Git Guidelines and Best Practices

This page describes conventions and best practices applicable to the Fedora Git repository.

Two things you should never do in git:

- NEVER force a push
  If you find yourself in a situation where your changes can't be pushed upstream, something is wrong. Contact another Fedora developer for help tracking down the problem.

- NEVER rebase a branch that you pushed, or that you pulled from another person
  Rebasing published branches can lead to duplicate commits in the shared repository.

In general, the preferred workflow is:

- create a branch from master, check it out, do your work
- test and commit your changes
- optionally push your branch up to the remote repository (origin) OR
- optionally rebase your branch to master (if your changes are unpublished)
- checkout master, make sure it's up-to-date with upstream changes
- merge your branch into master
- test again (and again)
- push your local copy of master up to the remote repository master (origin/master)
- delete your branch (and remotely, too, if you published it)

Overview of the Git Lifecycle

Git allows a developer to copy a remote subversion repository to a local instance on their workstation, do all their work and commits in that local repository, then push the state of that repository back to a central facility (**github**).

Bearing in mind that you will always being doing your work and commits *locally*, a typical session looks like this:

`git clone git@github.com:fcrepo/fcrepo.git && cd fcrepo`
Get a copy of the central storage facility (the repository).

`git branch fcrepo-756`
Create a *local* branch called "fcrepo-756".

`git checkout fcrepo-756`
Create a *local* copy of the branch from *master* if it doesn't exist, make it your active working branch.

Now, start creating, editing files, testing. When you're ready to commit your changes:

`git add [file]`
This tells git that the file(s) should be added to the next commit. You'll need to do this on files you modify, also.

`git commit [file]`
Commit your changes locally.

Now, the magic:

`git push origin fcrepo-756`
This command pushes the current state of your local repository, including all commits, up to github. Your work becomes part of the history of the fcrepo-756 branch on github.

`git push` is the command that changes the state of the remote code branch. Nothing you do locally will have any affect outside your workstation until you `push` your changes.

`git pull` is the command that brings your current *local* branch up-to-date with the state of the *remote* branch on github. Use this command when you want to make sure your local branch is all caught up with changes `push`'ed to the remote branch.

Some useful terms

master: this is the main code branch, equivalent to *trunk* in Subversion. Branches are generally created off of **master**.

origin: the default remote repository that all your branches are `pull`'ed from and `push`'ed to. This is defined when you execute the initial `git clone` command.

unpublished vs. **published** branches: an **unpublished** branch is a branch that only exists on your local workstation, in your local repository. Nobody but you know that branch exists. A **published** branch is one that has been `push`'ed up to github, and is available for other developers to checkout and work on.

fast-forward: the process of bringing a branch up-to-date with another branch, by fast-forwarding the commits in one branch onto the other.

rebase: the process by which you cut off the changes made in your local branch, and graft them onto the end of another branch.

Line endings

All text files in must be normalized so that lines terminate in the unix style (LF). In the past, we have had a mixture of termination styles. Shortly after the migration to Git the master and maintenance branches were normalized to LF. Please do not commit files that terminate in CRLF!

Configuring git to enforce LF normalization

There are several way to enforce LF normalization. Each method carries some consequences, and the consequences & methods differ between versions of Git.

`autocrlf` property

Normalization rules for all text files can be addressed by the 'autocrlf' configuration property. There are two useful values for this property, depending on your platform

- autocrlf = input. Use on unix-like platforms. This will perform no conversion upon checkout, but will normalize any crlf files upon commit.
- autocrlf = true. Useful on Windows platforms. This will have the effect of converting all text files into dos-style (CRLF) in the working copy upon checkout. Upon commit, all files will be normalized to LF on their way into the repository, but remain in CRLF in the local working copy directory.

This property can be set globally for all local git repositories, or locally for a single git repository.

The `autocrlf` property can be set **globally** via the command line. For example:

```
git config --global core.autocrlf input
```

Executing this command is identical to editing your ~/.gitconfig file and adding:

```
[core]
        autocrlf=input
```

The `autocrlf` property can also be set **locally** for a given git repository, such as the local clone of the fcrepo. For example, from within the local working directory:

```
git config core.autocrlf input
```

Executing this command is identical to editing the .git/config file within the git working directory and adding:

```
[core]
        autocrlf=input
```

`.gitattributes` file

The presence of a committed `.gitattributes` file within the code can also be used to apply line-ending rules. This has the benefit of being part of the managed sources (and this part of a given branch, tag, etc), but is not understood by all versions of git. The fedora master branch has a .gitattributes file containing `* text=auto`, which instructs git to detect text files, and normalize to LF at each commit.

Git 1.7.1 and earlier

Earlier versions of git do not understand the necessary directives in `.gitattributes` file, so `autocrlf` is the only way to assure compliance with the LF normanlization. Thus

- Unix and mac users should set `autocrlf = input` either globally or locally
- Windows users should set `autocrlf = auto` either globally or locally.

