A Git Tutorial

What it is, how you use it, and what it's good for

Patrick Sanan

ETH Zurich
patrick.sanan@erdw.ethz.ch

https://github.com/psanan/git_tutorial

Section 1

Who is this for?

Assumed Audience

Assumptions about the audience:

- You use code
- You sometimes feel like you're wasting your time when you work with code, especially when it comes time to collaborate
- You know how to use a terminal, shell, terminal-based text editor, and login file on your computer
- If you have looked a (short) git tutorials online, you don't find them wholly satisfying

Purpose of this Presentation

- Give an idea of the fairly simple (beautiful) data structure which git manipulates
- Give an introduction to the (less beautiful) way you can interact with this from the command line
- Show some of the benefits of using services like Bitbucket, GitHub, or GitLab
- Give a few very simple demonstrations
- Hopefully, convince you that these tools are worth investing the time to learn more thoroughly

Section 2

What is git?

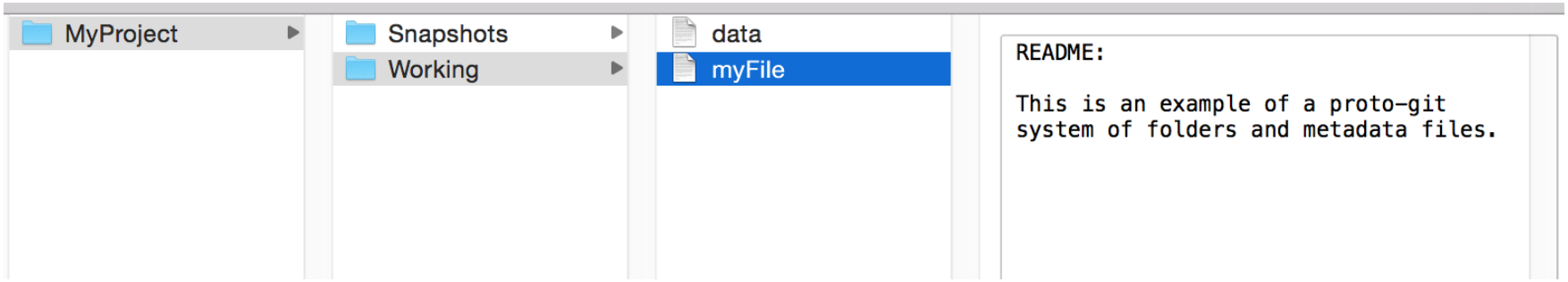
What is Version Control?

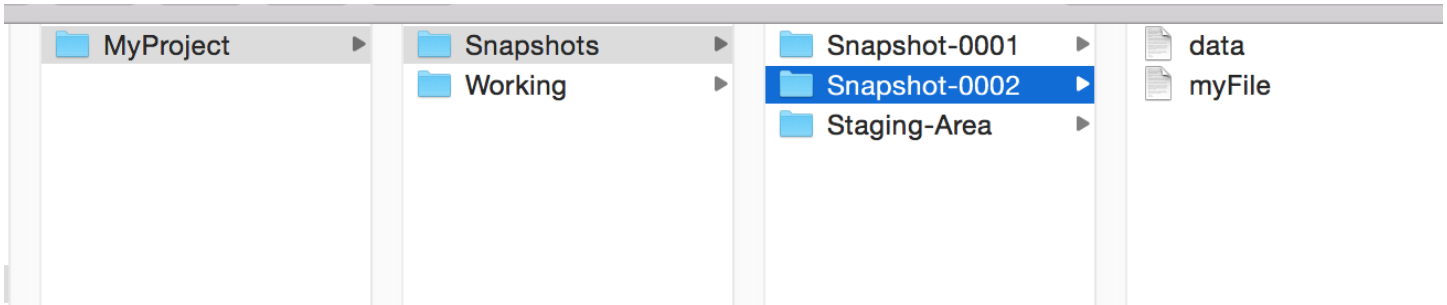
- git is a *Version Control System (VCS)*
- It's a system to help you keep track of the history and versions of a "project" (usually based on source code)
- We will show how one might invent such a thing.[1]

[1] inspired by "The Git Parable" by Tom Preston-Werner

Tracking History - Snapshots

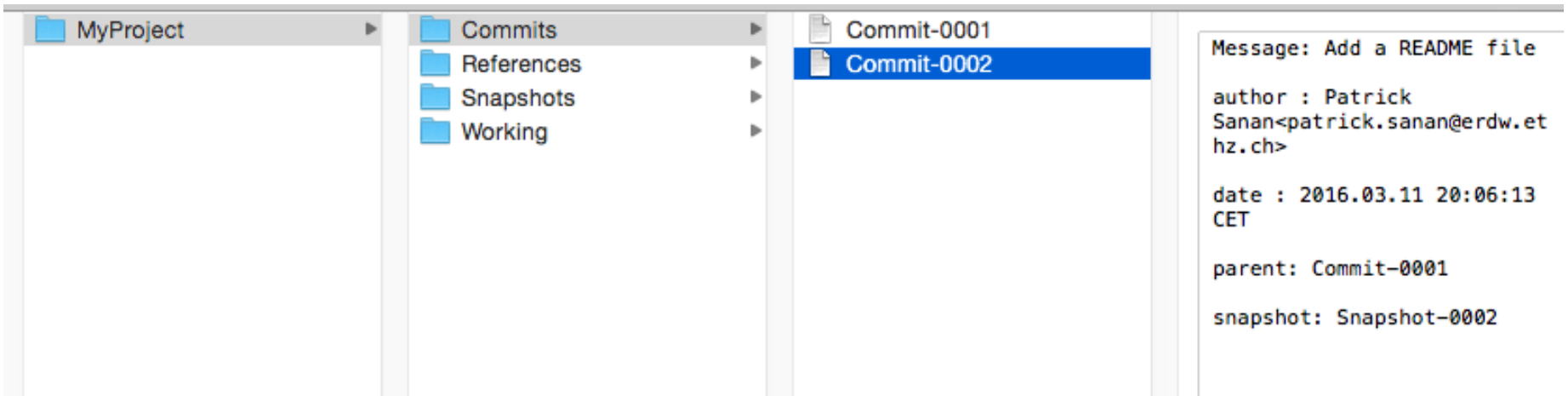
- When working on code, it's natural to want to save certain states.
- You might do this manually, creating folders with different *snapshots* of your data every once in a while.





