### **A**

**Project Report**

**On**

**CONTINIOUS INTEGRATION AND TESTING OF A BACKEND APP USING JEST AND JENKINS**

Submitted in the partial fulfilment of the Devops Laboratory of

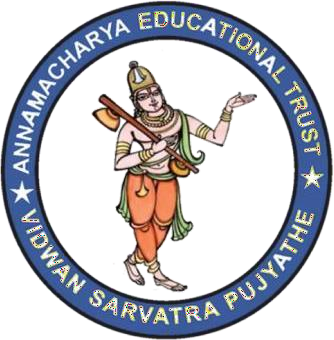
Master of Computer Applications

By

K. Hari Prasad

Regd.No:235N1F0035

Faculty  
DR. C. SHIVA KRISHNAIAH

Assistance professor

Department of Master of Computer Applications

Annamacharya P.G. college of Computer studies

New Boyanapalli, Rajampet

2024

**ABSTRACT**

This project focuses on creating an automated testing pipeline for a backend application using Jest as the testing framework and Jenkins as the CI/CD orchestrator. In modern software development, ensuring reliable and efficient testing is essential for maintaining code quality, reducing errors, and accelerating delivery timelines. Jest, a robust testing library, provides capabilities for unit, integration, and end-to-end testing, making it a versatile tool for backend application testing. Integrating Jest with Jenkins allows for an automated testing workflow that triggers on each code update, providing real-time feedback to developers and ensuring the application remains functional and reliable as it evolves.

The project includes configuring Jenkins to automate test runs, generate reports, and send notifications based on test outcomes. The automated pipeline enables early detection of bugs, improves test coverage, and fosters a culture of continuous testing and delivery. Additionally, by leveraging Jenkins’ ability to manage stages and parallel tasks, we optimize testing time and resources. This project underscores the value of automated backend testing, highlighting how Jest and Jenkins together create an efficient, scalable testing pipeline that supports agile development and enhances overall software quality.

**CONTEXT**

|  |  |
| --- | --- |
| **Section** | **Sub Section** |
| **1.Introduction** |  |
|  | 1.1 About Project |
|  | 1.2 Scope |
|  | 1.3 Objective |
| **2.Software Requirement Specifications** |  |
|  | 2.1 Architecture |
|  | 2.2 Product Features |
|  | 2.3 User constrains |
|  | 2.3.1 Technical |
|  | 2.3.2 Operational |
| **3.System Design** |  |
|  | 3.1 UML modelling |
|  | 3.2 data modelling |
|  | 3.2.2 ER Diagram |
|  | 3.2.3 Database diagram |
| **4.Implementation** |  |
| **5.conclusion** |  |

**1.INTRODUCTION**

**INTRODUCTION**

The Continuous Integration and Testing of a Backend Application using Jest and Jenkins project aims to establish an automated testing and deployment pipeline to ensure the stability, quality, and reliability of a backend application. Continuous integration (CI) practices are essential for modern software development, allowing teams to detect and address issues early by automatically building, testing, and validating code changes.

In this project, Jest serves as the primary testing framework, offering powerful tools for writing unit and integration tests for Node.js applications. Jenkins, a widely-used CI/CD automation server, will be used to create and manage the CI pipeline. Each time a developer pushes code changes to the version control repository, Jenkins will trigger the pipeline to perform essential tasks: checking out the latest code, installing dependencies, executing tests with Jest, and generating test and coverage reports.

The CI pipeline will provide immediate feedback on the code's health, helping the team quickly identify and fix issues. By using Jenkins to automate these processes, the project enables consistent quality control, rapid testing, and smoother deployments, ultimately leading to more reliable releases and streamlined development workflows. This setup is designed to improve collaboration, reduce integration issues, and increase development speed, making it an invaluable asset for teams aiming to maintain high-quality code in an efficient and automated manner.

**About the project**

The **Continuous Integration and Testing of a Backend Application using Jest and Jenkins** project is designed to automate the testing, integration, and reporting processes for a backend Node.js application. The key objectives are to ensure code quality, enhance development efficiency, and detect issues early in the development cycle by automating testing and integration tasks.

**Project Goals:**

1. **Automated Testing**: Using **Jest**, the project includes comprehensive testing for the backend application. Jest provides an environment to create unit tests, integration tests, and code coverage reports, helping maintain high-quality code by ensuring critical parts of the application are thoroughly tested with each code update.
2. **Continuous Integration (CI)**: The project uses **Jenkins** to set up a CI pipeline that automatically triggers on each code commit to the repository. Jenkins will pull the latest code changes, install dependencies, run Jest tests, generate reports, and notify the team of the test results. This process minimizes manual testing efforts and provides quick feedback to developers.
3. **Automated Reporting**: Jenkins also collects and archives test results and coverage reports after each pipeline run. This feature provides valuable insights into code coverage and areas needing improvement, helping the team maintain consistent quality.
4. **Efficient Team Collaboration**: By integrating Jenkins with the team’s messaging platform (e.g., Slack or email), the pipeline will notify the team of test statuses, allowing everyone to stay updated on code quality and enabling faster issue resolution.