These versions of git may apply/detect autocrlf settings to all files in the working copy immediately. Thus, if checking out older branches/tags/commits that still have crlf files in the repository, these files will be seen as automatically 'modified' when doing a 'git status'. This may have confusing side-effects. When working with older branches containing a mixture of line endings, you may want to either turn autocrlf off, or just go ahead and convert all files in the branch to LF.

Git 1.7.2+

These versions of git heed the `.gitattributes` directive, so it is not strictly necessary to set autocrlf, but it is recommended.

These versions of git will apply the autocrlf setting to **new** files - preventing the introduction of non-normalized crlf files into the repository, but ignoring existing crlf files.

Working with older branches

Text files were normalized to use LF in commit 5275b... Any existing branches/tags that are not normalized may contain a mix of files. Merging changes from a crlf file into a normalized branch may be problematic. In particular, merging any modified crlf file into its normalized counterpart will produce a conflict with a whole-file diff (i.e. it will appear as the entire file is in conflict). Converting any such files to use LF endings in the originating (old) branch is a reasonable solution, and will result in merges that behave as expected.

Commit Messages

Commit messages should follow the guidelines described in detail at http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html.

In summary:

- First line: JIRA issue ID in all caps (if applicable), followed by a brief description (~ 50 characters)
- Second line: blank
- Following lines: more detailed description, line-wrapped at 72 characters. May contain multiple paragraphs, separated by blank lines. Link to the JIRA issue, if applicable.

Use the present tense when writing messages, i.e. "Fix bug, apply patch", not "Fixed bug, applied patch."

Two sample commit messages

- linked to a JIRA issue:

```
FCREPO-780: NPE thrown on disseminations

Fix for the following bug: Fedora throws a null pointer exception if
you call a disseminator that fronts a web service whose response does
not contain a "Content-type" header.

https://jira.duraspace.org/browse/FCREPO-780
```

- general issue:

```
Create .gitattributes file to normalize line feeds

Create .gitattributes file requesting all text files normalised to LF.
Will be ignored by git versions < 1.7.2

See https://wiki.duraspace.org/display/FCREPO/Git+Guidelines+and+Best+Practices
for more information.
```

Pulling and pushing to master

All `pull` or `merge` operations from remote/master into the local master branch should be fast-forward. Do not perform development in the master branch, periodically update with pull, and then push your local master. Instead, perform local commits in a separate branch, and merge (or rebase and merge) with master right before pushing it.

`git pull -ff-only` can be used to assure that a pull is fast-forward only. If a fast-forward pull is not possible, this flag will cause git to exit with an error, and leave the local branch untouched.

Development directly in master

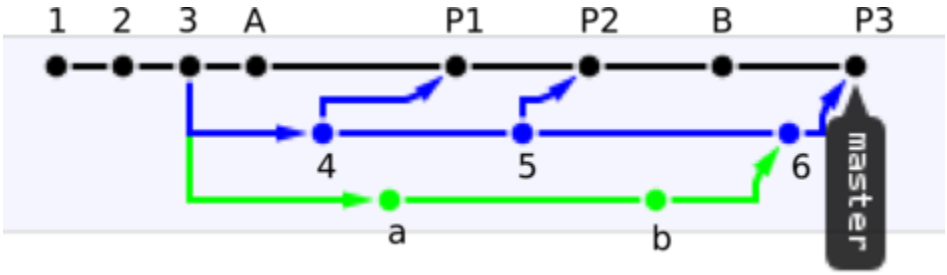
This should be avoided for all but the simplest commits that are immediately pushed. If you have several un-pushed commits, and then use `git pull` to merge in remote changes, that pull will be non-fast-forward. In other words, git pull will automatically create a merge commit which merges origin/master into your local branch. A subsequent `push` will publish your local master to the central repository, and the presence of the merge commit with origin/master might make a confusing-looking history. In fact, github 'network view' of github will make it appear that commits that were merged in with `git pull` came from another branch!

As an example, suppose there are three active developers working simultaneously - Tom, Dick, and Harry. Harry develops directly in master for some time before pushing his changes.

- Tom commits his changes to master, and pushes immediately. His commits are {1,2,3,4,5}
- Dick commits changes to a local, unpublished branch. His commits are {a,b}. After he is done developing locally, he merges his branch into master and pushes immediately, resulting in commit 6.
- Harry commits his changes to his local master branch. His commits are {A, B} Periodically, he uses `git pull` to bring in changes from the remote master branch, resulting in auto-generated merge commits {P1, P2, and P3}. At the end, he pushes his changes to the repository.

