CODE REVIEW PROCESS

Java/SpringBoot

Version Number: 1.1
Last Updated Date: 22-Jan-2024
Last Updated By: Amit Srivastava

Abstract

This document captures the essence of an effective code review process, with emphasis on Java/SpringBoot applications, focusing on the key areas for detecting the gotchas.

Created By: Amit Srivastava
Email Id: amit.srivastava1@publicissapient.com

Table of Contents

Code Review Process

Code review process is an integral part of building robust and maintainable software systems. It involves methodical assessments of code designed to identify bugs, increase code quality, and help developers learn the source code.

Key Purposes of Code Review

Code reviews play a crucial role in enhancing code quality, fostering collaboration among team members, sharing knowledge, and promoting continuous improvement in the software development process.

The key purposes of code reviews can be summarised as below:

1. **Quality Assurance**
   - Bug Identification: Code reviews help identify and catch bugs, logic errors, and other issues early in the development process, reducing the likelihood of defects reaching the production environment.
   - Code Standards Adherence: Code reviews ensure that the code follows the established coding standards and best practices, promoting consistency and readability across the codebase.

2. **Knowledge Sharing & Collaboration**
   - Knowledge Transfer: Code reviews provide an opportunity for team members to share knowledge about the codebase. This helps in disseminating information about different parts of the application among team members.
   - Collaboration: Team members collaborate during code reviews, sharing insights, suggesting improvements, and learning from each other. It fosters a collaborative and supportive team environment.

3. **Codebase Maintainability**
   - Readability and Maintainability: Code reviews emphasize writing clean, readable, and maintainable code. Code that is easy to understand is more likely to be maintained successfully, reducing technical debt and facilitating future development.
   - Refactoring Opportunities: Code reviews may uncover opportunities for refactoring to enhance code structure, eliminate redundancy, and improve performance.

4. **Risk Mitigation**
   - Identifying Risks: Code reviews help identify potential risks early in the development process. This includes architectural risks, security vulnerabilities, and issues related to scalability or performance.
   - Compliance and Security: Code reviews contribute to ensuring that the code complies with security standards and best practices, reducing the risk of security vulnerabilities.

5. **Consistency & Standards**
   - Coding Standards: Code reviews enforce adherence to coding standards and conventions, ensuring a consistent style and structure across the codebase.
   - Design Consistency: Code reviews help maintain consistency in design patterns, architecture, and the overall structure of the code.

6. **Code Ownership & Accountability**
   - Shared Responsibility: Code reviews promote a sense of shared responsibility among team members. Everyone in the team participates in the review process, contributing to the overall quality of the codebase.
   - Accountability: Developers take ownership of their code changes and are accountable for the quality of their contributions. Code reviews provide a mechanism for ensuring accountability.

Effective Code Review

Effective code reviews contribute significantly to the overall success of a software development project.

Below are few of the best practices and strategies to make code reviews more effective:

1. **Set Clear Objectives**
   - Clearly define the objectives of the code review. Whether it's focused on finding bugs, ensuring adherence to coding standards, or sharing knowledge, a clear purpose helps guide the review process.

     Important Note: Do not make it a fault finding mission, rather it should be a fact finding mission.

2. **Review Small Code Changes**
   - Break down larger features into smaller, manageable code changes. Git Pull Request merge could be one such opportunity to do code review. Smaller changes are easier to review, understand, and provide feedback on.

3. **Regular & Timely Reviews**
   - Conduct code reviews regularly as part of the development process. Avoid accumulating a large number of changes before initiating reviews. Timely reviews help catch issues early.

4. **Establish Coding Standards & Adopt Checklist**
   - Define and communicate coding standards and best practices within the team. Also, adopt a code review checklist to ensure that key aspects are considered during the review. This ensures consistency and provides a baseline for code review expectations.

5. **Automated Tools & Linters**
   - Use automated tools and linters to catch common issues, enforce coding standards, and perform static analysis. This helps reduce the manual effort required during code reviews.

     Important Note: One of the most effective linter for the IDE is SonarLint. You can catch most of the *typical* code issues yourself and fix them before someone else catches them.

6. **Rotate Reviewers**
   - Rotate reviewers to ensure that different team members are involved in the review process. This promotes knowledge sharing and prevents single points of failure.

