Advanced Programming Techniques

Project Report

Expense Tracker Application

Name: Syed Muhammad Fasih Ali

Matriculation Number: 7126755

Repository: https://github.com/smfasihaly/TddProject

Submitted to: Professor Lorenzo Bettini

Table of Contents

Introduction

Managing personal finances can be a challenging task. Keeping track of expenses, understanding where the money goes, and making informed decisions are essential for financial well-being. I decided to develop an Expense Tracker Application—a tool designed to help individuals like me organize and monitor their spending habits more effectively.

The primary motivation behind developing this application is to demonstrate the effective use of advanced software development techniques learned during the course **Advanced Techniques and Tools for Software Development**. By implementing a real-world application, we can showcase proficiency in applying Test-Driven Development (TDD), comprehensive testing strategies, code coverage analysis, mutation testing, and continuous integration processes.

The Expense Tracker Application cohesively apply these methodologies. It allows for the exploration of designing an interface using Java Swing, managing data persistence with Hibernate, and ensuring code quality and reliability through rigorous testing and code analysis tools.

Through this project, the objectives are to:

- **Advanced Development Techniques:** Unit testing, integration testing using TDD, and end-to-end testing using BDD to ensure the application is robust and reliable.
- **High Code Quality:** Achieve 100% code coverage & eliminate surviving mutants using mutation testing.
- **Continuous Integration:** Used GitHub Actions for automated and efficient development workflow.
- **Tool Integration:** Integrate tools like Maven for build automation, Docker for containerization during testing, and SonarCloud for code quality analysis.

The development of the Expense Tracker Application provided a valuable opportunity to tackle real-world software development challenges. By following the provided guidelines and applying the advanced techniques outlined in our course textbook, **Test-Driven Development, Build Automation, Continuous Integration with Java, Eclipse, and Friends** by Prof. Lorenzo Bettini, this project exemplifies the practical application of modern software development methodologies in creating a functional, user-friendly, and high-quality solution.

Key Features of the Expense Tracker

Expense Management:

- Easily add, update, and delete expenses with all necessary details. You can record the amount, description, date, and assign each expense to a specific category.

Category Management:

- Create and manage categories to organize your expenses effectively. Whether it's Food, Transportation, Utilities, or any category.
- You cannot delete a category if it's assigned to an expense. This feature prevents accidental loss of important data and maintains the integrity of your expense records.

Detailed Expense Overview with Categories:

- View all your expenses along with their associated categories of all time. The application provides a comprehensive list of all recorded expenses.

Category-Specific Expense Viewing:

- Easily see all expenses associated with a single category by clicking the "Show Expenses" button on the category page.
- Total Expenditure Calculation - Instantly see the total amount spent across all expenses.

Application Flow

The application is divided into two primary views: **CategorySwingView** and **ExpenseSwingView**.

Application Entry Point:

The **ExpenseTrackerSwingApp** class serves as the entry point for running the application. When the application is launched, this class initializes the necessary components and sets up the user interface.

Key responsibilities of ExpenseTrackerSwingApp include:

- **Setting Up the Look and Feel**: The class configures the Java Swing look and feel to provide a consistent user interface experience.
- **Database Configuration:** The `ExpenseTrackerSwingApp` class configures the database based on the environment. In production, it uses the `hibernate.cfg.xml` file with a manually managed Docker container for the MySQL database. In testing environments (such as when running end-to-end BDD tests), the application switches to `hibernate-IT.cfg.xml` and uses Testcontainers when running in Eclipse, or Docker containers when run via Maven. The command-line options `--mysql-DB_URL`, `--mysql-user`, and `--mysql-pass` allow customization of the MySQL database connection when starting the application.
- **Command-Line Arguments with Picocli:** The application uses the Picocli library to parse command-line arguments for MySQL database configurations such as `--mysql-DB_URL`, `--mysql-user`, and `--mysql-pass`.
- **Instantiating Views and Controllers**: It creates instances of both ExpenseSwingView and CategorySwingView, along with their respective controllers (ExpenseController and CategoryController). These controllers manage interactions between the views and repositories (ExpenseMysqlRepository and CategoryMySqlRepository).
- **Running the Application**: The application starts by making the Expense view visible. The controllers load the necessary data (expenses and categories), ensuring that the UI is populated and ready for user interaction.

