TUS

Technological University of the Shannon: Midlands Midwest

Ollscoil Teicneolaíochta na Sionainne: Lár Tíre Iarthar Láir

www.tus.ie

A Comprehensive Project Report on **"Implementation of a CI/CD Pipeline for the Student Management API Project"**

Presented by: Atharva Dinesh Pailwan

Course Name: MSc. in Software Design with Cloud Native Computing

Subject: Continuous Build and Delivery

Professor: Dr. Mary Giblin, Dr. Amit Hirway

Student ID: A00325723

Email: a00325723@student.tus.ie

TABLE OF CONTENTS

1. Introduction

1.1 Student Management API

The Student Management API is a Java-based Spring Boot microservice designed to manage student records via a RESTful interface. This project was developed to demonstrate a robust application of Continuous Integration and Continuous Delivery (CI/CD) methodologies, focusing on automation in build, testing, and deployment within a microservice architecture..

1.2 Assignment Objectives:

This project aims to deliver a fully functional API with an accompanying automated CI/CD pipeline, fulfilling the following key requirements:

- **Automated Builds and Version Control:** Implementation of automated builds using Jenkins, with code housed in a Git repository.
- **Static Code Analysis:** Integration of SonarQube to ensure code quality across builds.
- **Test Automation and Coverage:** Comprehensive testing using JUnit and Mockito, with coverage analysis via JaCoCo.
- **Dockerization and Automated Deployment:** Utilization of Docker for containerization and Ansible for automated deployment.
- **Evaluation of Pipeline Efficiency:** Analysis and discussion of the pipeline's performance, including challenges faced and potential improvements.

This report details the successful implementation of the Student Management API, outlining the architectural choices, technology stack, and key components of the CI/CD pipeline. It concludes with a reflection on the pipeline's effectiveness and lessons learned throughout the project lifecycle.

2. **Project Architecture and Design**

The Student Management API was architecturally designed as a monolithic Spring Boot application, adhering to a clean, layered architecture consisting of controllers, services, repositories, and domain models. The project leveraged a range of technologies including Java 17, Spring Boot 3, Maven, JUnit, Mockito, Docker, and Ansible, among others.

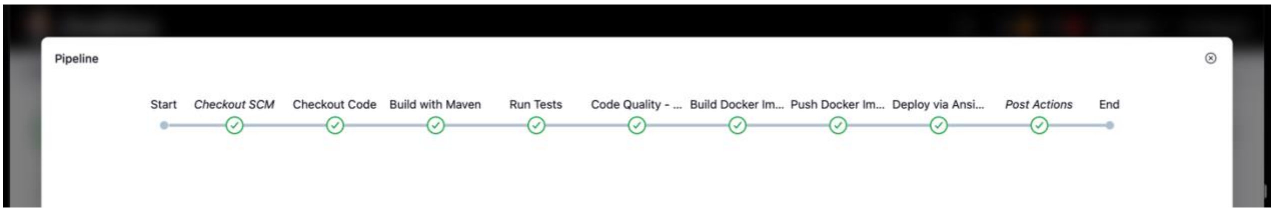
2.1 Key Components:

- **Controllers:** Managed HTTP request routing and response handling.
- **Services:** Encapsulated the business logic of the application.
- **Repositories:** Provided data access mechanisms, interfacing with the H2 database.
- **Domain Models:** Represented the data structure within the application.

The entire application was packaged using Maven, facilitating dependency management and build automation.

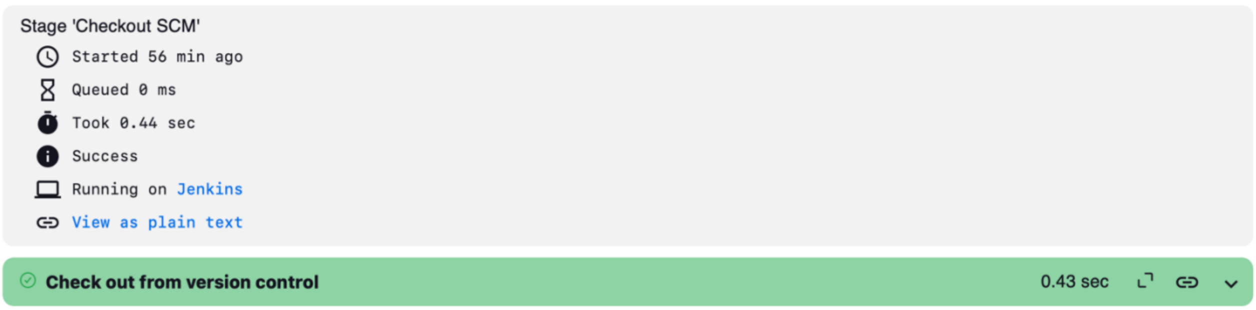
2.2 CI/CD Pipeline Overview

The CI/CD pipeline for the Student Management API, orchestrated through Jenkins, was meticulously designed to automate each step from code integration to deployment seamlessly. This comprehensive automation ensured consistent and efficient development practices, minimizing human error and maximizing reproducibility. Below are the expanded details of each key stage in the pipeline:



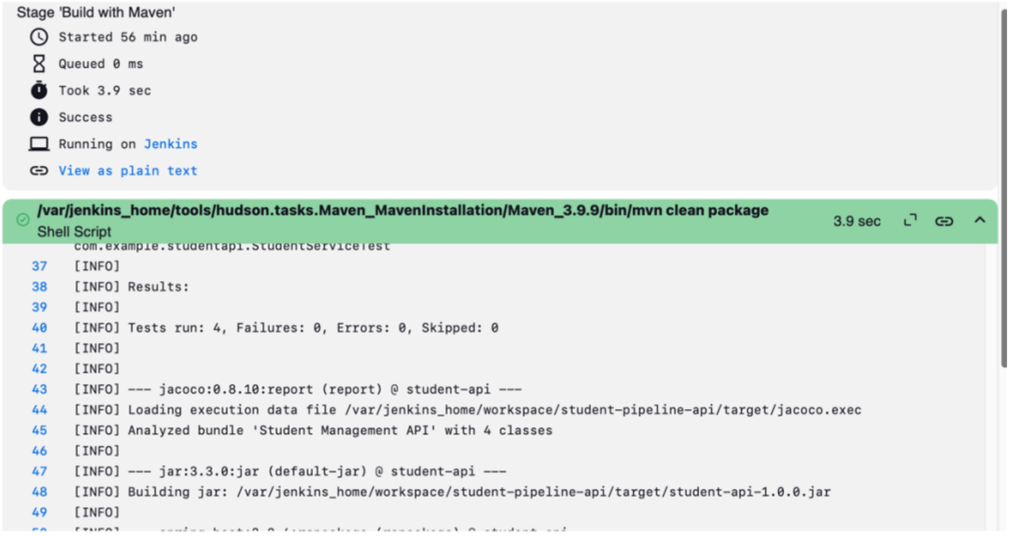
2.2.1 Code Checkout

The first stage in the pipeline involved the automated checkout of the latest code from the Git repository. This step was crucial as it ensured that the pipeline always used the most current version of the source code for every build. Jenkins was configured to watch for any commits to the repository; upon detecting a change, it automatically fetched the updated code. This setup supported a trigger-based mechanism where new commits prompted immediate pipeline execution, thereby integrating changes continuously.



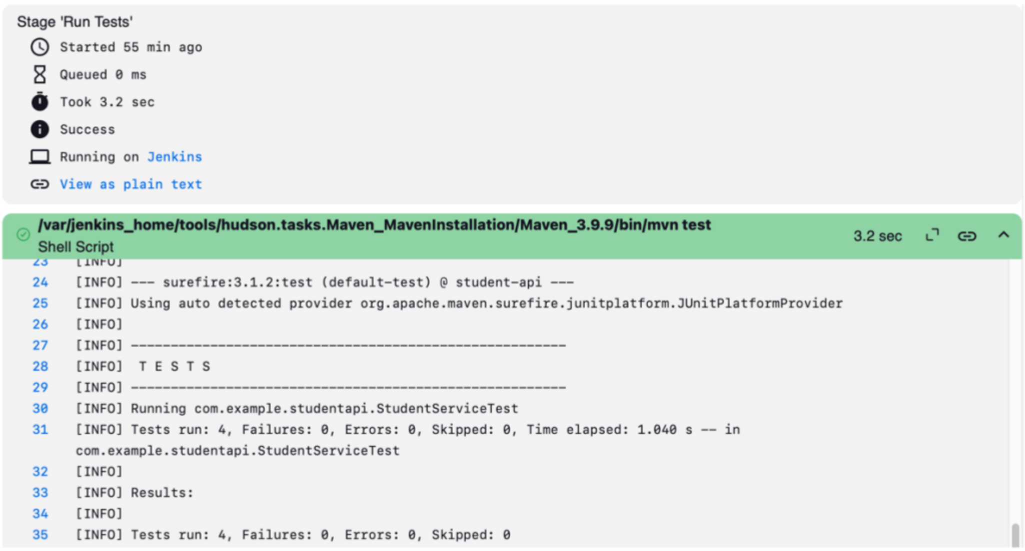
2.2.2 Build Execution

Following the code checkout, the pipeline executed the build process using Maven, the chosen build automation tool. Maven compiled the source code and packaged the application into an executable JAR file. This process included resolving and downloading dependencies defined in the pom.xml file, ensuring that the build environment was self-contained and consistent across different development and production environments. The automation of this step eliminated the manual build process, reducing the potential for human error and ensuring that the build outputs were always consistent.



2.2.3Testing

After the build, the pipeline focused on testing the application. This stage was critical for verifying the correctness of the application code before it moved further along in the pipeline. A comprehensive suite of unit tests, written using JUnit, was executed automatically. These tests were designed to cover various components of the application, particularly focusing on the business logic encapsulated in the service layer. The use of Mockito facilitated the mocking of dependencies, allowing for isolated testing of specific functionality without the need for actual database connections or external systems. Jenkins collected and reported the results of these tests, providing immediate feedback on their success or failure. This immediate feedback was vital for early detection of issues, enabling quick fixes that maintained the overall health of the application codebase.

