Git

CSCI 5828: Foundations of Software Engineering
Lecture 03

Lecture Goals

Present a brief introduction to git

You will need to know git to work on your homeworks and essays

Git and GitHub

We are asking that all essays this semester be uploaded to GitHub -

- That means you need to be comfortable with the following technologies -
  - Git
  - GitHub
  - Markdown
- You should also be comfortable with
  - HTML5 and CSS (if you want to deliver your presentation via HTML)
- Advanced users can get more out of GitHub's website support if they know
  - Jekyll

We'll provide a brief intro to git...

git

git is a distributed version control system https://git-scm.com

git was developed by the linux community in 2005 to help manage the development of linux itself -

- As a result, it needed to solve the problem of how to do version control of software systems consisting of 10,000s of files with 100s-1000s of developers all working on the project at once

To install - head to the Downloads page of the site above, and follow the instructions

- on Mac OS X - first install homebrew - details at <http://brew.sh>
- then - brew install git

What is version control? (I)

Just briefly, version control is keeping track of changes made to the files that make up a software system

The concept of version can be applied to -

- a file -
  - Java file starts with some initial content, say a class
    - you check that in; that's version zero
  - You then add a method to the class and check that change in
    - that's version one

Most version control systems handle tracking the versions of files automatically; instead developers focus on changes to "sets of files"

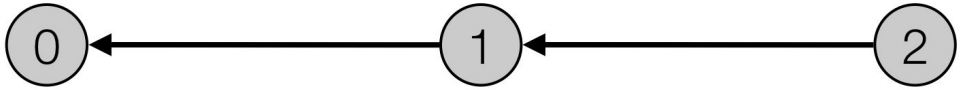
What is version control? (II)

The concept of version is thus typically applied to

- a set of files -
  - Your project starts with one source code file and a build script
    - you check that in; that’s version zero
  - You then add a second file, rename the original file, and modify the build script
    - you check in all three changes; that’s version one

A change to a set of files that gets checked in is called a “commit”

Versions form a line (called a “branch”)

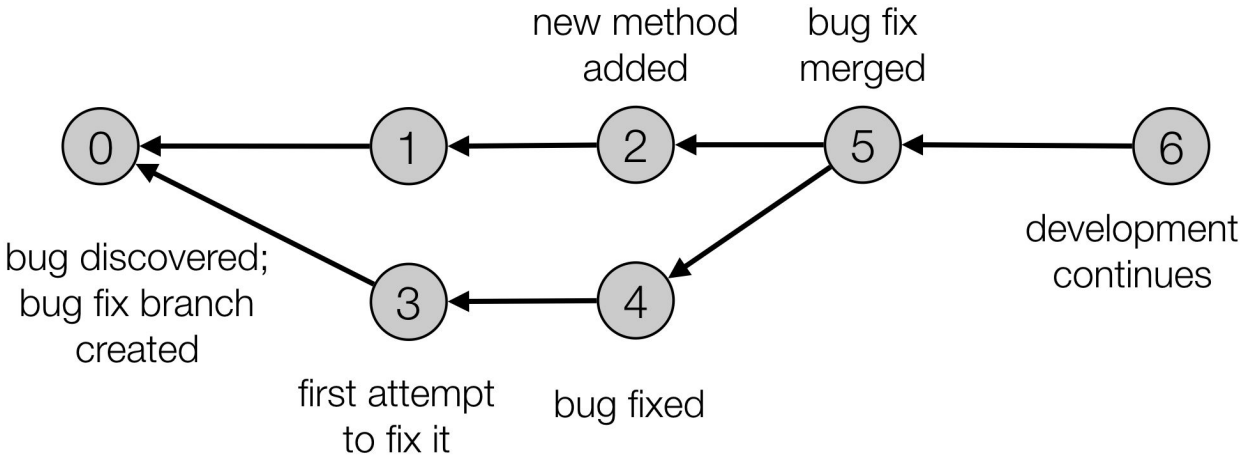


Repo created with one Java source file and a build script

New Java file added; original java file renamed; build script updated

Three new Java files added; resource directory created with three images; two previous Java files updated

Lines are called “branches” because that’s what they do



The **main** branch of development is known as the “master” branch or now **main** ;)
This is a convention. You could call it “frog-blast-the-vent-core” and your version control system wouldn’t care

Two Key Features of git (I)

git has two key features that enabled its success

Branches are quick and “cheap” to create

- In other version control systems, branches are “expensive” and hard to deal with; they discourage branch creation
- With git, you are encouraged to branch early and branch often
  - merges happen automatically most of the time
    - If there’s a conflict, git has a human sort it out

To master git, you need to become familiar and comfortable with branches and their associated operations

Two Key Features of git (II)

git has two key features that enabled its success

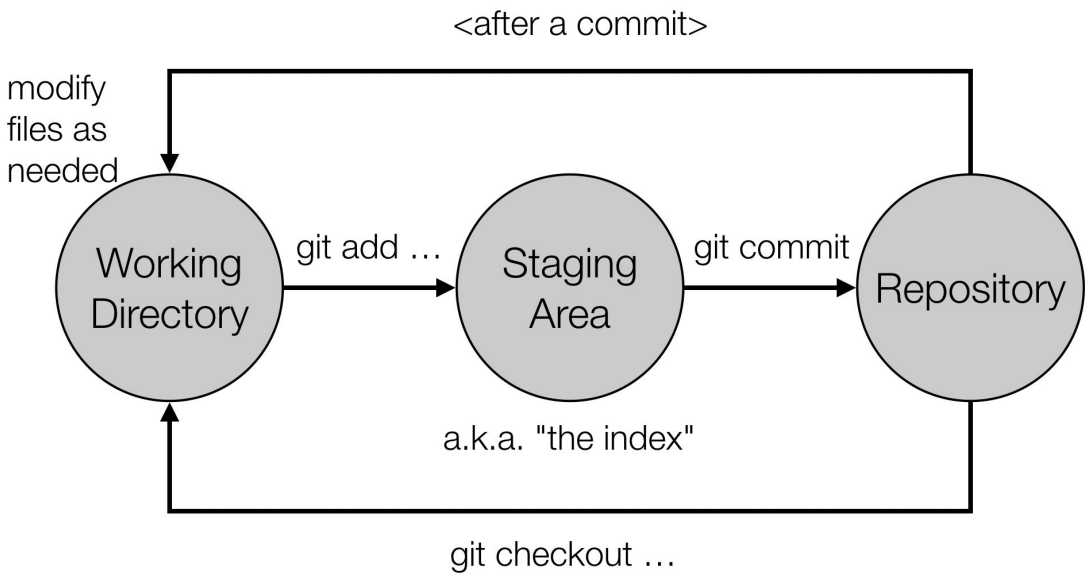
Every copy of the repository is “official”