Expense Management (Initial View)

Users can track and manage their expenses efficiently. Each expense is linked to a category defined in the Category Management view. The primary components of this view are:

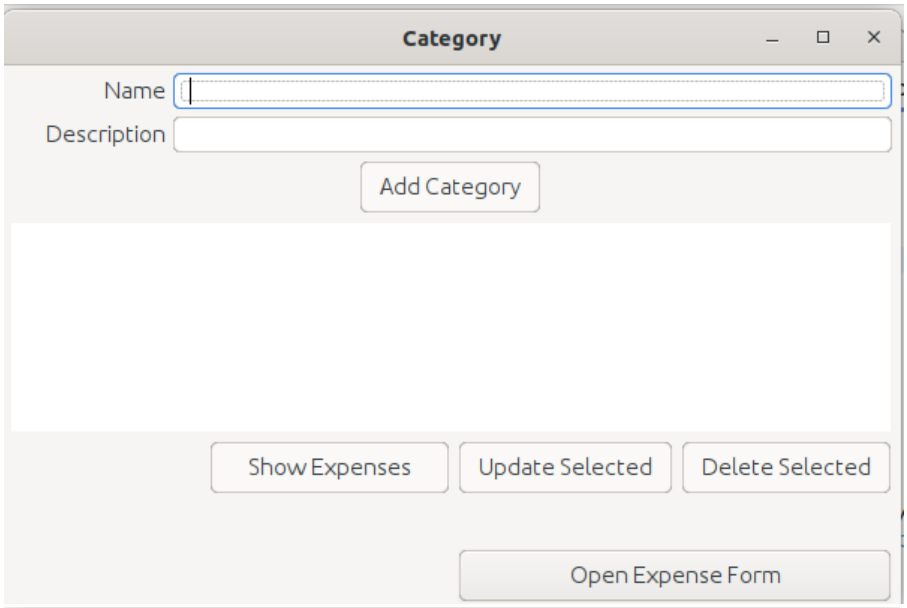
- **Description, Amount, Date, and Category Fields**: These input fields allow users to define the details of their expenses, including a description, the amount spent, the date of the expense, and the category it belongs to.
- **Add Expense Button**: Adds a new expense to the system based on the user input.
- **Update/Delete Selected Buttons**: Users can update or delete selected expenses.
- **Total Expenditure Display**: The total amount spent is calculated and displayed at the bottom, giving users a clear view of their overall spending.
- **Open Category Form Button**: Allows users to navigate to the Category Management form to add, update, or delete categories for organising expenses.

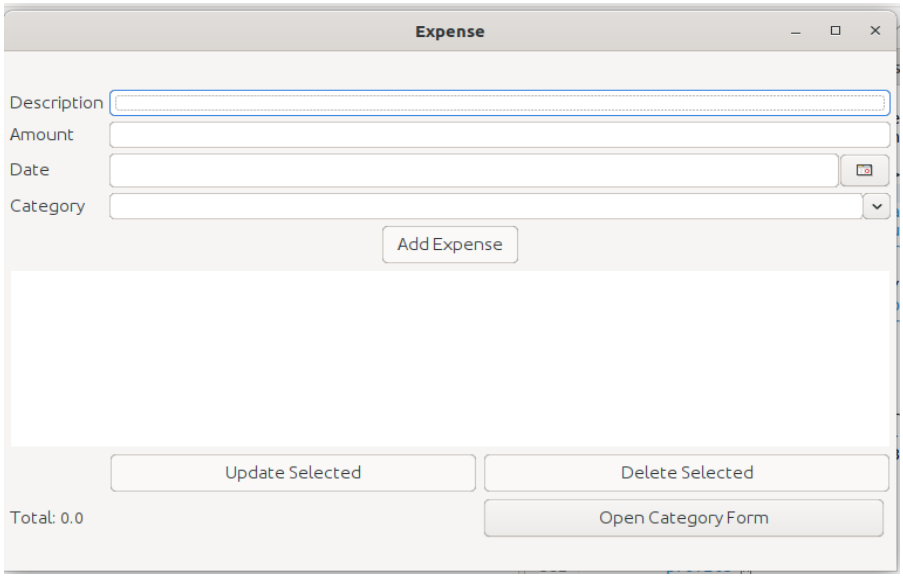
Category Management

The Category Management form allows users to define and manage categories.
The key components of the Category Management view are:

- **Category Name and Description Fields**: These fields allow users to define the name and description for each category.
- **Add Category Button**: Adds a new category based on the user input.
- **Show Expenses Button**: Displays all the expenses associated with the selected category with their total amount.
- **Update/Delete Selected Buttons**: Users can update or delete the selected category. The delete operation is restricted if the category is assigned to any expenses, ensuring data integrity.
- **Open Expense Form Button**: Navigates back to the expense management form.

Screenshot of application





Application Architecture

The Expense Tracker Application is designed using the **Model-View-Controller (MVC)** architectural pattern, which promotes a clear separation of concerns and enhances modularity, scalability, and maintainability.

Technologies Used

- **Programming Language**: Java 8
- **GUI Framework**: Java Swing
- **Database**: MySQL (for production), Embedded H2 Database (for testing),
- **Object-Relational Mapping (ORM)**: Hibernate
- **Build Tool**: Maven
- **Version Control**: Git and GitHub
- **Continuous Integration**: GitHub Actions
- **Testing Frameworks**: JUnit, Mockito, Cucumber
- **Code Coverage Tools**: JaCoCo, Coveralls
- **Mutation Testing**: PIT (Pitest)
- **Static Code Analysis**: SonarCloud

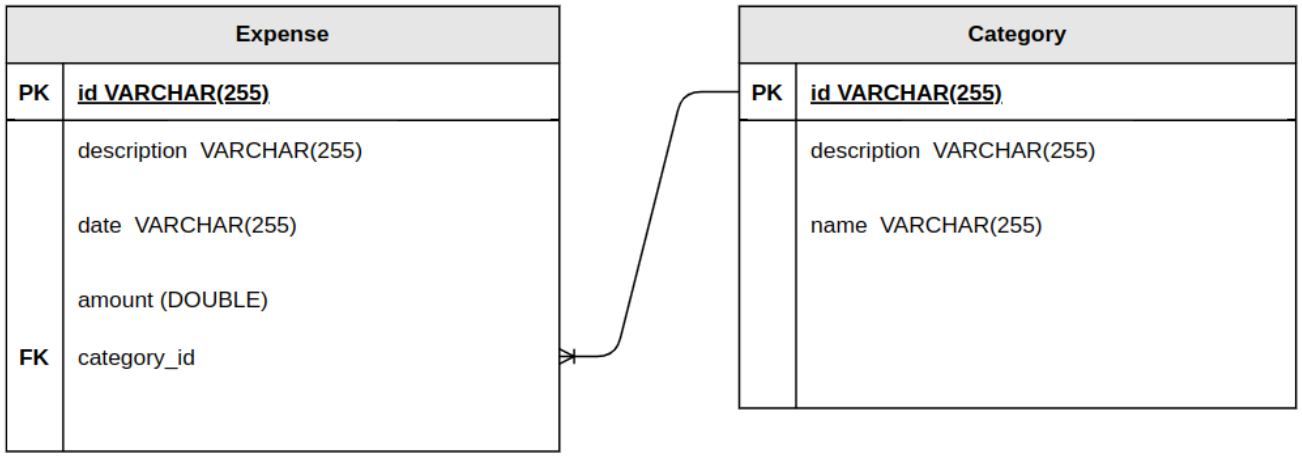
Database Configuration

The Expense Tracker application uses a flexible approach to database configuration, depending on the environment in which it is running. This ensures efficient performance during development and testing while maintaining consistency in production.