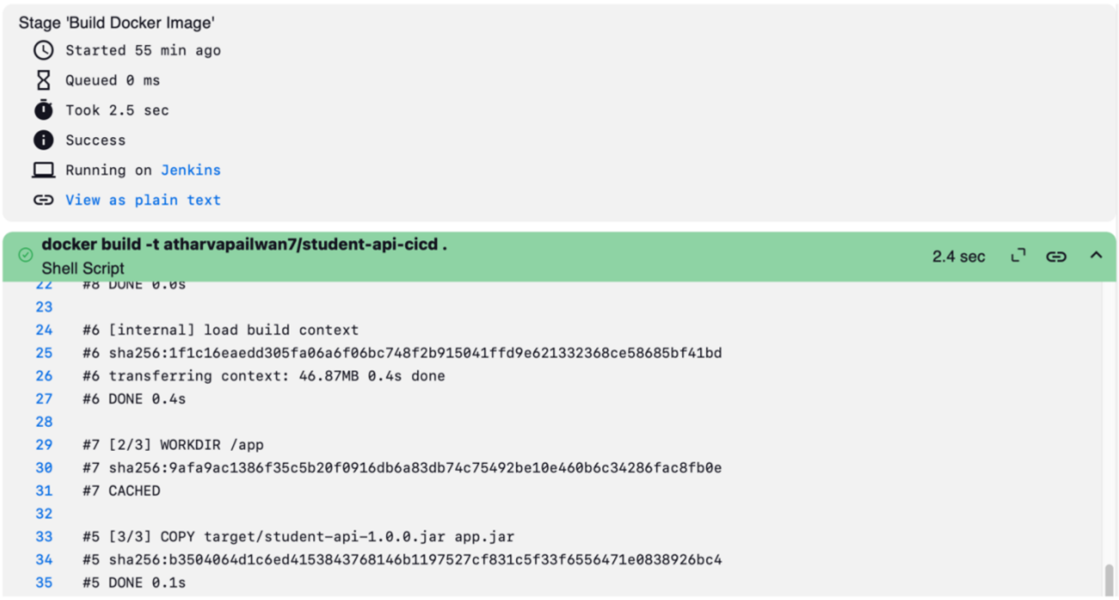


2.2.4 Static Code Analysis

Static code analysis was conducted using SonarQube, integrated into the pipeline following the testing stage. This tool analyzed the source code to identify potential bugs, vulnerabilities, code smells, and to ensure adherence to defined coding standards. SonarQube's analysis provided a detailed report on various quality metrics, including test coverage, complexity, duplicated code, and coding standard violations. The integration of SonarQube into the CI/CD pipeline ensured that code quality was assessed continuously, allowing developers to address issues promptly before merging changes into the main branch. This continuous quality control is essential in maintaining high standards throughout the project's lifecycle.

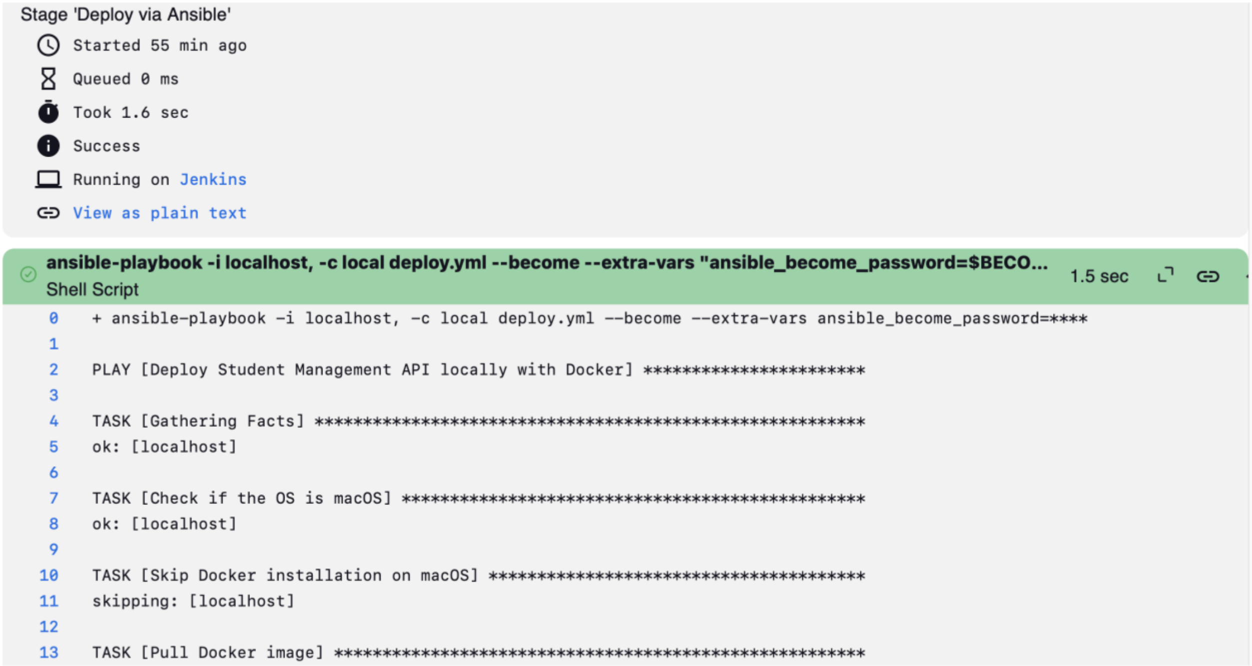
2.2.5 Containerization

Docker played a pivotal role in the containerization stage of the pipeline. Once the application had been built and tested, a Dockerfile was used to build a Docker image containing the application and its runtime environment. This Docker image encapsulated everything needed to run the application, ensuring that it could be deployed consistently in any environment that supports Docker. This stage was crucial for abstracting the application from the underlying infrastructure, facilitating portability, and ensuring that the application ran identically in development, testing, and production environments.



2.2.6 Deployment via Ansible

The final stage of the pipeline involved the deployment of the application, which was automated using Ansible. Ansible scripts were configured to deploy the Docker container to specified server environments. This automation included tasks such as pulling the latest Docker image from a registry, stopping any currently running containers, and starting the new container. This approach not only streamlined the deployment process but also minimized downtime and ensured a smooth transition between application versions. By automating deployment through Ansible, the project achieved a repeatable and error-free deployment process, crucial for maintaining the reliability and availability of the application in production.



3. Testing and Coverage

Testing Approach: The project's testing strategy focused primarily on unit testing the service layer. A JUnit 5 test class, StudentServiceTest, was written to exercise the business logic in StudentService. Using Mockito, the StudentRepository was mocked so that the service methods could be tested in isolation without needing a real database. Each method of StudentService has a corresponding unit test:

- *testFindAllStudents()* creates a list of dummy Student objects, configures the repository mock to return this list for findAll(), and then asserts that the service returns the expected list and size.
- *testFindById_StudentExists()* simulates a repository findById returning a Optional<Student> for a given ID and verifies that the service correctly returns the student when present.
- *testSaveStudent()* simulates saving a student and ensures the returned object is not null and has the expected field values.
- *testDeleteById()* uses Mockito's verify to ensure repository.deleteById() was called when the service's delete method is invoked.

3.1 Code Coverage Results

Code coverage was measured using JaCoCo, which produces a report on how much of the code was executed by the tests. The overall coverage is relatively low – the JaCoCo report indicates about 28% instruction coverage (i.e., 28% of the bytecode instructions were executed during tests), which corresponds to roughly 34.1% line coverage. In other words, only about one-third of the lines of code are covered by tests. This is because the unit tests targeted only the service layer; the controller's methods and the main application (and some parts of the entity) were never invoked in tests. Below is a breakdown of coverage by class:

- *StudentService* – 100% of lines covered. All service methods were executed by the unit tests, so this class is fully tested.
- *Student* (Entity) – ~41% of lines covered. The entity is partially covered indirectly through the tests (the tests create Student objects and call a couple of getters). Many of the entity's simple getters/setters are not explicitly tested, which is generally acceptable since they are trivial, but they count toward uncovered lines.
- *StudentController* – 0% lines covered. None of the controller endpoints were tested, so this class's 14 lines of logic (mostly the endpoint methods) were never executed in the test run.
- *StudentApiApplication* – 0% lines covered. The main method and configuration were not tested (commonly, the main class isn't explicitly tested in unit tests).

Overall, the project's line coverage is ~34%, which is below typical industry recommendations (many projects aim for ~80% or higher). This low figure is a result of focusing on unit tests for one layer and the absence of integration tests. The test suite effectively validates the core service logic but does not ensure the controller endpoints work as expected. Despite the low coverage, the most critical logic (service methods) has been verified by tests, which gives some confidence in those parts of the code.

