BTLS

Snap Vision

Testing Policy Document

Demo 4

Table of Contents

1. Introduction

This testing policy outlines the testing strategy for the Snap Vision React Native app. The policy covers functional testing (unit, integration, and E2E testing), CI/CD automation, and non-functional aspects such as security, performance, and latency. Adherence to this policy guarantees consistent code quality, predictable deployments, and a reliable user experience.

2. Overview

1. **Jest**
   Tests both the frontend (React Native app in snap-vision) and the backend (Express.js app in snap-vision-backend)due to its compatibility with JavaScript/TypeScript, mocking capabilities, and widespread adoption in the React Native ecosystem.

2. **React Testing Library**
   Renders components and simulates user interactions in frontend tests.

3. **Supertest**
   Tests Express.js API endpoints through HTTP requests.

4. **GitHub Actions**
   Automates test execution on pull requests and merges, ensuring CI/CD compliance and simplifies CI/CD without third-party dependencies.

5. **CodeCov**
   Monitors code coverage for simple integration with GitHub Actions, reliable coverage reporting, and good documentation.

3. Testing Processes

3.1. Running Tests

- To executes all tests locally :

```
npm test
```

- CI/CD: Tests are automatically run on every pull request and merge to dev or main branches via GitHub Actions.
- To generate a code coverage report :

```
npm run test:coverage
```

3.2. Review Approval

- All code reviews must verify that test coverage meets the project's minimum threshold.
- Tests must pass successfully before merging any branch into dev or main.

3.3. CI/CD Testing Workflow

Automation:
GitHub Actions handles automated testing on all pull requests and merges.

Workflow Steps:

1. Run unit tests (frontend & backend)
2. Run integration tests and end-to-end scenarios
3. Execute security tests (Firestore rules, role-based access)
4. Check code coverage thresholds
5. Perform linting and code quality checks
6. Build verification for deployment readiness

3.3. Continuous Improvement

- Regularly evaluate the performance and effectiveness of the test suite.
- Incorporate lessons learned from bugs, failures, and user feedback to improve testing strategies.
- Update the testing policy and workflow to reflect project evolution, new features, and technology updates.

3. Functional Testing

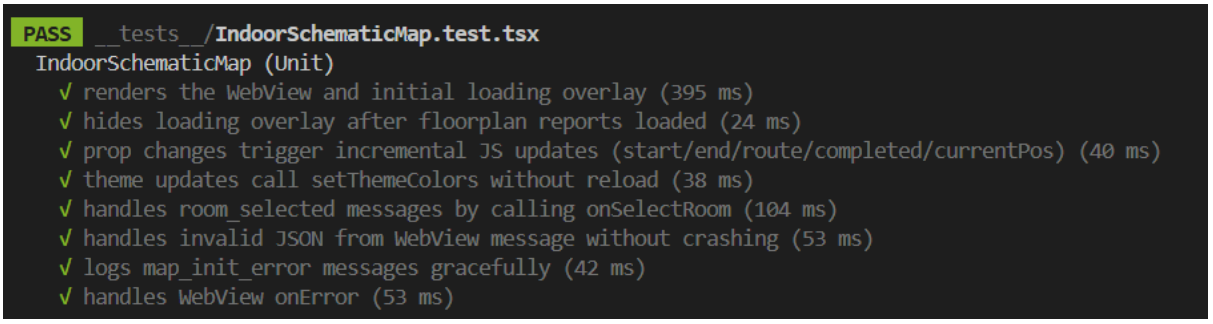
3.1. Frontend Testing

3.1.1 Unit Testing

Unit tests isolate individual components, hooks, and utilities while mocking external dependencies such as Firebase and navigation. This ensures that each component functions correctly in isolation.

Tools:

- Jest: Framework for running tests and mocking.
- React Testing Library: For querying, rendering, and interacting with React components in a way that simulates real user behavior.

