Git Workflow in Azure DevOps

This document describes a basic workflow for working with the `psi` repository on the NextGen project in Azure DevOps (ADO). It assumes the developer tools are installed and the repository has been cloned on the developer's workstation. Actions are described using a web browser in the ADO NextGen project and using Visual Studio Code (VSCode).

See the `git-quick-start` document for an introduction to git commands and concepts.

Create a branch (ADO)

To create a branch in the ADO NextGen project, navigate to `Repos > Branches > psi`. Click `New branch`. Give the branch an appropriate name. The branch should be based on the `main` branch. If desired, add a link to an existing ADO work item. Click `Create` to create the branch.

Branch name

Letters in branch names should be all lowercase. Feature branches should start with `feature/`. Bug fix branches should start with `bugfix/`. If the branch aligns with an existing work item, the branch name can include the work item number. A descriptive name is appropriate if a work item number is insufficient.

Good

```
feature/ngs-1234
feature/ado-987
feature/single-sign-on
bugfix/ngs-5678
```

Bad

```
Feature/ngs-1234  # captial letters
feature/NGS-1234  # captial letters
ngs-5678          # missing folder name
```

VSCode-Git integration

VSCode includes git features. Git commands can be invoked through the various graphical elements or the command palette. We describe how to find a single list on the Source Control tab.

In VSCode, click on the `Source Control` tab (shortcut `ctrl+shift+G`). For the repository, click on the `...` (three dot) button for a list of most common git commands.

VSCode can be configured to automatically fetch metadata periodically. The status bar indicates the active branch and status of pending changes on the branch.

Pull the branch (VSCode)

To pull the branch from the ADO server (remote) to the developer workstation (local), first fetch the metadata: `... > Fetch`. Checkout the branch: `... > Checkout to...` By default, the remote is named `origin`; the remote branch will start with origin. Example: `origin/feature/ngs-1234`. Click on the remote branch to be checked out. It will be pulled locally. Check the status bar to confirm the active branch.

Update and commit (VSCode)

Edit files to implement the desired behavior. Changes in the working directory are shown in the `Source Control` tab. Click the `+` button for each file that you would like to stage. Click the `+` button on the `Changes` banner to stage all changes.

A commit message is required. The commit message should provide context for the change. The diff tools will show us what changed; the commit message should tell us *why*.

If a short commit message is desired, use the text box above the commit button and click `Commit`.

If a more detailed commit message is desired, leave the text box blank. An new text editor tab will be opened to allow a more detailed commit message. The first line should be the subject of less than 50 characters. Skip a line before starting the body. Lines in the body should be less than 72 characters. Add helpful information so readers can understand the change.

Example:

```
The subject should be less than 50 characters

This is the body. The body can provide more detailed information
if required. Keep line length less than 72 characters. Add helpful
context.

Changes to be committed:
    modified:   important_package.sql

Resolves: NGS-5678
```

Push to remote

The changes have been committed to the local feature branch on the developers workstation. To push the changes to the feature branch on the remote server, click the `Sync Changes` button.

Integrate with `main` (ADO)

Pull request

Create a pull request in the ADO NextGen project to integrate the feature branch with the `main` branch. Use the `Create a pull request` helper button or navigate to `Repos > Pull requests` and click `New pull request`.

Confirm/configure the pull request to pull changes from your feature branch into `main`. The title and description will be populated based on the commit message. Change it if desired. Add optional or required reviewers as desired. Click `Create` to create the pull request.

It should not normally be necessary to specify reviewers. Two reviewers are required to approve the pull request.

Auto-complete ADO provides the option to automatically complete the pull request and merge when all policies are met. This is a feature to help expedite integration but should not become a fire-and-forget button. Consider using auto-complete when: - The change is trivial and clearly safe to merge. - After some initial feedback and you have high confindence.

Code review

The pull request also documents the code review.

In addition to serving as a check for compliance with guidelines, code reviews can 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. See the code-guidelines and code-review-guidelines documents for more information.

