Code Review Document

Version : 1.0.0 | Date : 22/01/2019

Introduction

Objectives

1. Finding bugs
2. Create common Coding style
3. Learning good coding style and best practices
4. Teaching best practices
5. Efficiency (getting pull requests reviewed quickly)
6. Ensuring that the pull request guidelines are followed.

Code formatting

While going through the code, check the code formatting to improve readability and ensure that there are no blockers:

1. Use alignments (left margin), proper white space. Also ensure that code block starting point and ending point are easily identifiable.
2. Ensure that proper naming conventions (Pascal, CamelCase etc.) have been followed.
3. Code should fit in the standard 14 inch laptop screen. There shouldn't be a need to scroll horizontally to view the code. In a 21 inch monitor, other windows (toolbox, properties etc.) can be opened while modifying code, so always write code keeping in view a 14 inch monitor.
4. Remove the commented code as this is always a blocker, while going through the code. Commented code can be obtained from Source Control (like SVN), if required.

Architecture

The code should follow the defined architecture.

1. Separation of Concerns followed
    a. Split into multiple layers and tiers as per requirements (Presentation, Business and Data layers).
    b. Split into respective files (HTML, JavaScript and CSS).
2. Code is in sync with existing code patterns/technologies.

3. Design patterns: Use appropriate design pattern (if it helps), after completely understanding the problem and context.

Non Functional requirements

Maintainability

The application should require the least amount of effort to support in near future. It should be easy to identify and fix a defect.

Readability: Code should be self-explanatory. Get a feel of story reading, while going through the code. Use appropriate name for variables, functions and classes. If you are taking more time to understand the code, then either code needs refactoring or at least comments have to be written to make it clear.

Testability: The code should be easy to test. Refactor into a separate function (if required). Use interfaces while talking to other layers, as interfaces can be mocked easily. Try to avoid static functions, singleton classes as these are not easily testable by mocks.

Debuggability: Provide support to log the flow of control, parameter data and exception details to find the root cause easily. If you are using winston like component then add support for stackdriver logging also, as logs can be stream to Kibana which is easy to query.

Configurability: Keep the configurable values in place (JSON file, database table) so that no code changes are required, if the data is changed frequently.

Reusability

1. DRY (Do not Repeat Yourself) principle: The same code should not be repeated more than twice.
2. Consider reusable services, functions and components.
3. Consider generic functions and classes.

Reliability

Exception handling and cleanup (dispose) resources.

Extensibility

Easy to add enhancements with minimal changes to the existing code. One component should be easily replaceable by a better component.

Security

1. Does the code appear to pose a security concern?
    a. Passwords should not be stored in the code. In fact, we have adopted a policy in which we store passwords in runtime properties files.
    b. Connect to other systems securely, i.e., use HTTPS instead of HTTP where possible.
2. Do input validation against security threats such as SQL injection and Cross Site Scripting (XSS)
3. Encrypt the sensitive data (passwords, credit card information etc) and PII data

Performance

1. Use a data type that best suits the needs.
2. Lazy loading, asynchronous and parallel processing.
3. Caching and session/application data.

Scalability

Consider if it supports a large user base/data? Can this be deployed into cloud platform?

Usability

Put yourself in the shoes of a end-user and ascertain, if the user interface/API is easy to understand and use. If you are not convinced with the user interface design, then start discussing your ideas with the business analyst.

SOLID Principles

Single Responsibility Principle (SRS)

Do not place more than one responsibility into a single class or function, refactor into separate classes and functions.

Open Closed Principle

While adding new functionality, existing code should not be modified. New functionality should be written in new classes and functions.

Liskov substitutability principle

The child class should not change the behavior (meaning) of the parent class. The child class can be used as a substitute for a base class.

Interface segregation

Do not create lengthy interfaces, instead split them into smaller interfaces based on the functionality. The interface should not contain any dependencies (parameters), which are not required for the expected functionality.

Dependency Injection

Do not hardcode the dependencies, instead inject them.

In most cases the principles are interrelated, following one principle automatically satisfies other principles. For e.g: if the 'Single Responsibility Principle' is followed, then Reusability and Testability will automatically increase.

In a few cases, one requirement may contradict with other requirement. So need to trade-off based on the importance of the weight-age, e.g. Performance vs Security. Too many checks and logging at multiple layers (UI, Middle tier, Database) would decrease the performance of an application. But few applications, especially relating to finance and banking require multiple checks, audit logging etc. So it is ok to compromise a little on performance to provide enhanced security.

Coding best practices

1. No hard coding, use constants/configuration values.
2. Write comments and documentation
3. Use helper methods
4. Write test cases. Don't forget the edge cases: 0s, empty strings/lists, nulls, etc.