**Key Components: -**

The **Key Components** of the Continuous Integration and Testing of a Backend Application using Jest and Jenkins project include:

1. **Jest Testing Framework**:

Jest is a JavaScript testing framework used for writing and running tests in Node.js applications.

It provides tools for unit testing, integration testing, and test coverage reporting, helping to ensure that each part of the backend application functions correctly.

Jest’s built-in assertion library and mocking capabilities make it easy to test different parts of the code in isolation.

1. **Jenkins CI Server**:

Jenkins is an open-source CI/CD automation server that allows us to set up and manage the continuous integration pipeline.

It automates the process of building, testing, and deploying the application, ensuring quick feedback on the status of the codebase with every commit.

Jenkins is integrated with version control systems like Git, allowing automatic triggering of the pipeline whenever changes are pushed to the repository.

1. **Version Control System (e.g., Git)**:

A Git repository (such as GitHub, GitLab, or Bitbucket) is used for managing and storing code changes.

It serves as the source for Jenkins to pull the latest code updates, ensuring that the CI pipeline runs on the most recent version of the code.

Webhooks can be set up to automatically trigger Jenkins jobs on code changes.

1. **Jenkins Pipeline (Jenkins file)**:

The Jenkins file defines the CI pipeline structure, detailing each stage and the specific actions required to test and build the application.

It specifies stages such as checking out code, installing dependencies, running tests with Jest, and archiving test results and coverage reports.

The pipeline is configurable and can be customized to add additional stages if needed (e.g., deployment).

1. **Automated Test Reporting and Notifications**:

Jenkins collects and publishes test reports and code coverage reports after each pipeline run.

Test results are displayed on the Jenkins dashboard, providing developers with immediate feedback on code quality.

Notification plugins (e.g., Slack or Email) send alerts when tests fail, allowing the team to take quick action to resolve issues.

**1.2 scope**

The scope of this project is to design and implement an automated system for managing system and application logs through processes of rotation, archiving, and deletion. The project aims to configure log rotation policies, define retention and deletion rules, and establish an archiving mechanism that uses both local storage and optional cloud storage for scalable, long-term log preservation.

**Key Areas of Scope**

**1.Test automation**

- Design and implement unit, integration, and functional tests using **Jest** to validate the backend application’s functionality.

-Ensure comprehensive test coverage across the application’s core components to catch bugs early and verify new features without manual intervention.

**2.Pipeline Setup and Configuration**

- Set up a **Jenkins CI pipeline** to automate the build, test, and integration processes.

- Configure Jenkins to pull the latest code changes from the Git repository, run tests automatically on each commit, and generate test and coverage reports.

**3. Automated Test Reporting and Alerts:**

- Implement automated reporting in Jenkins to display test results and code coverage after each pipeline run.

- Configure notification systems (e.g., email or Slack) to alert developers of test failures or code quality issues, ensuring rapid response to problems.

**4. Code Coverage Tracking:**

- Utilize Jest’s coverage analysis tools to ensure adequate test coverage across the application.

- Integrate coverage reports into Jenkins to monitor and maintain a minimum coverage threshold, promoting a high level of code quality.

**5. Integration with Version Control (Git):**

 Set up webhooks and integrate Jenkins with a Git repository to trigger the pipeline on each push or pull request.

 Implement a streamlined workflow that ensures only tested and verified code is merged into the main branch.

**6. Dependency Management and Environment Consistency:**

 Automate dependency installation and ensure environment consistency for each pipeline run, using tools like npm.

 Maintain version control of dependencies to prevent compatibility issues and ensure each test run is conducted in a stable environment.

**1.3 objective**

The objective of this project is to implement an automated log management solution that efficiently handles log rotation, archiving, and cleanup to optimize storage, maintain system performance, and ensure compliance with data retention policies. The system will minimize manual intervention by automatically rotating logs based on customizable schedules, archiving them in secure, long-term storage locations (both local and cloud), and deleting outdated logs according to retention policies.

**Key objectives**

**1. Automate Log Rotation**

- Implement an automated log rotation system to manage log file size and prevent disk overuse, ensuring consistent system performance.

**2.Implement Efficient Log Archiving**

- Establish a system to compress and archive rotated logs in both local and cloud storage, supporting scalable and accessible long-term storage.

**3.Enforce Retention and Cleanup Policies**

- Automate the deletion of logs beyond defined retention periods to optimize storage usage and maintain compliance with organizational and regulatory requirements.

**4. Integrate Monitoring and Alerting**

- Set up monitoring and alerting to track the success of log management tasks and promptly notify administrators of any issues or failures in the rotation and archiving processes.

**5. Enhance Security and Access Control**

- Enforce access restrictions and encryption for archived logs, especially in cloud storage, to ensure data security and protect sensitive information.

**6.Provide Comprehensive Documentation and Training**

- Develop detailed documentation and training resources for administrators to understand configuration, maintenance, and troubleshooting, ensuring effective operation and sustainability of the log management solution.