- There is no one “centralized” repo that is the official one
  - each copy has the complete contents of the repo when it is first created
  - contents can drift, of course, as developers perform work on their local copies, but they can then be easily synchronized

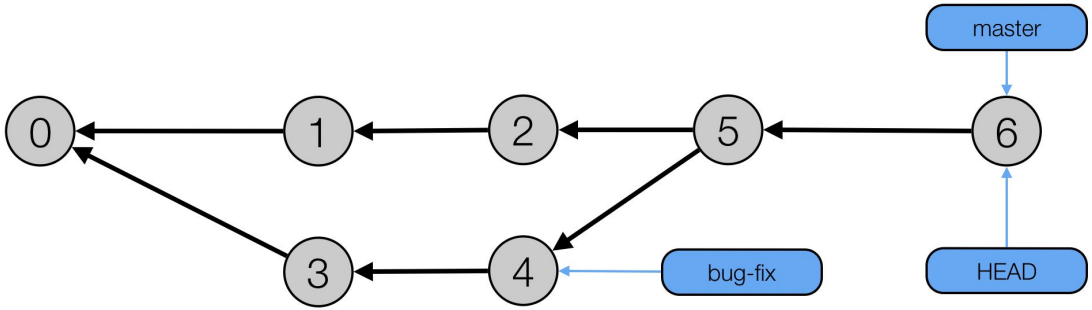
How a group synchronizes their repositories is left up to them

- because of this git supports a wide range of workflows that can support the work of 1 developer, or 5 developers, or even 1000s of developers. See https://git-scm.com/about/distributed for example workflows

Key Concepts to Understand git (I)



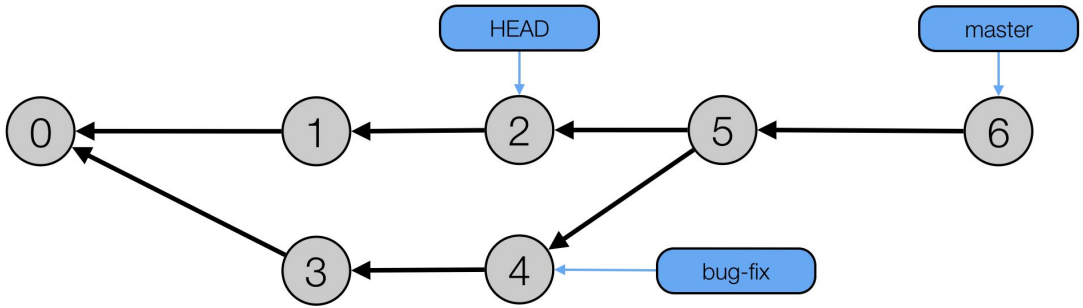
Key Concepts to Understand git (II)



Branches are just pointers; there is one default pointer called HEAD that (usually) just points at the latest commit of the current branch; the other branch pointers point at the latest commit of their respective branches

The diagram above shows a situation in which the user has checked out the master branch. The bug-fix branch points at its last commit and both HEAD and master happen to be pointing at the same thing.

Key Concepts to Understand git (II)



If I checkout a specific commit, say commit 2, then HEAD moves to point at that commit but master and bug-fix do not move.

I might do this if I wanted to then create a branch that used commit 2 as a starting point.

Common git Commands (I)

To create a new git repository

- git init

This command works in an empty directory or in an existing project directory

- It creates a .git subdirectory in that directory where it stores all of the information about the repository
- On Unix-based systems calling it “.git” makes the directory “invisible”

If you have a set of changes “staged”, you can commit them to the repository with the command

- git commit

An editor will open where you can type the log message for the commit

Common git Commands (II)

To view the current status of a repo

- git status

This will show what branch you are on, what changes there are in the working directory, and what changes have been staged

If you have a modified file or a new directory that you want to “stage”:

- git add <fileOrDirName>

If you want to get rid of a file or directory:

- git rm <fileOrDirName>

If you want to rename a file or directory:

- git mv <oldFileOrDirName> <newFileOrDirName>

Common git Commands (III)

To see the current branches -

- git branch

To create a new branch

- git checkout -b <newBranchName>

To switch to an existing branch

- git checkout <branchName>

To push commits to a remote repository

- git push

To retrieve commits from a remote repository

- git pull

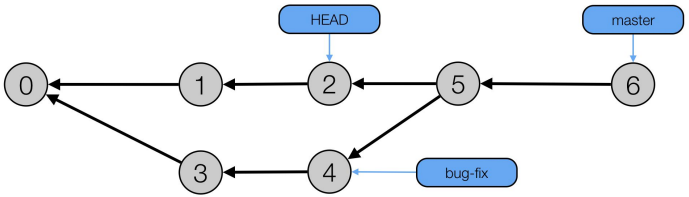
Example

Let’s step through a simple example that recreates the graph of slide 13

- Word of warning: git does not generate simple integers for its version numbers. Instead, commit version numbers look like this:
  - fa840cc7775b1d2daa51dd6e4b0c66384d3554e3

These “hashes” allow git to uniquely identify files and commits

- We don’t have time to cover the cool things that the use of these hashes enable for git; if you are curious, this book has a great explanation
  - Version Control with Git, 2nd Edition
  - by Jon Loeliger, Matthew McCullough
  - Publisher: O’Reilly; 2012



