Git (cookbook)

Git

Initial Repo Configuration

```
git config user.name "UserName"
git config user.email gitHubAccount@email.address
```

Fundamental Concepts

Local branch vs. remote branch

- **local branch**: a branch only the local user can see. It exists only on your local machine.
  - Ex. Create local branch named "myNewBranch": `git branch myNewBranch`
- **remote branch**: a branch on a remote location (in most cases 'origin'). Local branches can be pushed to 'origin' (a remote branch), where other users can track it.
  - Ex. Push local branch, "myNewBranch", to the remote, "origin" so that a new branch named "myNewBranch" is created on the remote machine ("origin"):

    ```
    git push -u origin myNewBranch
    ```
- **remote tracking branch**: A local copy of a remote branch. When 'myNewBranch' is pushed to 'origin' using the command above, a remote tracking branch named 'origin/myNewBranch' is created on your local machine.
- **local tracking branch**: a local branch that is tracking another branch.

source(s): SNce & Brian Webster.stackoverflow.com

HEAD, master, and origin

I highly recommend the book "Pro Git" by Scott Chacon. Take time and really read it, while exploring an actual git repo as you do.

- **HEAD**: the current commit your repo is on. Most of the time HEAD points to the latest commit in your current branch, but that doesn't have to be the case. HEAD really just means "what is my repo currently pointing at".
  In the event that the commit HEAD refers to is not the tip of any branch, this is called a "**detached head**".
- **master**: the name of the default branch that git creates for you when first creating a repo. In most cases, "master" means "the main branch". Most shops have everyone pushing to master, and master is considered the definitive view of the repo. But it's also common for release branches to be made off of master for releasing. Your local repo has its own master branch, that almost always follows the master of a remote repo.

- **origin**: the default name that git gives to your main remote repo. Your box has its own repo, and you most likely push out to some remote repo that you and all your coworkers push to. That remote repo is almost always called origin, but it doesn't have to be.
- `HEAD` is an official notion in git. `HEAD` always has a well-defined meaning. `master` and `origin` are common names usually used in git, but they don't have to be.

source: HEAD, master, and origin. Matt Greer & Jacqueline P. via stackoverflow.com

Branching

Suppose your application is stable. Later, you discover a gigantic bug that was passing silently. You want to write some tests, fix the bug, and eventually have a stable, passing application once again. To do this, you'd create a branch for the fix and push the branch to the remote so that all of the developers on your team can collaborate and make the fix.

Once all of the necessary changes have been made and the application is stable, someone from the team would commit merge the commits from the other branch into master. Since the commit history from the branch will have been saved to master, the new branch could be deleted without loss of information (if you no longer wanted to work on this branch).

- **View all local branches**: `git branch`
- **Switch branches**: `git checkout [branch-name]`
- **Grab a file from a specific branch**: `git checkout [branch_name] [paths]` . Note that if the files are on a remote branch, you'll have to use `git checkout origin/[branch_name] [paths]` instead.

Merging

Merge the specified branch's history into the current one. `git merge [branch]`

Deleting branches

Delete local branch `git branch -d [branch-name]`

Delete remote branch `git push origin --delete [branch-name]`

Git flow

When working on a new feature, branch off from the `develop` branch:

```
git checkout -b newFeature develop
```

Merge finished features into the development branch to add them to the upcoming release. Use the "no fast forward" flag, `--no-ff` , to cause the merge to create a new commit object even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature.

```
git checkout develop
git merge --no-ff newFeature
git branch -d newFeature
git push origin develop
```

Editting commit history

Permanently removing files from commit history

WHy do this? You may have commited a password, some other sensitive information, or a large file that you want to remove from github. If the change is only a few commits back, you can "rebase" changes out of the history. I actually need to do this for a much older set of files and accidentally uploaded a textbook that takes up almost a GB of space.

```
git filter-branch --force --index-filter "git rm --cached --ignore-unmatch PathToSensitiveFile" --prune-empty --tag-name-filter cat --
```

All you need to change is the `PathToSensitiveFile` item. Once you've used this command for all of the files you'd like to get rid of, update the origin by typing `git push origin --force --all` .

How to merge a specific commit from another branch?

Ex.: `git cherry-pick [commit-codename]`

Making a small edit to the most recent commit

```
git commit --amend
```

This allows you to combine staged changes with the previous commit without writing a new commit. It simply edits the previous one in-place.

