VERSION CONTROL

How we all git along

Daniel van Berzon

David Richards

@CodebarBrighton

http://gitforthe.win

OCASTA

@ocastahq ocasta.com

We are a friendly bunch who build apps, web, and digital products that improve customer and employee engagement.

Look familiar?

myfile_version1

myfile_version2

myfile_version_FINAL

myfile_MARCH_2004

myfile_version_LAST_ONE_PROMISE

myfile_OLD_copy

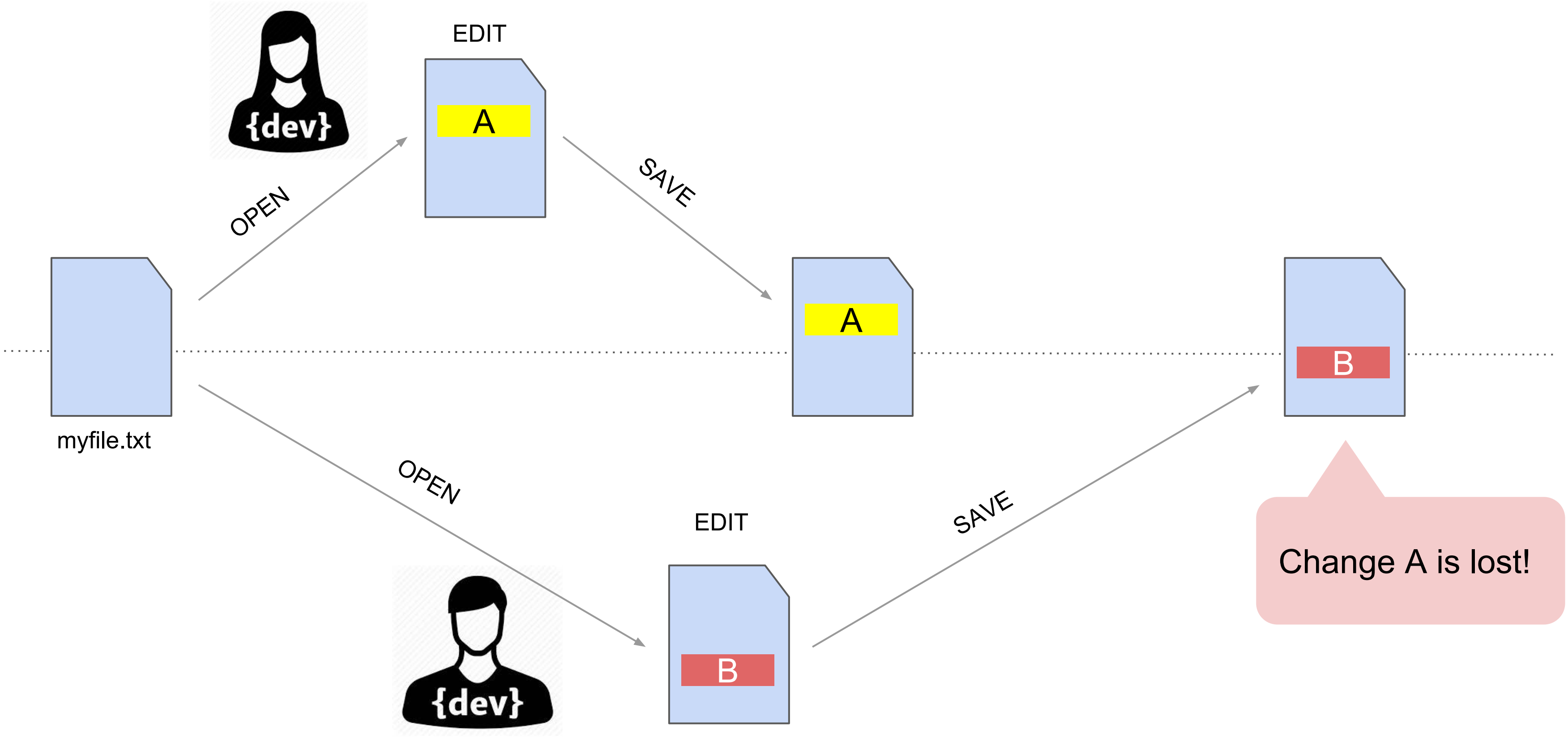
myfile_version29292

myfile_OLD!

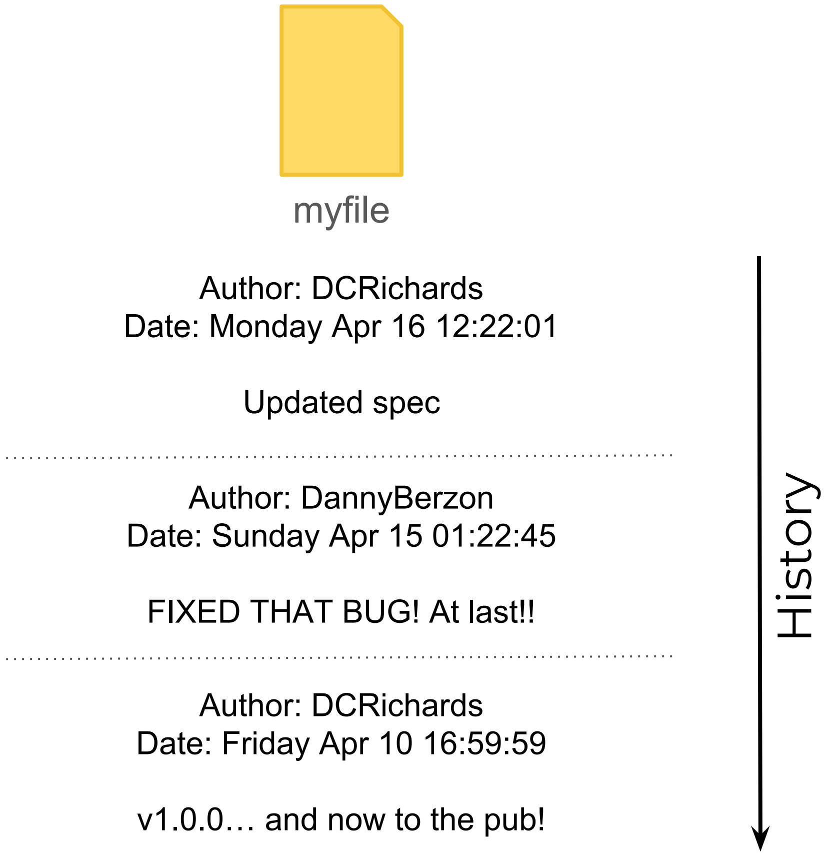
myfile_version2.1.1

myfile_version33

Collaboration?