- **Production Environment**: MySQL is the primary database used in production. The database configuration is managed through the **hibernate.cfg.xml** file, and the MySQL instance is typically run within a Docker container to ensure portability and consistency across different environments.
- **Unit Testing Environment**: For unit tests, the application utilizes an in-memory H2 database. This configuration is specified in the **hibernate-test.cfg.xml** file, enabling quick and lightweight testing without the need for a MySQL instance. The H2 database provides an efficient way to simulate database interactions for the repository layer during unit testing.
- **Integration Testing and BDD Testing**: For integration and BDD tests classes namely *ExpenseControllerIT*, *CategoryControllerIT*, and BDD tests in *DatabaseSteps*, both **Testcontainers** and **Maven Docker** configurations are used depending on the environment.
- **Testcontainers (Eclipse environment)**: When running tests through Eclipse, Testcontainers are employed to automatically start a MySQL Docker container. This dynamically configures the connection details for the test environment.
- **Maven Docker Container (Maven environment):** When running tests via Maven, Docker containers are managed by Maven itself. The MySQL container is automatically started, and the database connection is configured accordingly.

Both environments Testcontainers and Maven Docker use the same hibernate-IT.cfg.xml file, with the database URL, username, and password dynamically adjusted.

Entity Relationship Diagram ERD



Application Structure

Models

Category Model: Represents expense categories with attributes such as id, name, and description. Uses Hibernate annotations to map to the database.

Expense Model: Represents individual expenses with attributes like id, amount, description, date, and a reference to a Category. Also uses Hibernate for ORM.

Views

CategorySwingView: Manages the user interface for category management, including adding, updating, and listing categories.

ExpenseSwingView: Manages the user interface for expense management, including adding, updating, and listing expenses.

Controllers

CategoryController: Handles user actions related to categories, communicates with the CategoryRepository, and updates the CategorySwingView.

ExpenseController: Manages user actions related to expenses, interacts with the ExpenseRepository, and updates the ExpenseSwingView.

Repositories

CategoryRepository: Provides CRUD operations for categories, handling data persistence through Hibernate.

ExpenseRepository: Offers CRUD operations for expenses, also utilizing Hibernate for data persistence.

Utils

ValidateUtils: data validation by checking required inputs such as strings, amounts, and dates. It contains three key methods:

String Validation: Ensures that strings (like descriptions or category names) are not null or empty.

Amount Validation: Verifies that expense amounts are positive values.

Date Validation: Confirms that dates are not null or in the future.

The custom ValidationException is also thrown when validation fails, ensuring data integrity. This centralized validation approach enhances the reliability and maintainability of the application.

Testing Components

A comprehensive testing strategy was implemented to ensure the reliability, robustness, and correctness of the Expense Tracker Application. The testing approach encompasses several tests, each targeting different aspects of the application to guarantee high quality and performance.

Unit Tests

Unit tests were written following **Test-Driven Development (TDD)** techniques to ensure each individual component was designed and implemented with correctness in isolation. For repository classes, an in-memory H2 database was used to simulate real database operations, ensuring fast and reliable tests without external dependencies. This approach allowed for continuous feedback during development, promoting cleaner and more robust code.

For testing the UI components, I utilized **AssertJSwingJUnitTestCase**, a framework specifically designed for Swing-based graphical user interface testing. This ensured the correctness of the user interface interactions and validated UI behavior, providing confidence that the UI responded correctly to various user inputs.

Integration Tests

Integration tests verified the proper interaction between different components using a real MySQL database instance through **Testcontainers**, simulating production-like conditions. These tests ensured that the components work together as expected in a near-production environment.

For **controller's** IT tests, different approaches were used based on the execution environment:

Maven: When running the tests from Maven, **Docker** was used to spin up a MySQL container, allowing the tests to interact with a real MySQL instance.

Eclipse: When running the tests from Eclipse, **Testcontainers** were utilized to manage the MySQL instance for testing, providing a flexible, containerized solution.

For repository and UI tests, Testcontainers was consistently used to manage the MySQL instance, ensuring seamless testing of database interactions for both repositories and the UI components. This approach provided a stable environment for testing without external dependencies, ensuring reliable tests that simulate real database operations.