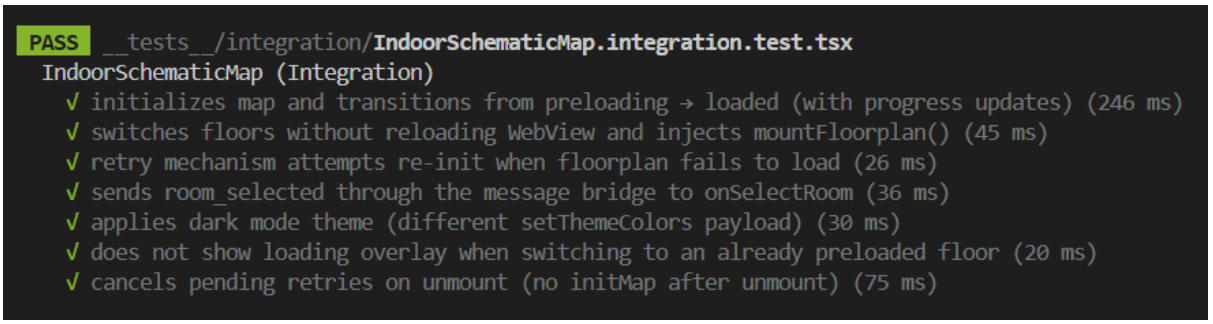


3.1.2 Integration Testing

Integration tests validate interactions between components and user flows without mocking dependencies. Full contexts such as ThemeProvider and NavigationContainer are used to simulate realistic scenarios.

Tools:

- Jest with React Testing Library for testing assemblies of components and their interactions.



3.2. Backend Testing

3.2.1 Overview

The Snap Vision backend is built with Express.js and uses the Firebase Admin SDK for authentication and Firestore interactions. Testing focuses on API endpoints, middleware, and security rules. External services like OpenRouteService are mocked to isolate functionality.

3.2.2 Unit Testing

Unit tests isolate individual server routes and functions, mocking external dependencies such as Axios to test logic without relying on network calls.

Tools :

- Jest: For test execution, assertions, and mocking.
- Supertest: For simulating HTTP requests to the Express server.

```
PASS __tests__/unit/index.unit.test.js
  index.js (unit)
    √ GET / → 200 with running message (67 ms)
    GET /api/directions
      √ returns 400 if start or end missing (13 ms)
      √ calls axios and returns geojson on success (8 ms)
      √ returns 500 if axios throws (15 ms)
```

3.2.3

Integration Testing

Integration tests run the full Express server, simulating real HTTP requests and verifying end-to-end behavior, including Firestore cleanup. External APIs are mocked, and Firebase emulators are configured for local testing.

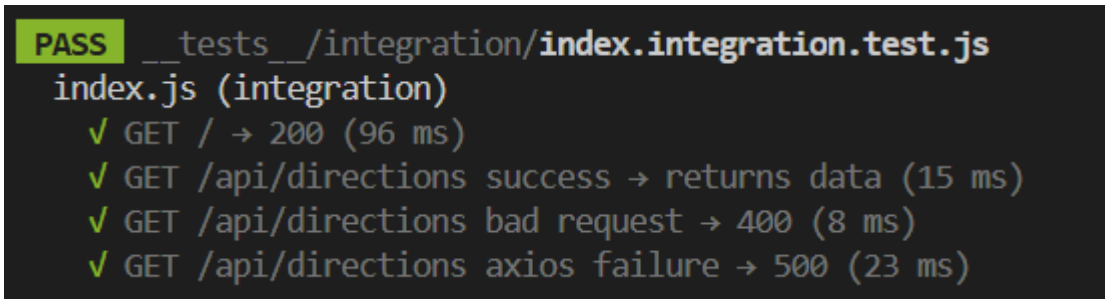
Tools :

- Jest with Supertest: For HTTP request testing against the running server.
- Axios mocks: To simulate from external services responses OpenRouteService.
- Firebase Admin SDK: For Firestore operations in tests.