A better way



Author: DCRichards
Date: Monday Apr 16 12:22:01

Updated spec

Author: DannyBerzon
Date: Sunday Apr 15 01:22:45

FIXED THAT BUG! At last!!

Author: DCRichards
Date: Friday Apr 10 16:59:59

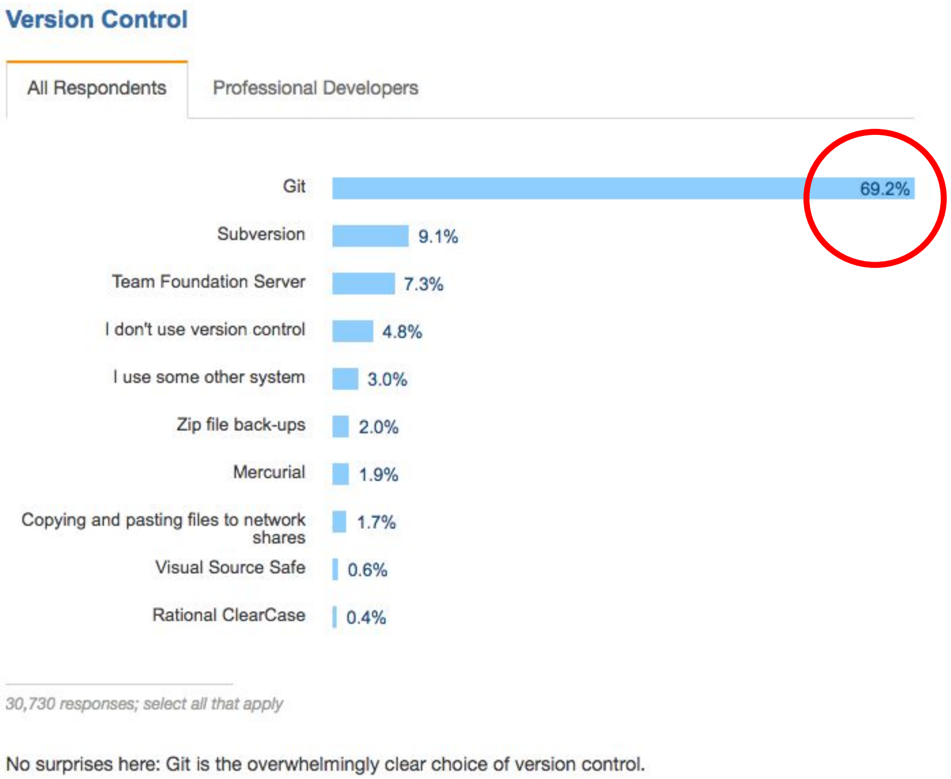
v1.0.0… and now to the pub!

What is Version Control?

- It allows you to keep track of changes to files in a project.
- You can travel in time and keep changes safe.
- It makes it ~~easier~~ possible to collaborate on projects and work together.

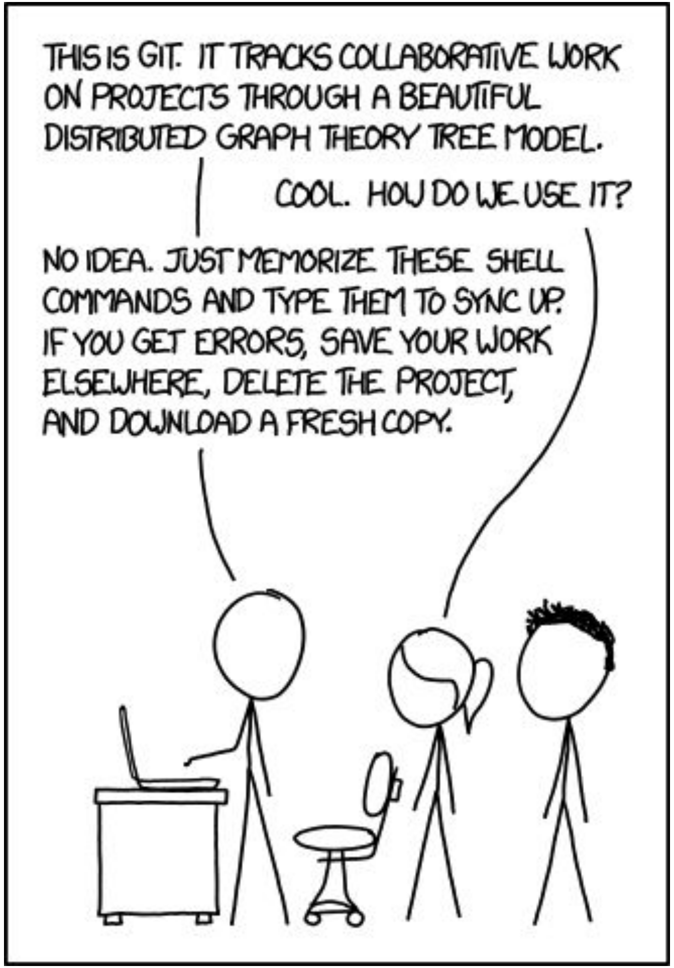
What is Git?

- A **Version Control System**.
- The industry standard system.
- The one you’re most likely to deal with.



30,730 responses; select all that apply

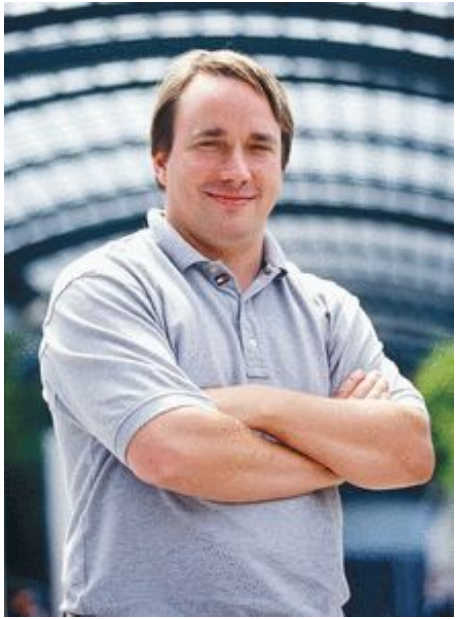
No surprises here: Git is the overwhelmingly clear choice of version control.


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

https://xkcd.com/1597/

A brief history

- Created by Linus Torvalds - creator of Linux.
- As with many great things in life, Git began with a bit of creative destruction and fiery controversy.
- Linux project originally used BitKeeper as its version control, when BitKeeper’s free licence was revoked, Torvalds set out to make his own.
- “I'm an egotistical bastard, and I name all my projects after myself. First Linux, now **Git**”.
- 4 days after development began, Git was used as the version control for Git 🔥.