**2.software requirement specifications(SRS)**

**SOFTWARE REQUIREMENT SPECIFICATIONS(srs)**

The **Software Requirements Specification (SRS)** for the **Continuous Integration and Testing of a Backend Application using Jest and Jenkins** project provides a detailed description of the functional and non-functional requirements, system features, and constraints for automating testing and integration processes. The SRS helps ensure that the project meets its objectives and provides a roadmap for implementation.

* 1. **Functional Requirements**

**2.1.1. Version Control Integration**

**Requirement**: The CI system shall be integrated with a Git-based version control system (e.g., GitHub, GitLab, Bitbucket).

**Functionality**:

Automatically trigger the CI pipeline upon each commit or pull request.

Fetch the latest code changes from the repository to run tests.

**2.1.2. CI Pipeline in Jenkins**

**Requirement**: The Jenkins server shall manage and automate the CI pipeline.

**Functionality**:

**Pipeline stages**: Code checkout, dependency installation, test execution, and reporting.

Configurable Jenkinsfile to manage pipeline tasks and define steps and conditions.

**2.1.3. Automated Testing with Jest**

**Requirement**: The system shall execute Jest tests to validate the functionality of the backend application.

**Functionality**:

Run unit, integration, and functional tests with Jest.

Generate code coverage reports to track the percentage of code tested.

**2.1.4. Reporting and Notifications**

**Requirement:** The CI system shall provide feedback on test results.

**Functionality**:

Generate test reports and display them on the Jenkins dashboard.

Send notifications of test results via email or messaging platforms (e.g., Slack) in case of test failures.

**2.1.5. Test Coverage Analysis**

**Requirement**: The system shall track and report test coverage.

**Functionality**:

Jest shall provide code coverage metrics.

Jenkins shall archive and display test coverage reports, enabling continuous monitoring.

**2.1.6. Dependency Management**

**Requirement**: The system shall manage and install dependencies required for testing.

**Functionality**:

Install dependencies (using npm) specified in package.json.

Verify all dependencies are compatible with the Jenkins environment for each test run.

**2.2 product Features**

**1. Automated Testing with Jest**

* **Unit Testing**: Provides a structured approach for validating individual functions and modules, ensuring each component of the backend application works as expected in isolation.
* **Integration Testing**: Verifies the interaction between different modules, ensuring they work together as expected.
* **Mocking and Assertions**: Jest's built-in mocking and assertion libraries simulate dependencies and verify code behavior accurately.
* **Code Coverage Analysis**: Tracks the percentage of code covered by tests and identifies untested areas for improvement.

**2. Jenkins CI Pipeline**

* **Pipeline Automation**: Automates the build, test, and reporting process using Jenkins, providing a smooth and repeatable workflow for testing code changes.
* **Pipeline as Code**: Uses a Jenkinsfile to define the CI pipeline, making the configuration version-controlled and easy to update.
* **Pipeline Stages**:
  + **Checkout Code**: Pulls the latest code from the repository.
  + **Install Dependencies**: Ensures all required packages and dependencies are installed before tests are run.
  + **Run Tests**: Executes all Jest tests automatically with each new commit.
  + **Generate Reports**: Compiles test results and code coverage data into reports.

**3. Continuous Integration with Git Integration**

* **Automatic Triggering**: The pipeline automatically starts on every code commit or pull request, ensuring that the latest code is tested continuously.
* **Branch Management**: Supports multiple branches to allow for testing in different environments (e.g., development, testing, staging).
* **Pull Request Testing**: Tests code changes in pull requests before they are merged, reducing the likelihood of introducing bugs into the main codebase.

**4. Automated Reporting and Notifications**

* **Real-Time Test Reports**: Displays detailed test results on the Jenkins dashboard for each pipeline run, providing visibility into the status of the codebase.
* **Code Coverage Reports**: Summarizes the percentage of code tested, helping maintain a high standard of test coverage.
* **Notifications**: Sends alerts through email, Slack, or other messaging services when tests fail, keeping developers informed and enabling quick response.

**5. Dependency Management**

* **Environment Setup**: Automatically installs dependencies using npm to ensure all required packages are available for each test run.
* **Environment Consistency**: Ensures a consistent testing environment across pipeline runs, reducing errors caused by dependency issues.

**6. Error and Failure Tracking**

* **Error Logging**: Provides logs for each test failure, helping developers quickly diagnose and fix issues.
* **Pipeline Failure Alerts**: Notifies team members of pipeline failures due to code issues or dependency errors, enabling a quick response and minimizing downtime.

**7. Scalability and Flexibility**

* **Modular Test Cases**: Allows the addition or modification of tests as the application grows and new features are added.
* **Expandable Pipeline**: Designed to integrate additional stages (e.g., for deployment) or features (e.g., load testing) as the project requirements evolve.
* **Parallel Execution**: Configurable for parallel test execution to handle large test suites and minimize CI pipeline runtime.
  1. **tools and techonologies used**

The **Tools and Technologies** used in the Continuous Integration and Testing of a Backend Application using Jest and Jenkins project are essential for implementing and automating the testing, integration, and reporting processes. These tools enhance the reliability, efficiency, and maintainability of the project’s CI/CD pipeline.

