Reviews and CI Pipeline Document

Review Techniques and Results

Automated code review was the primary method of code review that was widely applied throughout the development process. Intellij IDEA completed the task, including error flagging and warnings. Although it was supplemental, manual review was nonetheless quite helpful. If the review was going to be completed by a different individual, such as an assigned reviewer), code and test distribution should happen first, then there should be a "over-the-shoulder" review, a talk about what the code is supposed to do, and a clear review report.

Ad-hoc review is another method of review that is employed in the code review procedure. This method claims that there is only an examination of the entire software and no predetermined plan for what needs to be done. In order to guarantee readability, proper code documentation, and a check for any missing Maven dependencies or potential problems, this technique was employed as part of the code quality review. We checked the code throughout this review to make sure all classes and public methods had the proper Javadoc documentation, and that naming conventions—like using camelCase for variable names—were adhered to.

In addition to other methods, I have employed log debugging within my code and interspersed it between various tests. This approach allows me to observe the intricate details of the program's behavior at specific moments, providing invaluable insights that enable me to effectively debug and refine my code for subsequent, more rigorous testing phases.

Also, code style maintained manually, it would have been automated using VCS like Git.

CI Pipeline Construction

Source Code Management with GitHub: The Continuous Integration (CI) workflow initiates in a robust GitHub repository. Here, developers actively contribute code to various branches. Each push to these branches automatically triggers the CI pipeline, marking the beginning of the automated build and test processes.

Automated Unit Testing with JUnit: Upon any code update, the pipeline launches an automated unit testing phase leveraging JUnit. This crucial stage focuses on thoroughly testing individual components, like the PathFinder module and other essential functionalities of PizzaDronz, to validate their correct operation in isolation.

Enhanced Automated Unit Testing with Code Coverage Analysis: In addition to using JUnit for unit testing, integrate a code coverage tool like JaCoCo. This measures how much of the codebase is covered by tests, ensuring a comprehensive test suite and identifying areas lacking test coverage.

Static Code Analysis and Security Vulnerability Scanning: Include static code analysis tools such as SonarQube, along with security vulnerability scanners like OWASP Dependency-Check. This step helps in identifying potential security vulnerabilities and maintaining code quality by enforcing coding standards and detecting anti-patterns.

Integration Testing via Mock Server and REST Integration: Following successful unit testing, the workflow advances to integration testing. This phase incorporates a mock server to emulate interactions with external REST services. The objective is to ensure seamless integration of different system components, effectively identifying any discrepancies in data handling and process interoperability.

Code Quality Analysis and Standard Enforcement: An integral part of the CI pipeline is the code quality analysis, implemented using specialized tools. This step is essential for ensuring adherence to predefined coding standards, identifying potential code issues early, and maintaining high-quality, clean code within the PizzaDronz project.

Automated Functional Testing for Comprehensive Validation: The pipeline integrates sophisticated automated functional testing, possibly using tools like TestNG. This stage is crucial for verifying the complete spectrum of PizzaDronz's functionalities, including critical aspects like argument validation, efficient order processing, and optimization of the pathfinding algorithms, ensuring they meet all specified requirements.

Performance Testing with Apache JMeter: A pivotal component of the pipeline is performance testing, executed using tools like Apache JMeter. This stage evaluates how PizzaDronz performs under various load conditions, focusing on compliance with critical performance benchmarks like the 60-second runtime constraint and identifying any performance-related bottlenecks.

Load and Stress Testing in Cloud-based Environments: Leverage cloud services to perform extensive load and stress testing. Tools like AWS Load Balancing or Azure Load Testing can simulate high traffic and usage, ensuring that PizzaDronz can handle peak loads.

Artifact Creation: JAR Files and JSON Documents: Upon successful completion of previous stages, the CI process facilitates the generation of artifacts. These include executable JAR files encapsulating PizzaDronz's core functionalities and sample JSON documents designed for testing scenarios, ensuring readiness for deployment.

Chaos Engineering for Resilience Testing: Implement chaos engineering principles by introducing controlled disruptions in the system (like server crashes or network failures) to test and improve the resilience and reliability of PizzaDronz.

Comprehensive Notification and Reporting System: To maintain high levels of communication and awareness, the CI pipeline incorporates an advanced notification and reporting system. This includes sending out automated alerts, such as email notifications, and generating detailed CI reports.

Developers are thus kept informed about the build status and any issues that arise during the CI process, enabling prompt and effective responses to any challenges.

Automating Some Aspects of Testing:

For Automatic of some aspects of of testing I would take these steps in order to do it:

1)Pinpoint the elements of the testing procedure that are repetitive and time-intensive.

2)Select a suitable testing framework or automation tool for the tests you plan to automate.

3)Develop test scripts or utilize record-and-playback functionalities to document the steps of the test.

4)Incorporate the automated tests into both the development workflow and testing environment.

5)Set up a routine schedule for the automated tests, such as integrating them into a continuous integration (CI) system.

6)Regularly review and interpret the outcomes of the tests to spot any failures or patterns.

7)Continuously revise and refine the automated tests to align with software changes and updates.

8)Automating these testing aspects will enhance time efficiency, ensure consistent and precise testing, and boost the overall productivity of the development cycle.

Demonstration of pipeline works as expected:

Source Code Management with GitHub:

-Demonstrate by pushing a small code change to your GitHub repository.

-Show the automatic triggering of the CI pipeline as a result of this push.

-This can be done by displaying the GitHub Actions log or a similar CI tool integrated with your repository.

Automated Unit Testing with JUnit:

-Execute the unit tests using JUnit as part of the CI process.

-Show the test results, focusing on how individual components like the PathFinder module are tested.

Enhanced Automated Unit Testing with Code Coverage Analysis:

-Integrate JaCoCo (or a similar tool) to analyze code coverage.

-After the tests run, display the code coverage report to show which parts of the code are well tested and identify areas with less coverage.

Static Code Analysis and Security Vulnerability Scanning:

-Run tools like SonarQube and OWASP Dependency-Check.

-Demonstrate the output from these tools, highlighting any detected security vulnerabilities or code quality issues.

Integration Testing via Mock Server and REST Integration:

Show the setup of a mock server and how it's used in integration tests.

Execute integration tests and display the results, demonstrating the interaction with mocked external REST services.

Code Quality Analysis and Standard Enforcement:

-Run the code quality analysis tools.

-Present the results, focusing on adherence to coding standards and identification of potential code issues.

Automated Functional Testing:

-Use a tool like TestNG to run functional tests.

-Demonstrate these tests in action and show how they validate critical aspects like order processing.

Performance Testing with Apache JMeter:

-Conduct performance tests using JMeter.

-Display the results, especially focusing on compliance with performance benchmarks like the 60-second runtime constraint.

Load and Stress Testing in Cloud-based Environments:

-Demonstrate load and stress testing using cloud services.

-Show how the system behaves under simulated high traffic and usage conditions.

Artifact Creation: JAR Files and JSON Documents:

-Show the process of artifact creation at the end of the CI pipeline.

-Present the generated JAR files and JSON documents, ready for deployment.

Chaos Engineering for Resilience Testing:

-Introduce controlled disruptions in the system.

-Demonstrate how the system copes with these disruptions, showcasing its resilience.

Comprehensive Notification and Reporting System:

-Finally, show how the CI pipeline sends automated alerts and generates detailed reports.

-Display an example of such a notification or report, emphasizing how it keeps developers informed.