

Git Feature Branch Workflow

The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the master branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means

the master branch will never contain broken code, which is a huge advantage for continuous integration environments.

Encapsulating feature development also makes it possible to leverage pull requests, which are a way to initiate discussions around a branch. They give other developers the opportunity to sign off on a feature before it gets integrated into the official project. Or, if you get stuck in the middle of a feature, you can open a pull request asking for suggestions from your colleagues. The point is, pull requests make it

incredibly easy for your team to comment on each other's work.

The Git Feature Branch Workflow is a composable workflow that can be leveraged by other high-level Git workflows. We discussed other Git workflows on [the Git workflow overview page](#). Git Feature Branch Workflow is branching model focused, meaning that it is a guiding framework for managing and creating branches. Other workflows are more repo focused. The Git Feature Branch Workflow can be incorporated into other workflows. The [Gitflow](#), and [Git](#)

Forking Workflows traditionally use a Git Feature Branch Workflow in regards to their branching models.

How it works

The Feature Branch Workflow assumes a central repository, and `master` represents the official project history. Instead of committing directly on their local `master` branch, developers create a new branch every time they start work on a new feature. Feature branches should have descriptive names, like `animated-`

menu-items or issue-
#1061. The idea is to give
a clear, highly-focused
purpose to each branch.
Git makes no technical
distinction between the
master branch and
feature branches, so
developers can edit,
stage, and commit
changes to a feature
branch.

In addition, feature
branches can (and
should) be pushed to the
central repository. This
makes it possible to
share a feature with other
developers without
touching any official
code. Since master is the
only “special” branch,
storing several feature

branches on the central repository doesn't pose any problems. Of course, this is also a convenient way to back up everybody's local commits. The following is a walk-through of the life-cycle of a feature branch.

Start with the master branch

All feature branches are created off the latest code state of a project. This guide assumes this is maintained and updated in the `master` branch.

```
git checkout master
git fetch origin
git reset --hard o
```

This switches the repo to the master branch, pulls the latest commits and resets the repo's local copy of master to match the latest version.

Create a new-branch

Use a separate branch for each feature or issue you work on. After creating a branch, check it out locally so that any changes you make will be on that branch.

A screenshot of a terminal window with a dark blue background. The text 'git checkout -b new-feature' is displayed in a light green monospace font. Below the text is a horizontal scrollbar with a light gray track and a white slider, indicating the text is scrollable.

```
git checkout -b new-feature
```

This checks out a branch called new-feature based

on master, and the -b flag tells Git to create the branch if it doesn't already exist.

Update, add, commit, and push changes

On this branch, edit, stage, and commit changes in the usual fashion, building up the feature with as many commits as necessary. Work on the feature and make commits like you would any time you use Git. When ready, push your commits, updating the feature branch on Bitbucket.


```
git status  
git add <some-file>  
git commit
```

Push feature branch to remote

It's a good idea to push the feature branch up to the central repository.

This serves as a convenient backup, when collaborating with other developers, this would give them access to view commits to the new branch.

```
git push -u origin
```

This command pushes new-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, `git push` can be invoked without any parameters to automatically push the new-feature branch to the central repository. To get feedback on the new feature branch, create a pull request in a repository management solution like [Bitbucket Cloud](#) or [Bitbucket Server](#). From there, you can add reviewers and make sure everything is good to go before merging.

Resolve feedback

Now teammates comment and approve the pushed commits. Resolve their comments locally, commit, and push the suggested changes to Bitbucket. Your updates appear in the pull request.

Merge your pull request

Before you merge, you may have to resolve merge conflicts if others have made changes to the repo. When your pull request is approved and conflict-free, you can add your code to the master

branch. Merge from the pull request in Bitbucket.

Pull requests

Aside from isolating feature development, branches make it possible to discuss changes via pull requests. Once someone completes a feature, they don't immediately merge it into `master`. Instead, they push the feature branch to the central server and file a pull request asking to merge their additions into `master`. This gives other developers an opportunity to review the

changes before they become a part of the main codebase.

Code review is a major benefit of pull requests, but they're actually designed to be a generic way to talk about code. You can think of pull requests as a discussion dedicated to a particular branch. This means that they can also be used much earlier in the development process. For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they'll be able to see the

question right next to the relevant commits.

Once a pull request is accepted, the actual act of publishing a feature is much the same as in the [Centralized Workflow](#).

First, you need to make sure your local master is synchronized with the upstream master. Then, you merge the feature branch into master and push the updated master back to the central repository.

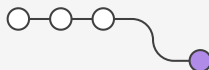
Pull requests can be facilitated by product repository management solutions like Bitbucket Cloud or Bitbucket Server. View the Bitbucket Server pull

requests documentation
for an example.


Example

The following is an example of the type of scenario in which a feature branching workflow is used. The scenario is that of a team doing code review around on a new feature pull request. This is one example of the many purposes this model can be used for.