Harry's practice can cause some unintuitive-looking history graphs. His workflow looks something like:

- (master) `git pull`
- (master) `git commit -m "A"`
- (master) `git pull` (results in a silent, automatic merge commit P1 since this pull is not fast-forward)
- (master) `git pull` (results in another silent, automatic merge commit P2 since this pull is not fast-forward)
- (master) `git commit -m "B"`
- (master) `git pull` (Yet another merge commit P3)
- (master) `git push` (The repository master now is identical to his local master)



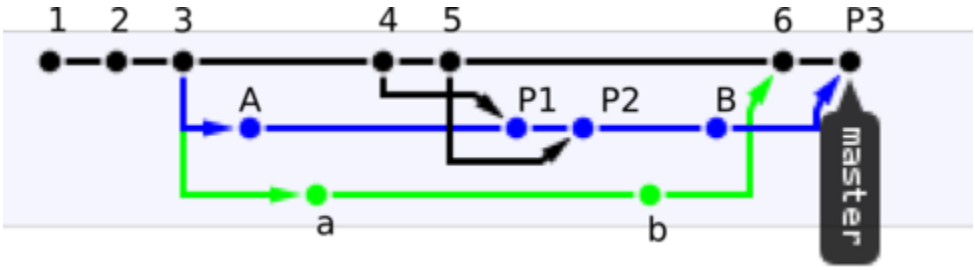
In this graph, Harry's local commits and pull merges appear to have occurred in master. Tom's commits (which were always pushed immediately to master) appear to have occurred in a separate branch. In a way, this is actually an accurate representation of what has occurred. Harry made some commits in his master branch, merged in changes from a another branch three times, then replaced the repository master with his own.

Development in a local branch

Development in a local branch (even with occasional merges with master) is a valid and recommended development pattern. If parallel commits have been pushed to master in the meantime, this workflow will represent your local changes as if it indeed were a separate branch.

Let us use the same Tom, Dick, & Harry example, except with Harry performing his development in a local, non-published branch. In this example, Harry's workflow looks like the following:

- (master) `git pull`
- (master) `git branch harry_branch origin/master --track`
- (harry_branch) `git commit -m "A"`
- (harry_branch) `git pull` (results in a silent, automatic merge commit P1 since this merge is not fast-forward)
- (harry_branch) `git pull` (results in a silent, automatic merge commit P2 since this merge is not fast-forward)
- (harry_branch) `git commit -m "B"`
- (master) `git pull` (Is fast-forward. No merge commit created)
- (master) `git merge harry_branch` (results in an explicit merge commit P3}
- (master) `git push` (The repository master now is identical to his local master)



As is evident, the github history graph is still complex, but perhaps more "intuitive" in the sense that it preserves the fact that commits 1,2,3,4,5 and 6 had been published in master, and that Harry's commits A and B occurred in some other branch. Harry's pull merges are also preserved - but this time it is clear that changes (commits 4 and 5) were propagated from master into his own branch during the pull/merge, and that he merged his branch back into the published master at the end.

With this technique, pushing the local branch (harry_branch) to the repository occasionally would make no difference, and would be safe. This pattern has an identical end result to maintaining a published fcrepo-XXX feature/bug branch, and merging it with master in the end.

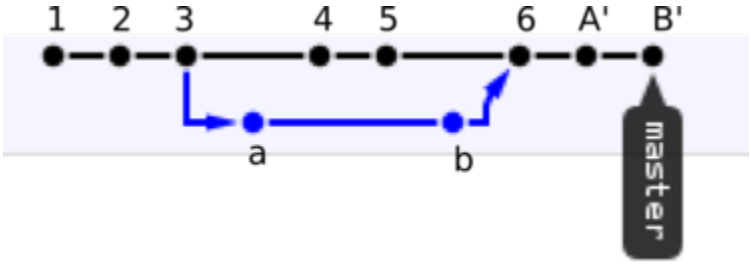
Development in a local branch with rebase

Development in an **unpublished** local branch, and using `git rebase` instead of `pull` or `merge` to update the local branch with changes to master is also a valid pattern. This technique results in the elimination of the local branching history, and rather than a final merge applies all local commits in sequence to the end of the current master. This may be used when the local branch and merge history is unimportant or unnecessary (perhaps bad luck - while making two trivial local commits, somebody happened to push master in the meantime).

It is important to **never rebase a published branch** if you intend on ever pushing that branch again. As a safety, Git will refuse to push a branch that has had its history re-written with rebase. Although it is possible to force the changes through with `--force`, never do that!

Let us use the same Tom, Dick, & Harry example, except with Harry performing his development in a local, non-published branch, with occasional rebasing to track the changes in master.

- (master) `git pull`
- (master) `git branch harry_branch origin/master --track`
- (harry_branch) `git commit -m "A"`
- (harry_branch) `git fetch; git rebase origin/master` (Modifies Harry's A commit so that it appears to have occurred after all changes that have been imported from master)
- (harry_branch) `git fetch; git rebase origin/master` (Modifies Harry's A commit so that it appears to have occurred after all changes that have been imported from master)
- (harry_branch) `git commit -m "B"`
- (harry_branch) `git fetch; git rebase origin/master` (Modifies Harry's A and B commits so that they appear to have occurred after all changes that have been imported from master)
- (master) `git pull` (Is fast-forward. No merge commit created)
- (master) `git merge harry_branch` (fast-forward. Does not result in merge commit)
- (master) `git push` (The repository master now is identical to his local master)



This results in a **very** simple history. Since rebase operations result in new commits at the end of a tree, Harry's a and B commits were transformed into A' and B', which could be simply applied almost as a patch directly to the end of master in a fast-forward merge. The end result is exactly the same as if Harry were to `git cherry-pick` the two commits from his harry_branch onto master - they both result in new commits at the tail of a branch.