Step 1: Create Repo

Create a new directory: example_project

- mkdir example_project

Enter that directory and create a file called README.md

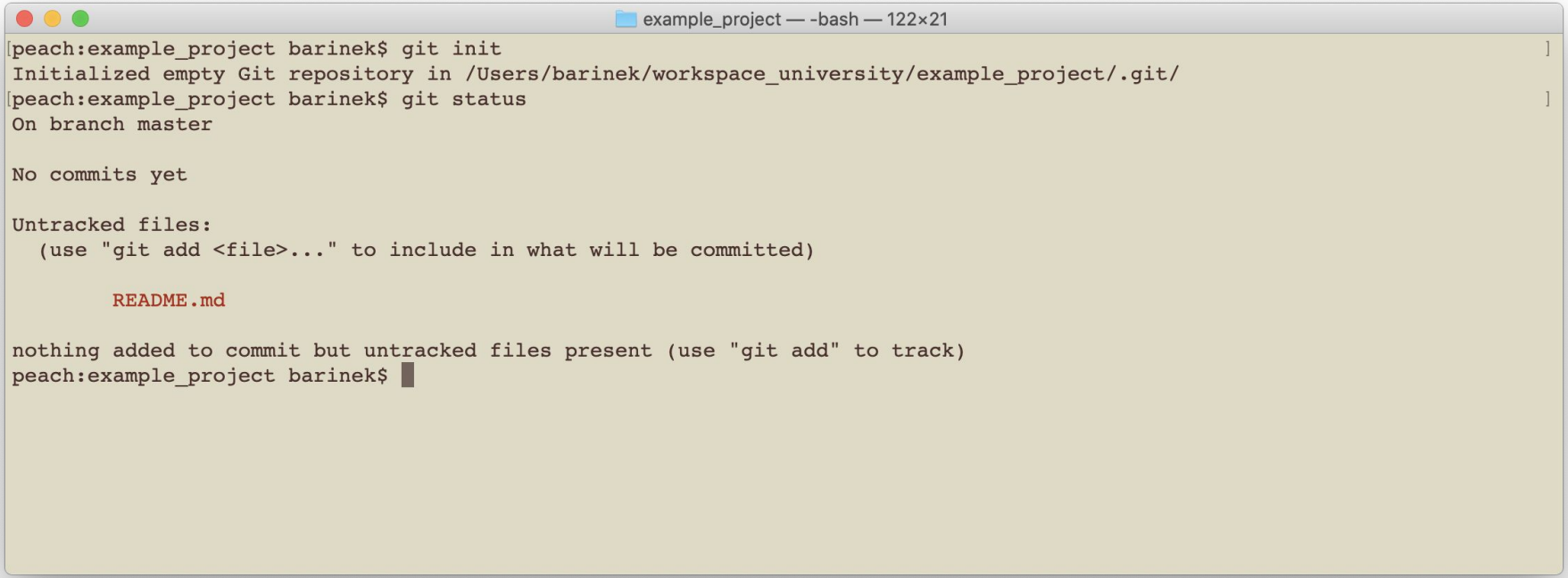
- cd example_project; vi README.md

Edit the file to contain the following contents

Example Project

Save it. At the prompt, type “git init”

Current Status



We've created a repository but we haven't committed anything to it. We have one file in our working directory that git knows nothing about. It refers to that file as being "untracked".

