colinsurprenant / **step-by-step-git-workflow.md**

Last active 5 months ago • Report abuse

step by step git workflow

step-by-step-git-workflow.md

Git/Github step-by-step Workflow

Step-by-step guide for creating a feature or bugfix branch, submit it for code review as a pull request and once approved, merge upstream. This guide is intended for internal developers with push access to all relevant repos.

You should understand rebasing and squashing. For a very good explanation on rebasing and squashing with pull requests see How to Rebase a Pull Request. Also worth reading is the Hacker's Guide to Git.

Setup

1- fork user/project repo

2- clone your fork locally

```
$ git clone git@github.com:{username}/project.git
```

3- configure remotes

There are different philosophies for naming remotes. The *Github standard* is to use `origin` for your forked repo and `upstream` for the original repo. Another naming strategy is to use descriptive remote names and use `<your name>` instead of `origin` and `<project name>` instead of `upstream`. This is particularly interesting when you also need to pull somenone else's branch; you then create a remote with that person's name and your naming remains consistent.

For the sake of the examples below, we will use `origin` and `upstream`.

```
git remote add upstream git@github.com:user/project.git
```

- at this point you can push commits to your forked repo with

```
$ git push origin master
```

- and you can pull in upstream changes

```
$ git fetch upstream
$ git merge upstream/master
```

4- add the pull request git alias

Add this at the end of `.git/config`

```
[alias]
pr = "!f() { git fetch upstream pull/$1/head:pr/$1; }; f"
```

This can be used to fetch a pull request locally using

```
$ git pr 1234
```

5- rerere

For handling frequent rebase/merges (especially with long-lived branches) it's handy to enable the `rerere` cache by adding:

```
[rerere]
  enabled = 1
```

to `~/.gitconfig` (or run `git config --global rerere.enabled true`). This cache can be used to resolve merge conflicts that you have resolved once before, using the cached version of the resolution.

Work & commits

1- create a feature or bugfix branch

```
$ git checkout -b feature/<feature name>
```

```
$ git checkout -b fix/<issue id>
```

2- work, commit you changes, rince & repeat

For example, lets pretend 2 commits:

```
$ git commit -m “some change” file_a
$ git commit -m “some other change” file_b
```

Generate pull request for code review

1- pull in latest upstream master

```
$ git checkout master

$ git fetch upstream
$ git merge upstream/master
```

2- rebase branch against master

```
$ git checkout feature/foobar
$ git rebase master
```

3- push to your forked repo

If you rebased, you are changing the history on your branch. As a result, if you try to do a normal git push after a rebase, git will reject it. Instead, you'll need to use the `-f` or `--force` flag to tell git that you really know what you're doing. Note that you typically want to avoid using `-f` on shared branches.

```
$ git push origin feature/foobar -f
```

4- submit pull request

On Github in your forked repo, submit the pull request for the branch

Code review on pull request

In the review process, you can create new commits and push them to your forked repo to update the pull request.

```
$ git commit -m “review fix” file_a
$ git push origin feature/foobar
```

Merge branch upstream

Once the code review is complete, merge the branch upstream

1- squash commits (optional)

It is not absolutely required to squash commits. For bigger changes or refactoring the changeset makes more sense with separate commits. Also you should never squash commits for the pull request submission. Squashing must happen only after the review process, just before merging.

- You can squash using your branch base commit with `merge-base` to retrieve the base commit hash:

```
$ git merge-base my-branch master
$ git rebase -i <base commit hash>
```

- Or you can squash on the last N commits:

```
$ git rebase --i HEAD~N
```

This will result in presenting an editor with something like:

```
pick f73ed64 some change
pick b8d914c some other change

# Rebase b710536..b8d914c onto b710536
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

For every line **except the first**, you want to replace the word `pick` with the word `squash`. It should end up looking like this:

```
pick f73ed64 some change
squash b8d914c some other change
```

Save-quit the editor, and you'll get another editor with something like:

```
# This is a combination of 2 commits.
# The first commit's message is:

some change

# This is the 2nd commit message:

some other change

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# rebase in progress; onto b710536
# You are currently editing a commit while rebasing branch 'testflow' on 'b710536'.
#
# Changes to be committed:
#       new file:   some change
#       new file:   some other change
```

```
#
# Untracked files:
#       .ruby-version
#
```

You can add more comments at the end of the file. **You should add on the last list the reference to the issue it closes** which will automatically close the issue

```
closes 1234
```

Save-quit the editor

2- rebase agaist master

First update master

```
$ git checkout master
$ git fetch upstream
$ git merge upstream/master
```

Rebase branch against master

```
$ git checkout feature/foobar
$ git rebase master
```

3- push to your forked repo

If you did a rebase/squash, you are changing the history on your branch. As a result, if you try to do a normal git push after a rebase, git will reject it. Instead, you'll need to use the `-f` or `--force` flag to tell git that you really know what you're doing. Note that you typically want to avoid using `-f` on shared branches.

```
$ git push origin feature/foobar -f
```

4- merge into master

If the commits were not squashed, optionally use the `--no-ff` to merge so that git always add a merge commit to maintains the history of the feature branch.

```
$ git checkout master
$ git merge feature/foobar
```

5- push upstream

```
$ git push upstream master
```

This will automatically **close the pull request** and **close the referenced issue** in the commit comment.

6- back-port in release branches if required

Do not forget to back-port your commit in the release branches, for example in 1.x.

Checkout branch

```
$ git checkout 1.x
$ git fetch upstream 1.x
$ git merge upstream/1.x
```

Merge commit

```
$ git cherry-pick {commit hash}
```

Push upstream

```
$ git push upstream 1.x
```

7- label PR/issue

Label the PR/issue with the branches it was back-ported into, for example with the v1.x label.

Reviewing a PR

1- fetch PR locally

Make sure you added the pr alias as described in the setup section.

```
$ git pr 1234
```

2- rebase agaist master

First update master

```
$ git checkout master
$ git fetch upstream
$ git merge upstream/master
```

Rebase branch against master

```
$ git checkout pr/1234
$ git rebase master
```

3- test locally and optionally create new commits

```
$ git commit -m “some change” file_a
$ git commit -m “some other change” file_b
```

4- merge pr upstream

Once the code review is complete, merge the pr upstream

1- squash commits (optional)

See earlier section on commits squashing

2- merge into master

If the commits were not squashed, optionally use the `--no-ff` to merge so that git always add a merge commit to maintains the history of the feature branch.

```
$ git checkout master
$ git merge feature/foobar
```

3- push upstream

```
$ git push upstream master
```

This will automatically close the referenced pull request and close the referenced issue in the commit comment, by addding

```
closes #1234
```

4- back-port in release branches if required

Do not forget to back-port your commit in the release branches, for example in 1.x.

Checkout branch

```
$ git checkout 1.x
$ git fetch upstream 1.x
$ git merge upstream/1.x
```

Merge commit

```
$ git cherry-pick {commit hash}
```

Push upstream

```
$ git push upstream 1.x
```

5- label PR/issue

Label the PR/issue with the branches it was back-ported into, for example with the v1.x label.