5. Group similar values under an enumeration (enum).
6. Comments – Do not write comments for what you are doing, instead write comments on why you are doing. Specify about any hacks, workaround and temporary fixes. Additionally, mention pending tasks in your to-do comments, which can be tracked easily.
7. Avoid multiple if/else blocks.
8. Use framework features, wherever possible instead of writing custom code.
9. Keep the installation structure simple: Files and directories should be kept to a minimum. Don't install anything that's never going to be used.
10. Keeping dependencies up-to-date and secure

Git Feature Branch Workflow

The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the master branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the master branch will never contain broken code, which is a huge advantage for continuous integration environments.

Encapsulating feature development also makes it possible to leverage pull requests, which gives other developers the opportunity to sign off on a feature before it gets integrated into the main project.

How it works

1. Create a new-branch
2. Update, add, commit, and push changes
3. Push feature branch to remote
4. Open a Pull request
    a. Make sure the person who is opening the request and merging the request should be different
5. Code Review
6. Resolve feedback
7. Lead/peer approval
    a. If the request is rejected mention reasons in a comment

b. Accepting code on below metric
    i. General code standards
    ii. Business logic
    iii. Not affecting other code
    iv. Modularized code
    v. Code reusability
    vi. Optimization of code
    vii. Any other project Specific metric
c. Merging in the Development branch (Check In Dance)

Code consistency and Quality

Being able to get code written by multiple developers to fit a consistent style and high quality is a good challenge, and one that can be supported by a variety of tools and services such as JShint, Code Climate, SonarQube, NDepend, FxCop, TFS code analysis rules.

To track the code review comments use the tools like GitLab, Bitbucket and TFS code review process.

Code Climate (Recommended for Javascript)

Code Climate is a code analysis tool that helps to identify issues with code similar to linting, but extends on top of this by analysing the complexity of the code, spotting duplicate code, and helping to identify which files could use improvements.

Security Guidelines

Identify the use case of your application. This includes determining whether the application in an intranet application or an internet facing application.

1. If the application is just an intranet application then only the Infrastructure Security measures are adequate.
2. If the application is an internet facing application then the Infrastructure and Application Security measures are mandatory.

Infrastructure Security

Data Security

1. Data lockdown

   Make sure that data is not readable, encrypted at rest and that the solution offers strong key management.

2. Access policies

   Implement access policies that ensure only authorized users/machines can gain access to sensitive information, so that even privileged users such as root user cannot view sensitive information

3. Security intelligence

   Incorporate security intelligence that generates log information, which can be used for behavioral analysis to provide alerts that trigger when users are performing actions outside of the norm

Data in Transit

1. Where the covered device is reachable via web interface, web traffic must be transmitted over Secure Sockets Layer (link is external) (SSL), using only strong security protocols, such as Transport Layer Security (link is external).
2. Application data/PII data (system generated emails) transmitted over email must be secured using cryptographically strong email encryption tools such as PGP or S/MIME. Alternatively, prior to sending the email, developer should encrypt covered data using compliant File Encryption tools and attach the encrypted file to email for transmission.
3. Non-web transmission of covered data should be encrypted via application level encryption.
4. Where application level encryption is not available for non-web covered data traffic, implement network level encryption such as SSH tunneling.

5. In general, encryption should be applied when transmitting covered data between devices in protected subnets with strong firewall controls.

IAM Policies

1. Enable multi-factor authentication (MFA) for privileged users
2. Use Policy Conditions for Extra Security
3. Remove Unnecessary Credentials
4. Use AWS-Defined Policies to Assign Permissions Whenever Possible
5. Use Groups to Assign Permissions to IAM Users
6. Grant Least Privilege
7. Use Customer Managed Policies Instead of Inline Policies
8. Do Not Share Access Keys
9. Rotate Credentials Regularly

VM Instance Access

1. AWS:
    a. Generate your public/private key pair and upload the public key on the instance.
    b. Make sure you generate SSH keys with password
2. GCP:
    a. The SSH access to the instances should be allowed only from Web browser and GCLOUD CLI from Quantiphi/Client's Network.

Cloud Storage

1. Make sure the security best practices are followed for the objects stored on Cloud Storage.
    a. AWS: https://aws.amazon.com/premiumsupport/knowledge-center/secure-s3-resources/
    b. GCP: https://cloud.google.com/storage/docs/best-practices#security
2. Use Cloud CDN to expose static objects stored in buckets
3. For temporary access to the objects use the Pre-Signed URLs.

a. AWS: https://docs.aws.amazon.com/sdk-for-go/v1/developer-guide/s3-example-presigned-urls.html
b. GCP: https://cloud.google.com/storage/docs/access-control/signing-urls-with-helpers

Firewall Rules

1. Check that only the required ports are open and only to the required sources. Eg. If you have hosted your application which listens on port 3000 only from client known IP subnets, then mention those subnets in the source filter of the firewall rules.
2. If your application is expected to receive the traffic from all over the world (0.0.0.0/0), then make sure you expose only port 443
3. Maintain a Global Deny Rule at the bottom of the rule list
4. Ref: https://cloud.google.com/docs/enterprise/best-practices-for-enterprise-organizations#firewall-rules

