Software Engineering Standards and Guidelines for Development Projects

This guide and sub pages associated with it are to serve as the basis for how software engineering is to be conducted within the Software Engineering group.

Definitions

- The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.
- Software is considered "production-ready" or "production grade" when it has passed formal acceptance criteria and will be deployed for a downstream client.
- Software is considered "not production-ready" when it has not passed formal acceptance criteria, is actively under development or is actively being tested.
- Deployment is the process of taking an application's runnable artifact - a JAR, WAR, executable script or otherwise - from an artifact store (such as Artifactory) and delivering it to a server.

Build, Deploy and Artifact Store

Artifact Store

1. Artifacts uploaded to any artifact store must not contain any hard-coded secrets.
2. All software written for production deployment shall be deployed to an internal artifact store for distribution and tracking purposes, with the following exceptions:

a. Applications which rely on legacy credential management

3. Applications should follow the artifact convention of the platform in which they're written for.

a. Java artifacts must be deployed to a Maven-backed repository.

i. Production-ready artifacts must deploy to a -release repository.

ii. Non-production-ready artifacts must deploy to a -snapshot repository.

Automated Build and Deploy

1. All software written for production deployment shall have an associated production-grade build on a build platform, such as Jenkins.

2. Non-production builds must run the unit tests of the project.
a. Builds are recommended but not required to run their unit tests when new code is committed on any branch.
3. All software written for production deployment shall have an associated production-grade deployment workflow to ensure that the right artifacts are published, and that they are published to the correct environments.
a. This workflow should ideally be built off of existing work to deploy to prior existing environments, such as development or test.

Source Control

Source Control

1. The master branch should represent code in a production-ready state.
2. No sensitive credentials, hard-coded or otherwise, shall be committed to source control.
3. All software written shall be placed into our source control repository (currently https://stash.int.colorado.edu), in an appropriately named repository.
a. Engineers may elect to place their code into a derivative project repository as opposed to the main repository (D&I).
4. Projects should have a README under source control so that a summary of information about the application is made available to future maintainers.
a. Common information in this would include how to build and deploy the application, what access is required (database, server, etc), what the application itself depends on, and what credentials, if any, are expected to be used.
5. Engineers should commit their code as frequently as possible to ensure no loss of work.
6. Commit messages should be descriptive and contain sufficient detail to illustrate what change has transpired.
7. Commits should be tagged to indicate which version has been released most recently.
a. It is required that tags follow the spirit of SemVer by tagging with major.minor.patch.
8. Feature work done should be done on a separate branch to master.
a. Feature branches should be prefixed with an associated JIRA number for the task along with a description of the feature.
i. e.g. A JIRA task ABC-123 with a feature to "Modify logging around Baz widgets" should have a branch name of ABC-123-modify-baz-widget-logging, as an example.
ii. This allows JIRA to link with Bitbucket to allow for easier tracking of effort.
b. Feature branches shall be merged into master once the feature is complete.
c. Feature branches should be deleted from the repository once they have been merged, or once they no longer serve any purpose.
9. It is recommended but not required for project teams to adopt a branching convention.
a. Git Flow is but one branching convention out there.

Testing and Code Reviews

Unit Testing

1. All software written for production deployment shall be tested.
2. Unit tests should typically cover one public-facing method.
a. It is recommended to have one assert statement for each test.
b. If it is not possible to have a single assert, it is recommended to have the asserts closely related to one another.
3. Exercise restraint **and** caution with mocking frameworks and mock/stub data.
a. Be certain what you're testing only uses the mock or stub as a test invariant.
4. Reference Clean TDD 2015 as a guideline on what to do in general.
. This is not necessarily a recommendation for TDD or TDD principles.

Integration Testing

1. Integration tests should be leveraged when appropriate.
a. It is recommended to write integration tests sparingly and when needed to exercise several units of code.
2. The use of mocking frameworks when writing integration tests must be thoroughly documented and justified.
3. Software which contains integration tests must allow for these to be run underneath a separate profile to separate them out from quicker unit tests.

Code Reviews

1. All software written for production deployment shall be subject to a code review.
2. Code reviews may take place between peers on the same project, or peers with domain knowledge on a different project.
3. Reviewers shall inspect code...
a. ... to be sure that no known and apparent security vulnerabilities - reference the current OWASP Top 10 as a *baseline*.
b. ... to be sure that there is sufficient documentation where appropriate.
c. ... to be sure that there isn't a lot of commented-out code, as that adds clutter to the actual code itself.
d. ... to be sure that secrets, where required, are not hard-coded in the application or are otherwise checked into source control.
e. ... to be sure that the code contains clear variable, method/function and class names, and is well-formatted.
f. ... to be sure that the code does not contain any unused or dubious dependencies.
g. ... to be sure that the additions to the application are sufficiently tested.
4. It is recommended, where possible, for reviewers to load and run the code to ensure that the behavior listed in the code review is as described.

Code Style

1. It is strongly recommended to preserve the style of the code "in its natural state".
a. Making wholesale style changes (such as changing tabs to spaces, or changing indentation levels) without first discussing it within the project team is discouraged.
2. The conventions of the programming language dictate how variables and method names are defined.
a. For Java, JavaScript and PHP, camelCase method/function and variable names. TitleCase for class names.
i. XML used in POMs should ideally be indented by four (4) spaces.
ii. Spring (Boot) configuration should use hyphen-case for long variable names.
b. For Python and Ruby, snake_case function and variable names. TitleCase for class names.
3. It is recommended to chop down chained method calls, such that each method appears on its own separate line.
4. Efforts which contribute to open source projects must follow the code style as mandated and laid out in that project, superseding *all* internal policy.
a. If a project does not mandate a style, follow the style of the code in its natural state.

Language, Library, Framework and Platform

Security Considerations

1. Written software should not leverage any libraries or frameworks with known CVE vulnerabilities to it.
2. Software which has a known vulnerability should be triaged to determine the effort of patching and work out a reasonable time frame to complete the patching effort.

Selection

1. Be sure to consider the opinion and thoughts of those participating on the project when selecting a language, library or framework - especially if it deviates from what most projects have leveraged.
2. Decisions about this should be presented to the Architects Review Board, and presenters must be prepared to justify their decisions.
3. When evaluating a language, consider:
a. How many people know it already
b. How long it would take others to pick up
c. How accessible documentation and resources are for those starting it
4. When evaluating a library, consider:
a. How well it solves the problem faced
b. How well documented it is
c. How well supported it is
d. How frequently it has been updated or maintained by core developers
e. How easy it is to integrate with your code
f. How easy it is to validate and test your usage of it
5. When evaluating a framework, consider:
a. How well it suits the domain it's being applied to
b. How well documented it is
c. How well supported it is
d. How frequently it has been updated or maintained by core developers
e. How easy it is to validate and test its usage

In-house Maintenance of Libraries

1. At Software Engineering's discretion, patches and bug fixes may be submitted upstream to the maintainers of a library.
2. Until upstream patches are merged into the main project, Software Engineering may continue to leverage a fork of said upstream library.