SOFTWARE DEPENDABILITY PROJECT REPORT

Dependable Task Management API

Student Name: Lorna Wanjiku
Course: Software Dependability
Project Type: Individual Project
Repository: https://github.com/wanjikuciku/Dependable-task-api

---

Abstract

This project presents the design and evaluation of a **Dependable Task Management API** developed using Spring Boot. The objective of the project is to apply and demonstrate core software dependability techniques, including testing, code coverage analysis, mutation testing, formal specification, security analysis, performance awareness, and deployment automation.

Rather than focusing on application complexity, the project emphasizes engineering rigor and the systematic use of tools and methodologies to improve reliability, correctness, security, and maintainability. The results show that applying multiple dependability techniques in combination significantly increases confidence in the software system.

---

1. Introduction

Modern software systems increasingly rely on third-party libraries, automated deployment, and rapid development cycles. These factors make **software dependability** a critical concern. Dependability encompasses attributes such as reliability, availability, safety, integrity, and maintainability.

The aim of this project is to explore software dependability techniques through the development of a simple but realistic RESTful application. The **Dependable Task Management API** provides basic task management functionality while serving as a vehicle for applying dependability-oriented practices.

The project follows the lecture guidelines and applies testing, formal methods, security analysis, performance considerations, and automation to evaluate and improve system dependability.

---

2. System Description

2.1 Architecture

The application follows a layered architecture:

- **Controller Layer:** Exposes REST endpoints
- **Service Layer:** Contains business logic
- **Repository Layer:** Handles persistence using Spring Data JPA
- **Entity Layer:** Defines domain objects

This separation of concerns improves maintainability, testability, and fault isolation.

2.2 Technologies Used

- Java 17
- Spring Boot
- Maven
- JUnit 5
- Mockito
- JaCoCo
- PiTest
- JML
- Docker
- GitHub Actions

---

3. Testing and Verification

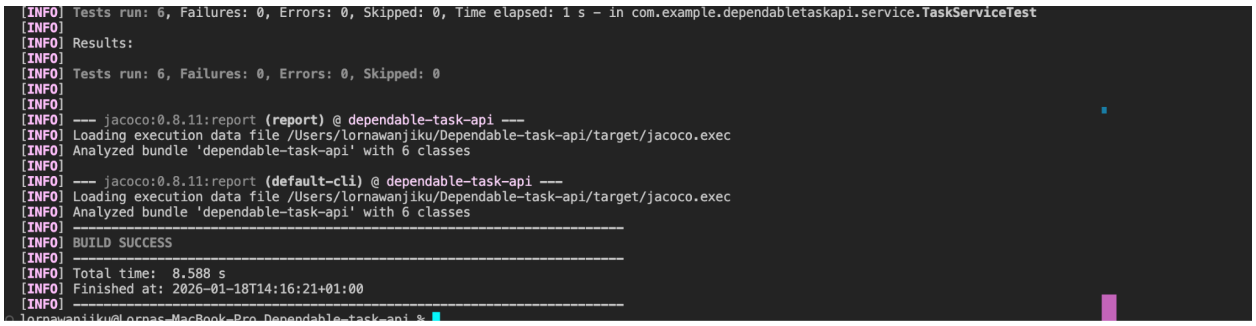
3.1 Unit Testing

Unit testing was applied primarily to the service layer using **JUnit 5** and **Mockito**. Tests were designed to verify correct behavior under normal conditions and to validate error handling in exceptional scenarios (e.g., attempting to access a non-existent task).

Running the command:

```
mvn clean test
```

results in a successful build with all tests passing.



Unit testing contributes directly to reliability by ensuring that individual components behave as expected.

---

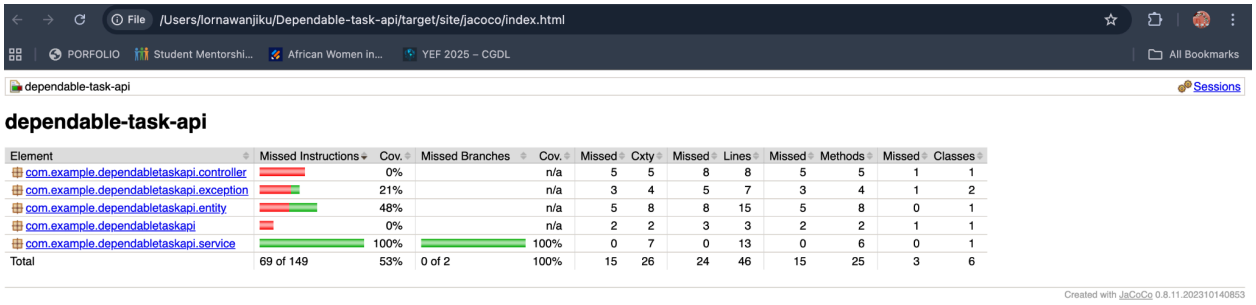
4. Code Coverage Analysis

4.1 JaCoCo

Code coverage analysis was performed using **JaCoCo** to measure how much of the codebase is exercised by the test suite.