To amend with an updated commit message: `git commit --amend -m "updated message"`

To amend without changing the commit message: `git commit --amend --no-edit`

Commit part of a file rather than all of its changes

git add --patch [file-name]

Ref: https://stackoverflow.com/a/1085191

Special Topics

Git Aliases

Ref: Git Basics - Git Alises

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.ac '!git add -A && git commit'
```

Changing remote after a repo name change

If you change the name of a repository, the fetch/pull commands may stop working. Git will show you an error along the lines of " `error: failed to push some refs to '{repo_url}.git'` ".

To check which URL the remote references to, use `git remote -v` . Then, set a new URL for remote origin:

```
git remote set-url origin {new_repo_url}
```

SSH keys

An SSH key is an alternative to username/password authorization on GitHub. This will allow you to bypass entering your username and password for future GitHub commands.

SSH keys come in pairs, a public key that gets shared with services like GitHub, and a private key that is stored only on your computer. If the keys match, you're granted access.

The cryptography behind SSH keys ensures that no one can reverse engineer your private key from the public one.

SSH Keys for GitHub

article

Generating a new SSH key: Follow Generating a new SSH key and adding it to the ssh-agent

article

Large File Storage

See https://git-lfs.github.com

When the .gitignore just won't ignore

```
git rm -rf --cached .
git add .
```

If you already have unstaged changes, you must run the following after editing the ignore patterns.

```
git update-index --skip-worktree <file-list>
```

You can reverse this with:

```
git update-index --no-skip-worktree <file-list>
```

References:

- A successful Git branching model. [web]

GitHub (gh)

GitHub CLI (gh) Cheatsheet

Set the editor as vim instead of the default, nano.

```
gh config set editor vim
```

Other examples of `gh config set`:

```
gh config set git_protocol ssh --host github.com
gh config set prompt disabled
```

```
# list commands
gh issue list
gh pr list
```

```
# list commands
gh pr status
gh repo view
```

`gh pr create`: Creates a pull-request

`gh pr checks`:

GH Issues:

- `gh issue close`:

```
gh issue list
gh issue create --label bug
gh issue view 123 --web
```

Releases and Tags

A Git tag is similar to a Git reference, but the Git commit that it points to never changes. Git tags are helpful when you want to point to specific releases.

Tag objects (created with `-a`, `-s`, or `-u`) are called "annotated" tags; they contain a creation date, the tagger name and e-mail, a tagging message, and an optional GnuPG signature. Whereas a "lightweight" tag is simply a name for an object (usually a commit object).

```
git tag -s v0.0.16 -m "Release description for v0.0.16"
```

Refs:

- Managing Tags - GitHub docs
- `git tag` docs

```
# set env var in sh script
export GPG_TTY=$(tty)
```

Releasing Projects on GitHub

Releases are deployable software iterations you can package and make available for a wider audience to download and use.

Ref: https://docs.github.com/en/repositories/releasing-projects-on-github

About Releases (Docs): https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases

GitHub Workflows and `act`

Run and test workflows locally with nektos/act. Note that it depends on `docker` to run the workflows.

Installation bash script

```
curl https://raw.githubusercontent.com/nektos/act/master/install.sh | sudo bash
```

On MacOS:

```
brew install act
```

Example commands:

```
# Command structure:
act [<event>] [options]
If no event name passed, will default to "on: push"

# Run a specific job:
act -j test

# List the actions for the default event:
act -l

# List the actions for a specific event:
act workflow_dispatch -l

# Run the default (`push`) event:
act

# Run a specific event:
act pull_request

# Run in dry-run mode:
act -n

# Enable verbose-logging (can be used with any of the above commands)
act -v
```

GitHub Badges

Badges are concise, consistent, legible images for open source projects. They are common in GitHub `README`s and other webpages and support many package registries, distributions, continuous integration services, and social networks. They look like this:

Badges come from shields.io. Its source code is hosted at github.com/badges/shields and serves roughly a billion images each month.

Colors

Color options are available with keywords, 6-symbol hex codes, and 3-symbol hex codes.

Template for a Custom Badge

```
https://img.shields.io/badge/<leftLabel>-<rightMsg>-<color>
```

GitHub Markdown

```
> **Note**
> This is a note

> **Warning**
> This is a warning
```

Becomes:

GitHub Events

Pull Request Events

`pull_request` docs

`synchronize` (GitHubEvent): Triggered when a pull request's head branch is updated. For example, when the head branch is updated from the base branch, when new commits are pushed to the head branch, or when the base branch is changed.

Pull-request checklist

- [ ] Describe how to test the PR
- [ ] Screenshot the new behavior if applicable
- [ ] Add a description for context on the chosen implementation strategy
- [ ] Refer to related issues/tasks/cards
- [ ] Resolve merge conflicts: Make sure the target branch is merged into the PR branch.

Self-review

- [ ] Add code comments for lines that reviewer might not understand correctly. Consider refactoring the names of variables and functions for clarity.
- [ ] DRY: Don't repeat yourself
- [ ] KISS: Keep it simple, sweetie

- [ ] YAGNI: You aren't gonna need it. Check that you are not overcomplicating something for the sake of "making it future-proof". Fowler said "Yagni only applies to capabilities built into the software to support a presumptive feature, it does not apply to effort to make the software easier to modify".