Source code management proposal

This document describes proposed processes for managing and developing PSI source code. A guiding principle is to keep it simple. Additional complexity may be needed, but we will avoid introducing complexity until the need is clear. In short, *start simply*.

Summary

- Host all source code in a single Git repository.
- Use a single mainline branch `main`.
- Developers create feature branches from `main`.
- Developers create unit test covering code that is affected by the change.
- Feature branches pulled into `main` daily with pull request.
- Code review required to approve pull request.
- Other branches can be created for activities (release, hotfix) but should be removed when the activity is complete.

Overview

We are striving for continuous integration (CI), high frequency commits to the mainline source control branch, and continuous delivery (CD), automatically building and testing on every commit. This document describes processes and gaps for how we might achieve CI/CD in the PSI project.

Continuous integration requires automatic building and automatic unit testing. Long-lived feature branches that are infrequently integrated into the mainline branch are contrary to CI. Pre-integration code reviews are a source of friction against frequent integration. As we mature and code covered by unit tests (code coverage) grows, we may choose not to require pre-integration code reviews. If we learn to commit only healthy code (and we can prove it) frequent commits to the mainline may be superior.

Continuous delivery requires automated system/integration testing and automated deployment. Release branches to fix bugs in the main branch are contrary to CD. As we prove the mainline branch is healthy with a comprehensive suite of automated full system tests, we may choose not to require release branches.

We are not ready for CI/CD, but we can start make steps in that direction.

Source code management

We will use the Git version control system. Everything required to build an environment must be included. Build artifacts based on the source code should not be included. Large binary files that change frequently should not be included.

To *start simply*, the only long-lived branch should be the mainline branch (`main`). The mainline branch should always be healthy; it should build cleanly with passing tests. Development takes place on a feature branch. Ideally feature branches should exist for only a single day. When the feature is ready, the developer creates a pull request to merge the feature branch into the mainline branch. The pull request will automatically check for merge conflicts, build*, and test the feature branch code. A code review will be required to approve the pull request. The merge to the mainline takes place upon approval. If the feature is complete, the feature branch is removed.

** Automatic build is currently a major gap. Until resolved, automated testing should be initiated on the development environment.*

If a feature branch is long-lived, it should still be integrated into the main branch each day. This requires the code to be in a condition to allow a commit to main. Consider adding features with an option to toggle them on or off. This feature flag would normally be removed when the feature development is fully complete.

Other branches may be needed for particular situations (like hotfix in production), but the branches should not be created until they are needed.

Avoid operations that change history of shared branches like rebasing or cherry-picking. There is no file locking in Git out of the box. The `git-lfs` extension does have a file locking feature in "early release". To *start simply*, we will not use file locking.

Integrate often

Frequent commits to the mainline minimize the scope of merge conflicts. One large merge conflict can take far more time and energy than several small ones.

Pulling from the mainline alone is not enough to avoid the problem. Consider Alice and Bob, working on their own branches.

```
Alice     o--a1--a2--x2-a3-x3-a4------
          |          |     ||
mainline ----m1-------------x3-----x4--
          |       |               ||
Bob       o---b1--x1--b2---b3--b4-x4---
```

Both Alice and Bob pull from the mainline. At x1 Bob only has to merge m1 and b1. At x2, Alice only has to merge m1 and a1-2. At x3, there's nothing new in the mainline, so Alice merges without any chance of conflict. However at x4, Bob must merge all of his changes with all of Alices' (a1-4 and b1-4)

Instead, if Alice and Bob merge their changes with the mainline more often each merge has a smaller scope with less opportunity for complex conflicts.

```
Alice     o--a1--x1--a2--x3----a3--
          |      ||      ||
mainline ----m1---x1--x2--x3--x4---
          |          ||       ||
Bob       o---b1-----x2----b2--x4--
```

As x1 Alice merges a1 and m1. At x2, Bob can see Alice's change a1; he merges a1, b1, and m1. At x3, Alice sees bob's change; she only needs to merge b1 and a2. At x4 Bob only needs to merge a2 and b2.

With frequent integration, the scope of potential merge conflicts and the time until they are discovered is kept small. If branches remain isolated from the mainline for a long period of time, there is a larger potential for bad merge conflicts requiring several painful hours to resolve.

Developers should integrate with the main branch each day.