**1. Jest**

* **Purpose**: Testing framework
* **Use**: Jest is a JavaScript testing framework used for creating and running unit tests, integration tests, and code coverage reports. It provides powerful tools for assertion, mocking, and coverage analysis, ensuring the backend application is thoroughly tested.
* **Features**:
  + Supports automated test runs
  + Built-in mocking capabilities
  + Code coverage tracking and reporting
  + Snapshot testing for verifying output consistency

**2. Jenkins**

* **Purpose**: Continuous Integration (CI) and Continuous Deployment (CD) server
* **Use**: Jenkins is used to create and manage the CI pipeline, automating the build, test, and deployment process. It triggers test runs on each commit, manages pipeline stages, and provides feedback on the status of code changes.
* **Features**:
  + Pipeline-as-code using Jenkinsfile
  + Integration with version control systems
  + Real-time test reports and logs
  + Role-based access control and secure credentials management
  + Integration with deployment tools for optional CD

**3. Git (GitHub, GitLab, or Bitbucket)**

* **Purpose**: Version control system
* **Use**: Git is used for source code management, enabling developers to track changes, create branches, and collaborate on the codebase. Integration with Jenkins allows the pipeline to trigger on code commits and pull requests.
* **Features**:
  + Version history and rollback capabilities
  + Branching for feature development and release management
  + Pull requests and code review workflows
  + Webhooks for automatic CI pipeline triggering

**4. Node.js and npm**

* **Purpose**: Runtime environment and package manager
* **Use**: Node.js provides the runtime environment for executing JavaScript on the server side, while npm (Node Package Manager) is used to install and manage dependencies. The project relies on npm to ensure all dependencies are up-to-date for each pipeline run.
* **Features**:
  + Cross-platform JavaScript runtime
  + Large ecosystem of packages and libraries
  + Dependency versioning and installation with package.json
  + Scripts for automating commands (e.g., testing with Jest)

**5. Slack or Email (for Notifications)**

* **Purpose**: Notification system
* **Use**: Slack or email is used to send automated notifications when tests fail or when the CI pipeline completes. This integration keeps the development team informed about code quality and pipeline status.
* **Features**:
  + Real-time alerts for quick response to issues
  + Customizable notifications for pipeline events (e.g., failure, success)
  + Integration with Jenkins for direct alerts from the CI/CD server

**6. npm Scripts**

* **Purpose**: Task automation within Node.js applications
* **Use**: npm scripts are used to automate commands like running tests, generating coverage reports, and managing dependencies. These scripts streamline the process for developers and integrate smoothly with Jenkins.
* **Features**:
  + Customizable commands in package.json
  + Environment-specific script execution
  + Simplifies test execution, environment setup, and build processes

**7. Code Coverage Analysis Tools (e.g., Jest Coverage)**

* **Purpose**: Track test coverage metrics
* **Use**: Jest provides code coverage analysis to show the percentage of code that is tested, helping the team monitor coverage and identify untested areas. Coverage reports are displayed in Jenkins and used as a metric for code quality.
* **Features**:
  + Generates detailed coverage reports
  + Helps maintain high test coverage
  + Allows configuration to set minimum coverage thresholds
  + Supports visual and tabular representation of code coverage

**8. Docker (Optional for Containerized Testing and Deployment)**

* **Purpose**: Containerization platform
* **Use**: Docker can be used to create a consistent testing and deployment environment by running the CI pipeline in containers. It helps manage dependencies and environment consistency, ensuring that tests run the same way across different systems.
* **Features**:
  + Isolates the testing environment
  + Ensures dependency and environment consistency
  + Supports containerized deployment to staging or production

**9. IDE (e.g., Visual Studio Code)**

* **Purpose**: Code editor for development
* **Use**: An integrated development environment (IDE) like Visual Studio Code provides developers with the tools to write, test, and debug code effectively. IDE integrations with Git, linting tools, and testing frameworks like Jest help streamline the development process.
* **Features**:
  + Built-in support for Git version control
  + Extensions for Jest and CI integration
  + Linting, debugging, and formatting tools
  + Terminal support for running npm scripts and commands directly

**10. Shell Script (for Custom Jenkinsfile Scripting)**

* **Purpose**: Custom pipeline scripting
* **Use**: Shell scripts can be used within the Jenkinsfile to define custom stages or commands, offering flexibility to customize the CI pipeline. It allows for specific setups, configurations, and tasks that need to run at different stages.
* **Features**:
  + Runs specific commands or tasks during the pipeline
  + Supports automation beyond what’s predefined in npm scripts
  + Customizable for unique CI/CD needs

**2.4 user constrains:**

For the **Continuous Integration and Testing of a Backend Application using Jest and Jenkins**, certain **user, technical, and operational constraints** need to be considered to ensure efficient setup, smooth operation, and alignment with organizational policies. These constraints can impact system functionality, usability, and maintainability, so they must be factored into the project’s design and implementation.

