Testing Strategy and Quality Assurance

Target Audience: QA Engineers, Developers
Purpose: Defining the pyramids of testing required to ship code with confidence and minimize regressions.

---

1. The Testing Pyramid

We adhere to the standard industry pyramid:

- **70% Unit Tests:** Fast, isolated, checking logic.
- **20% Integration Tests:** Checking boundaries (Service + DB, Service + RabbitMQ).
- **10% End-to-End (E2E) Tests:** Slow, checking full user workflows (Browser/App).

---

2. Unit Testing (The Foundation)

- **Scope:** Single functions, classes, or small modules.
- **Dependencies:** All external dependencies (DB, API, network) MUST be **Mocked**.
- **Frameworks:** `pytest` (Python), `Jest` (JS/TS).
- **Goal:** Validate business logic branches (If/Else).
- **Example:**
  - *Input:* `calculate_fare(distance=100km, type=luxury)`
  - *Assert:* Returns `$50`.
- **Speed:** Should run in milliseconds.

---

3. Integration Testing (The Glue)

- **Scope:** A single Service + its Infrastructure (DB, Cache).
- **Dependencies:**
  - **Database:** Accesses a specific "Test DB" (Spun up via Docker Container).
  - **External APIs:** Mocked (using tools like `WireMock` or `pytest-vcr`) to ensure deterministic results.
- **Goal:** Ensure SQL queries are correct and API serialization works.
- **Example:**

- *Action:* `POST /api/bookings`
- *Assert:* Record is inserted into Postgres DB with correct status.

---

4. End-to-End (E2E) Testing (The User Reality)

- **Scope:** The entire system black-box.
- **Tools:** Playwright or Cypress.
- **Environment:** Runs against a transient "Staging-like" environment.
- **Goal:** Verify critical "Happy Paths".
- **Example (The "Booking Flow"):**
  1. Open Browser.
  2. Search for "Kigali to Huye".
  3. Select Seat 4.
  4. Checkout.
  5. Assert: "Ticket Confirmed" screen appears.

---

5. Test Data Strategy

"Tests are only as good as their data."

- **Unit/Integration:** Use **Factories** (e.g., `FactoryBoy`, `Faker`) to generate randomized, valid data on the fly. Do not rely on a shared static database state, as parallel tests will corrupt each other.
- **E2E:** Use **Seeding**. Before the test suite runs, a script populates the database with "Golden Data" (Standard Routes, Standard Admin User).
- **Cleanup:** Tests must be transactional. They roll back their changes or truncate tables after execution.

---

6. Testing Failure Scenarios (Chaos)

We don't just test success; we test survival.

- **Network Timeouts:** Configure the API Client Mock to hang for 30 seconds. Does the app crash or show a "Retry" button?
- **Bad Data:** Send malformed JSON. Does the API return `500 Server Error` (Bad) or `400 Bad Request` (Good)?

- **Concurrency:** Use tools like `k6` to simulate 100 users booking the same seat. Ensure only 1 succeeds.

---

7. Continuous Integration (CI) Expectations

Tests are useless if they aren't run.

- **Trigger:** Every Push to any branch.
- **Gate:** Merging to `main` is **Blocked** if any test fails.
- **Coverage:** We aim for >80% Code Coverage. Codecov reports drops in coverage as failures.
- **Performance:** The full test suite must run in <10 minutes. If slower, we embrace parallel execution (sharding coverage).

---

8. Mobile & Desktop Specifics

- **Mobile (React Native):** Use `Detox` or `Appium` for on-device testing.
- **Desktop (Avalonia):** Use `Appium` with WinAppDriver for Windows UI testing.
- **Offline Mode Testing:** Specifically switch off the network adapter during a test to verify the "Queue & Sync" logic of the POS app.