The command line

If you've never used the command line before, don't be afraid. We'll be using the following format to show a command you can run:

```
> whoami

> ls -al
```

- Follow along with the slides by running these commands in your terminal
- To run a command, type in the part after the > and then press enter
- You can copy and paste them from: http://gitforthe.win/commands
- And please ask if you're stuck!

Let's Git started...

https://gitforthe.win/setup

Mac

- Open `Terminal`
  Applications/Utilities/Terminal

  ```
  > git --version
  ```

- If you see a prompt for XCode, accept it.

Windows

- Head to gitforwindows.org
- Follow the instructions on the link above

Linux... anyone?

```
> sudo apt install git-all
```

Configuration

- Now it's installed, let's open the `Terminal` on the mac, or `Git Bash` on Windows.
- Just one small thing. We need to tell git who we are:

```
> git config --global user.name "Your name"
> git config --global user.email "name@example.com"
```

- Sorry about that… now let's dive in.

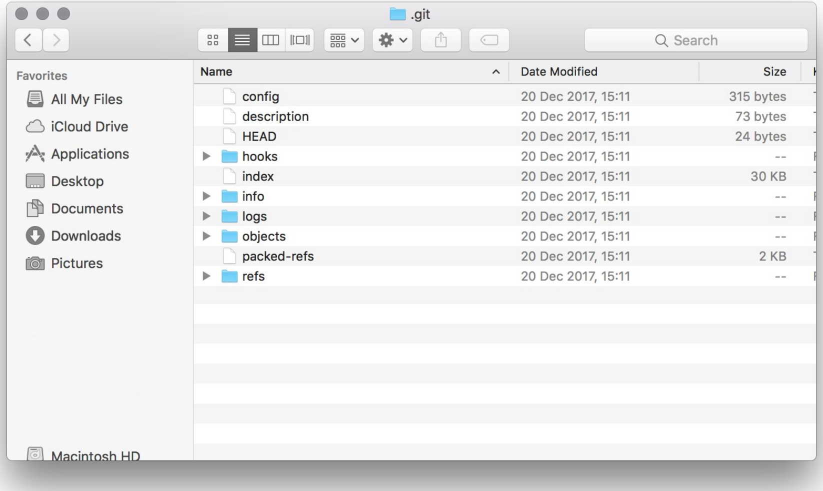
Git init

```
> cd
> cd "My Documents" Windows
> cd Documents      Mac
> mkdir my-repo
> cd my-repo
> git init
```

Git init

- You'll notice there's now a `.git` folder.
- This folder contains all of the information git needs to version control your code.

```
> ls -al .git
```



Name	Date Modified	Size
config	20 Dec 2017, 15:11	315 bytes
description	20 Dec 2017, 15:11	73 bytes
HEAD	20 Dec 2017, 15:11	24 bytes
hooks	20 Dec 2017, 15:11	--
index	20 Dec 2017, 15:11	30 KB
info	20 Dec 2017, 15:11	--
logs	20 Dec 2017, 15:11	--
objects	20 Dec 2017, 15:11	--
packed-refs	20 Dec 2017, 15:11	2 KB
refs	20 Dec 2017, 15:11	--

Our first command

```
> git status
```

- Shows the current state of changes.
- Showing both tracked and untracked files.

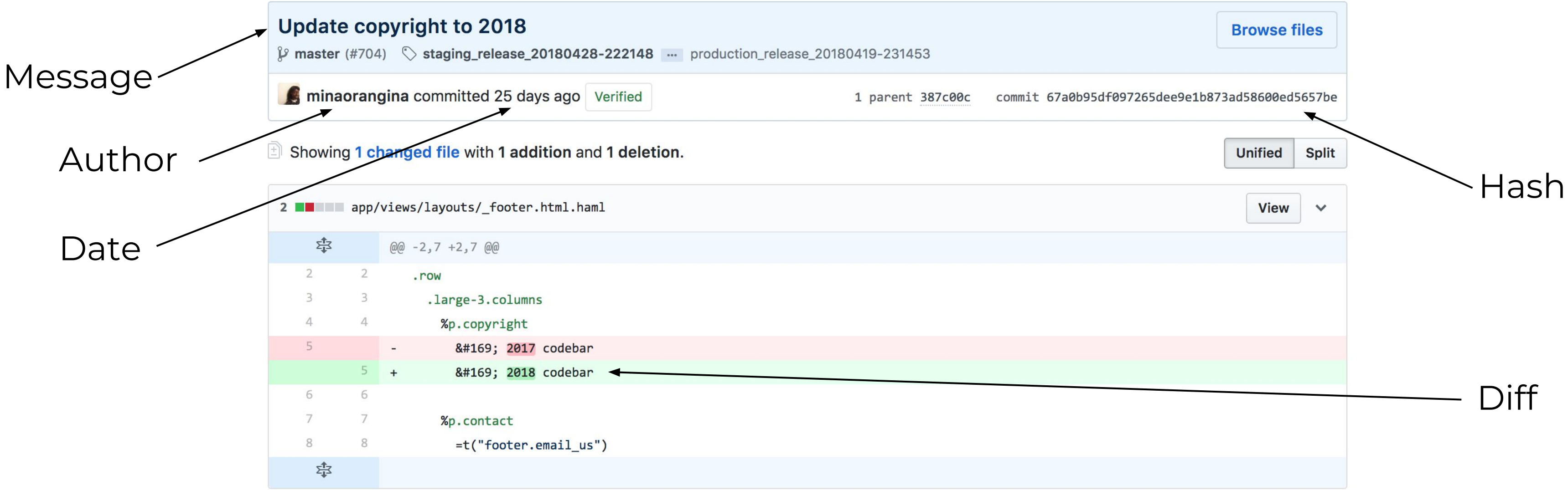
Commits

- A commit can be thought of as a snapshot of your files
- Commits contain a **hash**, **date, author, message** and set of changes since the last snapshot.

```
commit a7290cb122571bffa73613809012be19b6e4ccd0
Author: Hip Ster <hip@ster.io>
Date:   Tue May 1 12:24:40 2018 +0100

    Re-write with React Native 3.0-alpha1
```

Commits

- Files can be **added**, **modified**, **deleted** or **renamed**.
- If a file is modified and committed, git stores the new version of the file
- For most files, Git is aware of the line-by-line changes. This is called the <u>diff</u>