Application Security

Configuration Management

1. Check for commonly used application and administrative URLs
2. Check for old, backup, and unreferenced files
3. Check HTTP methods supported and Cross Site Tracing (XST)
4. Test file extensions handling
5. Test RIA cross domain policy
6. Test for security HTTP headers (e.g. CSP, X-Frame-Options, HSTS)
7. Check for sensitive data in client-side code (e.g. API keys, credentials)

Secure Transmission

1. Check SSL version, algorithms, and key length
2. Check for digital certificate validity (duration, signature, and CN)
3. Check that credentials are only delivered over HTTPS

4. Check that the login form is delivered over HTTPS
5. Check that session tokens are only delivered over HTTPS
6. Check if HTTP Strict Transport Security (HSTS) in use
7. Test ability to forge requests
8. Test web messaging (HTML5)
9. Check CORS implementation (HTML5)

Authentication

1. Application Password Functionality
    a. Test remember me functionality
    b. Test password reset and/or recovery
    c. Test password change process
    d. Test CAPTCHA
    e. Test for logout functionality presence
    f. Test for default logins
    g. Test for out-of-channel notification of account lockouts and successful password changes
    h. Test for consistent authentication across applications with shared authentication schema/SSO and alternative channels
2. Additional Authentication Functionality
    a. Test for authentication bypass
    b. Test for brute force protection
    c. Test for credentials transported over an encrypted channel
    d. Test for cache management on HTTP (eg Pragma, Expires, Max-age)
    e. Test for user-accessible authentication history

Session Management

1. Establish how session management is handled in the application (eg, tokens in cookies, token in URL)
2. Check session tokens for cookie flags (httpOnly and secure)
3. Check session cookie scope (path and domain)
4. Check session cookie duration (expires and max-age)
5. Check session termination after a maximum lifetime

6. Check session termination after relative timeout
7. Check session termination after logout
8. Test to see if users can have multiple simultaneous sessions
9. Test session cookies for randomness
10. Confirm that new session tokens are issued on login, role change, and logout
11. Test for consistent session management across applications with shared session management
12. Test for session puzzling
13. Test for CSRF and clickjacking

Authorization

1. Test for path traversal
2. Test for vertical access control problems (a.k.a. privilege escalation)
3. Test for horizontal access control problems (between two users at the same privilege level)
4. Test for missing authorization
5. Test for insecure direct object references

Cryptography

1. Check if data which should be encrypted is not
2. Check for wrong algorithms usage depending on context
3. Check for weak algorithms usage
4. Check for proper use of salting
5. Check for randomness functions

Data Validation

1. Test for HTML Injection
2. Test for SQL Injection
3. Test for LDAP Injection
4. Test for ORM Injection
5. Test for XML Injection
6. Test for XXE Injection
7. Test for SSI Injection

8. Test for XPath Injection
9. Test for XQuery Injection
10. Test for IMAP/SMTP Injection
11. Test for Code Injection
12. Test for Expression Language Injection
13. Test for Command Injection
14. Test for NoSQL injection
15. Test for Overflow (Stack, Heap and Integer)
16. Test for Format String
17. Test for incubated vulnerabilities
18. Test for HTTP Splitting/Smuggling
19. Test for HTTP Verb Tampering
20. Test for Open Redirection
21. Test for Local File Inclusion
22. Test for Remote File Inclusion
23. Test for HTTP parameter pollution
24. Test for auto-binding
25. Test for Mass Assignment
26. Test for NULL/Invalid Session Cookie
27. Test for integrity of data

File Uploads

1. Test that acceptable file types are whitelisted and non-whitelisted types are rejected
2. Test that file size limits, upload frequency and total file counts are defined and are enforced
3. Test that file contents match the defined file type
4. Test upload of malicious files
5. Test that unsafe filenames are sanitized
6. Test that uploaded files are not directly accessible within the web root
7. Test that uploaded files are not served on the same hostname/port

Error Handling

1. Check for Error Codes
2. Check for Stack Traces

What not to do

Design

Code review is a process to review existing code and learn from it, not the time or place to discuss design ideas. If design ideas come to mind, write them down separately and discuss them outside of the code review.

Writing code

If you are writing any code, you are not reviewing. The only exception would be a short line of two of code within a comment to help communicate a defect or suggested fix.

Judge

Code review is a chance for everyone to learn and make better code while improving the code for the community. We all make mistakes and we all have opportunities to learn. Defects in code are an opportunity to learn and improve for everyone.

Version Control

Sr No	Version No	Date	Done By- Name (Designation)	Reviewed by- Name & Designation	Approved by Designation
1	1.0	24 January 2019	Parvez Javed Shaikh (Technical Lead) Nilay Kamat (Senior Software Developer) Lalit Bhosle (Platform Engineer)	Ritesh Patel (Founder)	Ritesh Patel (Founder)