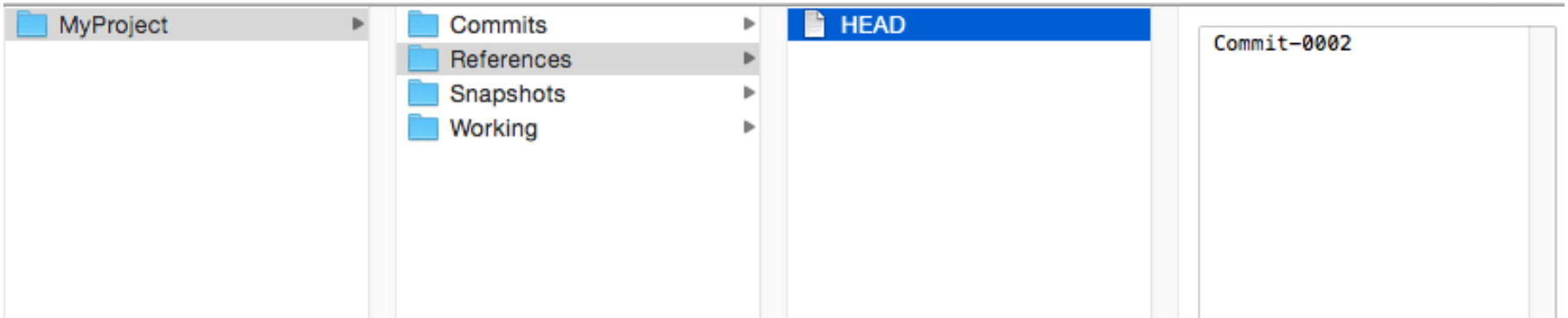
Adding Metadata - Commits

- I might decide that I want more information about each snapshot
- I introduce data in a new folder of *commits*
- I want to be able to "rewind", so I record a *parent* commit for each commit (except the first)



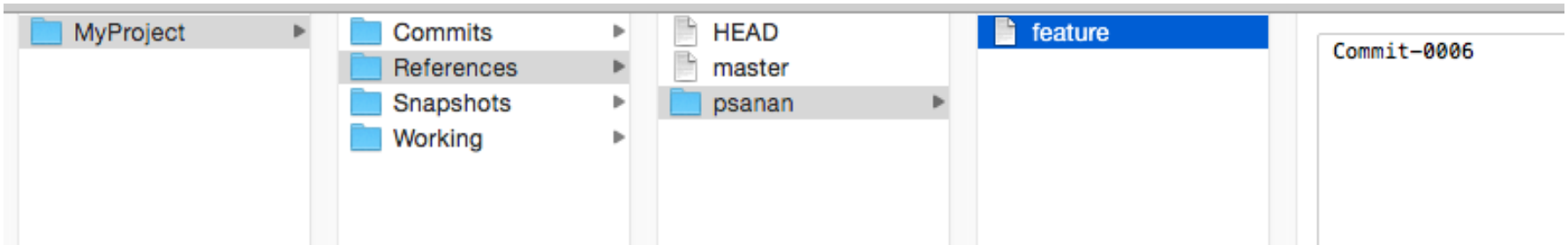
Adding Metadata - Commits

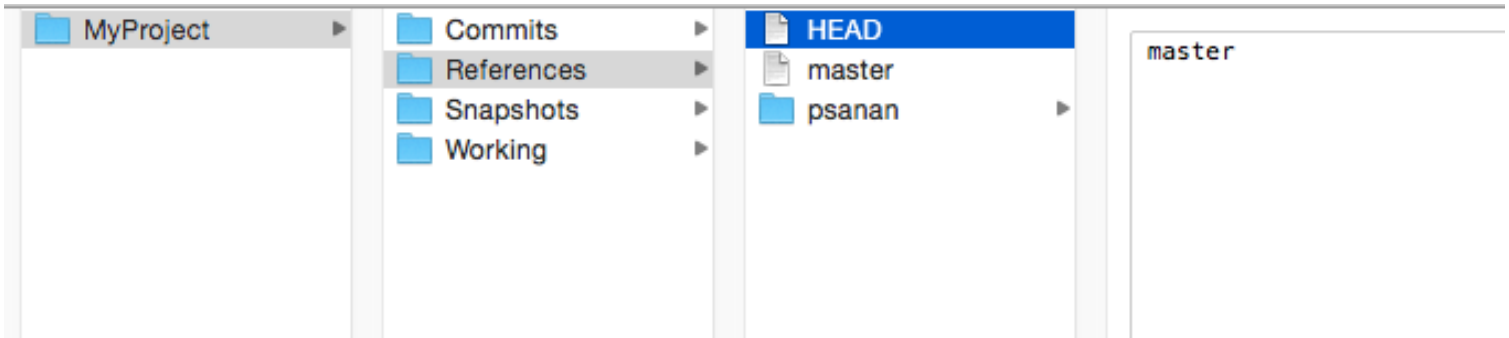
- I also introduce a new file called `HEAD` which points to the latest commit
- To save my state,
  1. Copy the state indicated by `HEAD` to a *staging area* and pick some or all of the changes from my *working directory* to add there
  2. Move everything from the staging area directory to a new snapshot
  3. Create a new commit pointing to my new snapshot, using `HEAD` to define the parent.
  4. Update `HEAD` to the new commit