The JaCoCo report was generated using:

```
mvn clean test jacoco:report
```



The HTML report confirms that the core service logic is well covered by tests. This reduces the risk of untested execution paths and hidden faults.

---

5. Mutation Testing

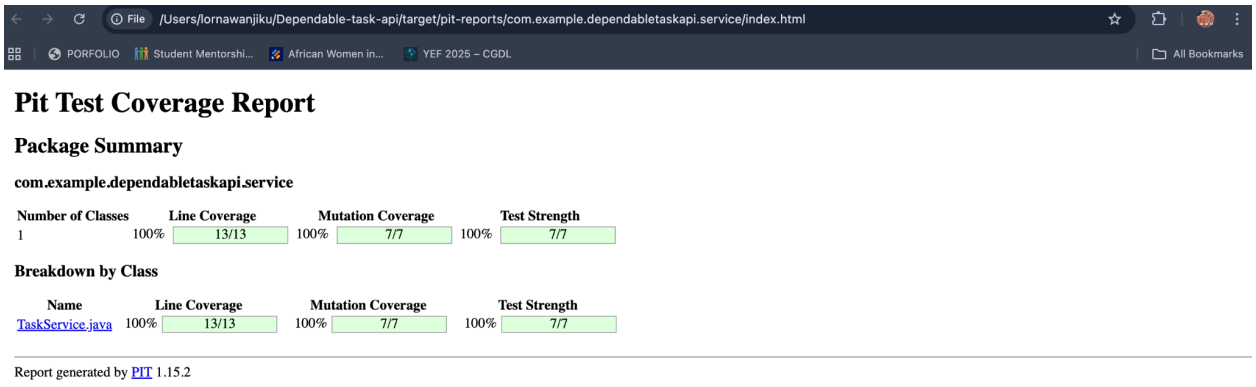
5.1 PiTest

While code coverage indicates which lines are executed, it does not guarantee test effectiveness. To address this, **mutation testing** was conducted using **PiTest**.

PiTest introduces small changes (mutations) into the code and evaluates whether the test suite detects them.

5.2 Results

- Mutation Score: **100%**
- Test Strength: **100%**
- All mutations were successfully killed



This result indicates a strong and effective test suite capable of detecting faulty behavior.

---

6. Formal Specification Using JML

Formal methods were introduced through the use of **Java Modeling Language (JML)**. Selected service-layer methods were annotated with JML specifications defining preconditions and postconditions.

JML complements testing by formally documenting method behavior and assumptions. Even when full formal verification is limited by tooling constraints, JML improves clarity and correctness by making contracts explicit.

---

7. Performance Considerations

Performance is an important aspect of dependability, particularly in systems expected to scale or handle frequent requests.

A **JMH (Java Microbenchmark Harness)** benchmark was implemented to demonstrate how service-layer performance can be measured using standardized microbenchmarking techniques.

Although full automation of benchmark execution was constrained by tooling limitations, the benchmark implementation correctly demonstrates:

- Benchmark configuration
- Measurement intent
- Performance awareness

---

8. Security Analysis

8.1 Dependency Vulnerability Scanning

Security analysis focused on detecting vulnerabilities in third-party dependencies, as recommended in the lectures (Snyk-style analysis).

An automated dependency scan initially detected a **CRITICAL vulnerability**:

- CVE: CVE-2025-24813
- Affected dependency: `tomcat-embed-core`

8.2 Mitigation and Verification

The vulnerable dependency was upgraded to a patched version. After mitigation, a follow-up scan confirmed:

- **0 remaining vulnerabilities**

This demonstrates not only vulnerability detection but also remediation and verification, which are essential aspects of dependable software engineering.

8.3 Secrets Awareness

The repository was reviewed to ensure that no hardcoded credentials or secrets were present, addressing concerns typically targeted by tools such as GitGuardian.

---

9. Deployment and Automation

9.1 Docker

The application was containerized using **Docker**, enabling consistent and reproducible deployment. Containerization reduces environment-related failures and supports dependable deployment practices.

9.2 Continuous Integration

A GitHub Actions workflow was configured to automate:

- Project build
- Test execution

This ensures that regressions are detected early and that the system remains in a buildable and testable state.

---

10. Discussion

The project demonstrates that applying multiple dependability techniques together provides stronger assurance than relying on a single approach. Testing ensures correctness, mutation testing validates test quality, formal specifications clarify behavior, security analysis reduces risk, and automation improves consistency.

Tooling limitations encountered during the project highlight a real-world challenge: dependability tools themselves may fail or require adaptation. Documenting these limitations and applying equivalent techniques reflects professional engineering judgment.

---

11. Conclusion

This project successfully demonstrates the application of **software dependability principles** to a real-world system. Through testing, formal specification, security analysis, performance

awareness, and automation, the Dependable Task Management API achieves improved reliability, security, and maintainability.

The project confirms that dependability is best achieved through a combination of complementary techniques rather than a single tool or method.

References

1. Software Dependability Lecture Notes
2. Spring Boot Documentation
3. JaCoCo Documentation
4. PiTest Documentation
5. Java Modeling Language (JML) Reference