3.3. Implementation

- Tests are written concurrently with feature development to ensure immediate verification of functionality.
- Test files are located in __tests__/ directories within the respective modules.

- The project aims for ≥80% code coverage, tracked via CodeCov for critical files such as components, services, and hooks.
- CI/CD automation using GitHub Actions ensures that all tests run on pull requests and pushes to dev and main. Workflows enforce passing tests and minimum coverage thresholds.



Repository Access:
https://github.com/COS301-SE-2025/Snap-Vision/tree/main/snap-vision/__tests__

4. Non-Functional Testing

4.1. Overview

Non-functional testing ensures that the Snap Vision app meets requirements beyond functional correctness, including security, performance, latency, and usability. These tests help maintain reliability, responsiveness, and a safe user experience under various conditions.

4.2. Security Testing

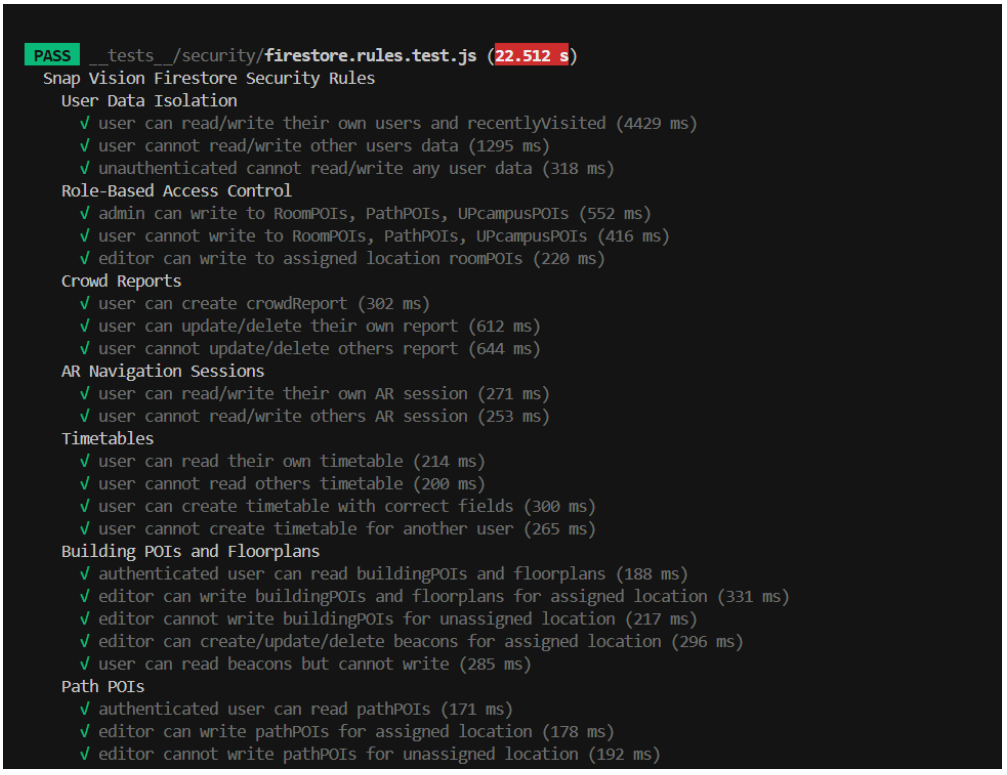
Security tests focus on protecting data and enforcing role-based access.

Key examples include:

- Verifying admin-only operations
- Ensuring user data isolation
- Testing authentication flows

Tools:

- Firebase Rules Unit Testing: Simulates Firestore rule enforcement to validate correct access control and data protection.



