Testing Strategy for the Ahara Food Redistribution Platform

Abstract

The Ahara Food Redistribution Platform is a multi-role mobile application connecting food donors, recipients, and volunteer delivery partners across Android, iOS, and Web platforms. The system manages listing creation, order processing, volunteer coordination, and payment handling using a Node.js backend and MongoDB database. This document presents a structured testing strategy aligned with the system architecture. The approach integrates unit testing, integration testing, end-to-end validation, performance benchmarking, and security verification within a CI/CD pipeline to ensure reliability and scalability.

Keywords

Software Testing, Flutter, Node.js, CI/CD, End-to-End Testing, Performance Testing, Security Testing

I. Introduction

The Ahara platform supports interactions between Sellers, Buyers, and Volunteers. Due to cross-role workflows and cross-platform deployment, a layered and systematic testing strategy is required to ensure correctness, stability, and security.

II. Testing Objectives

The testing strategy aims to:

- Ensure correctness of multi-role workflows.
- Validate frontend and backend integration.
- Detect defects early using automation.
- Prevent regression during development.
- Validate performance and responsiveness.
- Enforce authentication and role-based access control.

III. Architecture-Aligned Testing

A. User Interface Layer (Flutter)

- Widget testing for UI components.
- Integration testing for user flows.
- Cross-platform validation (Android, iOS, Web).
- Responsive design testing.

B. State Management Layer (Provider)

- Unit testing of ChangeNotifier providers.
- State mutation validation.
- Dependency injection testing.

C. Backend API Layer (Node.js/Express)

- Unit testing of controllers and services.
- Integration testing of REST endpoints.
- Database transaction validation.

D. Data Layer (MongoDB)

- Schema validation.
- Relationship integrity verification.
- Transaction rollback testing.

IV. Testing Levels

A. Unit Testing

Frontend unit testing uses flutter_test with Mockito. Backend unit testing uses Jest to validate controllers and business logic.

B. Integration Testing

Integration tests validate workflows such as Seller listing management, Buyer order placement, and Volunteer delivery confirmation. Backend integration tests use Jest and Supertest.

C. API Testing

API testing validates REST endpoint correctness, contract compliance, and multi-endpoint workflows.

- Route-level validation using Supertest.
- API contract validation using Swagger/OpenAPI.
- Workflow validation using Postman collections executed via Newman CLI.
- Load testing of endpoints using Artillery.

D. End-to-End Testing

Complete user journeys from login to transaction completion are validated using Flutter's integration_test framework on emulators and real devices.

E. Performance Testing

UI performance is measured using Flutter DevTools. Backend scalability is evaluated using load testing tools such as Artillery or k6.

F. Security Testing

Security testing includes authentication validation, role-based authorization enforcement, injection prevention, and rate-limiting verification.

V. Test Automation Strategy

A. Continuous Integration

Backend and frontend tests are executed automatically on every commit.

B. Pre-Release Validation

All integration tests are executed before merging to the main branch. Coverage verification and static analysis are performed.

C. Nightly Build

Performance profiling and backend load testing are conducted during nightly validation.

D. Manual Testing Gates

Cross-device emulator testing, real-device validation, and User Acceptance Testing are performed before release.

VI. Test Data Management

A dedicated test database is used for backend validation. Automated seeding ensures reproducibility. Frontend tests use mocked API responses and Firebase Emulator Suite for authentication testing. Unique identifiers and transaction rollbacks maintain isolation.

VII. Release Criteria

Production readiness requires:

- 100% pass rate for critical tests.
- Minimum 70% coverage for critical modules.
- Successful end-to-end validation of all roles.
- Performance benchmarks within defined limits.
- Security enforcement verification.

VIII. Ahara Testing Technology Stack

A. Backend Testing Stack

1. Unit Testing

- Tool: Jest (v30.2.0)
- Purpose: Controller and utility function testing
- Command: npm test

2. API Testing

- Tool: Supertest
  - Purpose: HTTP request and response validation
- Tool: Swagger/OpenAPI (v3.0)
  - Purpose: API contract documentation and schema validation
- Tool: Postman Collections + Newman CLI
  - Purpose: Multi-endpoint API workflow testing
- Tool: Artillery
  - Purpose: API load and stress testing

3. Mocking

- Tool: node-mocks-http (v1.17.2)
- Purpose: Mock HTTP request and response objects

4. Test Database

- Tool: MongoDB Memory Server (v11.0.1)
- Purpose: In-memory isolated MongoDB instance

5. Environment Configuration

- Tool: cross-env (v10.1.0)
- Purpose: NODE_ENV=test configuration

6. Backend Framework Under Test

- Express (v5.2.1)
- MongoDB 6.x with Mongoose (v9.1.6)
- Test Database: ahara_test

Current Status: 25 / 25 tests passing (100%)

B. Frontend Testing Stack

- flutter_test – Unit testing
- Mockito (v5.x) – Mocking
- integration_test – End-to-end workflows
- Provider (v6.x) – State management testing
- Firebase Emulator (v12.x) – Auth testing
- flutter_localizations – Localization validation

- Flutter 3.x – Application framework

Current Status: Tests can be restructured and re-integrated.

