# 1. Coding Standards

Here's a summary of coding standards focusing on Java naming conventions, code formatting, and comments:

## Java Naming Conventions

1. **Classes and Interfaces**
   - **Classes**: Use PascalCase (e.g., `RemitService`, `AccountManager`).
   - **Interfaces**: Use PascalCase (e.g., `Readable`, `Persistable`).
2. **Methods**
   - Use camelCase (e.g., `calculateTotal`, `findUserById`).
3. **Variables**
   - Use camelCase (e.g., `userAge`, `totalAmount`).
4. **Constants**
   - Use UPPER_SNAKE_CASE (e.g., `MAX_VALUE`, `DEFAULT_TIMEOUT`).
5. **Packages**
   - Use lowercase (e.g., `com.swifttech.remittance`, `org.mycompany.utils`).
6. **Parameters**
   - Use camelCase (e.g., `userName`, `filePath`).

## Code Formatting

1. **Indentation**
   - Use 4 spaces per indentation level (no tabs).
2. **Line Length**
   - Keep lines of code to a maximum of 80-120 characters.
3. **Braces**
   - **Class and Method Declarations**: Braces go on the same line.

     ```
     public class UserController {
         public void createUser() {
             // code
         }
     }
     ```

   - **Control Statements**: Braces should always be used, even for single statements.

     ```
     if (condition) {
         // code
     }
     ```

4. **Blank Lines**
   - Use blank lines to separate methods and logical sections of code.
   - No extra blank lines between class members.
5. **Spaces**
   - Use a single space after keywords (`if`, `for`, `while`) and before opening braces.

- No space between method names and parentheses.
- One space after commas, semicolons, and colons.

6. **Comments and Documentation**
   - Use single-line comments (`//`) for brief explanations.
   - Use block comments (`/* ... */`) for longer explanations or sections.
   - Use Javadoc comments (`/** ... */`) for documenting classes, methods, and fields.

## Comments

1. **Single-Line Comments**
   - Use `//` for short explanations or notes.

```
int maxUsers = 100; // maximum number of users allowed
```

2. **Block Comments**
   - Use `/* ... */` for longer comments or to disable sections of code.

```
/*
 * This method calculates the total amount
 * based on the input values provided.
 */
public double calculateTotal(double[] amounts) {
    // code
}
```

3. **Javadoc Comments**
   - Use `/** ... */` for documenting public classes, methods, and fields. Provide a clear description of the method's functionality, parameters, and return values.

```
/**
 * Calculates the total amount from an array of amounts.
 *
 * @param amounts an array of double values to sum up
 * @return the total amount
 */
public double calculateTotal(double[] amounts) {
    // code
}
```

4. **TODO and FIXME Comments**
   - Use `// TODO:` and `// FIXME:` to mark places where code needs improvement or correction.

```
// TODO: Handle edge cases for input validation
```

By adhering to these standards, your code will be more readable, maintainable, and consistent.

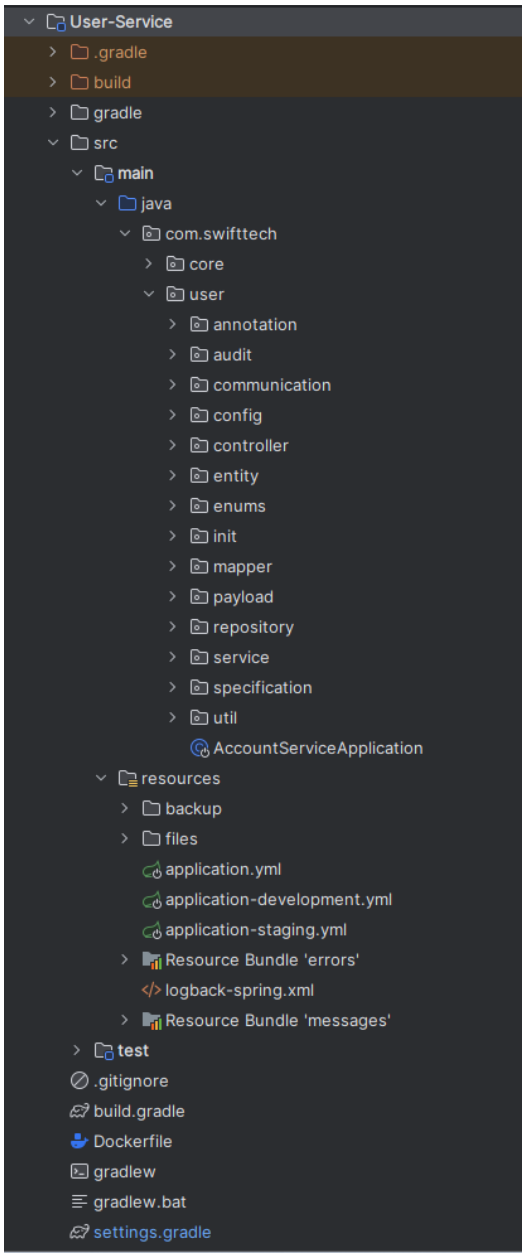
## 2. Project Structure

A well-structured project layout and proper package organization are crucial for maintaining and scaling a Java project. Here's a breakdown of recommended practices for directory layout and packages:

### Directory Layout

#### Root Directory

- **`src/`**: Contains source code.
  - **`main/java/`**: Contains main application code.
  - **`main/resources/`**: Contains resources like configuration files and non-code assets.
  - **`test/java/`**: Contains unit tests.
- **`build/`** or **`target/`**: Contains build artifacts (usually generated by build tools Gradle).
- **`lib/`**: Contains external libraries




## Base Package Name

### Base Package

- **`com.swifttech`**: The base package name should reflect the domain name in reverse. For Domain name swifttech.com, the package name would be com.swifttech

## Top-Level Packages

- **`com.swifttech.entity`**: Data entity classes.
- **`com.swifttech.service`**: Service layer classes.
- **`com.swifttech.controller`**: Controllers or endpoints (e.g., for a web application).
- **`com.swifttech.repository`**: Data access objects (DAOs) or repository classes.
- **`com.swifttech.util`**: Utility classes and helpers.
- **`com.swifttech.config`**: Configuration classes.
- **`com.swifttech.exception`**: Custom exceptions.
- **`com.swifttech.payload`**: Data Transfer Objects for request and response object.
- **`com.swifttech.payload.request`**: Data Transfer Objects for request.
- **`com.swifttech.payload.response`**: Data Transfer Objects for response.