Different Versions

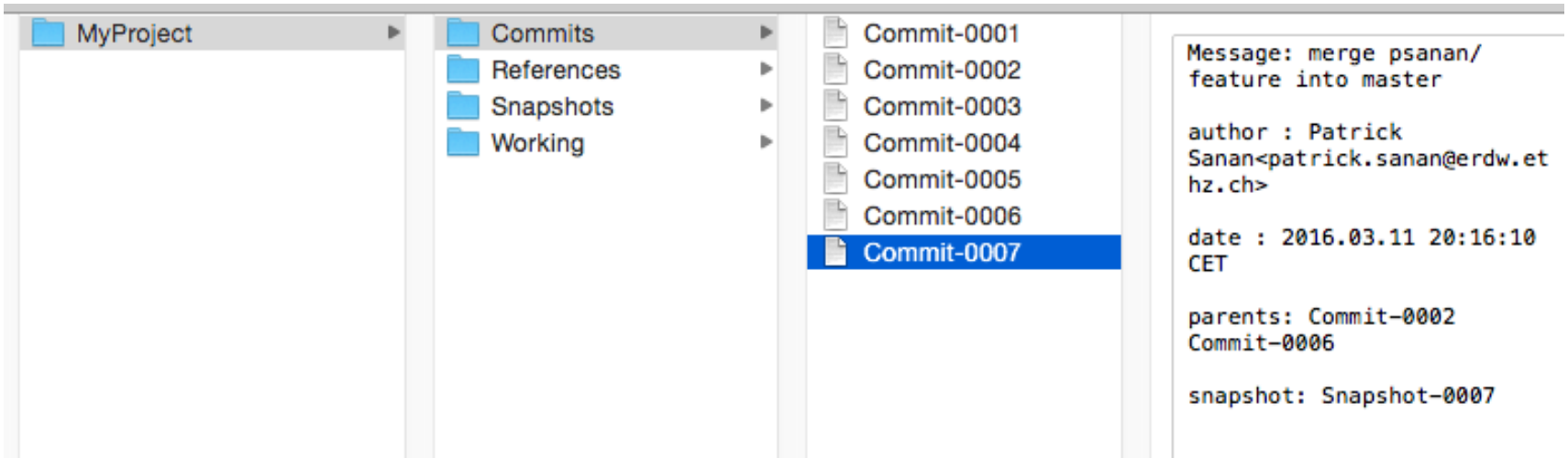
- We can keep track of multiple versions of our project by adding more *references* and updating them.
- We change `HEAD` to now point to one of these *branches*
- As I add new commits, I update the branch pointed to by `HEAD` to point to the new commit





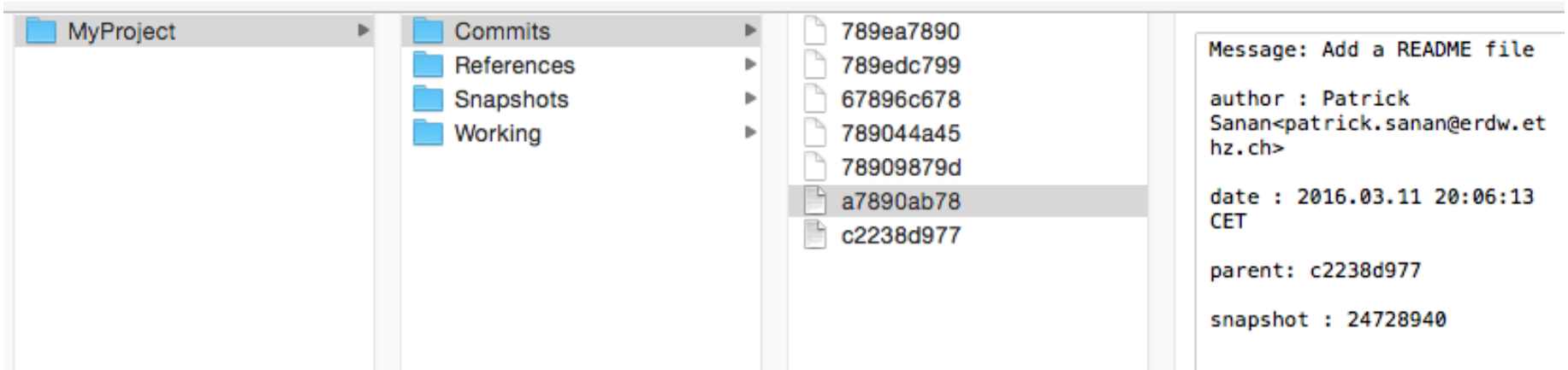
Different Versions

- We realize that we can have multiple parents in a commit, which allows us to *merge* branches.



Other People

- Now, suppose you and I both have a copy of this *repository* (`MyProject`) and we want to collaborate.
- We have the clever idea that we can name the files by applying a function[2] to their contents. Now, we have files with the same name if and only if we have the same data[3].