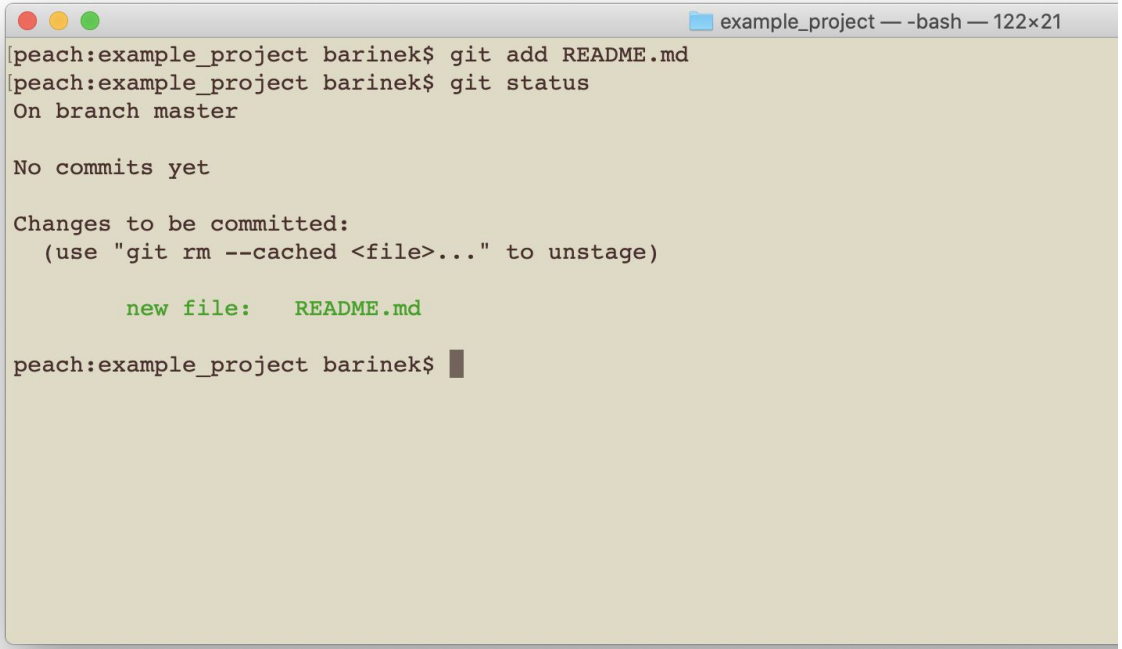
Step 2: Track a file

Let’s tell git to track this file

- git add README.md

Check the status

- git status



Step 3: Submit First Commit

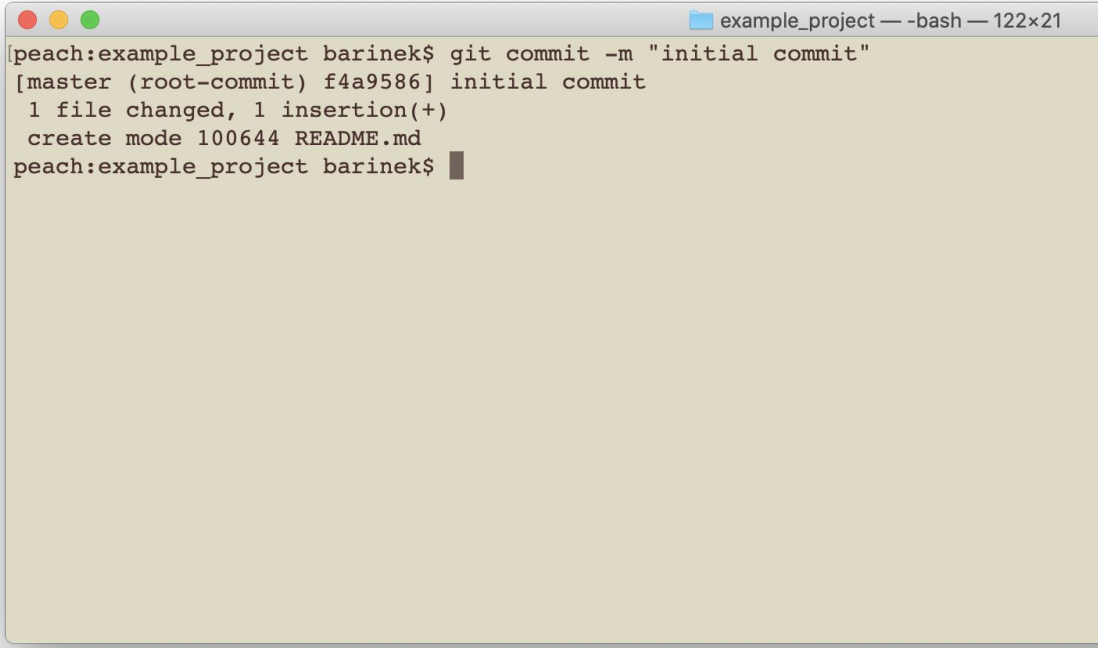
Let’s commit this file to the repository

- git commit -m “Initial Commit”

Our change to the repository was “saved”.

It’s now permanent.

Our working directory is back to being “clean” that is unchanged from the current commit.



How do we see our commit? (I)

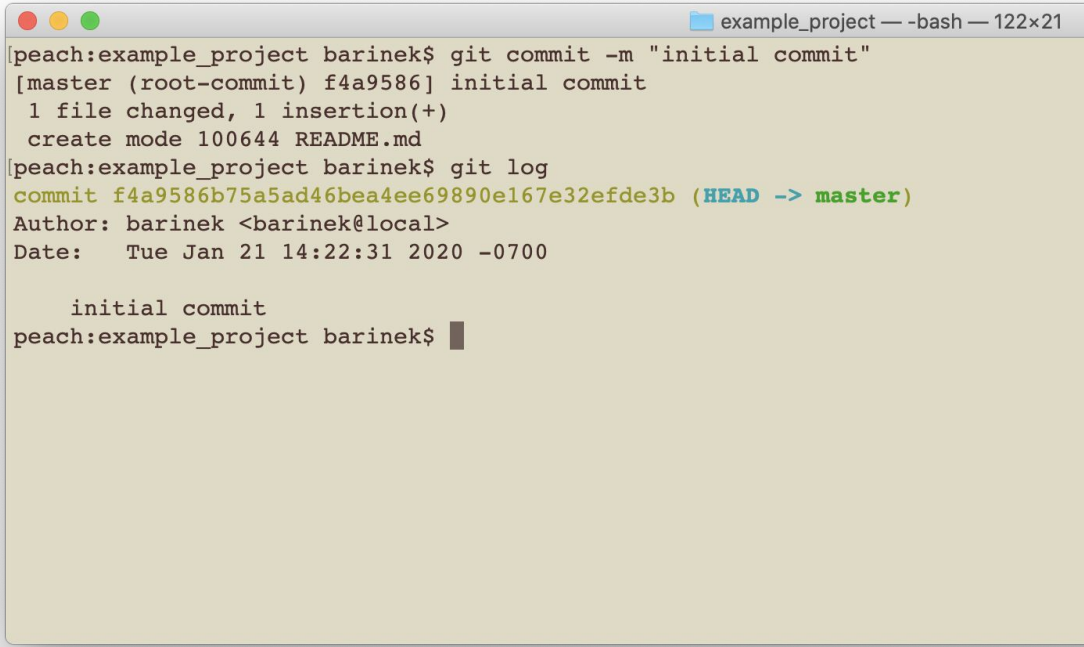
git log

// show a list of commits made
to the repository

From this we see that our commit's "name" is "f4a..."

- {hash}

And, it's log message was "initial commit"

