git's not a mystery

Basics from the command line

Cloning:

```
git clone https://albert-jan.nijburg@stash.centrastage.com:8443/scm/agent/centrastage.git
```

Creating a branch:

```
git branch -b 'name-of-branch'
```

Pushing a branch to a remote:

```
git push -u origin name-of-branch
```

Commit and add all changed files (use with caution):

```
git commit -am 'commit message'
```

Update the current branch from the tracking remote branch:

```
git pull
```

Update the local repository with everything from the remote but don't change your local branches:

```
git fetch
```

Update the local branch after a fetch:

```
git merge origin/name-of-branch
```

Get a list of commit messages: (where n is the number of messages you want to see.)

```
git log -n
```

Getting yourself out of a hole

You've screwed up your repository or branch and you want to undo go back or revert or reset.

Un-stage files:

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

Or if you want to reset everything:

```
git reset
```

Accidentally removed a file or changed it, but you haven't commited it:

```
git checkout HEAD -- path/to/file
```

If you have commited it and you don't want to revert the entire commit:

```
git checkout HEAD^ -- path/to/file
```

Throw away everything you changed after the last commit:

```
git reset --hard
```

Did a bad merge and want to go back to the state of a branch on the remote.

```
git reset --hard origin/branch-name
```

Halfway through a rebase but not feeling it:

```
git rebase --abort
```

Commiting code like a pro

Make sure only changes that belong together are commited together:

```
git add -p
```

for that. It will present you with a change called a chunk and lets you choose whether or not you want that to be staged.

For example there's a bug you fixed and you've written a unittest and the code that fixes it. You will want to commit the unittest seperately from the code that fixes it. (red-green-refactor and all). You can easily add that to different commits.

Sizeable chucks can be split too. And you can edit the patch for a chunk if you want to commit one line of a two line chunk.

Lets say you're working on a bigish feature, and you have 15 commits, in your branch. I'll bet you some of them will have a message like "and this too" or "forgot about this file.". Those are *not* useful to the reviewer. Here where interactive rebase comes in.

Interactive rebase

Interactive rebase, means you're going to be rewriting history. This works best if you don't have merge commits. (read next chapter) With interactive rebase you can reorder, squish/fixup, edit, reword. So it's like the swiss army knife for your PR.

We're going to rebase the last 11 commits:

```
git rebase -i HEAD~11
```

This will open up your editor with a file that looks something like:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
pick ac9dee7 misc bugfixes
pick 204cb44 code additions/edits
pick 719d389 more code
pick f0f9b91 here have more code
pick 76b33dc aaaaaaaaa
pick 0822c0d adkjslkdfjsdklfj
pick 737477f my hands are typing words
pick 8263815 haaaaaaaands

# Rebase 710f0f8..8263815 onto 710f0f8
#
# 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
```

This gives you the opportunity to group and squash the commits so they are grouped per change. After you save and close this file it will open a new file for every commit you've squashed and reworded. Make sure you don't throw away information from commit messages (do throw away when it's useless).

Keeping the history clean

If you consider it your responsibility to get a PR merged into the branch. You should care about the commits in it. Having a merge commit in there or commits that are already in the branch that you want it to get merged into will not help .

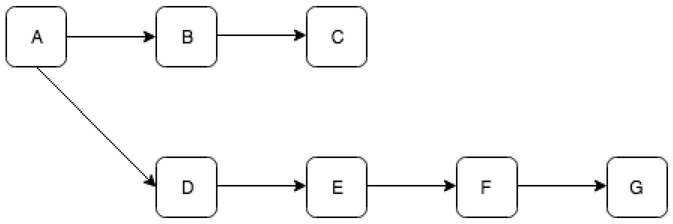
So when there are conflicts with the merge into the upstream branch, do a rebase. A rebase rewinds your commits pull the commits from the upstream branch and then replays your commits after them, letting you resolve any conflicts on a per commit basis.

This nicely hooks into what I described before: Group all changes together in a commit.

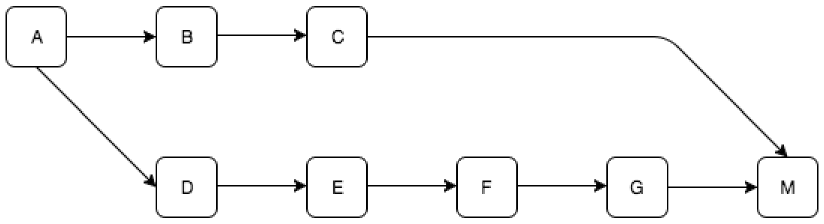
Rebasing basically rewinds the branch to the last time the two branches were the same. Then it adds the changes from the remote, and replays your commits after it. It will force you to resolve conflicts for every commit, this is probably the biggest drawback, and one of the reasons you might want to do a merge instead.

To illustrate how the repository looks before and after a merge and a rebase:

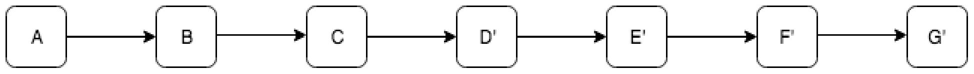
You've done some commits and there are some commits on the remote.



When you merge upstream back into your branch:



When you rebase instead of merge:



You should remember that this will be the state of your PR, so another merge will be added when the PR is merged.