Message

Author

Date

Hash

Diff

Update copyright to 2018

master (#704) staging_release_20180428-222148 ··· production_release_20180419-231453

minaorangina committed 25 days ago Verified

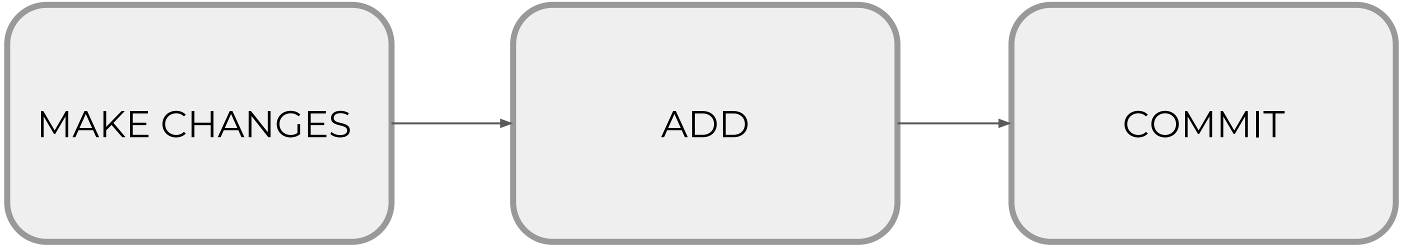
1 parent 387c00c commit 67a0b95df097265dee9e1b873ad58600ed5657be

Showing 1 changed file with 1 addition and 1 deletion.

2 app/views/layouts/_footer.html.haml

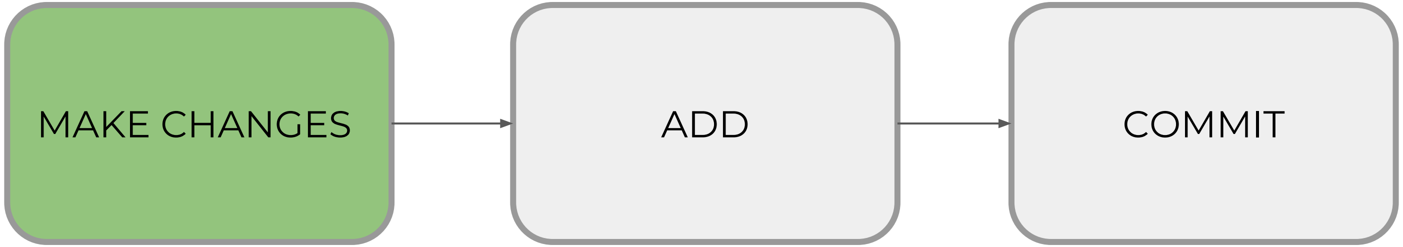
```
@@ -2,7 +2,7 @@
   .row
     .large-3.columns
       %p.copyright
-        &#169; 2017 codebar
+        &#169; 2018 codebar

       %p.contact
         =t("footer.email_us")
```

Simplest workflow



Create a new file

- Create a new file **README.txt** in your **my-repo** folder and add some text to it.



MAKE CHANGES → ADD → COMMIT

Git add

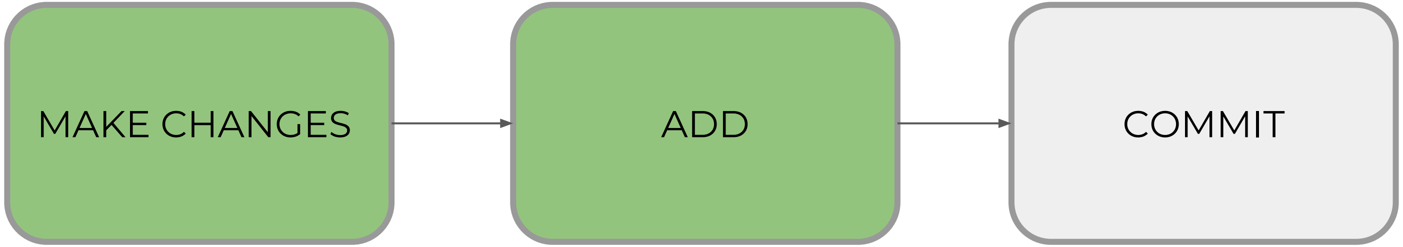
- We need to tell git which files we want to commit. This is done with `add`.
- This adds files to the index. We'll look at this next.

```
> git add README.txt
> git status
```

The index

- You can think of the index as a waiting area for changes.
- This is analogous to a Shopping Basket, you add items to your basket before buying them.

Simplest workflow

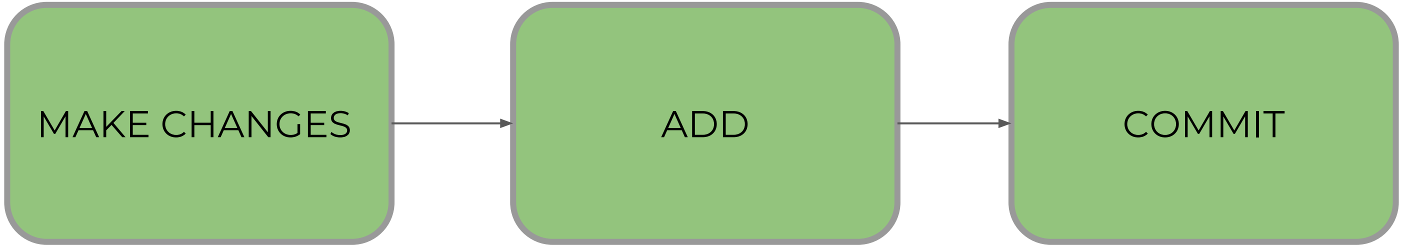


Our first commit

- Now the items are in our Shopping Basket, we need to head to the till and buy them.
- `git commit` creates an entry in our history, with the changes we just **added.** We can then view the commit with `git log`.
- Commits require a message, or if you don't specify one, you'll be asked to type one when you hit enter. Short and descriptive messages are best.

```
> git commit -m "My first commit"

> git log
```

Simplest workflow



MAKE CHANGES

ADD

COMMIT

Editing a file

- Git can also store changes to a file.
- Open **README.txt** in your **my-repo** folder and make some changes to the text.
- Run git status to see that the file has been modified.

```
> git status
```

- To see the actual changes you can use `git diff`

```
> git diff
```

Git add and commit

- The process to commit our changes to the file are exactly the same as for a new file:

```
> git add README.txt
> git commit -m "Edited Readme"
```