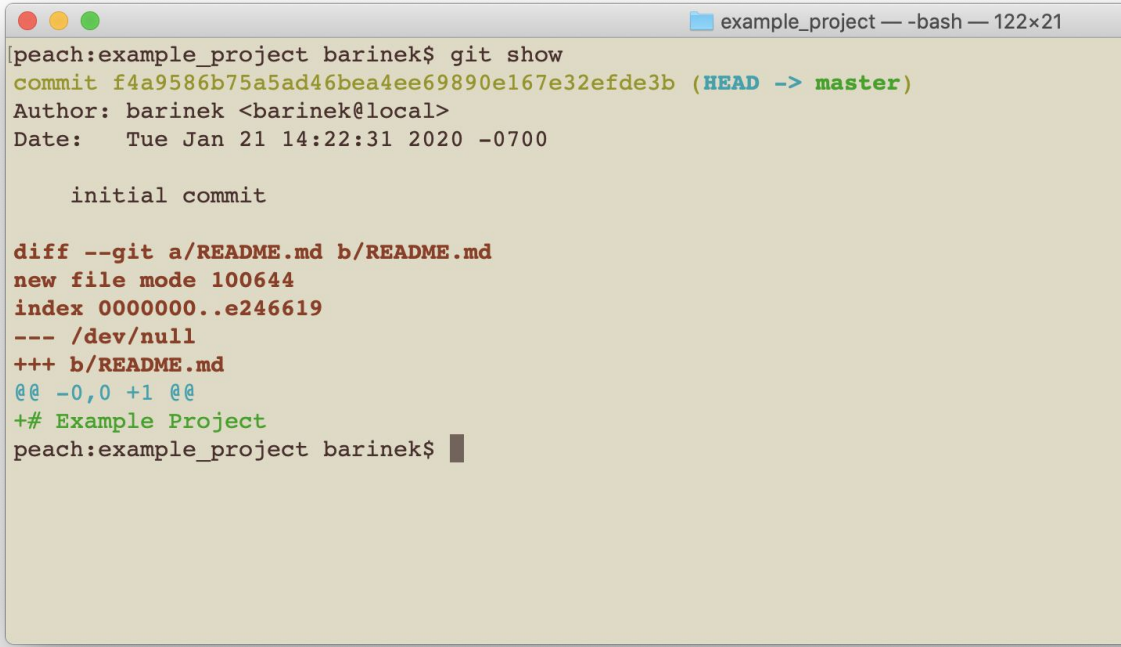


Note: "git log" is infinitely customizable; see "man git-log" for details

How do we see our commit? (II)

git show

// show info on the most recent commit

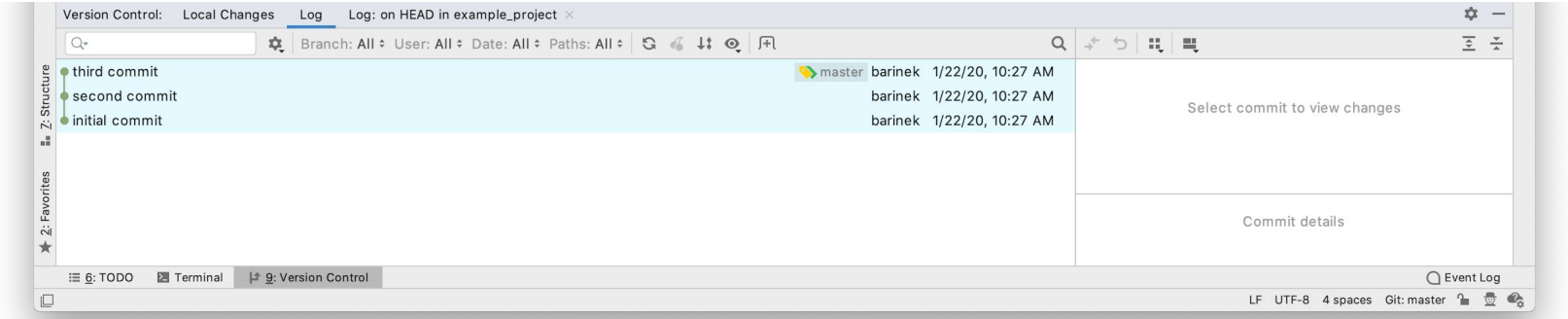


Step 4: Add Commits

Create two more commits on the master branch

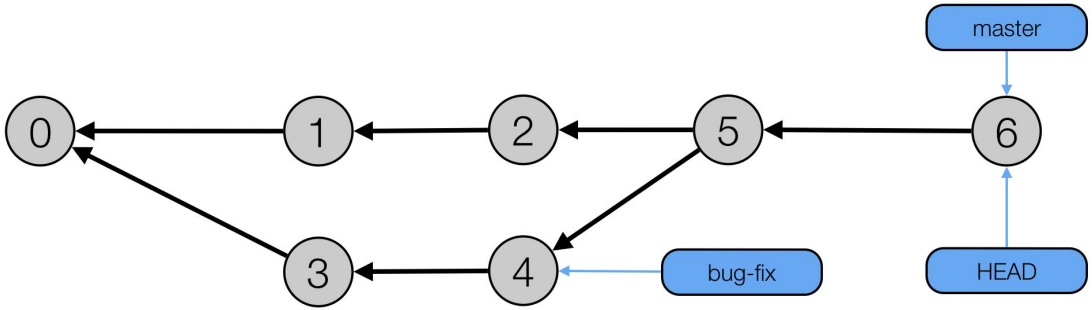
- Edit README.md to contain a new line: “First Edit”
- Save it. git add; git commit -m “Second commit”
- Edit README.md to contain a new line: “Second Edit”
- Save it. git add; git commit -m “Third commit”

Our version tree now looks like this -



Reminder

Recall that we are trying to create this graph



So, it’s time to create a branch and add commits “3” and “4”. (We currently have commits “0”, “1”, and “2”.

Step 5: Create a Branch (I)

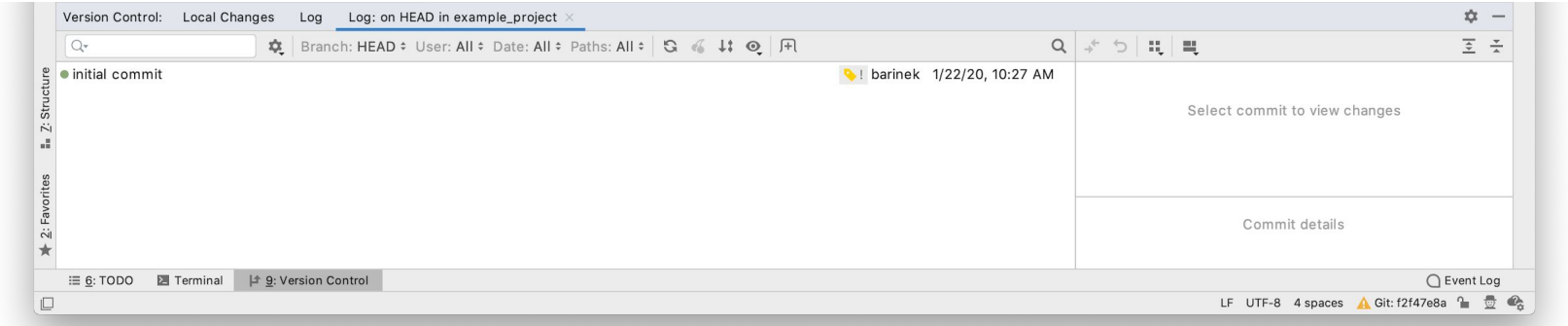
We want to create a branch off the very first commit

- As a result, we need to jump back to it

We do that with the checkout command

- git checkout {git hash}

New version tree - check README.md... our two edits are gone!

