# SOFTWARE TESTING POLICY

**PREPARED FOR**

Andrew Broekman

Avinash Singh

Stacy Baror

**PREPARED BY**

Team Enigma

# Table of Contents

# Introduction

The testing policy describes the high level approach that will be undertaken towards System and Software Testing. It describes the test approach, the test environment, testing tools, release control, risk analysis and review and approvals.

## Principles

### Test at an early stage

To plan testing early in the project so that testing activities may be started as soon as possible in the project life cycle, and resources planned accordingly.

### Test-driven development

To follow a test-driven approach that focuses on the desired functionality of the application as well as other aspects such as program structure and readability.

### Risk-based approach

To adopt a risk-based approach to testing to help target, focus and prioritize testing with the added benefit of making efficient use of resources.

### Testing reveals defects

To focus on ensuring defect discovery (be they code defects, requirement defects etc.) takes place as soon as possible to prevent the escalating costs of identifying and fixing bugs later on in the project/software lifecycle.

### Test functional requirements

Ensuring full traceability of testing coverage back to original functional requirements to ensure key functional requirements are covered by testing.

### Exhaustive testing is impossible

Comprehensive testing (all combinations of inputs/outputs and preconditions) is not feasible, except in trivial cases. Instead of carrying out extensive testing, risk analyses and priorities must be used in order to restrict the effort involved in carrying out tests to the tests that genuinely need to be carried out.

# Test Environment

### Programming Languages

All tests were written in ***Dart***[2] for frontend development and in ***Kotlin***[11] and ***Java***[7] for backend development.

### Testing Frameworks

The **api** and **optimizer** repositories both made use of ***Gradle***[5] for creating tasks for tests.

The **api** repository made use of ***Kotest***[10] and ***Mockk***[15] to carry out testing.

The **optimizer** repository made use of ***JUnit***[8] and ***Mockito***[14] to carry out testing.

Both of these repositories made use of ***Spring***[20] in order to run tests on a test profile defined within each repository.

### GitLab CI/CD

The backend repositories made use of ***GitLab CI/CD***[4] in order to create pipelines that would run on merge requests and merge commits. This configuration was defined in a **.gitlab-ci.yml.**

### OpenJDK 14

The backend repositories require **JDK 14** in order to successfully execute tests.

# Assumptions

- Running ***Keycloak***[9] server.
- Internet connection.

# Executing Tests

All tests can be performed on the backend repositories by running the following command:

```
$ ./gradlew check
```

This will run the **test** and **e2e** tasks as well as **jacocoTestReport** and **sonarqube**. In order to prevent the ***SonarCloud***[19] analysis, run the following command:

```
$ ./gradlew check -x sonarqube
```

# Functional Requirement Testing

### User Management

One of the most important aspects of the system was the idea of three separate users, i.e., customer, fleet manager and trucker. Each user had different privileges which needed to be tested. Some of these included:

- Authorizing only fleet managers to access company management endpoints.
- Authorizing only customers to access tender management endpoints.

Furthermore, testing was carried out on basic user CRUD operations as well as authentication for retrieving both access tokens and refresh tokens.

### Company Management

Testing was carried out to ensure that fleet managers could perform the following actions:

- CRUD on company trucks.
- CRUD on company employees (truck drivers).

### Tender, Bid and Job Management

Testing was carried out on the following:

- CRUD on tender management.
- CRUD on bid management.
- CRUD on job management.
- Authorizations for tenders, bids and jobs.

### Bid Optimization

Testing different cases involved with the job allocations for a bid.

### Errand Management

Testing was carried out on the following:

- CRUD for errands.
- Authorizing truck drivers to start and end errands.

- Ensure that truck drivers are available between a time period.

### Chats

Testing was carried out on the following:

- Authorizing specific customers, fleet managers and truck drivers on chats.
- CRUD on messages.

### Reviews

Testing was carried out on the following:

- Create and read reviews.
- Authorize only customers and fleet managers.
- Allow for the creation of only a single review per bid.

## Non-functional Requirement Testing

### Maintainability

*SonarCloud*[19] is used to analyze and grade the backend codebase to identify code smells and technical debt which provides you with an idea of how maintainable your project is. *JaCoCo*[6] was used to generate code coverage reports for the **api** and **optimizer** repositories. The *Quality Gate*[16] for our projects can be viewed.

### Security

*SonarCloud*[19] is used to analyze and grade the backend codebase to identify security vulnerabilities and poor code practices.

### Reliability

*Litmus*[13] is used with *Kubernetes*[12] for chaos testing, by executing and analysing chaos tests you are able to verify the reliability of a system.

# Release Control

### Versioning

Our api and optimizer repositories make use of a plugin called ***Semantic Release***[18] in order to automatically generate a changelog, tag and release on each relevant merge commit. The entries in the changelog and the descriptions of our release are determined by the commit messages introduced in each merge request. The following version bumps occur based on the conventional commits in the git history since the latest release.

Furthermore, these projects use the ***SemVer***[17] schema and the following commits cause major release version bumps based on the following commit types:

- *fix* - bumps patch version
- *feat* - bumps minor version
- *BREAKING CHANGE* - bumps major version

### Commit Control

***Conventional Commits***[1] are used as a commit naming guideline for this repository. For more information on this, please visit the following link.

Note that ***GitLab***[3] automatically links commits to issues when an issue number is referenced with **#<issue no.>**.

Each commit message follows the pattern,

Regex: *(build|chore|ci|docs|feat|fix|perf|refactor|style|test)(\(#\d+\))?: (.+) ((\n\n|.)+)* Simplified Pattern: *<type>(<optional scope>): <description>*

# Risk Analysis

***SonarCloud***[19] runs on both the **api** and **optimizer** repositories whenever a new merge is created, this provides a report which is linked to the merge via an automated commit. Part of this report includes a security analysis.

# Reviews and Approvals

When making a merge request, the pipeline must pass in order for the merge request to be permitted. The following rules apply:

- All protected pre-release branches (alpha and beta) require at least one approval, not including the author, in order to accept a merge request. This ensures that before any code is moved into the development branch it is reviewed by at least one other member of the team.
- Merges made into the master branch require approval of all three of the project owners, the Truckin-IT project clients.

# References


1. *Conventional Commits*
2. *Dart*
3. *GitLab*
4. *GitLab CI/CD*
5. *Gradle*
6. *JaCoCo*
7. *Java*
8. *JUnit*
9. *Keycloak*
10. *Kotest*
11. *Kotlin*
12. *Kubernetes*
13. *Litmus*
14. *Mockito*
15. *Mockk*
16. *Quality Gate*
17. *SemVer*
18. *Semantic Release*
19. *SonarCloud*
20. *Spring*