**1. User Constraints**

** Technical Proficiency**: Team members may have varying levels of proficiency with CI/CD tools, which can impact the effectiveness of using and maintaining Jenkins and Jest. Training or documentation may be required to bring everyone to the necessary skill level.

 **Access Control and Permissions**: Only authorized users should be able to configure or modify the CI pipeline. Restricted access based on roles is required to prevent unauthorized modifications and maintain pipeline integrity.

 **Time Availability**: The CI process requires commitment from developers to address test failures, fix code quality issues, and manage merge conflicts. This may impact time allocation for new feature development.

**2.Technical Constraints**

 **Hardware and Server Requirements**: Jenkins requires a dedicated server or cloud instance with sufficient processing power, memory, and storage to handle builds, run tests, and store artifacts, especially for larger projects with extensive test suites.

 **Tool Compatibility**: Jenkins and Jest must be compatible with the backend application’s existing codebase and dependencies. Compatibility issues may arise if the backend application relies on specific libraries, versions, or languages not fully supported by Jenkins or Jest.

 **Dependency Management**: Ensuring consistent dependency versions is crucial for stable CI pipeline performance. Conflicts or mismatches in versions could cause pipeline failures and test inconsistencies.

 **Network Constraints**: Since Jenkins may need to pull code from remote repositories and send notifications, a reliable and fast network connection is essential. Limited network bandwidth or downtime can lead to pipeline delays and hinder performance.

 **Code Coverage Threshold**: The CI pipeline may enforce a minimum code coverage threshold (e.g., 80%) to maintain code quality. Achieving and maintaining this threshold could be challenging, especially as the codebase grows.

 **Version Control Limitations**: The CI pipeline’s integration with the version control system relies on webhook configurations and branch policies. Any limitations or restrictions within the chosen version control platform (e.g., GitHub or GitLab) could impact the functionality and responsiveness of the pipeline.

 **Testing Environment**: Jenkins might not replicate all production configurations, meaning certain environment-specific issues may not be caught in CI tests, which could require additional testing environments.

**3.Operational Constraints**

 **Resource Allocation**: Jenkins requires ongoing resource allocation for server hosting, maintenance, and upgrades. Limited resources may affect the system’s performance and reliability, particularly for larger teams or high-frequency deployments.

 **Security Policies**: Organizational security policies may restrict access to certain CI/CD configurations or mandate specific security protocols for managing credentials and sensitive data in Jenkins.

 **Compliance and Auditing**: Organizations in regulated industries may require CI/CD pipelines to comply with specific regulatory standards (e.g., GDPR, HIPAA). This can introduce additional steps or reviews in the CI pipeline to ensure compliance.

 **Pipeline Execution Time**: Long-running tests and builds could slow down development cycles. Optimizing pipeline stages is essential to maintain a reasonable pipeline runtime without sacrificing test coverage.

 **Downtime and Maintenance**: Regular maintenance of Jenkins, along with updates for Jest and other dependencies, may require downtime or temporary pipeline suspension, which can impact development timelines.

 **Error Tracking and Monitoring**: Effective CI operation relies on prompt detection and resolution of issues within the pipeline. Lack of monitoring tools or alerting mechanisms can delay the identification and resolution of test failures, impacting delivery schedules.

 **Backup and Disaster Recovery**: Jenkins jobs, configurations, and test artifacts should be backed up regularly to prevent data loss. However, limited resources for backup and recovery can be a constraint in ensuring pipeline continuity.

**3.SYSTEM DESIGN**

**3.SYSTEM DESIGN**

The **System Design** for Continuous Integration and Testing of a Backend Application using Jest and Jenkins involves several key components to ensure reliable code quality, automated testing, and a seamless integration process. This design outlines how these components interact and function to create an effective CI pipeline.

1. **Architecture Overview**

The architecture is divided into several modules:

**Version Control System (VCS)**: Stores the application’s source code and triggers the CI pipeline on code changes.

**Jenkins CI Server**: Manages the CI pipeline, including code checkout, build, testing, and reporting stages.

**Testing Module (Jest)**: Executes unit and integration tests, generates test results, and tracks code coverage.

**Notification and Reporting Module**: Provides feedback to developers via notifications and detailed reports.

**2. Component Design**

**2.1 Version Control System (Git)**

**Components**: Git repository (e.g., GitHub, GitLab, Bitbucket)

**Responsibilities**:

Stores source code and tracks changes through branches and commits.

Webhooks or triggers initiate Jenkins pipelines upon code commits or pull requests.

Branch management and pull requests allow for testing and code review workflows.

**2.2 Jenkins CI Server**

**Components**: Jenkins server, Jenkinsfile (pipeline configuration), Jenkins Plugins

**Responsibilities**:

**Pipeline Automation**: The Jenkins pipeline is configured using a Jenkinsfile that defines stages for checkout, dependency installation, testing, and reporting.