## 3. Version Control

Using version control effectively is crucial for managing code changes, collaborating with team members, and maintaining a history of changes. Here's an overview of best practices for branching strategy and commit messages:

### Branching Strategy

A well-defined branching strategy helps streamline development workflows and manage releases. Here's a commonly used branching strategy:

1. **Main Branches**
    - **`main`** (or **`master`**):
        - This branch contains the stable production-ready code. It should always be stable and deployable.
        - Only merge tested and approved changes into this branch.
    - **`dev`**:
        - This branch contains the latest development changes that are ready for the development release.
        - It is an integration branch for features and fixes before they are merged into the `staging` branch.
    - **`staging`**:
        - This branch contains the qa environment changes that are ready for the testing.
        - It is an integration branch for features and fixes before they are merged into the `uat` branch.
    - **`uat`**:
        - This branch contains the UAT environment code. It should always be stable and deployable.
        - Only merge tested and approved changes into this branch.
2. **Supporting Branches**
    - **Feature Branches**:
        - **Branch Name**: `feature/<feature-name>`

        - Created from `dev` and used to work on new features or enhancements.
        - Example: `feature/login-system`
    - **Bugfix Branches**:
        - **Branch Name**: `bugfix/<bugfix-name>`
        - Created from `dev` or `main` to address specific bugs or issues.
        - Example: `bugfix/fix-null-pointer`
3. **Branch Workflow**
    - **Feature Development**:
        - Create a feature branch from `dev`.
        - Work on the feature and commit changes.
        - Merge the feature branch back into `dev` when complete and tested.
    - **Bug Fixes**:
        - Create a bugfix branch from `dev` or `main`, depending on where the bug exists.
        - Fix the bug and test the changes.
        - Merge the bugfix branch back into `dev`.
    - **Release Preparation**:
        - Create a release branch from `dev`.
        - Perform final testing and minor bug fixes on the release branch.
        - Merge the release branch into both `main` and `dev` once the release is ready.
    - **Hotfixes**:
        - Create a hotfix branch from `main`.
        - Apply critical fixes and test them.
        - Merge the hotfix branch into both `main` and `dev` to ensure the fix is included in future releases.

## Commit Messages

It improves readability and help track changes. We Followed these best practices for writing effective commit messages:

1. **Structure**
    - **Header** (required):
        - A brief summary of the change (50 characters or less).
    - **Body** (optional):
        - A detailed description of the change (wrap text at 72 characters).
    - **Footer** (optional):
        - Include any relevant issue tracking information or breaking changes.
2. **Format**

```
Detailed description of the change, including reasoning and context.
```

- **Example:**

```
Implement user login functionality with session management.

- Added login form with validation
```

```
- Integrated with authentication service
- Added error handling for incorrect credentials
```

- **Length:** Ideally between 72 and 100 characters per line.
- **Content:** Explain the "what" and "why" of the changes, but avoid the "how" (which should be evident from the code itself).

3. **Types**

   Use conventional commit types to categorize changes:

   - **feat**: New feature.
   - **fix**: Bug fix.
   - **docs**: Documentation changes.
   - **style**: Code style changes (formatting, etc.).
   - **refactor**: Code changes that neither fix a bug nor add a feature.

4. **Scope (Optional)**
    - Include the scope to indicate the area of the codebase affected (e.g., `auth`, `db`, `ui`).
5. **Consistency**
    - Be consistent with the format and type of commit messages throughout the project.
    - Follow the project's commit message conventions if they are defined.

### Summary

- **Branching Strategy**: Use a branching model with `main`, `dev`, and supporting branches (`feature`, `bugfix`, `release`, `hotfix`) to manage code changes and releases effectively.
- **Commit Messages**: Write clear, structured commit messages with a short summary, optional detailed description, and footer for related issues or breaking changes.

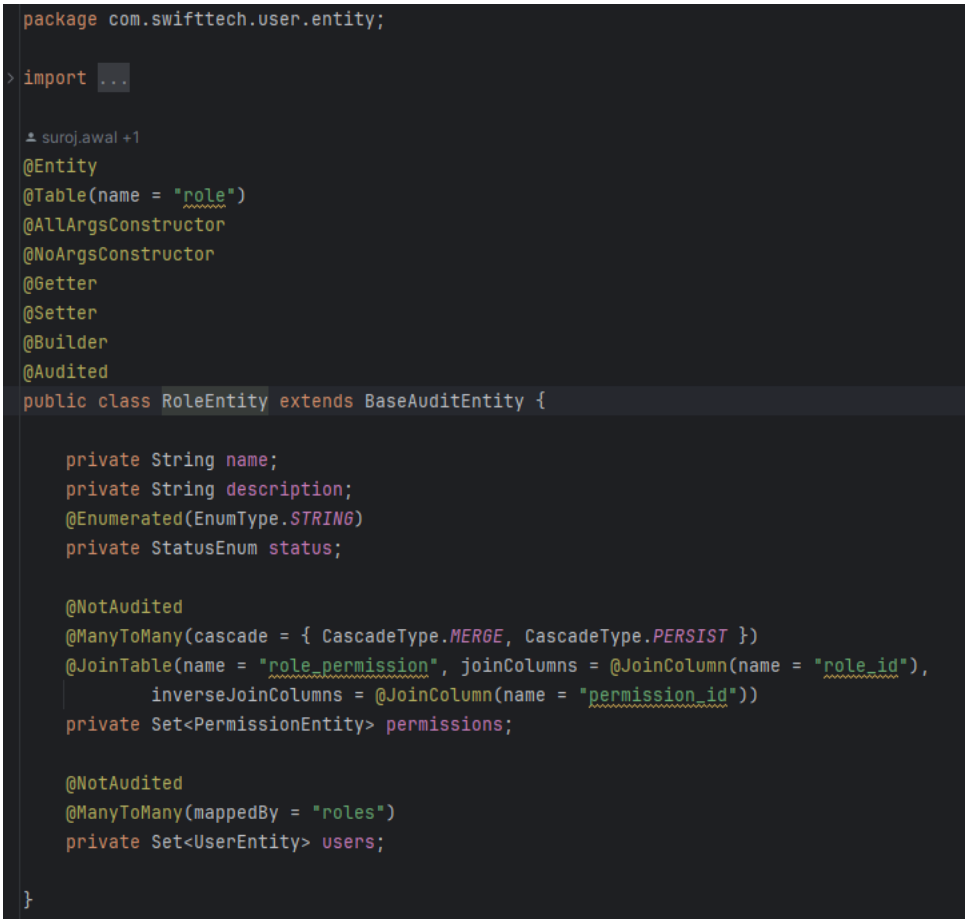
## 4. Database Standards

Database standards are essential for maintaining consistency, clarity, and efficiency in database management. Let's break down the key aspects related to naming conventions, schema design, and SQL queries:

### Naming Conventions

#### 1. Tables:

- Use singular nouns for table names (e.g., CustomerEntity, OrderEntity) or plural if that fits your organization's standards.
- Use clear, descriptive names that indicate the table's purpose.
- Use lowercase or snake_case (e.g., customer_orders), especially in systems that are case-sensitive.
- Avoid abbreviations unless they are widely understood.



```java
package com.swifttech.user.entity;

import ...

suroj.awal +1
@Entity
@Table(name = "role")
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@Builder
@Audited
public class RoleEntity extends BaseAuditEntity {

    private String name;
    private String description;
    @Enumerated(EnumType.STRING)
    private StatusEnum status;

    @NotAudited
    @ManyToMany(cascade = { CascadeType.MERGE, CascadeType.PERSIST })
    @JoinTable(name = "role_permission", joinColumns = @JoinColumn(name = "role_id"),
            inverseJoinColumns = @JoinColumn(name = "permission_id"))
    private Set<PermissionEntity> permissions;

    @NotAudited
    @ManyToMany(mappedBy = "roles")
    private Set<UserEntity> users;

}
```

**2. Columns:**

- Use descriptive names that clearly identify the data stored (e.g., customer_id, order_date).
- Avoid using ambiguous names like data or info.
- Use consistent naming conventions (e.g., camelCase, snake_case) across all columns.

**3. Constraints:**

- Prefix constraint names with their type (e.g., pk_ for primary keys, fk_ for foreign keys, chk_ for check constraints).
- For example, pk_customer_id for a primary key constraint on customer_id.

**4. Indexes:**

- Use a naming pattern that includes the table name and the column(s) being indexed (e.g., idx_customer_email).

**5. Stored Procedures and Functions:**

- Use a consistent prefix to identify their purpose (e.g., sp_ for stored procedures, fn_ for functions).
- Be descriptive (e.g., sp_getCustomerById, fn_calculateDiscount).

## Schema Design

**1. Normalization:**

- Aim for at least the third normal form (3NF) to reduce redundancy and improve data integrity.
- Normalize data to eliminate duplicate data and ensure relationships between tables are logical.

**2. Relationships:**

- Use primary keys to uniquely identify records.
- Establish foreign keys to define relationships between tables.
- Use indexing on foreign keys to improve join performance.

**3. Data Types:**

- Choose appropriate data types for each column (e.g., `INT` for IDs, `VARCHAR` for text fields, `DATE` for dates).
- Avoid using overly large data types; be mindful of storage and performance implications.

**4. Constraints:**

- Implement constraints to enforce data integrity (e.g., `NOT NULL`, `UNIQUE`, `CHECK`).
- Ensure constraints reflect business rules and validation requirements.

**5. Security:**

- Define user roles and permissions to control access to the database.
- Use views to restrict access to sensitive data.

## SQL Queries

**1. Select Queries:**

- Use `SELECT` statements to retrieve data; be explicit about the columns you need (e.g., `SELECT customer_id, customer_name FROM customers`).
- Use `JOIN` clauses to retrieve related data from multiple tables (e.g., `INNER JOIN`, `LEFT JOIN`).

**2. Insert, Update, Delete:**

- Use `INSERT INTO` to add new records.
- Use `UPDATE` to modify existing records, with proper `WHERE` clauses to target the correct rows.
- Use `DELETE` to remove records, also ensuring a `WHERE` clause to prevent unintentional data loss.

**3. Performance:**

- Use indexing to speed up query performance.
- Avoid `SELECT *` in production queries; specify needed columns to improve efficiency.
- Use query optimization techniques like `EXPLAIN` to understand query performance.

**4. Transactions:**

- Use transactions (BEGIN, COMMIT, ROLLBACK) to ensure data integrity during multi-step operations.
- Ensure transactions are used to maintain consistency and avoid partial updates.

**5. Error Handling:**

- Handle errors gracefully in SQL queries to maintain data integrity.
- Use TRY...CATCH blocks where supported to manage exceptions.

By adhering to these standards, you ensure that your database is well-structured, maintainable, and performs efficiently.

## 5. Development Tools and Environment

Choosing the right development tools and setting up a productive environment is crucial for efficient and effective software development. Here's a breakdown of each category:

### IDE (Integrated Development Environment)

**1. Popular IDEs:**

- **IntelliJ IDEA:** Excellent for Java development but also supports other languages. Known for its powerful features and integrations.

**2. Key Features to Look For:**

- **Code Editing:** Syntax highlighting, code completion, and refactoring capabilities.
- **Debugging Tools:** Integrated debuggers for stepping through code, breakpoints, and variable inspection.
- **Version Control Integration:** Support for Git, SVN, or other version control systems.
- **Plugin Ecosystem:** Availability of extensions or plugins for additional functionality.
- **Performance:** Efficient handling of large codebases and projects.

### Build Tools

**1. Popular Build Tools:**

- **Gradle:** Flexible build automation tool that works well with Java, Groovy, Kotlin, and more. It's known for its performance and incremental builds.

**2. Key Features to Look For:**

- **Dependency Management:** Ability to handle libraries and dependencies efficiently.
- **Task Automation:** Automate repetitive tasks like compiling, testing, and packaging.
- **Customizability:** Ability to define and configure custom build processes.
- **Integration:** Compatibility with IDEs and version control systems.