This unified approach of using Testcontainers across different test types ensures that the integration tests can be executed seamlessly in both local development environments and continuous integration pipelines, offering flexibility and reliability.

End-to-End Tests

End-to-end tests were conducted using **Behavior-Driven Development (BDD)** principles with **Cucumber**, ensuring that the full application flow from the user's perspective was verified. These tests simulate real-world

user interactions with the application's user interface, validating that the application behaves as expected in various scenarios.

BDD with Cucumber: End-to-end tests were written in Gherkin syntax, allowing for clear and concise test scenarios. These scenarios defined the expected behavior of the application, focusing on high-level interactions such as adding, updating, and deleting expenses or categories.

Like the integration tests, two different approaches were used for running the end-to-end tests:

Maven: When executing tests through Maven, Docker was used to spin up a MySQL container, allowing the application to interact with a real MySQL instance.

Eclipse: When running the tests from Eclipse, Testcontainers was used to manage the MySQL instance, providing flexibility and ensuring a production-like environment for testing.

By following this approach, the end-to-end tests ensured that the entire application worked correctly, from the front-end interactions to the back-end database operations, in both development and continuous integration environments.

Resources

The Expense Tracker Application uses various resources to configure the application, manage database connections, and control logging. These resources are divided into main resources and test-specific resources, ensuring flexibility for different environments.

Main Resources (located in src/main/resources/)

hibernate.cfg.xml: Configures Hibernate for the production MySQL database, including connection settings and entity mappings for Category and Expense.

log4j2.xml: Manages logging behavior, defining log levels and formatting for debugging and monitoring the application.

Test Resources (located in src/test/resources/)

hibernate-test.cfg.xml: Configures an in-memory H2 database for unit tests, enabling fast, isolated testing without external database dependencies.

hibernate-IT.cfg.xml: Used for integration tests with MySQL in a Docker container, ensuring that real database interactions are tested.

Maven

POM (Project Object Model)

The POM file for the Expense Tracker Application defines the project's dependencies, plugins, build configurations, and profiles. It ensures that the project follows a well-structured build process and facilitates continuous integration and testing. Here's a breakdown of the key aspects of the POM file:

Dependencies

- **Java Version**: The project uses **Java 8** for compilation and runtime.
- **Hibernate**: ORM for database interactions (version: 5.4.15.Final).
- **H2 Database**: Used for in-memory testing (version: 1.4.200).
- **MySQL Connector**: For interaction with MySQL in production (version: 8.0.33).
- **JUnit, Mockito, AssertJ**: Libraries used for writing unit tests.
- **Cucumber**: Supports BDD and end-to-end tests (version: 7.0.0).
- **Testcontainers**: Utilized for using MySQL in Docker containers during integration testing.
- **JaCoCo**: For code coverage analysis.
- **PIT (Pitest)**: For mutation testing.
- **SonarCloud**: For static code analysis and quality checks.
- **Log4j**: Used for logging and debugging.
- **picocli**: Used for creating command-line interfaces. (Version 4.6.1)

Plugins

- **Maven Surefire Plugin**: Executes unit tests during the build lifecycle.
- **Maven Failsafe Plugin**: Runs integration tests separately from unit tests.
- **Docker Maven Plugin**: Manages Docker containers during integration testing, specifically for running MySQL in Docker.
- **JaCoCo Plugin**: Provides code coverage reports during testing.
- **Pitest Plugin**: Executes mutation testing for improving test effectiveness.
- **SonarCloud Plugin**: Integrates static code analysis and reports directly to SonarCloud.
- **Coveralls Plugin**: Sends coverage reports to Coveralls for further analysis and metrics.

Build Profiles

- **Unit Test Profile**: Configures unit testing with JaCoCo and mutation testing. It uses the H2 database for fast and isolated tests.
- **Integration Test Profile**: Configures integration tests, spinning up a MySQL instance using Testcontainers to verify database interactions.
- **Jacoco Profile**: Ensures code coverage is collected and verified against a coverage threshold.
- **Coveralls Profile**: Sends coverage reports to Coveralls for CI/CD pipelines.

