Branching Strategy

12/29/2023 1:28 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

We will be using a form of the GitHub Flow.

Visit this video by Microsoft explaining this flow in more detail (as well as some other good branching recommendations).

The basic idea that we're going to adopt is that all work happens in a branch, and it eventually gets merged into master.

But first, let's understand the branches:

- master - the main code line. Generally speaking, all development starts and ends here.
- feature/ - the directory where features are contained.
- pbi/ - the directory where Product Backlog Item/User Stories code is developed.
- bug/ - the directory where Bug code is developed.
- release/ - the directory where the Releases are stored for production release.

Branch names in feature/, pbi/ and bug/ will have the following pattern:

<work-item-type>/<id>-<short-description>1

1 When I use angle brackets in a Git branch, I mean for them to be interpreted as placeholders for actual values. You should not actually use angle brackets in branch names.

Where:

- <work-item-type> is the Work Item type (i.e., feature, pbi, or bug)
- <id> is the Work Item Id.
- <short-description> is a short description of the Work Item.

All of these should be lower-case and should join words with dashes and NOT underscores.

For example, for Bug 1234 - Fix the Display on the Customer Screen, you would have a branch named:

bug/1234-fix-customer-display

Branch names in release will have the following pattern:

release/<year>Q<x>S<y>-<major><minor>u<patch>

Attention: We apply a tag to the Master branch prior to creating any release branches.

Where:

- <year> is the year of the sprint (e.g., 2019)
- <x> is the Quarter (e.g., 2)
- <y> is the Sprint (e.g., 3)
- <major> is the targetted Major Version (e.g., 2)
- <minor> is the targetted Minor Version (e.g., 1)
- <patch> is the targetted Update Number (e.g., 6)

E.g., release/2019Q2S3-21u6

Alternately, if you know your target release, you may use the following convention instead:

release/<major>.<minor>

Where:

- <major> is the major release number (e.g., 4)
- <minor> is the minor release number (e.g., 6)

e.g., release/4.6

If it makes sense for your versioning, you may also add the patch level.

e.g., release/4.6.1

The Nominal Flow

1. Developer gets assigned a Feature (e.g., 1235 - Add Widgets)

2. Developer branches master into feature/1235-add-widget

3. Developer gets assigned a User Story2 (e.g., 1236 - Customer Graph Widget)

4. Developer branches feature/1235-add-widget to pbi/1236-customer-graph-widget

5. Developer implements User Story.

6. Developer issues pull request to feature branch.
    a. Reviewer examines code changes and provides feedback as necessary, otherwise, approves.
    b. QA determines if acceptance criteria for User Story is met and provides feedback as necessary, otherwise, approves.
    c. Feature owner completes pull request.
        i. If there are more User Stories or Bugs, these should merge the feature branch into their branches before continuing.
        ii. Developers will repeat this flow as necessary to complete all children of the feature.

7. When the feature is complete, the Developer will issue a pull request to master.
    a. Reviewer examines code changes and provides feedback as necessary, otherwise, approves.
    b. QA determines if acceptance criteria for Feature is met and provides feedback as necessary, otherwise, approves.
    c. Product Owner completes the pull request.
        i. Any open features should at this time merge from master, and the pbi/ and bug/ branches should merge from the feature branch.

8. Before release to production, Product Owner creates a release branch from master.

9. QA performs regression testing on this branch. This release branch is locked to only critical issues.

10. Details: QA must complete the following:
    - Have fingers crossed
    - Throw salt over shoulder
    - Say a little prayer
    - Roll eyes
    - Jump backward 3 times and do the hokey pokey.... twice....
    - Know that these particular QA steps are only here to ensure you are still reading.

11. Build Manager releases branch to production and tags release.

2 The flow for a bug is the same.

Sub-Nominal Situations

For all of these situations, the best ally is direct communication between the affected parties.

Developer Discovers Issue With Code Review

If a reviewing developer discovers an issue with the code review, s/he flags the pull request as rejected and provides a comment regarding the issues discovered (DevOps lets you comment on individual lines of code). Likewise, reviewing developer moves Work Item from "done" to "doing" column under Development.

The original developer will review the suggestions and fix any legitimate issues. When the work is done, s/he will update the original branch (e.g., pbi/1236-customer-graph-widget) and push the changes to DevOps. This will add a new update to the Pull Request. The original developer will also mark all comments with an appropriate resolution code (i.e., "Resolved" or "Won't Fix", the latter preferably with a comment regarding why it won't get fixed). Likewise, the original developer moves Work Item from "doing" to "done" column under Development.