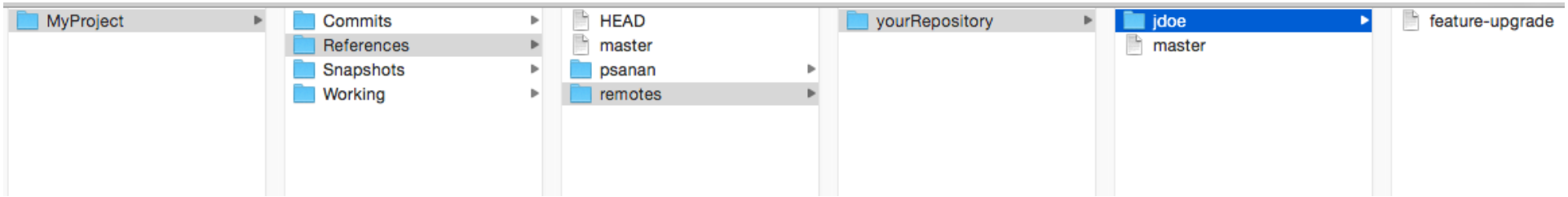


[2] a *cryptographic hash function* such as SHA-1

[3] with overwhelming probability

Other People

- I keep a special set of *remote references* to branches which correspond to your repository
- When I want your changes from a branch
  - You send me the reference to your version of the branch.
  - I look at the reference and ask for any files (commits and snapshots) that I am missing.
  - I can make a merge commit as before.
  - I can then send you my reference and you can repeat the steps so that we are synchronized.



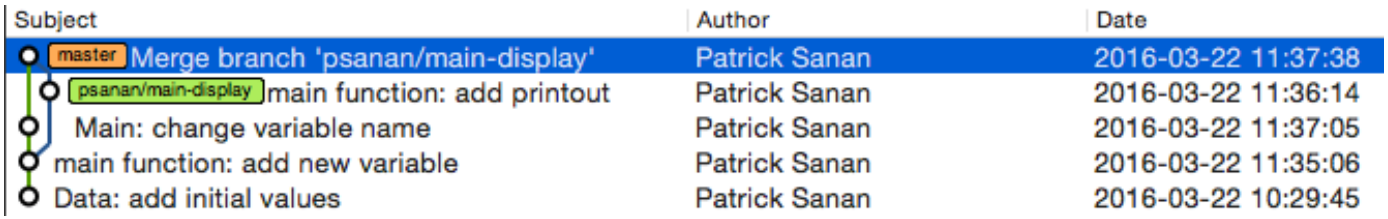
That's it!

Git is simply software that, at its core, does all the things we just mentioned, properly implemented:

- Defines data in a *repository*
- Records *snapshots* of a set of files
- Gives you a way to obtain a copy of the files, edit them, and add your changes as a new snapshot.
- Keeps track of *commits* for each snapshot: a summary, who made the changes, when, and from what previous snapshot.
- Keeps track of *branches*, which are pointers to commits. It provides a way to increment them, as new commits are added, and a way to merge them.
- Keeps track of references to *remote repositories* and provides a way to send and recieve repository data.

A Git Repository in Graphical Form

- Git repositories are commonly drawn as Directed Acyclic Graphs (DAGs) (or, imprecisely, "trees")
- A commit is a node and edges from its parent(s).
- A branch is a box which labels a commit
- `HEAD` is a label for a branch
- Merges are commits with exactly two parents
- Bitbucket, GitHub, GitLab, and GUI tools can draw these nicely for you



(Here, `HEAD` is marked by coloring the `master` branch)

Let's draw one on the board, and see what happens when we add new commits, including one which merges two branches.

Section 3

How do I use it?

Subsection 1

Basic Usage

Obtaining Git

There are many ways to obtain git.

- You can download a binary from git-scm.com
- If you are using Linux, you likely already have it (if not, try `sudo apt-get install git`).
- You can install git from Macports or Homebrew on OS X.

```
sudo port install git
brew install git
```

Setting Git Up

- To be able to keep track of who made which changes, git needs to know who you are.
- A way to do this is to tell git these things each time you log into a terminal, by calling the `git config` command. For instance, add commands to your `~/.bashrc` file.
- In addition to specifying your name and email address, I recommend you turn on colors and set your favorite text editor to edit commit messages.

```
git config --global user.name "Patrick Sanan"
git config --global user.email "patrick.sanan@gmail.com"
git config --global color.status auto
git config --global color.branch auto
git config --global core.editor vim
```

Creating a Repository

- From now, we'll start working on a real example.
- I assume that I have a working `git` executable and that I have set up my login file to establish my identity.
- I create a new directory and create a new git repository there:

```
mkdir myDemoProject
cd myDemoProject
git init
```

- Data is created in the `.git` directory. Never change anything in this directory.
- (I'll be doing everything from the terminal here, but you can also use various GUI tools to work with git)

Adding and Tracking Changes to Files

- `git add` adds files to the staging area.
- `git commit` creates a new snapshot from staging area, creates a new commit, and updates the branch reference.
- `git status` lets you know the current state.
- Try the following (use your favorite editor instead of vim)

```
vim data.txt
git status
git add data.txt
git status
git commit
git log
```

- (If you only want a short commit message, you can use `git commit -m"Component: summary"`)

Writing Good Commit Messages

```
Component: summary

After a blank line, describe what you did.
This will be something read later on by you,
and by other people trying to figure out what
broke their code. If you did something that could
cause problems for someone, note it here. It's
also a good idea to wrap the lines yourself.
```

What's in a commit?