### Database Tools

**1. Popular Database Tools:**

- **pgAdmin:** A powerful management tool for PostgreSQL databases.
- **DBeaver:** An open-source database tool that supports multiple databases (e.g., MySQL, PostgreSQL, Oracle) with a unified interface.

**2. Key Features to Look For:**

- **Query Editor:** Robust SQL query editor with syntax highlighting and auto-completion.
- **Schema Design Tools:** Visual tools for designing and managing database schemas.
- **Database Management:** Capabilities for backup, restore, and performance monitoring.
- **Data Visualization:** Tools for visualizing query results and data relationships.

## Linting and Formatting

The coding standard proposed here adheres to the Google Java Style Guides with some adaptations. The main reason to select Google over Oracle is the update date.

## Adaptations to Google Java Style Guide

The Google Java Style Guide is a good reference for the most part, but is a little permissive on some topics. On the other hand, as a Java programmer you must be used to 4 spaces for code indentation, among other things. The following are some of the main rules gathered from Google Java Style Guide, the adaptations are marked with (*):
• **File encoding:** UTF-8.
• **Wildcard imports:** not permitted.
• **Static import for classes:** not permitted.
• **Imports ordering and spacing:** blank line between groups are not permitted(*).
• **Method Overloads:** never split, must appear sequentially.
• **Braces:** always used, even when the body is empty or contains a single statement.
• **Block and Switch indentation:** 4 spaces(*).
• **Indent continuation lines:** 8 spaces(*).
• **Column limit:** 120(*). New widescreen monitors allow higher column limit.
• **Blank line before the first member of the class:** required(*).
• **Blank line after the last member of the class:** not permitted(*).
• **Multiple consecutive blank lines:** not permitted(*).
• **Array initializers:** single space inside both braces is not permitted(*).
• **Horizontal alignment:** not permitted(*). As indicated in the guide, alignment can aid readability, but it creates problems for future maintenance.
• **Grouping parentheses:** recommended.
• **Variables:** declared when needed, one variable per declaration.
• **C-style array declarations:** not permitted.
• **Switch statements:** 2 spaces indentation, fall-through commented, default statements required.
• **Enum constant:** after each comma, a line break is required(*).
• **Annotation for class, method or constructor:** one annotation per line, no exceptions(*).
• **Annotation for a field:** one annotation per line(*).
• **Block (multiline) comment style:** must be in /* ... */ style, subsequent lines must start with * aligned with the * on the previous line. Example:

/* * This is okay // This is not permitted /* This is also not * for multiline // for multiline comments. * permitted for * comments. * multiline comments */ */

• **Modifiers:** appear in the order recommended by the Java Language Specification.
• **Naming:** special prefixes or suffixes are not permitted. 3 or more consecutive uppercase characters are not permitted except DTO(*).
• **Class names:** in UpperCamelCase.
• **Method names:** in lowerCamelCase.
• **Constant names:** use CONSTANT_CASE. Note that every constant is a static final field, but not all static final fields are constants, for example: logger must be used instead of LOGGER.
• **Caught exceptions:** not ignored.
• **Javadoc:** required for public or protected classes/class members, with few exceptions.
• **Javadoc paragraphs:** it may be difficult to understand. This example should help:

/** * Sample javadoc comment. One blank line appears between paragraphs. * * <p>Each paragraph but the first has <p> immediately before the first * word, with no space after. * * <p>One blank line appears before the group of block tags if present, * with no new line after @param tags. * * @param a The first parameter. For an optimum result, this should be * an odd number between 0 and 100. * @param b The second parameter. * @return The result of the foo operation, usually within 0 and 1000. */ int foo(int a, int b);

This is just a brief enumeration.

**Additional Rules**
The following rules are not mentioned in the Google Java Style Guide but they are needed in order to have a uniform coding style:
• **Blank line at start or end of a block:** not permitted.
• **Multiple empty statements:** not permitted. Example:

void foo() { ;; // This is not permitted. ...}

• **New line at end of file:** required (see Programming Best Practices below).
• **Assertions:** single space on both sides of the colon ( : ). Example:
assert condition : reportError();
• **Annotation for parameter:** no new line after annotation. Example:
public Response getMsg(@PathParam("param") String msg) {
• **Functional code and qualified invocations:** each element in a new line except the first one if not necessary. Example:

List<String> strings = Arrays.stream(names) .map((x) -> x.toUpperCase())**** .filter((x) -> x.contains("(M)")) .collect(Collectors.toList());

## Required actions

In order to comply with the Java style guidelines outlined above, the developer must take the following actions:

1. Import the proper Java Code Style Formatter into the integrated development environment (IDE).
2. Configure Save Actions in Eclipse or similar functionality in other IDEs.
3. Configure Checkstyle as the static code analysis tool to check for violatioto the coding standard ns before pushing the code to the repository.

## Most used Java coding standards

Here are some of the most used Java coding standards that can be found in the coding conventions from Oracle and Google, as well as other documents of this kind.
• Follow proper naming convention;
• Add comments;
• Identifier means a symbolic name that refers to the name of classes, packages, methods and variables in a Java program;
• The variable name should be related to its purpose;
• Name of the method should relate to the method's functionality;
• Method should not contain more than 50 lines;
• There should be no Duplicate code in the same class or other class;
• Declare global variables only if necessary to use in the other methods;
• Double check creation of static variables inside a class;
• Avoid accessing variables directly from other classes instead use getter and setter methods;
• All business logic should be handled in the service class only;
• All DB related code should be in the DAO classes only;
• Use getters and setters;
• Declare instance variable as private;
• Keep the scope of variables minimal;
• Assign meaningful names to variables;
• Avoid memory leaks by releasing database connections when querying is complete;
• Try to use Finally block as often as possible;
• Use the Executor framework for multithreaded programming.

# 6. Testing Guidelines

Testing is crucial for ensuring the quality and reliability of software. Here's a detailed guide on various aspects of testing, including unit testing, integration testing, test naming conventions, and test data.

**Unit Testing**

**1. Purpose:**

- **Isolation:** Verify the functionality of individual units or components in isolation.
- **Validation:** Ensure each unit of code performs as expected independently.

**2. Best Practices:**

- **Test One Thing:** Each unit test should focus on a single aspect or behavior of the unit.
- **Use Mocks/Stubs:** Isolate the unit from external dependencies (e.g., databases, APIs) by using mocks or stubs.
- **Automate Tests:** Ensure unit tests are automated and part of the continuous integration pipeline.