Removing files

- To remove a file **already tracked** by git, we can use `git rm`.
- This marks them for removal. Remember though, they still exist in history.

```
> git rm README.txt
> git status
> git commit -m "Removed readme"
```

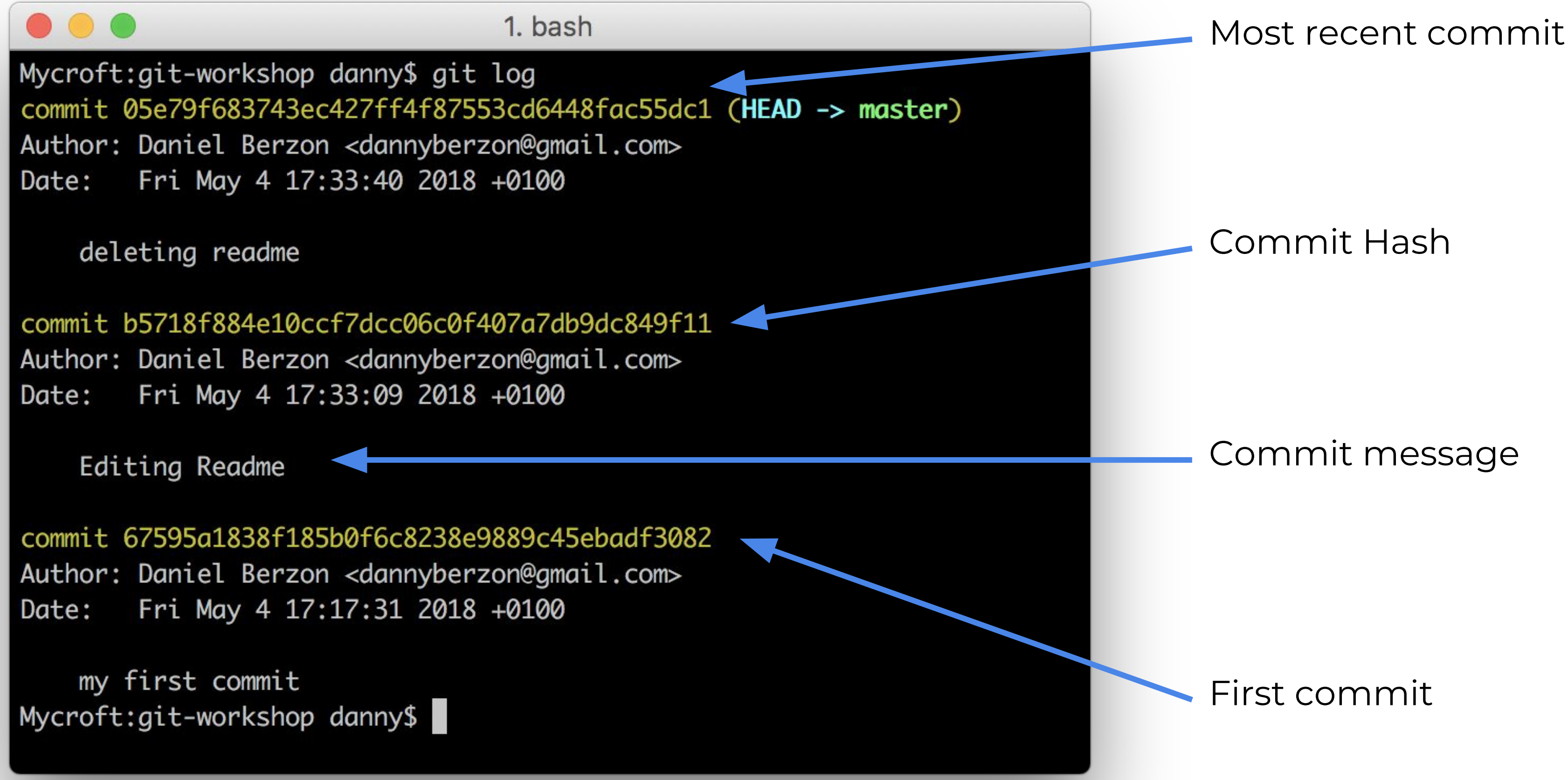
History

- We’ve made three commits. We can look at our commit **history**.
- Git log, will give us a list of our commits in reverse order.

```
> git log
```

- If this output of `git log` fits on more than one terminal screen it will go into a scrolling mode. Press `spacebar` to scroll down a page, and when you get to the end press `q` to exit.

Git log



1. bash

```
Mycroft:git-workshop danny$ git log
commit 05e79f683743ec427ff4f87553cd6448fac55dc1 (HEAD -> master)
Author: Daniel Berzon <dannyberzon@gmail.com>
Date:   Fri May 4 17:33:40 2018 +0100

    deleting readme

commit b5718f884e10ccf7dcc06c0f407a7db9dc849f11
Author: Daniel Berzon <dannyberzon@gmail.com>
Date:   Fri May 4 17:33:09 2018 +0100

    Editing Readme

commit 67595a1838f185b0f6c8238e9889c45ebadf3082
Author: Daniel Berzon <dannyberzon@gmail.com>
Date:   Fri May 4 17:17:31 2018 +0100

    my first commit
Mycroft:git-workshop danny$ 
```