7. **Provide Constructive Feedback**
    - Focus on providing constructive and actionable feedback. Instead of just pointing out issues, suggest improvements and alternatives. Frame feedback in a positive and supportive manner.

8. **Prioritize & Categorize Feedback**
    - Prioritize feedback incorporation, based on its severity and impact. Categorize feedback into different types (e.g., bugs, improvements, suggestions) to help developers focus on the most critical issues first.

9. **Learning Opportunities**
    - Treat code reviews as learning opportunities. Share knowledge about different parts of the codebase, coding techniques, and best practices. Encourage questions and discussions.

10. **Document Decisions**
    - Document decisions made during the code review, especially those related to design or architectural choices. This helps maintain a record of discussions and reasoning.

11. **Use Collaboration Tools**
    - Leverage collaboration tools for code reviews, such as pull request systems with inline commenting features. These tools streamline the review process and facilitate discussions.

12. **Track Metrics**
    - Track and analyse code review metrics, such as review time, comments, and reviewer satisfaction. Use these metrics to identify areas for improvement in the review process.

Key Focus Areas of Code Review

It’s much easier to understand a large codebase when all the code in it is in a consistent style.

It's essential to consider various aspects to ensure the code is robust, maintainable, and follows best practices. The specific areas of focus may vary depending on the project requirements and the team's coding standards. Some of the key focus areas are as below:

1. Functionality & Requirements

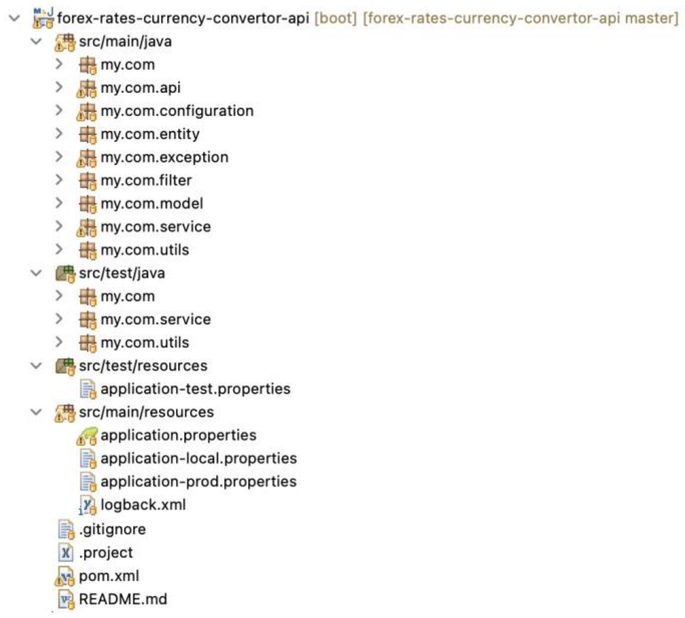
- Verify that the code meets the functional requirements and specifications.
- Ensure that all edge cases and error scenarios are handled appropriately.
- Confirm that the business logic is implemented correctly and efficiently.

Important Note: The best practice is to define the Acceptance Criteria of a story in Gherkin / Cucumber BDD style, which makes it a lot easier and effective to verify the conformance of the code with the expected outcome by running automated test cases. Read More.

2. Code Structure & Organization

- Check if the code follows a clear and consistent module structure/layering, adhering to the project's coding standards and guidelines.

  Important Note: In a very typical Java/SpringBoot Maven project set-up, one would see the following code structure.



```
forex-rates-currency-convertor-api [boot] [forex-rates-currency-convertor-api master]
  src/main/java
    my.com
    my.com.api
    my.com.configuration
    my.com.entity
    my.com.exception
    my.com.filter
    my.com.model
    my.com.service
    my.com.utils
  src/test/java
    my.com
    my.com.service
    my.com.utils
  src/test/resources
    application-test.properties
  src/main/resources
    application.properties
    application-local.properties
    application-prod.properties
    logback.xml
  .gitignore
  .project
  pom.xml
  README.md
```

- Review the organization of packages, classes, and methods.
- Ensure that code is modular and follows the principles of maintainability.
- Ensure that there is no inconsistent or incorrect use of access modifiers.