- **Maintainability:** Write tests that are easy to read, understand, and maintain. Use descriptive names and clear assertions.
- **Fast Execution:** Unit tests should run quickly to encourage frequent testing and feedback.

**3. Tools:**

- **JUnit:** It is a testing framework for Java that provides annotations and assertions to create and manage test cases. It helps in writing repeatable tests, managing test execution, and reporting results.

**Annotations:** Simplify test creation and management.
`@Test`: Marks a method as a test method.
`@Before`: Runs before each test method.
`@After`: Runs after each test method.
`@BeforeClass`: Runs once before all test methods in the class.
`@AfterClass`: Runs once after all test methods in the class.

**Assertions:** Methods to validate test results.
`assertEquals(expected, actual)`: Checks if the expected value matches the actual value.
`assertTrue(condition)`: Checks if the condition is true.
`assertFalse(condition)`: Checks if the condition is false.
`assertNull(object)`: Checks if the object is null.
`assertNotNull(object)`: Checks if the object is not null.

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorTest {

    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals(5, result, "2 + 3 should equal 5");
    }
}
```

- **Mockito:** It is a mocking framework for Java that allows you to create mock objects, which can simulate the behavior of real objects. It is used to isolate the unit being tested by controlling the behavior of dependencies.

**Key Features:**

- **Mocking:** Create mock objects to simulate complex or external dependencies.
- **Stubbing:** Define behavior for mock objects when specific methods are called.
- **Verification:** Verify interactions with mock objects to ensure they occur as expected.
- **Spy:** Create partial mocks that allow you to call real methods while still allowing stubbing and verification.

```
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class UserServiceTest {

    @Test
    public void testGetUser() {
        // Create a mock of UserRepository
        UserRepository mockRepository = mock(UserRepository.class);

        // Define behavior for the mock
        when(mockRepository.findById(1)).thenReturn(new User(1, "John Doe"));

        // Use the mock in the test
        UserService userService = new UserService(mockRepository);
        User user = userService.getUser(1);

        // Verify the result
        assertEquals("John Doe", user.getName());
    }
}
```

**Integration Testing**

**1. Purpose:**

- **Interaction Testing:** Verify that multiple components or systems work together correctly.
- **End-to-End Scenarios:** Ensure integrated components interact as expected in a more realistic environment.

**2. Best Practices:**

- **Environment:** Use a testing environment that closely mirrors production to detect issues early.
- **Isolate Tests:** Minimize external dependencies; use test doubles where possible.
- **Setup and Teardown:** Properly configure and clean up the environment before and after tests to avoid interference.

**Test Naming Conventions**

**1. General Principles:**

- **Descriptive Names:** Commit messages should clearly describe the purpose and scope of the change.
- **Consistency:** Use a consistent naming pattern across your test suite to improve readability and understanding.

**2. Naming Patterns:**

- **Unit Tests:**
  - **Format:** `methodName_StateUnderTest_ExpectedBehavior`
  - **Example:** `calculateTotalAmount_ValidItems_ReturnsCorrectTotal`
- **Integration Tests:**
  - **Format:** `integration_testName_Scenario_ExpectedOutcome`
  - **Example:** `integration_testUserLogin_ValidCredentials_Success`
- **End-to-End Tests:**
  - **Format:** `featureName_Scenario_ExpectedOutcome`
  - **Example:** `checkoutFlow_UserCompletesPurchase_OrderPlacedSuccessfully`

**Test Data**

**1. Types of Test Data:**

- **Static Data:** Fixed data used for common scenarios.
- **Dynamic Data:** Generated during test execution, useful for scenarios with variable input.
- **Mock Data:** Simulated data for testing purposes, avoiding the use of actual production data.

**2. Best Practices:**

- **Representativeness:** Ensure test data covers a range of scenarios, including edge cases and invalid inputs.
- **Isolation:** Use isolated test data to prevent tests from interfering with each other.
- **Clean-Up:** Ensure that test data is cleaned up after tests to maintain a consistent state.
- **Consistency:** Use a consistent approach to generate and manage test data.

**3. Tools and Techniques:**

- **Factory Libraries:** Tools like Factory Boy (Python) or Java Faker for generating test data.
- **Database Fixtures:** Tools or scripts to load predefined data into databases for testing.

- **Mocking Libraries:** Mocking frameworks like Mockito (Java) or unittest.mock (Python) to simulate external dependencies.

- **Focus on Critical Paths:** Test key interactions and scenarios that are likely to be used in real-world situations.

**4. Tools:**

- **Spring Boot Test:** For Java applications using Spring.
- **Postman:** For API testing and integration.

# 7. Documentation and Code Comments

**JavaDoc**

**1. Overview:**

- **JavaDoc** is a documentation tool for Java source code. It generates HTML-based documentation from comments in the source code.
- It provides a structured way to document classes, methods, fields, and other code elements.

**2. Key Features:**

- **Standardized Format:** Provides a consistent format for documenting code elements.
- **Automatic Generation:** Generates HTML documentation that can be used to create user manuals and API references.

**3. JavaDoc Tags:**

- `@param`: Documents a parameter of a method.
  - **Example:** `@param name The name of the person.`
- `@return`: Documents the return value of a method.
  - **Example:** `@return The full name of the person.`
- `@throws` or `@exception`: Documents exceptions thrown by a method.
  - **Example:** `@throws IllegalArgumentException If the input is invalid.`
- `@see`: References related classes, methods, or external resources.
  - **Example:** `@see java.util.List`
- `@since`: Indicates the version since the feature has been available.
  - **Example:** `@since 1.0`
- `@deprecated`: Marks elements that are outdated and should not be used.
  - **Example:** `@deprecated Use the newMethod() instead.`

```java
/**
 * Represents a person with a name and age.
 *
 * @since 1.0
 */
public class Person {

    private String name;
    private int age;

    /**
     * Constructs a Person with the specified name and age.
     *
     * @param name The name of the person.
     * @param age The age of the person.
     * @throws IllegalArgumentException If age is negative.
     */
    public Person(String name, int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative.");
        }
        this.name = name;
        this.age = age;
    }

