Standard of Practice

Middleware Software Development Standards

Summary

1. The GitLab service at https://code.vt.edu is the hub of the software development and deployment process.
2. Jira is the issue management system for most software products.
3. Middleware uses a review-then-merge process for changes to the mainline branch.
4. Projects MUST use a build system to produce software artifacts.
5. A GitLab CI pipeline is the standard process for executing the build system to produce and deploy software artifacts.
6. Style checkers and linters should be integrated into the build to ensure consistent style.
7. Unit tests are required; integration tests are optional, but encouraged.
8. Technical documentation should be provided as needed: code, module, subsystem, product.

Software Development Workflow

The Virginia Tech GitLab service is the hub of the software development workflow. All source code lives in GitLab and all meaningful changes are made to the mainline branch (typically *master*/*main*) by way of merge requests (MR). All merge requests should be reviewed by a distinct developer from the one that wrote the code. While reviews are often perfunctory rubber stamps in practice as the reviewer may not have the level of expertise of the author in the target module or product, reviewers are encouraged to evaluate code on the following bases at a minimum:

1. Sensible naming (component, property, and variable)
2. Use of common design patterns
3. Adequate test coverage

A code review is not a doctoral dissertation defense; reviewers should avoid blocking forward progress unless there are serious diversions from convention or significant implementation problems are discovered. A reviewer should generally spend no more than a couple hours reviewing changes; developers should generally spend no more than a half-day responding to code review feedback. Large change sets are of course an obvious exception to this rule of thumb.

Merge requests are typically accepted and merged if there are no conflicts. In the case of a merge conflict, the reviewer simply accepts the MR, which is the trigger for the developer to rebase changes and force-push to the ephemeral branch that is the base of the MR. The developer then merges the branch using the GitLab UI. **Be careful** to use the *Modify commit message* button to pick an existing commit message; by default only the summary of the MR is used in the commit message, which often drops useful if not vital context about the change. Since most Middleware projects use a simple commit strategy rather than merge commit, the text of the merge request commit is vitally important.

When a merge request is used to implement a feature or fix a software defect that is tracked in Jira, it is important to cross-reference the merge request in GitLab with the Jira issue. This is most easily accomplished by mentioning the Jira issue number in the commit summary (see next section) and adding a comment to the Jira issue that contains a link to the merge request.

Commit Messages

The following guidelines are in the spirit of https://cbea.ms/git-commit/, which is arguably the gold standard for git commit message best practices. Please use the following example as a template for writing commit messages:

```
JIRA-1234 Fix resource leak in HTTP conn pool.

Use try-with-resources pattern to ensure connections are always
closed. Clean up thread-local variables in finally block.
```

The first line should include the Jira issue number for any issue that is fixed or feature that is implemented by the commit. The summary should be at most 50 characters. If there are notable aspects of the commit, they should be summarized on subsequent lines with a blank line between the summary and details. Use clear and concise language that summarizes changes rather than repeat implementation details. Lines in the details should not exceed 72 characters.

Git Branch Naming Conventions

The following sections discuss common cases for which we have defined naming conventions. In general, branch names should use kebab-case when multiple words are required to describe the branch; exceptions are clearly noted below.

Main Branch

The main branch in every Middleware project is named *main* according to current best practice for Git repositories.

Version Branch

Sometimes it is necessary to maintain a branch that tracks a different software version than the main branch; for example, when a new feature is being developed for the next version that would conflict with whatever is in production. For this special case, we simply use the static name *prod_branch* to describe the branch that tracks production. If there were ever a case where *multiple* branches needed to be maintained, such as a library project where multiple older versions are still being maintained, the following convention should be used:

```
v-$version
```

When the product lifecycle results in end of life for the version, the branch is typically deleted.

Feature Branch

Feature branch names are long-lived branches where disruptive or experimental features are being developed that would present problems if development were done on the main branch, so a feature branch is used to insulate the main branch from churn. A feature branch should conform to the following convention:

```
feature-$summary
```

Where `$summary` is a brief description of the feature under development; for example a feature branch targeting an upgrade to Hibernate 6 may be called:

```
feature-hibernate-6
```

Feature branches should be treated similarly to the main branch where merge requests are used to incorporate software changes in a task-oriented fashion for Kanban process integration; merge requests also facilitate peer review of code in line with best practice. When software development is completed on the feature branch, the branch is typically merged to the main branch and the branch is deleted.

Task/Issue Branch

This is the common case for the software development workflow where a development task to be completed is tracked on a separate Git branch that ends in a merge request onto the main branch. Task branches are named as follows:

```
$issue-$summary
```

Where `$issue` is the issue number and `$summary` is a brief description of the task or issue. The summary should be brief and succinct; it only needs to jog the memory about the topic of development. The issue number can easily be used as a reference to get the full context of the task. Task branches are deleted upon merge.

Sudden Feature Branches

Sometimes during the course of work on a task it becomes clear that the work will not be completed in the typical Kanban time box of 1 - 3 days. The cause is often a combination of inadequate decomposition and poor estimation; another possible cause is discovery that the resulting changes would be disruptive to the main branch. In any case it happens and should not be a cause for alarm. In these cases the task branch should target a feature branch rather than main. The first step is to create the feature branch locally and push to the remote repository. If a merge request HAS NOT been prepared, simply note that the eventual merge request will target the feature branch rather than main and continue as normal. If a merge request HAS been created, then the merge request should be updated to target the feature branch rather than the main branch. Software development then proceeds as usual: the merge request is reviewed and eventually merged with the feature branch.

Build System

Every project that produces a software artifact should have a build system to ensure a reproducible build; for projects based on Java or Kotlin the build system MUST be Apache Maven. Software projects that define deployable services should also produce Docker images for deployment in a container technology. The current technology stack for deployment at the moment is AWS ECS/Fargate, so there should be Cloud Formation templates that describe runtime deployment infrastructure on AWS as well.