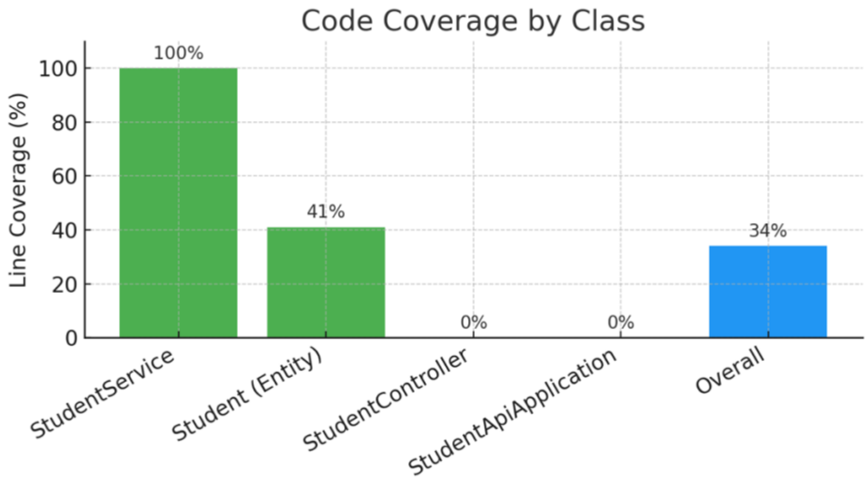
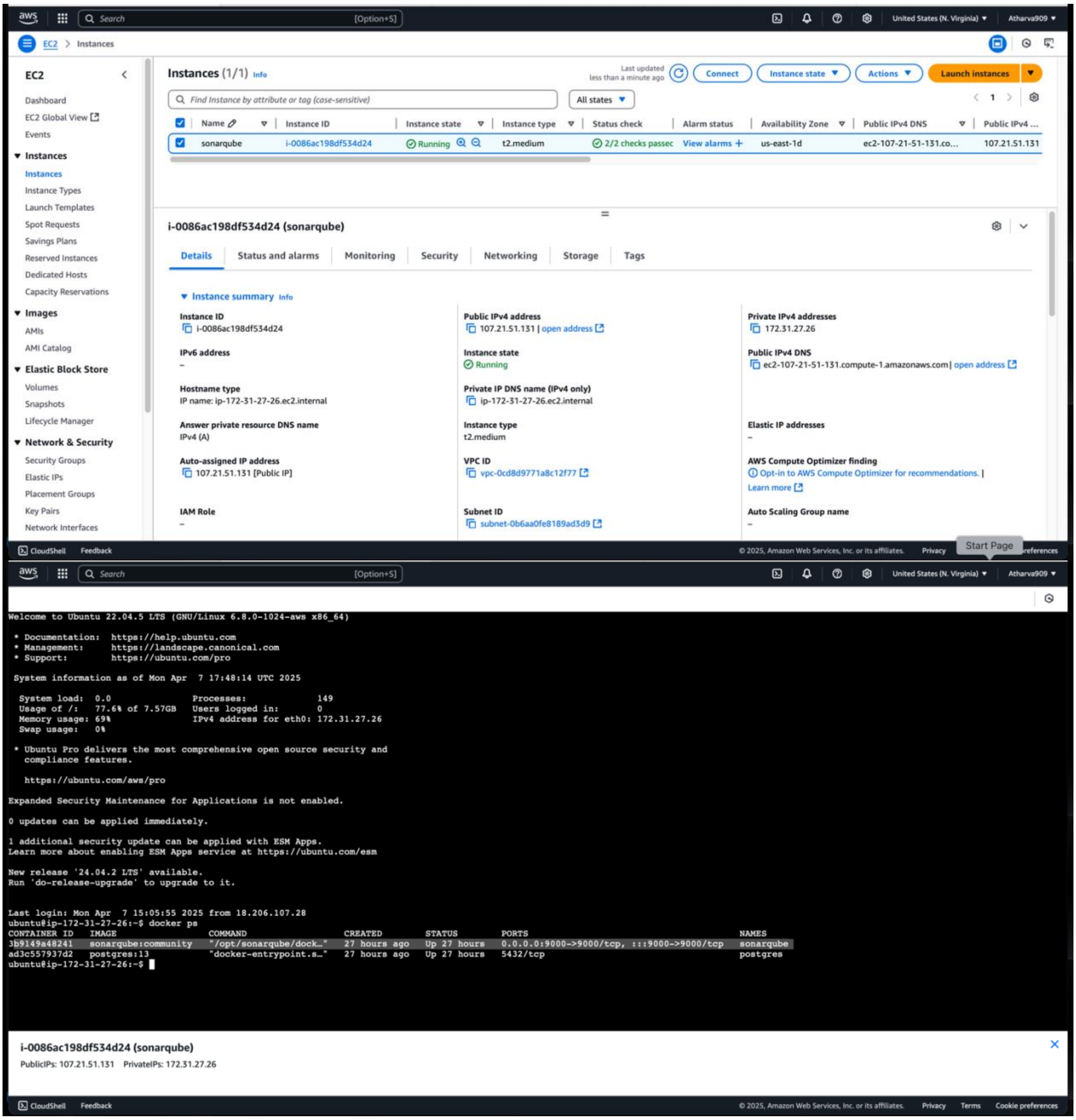


Figure : Code Coverage by Class.

This chart (generated from the JaCoCo report) shows the percentage of lines covered by tests for each major class in the application. As discussed, the service layer has 100% coverage, while the controller and the main application have 0%. The overall line coverage is ~34%. Improving these numbers would involve writing additional tests for the controller (and possibly repository integration), as outlined in the report.

4. Static Code Analysis using SonarQube

Static code analysis was performed using SonarQube, which was integrated into the CI pipeline. After each build, the code was analyzed for potential issues in reliability, security, and maintainability, as well as to compute duplicate code and coverage metrics. It was implemented on the AWS EC2 instance as shown in the figure below.



The SonarQube dashboard for the Student Management API project shows that the application passed all configured quality gates. In particular, the project earned the following ratings and metrics:

- **Reliability:** A (Grade A, meaning zero bugs were detected by SonarQube) – The code has no known reliability issues. All functions appear to behave as intended without detecting any bug patterns.
- **Security:** A (no vulnerabilities found) – SonarQube did not identify any security vulnerabilities or hotspots in the code. This is expected given the simplicity of the application (no external inputs besides the REST API, and using Spring Boot which by default handles common vulnerabilities if used properly).
- **Maintainability:** A (zero code smells or very low technical debt) – The codebase is small and straightforward, resulting in no code smell issues reported. SonarQube calculates a maintainability rating based on technical debt; an A rating indicates that if there are any code smells, they are minor and would take very little time to fix.
- **Coverage:** 34.1% **(test coverage)** – This is the line coverage percentage imported from JaCoCo. While this number is relatively low, it still passed the quality gate in SonarQube (possibly the quality gate was configured to not fail below a certain threshold, or the focus was on ensuring *some* coverage and clean code rather than enforcing a high coverage minimum). This metric reinforces the earlier analysis that only about one-third of the code is covered by tests.
- **Duplications:** 0.0% – There is no duplicated code in the project according to SonarQube's duplication detection. This is unsurprising given the project's size; each piece of functionality is defined only once (no copy-pasted code). The presence of a service layer also means logic isn't repeated in the controller, for example.

Because all these metrics meet the defined standards (e.g., no critical issues, coverage above 0%, etc.), the Quality Gate status is marked as Passed. In SonarQube's terminology, a quality gate is a set of conditions that the project must fulfill (for instance, "Coverage > 80%" or "No new blocker issues"). The assignment likely used a default or custom quality gate that this project satisfied. Achieving an all-green dashboard was a key goal – it demonstrates that the code is of high quality (at least by automated checks). The static analysis process helped ensure that the team (or individual developer) adhered to good coding practices. For example, SonarQube would have flagged issues like unused variables, complexity, or bad practices if they existed; none of those appear in the final code. This implies that any such issues were corrected during development or none were introduced.

5. Challenges and Improvements

While the project met its primary objectives, there were several challenges encountered and areas for improvement identified during implementation:

- **Test Coverage and Depth:** Achieving comprehensive testing was challenging within the project timeline. As noted, only unit tests for the service layer were implemented, leading to modest coverage. **Improvement:** In the future, more tests should be added to increase coverage. This includes **integration tests** for the web layer (using Spring's MockMvc or an in-memory HTTP client) to test the StudentController endpoints. For example, tests could be written to simulate HTTP GET/POST requests and assert the correct HTTP responses and database state. Additionally, testing edge cases (such as requesting a non-existent student ID and expecting a 404 error) would improve reliability. If end-to-end testing is desired, one could use tools like Postman or Karate to test the running application as a black box. Increasing the coverage closer to the recommended 80% would better satisfy industry best practices and assignment expectations.

- **Exception Handling:** The current controller simply throws a runtime exception when a student is not found. This was a quick solution but not ideal, as it results in a 500 Internal Server Error. **Improvement:** Implement a custom exception (e.g., StudentNotFoundException) and a global exception handler to return a 404 Not Found status with a meaningful message. This would make the API more user-friendly and align with RESTful principles.

- **Continuous Integration Setup:** Setting up Jenkins and SonarQube integration involved dealing with credentials and configurations (for example, generating a SonarQube token, configuring Jenkins agents with Docker and Ansible, etc.). Ensuring the Jenkins pipeline worked end-to-end required troubleshooting environment issues (like installing the correct Maven version on the Jenkins node, Docker daemon access, etc.). **Improvement:** One possible enhancement is to add a SonarQube **quality gate check** in the pipeline. Currently, the pipeline runs SonarQube analysis but does not explicitly stop the build if the quality gate fails. Jenkins has a way to wait for SonarQube to compute the quality gate result and then mark the build failed if it didn't pass. Integrating that would enforce quality standards strictly (e.g., if in the future we set a higher coverage threshold or other conditions).

- **Pipeline Performance:** The CI/CD pipeline, while functional, could be optimized. For instance, the build and test stages could potentially be combined (since mvn package already runs tests by default). Alternatively, one could use Maven's parallel test execution to speed up the test stage (not crucial here due to few tests). Caching Maven dependencies on the Jenkins agent is another improvement – this would avoid downloading the same libraries on each build, thus speeding up the build process.

- **Docker and Deployment:** Containerizing the application was straightforward with a multi-stage Dockerfile, but deploying via Ansible required the target environment to have Docker and the correct configuration. One challenge was ensuring the Ansible playbook could run on the Jenkins host (using localhost in this case) with appropriate permissions. Using Ansible's **Docker modules** or even Docker Compose could simplify the deployment. **Improvement:** In a more advanced scenario, the deployment could be improved by deploying to a Kubernetes cluster for scalability, but that was beyond the scope. Another small improvement is to use versioning tags for Docker images (e.g., tagging images with the Git commit or version number instead of always "latest") for better traceability of deployments.

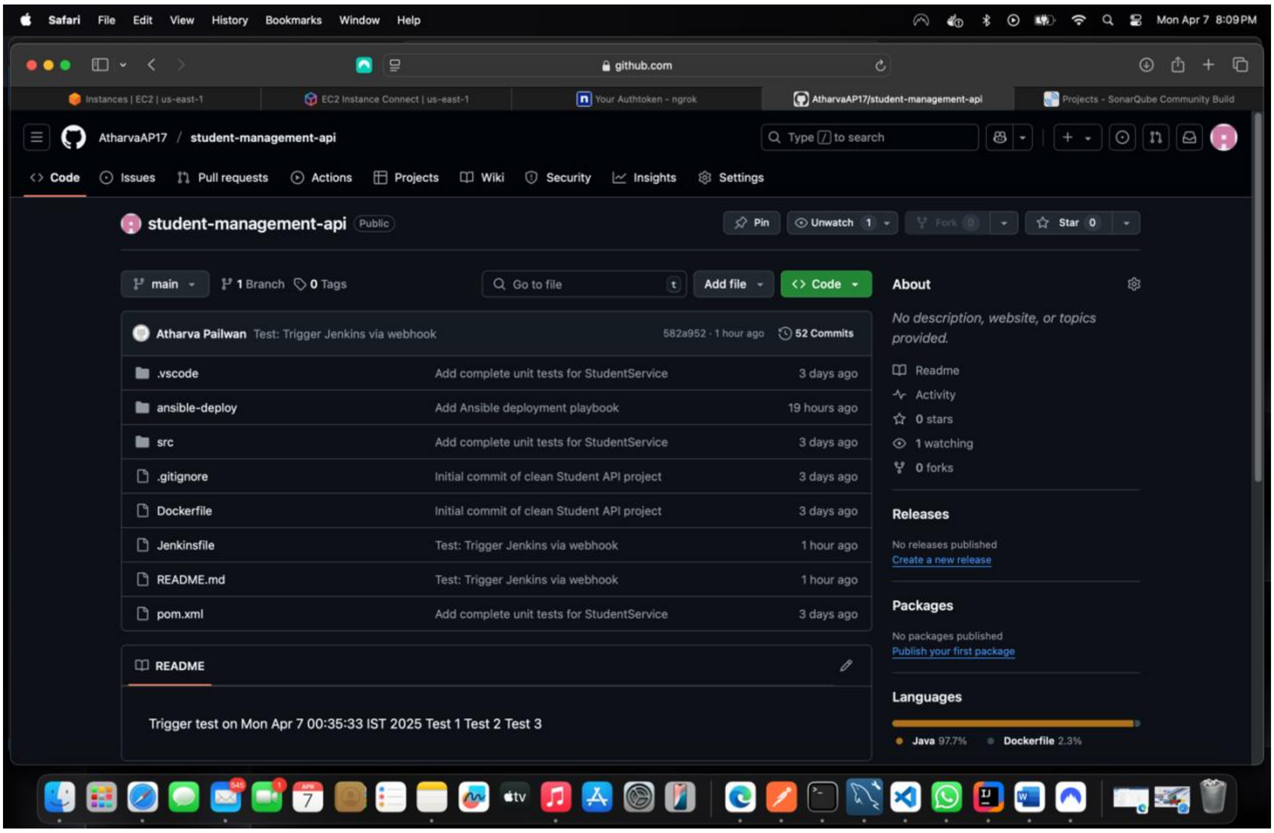
- **Security Considerations:** Although SonarQube didn't find security issues, real-world improvements could include adding authentication/authorization to the API (e.g., using Spring Security with JWT). This wasn't required in the assignment, but it's an area to consider for making the service production-ready. Also, currently the H2 database is used without credentials (H2 console enabled in dev mode), which is fine for a demo but would be locked down in a production environment.