- For basic usage, anything you changed since the last time your code worked.
- For work in teams or on larger projects,
  - Changes related to a particular task on a particular component
  - Something that is reasonably atomic (can't obviously be broken down)
  - Something which won't interfere with other people's work without cause

Branching

- By default you are on a branch called `master`
- Create branches with `git branch new-branch-name`
- A good way to name branches is `yourname/component-description`.
- *Check out* a branch with `git checkout`: update `HEAD` to point to the branch, update the working directory to the snapshot indicated by the branch.
-

```
git branch psanan/data-reorganize
git checkout psanan/data-reorganize
vim data.txt
git add data.txt
git commit
git log
git checkout master
vim data.txt
git log
```

- Common mistake: committing onto the wrong branch. Check with `git branch` (or better yet use the git prompt mentioned later)

Merging and Resolving Merge Conflicts

- To merge another branch into your current branch (`HEAD`), use
  `git merge <branch-to-merge>`
- Recall that a merge commit is like a normal commit, but with two parents.
- When you merge two branches and git cannot figure out how to merge the changes, you will have to do some work. This can be frustrating.
- Files will be specially flagged and annotated with special markers:

```
<<<<<<< HEAD
numIterations = 3;
=======
numIterations = 4;
>>>>>>> psanan/data-reorganize
```

- You must remove the special annotations and stage the file
- Once all flagged files are staged, commit as usual

Resolving Merge Conflicts

Let's perform an example with our simple repository.

```
git checkout master
git merge psanan/data-reorganize
git status
vim data.txt
git add data.txt
git status
git commit
git log
```

States of Files

It's worth reiterating the different states that files in your working directory can be in.

1. They can be untracked
2. They can be be ignored (if you create a special `.gitignore` file)
3. They can be unmodified from the current snapshot (determined by `HEAD`)
4. They can have unstaged changes
5. They can have staged changes (it is possible to stage parts of files)

`git status` will tell you about the states of files

The Most Important Commands

We have seen most of the most important commands

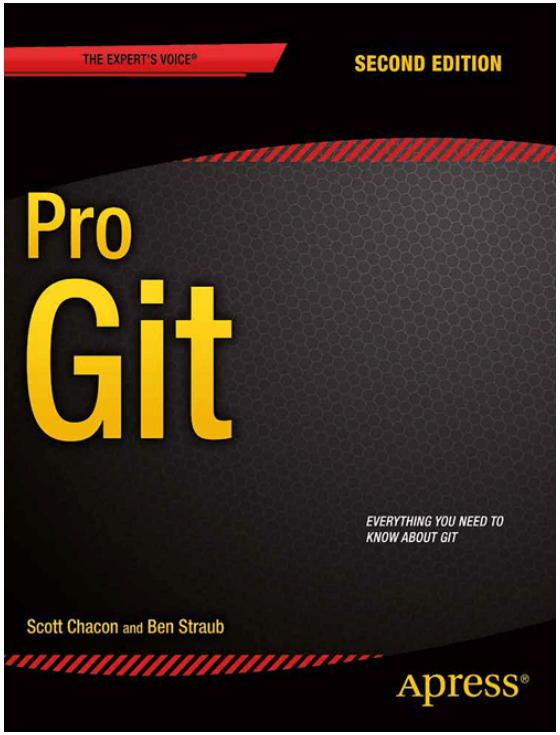
- `git add`
- `git commit`
- `git branch`
- `git checkout`
- `git merge`

Some more handy ones

- `git help [command]`
- `git diff [filename]` (what's changed?)
- `git log -10` (see last 10 commits)

For (Much) More

The Git Book : free at git-scm.com/book



Subsection 2

Remote Repositories and Tools

Getting a Copy of an Existing Project

- The most common way to start working on a git project
- Make a copy of a project with `git clone`, knowing its URL
- You only need to know the address of the repository. You often copy it from a Bitbucket/GitHub/GitLab page.

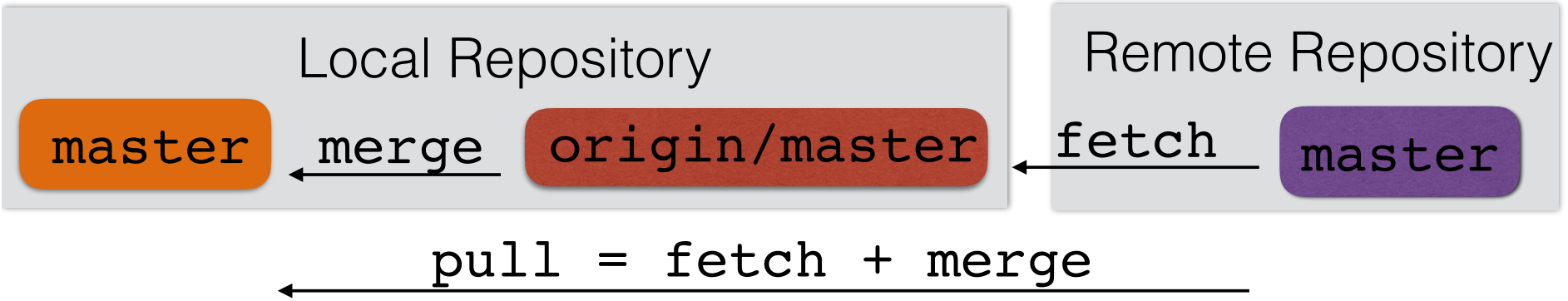
```
git clone https://bitbucket.org/psanan/example_bib_repo
```

- You may have to enter your username and password
- This creates a new local repository as a copy of the remote repository.
- To simply obtain code with git, this is all you need to know!