**Environment Configuration**: Jenkins provides isolated build environments, such as Docker containers or virtual environments, for consistent testing.

**Build Triggers**: Jenkins initiates builds automatically on code changes, enabling continuous integration.

**Stages:**

**Code Checkout:** Jenkins pulls the latest code changes from the Git repository.

**Dependency Installation:** Installs project dependencies via npm.

**Testing:** Runs Jest tests for unit and integration validation.

**Reporting and Archiving:** Generates and archives test reports and coverage results.

**Notifications:** Sends alerts on build success or failure, as configured in the Jenkinsfile.

**2.3 Testing Module (Jest)**

**Components:** Jest testing framework, test scripts, mock data

**Responsibilities:**

**Test Execution:** Jest runs the defined unit and integration tests for backend application components.

**Mocking and Coverage:** Mocks dependencies as needed to isolate tests and generates coverage reports to measure code quality.

**Test Result Generation:** Provides results in formats compatible with Jenkins for reporting.

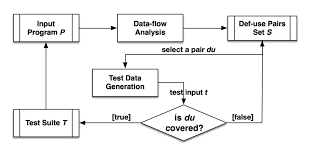
**2.4 Notification and Reporting Module**

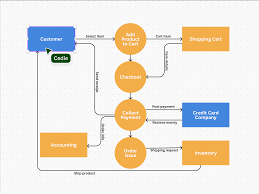
**Components:** Slack or email notifications, Jenkins dashboard

**Responsibilities:**

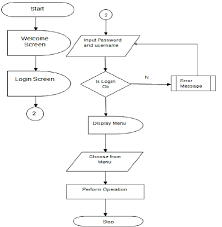
**Real-Time Feedback:** Notifies developers of test outcomes through channels like Slack or email.

**3.3 Data flow diagrams**





**3.4 E.R. DIAGRAMS**



**3.5 Data base design**

For an automated log rotation and archiving system, a database is typically used to track and manage metadata related to log files, rotation schedules, archiving status, retention policies, and monitoring information. Here’s a database design that captures these aspects effectively.

**Database Tables**

1. **Logs Table**
   * **Purpose**: Stores metadata for each log file generated by the applications, including details about the file, its rotation history, and archival status.

|  |  |  |
| --- | --- | --- |
| Field | type | description |
| Log\_id | INT (PK) | Unique identifier for each log file |
| Source | VARCHAR (255) | Application or service that generated the log |
| File\_path | VARCHAR (255) | Path to the log file on the server |
| File\_size | BIGINT | Size of the log file (in bytes) |
| Rotation\_date | DATETIME | Date and time when the log was last rotated |
| Archived | BOOLEAN | Status indicating if the log is archived |
| Archive\_path | VARCHAR (255) | Path to the archived file in cloud/local storage |
| Encryption\_status | BOOLEAN | Indicates if the log file is encrypted |
| Retention\_period | |  | | --- | |  |  |  | | --- | | INT | | Retention period in days |

**2.Rotation Policies Table**

* **Purpose**: Stores rotation policies that can be applied to different logs based on source, size, and frequency requirements.

|  |  |  |
| --- | --- | --- |
| Field | Type | description |
| policy\_id | INT (PK) | Unique identifier for each rotation policy |
| source | VARCHAR(255) | Source application/service this policy applies to |
| rotation\_frequency | ENUM('daily', 'weekly', 'monthly') | Frequency of rotation |
| max\_size | BIGINT | Maximum file size before rotation (in bytes) |
| compression | BOOLEAN | Indicates if log should be compressed post-rotation |
| created\_at | DATETIME | Policy creation date |
| updated\_at | DATETIME | Last updated timestamp |

**3.Archive Records Table**

* **Purpose**: Tracks the status and details of archived log files, including information about location, storage provider, and integrity checks.

|  |  |  |
| --- | --- | --- |
| FIELD | TYPE | DESCRIPTION |
| archive\_id | INT (PK) | Unique identifier for each archived record |
| log\_id | INT (FK) | Foreign key linking to the original log entry |
| cloud\_provider | VARCHAR(50) | Cloud provider (e.g., AWS, GCP, Azure) |
| archive\_path | |  | | --- | |  |  |  | | --- | | VARCHAR(255) | | Path/URL to the archived file |
| storage\_class | VARCHAR(50) | Cloud storage class (e.g., Standard, Archive) |

**4.IMPLEMENTATION**

**4.Implementation**

**1. Environment Setup**

* **Choose the Infrastructure**: Decide on the servers (e.g., on-premises, cloud instances) where log rotation and archival scripts will run.
* **Setup Cloud Storage**: Configure cloud storage (e.g., AWS S3, Google Cloud Storage) with secure access credentials for archiving logs.

**2. Database Setup**

* **Install and Configure Database**: Use PostgreSQL or MySQL for a relational database (or MongoDB for NoSQL). Set up the database according to the schema defined in the previous response.
* **Create Tables**: Execute SQL scripts to create tables for **Logs**, **Rotation Policies**, **Archive Records**, **Retention Policies**, **Alerts**, and **User Access**.