- **Project Structure:** All code resides in one package (com.example.studentapi). For better organization, especially as the project grows, the code could be refactored into sub-packages (e.g., .controller, .service, .repository, .model). This doesn't change runtime behavior but improves maintainability by clearly separating layers. It's a minor structural improvement

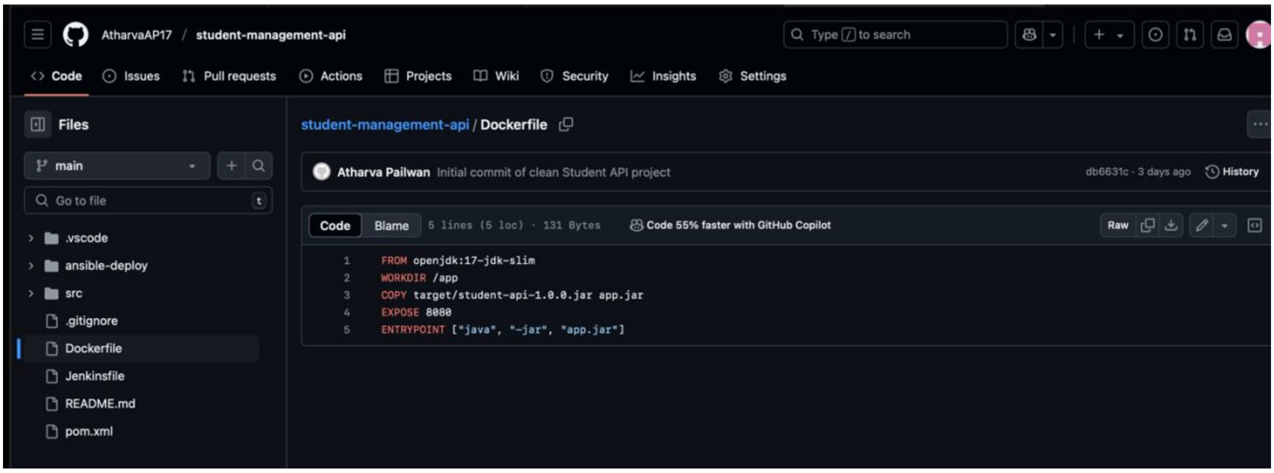
6. Screenshots

Code file pushed to the git repository containing all the files(Jenkinsfile, Dockerfile, Ansible playbook, etc.):

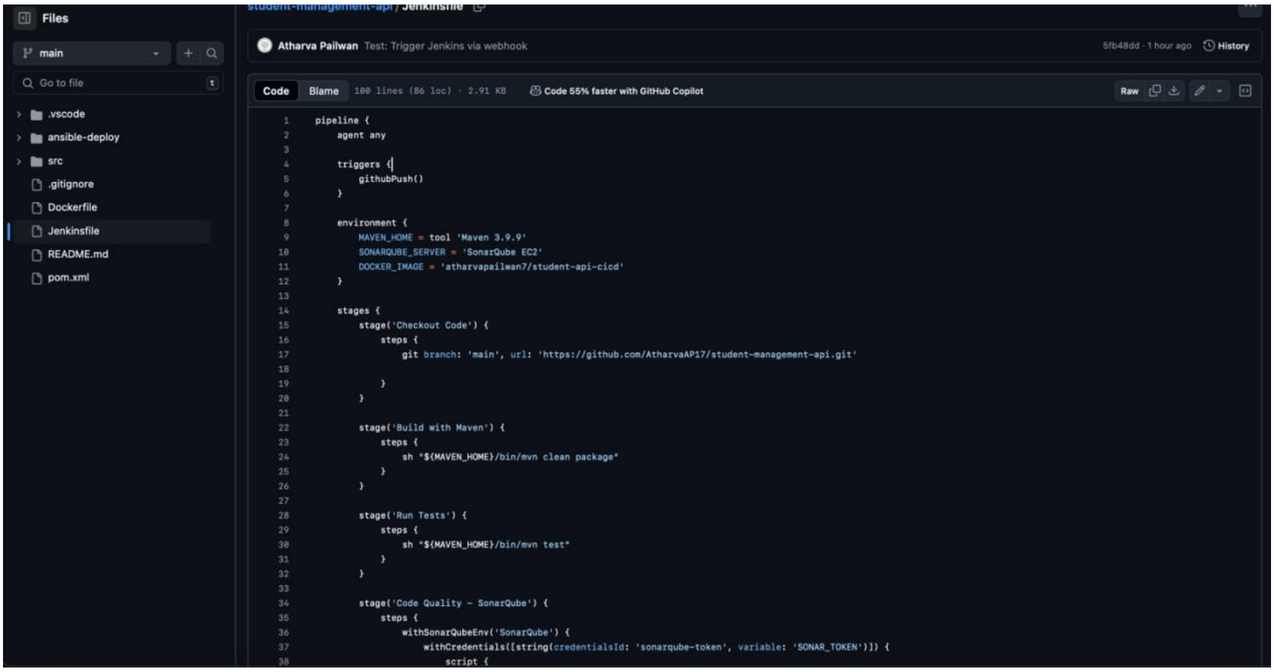
GitHub Repository link*:* [*https://github.com/AtharvaAP17*](https://github.com/AtharvaAP17)



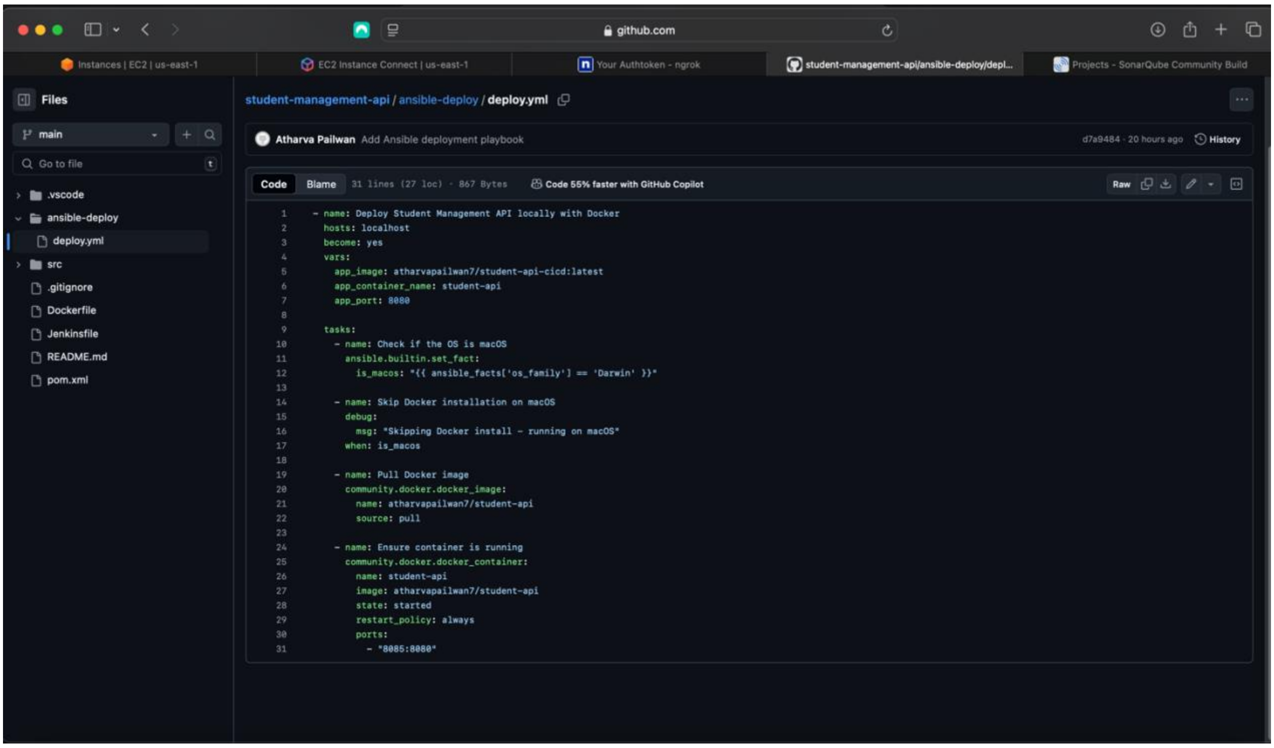
Dockerfile:



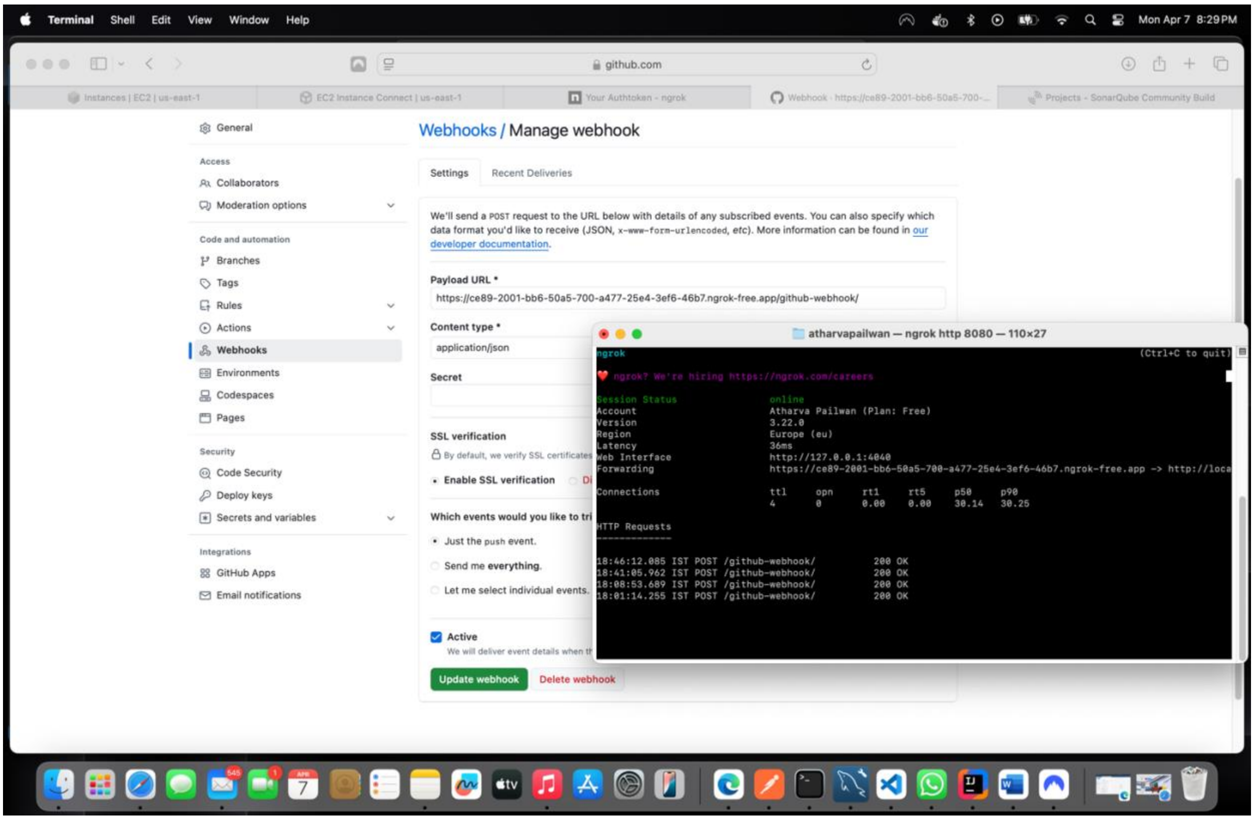
Jenkinsfile:



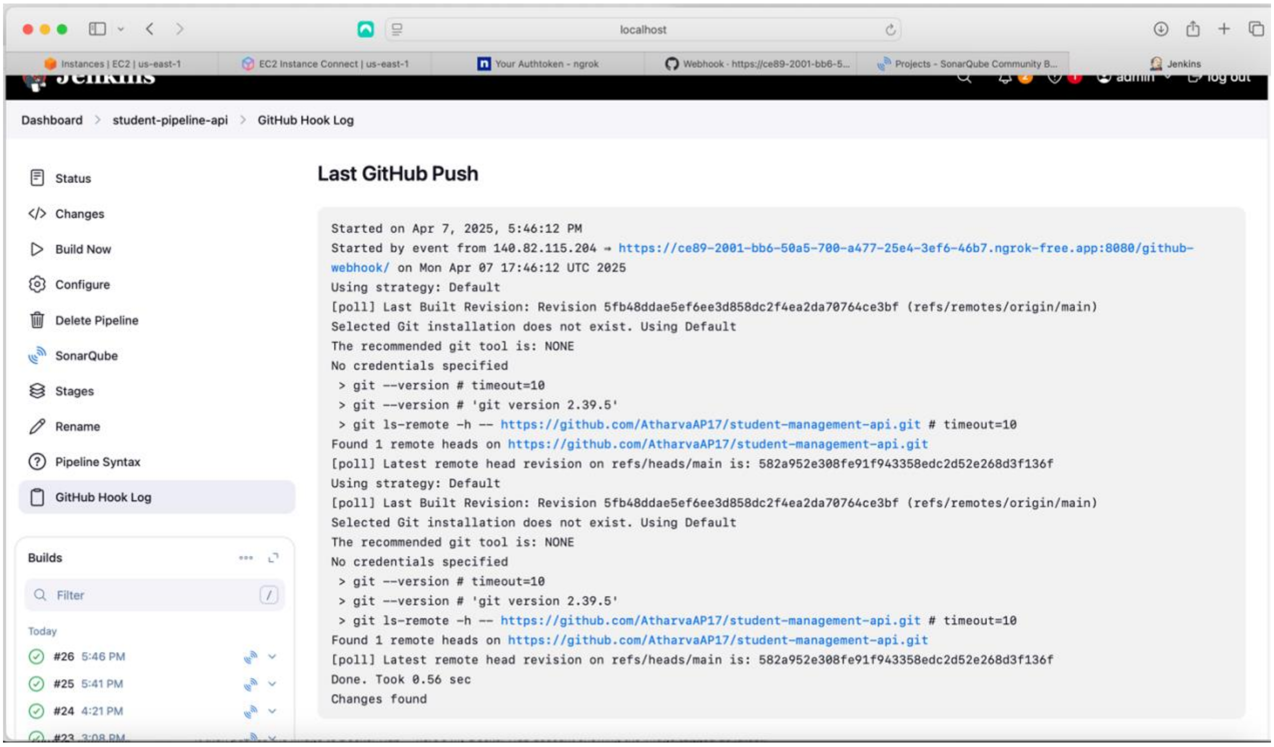
Ansible deploy.yml file:



For creating an automated pipeline trigger, github webhook was created with ngrok's port forwarding which can be seen in the screenshot below:



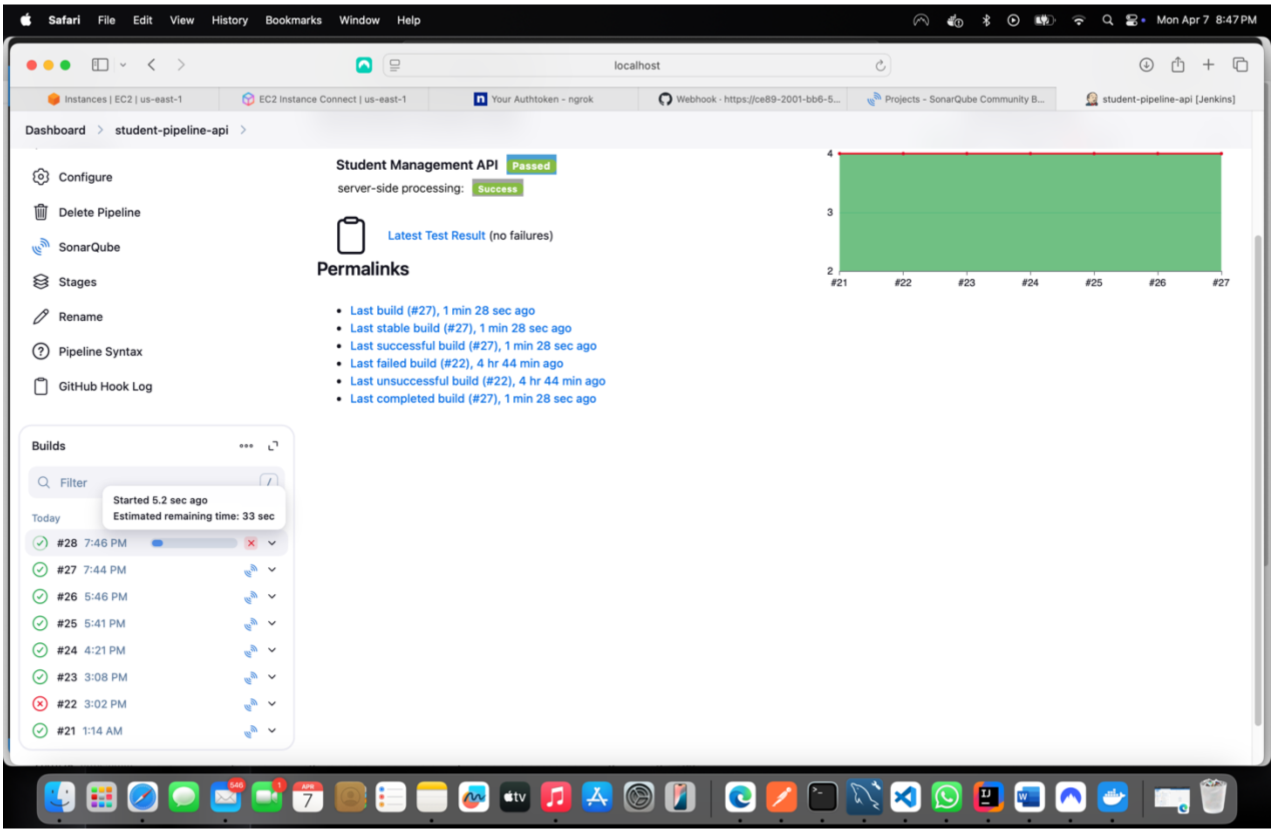
Github Webhook log:



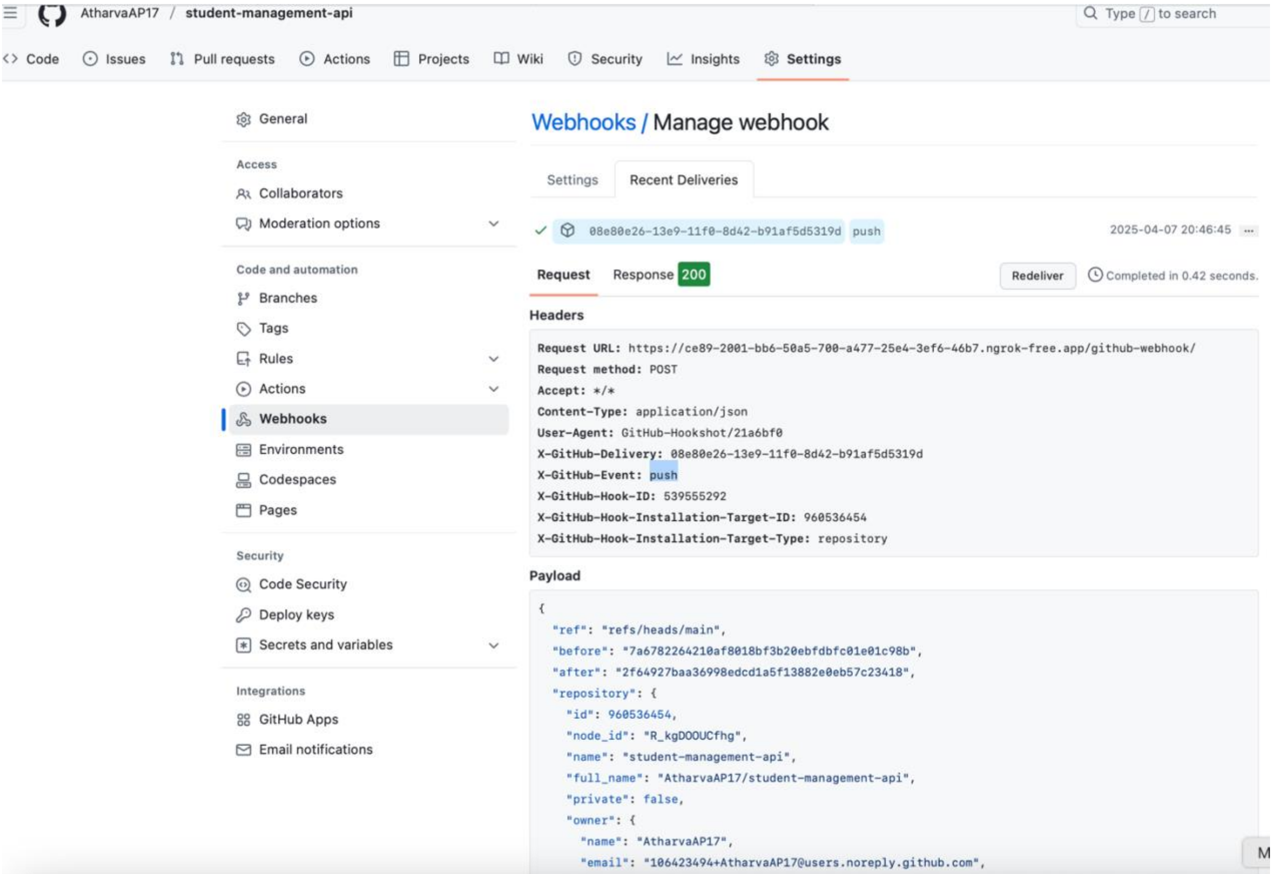
Pushing code to the github repo which triggers the pipeline:

```
TERMINAL
atharvapailwan@Atharvas-MacBook-Pro student-management-api % git commit -m "Test: Trigger Jenkins via webhook"
[main 582a952] Test: Trigger Jenkins via webhook
 Committer: Atharva Pailwan <atharvapailwan@mac.home>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 2 insertions(+), 1 deletion(-)
atharvapailwan@Atharvas-MacBook-Pro student-management-api % git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 11 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 312 bytes | 312.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/AtharvaAP17/student-management-api.git
   5fb48dd..582a952  main -> main
atharvapailwan@Atharvas-MacBook-Pro student-management-api % git add README.md
atharvapailwan@Atharvas-MacBook-Pro student-management-api % git commit -m "Test: Trigger Jenkins via webhook"
[main 7a67822] Test: Trigger Jenkins via webhook
 Committer: Atharva Pailwan <atharvapailwan@mac.home>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 1 insertion(+), 1 deletion(-)
atharvapailwan@Atharvas-MacBook-Pro student-management-api % git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 11 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 303 bytes | 303.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
```

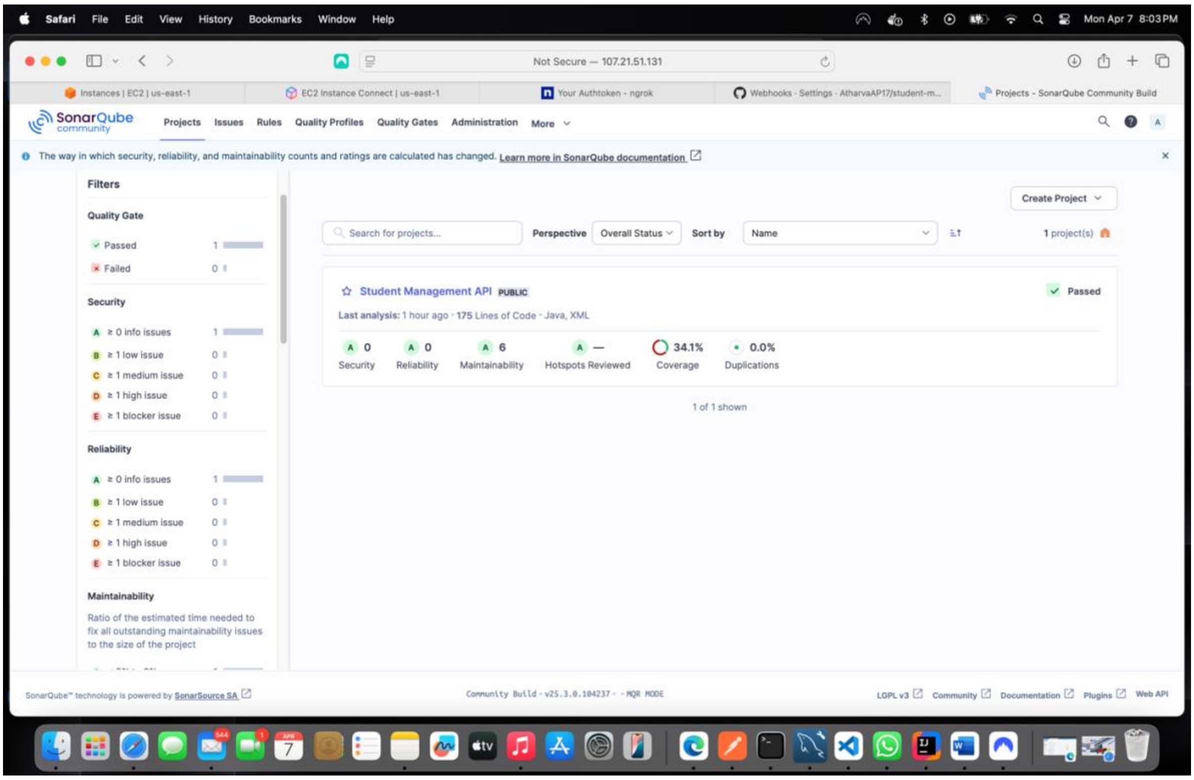
Pipeline starts automatically:

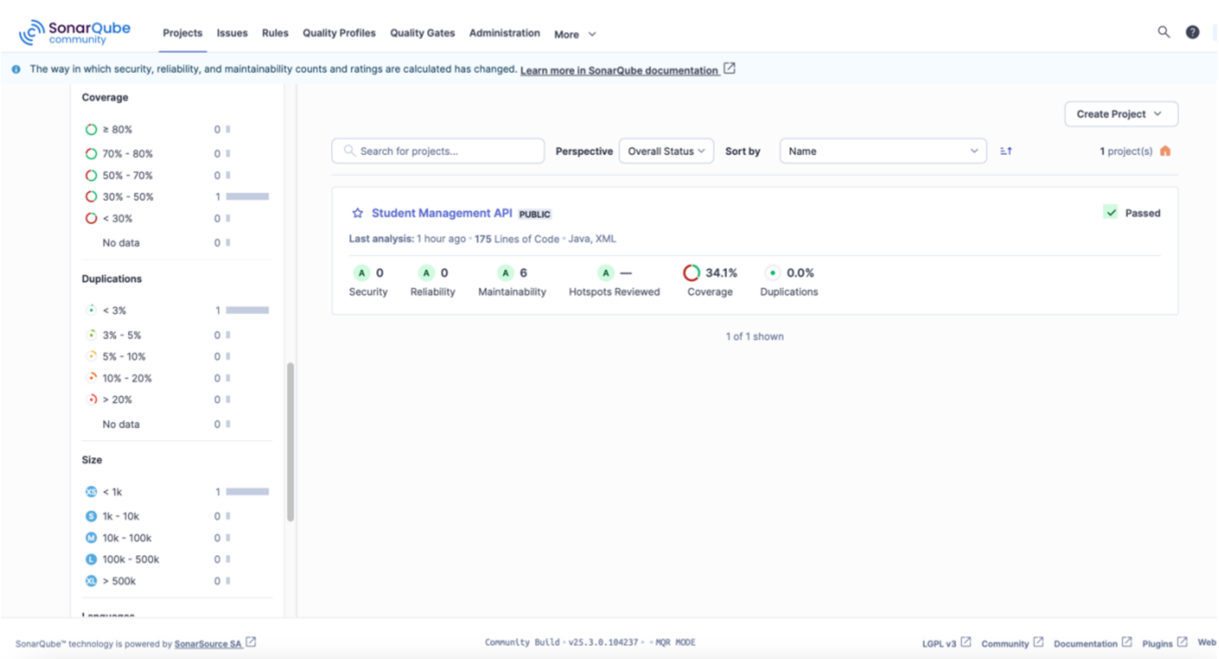


Github Webhook's recent deliverables confirming the trigger response:

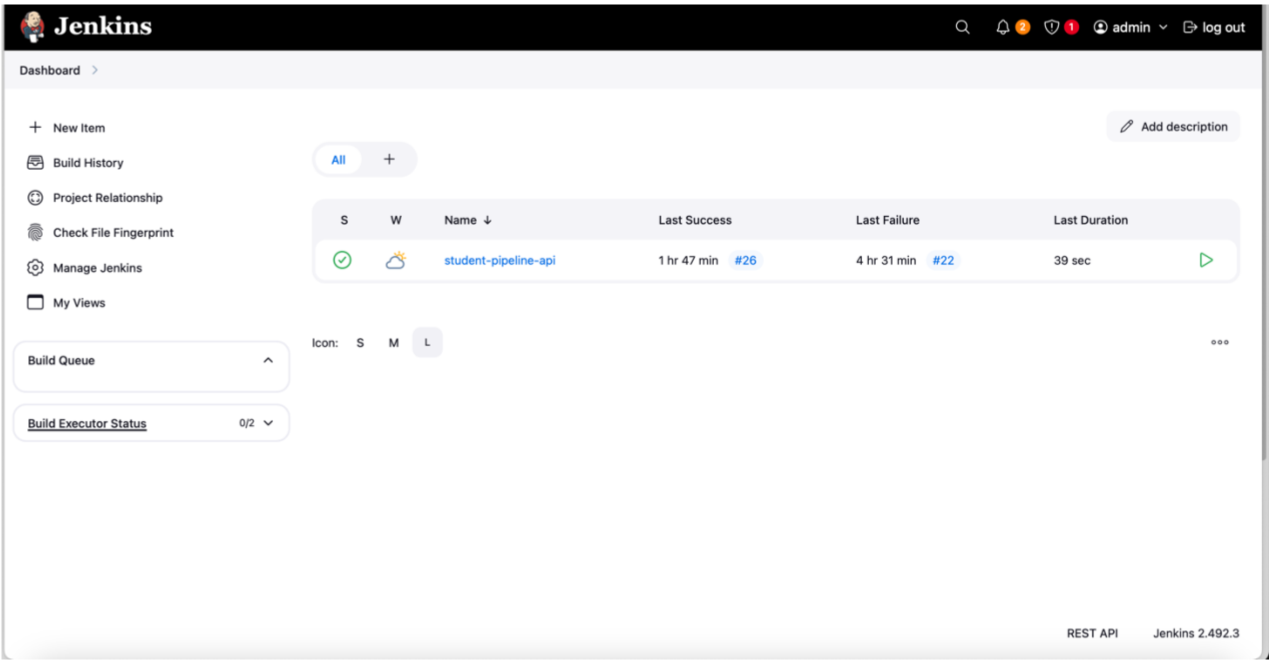


The SonarQube screenshots below illustrates the summary of these metrics and the quality gate result. Notably, the A-grade ratings in reliability, security, and maintainability indicate a clean bill of health in those categories, which is a strong positive outcome of the static analysis. The main area of improvement highlighted by SonarQube is the test coverage (which is marked, but still green in the context of the quality gate). Overall, the static analysis confirms that the Student Management API code is well-written and maintainable for the scope of the project.