    /**
     * Gets the name of the person.
     *
     * @return The name of the person.
     */
    public String getName() {
        return name;
    }

    /**
     * Sets the name of the person.
     *
     * @param name The new name of the person.
     */
    public void setName(String name) {
        this.name = name;
    }
}
```

**Inline Comments**

**1. Overview:**

- **Inline Comments** are comments placed within the code to explain specific lines or sections.
- They help in understanding the logic and purpose of the code at a granular level.

**2. Best Practices:**

- **Be Concise:** Comments should be brief but clear. Avoid unnecessary verbosity.

- **Focus on Why, Not How:** Explain why something is done, not just how. The code itself should be clear on the "how."
- **Avoid Redundancy:** Do not restate what the code is doing; instead, provide additional context or reasoning.
- **Keep Comments Updated:** Ensure comments reflect any changes made to the code.

**3. Types of Inline Comments:**

- **Single-Line Comments:** Use `//` for comments that fit on one line.
    - **Example:** `// Increment the counter`
- **Block Comments:** Use `/* ... */` for comments that span multiple lines

```java
public class Calculator {

    // This method adds two numbers.
    public int add(int a, int b) {
        // Perform the addition
        return a + b;
    }

    /*
     * This method divides two numbers.
     * Throws ArithmeticException if the divisor is zero.
     */
    public double divide(double dividend, double divisor) {
        if (divisor == 0) {
            throw new ArithmeticException("Cannot divide by zero.");
        }
        return dividend / divisor;
    }
}
```

## 8. Security Practices

Ensuring the security of software systems involves a range of practices and techniques to protect against vulnerabilities and unauthorized access. Here's a detailed guide on code security, SQL injection prevention, and authentication and authorization practices:

**Code Security**

1. **General Principles:**

- **Secure Coding Standards:** Follow established secure coding guidelines and practices, such as those provided by OWASP or similar organizations.
- **Input Validation:** Always validate and sanitize user inputs to prevent injection attacks and ensure data integrity.
- **Error Handling:** Handle errors gracefully and avoid exposing stack traces or internal error details to end users.
- **Dependencies Management:** Regularly update libraries and dependencies to address known vulnerabilities.
- **Code Reviews:** Conduct regular code reviews to identify potential security issues and enforce secure coding practices.

2. **Common Practices:**

- **Least Privilege Principle:** Ensure that each part of the system operates with the minimum privileges necessary.
- **Secure Data Storage:** Encrypt sensitive data at rest and in transit. Use strong encryption algorithms and manage keys securely.
- **Secure Configuration:** Ensure that configuration settings, such as those for databases or servers, are secure and follow best practices.
- **Regular Patching:** Apply security patches and updates promptly to fix known vulnerabilities.

**SQL Injection**

1. **Overview:**

- **SQL Injection** is a code injection technique that exploits vulnerabilities in an application's interaction with a database. It allows attackers to execute arbitrary SQL queries.

2. **Preventive Measures:**

- **Use Prepared Statements:** Always use prepared statements or parameterized queries to separate SQL logic from data inputs.
    - **Example (Java with JPA):**

```java
@Repository
public interface UserRepository extends JpaRepository<UserEntity, UUID>, JpaSpecificationExecutor<UserEntity> {

    10 usages  suroj.awal
    UserEntity findByUsername(String username);

    1 usage  suroj.awal
    boolean existsByUsernameAndIdNot(String email, UUID id);

    no usages  suroj.awal
    Page<UserEntity> findAllByIdNot(UUID userId, Specification<UserEntity> userEntitySpecification, Pageable pageable);

    1 usage  suroj.awal
    List<UserEntity> findAllByIdIn(List<UUID> userIds);
}
```

- **Input Validation:** Validate and sanitize all user inputs to ensure they meet expected formats and constraints.
- **Stored Procedures:** Use stored procedures with parameters to encapsulate SQL logic, but be cautious as they are not immune to SQL injection if used improperly.
- **Least Privilege Database User:** Use a database user with the least privileges necessary for the application to operate. Avoid using administrative accounts for application access.
- **Escape User Inputs:** Properly escape user inputs if dynamic SQL is unavoidable. However, this is less secure compared to prepared statements.

**Example of Vulnerable Code:**

```java
Statement statement = connection.createStatement();
String query = "SELECT * FROM users WHERE username = '" + username + "'";
ResultSet resultSet = statement.executeQuery(query);
```

**Mitigated Version:**

```java
@Repository
public interface RoleRepository extends JpaRepository<RoleEntity, UUID>, JpaSpecificationExecutor<RoleEntity> {

    4 usages  suroj.awal
    RoleEntity findByName(String name);

    1 usage  suroj.awal
    List<RoleEntity> findAllByIdIn(List<UUID> id);

    2 usages  suroj.awal
    @Query("SELECT COUNT(u) FROM RoleEntity r JOIN r.users u WHERE r.id = :roleId")
    int getUserCountForRole(@Param("roleId") UUID roleId);

    1 usage  Binod Nakhan
    boolean existsByNameAndIdNot(String name, UUID id);


}
```

**Authentication and Authorization**

1. **Authentication:**

- **Overview:** Authentication is the process of verifying the identity of a user or system.
- **Best Practices:**
    - **Strong Password Policies:** Enforce strong password policies, including complexity, length, and expiration.
    - **Multi-Factor Authentication (MFA):** Implement MFA to add an extra layer of security beyond just passwords.
    - **Secure Password Storage:** Store passwords securely using hashing algorithms like bcrypt or Argon2. Never store passwords in plaintext.
    - **Use Authentication Libraries:** Leverage well-established authentication libraries or frameworks to handle authentication securely.

2. **Authorization:**

- **Overview:** Authorization determines what actions a user or system is allowed to perform after authentication.
- **Best Practices:**
    - **Role-Based Access Control (RBAC):** Implement RBAC to manage permissions based on user roles. Ensure users have access only to the resources and actions they need.
    - **Principle of Least Privilege:** Grant the minimum level of access necessary for users to perform their tasks.
    - **Access Control Lists (ACLs):** Use ACLs to define permissions for individual resources or objects.

3. **Example of Authentication Using Spring Security (Java):**