Assuming there are no other issues, the reviewing developer will then approve the pull request (if there are remaining issues that are non-critical that the original developer marked as "Won't Fix", the reviewing developer may opt to "Approve with Suggestions," depending on how important these issues are.)

QA Discovers Issue With User Story/Bug

If a QA Engineer discovers a problem that prevents acceptance of a User Story/Bug, then s/he will reject the Pull Request, leave a comment with as much information as necessary to convey the issue to the developer, and move the Work Item from QA "doing" to Development "doing".

If the issue is with configuration, consider whether there is a software solution for ensuring that this problem is not being encountered. If the problem is a process issue (e.g., test system needs to have a script run against it), ensure that this is documented somewhere so it gets done in production as well.

The original developer will fix the issue (and test it to make sure it works correctly). It is on the developer to

ensure that the acceptance criteria is met prior to sending to QA. Once it is fixed, push the changes as one would with a code review issue. Move the card from Development "doing" to Development "done." This pull request will require a new code review (but the reviewing developer will most likely only need to review the Updates since the last review). QA will then move the Work Item from Development "done" to QA "doing" and start the review process anew.

Assuming there are no other issues, QA will flag the review as Approved and move the work item to QA "done".

Merge Problem with Feature

If while updating your branch with the feature branch, you encounter a merge issue, proceed with caution!

It should go without saying: before starting a merge, make sure that there are no uncommitted changes in your development environment.

If the merge is straightforward, carefully evaluate every merge (even the automatic merges) to ensure that nothing gets inadvertently broken. Take note of the modules affected by the merge, because you'll want to take a look at them to see that they still work as expected.

If it's a particularly gnarly merge, get the other developer involved. The goal for the merge is to keep the existing code unbroken while minimizing the breaking of your own code. After the merge, the code should build and run. If it does not, then fix these issue before moving on.

Be sure to test those modules you took note of. When you're satisfied that everything works correctly, have the other developer double check.

Also, when it's time to do a pull-request, make sure to assign the other developer to the pull request so s/he has some say over the results of the merge.

Feature is Incomplete

If a feature is incomplete (it's missing something or some functionality is broken), this most likely means that a User Story or Bug was not created. QA may note this sort of issue while evaluating a feature pull request to master.

QA will reject the pull request with a comment and move the Feature from QA "doing" to Development. Developer, Scrum Master, and Product Owner will work together to allocate additional Work Items to feature to meet the QA acceptance. Developer(s) will work on these issues until they are complete.

Release Branch has Bug (or Missing Critical Feature)

If a bug (or missing critical feature) is discovered during the regression testing or exploratory testing a a release branch (prior to it being released), then it will have to be fixed. This fix will be one of the more important work items for the Developer to whom it is assigned.

It it does not already exist, a new Feature will be created for the release. Any new bugs or critical features to get this release ready for production will be put under here. They will follow the normal flow as prescribed above;

however, the will have an additional step after the merge to master.

On the completed Pull Request to master, use the Cherry Pick command to send those changes to the release branch as well.

A Bug Is Found in Production, And It's a Big One!

If there is a bug in production, and it can't wait for the next release window, it needs to be fix as soon as possible.

A feature and a bug should be created to track it. From a code perspective, the branches should not start from master, but rather, from the tagged release (or, possibly, the release branch, since they should represent the same state). This way a HotFix can be created for that specific version to resolve the issue for the customer immediately.

When the work is finished, follow the standard flow (and the addenda prescribed in the "Release Branch has a Bug (or Missing Critical Feature)" section. The release branch is now ready to release once more.

Work Flow for Branching In DevOps

The branching strategy we use is GitHub Flow and is supported in Azure DevOps. While it is possible to create branches from Visual Studio or the command line, it is recommended to use the DevOps web portal because it automatically builds all the links to the work items for you. Additionally, even in the web portal, there are several places to do this. The cleanest method is to create the branch from the work item itself.

- Features: The branch should be created from Master by the feature owner. Naming is feature/#-Title
- User Story: This should be created from the Feature branch by the developer doing the work. Naming is pbi/#-Title (IF only one developer is doing the work and it can be completed in one sprint, then the work can be done directly in the Feature branch. That branch should be added as a link to the User Story for tracking) All User Stories should be under a Feature because QA will be testing completed Features, not individual User Stories.
- Bug: This should be created from Master by the developer. Naming is bug/#-Title

In Visual Studio (or command line) the first step is to do a Sync or Fetch from the server to get knowledge of that new branch. Then you can checkout and switch to the branch in your local repository to begin work. The local changes must be pushed to the server periodically.