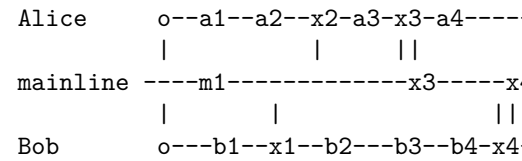
Two reviewers are required to approve a pull request. It is encouraged to have one reviewer from PSI and one reviewer from Nucor.

Once approved, the pull request must be completed if not set to auto complete. Generally the pull request author should complete the pull request. A reviewer may complete the pull request if necessary.

Integrate often

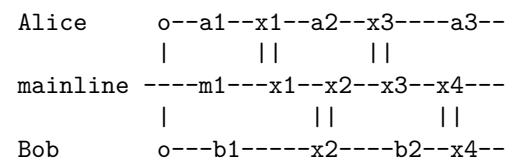
Frequent commits to the mainline minimize the scope of merge conflicts. One large merge conflict can take far more time and energy than several small ones.

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



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.



At 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.

To pull from the main branch using VSCode, go to `... > Branch > Merge branch...` and select the remote mainline branch `origin/main`. Push to the mainline branch using a pull request in ADO.

Branch duration

Ideally feature branches should exist for only a single day. 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 strategies to minimize exposing an incomplete feature.

> Developers using Continuous Integration need to get used to the idea of reaching frequent integration points with a partially built feature. They need to consider how to do this without exposing a partially built feature in the running system. Often this is easy: if I'm implementing a discount algorithm that relies on a coupon

> code, and that code isn't in the valid list yet, then my code isn't going to get called even if it is production. Similarly if I'm adding a feature that asks an insurance claimant if they are a smoker, I can build and test the logic behind the code and ensure it isn't used in production by leaving the UI that asks the question until the last day of building the feature. Hiding a partially built feature by hooking up a Keystone Interface last is often an effective technique.
>
> If there's no way to easily hide the partial feature, we can use feature flags. As well as hiding a partially built feature, such flags also allow the feature to be selectively revealed to a subset of users - often handy for a slow roll-out of a new feature.
>
> Integrating part-built features particularly concerns those who worry about having buggy code in mainline. Consequently, those who use Continuous Integration also need Self Testing Code, so that there's confidence that having partially built features in mainline doesn't increase the chance of bugs. With this approach, developers write tests for the partially built features as they are writing that feature code and commit both feature code and tests into mainline together (perhaps using Test Driven Development).
>
> Martin Fowler

If a feature branch lives longer than a day, consider merging into the feature branch from the mainline development branch, daily.

Examples

We can see that some branches are several commits behind the main branch. This isn't necessarily a problem as long as there are no merge conflicts the merge will be just fine, but the longer they stay out of sync the more potential for conflicts and difficult resolutions.

To mitigate this risk, it is recommended to merge from the remote main branch (origin/main) into your local feature branch, periodically (probably daily). You can do this in one step with:

```
git merge origin/main
```

Or in vscode source control menu > Branch > Merge Branch...

