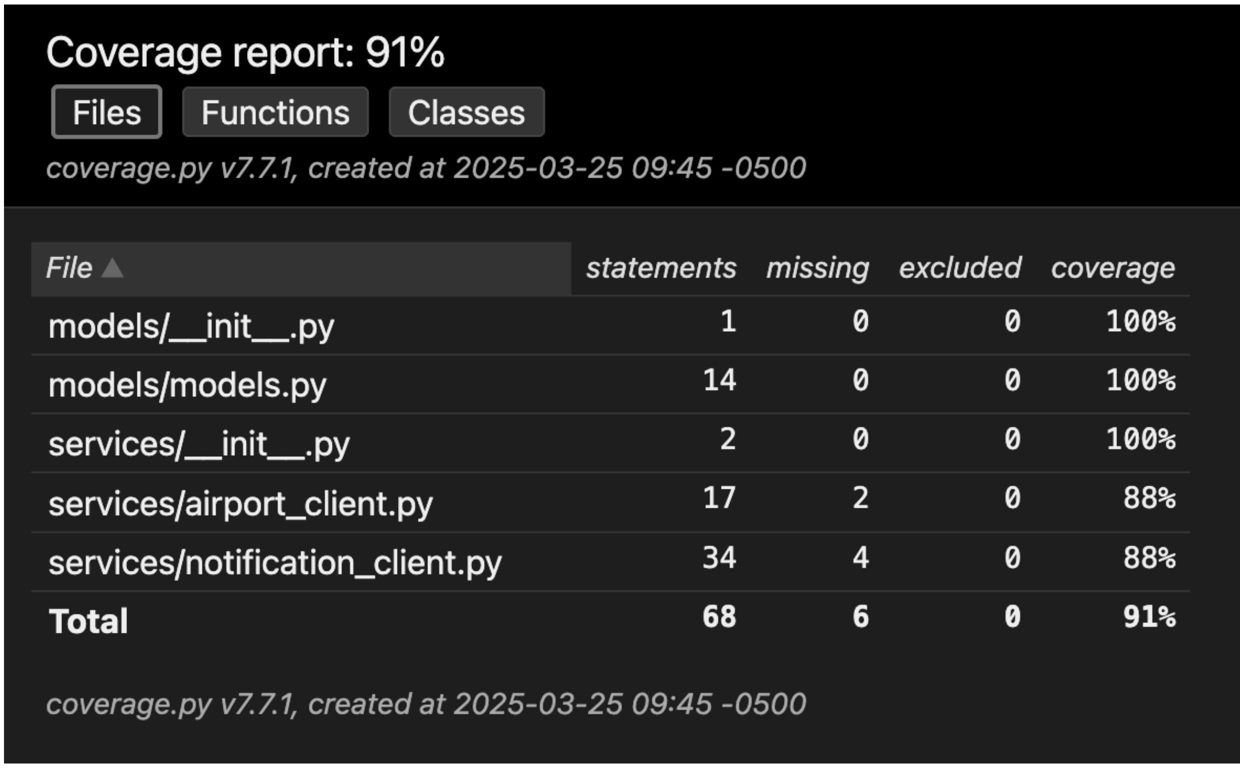
```
@RunWith(MockitoJUnitRunner::class)
class MainActivityTest {

    @Mock
    lateinit var mockContext: Context

    @Mock
    lateinit var mockNotificationClickEvent: INotificationClickEvent

    @Mock
    lateinit var mockPayload: JSONObject

    @Test
    fun testStartActivityWithCorrectIntentOnNotificationClick() {
        // Set up mock data
        val latitude = "32.9857"
        val longitude = "96.7502"
        `when`(mockNotificationClickEvent.notification.additionalData).thenReturn(mockPayload)
        `when`(mockPayload.optString("latitude")).thenReturn(latitude)
        `when`(mockPayload.optString("longitude")).thenReturn(longitude)

        // Prepare the intent that we want to verify
        val expectedUri = "http://maps.google.com/maps?saddr=32.9857,96.7502&daddr=$latitude,$longitude"

        val mainActivity = MainActivity()
        mainActivity.onCreate(null)
        mainActivity.openEventHandler.onClick(mockNotificationClickEvent)

        // Verify that startActivity was called with the right intent URI
        val argumentCaptor = ArgumentCaptor.forClass(Intent::class.java)
        verify(mockContext).startActivity(argumentCaptor.capture())
        val capturedIntent = argumentCaptor.value
        assertEquals(expectedUri, capturedIntent.data.toString())
    }
}
```

- Results from Unit, Integration, and E2E testing.



```
============================================ test session starts =============================================
platform darwin -- Python 3.10.16, pytest-8.3.5, pluggy-1.5.0 -- /Users/chijiokeroy/Documents/Projects/cargo_connect_backend/venv/bin/python3.10
cachedir: .pytest_cache
rootdir: /Users/chijiokeroy/Documents/Projects/cargo_connect_backend
configfile: pytest.ini
testpaths: tests
plugins: cov-6.0.0, Faker-37.0.2, mock-3.14.0
collected 5 items

tests/e2e/test_app_e2e.py::test_home_endpoint PASSED                                                   [ 20%]
tests/e2e/test_onesignal_api_e2e.py::test_notification_with_mocked_api PASSED                          [ 40%]
tests/integration/test_integration_airport_notify.py::test_fake_data_notification_flow PASSED          [ 60%]
tests/unit/test_airport_client.py::test_generate_fake_data PASSED                                      [ 80%]
tests/unit/test_notification_service.py::test_send_notification_success PASSED                         [100%]

--------- coverage: platform darwin, python 3.10.16-final-0 ----------
Coverage HTML written to dir htmlcov


============================================= 5 passed in 2.40s ==============================================
```

- 90%+ test coverage



Coverage report: 91%

coverage.py v7.7.1, created at 2025-03-25 09:45 -0500

File ▲	statements	missing	excluded	coverage
models/__init__.py	1	0	0	100%
models/models.py	14	0	0	100%
services/__init__.py	2	0	0	100%
services/airport_client.py	17	2	0	88%
services/notification_client.py	34	4	0	88%
Total	**68**	**6**	**0**	**91%**

coverage.py v7.7.1, created at 2025-03-25 09:45 -0500

The first three test cases mentioned above are working well, but the test case where DALI needs to send few contextual information to the app when the driver is on the way in not being shown for now.

3.10 Test Incident Report

1. Notification not received due to delay.

2. Airport System returned malformed data.

3. Triggering the query with periodic time delay was not successful sometimes.

Appendix A: Glossary

Term	Definition
Cargo Info	Includes truck ID, dock, ETA, and cargo availability
DALI	Traffic analysis system used for routing.
Time	Component that triggers time-based scheduling queries.
UC 1	Main use case - Schedule trucks to the airport.

Appendix B: References

- ISO/IEC/IEEE 29119 Test Documentation, 2013
- High Level design document VS 1.0
- Detail design document VS 1.0
- Requirement Analysis VS 2.0