Build Toolchains and Coupling

Several projects produce multiple software artifacts via distinct toolchains; for example, a typical Web application is composed of the following:

1. Javascript build for browser-based front-end (Vite.js)
2. Java/Kotlin build for REST API back-end (Apache Maven)
3. Docker build to produce container image (Docker)

Each software artifact is produced by a technology-specific toolchain mentioned in parenthesis. In order to support the CI/CD process (following section), toolchains MUST NOT be tightly coupled in a way that *requires* multiple toolchains to be present in order to build software artifacts. (We elaborate on why in the following section.) For example, the Java build system MUST NOT be *required* to produce Docker images that require the docker binary/daemon on the build host. The Java build *may* produce Docker images for local development and testing as a matter of convenience to developers for projects where a Dockerized runtime is required, but it MUST NOT be the sole or canonical process. A seeming exception to this rule is our Web applications that are built with Node.js by a Maven plugin that can manage the process without requiring a local install of Node.js on the host executing the build. Thus it meets the requirement that exactly one build toolchain, JDK/Maven, is required to build both Java and Javascript software artifacts.

Often distinct build toolchains need to coordinate on common configuration in order to produce a set of related artifacts; version number is a common use case. In this case we recommend generating one file per property/configuration element in the first step of the build where they are needed and subsequently consuming each file as needed in subsequent build steps. The following stepwise process summarizes the recommended approach:

Step1: Write version.txt file containing version in build #1.
Step2: Read version.txt file containing version in build #2.
...
Step N: Read version.txt file containing version in build #N.

CI Pipeline

A GitLab CI pipeline, directed by a *.gitlab-ci.yml* file, is responsible for executing the build system to produce software artifacts, including execution of unit and integration tests, as well as deploying software artifacts to a target deployment environment, typically AWS ECS/Fargate. The mainline branch (*master*/*main*) is always the default target for builds; tags are used to identify versions that are slated for production deployments. The special *pprd* tag moves to a version that is staged for production deployment, while a protected *vX.Y.Z* branch is used to identify a version that will be deployed to production.

Runners

A runner is the unit of execution that executes a build process to produce software artifacts. While there are several types of runners that GitLab supports, Middleware has adopted Docker runners for a number of reasons:

- Improves repeatability of builds
- Facilitates declarative dependencies of toolchains
- Build toolchains can be provided by off-the-shelf images

Thus Docker runners provide both convenience and robustness with few down sides. The last requirement is notable: in order to leverage off-the-shelf images, build systems MUST NOT require multiple toolchains to produce artifacts.

The *docker-runner* GitLab runner is available to all Middleware projects and SHOULD be used whenever possible. Build processes that require access to the Docker daemon (e.g. `docker build`) MUST use a special runner, *docker-runner-socket*, that provides access to the Unix domain socket that is exposed by the daemon. While this is not onerous in practice, it is a requirement that bears mentioning.

Coding Conventions

The material aspects of software development simply cannot be prescribed; at best we can offer some general guidelines that tend to steer developers toward good code:

- Strive for the simplest solution that solves the problem
- Decompose difficult problems into manageable chunks

- Apply modularity to encapsulate concerns
- Avoid undue coupling between components/systems
- Apply proven software patterns (correctly) where possible
- Balance use of third-party libraries with avoiding dependency bloat
- Don’t repeat yourself

In the end good software is something that good developers know and can easily identify, which is why the code review process is essential to the Middleware software development workflow. It can’t be prescribed, but it can be identified and cultivated through peer review and refinement.

The incidental aspects of code such as formatting and naming conventions can easily be prescribed with tools such as *Checkstyle* (Java), *ktlint* (Kotlin), and *eslint* (Javascript). A standard ruleset is available for Java and Kotlin, while any reasonable rule set for *eslint* is acceptable. The build system should be configured to fix rule violations where possible and fail builds otherwise.

Testing

Every software development task should be accompanied by at least some meaningful unit test coverage. For components or subsystems with substantial external dependencies, mocks may be a practical way of implementing meaningful tests. On the other hand, sometimes the mechanics and/or behaviors are so tightly coupled to an external fixture that it would be better to provide tests that execute against a real test fixture. While the use of external fixtures in tests is often categorized as integration tests, Middleware often finds that component unit tests sometimes merit these fixtures. In any case the Docker Compose technology provides a convenient way to package code, test fixtures, and simple configuration in a text file that can be executed as either part of the build or as a one-off test procedure. Many projects contain *docker-compose.yml* files in the source tree for this purpose.

Documentation

Documentation should accompany code as needed to clarify the purpose, requirements, assumptions, and notable interactions of software components, modules, and systems. Even product technical documentation may be warranted if there are notable large-scale interactions or behaviors that merit explanation or summarizing. Documentation in code itself should be principally concerned with describing public interfaces (functions and components) and data types and should adhere to platform-specific best practices. In the case of Java source code, our most common platform, the Checkstyle rules enforce good habits; elsewhere, best practices should be sought and followed.

Design considerations and decisions (the *why* of things) should be externalized to the issue tracking system where the feature (or bug) has been requested in order to elicit feedback from the development team *before* a decision is made; then the decision should be documented on the issue and executed in code. Mentioning the issue number in the commit message for the feature ties the implementation to the rationale in a convenient fashion.

If diagrams are warranted for large scopes of work, then UML diagrams SHOULD be created using PlantUML and stored as text with the source under the *doc* directory in the project root. The diagrams MAY be rendered as PNG files and included in the *doc* directory or even embedded in the project README if they are perceived as generally helpful.