C. Device and Platform Testing
- Android: API 28+ (Emulator ready)
- iOS: iOS 14+ (Simulator ready)
- Web: Chrome and Safari supported

D. Performance and Load Testing
- Flutter DevTools – Profiling
- Artillery – artillery run backend/load_test.yml
- k6 – k6 run script.js

E. CI/CD and Infrastructure
- GitHub with Git – Version control
- GitHub Actions – CI pipeline
- Gradle 8.11.1 – Android build tool
- VS Code – Development environment
- npm and pub – Package managers

IX. Conclusion
The Ahara testing strategy adopts a layered and automation-driven methodology aligned with system architecture. By integrating functional, performance, and security testing within CI/CD pipelines, the approach ensures production readiness, reliability, and maintainability.

DevOps Strategy for the Ahara Platform

I. Objective
The objective of the DevOps strategy is to automate the continuous integration and deployment of the Ahara backend to a production AWS EC2 instance. The strategy ensures that only validated and tested code is deployed, reducing manual intervention and minimizing production risks.

II. CI/CD Architecture Overview
The deployment pipeline follows a Continuous Integration and Continuous Deployment (CI/CD) model.

Architecture Components:

- Source Control: GitHub Repository (main branch)
- Continuous Integration: GitHub Actions (runs automated tests)
- Continuous Deployment: GitHub Actions (SSH-based remote deployment)
- Production Server: AWS EC2 (Ubuntu / Amazon Linux)
- Runtime Environment: Node.js (v18+) with PM2 process manager

Deployment Flow:

1. Developer pushes code to the main branch.
2. GitHub Actions triggers the CI workflow.
3. Backend tests (npm test) are executed.
4. If all tests pass, the deployment job is triggered.
5. GitHub Actions connects to the EC2 instance via SSH.
6. The latest code is pulled and the backend service is restarted using PM2.

III. Server Environment Configuration

A. EC2 Instance Requirements

- Operating System: Ubuntu 20.04 / 22.04 LTS (Recommended)
- Security Group Configuration:
  - Allow SSH (Port 22)
  - Allow Application Port (3000 or 5000)

B. Software Installed on EC2

- Node.js (v18+)
- npm
- Git
- PM2 (installed globally using npm install -g pm2)

PM2 ensures the backend process remains active and automatically restarts if it crashes.

C. SSH Access

- Private key (.pem file)
- Proper file permissions configured for SSH access

D. GitHub Repository Secrets

The following secrets must be configured in GitHub:

- EC2_HOST – Public IP of the EC2 instance
- EC2_USER – ubuntu (or ec2-user)
- EC2_SSH_KEY – Private key content of the .pem file

These secrets allow secure authentication during deployment.

IV. GitHub Actions Pipeline Design

The pipeline is defined in:

.github/workflows/main.yml

It consists of two primary jobs:

A. Backend Test Job (Continuous Integration)

Purpose:

- Execute npm test
- Ensure failing code is never deployed

Behavior:

- Triggered on push to main
- Must pass before deployment job executes

B. Deployment Job (Continuous Deployment)

Triggered only if:

- Backend tests pass successfully
- Code is pushed to the main branch

Deployment uses the appleboy/ssh-action GitHub Action to securely execute remote commands.

V. Deployment Commands Executed on EC2

The following commands are executed remotely:

cd /home/ubuntu/Ahara/backend git pull origin main npm install –production pm2 restart ahara-backend || pm2 start server.js –name ahara-backend

Command Explanation:

- git pull origin main – Fetch latest production code
- npm install –production – Install only production dependencies
- pm2 restart ahara-backend – Restart running service
- pm2 start server.js –name ahara-backend – Start service if not running

VI. Deployment Safeguards

The DevOps pipeline enforces:

- Test-first deployment (CI must pass)
- Deployment only from main branch
- Encrypted secret-based SSH authentication
- Automatic process management using PM2

VII. Benefits of the DevOps Strategy

- Fully automated backend deployment
- Reduced manual configuration errors
- Faster production release cycles
- Improved system reliability
- Scalable production infrastructure
- Production-ready CI/CD pipeline architecture