4.3. Performance and Latency Testing

Performance and latency tests monitor the real-time behavior of the app, including API response times, data loading efficiency, and user interaction speed.
Key Examples Include:

- Identify bottlenecks in network or computation
- Measure success rates for operations
- Ensure responsiveness under varying conditions, including slow networks or low-end devices

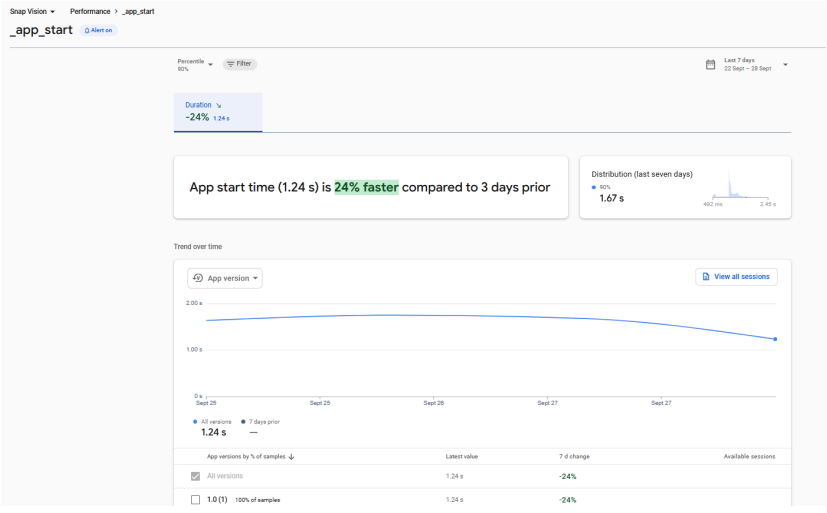
Tools:

Firebase Performance Monitoring (FPM):

- Automatically traces key operations in the React Native frontend
- Collects metrics on latency, success rates, and potential performance issues
- Provides actionable insights without requiring manual instrumentation for every operation

Examples:

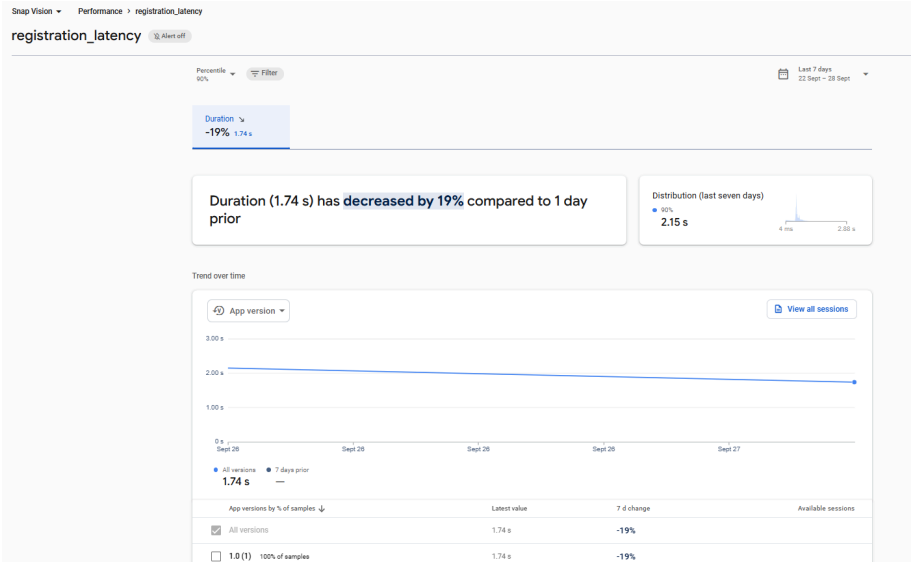
1. **App Launch Performance:**



- App now starts 24% faster for the majority of users
- 90% of users experience app startup in 1.24 seconds or less

- Consistent improvement observed over the last 3 days
- Performance gain is stable across all app versions

2. Registration Latency



- Registration process is now 19% faster
- 90th percentile indicates most users experience this improvement
- Consistent reduction across all app versions tested
- Improvement observed within 1 day