```java
package com.swifttech.user.config.security;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.oauth2.jwt.JwtDecoders;
import org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticationConverter;
import org.springframework.security.web.SecurityFilterChain;

suroj.awal +2
@Configuration
@EnableMethodSecurity(prePostEnabled = true)
public class SecurityConfig {

    @Value("http://localhost:8000")
    String issuerUri;

    suroj.awal +2
    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.authorizeHttpRequests(auth -> auth.requestMatchers("/") AuthorizedUrl
                        .permitAll() AuthorizationManagerRequestMat...
                        .anyRequest() AuthorizedUrl
                        .permitAll())
                .cors(AbstractHttpConfigurer::disable)
                .csrf(AbstractHttpConfigurer::disable)
                .oauth2ResourceServer(
                        oauth2 -> oauth2.jwt(jwt -> jwt.decoder(JwtDecoders.fromIssuerLocation(issuerUri)).jwtAuthenticationConverter(customJwtAuthenticationConverter())))
                .build();
    }

    Binod Nakhan
    @Bean
    public JwtAuthenticationConverter customJwtAuthenticationConverter() {
        JwtAuthenticationConverter converter = new JwtAuthenticationConverter();
        converter.setJwtGrantedAuthoritiesConverter(new CustomJwtConverter());
        return converter;
    }
}
```

**4. Example of Authorization Check in Code:**

```java
@Operation(summary = "Change Password")
@ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Password changed successfully."),
        @ApiResponse(responseCode = "400", description = "Bad request! when field validation failed"),
        @ApiResponse(responseCode = "500", description = "Internal server error! any unhandled exception will be catch here.")
})
@Parameters({@Parameter(name = "oldPassword", description = "Old Password"),
        @Parameter(name = "newPassword", description = "New Password"),
        @Parameter(name = "confirmNewPassword", description = "Confirm new Password")})
@PutMapping("/auth/change/password")
@PreAuthorize("hasAnyAuthority('SUPER_ADMIN', 'CHANGE_PASSWORD')")
public ResponseEntity<Response> changePassword(@Valid @RequestBody ChangePasswordRequest request) {
    return ResponseEntity.ok(authService.changePassword(request));
}
```

**Summary**

- **Code Security:** Follow secure coding standards, manage dependencies, validate inputs, handle errors properly, and conduct code reviews.
- **SQL Injection:** Prevent SQL injection by using prepared statements, validating inputs, using least privilege principles, and escaping inputs if necessary.
- **Authentication and Authorization:** Implement strong authentication practices (e.g., MFA, secure password storage) and enforce proper authorization using RBAC, ACLs, and least privilege principles.

# 9. Performance Optimization

Performance optimization is crucial for ensuring that software runs efficiently and scales effectively. Here's a detailed guide on performance optimization for Java code and SQL queries:

**Java Code Optimization**

1. **Profiling and Benchmarking:**

- **Profiling Tools:** Use profiling tools such as VisualVM, or JProfiler to identify performance bottlenecks in your application.
- **Benchmarking:** Use JMH (Java Microbenchmarking Harness) for precise measurement of code performance.

2. **Efficient Algorithms and Data Structures:**

- **Algorithm Choice:** Choose appropriate algorithms and data structures based on the problem. For example, use hash-based collections (e.g., `HashMap`) for quick lookups and linked lists for frequent insertions and deletions.
- **Complexity Analysis:** Analyze and optimize the time and space complexity of your algorithms.

3. **Avoid Premature Optimization:**

- **Focus on Critical Paths:** Optimize parts of the code that have a significant impact on performance, rather than making broad optimizations.

4. **Memory Management:**

- **Garbage Collection:** Understand and tune JVM garbage collection (GC) settings based on your application's needs. Consider using different GC algorithms like G1GC or ZGC if necessary.
- **Object Allocation:** Minimize unnecessary object creation to reduce the overhead on the garbage collector. Reuse objects where possible.

5. **Concurrency and Parallelism:**

- **Thread Management:** Use concurrency utilities from `java.util.concurrent` to manage threads and tasks efficiently. Avoid manual thread management and synchronization if possible.
- **Parallel Streams:** Use parallel streams for parallel processing where appropriate, but be aware of the overhead and potential performance impact.

6. **Efficient I/O Operations:**

- **Buffered I/O:** Use buffered I/O streams (`BufferedReader`, `BufferedWriter`) for reading and writing data to reduce the number of I/O operations.
- **Asynchronous I/O:** Use asynchronous I/O operations (e.g., NIO, AIO) for non-blocking operations.

7. **Database Access Optimization:**

- **Connection Pooling:** Use connection pooling libraries (e.g., HikariCP) to manage database connections efficiently.
- **Batch Processing:** Use batch processing for large numbers of database operations to reduce the number of database round-trips.

8. **Code Example:**

```java
// Inefficient example: creating objects in a loop
for (int i = 0; i < 1000; i++) {
    String str = new String("value"); // New object each time
}

// Optimized example: using intern() to reuse strings
for (int i = 0; i < 1000; i++) {
    String str = "value".intern(); // Reuse the same object
}
```

**SQL Performance Optimization**

1. **Indexing:**

- **Use Indexes:** Create indexes on columns that are frequently used in `WHERE` clauses, joins, and sorting.
- **Avoid Over-Indexing:** While indexes improve query performance, too many indexes can slow down write operations and consume additional disk space.

2. **Query Optimization:**

- **Use EXPLAIN:** Use the `EXPLAIN` statement to analyze how your queries are executed and identify bottlenecks.
- **Avoid SELECT :* Specify only the columns you need rather than using `SELECT *` to reduce the amount of data processed and transferred.
- **Optimize Joins:** Use appropriate join types (e.g., `INNER JOIN`, `LEFT JOIN`) and ensure join conditions are indexed.

3. **Database Schema Design:**

- **Normalization:** Normalize your database schema to reduce redundancy, but be aware of the trade-offs with query performance.

- **Denormalization:** In some cases, denormalization may be necessary for performance optimization, especially for read-heavy applications.

4. **Query Caching:**

- **Enable Caching:** Use query caching mechanisms provided by the database (e.g., MySQL Query Cache) to cache frequently accessed query results.
- **Application-Level Caching:** Implement caching strategies at the application level using tools like Redis or Memcached.

5. **Database Configuration:**

- **Tune Configuration:** Adjust database configuration settings such as buffer sizes, cache sizes, and connection limits based on your workload and performance needs.
- **Monitor Performance:** Regularly monitor database performance and adjust configurations as needed.

6. **Database Maintenance:**

- **Regular Maintenance:** Perform regular maintenance tasks such as optimizing tables, updating statistics, and rebuilding indexes.
- **Backup and Recovery:** Ensure that backup and recovery processes do not impact performance. Use tools and strategies that minimize downtime and performance degradation.

7. **Code Example:**

**Inefficient Query:**

