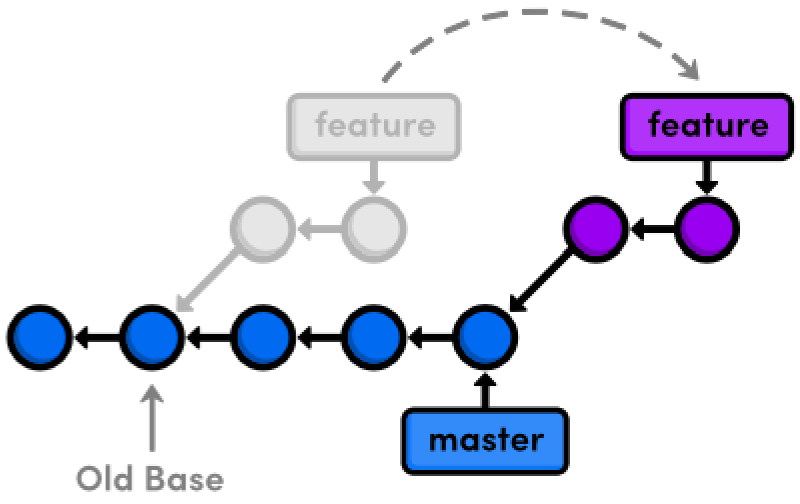
Git Rebase HOWTO

Rebases can be scary! They are all about rewriting your commit history. But it is a useful tool when working on your own branch in order to keep your branch clean and tidy.

What is a rebase doing?

Conceptually, a rebase is taking a branch and replaying the commits from that branch one-by-one onto the new 'base' set of commits. A visual demonstration:



The old commits are still there! The rebase merely creates new commits and updates the branch to point to the new ones. However, the old commits will disappear once git runs garbage collection unless there is another branch that still contains those commits.

There is also an excellent interactive demo of git that will give you a visual understanding of how the working tree gets updated after various commands (rebase, merge, cherry-pick).

Simple rebase onto master

This is the most common operation you will see:

```
git fetch; git rebase origin/master
```

This will pull down the latest information from origin, and then it will take your branch and base it on the current version of master. This way your branch doesn't need a dozen new 'merge from master' commits every time you'd like the newest updates to master in your current branch.

If this succeeds, you'll need to push -f so that your local rebase is reflected on your branch at Github. See the section "The dangers of push -f" below.

Dealing with conflicts

Sometimes your rebase runs will run into conflicts!

This is the error message you get:

```
Falling back to patching base and 3-way merge...
Auto-merging cms/urls.py
CONFLICT (content): Merge conflict in cms/urls.py
...
Failed to merge in the changes.
Patch failed at 0001 things now upload
When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase --skip".
To check out the original branch and stop rebasing run "git rebase --abort"
```

From here, you have three options:

1. Fix the merge conflicts and continue the rebase
2. Skip this particular commit and continue rebasing
3. Stop the rebase and roll everything back to how it looks pre-rebase

Fixing the rebase conflicts

Fix the conflicts within the files, and then when you are happy with the result, add the files to the stage

```
git add <file>
```

And then continue with your rebase:

```
git rebase --continue
```

Since git is trying to apply each individual commit on top of its new place in the tree, you may have to resolve more conflicts! One way to reduce the pain is to make sure your commit history is clean and that there's not a whole lot of overlap in commits on particular files. You can keep your history clean using interactive rebase (see below).

Skipping a commit

I don't recommend skipping commits when rebasing, but you can perform them with this command:

```
git rebase --skip
```

Aborting a rebase

If you are dealing with a conflict and would like to rollback your rebase, all you need to do is type:

```
git rebase --abort
```

The state of your working tree will be reverted to what it looked like before you started rebasing. Abort whenever you feel uncomfortable with what rebase is doing.

Interactive rebase

You can use rebase to modify your history in an interactive, powerful way. In most cases you'll want to just want to tell git that you want to interactively rebase the last X commits on your branch. For example, if you would like to rebase the last 3 commits on your current branch, you can do:

```
git rebase -i HEAD~3
```

This will give you a list of commits and some options for how to interact with them. The most recent commits are at the bottom.

```
pick a6c5530 thumnail image is a thumnail image
pick b3061a6 schema change to better normalize asset->thumbnail relationships
pick 92a9049 fix up commenting after refactoring. Also, don't reraise an exception
during thumbnail generation - we just want to swallow the exception, log, and
continue.
# Rebase 0d41907..92a9049 onto 0d41907
#
# 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.
```

When you save this file, git will write the new changes onto the branch. If you remove all the lines before saving, git will abort the rebase.

Simple squash of history

To do a simple squash of the history, all you need to do is change the 'pick' to 'squash'.

```
pick a6c5530 thumnail image is a thumnail image
pick b3061a6 schema change to better normalize asset->thumbnail relationships
squash 92a9049 fix up commenting after refactoring. Also, don't reraise an exception
during thumbnail generation - we just want to swallow the exception, log, and
continue.
```

This will merge commit 92a9049 and b306a6 into one single commit and give you the option of editing the commit message.

Editing an existing commit

git commit --amend

This technically isn't a rebase, but I have found it very useful. Forget a file in your last commit? Found a typo immediately after committing? First add the updated versions of the files to your stage.

```
git add <file>
```

Then commit the staged changes with the amend flag.

```
git commit --amend
```

This will overwrite your old commit with a new commit that contains both your old changes and your new changes.