**3. Log Rotation Setup**

* **Install Logrotate**: Install Logrotate on the server (usually available by default on Linux).
* **Configure Logrotate**:
  + Define configurations for each log type in /etc/logrotate.d/ or a custom directory.
  + Specify criteria (e.g., size, frequency) based on your **Rotation Policies** table.
* **Custom Rotation Script**: For specialized rotation needs, write custom shell scripts to handle rotation, then schedule with Cron.

# Example Logrotate configuration file (app.log in /etc/logrotate.d/app)

/var/log/app/\*.log {

size 100M

rotate 7

compress

missingok

notifempty

create 0640 appuser appgroup

postrotate

systemctl reload app.service

endscript

}

**4. Archiving Service**

* **Develop Archiving Script**:
  + Write a script (Python, Shell, or Bash) that checks for rotated logs, compresses them, and transfers them to cloud storage.
  + Use the cloud provider’s CLI (e.g., aws s3 cp, gsutil cp) to move compressed logs

import os

import subprocess

from datetime import datetime

def archive\_log(file\_path, cloud\_path):

compressed\_file = f"{file\_path}.gz"

subprocess.run(["gzip", file\_path])

subprocess.run(["aws", "s3", "cp", compressed\_file, cloud\_path])

os.remove(compressed\_file)

**Configure Encryption:**

* Enable encryption for sensitive logs before transferring (e.g., using OpenSSL or native cloud storage encryption).

**5. Retention and Cleanup**

* **Create Cleanup Script**:
  + Write a script that periodically checks archived logs, compares them with the retention period in the **Retention Policies** table, and deletes files that exceed this period.

import os

from datetime import datetime, timedelta

def cleanup\_archives(cloud\_path, retention\_days):

# Example to check and delete files older than retention period

for file in list\_files(cloud\_path):

if file\_age(file) > retention\_days:

delete\_cloud\_file(file)

**Deployment and Testing**

1. **Automated Testing**:
   * Test the log rotation and archiving workflow in a staging environment
   * Simulate large log files to test rotation, compression, and archiving efficiency.
   * Validate retention policies by altering dates and running cleanup scripts.
2. **Deployment**:
   * Deploy the scripts and schedule them using Cron or another scheduler.
   * Deploy the web interface on an internal server or cloud VM with restricted access.
3. **Load Testing and Optimization**:
   * Load test the system by generating extensive logs and verify that archiving and rotation processes don’t impact system performance.
   * Adjust rotation and cleanup frequencies based on performance results.
4. **Documenting and Training**:
   * Document installation, usage, and maintenance for administrators.
   * Train users on how to configure rotation policies, manage archives, and resolve common issues.

**Technologies and Tools Summary**

* **Log Rotation**: Logrotate, custom shell/Python scripts
* **Archiving**: Python, Gzip, AWS CLI / Google Cloud CLI for transfers
* **Database**: PostgreSQL / MySQL (Relational), MongoDB (NoSQL alternative)
* **Monitoring & Alerting**: Prometheus, Nagios, Slack integration for notifications
* **User Interface**: CLI (Python Click), Flask/Django for Web Dashboard (optional)
* **Scheduling**: Cron (Linux Scheduler)
* **Encryption**: OpenSSL (for files), SSL/TLS (for data in transit)

### Example Queries

1. **Retrieve Recent Builds**:
   * SELECT \* FROM Builds ORDER BY start\_time DESC LIMIT 10;
2. **Find Failed Test Cases for a Specific Build**:

Select tc.test\_name, tr.status, tr.error\_message

from testcases as tc join testresults as tr on tc

.test\_id = tr.test\_id

where tr.build\_id = ? And tr.status = 'failed'

3. **Calculate Average Coverage for Recent Builds**:

SELECT AVG(percentage\_covered) AS average\_coverage

FROM CoverageReports

WHERE build\_id IN (SELECT build\_id FROM )

**1. Database Setup (SQLite Example)**

The following Python code demonstrates how to set up a SQLite database for storing build data, test results, coverage metrics, and logs. This setup can be extended to other SQL databases, such as PostgreSQL or MySQL.

Python code

import sqlite3

def create\_tables():

connection = sqlite3.connect('ci\_cd.db')

cursor = connection.cursor()

cursor.execute('''CREATE TABLE IF NOT EXISTS Builds (

build\_id INTEGER PRIMARY KEY,

branch\_name TEXT,

commit\_id TEXT,

status TEXT,

triggered\_by TEXT,

start\_time TEXT,

end\_time TEXT,

duration INTEGER

)''')

cursor.execute('''CREATE TABLE IF NOT EXISTS TestSuites (

suite\_id INTEGER PRIMARY KEY,

build\_id INTEGER,

suite\_name TEXT,

status TEXT,

FOREIGN KEY(build\_id) REFERENCES Builds(build\_id)