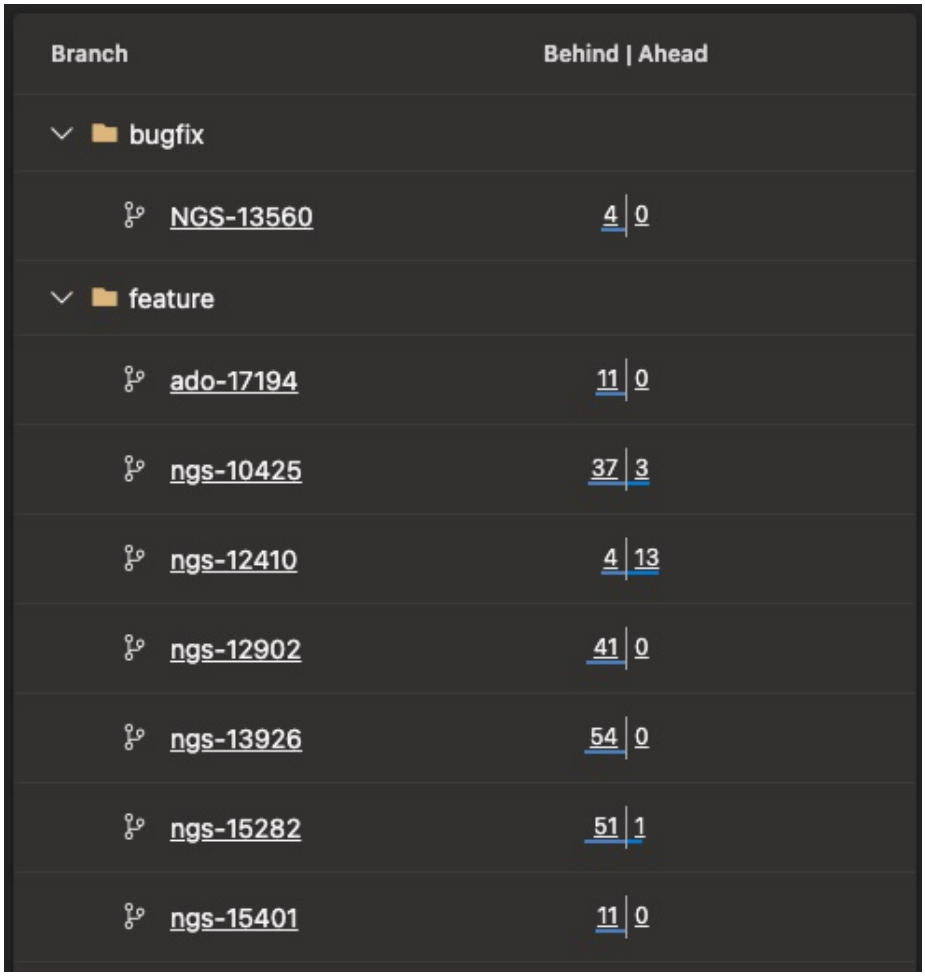


Figure 1: branches_behind_ahead.jpg

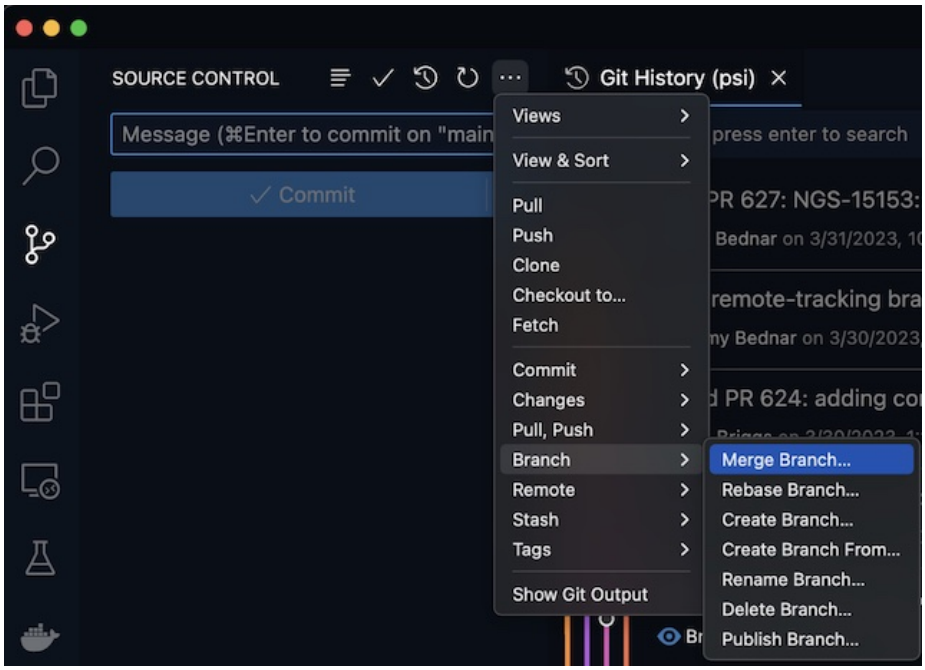


Figure 2: vscode_merge_from.png