3. Code Readability & Naming Conventions

- Ensure that variable and method names are descriptive and meaningful. E.g. Names such as x, temp, foo etc. are not descriptive and can make the code difficult to understand.
- Ensure that long and complex methods or classes are broken into smaller, more manageable methods, and complex classes should be refactored into smaller, more focused classes.
- Ensure that comments are used to explain complex logic or business rules, as well as to provide context and clarity to the code.
- Apply consistent coding style and templates, by integrating standard check style plugins with IDE, to bring consistency in formatting and indentation, making the code better and easier to read and understand.

4. Use of Design Principles & Patterns

- Ensure that there is adherence to SOLID design principles. E.g. Violation of SRP principle makes the code hard to maintain, while violation of Open/Close principle makes the code hard to extend (functionality). Read More.
- Ensure that there is efficient and appropriate use of design patterns. While design patterns provide proven solutions to common software design problems, their wrong use will make

matters complicated and/or inefficient. E.g. Using the Observer/Observable pattern for a simple one-to-one objects interaction will introduce unnecessary overhead. Similarly, lack of the use Factory design pattern will scatter the complex object creation, making it difficult to maintain the code. Read More.

- Ensure that the design adheres to the laid out architectural goals. E.g. If the audit logging is meant to be processed in asynchronous fashion, make sure it is designed and coded that way.

5. Error Handling & Logging

- Ensure adequate Exception handling in the code and ensure that the exception propagation is not broken or swallowed abruptly.
- Ensure that business exceptions are not generic ones e.g. Throw more specific checked exception than throwing *java.lang.Exception* or *java.lang.RuntimeException*
- Ensure adequate handling of the exception by having narrower catch blocks to handle specific exceptions separately. The error stack should be logged appropriately.

  Important Note: Never attempt to catch and/or swallow *java.lang.Throwable* or *java.lang.Error*, as these make the system unstable and its behaviour unpredictable.

- Prefer centralised exception handling by using Controller Advice in case of SpringBoot.
- In case of SpringBoot applications, ensure that meaningful Http status codes are sent in the response, masking the detailed error message, as appropriate.
- In case of SpringBoot, prefer using SpringAOP or AspectJ based logging to bring in consistency in the certain standardised logging aka cross-cutting concerns. E.g. Logging at every method entry and exit.
- Apply standardised logging pattern, which will help the SRE team in building better and more consistent visualizations and alerting rules.
- In case of SpringBoot, use the Logger's MDC (Mapped Diagnostic Context) feature to inject correlation id/trace id with each log statement, which allows better and efficient traceability of request/response call stack in the backend application's stack trace. Read More.
- Use appropriate log levels while logging application data. Typically, the log levels Trace & Debug are enabled only in lower environments; whereas, log levels from Info and above is preferred in Production.

6. Performance & Scalability

- Detect situations where the code makes unnecessary or redundant database calls, try to reduce the queries fetching unwanted data and ensure that fields that are being queried upon are properly indexed.
- Identify use of inefficient data structures algorithms with high time complexity that may result in slow performance.
- Ensure that the code uses *java.lang.StringBuilder* or *java.lang.StringBuffer* to improve the string concatenation performance.
- Identify inefficient file reading or writing operations. Recommend using buffered readers/writers and appropriate file handling techniques, and using try-finally to release the allocated resources once the work gets done.
- In Java/SpringBoot, leverage JVM memory management techniques offered through *java.lang.ref* package i.e. usage of *soft* and *weak* reference rather than *strong* reference, when loading memory consuming resources such as big images or configurations.

- Prefer techniques like lazy loading or caching for optimized resource management e.g. during the creation of heavy-weight object or making network calls.
- In case of Java/SpringBoot, ensure that Java memory profiling tools are run in order to detect memory leaks. JConsole and VisualVM are two most popular Java memory profiling tools.
- Ensure that proper thread synchronization and thread pooling strategies are applied, to avoid thread contentions or resource wastage.
- Look for opportunities to leverage asynchronous programming or parallelization techniques to improve responsiveness and to avoid the blocking operations in the main thread.
- To reduce logging-related performance overheads, ensure async logging and log rotations (for file-based logging) are configured appropriately. Also, ensure that the application is not flooding the logging system with unnecessary logs.
- In case of SpringBoot, evaluate and ensure that there is optimal and efficient use of Spring features such as caching, transactions, and dependency injection.
- In case of SpringBoot microservices, ensure that 12Factor App methodology is followed for building applications; thus, making them easy to scale horizontally.