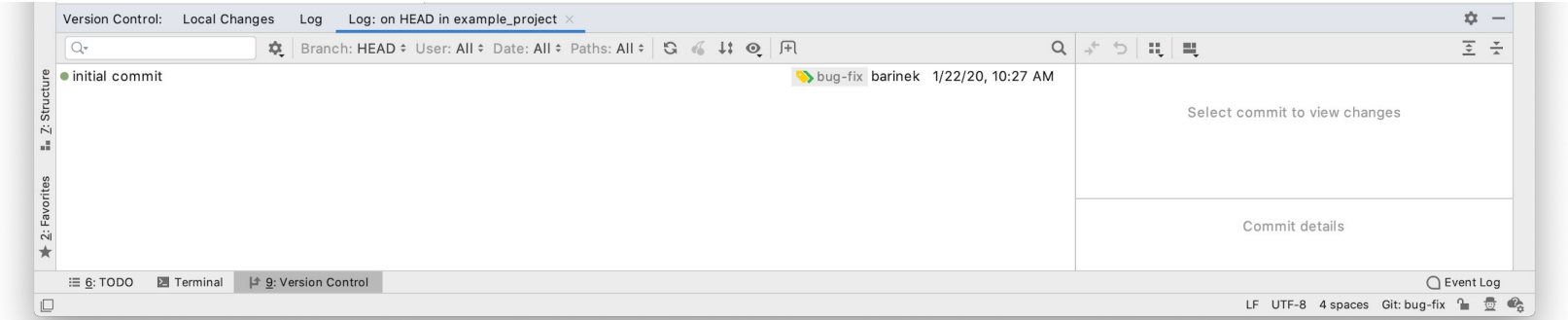


Step 5: Create a Branch (II)

Create the bug-fix branch

- git checkout -b bug-fix

The branch is created and conceptually the HEAD is on that branch; however, we don’t have a new commit (yet), so HEAD is really just pointing at our initial commit but any new commits will be added to the “bug-fix” branch

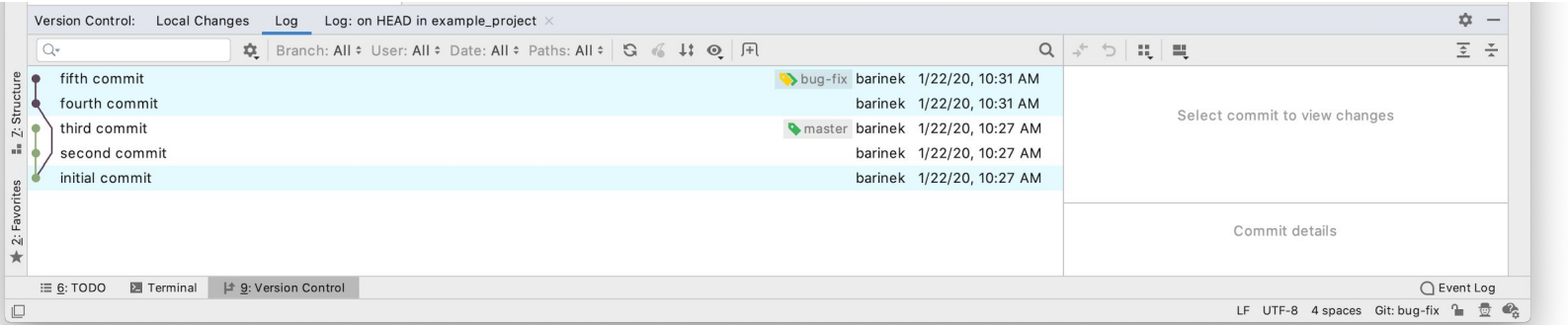


Step 6: Add Commits

Create two commits on the bug-fix branch

- Edit README.md to contain a new line: “Third Edit”
- Save it. git add; git commit -m “Fourth commit”
- Edit README.md to contain a new line: “Fourth Edit”
- Save it. git add; git commit -m “Fifth commit”

Our version tree now looks like this



Step 7: Time to merge (I)

Merging branches is easy

- and many times Git can perform the merge automatically
- If it cannot, you enter a “merge conflict” state and have to help git resolve the conflict

Let’s see what happens when we try to merge our “bug fix” changes into our master branch

The rules you need to know -

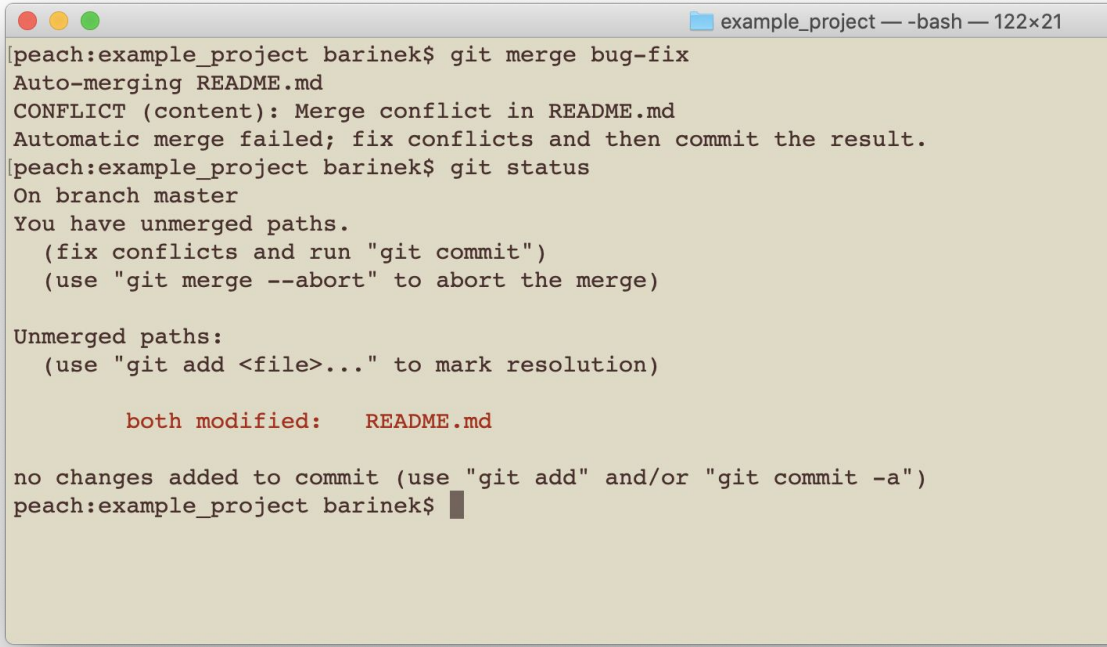
- checkout the branch that should receive the changes
- Invoke “git merge” and enter the name of the branch that has the changes to be merged