Most recent commit

Commit Hash

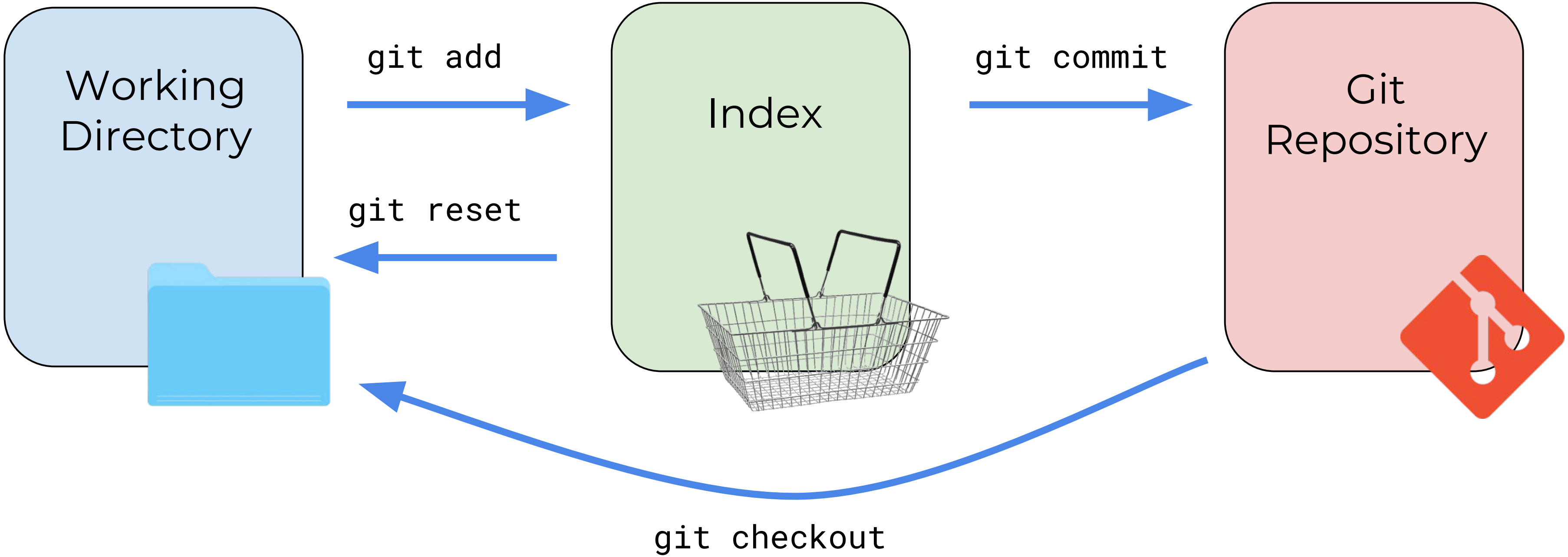
Commit message

First commit

Breathe!

Let’s take a break. Questions?

Story so far



Working Directory

Index

Git Repository

git add

git commit

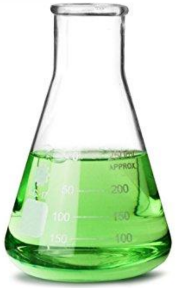
git reset

git checkout

Multi-tasking

Imagine…

You want to make some experimental changes



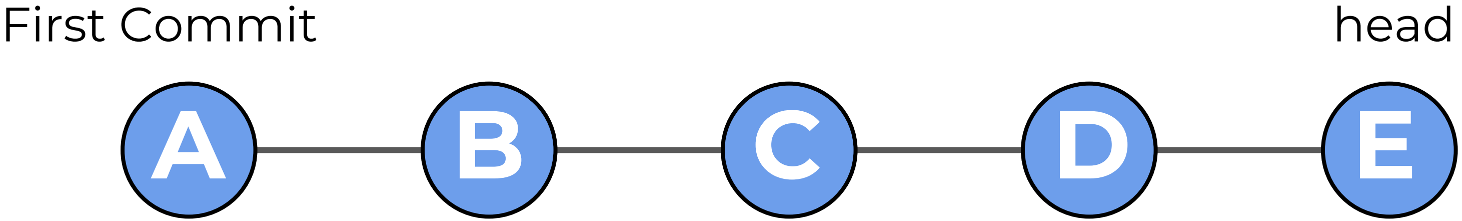
You want to start work on a new feature, and also fix bugs on the existing code.

You have more than one developer working on the same code

How can you track independent changes in parallel?

Branches

- Git allows work to be done in parallel using branches.
- A branch is just a series of commits, one after the other.
- Any new commits we make will go onto the end of the branch
- The most recent commit, is called the **head** of the branch



Master

- All repositories start with one branch, called **master**.
- All our commits so far have gone onto the master branch of our repository.
- If you run `git status`, you will see it says we are on branch master

```
> git status
```

Git branch

- Let’s make a new branch
- `git branch` *`branchname`* creates a new branch called *`branchname`*
- `git branch` on its own will list the branches in the repo

```
> git branch mybranch
> git branch
```

- Two branches are listed, * `master` and `mybranch`.
- The * next to `master` indicates that it is branch we are on.

Checkout

- To move onto a branch we use `git checkout`
- Let’s checkout onto our new branch, and make a commit. The command `touch` will make an empty file

```
> git checkout mybranch
> touch mybranch.txt
> git add mybranch.txt
> git commit -m "commit on mybranch"
> git log
```

Parallel branches

- Let’s checkout back onto master

```
> git checkout master

> git log
```

- Notice how the commit “`commit on mybranch`” isn’t there.
- If you look in the directory, the file `mybranch.txt` has also gone
- The same is true if we make a commit on master…

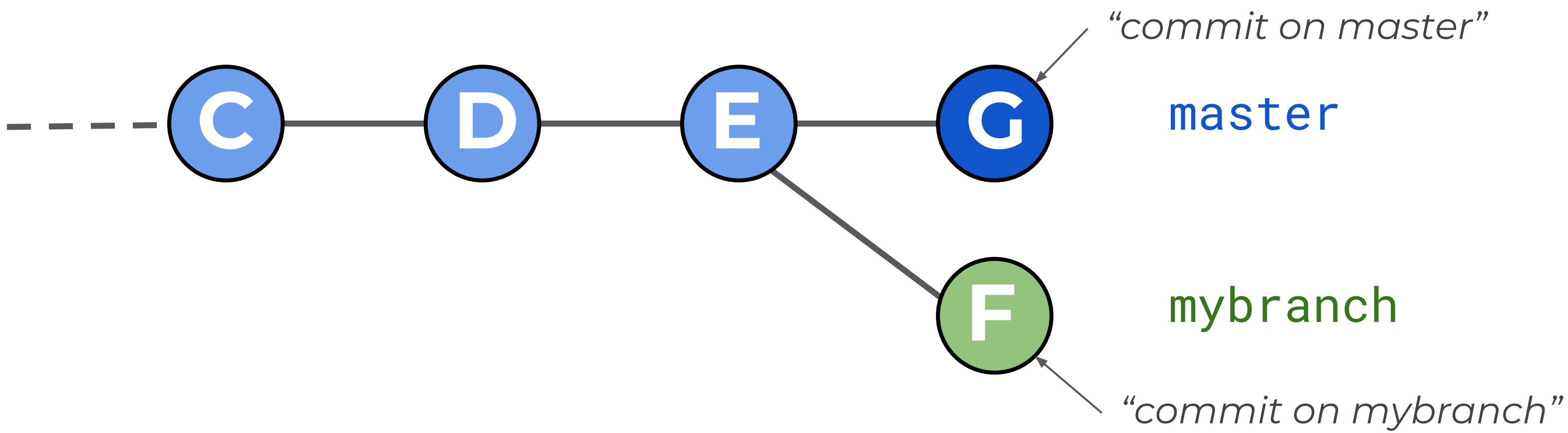
Vice versa

```
> touch master.txt
> git add master.txt
> git commit -m "commit on master"
> git checkout mybranch
> git log
```

- “`commit on master`” is gone. “`commit on mybranch`” is back
- In the directory, `master.txt` is gone, `mybranch.txt` is back

Independent branches

- You can work in parallel on `master` and `mybranch`. They are independent
- When you make commit, it will only go onto the branch you are checked out onto, the **current** branch.