Remote Basics (1/2)

- You can define remote repositories (*remotes*) with `git remote add <name> <address>`.
- There are two types of addresses you can use, HTTPS and SSH. The latter will let you use an SSH key for password-free usage.
- When you use `git clone`, a remote is automatically created for you with the name `origin`.
- You can have references to copies of branches on remote repositories
- you can tell your local branches to *track* remote branches. The remote branch is referred to as the *upstream* branch.
- This is set up for you for the `master` branch when you use `git clone` to obtain a remote repository.

Remote Basics (2/2)



- `git fetch` updates all references from the upstream repository for the current branch.
- `git pull` calls `git fetch` and also merges the remote branch into yours.
- `git push` sends your local branch to be merged with the remote branch.
- `git push -u <remotename> <branchname>` is a shortcut to push a new branch to the remote and track it.

An Extremely Common Mistake

You keep local copies of remote branches in your local repository. For instance, with `git branch -a` you might see some branches like this

```
$ git branch -a
 * master
   remotes/origin/HEAD -> origin/master
   remotes/origin/master
```

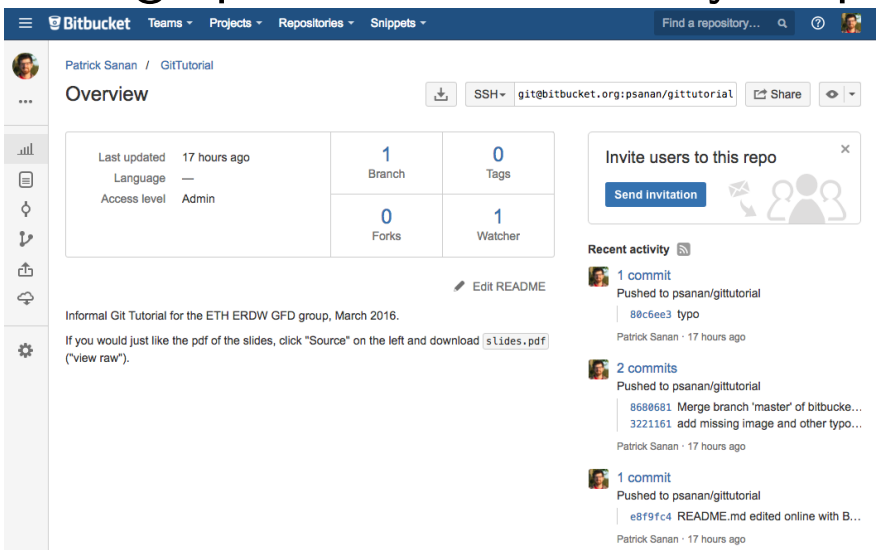
When you check out a branch that is tracking a remote branch, you will see something like this:

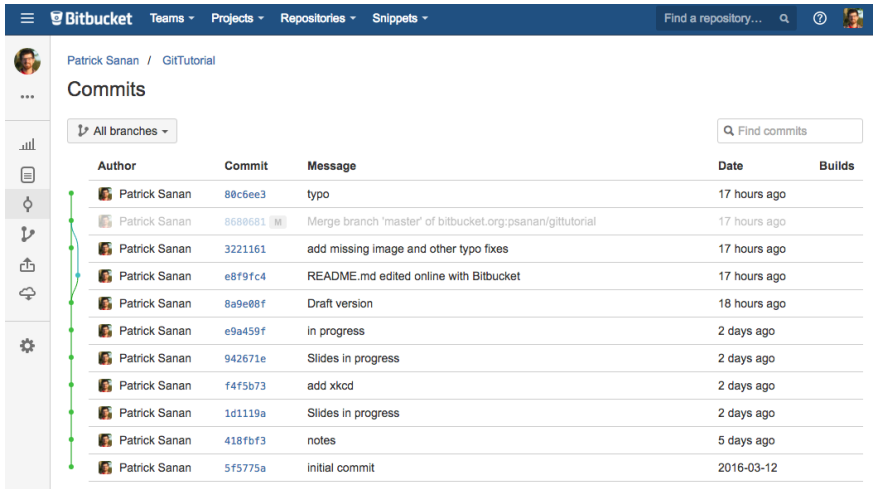
```
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

This does **NOT** mean that your branch is up to date with the branch on the remote repository! It means that your branch is up to date with the **local** copy of that branch, named `origin/master` here. If you want to make sure you are really up to date, you must call `git fetch` to update your local references.

Web-based repository hosting and tools: Bitbucket, GitHub, and GitLab

- These (freemium) services host remote git repositories for you.
- These repositories are essentially the same as your local ones.
- In addition, they provide additional workflow-management functionality through their web interfaces.
- Nice graphical overviews of your project and interactive web tools.

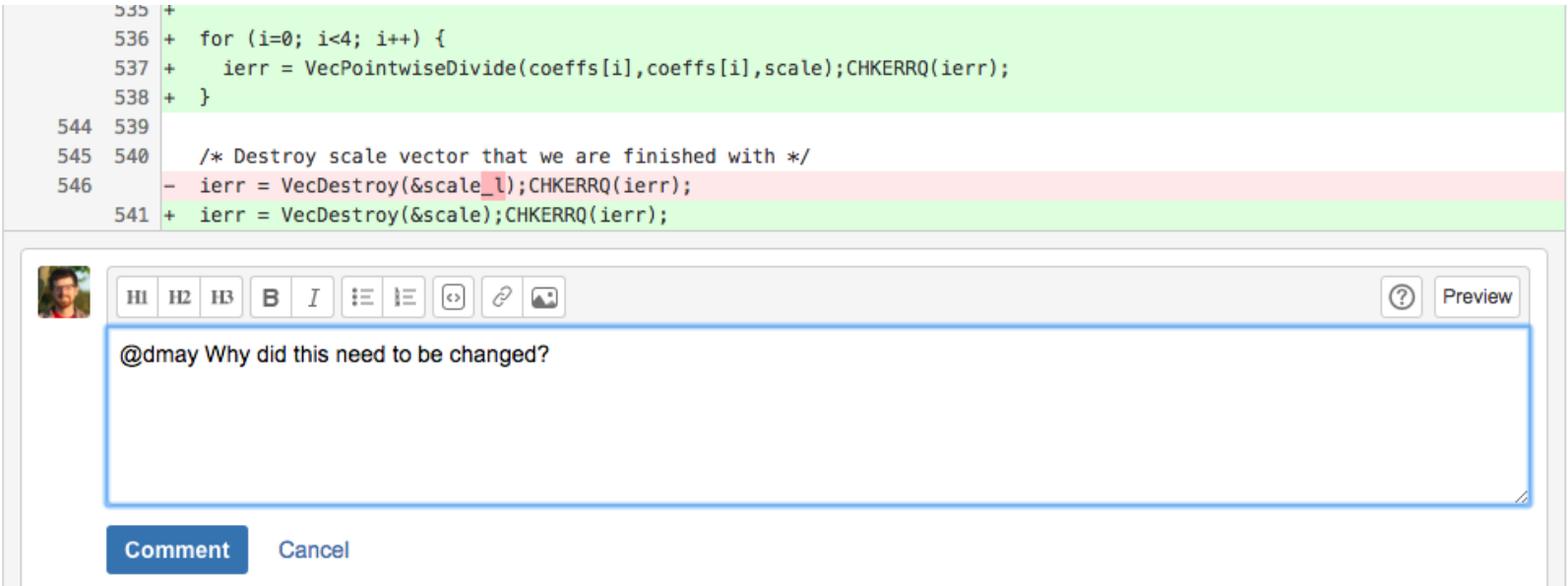




- Lots of documentation, very stable, many many users.

Web-based tools: key additional features

- Web-based visualization of source code and git history
- Commenting
- Pull Requests (PRs) or Merge Request (MRs) to formalize the process of asking one branch to be merged into another
- Interfaces to other services, notable testing and continuous integration (CI) tools
- "Social" features (particularly on GitHub)
- Let you set up SSH keys for passwordless interaction with the remote (highly recommended)



GitHub vs. GitLab vs. Bitbucket?

- Very similar in most ways
- Keep in mind: you control the material in your actual git repository, and you can very easily move it. Additional information (wikis, issue trackers, etc.) is at the mercy the companies that run these websites.
  - GitHub
    - targets open-source projects and communities
    - will give you perks for being an academic (you have to apply)
    - is owned by Microsoft
  - Bitbucket
    - focuses on professional teams, with fewer "social" features.
    - will give you perks for being an academic (use your educational email address)
    - is owned by Atlassian
  - GitLab
    - also allows institutions to set up private web-based services
    - is owned by GitLab, Inc.

Putting your Project on GitHub, Bitbucket, or GitLab

- With the website interface, create a git repository on the remote server
- Tell your local repository about the address

```
git remote add origin git@bitbucket.org:psanan/myproject.git
```

- Tell some or all your local branches to track new remote branches on this remote server

```
git push -u origin master
git push -u origin --all
```

Let's do this now for our demo project.

Subsection 3

Demo: Joining a project

- Fortunately, the workflow for joining an existing project hosted on a service like GitHub/Bitbucket/GitLab is very simple!
- To fully "play along" with the example, you'll need to sign up for an account on BitBucket (use your academic email address if possible)
- Let me know your BitBucket username, so I can give you access to a repository.

Let's perform a typical example, collaborating on a repository to which you have write access (this could be a private repository).