Step 7: Time to merge (II)

git checkout master

git merge bug-fix

CONFLICT: git can't figure out how to create a README.md file that contains all of our changes; (we edited the same two lines of the file)



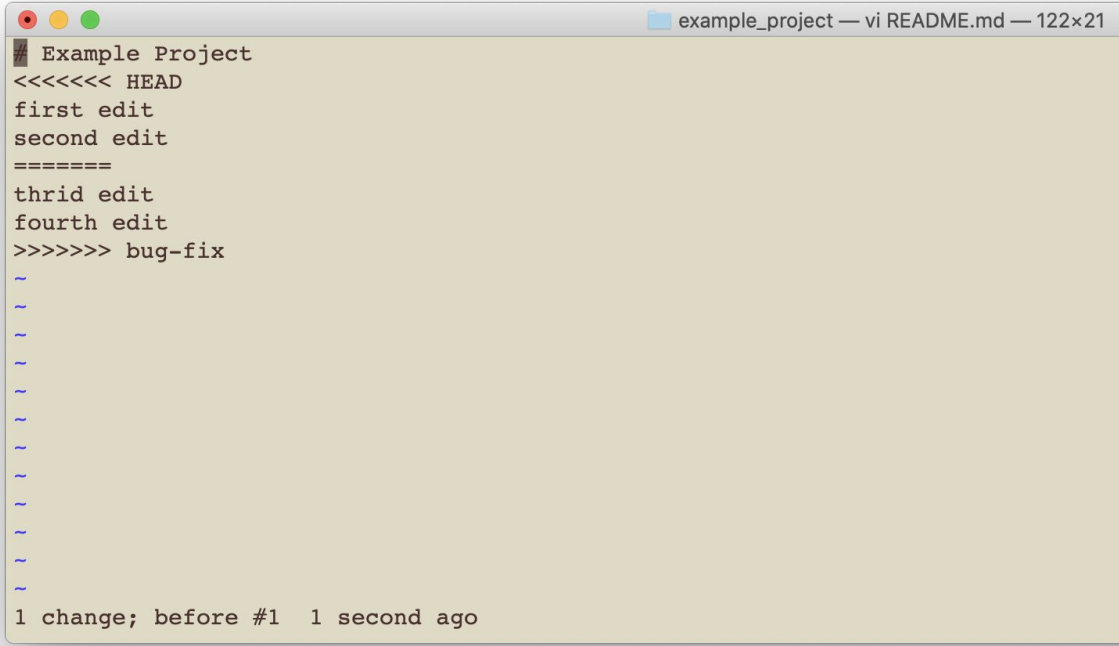
Step 8: Fix the Conflict

Take a look at the README.md file

Delete the lines that git added

git add README.md

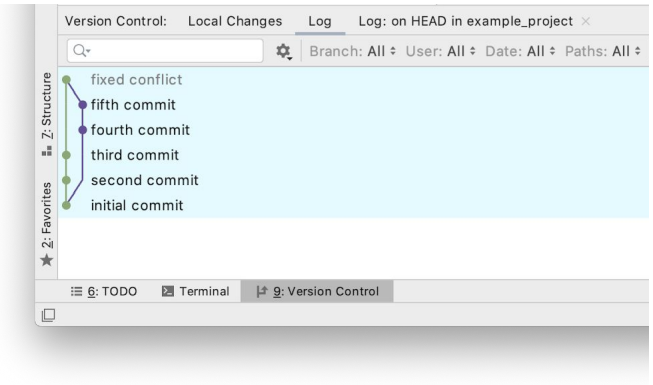
git commit -m “Fixed Conflict”

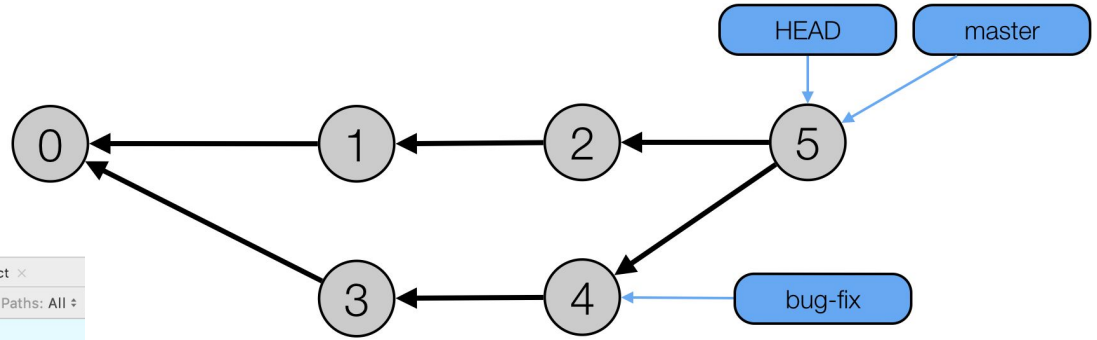


The Result?

Our current tree looks like the graph on the left. That corresponds to the graph below.