Unit tests

Automated unit testing is key to developing reliable, maintainable software. Unit tests should be easy to run without side-effects and provide rapid feedback to the developer. Unit tests are not created post-development; they are part of the development process. They must be first-class citizens of the code base. Good unit tests do not take more time to develop; they save time by providing a constant safety net. They give the developer courage to make changes without the fear of breaking things. They are a first check that our change does what we expect, **and all the other code covered by tests still does what it should**.

Unit tests should be fast while providing quick feedback to the developer. They should not have side effects. Running the tests several times should not fill up files or tables.

Ideally, the majority of all code would be covered by unit tests. We propose starting with requiring unit tests for PL/SQL using the utPLSQL unit test framework.

- All *new* functions and procedures should be covered by tests.
- A test-first development style is strongly encouraged.
- Writing tests for existing functions and procedures before making a change is strongly encouraged.
  - If a test cannot feasibly be implemented, the reason should be clearly explained and documented.

Automated build (pending)

> **Major gap:** must have the capability to build and update the database from the source code.

A pull request to the main branch should automatically trigger a build and test run. The pull request must pass tests before it is approved. Broken builds must be fixed immediately. Often this may mean reverting the last merge.

The build must be fast (goal: less than 10 minutes).

Code review

In addition to serving as a check for compliance with guidelines, code reviews can be a good way to promote collaboration and minimize silos. All developers should participate in code reviews. Developers are encouraged to review pull requests from all areas, not just the areas they know best.

Code reviews take time. They are a source of friction against the important goal of continuous integration. But the value gained should justify the cost. Make code reviews a priority.

Propose requiring two reviewers, other than the developer who creates the pull request, to approve each pull request. As a general guideline, one reviewer should be a PSI team member and the other reviewer should be a Nucor team member.

Package management (pending)

Required libraries or executables that cannot be built from the source should be added to package management repository.

Scenarios

Normal development

A feature branch is created from the the `main` branch. Development takes place on the feature branch.

```
main     ----o------
          \
feature    o---o--o-
```

Frequently (at least once a day), the branches are synchronized. The developer pulls from the main branch to handle merge conflicts on the developer's machine.

```
                pull request
                     |
main     ----o--------o--
          \          /
feature    o---o--o-o
```

The pull request is reviewed and approved before merging with the `main` branch. If the feature is complete, the feature branch is removed.

Complex feature (long-lived branch)

Ideally, complex features would be broken down into smaller component features that can reasonably be added to the mainline in a single day.

If a feature cannot be broken down, consider techniques to prevent exposing it. If applicable, implement all back-end functionality first and save the UI which exposes it for the last step. For a change to existing behavior, consider preserving the old behavior while implementing the new activated through a toggle exposed only for testing (feature flag). When development is complete, the old behavior and feature flag may be removed.

Release to production

If we have confidence in the health of our main branch because it has been subject to extensive automated testing, release branches may not be required. Until we reach that point, a release branch can be created to isolate development for bug fixes from ongoing feature development.

```
            bugfix bugfix
              |      |
release   o---o------o-----
         /     \      \
main   -----o---o---o--o---
            |       |
        feature  feature
```

Bug fixes are implemented in the release branch and merged into the mainline branch. The release branch should be removed after release activities are complete. Consider tagging the last commit to make it easier to find in the future.

Over time, we should to remove the need for feature branches by ensuring a proven release-ready mainline.

Patch

Imagine a scenario where a fix needs to be added to multiple branches but it is not desired to merge the branch. Consider two branches foo and bar. A bug is discovered and fixed in foo.

```
foo ----f1-----fix--
     |
bar  o---b1--b2-----
```

Bar needs the fix but does not want f1. Cherry-pick could grab only the desired commit, but can introduce problems. The fix may not be valid without some of the changes in f1. Instead of implementing the fix on the foo branch, we could use a patch branch.

```
foo  ----f1------fix--
      |         /
patch o--fix----
      |         \
bar   o---b1--b2-fix--
```

The patch can be merged into each branch handling the conflicts independently.

Hotfix

Consider a scenario where a release branch is created. The final commit is tagged and the release branch removed after the release has been deployed to production. The tag is not required but may make it easier to locate in the future.

```
              t1
              |
release   o---o
          |    \
main    ----o---o-o----o--
```

Development continues in the main branch. At some point in the future a serious bug is discovered in production. Serious bugs may require a hotfix in production. We can checkout the tagged commit and create a new branch consistent with production.

```
        t1     t2
        |      |
hotfix  o------o
                \
main   --o--o----o--o--
```

The final hotfix commit is tagged and deployed to production. The hotfix branch is merged with main and deleted.