Mary begins a new
feature



Before she starts developing a feature, Mary needs an isolated branch to work on. She can request a new branch with the following command:

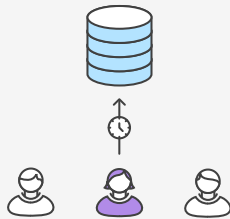
A screenshot of a terminal window with a dark blue background. The text 'git checkout -b marys-feature' is displayed in a monospaced font, with 'git' in orange, 'checkout' in white, '-b' in green, and 'marys-feature' in green. Below the text is a light gray horizontal bar with a small left-pointing triangle on the left and a small right-pointing triangle on the right, indicating a scrollbar.

```
git checkout -b marys-feature
```

This checks out a branch called `marys-feature` based on `master`, and the `-b` flag tells Git to create the branch if it doesn't already exist. On this branch, Mary edits, stages, and commits changes in the usual fashion, building up her feature with as many commits as necessary:


```
git status  
git add <some-file>  
git commit
```

Mary goes to lunch

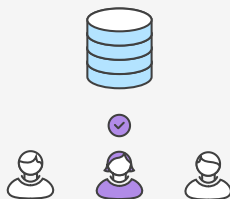


Mary adds a few commits to her feature over the course of the morning. Before she leaves for lunch, it's a good idea to push her feature branch up to the central repository. This serves as a convenient backup, but if Mary was collaborating with other developers, this would also give them access to her initial commits.

```
git push -u origin
```

This command pushes `marys-feature` to the central repository (`origin`), and the `-u` flag adds it as a remote tracking branch. After setting up the tracking branch, Mary can call `git push` without any parameters to push her feature.

Mary finishes her feature



When Mary gets back from lunch, she

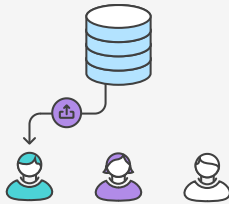
completes her feature. Before merging it into master, she needs to file a pull request letting the rest of the team know she's done. But first, she should make sure the central repository has her most recent commits:

```
git push
```

Then, she files the pull request in her Git GUI asking to merge marys-feature into master, and team members will be notified automatically. The great thing about pull requests is that they show comments right next to their related commits, so it's easy to ask questions

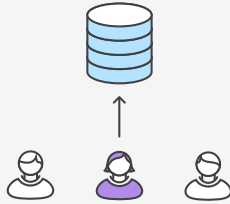
about specific
changesets.

Bill receives the pull request



Bill gets the pull request
and takes a look at
marys-feature. He
decides he wants to
make a few changes
before integrating it into
the official project, and
he and Mary have some
back-and-forth via the
pull request.

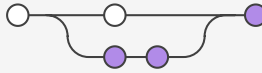
Mary makes the changes



To make the changes, Mary uses the exact same process as she did to create the first iteration of her feature. She edits, stages, commits, and pushes updates to the central repository. All her activity shows up in the pull request, and Bill can still make comments along the way.

If he wanted, Bill could pull `marys-feature` into his local repository and work on it on his own. Any commits he added would also show up in the pull request.

Mary publishes her feature



Once Bill is ready to accept the pull request, someone needs to merge the feature into the stable project (this can be done by either Bill or Mary):

```
git checkout master
git pull
git pull origin ma
git push
```

This process often results in a merge commit. Some developers like this because it's like a symbolic joining of the feature with the rest of the code base. But, if

you're partial to a linear history, it's possible to rebase the feature onto the tip of `master` before executing the merge, resulting in a fast-forward merge.

Some GUI's will automate the pull request acceptance process by running all of these commands just by clicking an "Accept" button. If yours doesn't, it should at least be able to automatically close the pull request when the feature branch gets merged into `master`.

Meanwhile, John is doing the exact same thing

While Mary and Bill are working on marys-feature and discussing it in her pull request, John is doing the exact same thing with his own feature branch. By isolating features into separate branches, everybody can work independently, yet it's still trivial to share changes with other developers when necessary.

Summary

In this document, we discussed the Git Feature Branch Workflow. This workflow helps organize

and track branches that are focused on business domain feature sets. Other Git workflows like the Git Forking Workflow and the Gitflow Workflow are repo focused and can leverage the Git Feature Branch Workflow to manage their branching models. This document demonstrated a high-level code example and fictional example for implementing the Git Feature Branch Workflow. Some key associations to make with the Feature Branch Workflow are:

- focused on branching patterns
- can be leveraged by other repo oriented workflows

- promotes collaboration with team members through pull requests and merge reviews

Utilizing [git rebase](#) during the review and merge stages of a feature branch will create enforce a cohesive Git history of feature merges. A feature branching model is a great tool to promote collaboration within a team environment.

Go one click deeper into Git workflows by reading our comprehensive tutorial of the [Gitflow Workflow](#).