Only two things left to do to recreate the graph from slide 13.





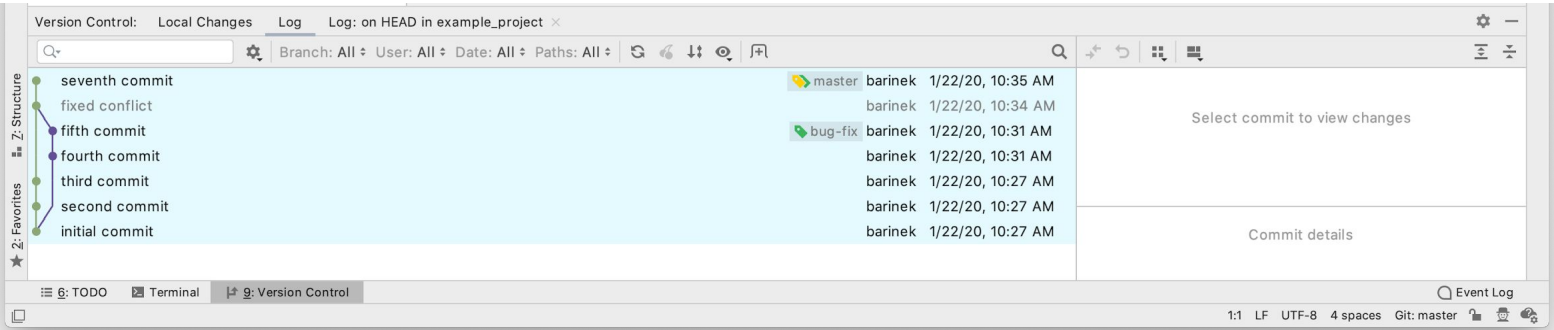
Step 9: Add a new commit

Add a new commit

- Edit README.md to contain a new line: “Fifth Edit”
- Save it. git add; git commit -m “Seventh commit”

The version tree now looks like this

One more thing to do...



Step 10: Move head to third commit

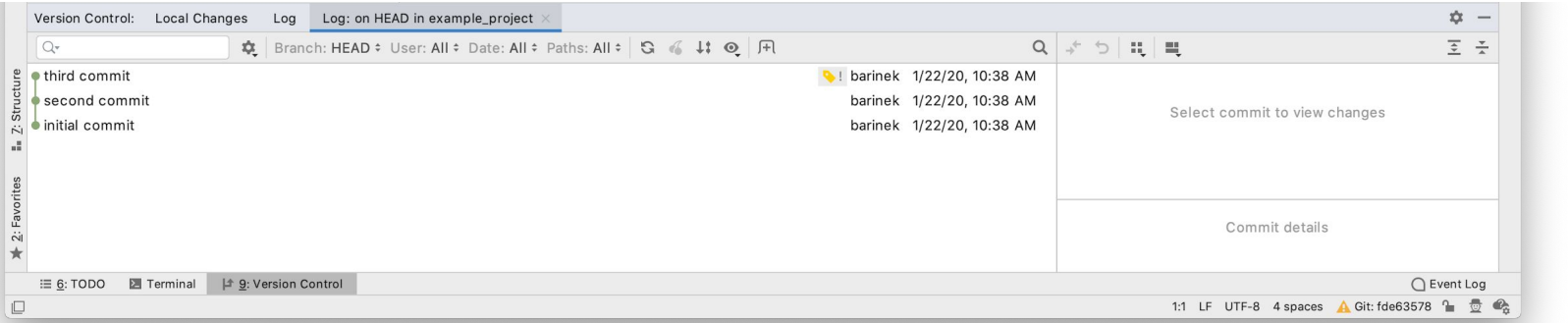
Use git log to discover the name of the third commit

- {git hash}

Check it out

- git checkout {git hash}

The version tree now looks like this. Follow all of these steps and convince yourself that the final state of the repo is equal to the graph we showed on slides 13 and 25.



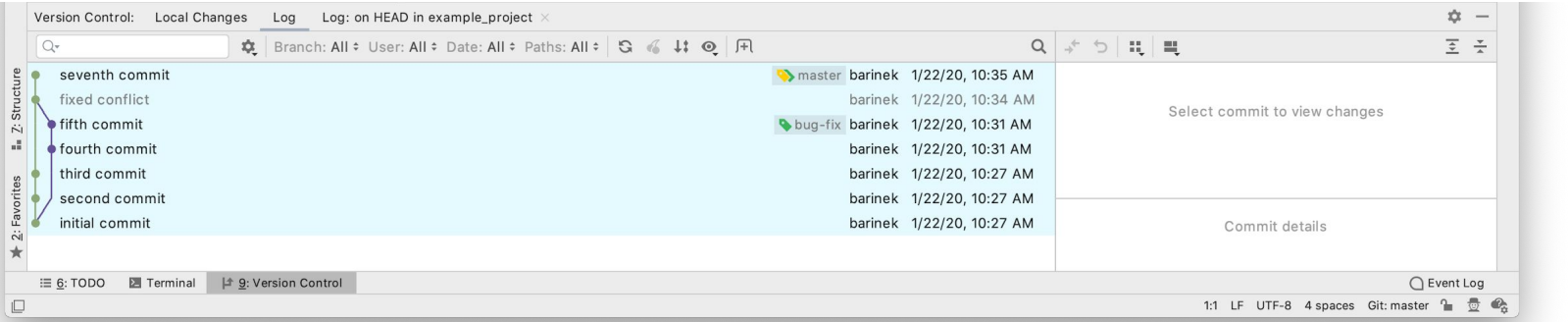
Step 11: Leave the repo in a good state

git checkout master

- Tell git to go back to the tip of the master branch
- git is now ready to receive new commits on that branch

Final State

Note: this is equivalent to the graph from slide 12



Only scratched the surface...

There is still a LOT to learn about git

- Be mindful, git is very powerful; people use it for all sorts of things!

We recommend

- Learn Version Control with Git from fournova, makers of Tower
- Become a git guru by Atlassian
  - See also: Getting Git Right by Atlassian
- Try Git by GitHub, Inc.

Questions?