GIT Tips and Tricks

On this Page

See Also

- git Documentation
- github help
- git manual
- Pro Git Book
- git-cheatsheet
- git Primer [PDF]
- YUI Theatre: Intro to Git
- Github Tips and Tricks

Best practices

Work in a branch

It is best to keep your master branch clean and up-to-date with the project repo. This will provide you with a clean space to compare your "working" code with, as well as a stable location to push changes back up to the project repo.

```
# list all branches, including remotes
git branch -a

# change branches, in this case switch to master
git checkout master

# make a branch of master, and immediately switch to the branch
# Note: it's best to name your branches after the jira you are working on (e.g. FLUID-xxxx)
git checkout -b FLUID-XXXX
```

It is best to name your branches after the jira that you are working on (e.g. FLUID-xxxx). This will make it easier for you to keep track of what the branch is for. When you issue a pull request, push it to a public repo space (e.g. github), and/or merge the branch into master the branch name will be visible and act as a means of indicating to others what changes are part of the branch.

There will be some occasions where you'll want to work directly in master, make sure you really mean to do it and that you are careful. Working directly in master is likely not a common workflow and should probably only occur for cases where you have only a single commit that will be pushed to the project repo right away.

Merging Branches into master, and pushing to the project repo

When you have a branch that is ready to go into the project repo you'll need to merge it into your clean, up-to-date master.

```
# get the latest changes, where upstream is the project repo
git fetch upstream

# view the changes that are in the project repo that aren't currently in your master
git log upstream/master ^master

# view the changes that are in your master, that aren't in the project repo (should be none)
git log ^upstream/master master

# update your master by merging in the changes from the project repo
# assuming you are already in the master branch
git merge upstream/master

# view the changes that are in master that aren't currently in your branch
git log master ^FLUID-xxxx

# view the changes that are in your branch, that aren't in the master
git log ^master FLUID-xxxx

# merge the branch into master.
# Note: --no-ff forces a merge commit
# Note: --log lists all the commits that are part of the merge (if there are lots of commits you may need to
increase the max number --log=n)
git merge FLUID-xxxx --no-ff --log

# push to your public repo, origin, first to make sure things worked
git push origin master

# push to the project repo
git push upstream master
```

When merging your branch back into master, always make sure to set the --no-ff and --log flags. The --no-ff flag forces a merge commit. If this isn't set, and a fast-forward merge occurs, the master branch can take on the characteristics of the branch as though master was a clone of the branch. This may be undesirable, for example the git's graphing feature will make the branch commits appear to be the mainline of code. The --log flag will place the summary line of the commits included in the merge, into the merge commit. This makes tracking which commits were part of the merge much easier.

It's also a good idea to push to your public repository before pushing to the project repo. This will give you another chance to make sure everything is as it should be. For example, in github you can review the commit logs and the network graph to make sure the merge was performed properly.

Commit Logs

In git, commit logs are structured a lot like e-mails. The first line is a summary, 50 characters or less. The remainder of the commit log should contain the detailed description of what is being committed.

```
#summary, it should start with a jira number.
FLUID-xxxx: Adding in feature x to component y

#This is just a mock example, a real commit log would have proper details about the changes in a commit
Feature x adds does such and such a function, which is necessary for component y.
```

The summary of each commit should start with the jira number for the issue being worked on. In the rare case where it is not necessary to file a jira for a change, "NOJIRA:" can be used instead.

Notice the empty line between the summary and the body of the commit log. Some clients have trouble distinguishing the two parts if they are not separated like this.

Tips

Log

(see: git-log)

Lists commits in branch x that aren't in branch y

```
git log x ^y
```

Lists commits and also shows the diff of the changes

```
git log -p
```

Diff

(see: git-diff)

Diff of unstaged changes

```
git diff
```

Diff of staged changes

```
git diff --cached
```

Diff between two commits

```
git diff commitHash1 commitHash2
```

Diff between branch x and y

```
git diff x y
```

Diff between a path at a revision and a path in the working tree

```
git diff rev:path1 path2
```

Resetting working copy

(see: git-reset)
(For a detailed description of all the types of reset available see: http://git-scm.com/2011/07/11/reset.html);

Unstage changes

```
git reset HEAD
```

Undoes all changes, staged or not, so that the working copy is back to the state of the last commit

```
git reset HEAD --hard
```

Sets the working copy back to the state of the specified commit

```
git reset commitHash --hard
```

Resetting Individual Files

commitHash can be replaced by either a commit hash, tag, or branch name.
You can also use the "~1" at the end of a commit hash to use the 1 before.
This is good when you know the commit with the error.
(See the "Reset with a path" and "Check it out" sections: http://git-scm.com/2011/07/11/reset.html);

```
git checkout commitHash path/to/file/
```

Another method, which preserves the working directory, is to use `git reset`

```
git reset commitHash path/to/file/
```

Deleting Branches

Deletes local branch x

```
git branch -d x

#to force a branch to delete even when it has unmerged changes
git branch -D x
```

Deletes branch x from remote repo

```
#note the ":"
git push remoteRepo :x
```

Remove stale branches fetched from the remote at remoteName

```
git remote prune remoteName
```

Stashing changes

Useful to temporarily save changes that aren't ready to be committed

```
git stash
```

Re-apply stashed changes. If more than one set of stashed changes, you can specify which stash to apply.

```
git stash apply <stash name>
```

List stashed changes

```
git stash list
```

Clear stash

```
git stash clear
```