Previous Branching Strategy

12/29/2023 1:39 pm EST

A basic understanding of distributed source control is assumed throughout this document. Concepts like committing, branching, merging, pushing, pulling, fetching, pull requests, tagging, and upstream/downstream repositories is required.

The need to standardize source control policy and procedure is driven by several key factors:

- **Reliability** - In this case, reliability means more than just producing a dependable software solution; it means
- **Consistency** - Description
- **Repeatability** - Description
- **Durability** - Description
- **Accountability** - Description

Overview

SDLC and Process Control Using Git

> Important! Please consult Branching Strategy for the most current process with respect to Branching

Along with work item management, source control patterns help shape the way software is written, tested, and delivered. This particular document covers a simplistic baseline approach to managing software in the development pipeline. Discreet branches are used for each major phase in the lifespan of a `feature`.

The following three branches are persistent throughout the lifespan of the entire application. Each of these branches has it's own governance and processes regulating the commits that can be applied to them. The branch names are identical from one repository to another.

- **Master *(Trunk)*** - `master` - The `master` branch represents fully tested and deployed code. The only time code moves into the master branch is immediately preceding a deployment of `master` `HEAD`. Additionally, new branches for features or bugfixes are *always* started from `master`. This ensures that each `feature` branch will be self-contained and deployable without relying on other potentially unstable branches.
- **Stage *(Release Candidate)*** - `stage` - The `stage` branch houses fully tested code that is ready to be (but not yet) deployed. Upon successful completion of testing, QA can push code from the `test` branch into `stage`. `stage` acts as holding tank for work that is production-ready but has not yet been deployed. Multiple features and bugfixes can coincide in stage when they are complete before moving to production. Just as developers use `master` as the starting point for new development, QA testers use `stage` as a starting point to ensure that no conflicts arise between independently tested features. Cherry-picking merge commits from `stage` to merge into `master` should be discouraged. Once code has been tested together as a cohesive unit, it should remain as such all the way through the lifecycle. Separating it after integration tests are complete has the potential to introduce

bugs that would have been impossible to catch during the test phase.

- **Test *(Quality Assurance)*** - `test` - Once a developer believes that they have completed an assignment, they can create a pull request to merge their code into `test`. QA regularly uses the `test` branch for validating and verifying work before they sign off on it's readiness. Prior to merging code from a `feature` branch into `test`, the code must undergo peer review.

Additional 'unregulated' branches offer the flexibility and power of git as a source control system without potential pitfalls impacting production code. These branches have standard naming conventions rather than standard names. These branches contain limited scopes defined by work items; therefore they should be considered transient. Typically, they are created by branching from master and deleted when a pull request has been accepted into stage.

- **Feature *(Deliverable)*** - `ft-some-description` - A `feature` may vary widely in scope but can ultimately be boiled down to a *self-contained deliverable unit of code*. This means that regardless of the status/progress of other ongoing efforts, a product manager can elect to deploy this feature to production. A feature branch should correspond to one or more work items in TFS.

> Undecided: Naming convention, gated commits to feature branches

Branching, Merging, and Forking

An effective resource for understanding branching and merging can be found here.

What is a branch & how is it different from TFSVC/SVN

- Centralized vs Decentralized Source Control
- Upstream repositories/branches
- Pull request as a branch
- Branch and release
- Protected branches

What is a Merge?

- Merging vs Rebasing

When should I create a branch?

Collaborating with others

- When is pushing ok?
- Pushing work branches
- Checklist for pushes
    - Does it build

    - Are there new (undocumented) dependencies
    - Does this require changes to the database

What is a Fork?

- Why would I ever fork?
- Can I keep a fork up to date with the main repo?

Pull Requests

Pull requests function as logical change groups that indicate to another person or people that code is ready for review. Pull requests at various stages may require a series of checks before they can be accepted.

Different gates that may be evaluated against a pull request include:

- Peer Review (Code review)
- QA Tests
- Automated Unit Tests
- Database review
- Changelog updates
- Build server evaluation

Pull requests can be created between any two branches and can be an effective tool for submitting code to development leads prior to being merged into a shared branch. They are required to promote code from a feature to test, from test to stage, and from stage to master.

Common Scenarios

The following sections are intended as a walkthrough guide to common situations within the workflow.