This POM provides a well-integrated environment for **test-driven development**, **continuous integration**, and **code quality** enforcement, ensuring a reliable and maintainable codebase throughout the development lifecycle.

Maven Profiles

The Expense Tracker Application uses different Maven profiles to manage various testing and build environments, enabling flexible configurations based on the environment.

Unit-Test Profile (unit-test)

This profile is used for executing unit tests. It includes the following **key configurations:**

- **Maven Surefire Plugin**: Runs the unit tests
- **JaCoCo Plugin**: Generates code coverage reports for the unit tests, excluding certain directories such as app, model, and bdd.
- **Pitest Maven Plugin**: Runs mutation testing to ensure the code is robust, with a 50% mutation coverage threshold.
- **Surefire Report Plugin**: Generates unit test reports.
- **Maven Site Plugin**: Generates a site for the project but disables report generation.

How to run it:

```
mvn clean verify -Punit-test
```

Integration-Test Profile (integration-test)

This profile is responsible for running integration tests that check the integration of different components in the application.

Key configurations:

- Maven Surefire Plugin: Skips unit tests.
- Failsafe Plugin: Executes the integration tests.
- Docker Maven Plugin: Automatically manages the MySQL Docker container, starting before integration tests and stopping after.
- Site Plugin: Generates reports for integration tests.

How to run it:

```
mvn clean verify -Pintegration-test
```

JaCoCo Profile (jacoco)

This profile generates code coverage reports for both unit and integration tests using **JaCoCo**.

Key configurations:

- JaCoCo Plugin: Prepares the environment for code coverage and checks the coverage ratio with a minimum threshold of 50%.

How to run it:

```
mvn clean verify –Pjacoco
```

Coveralls Profile (coveralls)

This profile uploads the JaCoCo coverage report to **Coveralls**, allowing external monitoring of the code coverage.

Key configurations:

- **Coveralls Maven Plugin** Uploads the coverage report to Coveralls for quality monitoring.

How to run it:

```
mvn clean verify -Pcoveralls
```

Workflows

We have three workflows defined below.

Java CI with Maven - Multi-Version Unit, Integration Testing, and E2E with Coverage

File name: Maven.yml

Purpose:

This workflow handles **unit tests**, **integration tests**, and **end-to-end tests** across different versions of Java (1.8 and 11). It ensures code quality by running these tests in parallel.

Key Steps:

- **Matrix Strategy**: Tests the application on Java 8 and 11 using a matrix strategy, ensuring compatibility with both versions.
- **Run Unit Tests**: Executes unit tests and code coverage using JaCoCo and PIT mutation testing.
- **Run Integration Tests**: Run integration tests using a real MySQL (Docker in Maven).
- **Run End-to-End (E2E) Tests**: Executes BDD-style end-to-end tests using Cucumber to verify the application's behavior from a user's perspective.
- **Cache Maven Packages**: Speeds up builds by caching dependencies.
- **Archive Test Reports**: Stores all test results and coverage reports as artifacts for analysis.

Triggers:

Runs on **push** and **pull_request** events.

Java CI with Maven, Docker, and SonarCloud in Linux

File name: maven_Sonar.yml

Purpose:

This workflow is designed to run **unit tests**, **integration tests**, and upload the results to **SonarCloud** for code quality analysis.

Key Steps:

- **Runs on Ubuntu**: Executes the build on a Linux environment.
- **Java Setup**: Installs Java 17 for the build.
- **Cache Maven Packages and Sonar Cache**: Caches Maven dependencies and SonarCloud results to improve build speed.
- **Build with Maven and SonarCloud**: Runs the full Maven build with coverage analysis using JaCoCo and uploads the results to SonarCloud for quality monitoring.
- **Docker Setup**: Uses Docker containers (via Maven Docker plugin) to run MySQL for integration tests.
- **Archive Test Reports**: Stores the test reports and coverage analysis as artifacts.

Triggers:

Runs on **push** and **pull_request** events to the master branch.