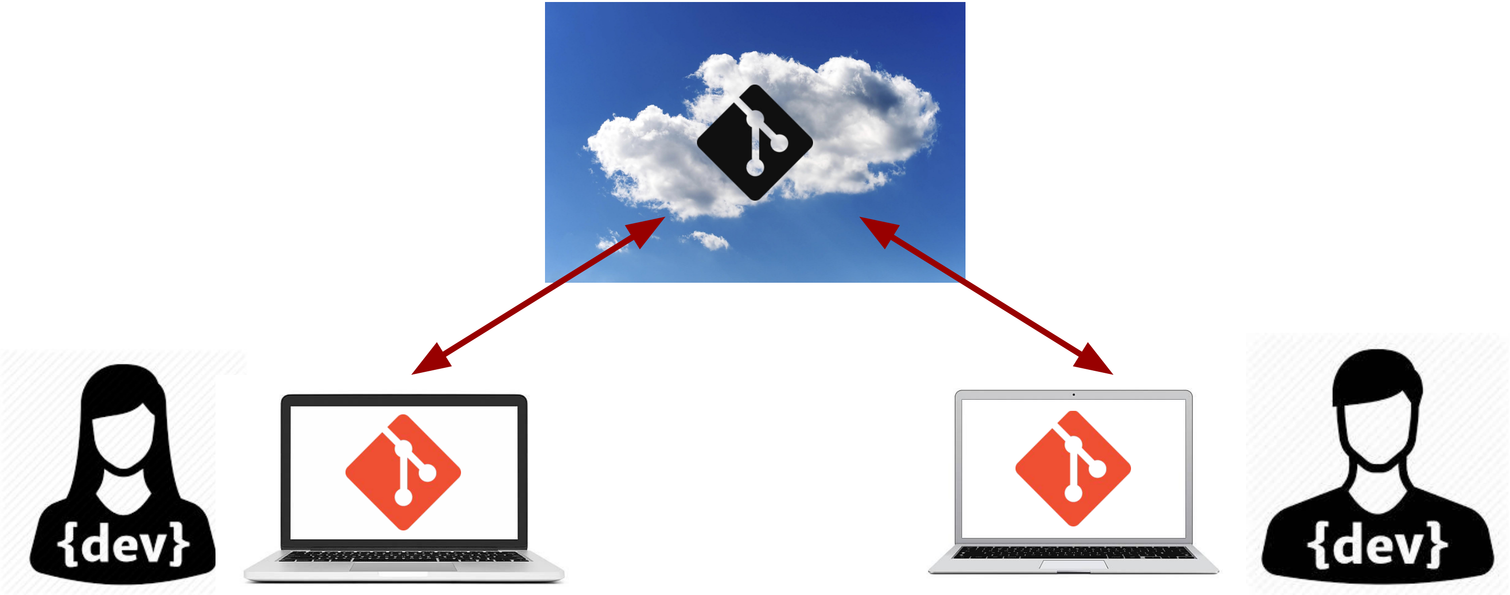


Collaboration

- Branches allow multitasking in a repository
- But what if you have multiple developers each with a repository on their own machine?
- How can you keep their work in sync?

Remote repository

- Each developer can keep their repository in sync with a remote repository on a server.



Remote repository

- Git is fully distributed. Each developer's git repository has a full copy of the history of the project.
- Developers work independently on their repository and then synchronise their work with a remote repository on a server.
- The remote repository could be a git installation on a web server, or it could be a hosted git service...

Bitbucket

used by OCASTA

GitHub

used by codebar

Github

- We’re going to take a look at a github repository.
- This is the repo we will be working on this afternoon.

http://repo.gitforthe.win

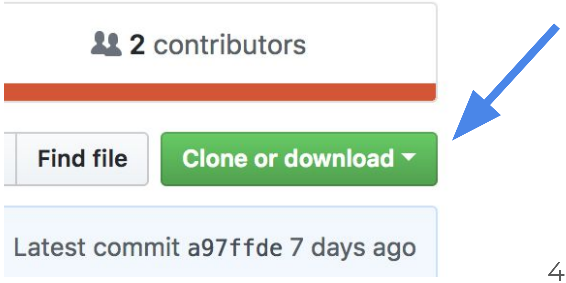
(https://github.com/ocastastudios/codebar-git-workshop)

Git clone

- In order to work on the repository, first we need to get a copy on our local machine. We do this with `git clone`

```
> cd ..

> git clone
https://github.com/ocastastudios/codebar-git-workshop.git
```

- You'll have to enter your github password
- You can copy the link from the github site using the clone or download button



Cloned repository

- You should see a new folder in your documents directory, next to `my-repo` called `codebar-git-workshop`
- If you look inside you should see the files from github repo, E.G `index.html`
- If we go into it in the terminal, we can run `git status` and `git log`, just like our other repo