Interactively amend a commit

You can also amend commits that are further down in your history by using an interactive rebase. Change the `pick` to an`edit` for the commit(s) you would like to edit.

```
pick a6c5530 thumnail image is a thumnail image
edit b3061a6 schema change to better normalize asset->thumbnail relationships
pick 92a9049 fix up commenting after refactoring. Also, don't reraise an exception
during thumbnail generation - we just want to swallow the exception, log, and
continue.
```

During the rebase, git will stop at this commit and give you the chance to amend this particular commit. Make any changes you would like to make, add them to the stage with

```
git add <file>
```

and then amend the commit with

```
git commit --amend
```

After you are satisfied with your changes to this commit, run

```
git commit --continue
```

to continue the rebase. All other commits being replayed on top of this edited commit will now have new hashes.

Use fixup to amend a commit

Switching `pick` to `fixup` will merge the commit into a previous commit and keep the commit message of the previous commit. One alternative to interactively amending the commit is to make a new commit, and then use a combination of reordering and `fixup` to fix problems with a commit.

First, commit the new changes onto the branch and run git rebase -i HEAD~3. This is what it looks like:

```
pick b3061a6 schema change to better normalize asset->thumbnail relationships
pick 92a9049 fix up commenting after refactoring. Also, don't reraise an exception
during thumbnail generation - we just want to swallow the exception, log, and
continue.
pick c927491 Fix pep8 and pylint violations.
```

You can then reorder and fixup:

```
pick b3061a6 schema change to better normalize asset->thumbnail relationships
fixup c927491 Fix pep8 and pylint violations.
pick 92a9049 fix up commenting after refactoring. Also, don't reraise an exception
during thumbnail generation - we just want to swallow the exception, log, and
continue.
```

The rebase will then squash the commits and discard the commit message talking about pep8 and pylint violations.

The dangers of push -f

If you try to push a rebased branch back up to origin (Github) with `git push`, it can complain about how the branchs are not properly in sync. You will see an error message that looks like this:

```
error: failed to push some refs to 'git@github.com:edx/edx-platform.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Check the upstream branch. Are there, in fact, new changes there? If your branch has the same commits, more or less, as the one on your local machine, continue. If not, see the section on Working with other people/across multiple machines below.

Even though the git instructions say to run `git pull`, don't do it. That will just mess up your rebased branch!

In order to get your upstream Github branch up to proper, sync, you will have to run:

```
git push -f
```

Easy, right?

But wait! There's a catch.

Force-pushing is dangerous
Be very careful when doing a force push! You are essentially overwriting the history of the branch on the repo If you have not set your git defaults correctly, then you run the risk of overwriting all of the branches in the repo, not just the one you are intending to overwrite.

Make sure your default push behavior is correct (verify that `git config --get push.default` returns`upstream` or `current`) before force-pushing! If your need to set this correctly, use: `git config ----global push.default simple`

Force-pushing to master
If you do ever force push to master (whether intentionally or accidentally), be sure to send email to both dev@edx.organd edx-code@googlegroups.org. Internal and external developers create branches from and rebase onto master frequently, and the commits that you removed by force pushing will appear in any pull request from a branch created while those commits were still on master. This can be very confusing, so you should be sure to inform everyone affected when it happens.

Rebase to a common ancestor

Sometimes you want to rebase your changes onto master from a common ancestor. (Helpful ascii art is from the official git documentation.) This will probably only happen if you are working off a long-running branch or if you accidentally make a branch off another branch.

If you end up in a situation like this:

```
o---o---o---o---o  master
         \
          o---o---o---o---o  next
                           \
                            o---o---o  topic
```

and you would like to turn it into this:

```
o---o---o---o---o  master
        |            \
        |             o'--o'--o'  topic
         \
          o---o---o---o---o  next
```

then you type in this command:

```
git rebase --onto master next topic
```

This will correctly find the common ancestor between `topic` and `next` and rebase `topic` onto `master`.

Working from different machines/with other people

One of the trickiest things about dealing with rebase is working with other people or across different machines.

Recommended best practices

Don't squash or edit commits after pushing if you are working with other people

Cleaning your history before pushing back up onto the branch is a useful practice to get into.

Don't be afraid to make branches off other branches

Branches off other branches make it easier to rebase off the newer version of the branch, much in the way you can rebase a branch off of master. This makes it easier to keep the two branches in sync if more than one person is working on a feature at the same time.

You can use git reset to reset to the version of the code that is on origin

If you would like to discard your current version of the branch and use the one that is on origin, type in:

```
git fetch; git reset origin/<branch name>
```

Have one person clean the commit history once you're ready to merge to master

This way, you don't have to worry about conflicts between team members.

Tools

- Magit for Emacs: http://magit.github.io/magit/
- Git Cola (a pretty good git gui): http://git-cola.github.io/
- GitX (OSX) gitx.org (rowanj fork)

Other reference materials:

- https://www.kernel.org/pub/software/scm/git/docs/git-rebase.html
- http://rypress.com/tutorials/git/rebasing.html
- http://git-scm.com/book/en/Git-Branching-Rebasing
- How to Rebase a Pull Request