7. Security & Vulnerabilities

- Ensure that secrets like passwords, API keys etc are not put in plain text in properties files or hard-coded in the source code.
- Ensure proper use of environment layering to allow different secrets and configurable values for different environments.
- Ensure that no access tokens or any such sensitive data is logged as part of application logs, as it violates InfoSec and other legal compliance needs.
- In the context of SpringBoot microservices, ensure that adequate input data validations are in place at the Controller layer.
- Ensure that CORS and CSP rules are implemented adequately. If encryption is used, it’s not weak (i.e. easily breakable) and if any tokens are generated (e.g. CSRF tokens), they’re not easily guessable.
- Preferably, run SAST and SCA tools like Coverity, Checkmarx, SonarQube SAST, Blackduck, Trivy etc to detect rogue code library dependencies, mis-configurations and other security vulnerabilities.
- Similarly, run Docker image scanning tools like Snyk, AquaSec etc to find out vulnerabilities in the Docker images. Ensure that, at least, all the critical & major vulnerabilities are addressed, or follow the standards set by the InfoSec team in that regards.
- When dealing with file uploads, ensure that the code applies necessary restrictions on the specific file types/content and the files are scanned for potential malware.
- Ensure that security-relevant events are logged adequately with complete details. Adequate visualisation and alerting of such events should be implemented.
- Detect improperly managed user sessions, which might potentially lead to session fixation or session hijacking vulnerabilities.
- Working with Zero-Trust Security requirements, ensure that HTTPS/SSL enforcement is done end to end i.e. data in-transit (from one node to another) and data at-rest is all encrypted.

  Important Note: Ensure that any applicable InfoSec or legal compliance requirements like PCI DSS, GDPR etc are met with without fail.

8. Testing & Test Coverage

- Identify lack of unit tests or inadequate unit test coverage or missing edge case unit tests, and get them fixed.
- Ensure that test executions are not utilising/blocking costly and time-consuming resources. Ensure that test data and mocks are built adequately and correctly.
- Ensure that integration test cases and load & performance test cases are available and are executed in timely fashion. Ensure that the test results are analysed duly and the feedback is incorporated.

9. Resiliency & Concurrency

- For the external network calls, ensure that connection/socket timeouts and read/write timeout values are configured appropriately. Wherever connection pools or thread pools are relevant, ensure that they are configured adequately i.e. as per the needs.
- To build resiliency in the application, ensure that the code has incorporated bulk head and circuit breaker patterns, and appropriate re-tries and failovers are built. Ensure that caching frameworks are utilised adequately in order to provide the needful failover.
- Ensure that there is efficient and adequate use of Proxy and DAO (Data Access Objects) design patterns to reduce unwanted network and database access calls.
- In case of SpringBoot microservices, prefer PUSH architecture to allow the dependent microservices to grab/cache the master data changes in their local cache in order to reduce network calls during Live Http requests. Similarly, prefer BFF (Backend For Frontend) pattern to reduce multiple long-hop network calls between client and server.

  Important Note: The more the network calls, the flakier is the eco-system; a lot can go wrong between the long network hops.

- Ensure that adequate logs are generated whenever application gets into a resiliency mode, so that such events can be visualised and proper alerts cab be raised for their remediation.

10. Code Duplication

- Identify and extract identical and common functionality into separate methods or classes to avoid redundancy.
- Detect similar patterns or logic structures that are repeated in multiple places and convert them into reusable parameterised components/functions.
- Identify duplicate POJOs, utility classes, constants, data transformation logic, database queries, config items and validation logic, and consolidate them into shareable libraries.

  Important Note: We❤ copying + pasting stuffs, and creating our Frankenstein monster? 😊

11. Documentation & Comments

- Ensure that code is adequately documented to provide context, explain complex logic, and make it easier for other developers to understand and maintain the code.
- Ensure that the comments in the code are used appropriately to explain the purpose of the code, provide context, and make it easier for other developers to understand and maintain the code.

- In case of Java, ensure that the classes have appropriate Javadocs to generate the HTML version of class documentation. Read More.
- In case of SpringBoot microservices, enable Swagger Documentation additionally. This ensures that the API documentation will neither be missing nor getting outdated.

Google Style Guide

This GitHub project (google/styleguide) links to the style guidelines that Google developers use. This style guide can serve as the common base for project specific style guideline.

Reference: https://google.github.io/styleguide/

- Java Style Guide
- C++ Style Guide
- C# Style Guide
- Go Style Guide