```
git clone https://bitbucket.org/psanan/example_bib_repo
cd example_bib_repo
git branch myname/new-reference     # change "myname" to your username
git checkout myname/new-reference   # change "myname" to your username
vim references.bib                  # or otherwise edit. Add your
    favorite paper!
git add references.bib
git status
git commit
git push -u origin myname/new-reference  # change "myname". You need
    write access!
# (Wait for a change on the master branch)
git fetch
git checkout master
git status
git pull
git checkout myname/new-references   # change "myname"
git merge master
git push
```

Aside: What if I don't have write acces?

- What if you don't have write access, but want to contribute to a repository?
- Simple - you create a *fork*.
- A fork is simply another copy of a repository, which you control
- GitHub/GitLab/BitBucket give you a button on their web interfaces to do this automatically.
- GitHub/GitLab/BitBucket's pull request / merge requests can be made between different forks.

Section 4

More selling points

Selling Points

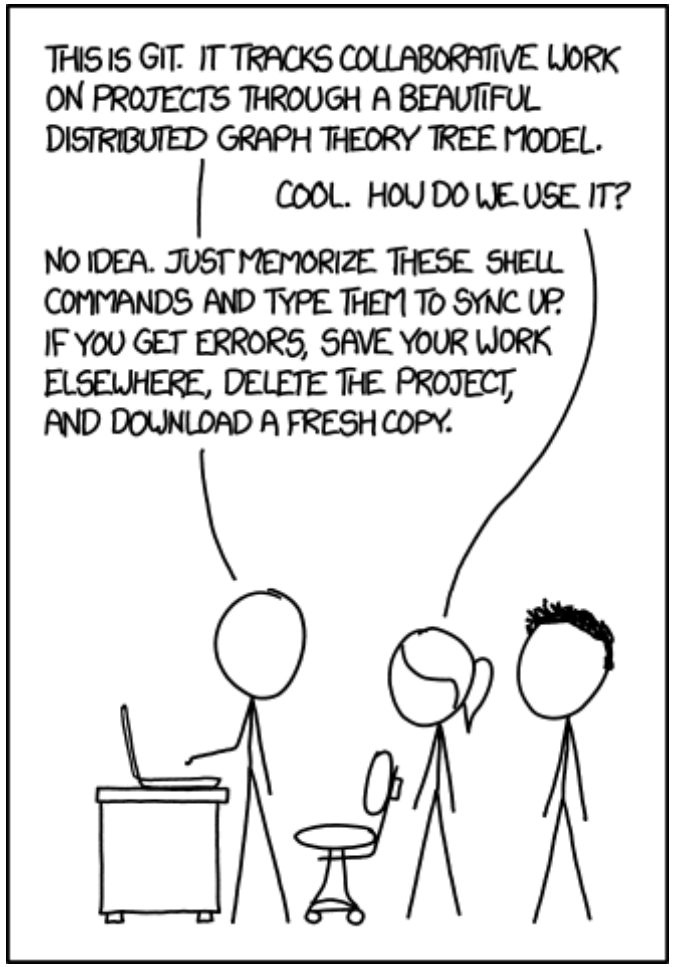
- A way to accidentally lose a lot less data
- A handy way of dealing with code which you commonly use on multiple machines (laptop, desktop, local cluster, supercomputer)
- A way to have a canonical version of code (on GitHub or Bitbucket)
- Some added peace of mind that your code is backed up
- An easy way to share your project
- An easy way to join or use other projects
- A nice way to organize your work
- A good way to efficiently work with remote collaborators
- A good way to uniquely identify versions of your code used in publications or experiments

Section 5

Annoyances and Solutions

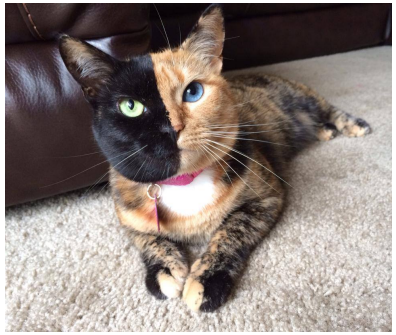
How do I remember all these stupid commands?!

- `git help`
- `git status` gives you hints
- `git help [command]` gives you the arbitrary sub-commands and flags
- A GUI tool can be helpful.
  - Built-in:
    - `git gui` (construct commits)
    - `gitk` (see the graph)
  - Many other options: http://git-scm.com/downloads/guis



http://xkcd.com/1597/

Merging is terrible!!

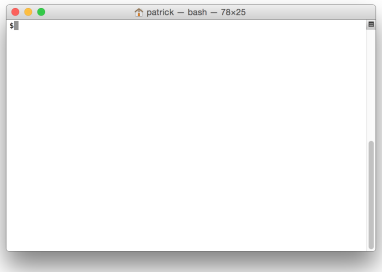


- Yes. It's fundamentally difficult. Often it will "just work" with git.
- Git cannot make many assumptions about your data.
- Try to make contained commits so that conflicts are localized.
- Don't let branches diverge. If you must work on a branch based on `master` for a long time, periodically merge `master` in.

```
git checkout psanan/my-complicated-feature
[ ... work work commit commit ..]
git checkout master
git pull
git checkout psanan/my-complicated-feature
git merge master
[ .. work work commit commit ..]
```

- `git status` will give you hints (like how to abort a merge).

Where am I?? What's going on??



???

- Turn on colors (see setup earlier).
- Remember useful git commands:

```
git status
git diff [filename]
git fetch
git log -10
git branch -avv # "git help branch" for the options
git remote -vv
```

- Use a terminal prompt which shows git status.
- Visualize your project with a local GUI tool or on a web-based service.

Uncommitted changes, need to work on another branch

- Option 0: finish your current commit
- Option 1: clone a new copy of the code

```
cd ..
git clone git@github.com:psanan/myproject.git myproject-copy
cd myproject-copy
git checkout psanan/some-other-branch
```

- Option 2: Stash the changes with `git stash`

```
git stash
git checkout psanan/some-other-branch
...
git checkout psanan/my-working-branch
git stash apply
```

angerous because you need to remember where you were.

- Option 3: Throw away all your changes with `git reset --hard HEAD` (**very dangerous: this deletes data!**)

The End

For More Help:

- This presentation: github.com/psanan/git_tutorial
- The Git Book: git-scm.com/book
- Myriad tutorials and cheat sheets on the web
- Beware StackOverflow: do not panic and start pasting mysterious commands
- If confused, take a moment to think about what you want to do in terms of
  - The DAG (The graph describing commits and their relationships)
  - Local vs. remote repositories

"... git actually has a simple design, with stable and reasonably well-documented data structures. In fact, I'm a huge proponent of designing your code around the data, rather than the other way around, and I think it's one of the reasons git has been fairly successful ... I will, in fact, claim that the difference between a bad programmer and a good one is whether [he or she] considers [his or her] code or [his or her] data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

–Linus Torvalds, 2006