Java CI with Maven - Unit Testing with Coverage on Windows and macOS

Purpose:

This workflow is designed to run **unit tests** on both macOS and Windows operating systems with Java versions 8 and 11 while generating code coverage reports using JaCoCo.

Key Steps:

- **Runs on macOS and Windows**: Tests the application on multiple operating systems to ensure cross-platform compatibility.
- **Java Setup**: Installs Java 8 and 11 using the actions/setup-java action.
- **Cache Maven Packages**: Caches Maven dependencies to speed up future builds.
- **Run Unit Tests**: Executes the unit tests using the Maven verify phase with the unit-test profile.
- **Archive Test Reports**: Saves the generated test and code coverage reports as artifacts to make them available after the workflow finishes.

Triggers:

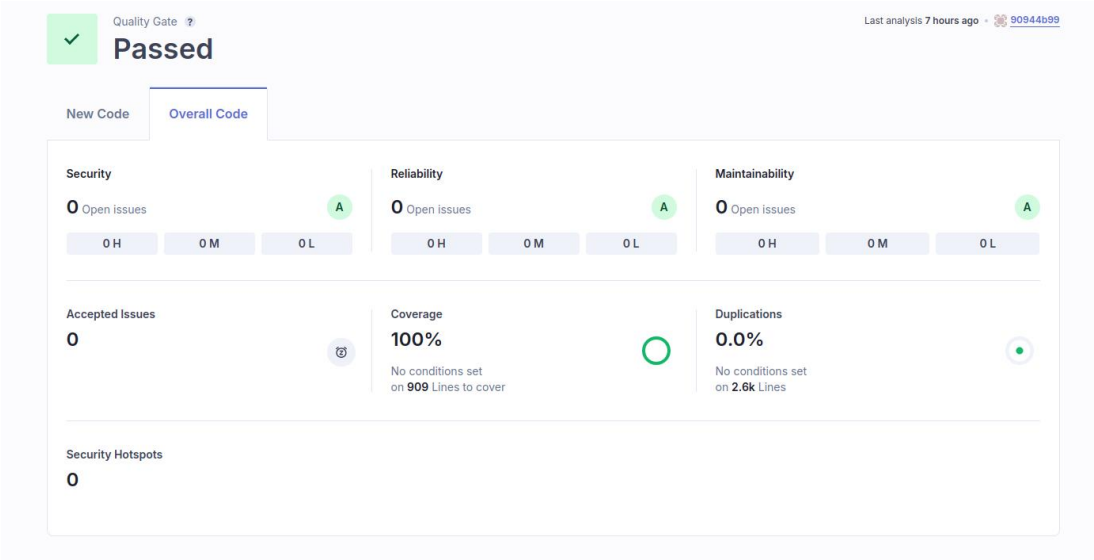
Runs on **push** and **pull_request** events.

Code Quality and Coverage with SonarCloud and Coveralls

The project integrates SonarCloud and Coveralls to ensure high code quality and comprehensive test coverage. These services are accessible directly from the GitHub repository via the badges displayed in the README file, which provide real-time insights into the codebase.

SonarCloud

SonarCloud performs continuous code quality analysis, detecting issues like bugs, code smells, and vulnerabilities. It also enforces 100% code coverage and checks for duplicated lines, maintaining zero technical debt. By clicking the SonarCloud badge, detailed reports on code quality can be accessed.



Coveralls

Coveralls is used to track code coverage, ensuring that all the code is well tested. It is integrated with the testing workflows to provide insights on what percentage of the code is covered by tests, and the badge reflects the current status of the coverage. Clicking the Coveralls badge redirects to a detailed coverage report on the Coveralls website.

Conclusion

The development of the Expense Tracker Application successfully demonstrated the integration of advanced software development techniques and rigorous testing methodologies. This project emphasized the practical application of Test-Driven Development, continuous integration, and automated code quality assessments, enhancing both the robustness and maintainability of the application. The challenges faced and their resolutions have provided valuable insights into effective software development practices, paving the way for future enhancements and ensuring the application's alignment with high standards of quality.