)''')

cursor.execute('''CREATE TABLE IF NOT EXISTS TestCases (

test\_id INTEGER PRIMARY KEY,

suite\_id INTEGER,

test\_name TEXT,

description TEXT,

expected\_outcome TEXT,

FOREIGN KEY(suite\_id) REFERENCES TestSuites(suite\_id)

)''')

cursor.execute('''CREATE TABLE IF NOT EXISTS TestResults (

result\_id INTEGER PRIMARY KEY,

test\_id INTEGER,

build\_id INTEGER,

status TEXT,

execution\_time INTEGER,

error\_message TEXT,

FOREIGN KEY(test\_id) REFERENCES TestCases(test\_id),

FOREIGN KEY(build\_id) REFERENCES Builds(build\_id)

)''')

cursor.execute('''CREATE TABLE IF NOT EXISTS CoverageReports (

coverage\_id INTEGER PRIMARY KEY,

build\_id INTEGER,

lines\_covered INTEGER,

lines\_missed INTEGER,

percentage\_covered REAL,

statements\_covered INTEGER,

branches\_covered INTEGER,

FOREIGN KEY(build\_id) REFERENCES Builds(build\_id)

)''')

cursor.execute('''CREATE TABLE IF NOT EXISTS Logs (

log\_id INTEGER PRIMARY KEY,

build\_id INTEGER,

log\_type TEXT,

log\_level TEXT,

log\_message TEXT,

timestamp TEXT,

FOREIGN KEY(build\_id) REFERENCES Builds(build\_id)

)''')

connection.commit()

connection.close()

create\_tables()

**2. Interfacing with Jenkins API**

This script connects to Jenkins to trigger builds, fetch build statuses, and collect logs. You’ll need the Jenkins server URL and an API token with appropriate permissions.

Python code

import requests

from requests.auth import HTTPBasicAuth

JENKINS\_URL = 'http://localhost:8080'

JENKINS\_USER = 'username'

JENKINS\_TOKEN = 'your\_api\_token'

def trigger\_jenkins\_build(job\_name):

url = f"{JENKINS\_URL}/job/{job\_name}/build"

response = requests.post(url, auth=HTTPBasicAuth(JENKINS\_USER, JENKINS\_TOKEN))

if response.status\_code == 201:

print("Build triggered successfully")

else:

print("Failed to trigger build")

def get\_build\_status(job\_name, build\_number):

url = f"{JENKINS\_URL}/job/{job\_name}/{build\_number}/api/json"

response = requests.get(url, auth=HTTPBasicAuth(JENKINS\_USER, JENKINS\_TOKEN))

if response.status\_code == 200:

data = response.json()

print(f"Build Status: {data['result']}")

return data['result']

else:

print("Failed to fetch build status")

return None

def get\_build\_logs(job\_name, build\_number):

url = f"{JENKINS\_URL}/job/{job\_name}/{build\_number}/consoleText"

response = requests.get(url, auth=HTTPBasicAuth(JENKINS\_USER, JENKINS\_TOKEN))

if response.status\_code == 200:

print("Build Logs:")

print(response.text)

return response.text

else:

print("Failed to fetch build logs")

return None

trigger\_jenkins\_build('example-job')

status = get\_build\_status('example-job', 1)

logs = get\_build\_logs('example-job', 1)

**3. Running and Testing the System**

1. **Jenkins Job**: Set up a Jenkins job that calls the Python scripts to:
   * Trigger builds
   * Fetch build statuses
   * Collect and store logs and test results
2. **Automated Test Runs**: The Python script for Jest results can be run post-build to parse and store data. Use Jest’s --json and --coverage flags to output results and coverage data in JSON for Python processing.
3. **Database and Logging**: Continuously log builds, test results, and coverage metrics in the database, which can later be used for reporting and analytics.

This project provides the foundation to manage backend CI/CD automation with Python, Jest, and Jenkins. The Python scripts interface with Jenkins for build management, handle Jest results, and store data for tracking quality metrics over time.

**CONCLUSION**

**CONCLUSION**

The ***Continuous Integration and Testing of a Backend Application using Jest and Jenkins*** project establishes a robust framework for automating testing, quality assurance, and integration in backend development workflows. By integrating Jest for comprehensive unit and integration testing with Jenkins for CI/CD automation, this system ensures code quality, accelerates feedback loops, and reduces the time and resources needed for manual testing.

Key benefits include:

-**Enhanced Code Quality**: Automated testing with Jest ensures that code is thoroughly tested and coverage metrics are maintained.

-**Efficient Workflow Automation**: Jenkins automates builds and tests on every commit, integrating seamlessly with version control and notifying developers of issues immediately.

- **Consistent Testing Environments**: Using standardized scripts and, optionally, containerized environments, the CI pipeline maintains consistency across development and testing environments, reducing bugs and deployment issues.

This project is not only a powerful tool for ensuring backend code quality but also a strategic asset in modern software development, enabling teams to build, test, and deliver high-quality software rapidly. As a continuous integration system, it promotes reliable releases and supports scalable development practices, ultimately fostering a culture of continuous improvement and quality assurance.