Starting a New Feature

A feature is defined as an independent deliverable unit of work which can traverse the entire SDLC without any unreleased dependencies.
The last bit is important in that it ensures that testing and deployment of the code can be verified/performed without the
added complexity of dependencies that are not production ready. This means that the only two branches that new features should
be created from are master and stage.

Feature branches are intended to be used for collaboration between developers as well as between developers and testers. All
feature branches should be pushed to the upstream repository on a regular basis.

Notes:

- Naming convention for feature branches - TBD
- Clone from master when starting fresh or stage to extend tested features that are ready for release

Example Git Commands:

```
git branch some-new-feature  # create the branch
git checkout some-new-feature  # switch to the new branch
# alternatively
git checkout -b some-new-feature # create and switch to a new branch

# ...do some work...
git add . -A  # add any new files to your commit
```

Preparing a Feature for Test

For developers to release code, it must be both tested and reviewed by peers. The mechanism for marking code as ready for testing and review is by creating a pull request. Pull requests can be created using the UI in Visual Studio or by accessing the DevOps site for the project and creating a new one. A pull request is at it's core a simple branch. It allows
the developer communicate that work is ready and ensures that the contents of the test branch are under the strict control of
the QA team. Contributors should not commit directly to test, stage or master.

- Submit a pull request to test
- Pull request is reviewed by peers & QA and rejected if there are suspected issues or problems in isolation
- When pull request is accepted, QA does further integration testing with other features from stage (if applicable)
- If integration testing fails, QA will remove the merge commit from the accepted pull request so that the feature is completely rolled back from the test branch

Testing a Feature

- First test the PR branch so the feature can be evaluated in isolation
- Reject the PR if there are any issues, do not amend commits
- Test a local merged copy of the PR with stage before accepting the PR
- Finally, test the build produced by CI once the PR has been accepted
- The feature can be rejected & rolled back at any point prior to the feature being merged to stage

Switching to a Different Feature

- Make sure that your current changes are committed (at least locally)

Versioning

https://semver.org/

Release

Releases can be generated at any point in time or based on a schedule (depending on business needs). Once a release has been scheduled, the contents of the `stage` branch are combined in a pull request to master. When the pull request has been accepted, `master` is tagged with the appropriate version number and the release is deployed. Tagging is crucial so that a relationship can be formed between work items, test cases, commits, and artifacts (binaries). Furthermore, in the event of a critical production bug-fix, a pull request can be created against the tagged release. This will ensure that the bug-fix is applied to the affected release as well as future releases.

Tags

What are Tags?

A tag in git is a named snapshot for a particular commit. It gives a meaningful name to a particular state of the code within the repository. Naturally, it is primarily used to mark releases and release candidates within a particular repository.

Since git works on the commit level, rather than on the virtual hierarchy presented within (e.g., an unchanged branch of `master` has the same state/head commit as its predecessor), a tag is affixed to a commit, and not to a particular branch. In the parenthetical example just given, a tag created from `master` or a branch that had just been created from `master` would be the same.

Functionally, as `master` will contain only code that is in a releasable state, our tags will be generated from the commit from `master`. In more complex release scenarios (e.g., hotfixes to legacy releases), the tags will be created from the relevant release branch for that scenario.

Why do we Need Tags?

Tags give us several key benefits.

First, if we tag our releases, we can quickly and trivially pull the code at the state it was for that particular release. The biggest benefit of this is we can quickly create and release bug fixes that are specific to that version. So, if a bug was introduced in a previous release, and we have several supported releases, we can fix the bug in the earliest supported release, send that change through the release pipeline, and then propagate the same change to the other supported versions, all while keeping it entirely clear that this is the only change that is being put into these different hotfix releases.

In addition to that, we also have better accounting of the system. By leveraging the data that's stored without our repository (and our DevOps system), we can trivially generate a list of all changes that have been introduced since the

last release. These changes are associated with pull requests, and those pull requests are associated with work items. Therefore, by tagging (and using our DevOps system effectively), we can easily and accurately give an account of every change that was entered into the system for a particular release.

One more benefit is that it paints a picture of how effective our releases are. By using Semantic Versioning (link above), we can gather valuable metrics. We can see how many releases we typically make in a period of time. We can see the latest release versions for all of our supported releases. We can also see how many "patch" releases we generally have per a release. We can then use that data to drive future research into potential benefits downfalls in our development/release process, optimizing our process for efficiency, waste mitigation, and company and customer success.