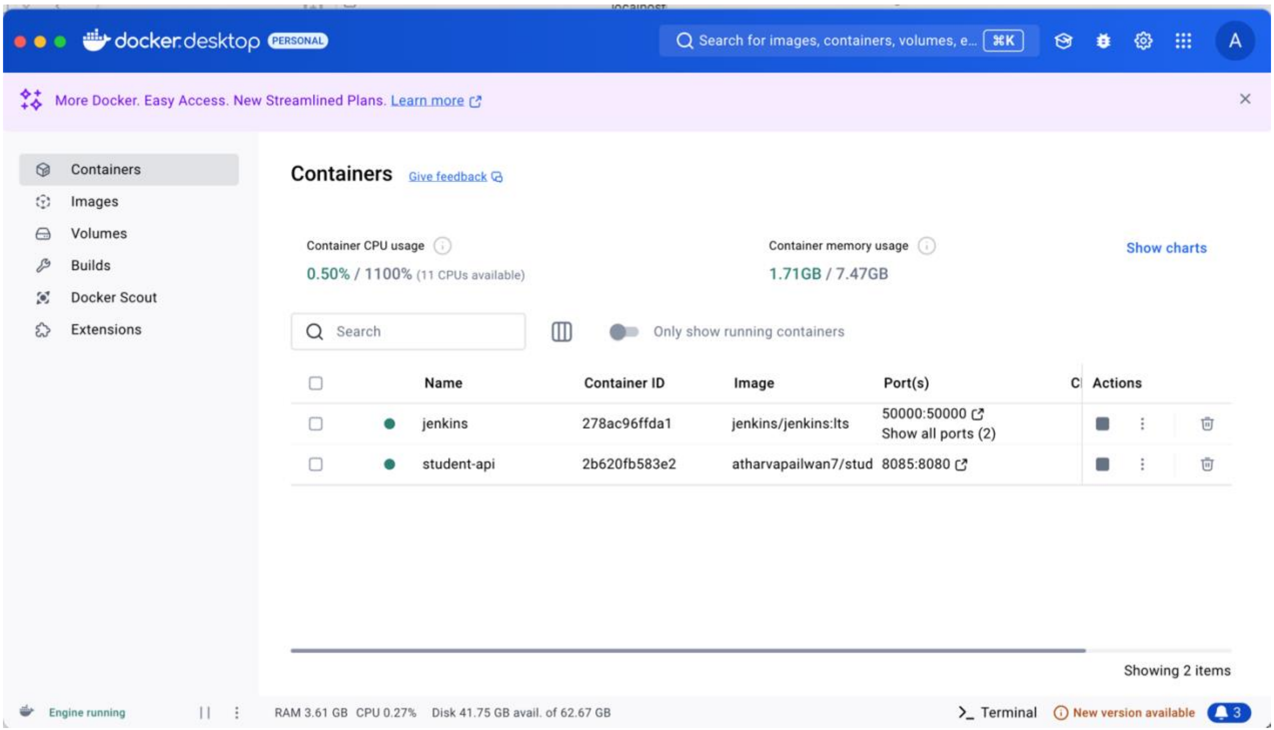




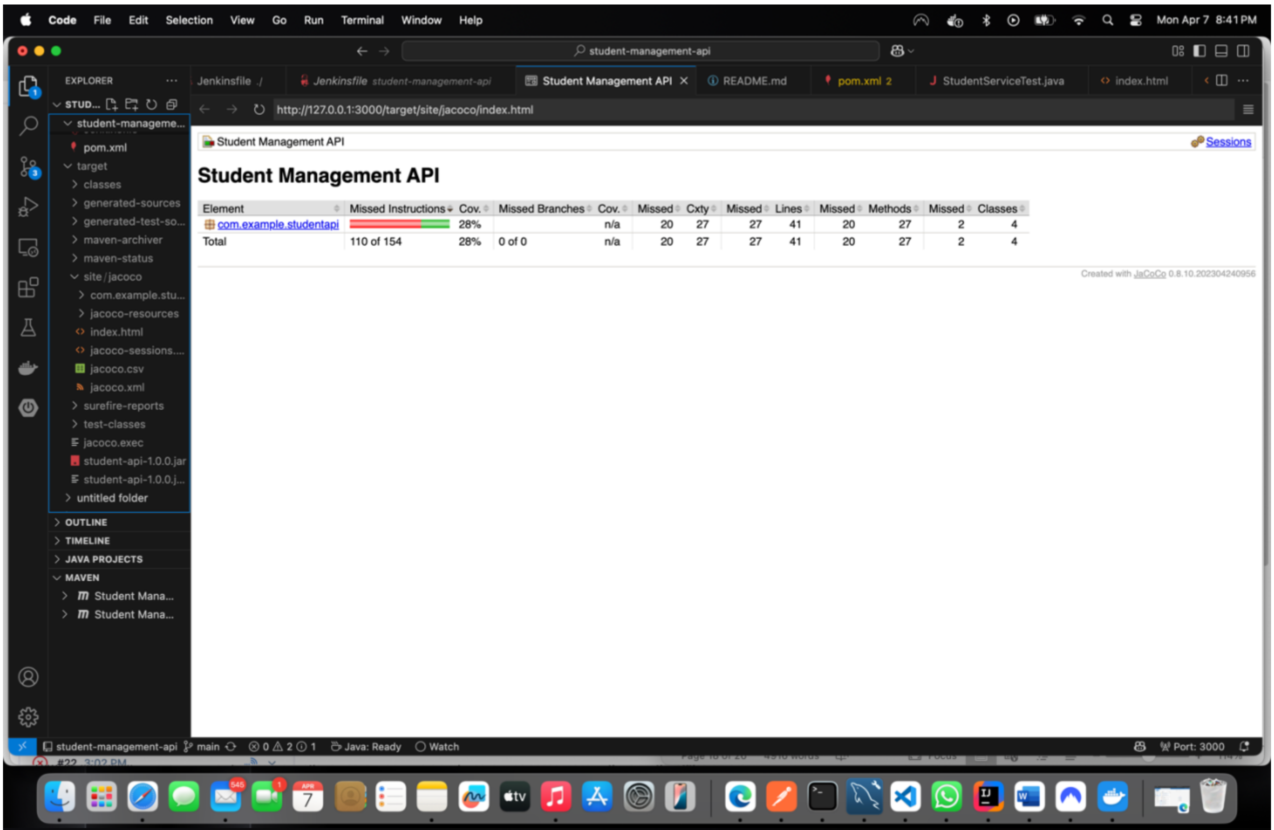
Jenkins Dashboard hosted on port 8080 showing the latest job status and last failure:

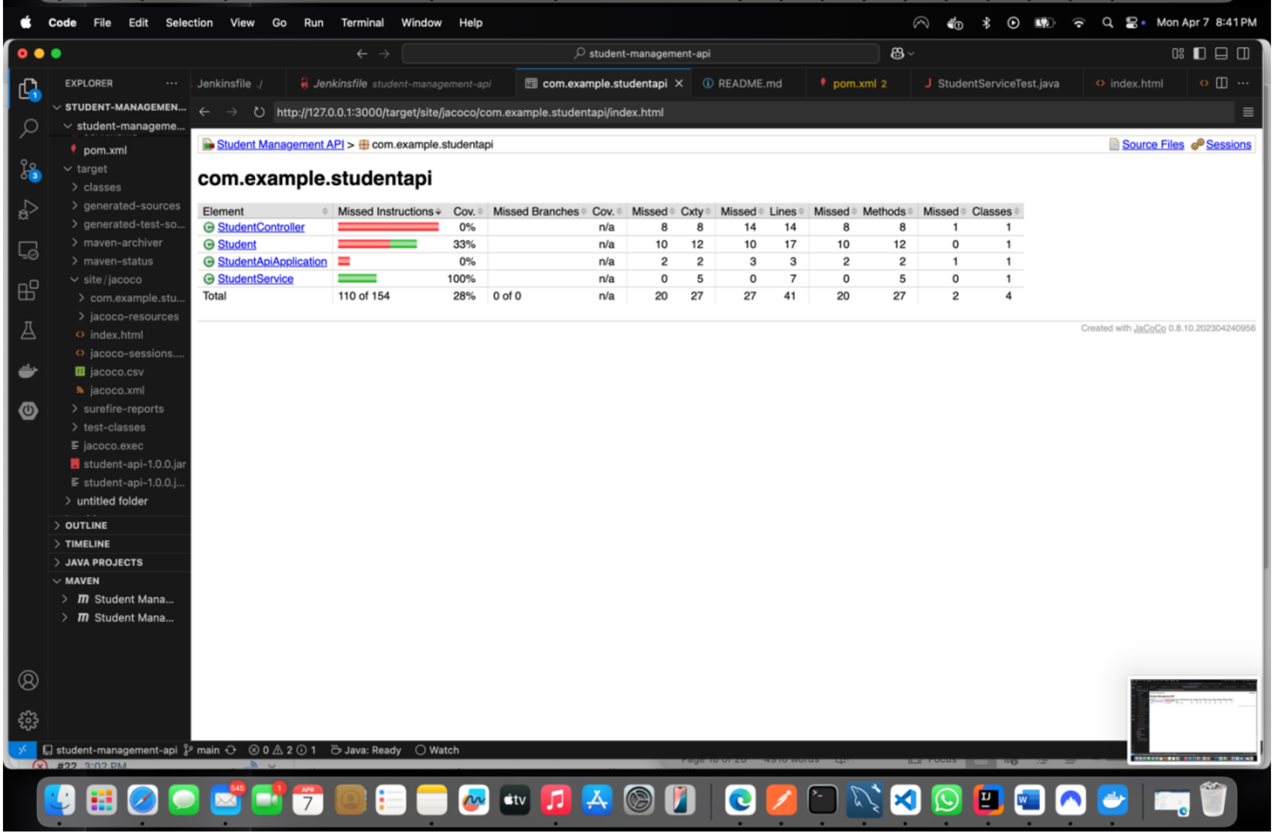


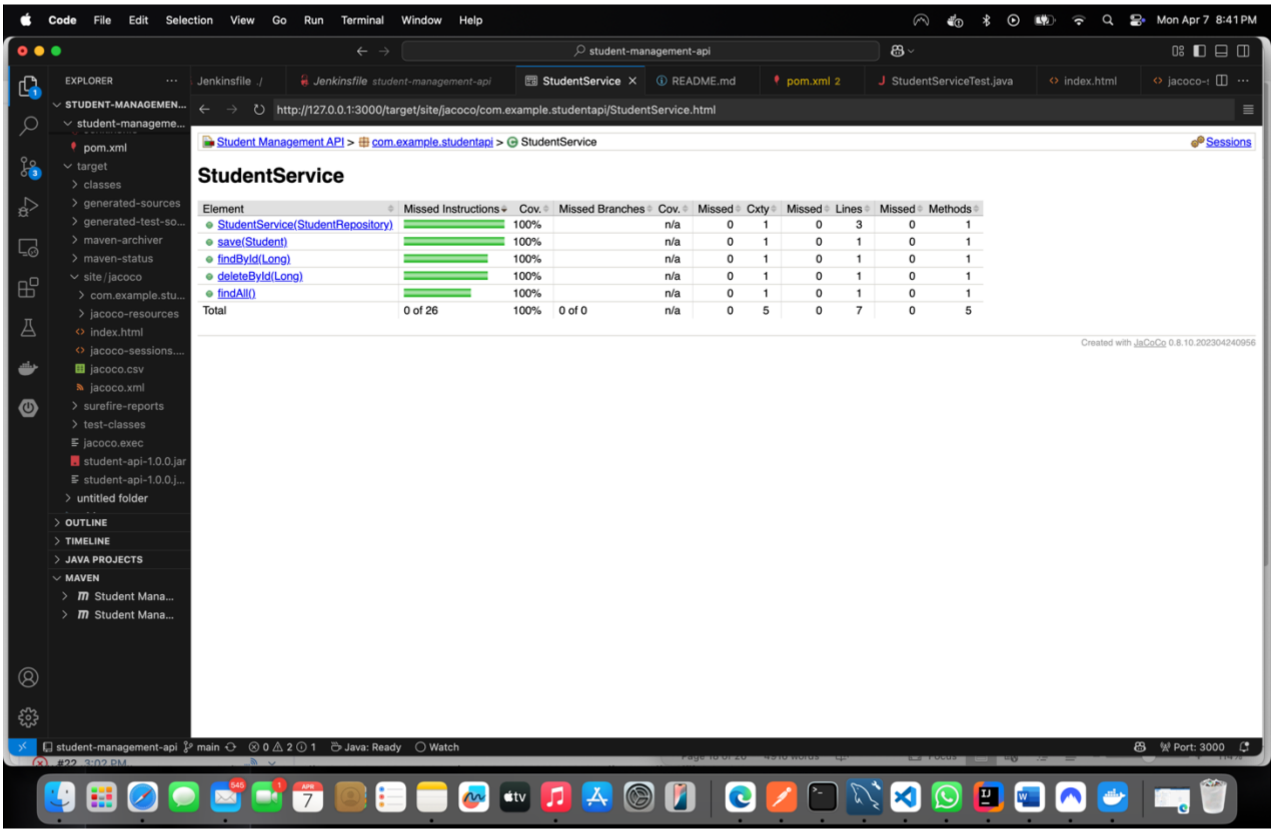
Running Jenkins and student-api Containers running on Docker desktop:

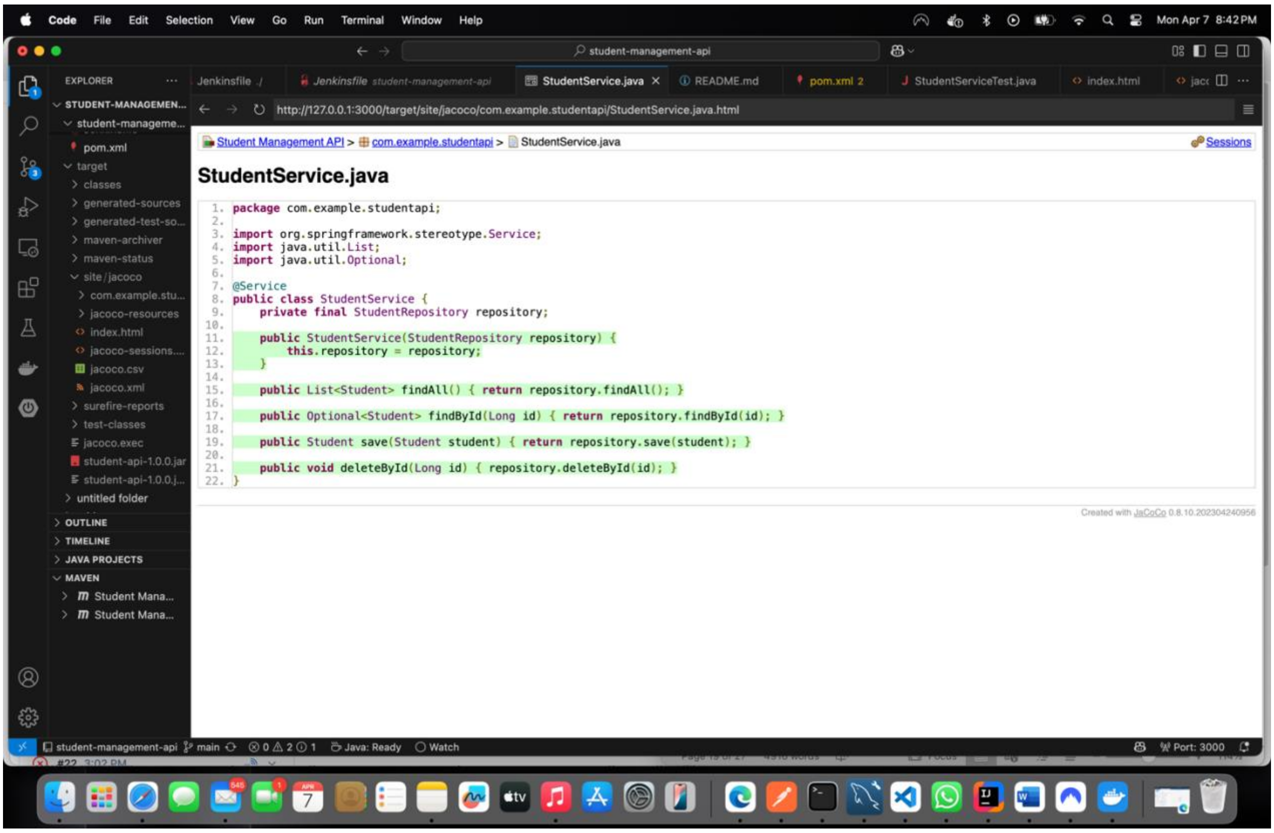


Code coverage using Jacoco:






http://127.0.0.1:3000/target/site/jacoco/com.example.studentapi/StudentService.html
Student Management API > com.example.studentapi > StudentService
StudentService
Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods
StudentService(StudentRepository) | 100% | n/a | 0 | 1 | 0 | 3 | 0 | 1
save(Student) | 100% | n/a | 0 | 1 | 0 | 1 | 0 | 1
findById(Long) | 100% | n/a | 0 | 1 | 0 | 1 | 0 | 1
deleteById(Long) | 100% | n/a | 0 | 1 | 0 | 1 | 0 | 1
findAll() | 100% | n/a | 0 | 1 | 0 | 1 | 0 | 1
Total | 0 of 26 | 100% | 0 of 0 | n/a | 0 | 5 | 0 | 7 | 0 | 5
Mon Apr 7 8:41 PM


http://127.0.0.1:3000/target/site/jacoco/com.example.studentapi/StudentService.java.html
Student Management API > com.example.studentapi > StudentService.java
StudentService.java
package com.example.studentapi;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;
@Service
public class StudentService {
private final StudentRepository repository;
public StudentService(StudentRepository repository) {
this.repository = repository;
}
public List<Student> findAll() { return repository.findAll(); }
public Optional<Student> findById(Long id) { return repository.findById(id); }
public Student save(Student student) { return repository.save(student); }
public void deleteById(Long id) { repository.deleteById(id); }
}
Mon Apr 7 8:42 PM

7. Conclusion and Evaluation:

The Student Management API project successfully demonstrates the implementation of a CI/CD pipeline for a Spring Boot microservice. All key deliverables of the assignment were achieved: the application code was developed with clear structure and layered architecture, and a fully automated pipeline was set up to build, test, analyze, containerize, and deploy the service. As a result, the project can be built and released at the push of a code commit, which is a significant outcome for continuous delivery practices. In terms of deliverables:

- The project's source is managed in Git and builds reproducibly with Maven, producing an executable JAR.
- A Jenkins pipeline automates the build and runs the test suite and static analysis on each push, ensuring continuous integration.
- Quality assurance is embedded into the process via SonarQube, and the project passed all quality gate metrics (with an A-rating in all categories and no critical issues).
- The service is containerized with Docker, and the pipeline automatically pushes the image to a registry.
- Deployment is handled through an Ansible script, demonstrating infrastructure-as-code and automated deployment to a server environment.
- Evidence of testing (JUnit results and a JaCoCo coverage report) and static analysis (SonarQube dashboard) have been provided, satisfying the assignment's reporting requirements.

Through this project, the following learning outcomes were realized: The importance of automated testing and early bug detection was underscored by the CI process, as even this small project benefited from having tests catch regressions immediately. Integrating SonarQube showed how maintaining code quality is an ongoing activity, not a one-time task. Working with Docker and Ansible in concert with Jenkins provided practical experience in deployment automation and the DevOps toolchain. Perhaps most importantly, the exercise reinforced the value of a coherent CI/CD pipeline — it increases confidence in each code change and lays the groundwork for rapid, reliable releases.

In conclusion, the Student Management API project met its objectives by delivering both a functional microservice and a robust continuous delivery pipeline. While there remains room for improvement (especially in expanding the test suite and refining deployment), the project as delivered provides a strong foundation. It illustrates how modern software engineering practices (CI/CD, automated testing, static analysis, and containerization) can be applied even to a relatively simple application to enhance its quality and maintainability. The experience gained from this assignment can be applied to more complex projects in the future, ensuring they too can achieve fast, high-quality, and continuous software delivery.