```
@Query("SELECT r FROM RoleEntity r JOIN r.users u WHERE r.id = :roleId")
RoleEntity getUserCountForRole(@Param("roleId") UUID roleId);
```

**Optimized Query:**

```
@Query("SELECT r.id, r.name FROM RoleEntity r JOIN r.users u WHERE r.id = :roleId")
RoleEntity getUserCountForRole(@Param("roleId") UUID roleId);
```

**Summary**

- **Java Code Optimization:** Focus on efficient algorithms, memory management, concurrency, and I/O operations. Use profiling tools and optimize critical paths without premature optimization.
- **SQL Performance Optimization:** Use indexing, optimize queries, design schemas efficiently, and configure databases properly. Regularly monitor and maintain database performance.

# 10. Continuous Integration

Continuous Integration (CI) and Continuous Deployment (CD) are crucial practices for modern software development, enabling frequent integration and deployment of code changes. Here's a detailed guide on CI/CD pipelines and build scripts:

**CI/CD Pipelines**

**1. Overview:**

- **Continuous Integration (CI):** Involves automatically testing and integrating code changes into a shared repository several times a day. The goal is to identify and fix issues early in the development cycle.
- **Continuous Deployment (CD):** Extends CI by automatically deploying code changes to production environments after passing CI tests, ensuring frequent and reliable releases.

**2. Key Components of a CI/CD Pipeline:**

- **Source Code Management (SCM):** Integration with version control systems like Git (GitHub, GitLab, Bitbucket) to automatically trigger pipelines on code changes.
- **Build Automation:**
  - **Compile Code:** Convert source code into executable artifacts (e.g., JAR files, Docker images).
  - **Run Tests:** Execute unit tests, integration tests, and other quality checks.
  - **Package Artifacts:** Create deployable artifacts, such as application packages or Docker images.
- **Deployment:**
  - **Staging Deployment:** Deploy artifacts to a staging or test environment for further validation.
  - **Production Deployment:** Automatically deploy to production if all tests pass and manual approval is given (for manual CD).
- **Monitoring and Notifications:**
  - **Monitoring:** Continuously monitor the pipeline and application for issues.
  - **Notifications:** Send alerts and notifications about pipeline status, build failures, or deployment issues.

**3. Example CI/CD Pipeline Workflow:**

- **Commit Code:** Developer commits code to a version control system.
- **Trigger Pipeline:** The commit triggers the CI/CD pipeline.
- **Build and Test:**
  - **Build:** Compile the code and package it into artifacts.
  - **Test:** Run unit tests, integration tests, and security checks.
- **Deploy:**
  - **Staging:** Deploy the artifacts to a staging environment.

    - **Manual Approval:** Wait for manual approval (if configured) before proceeding to production.
    - **Production:** Deploy the artifacts to the production environment.
- **Monitor:** Monitor the application and pipeline for any issues.

**4. CI/CD Tools:**

- **Jenkins:** An open-source automation server that supports building, deploying, and automating any project.
- **GitLab CI/CD:** Integrated with GitLab, providing built-in CI/CD capabilities.
- **CircleCI:** A cloud-based CI/CD service that offers scalable and fast build and deployment solutions.
- **Travis CI:** A cloud-based CI service that integrates with GitHub.
- **GitHub Actions:** CI/CD workflows integrated into GitHub repositories.

**5. Example Jenkins Pipeline Configuration (Declarative Pipeline):**

```
pipeline {
  agent any

  stages {
    stage('Git Clone') {
      steps {
        git branch: 'dev',
        url: 'https://kodmandali.swifttech.com.np/remit-v2/Backend/account-service-v2.git',
        credentialsId: 'neouser'
      }
    }

    stage('Build') {
      steps {
        sh '''

export BUILD_NUMBER='REMIT.'${BUILD_NUMBER}
sed -i s/PROFILE/development/g Dockerfile
docker build -t  harbor.swifttech.com.np/gmedemo/account:${BUILD_NUMBER} .
cat /var/jenkins_home/secrets/passwd.txt | docker login https://harbor.swifttech.com.np --username admin --password-stdin
docker push  harbor.swifttech.com.np/gmedemo/account:${BUILD_NUMBER}
'''
          }
       }

       stage('Deploy') {
          steps {
              sshPublisher(
                  continueOnError: false, failOnError: true, publishers: [
                  sshPublisherDesc(
                  configName: "k8-remit", transfers: [
                  sshTransfer(
                  sourceFiles: "target/*.zip", removePrefix: "target", remoteDirectory: "/root/", verbose: true, execCommand: '''
cd /root/kubernetes/application/Service1demo
sudo cp /root/kubernetes/application/Servicedemo/account.yml /root/kubernetes/application/Service1demo/
sudo sed -i 's/latest/REMIT.${BUILD_NUMBER}/g' account.yml
kubectl delete -f account.yml -n demo
kubectl apply -f account.yml -n demo
#kubectl rollout restart deployment account -n demo
sudo rm -rf account.yml
''')]])
                          }
      }
  }
}
```

**Build Scripts**

**1. Overview:**

- **Build Scripts** automate the process of compiling, packaging, and preparing code for deployment. They help ensure consistency and reproducibility in the build process.

**2. Common Build Tools:**

- **Gradle:** A flexible build automation tool that supports multiple languages (e.g., Java, Groovy, Kotlin). It uses a Groovy-based DSL (`build.gradle`)

**Summary**

- **CI/CD Pipelines:** Automate the process of integrating code changes, building, testing, and deploying applications. Key components include source code management, build automation, deployment stages, and monitoring.
- **Build Scripts:** Automate the process of building and packaging software. Tools like Maven, Gradle, and Ant help define and execute build processes.