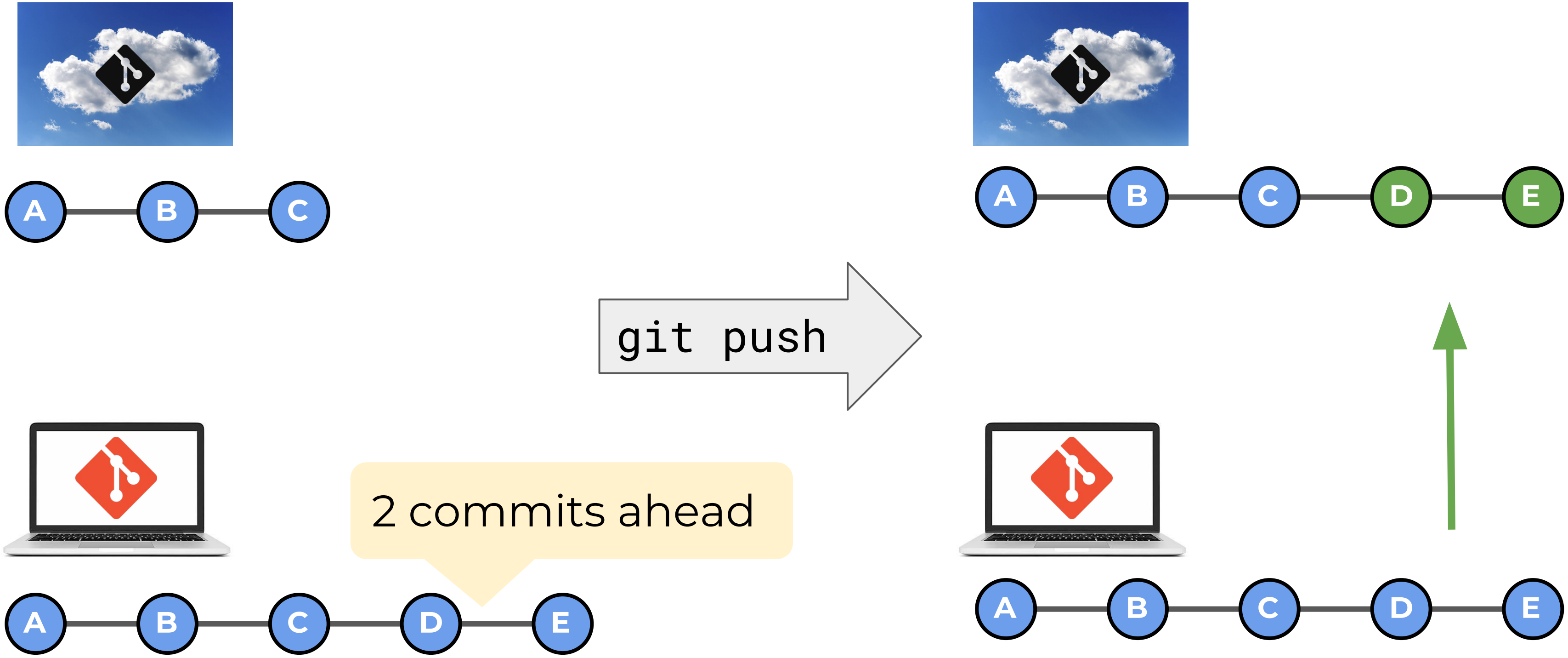
```
> cd codebar-git-workshop

> git status

> git log
```

Push and Pull

- In order to sync your local repository with the remote repository you would use the commands `git push` and `git pull`

- If you have commits locally that are not yet on the remote repository, you can `push` them up to the remote
- If someone has made commits on the remote that you don't have, you can `pull` them down to your repository.

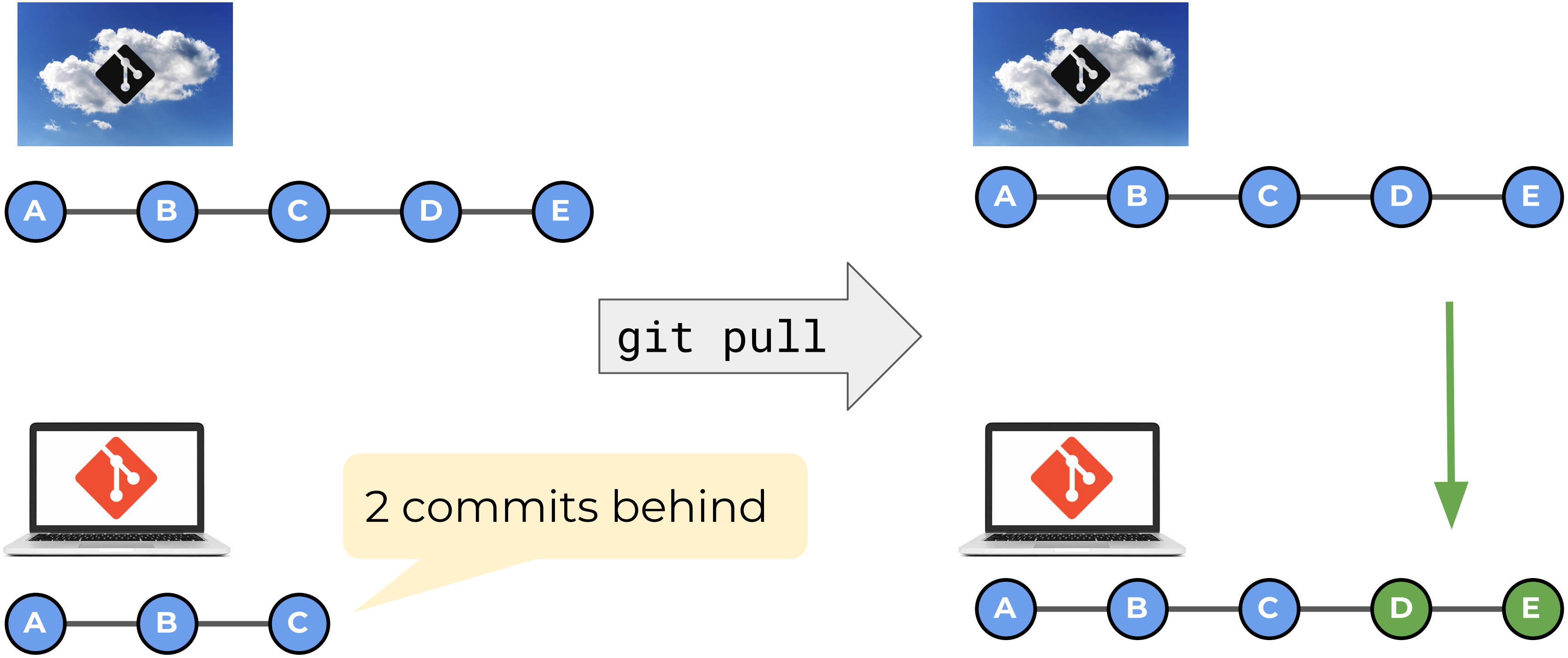
Git push



git push

2 commits ahead

Git pull



git pull

2 commits behind

Git fetch

- Let’s try a git pull and git push. First we will edit a file on the github repository to make a commit.
- Now in your cloned repo, run the following command:

```
> git fetch
```

- The `git fetch` command will retrieve the latest information about commits from the remote. Now lets run:

```
> git status
```

Git pull

- Git status told us the following

```
Your branch is behind 'origin/master' by 1 commit…
```

- `origin` is our remote on github, and `master` is the branch. We are behind by one commit, which we can pull down:

```
> git pull
```

- Our local repository is now up to date with the edited file

Pushing

- Let’s try a `git push` now.
- Because we are all pushing to the same repository let’s each make an individual branch, make a commit and then push that up.
- First let’s make the branch. In the commands below change `myname` to your actual name with no spaces (eg. `danielvanberzon`).

```
> git branch myname
> git checkout myname
```

Make a commit

- Now let’s make a commit

```
> touch myname.txt
> git add myname.txt
> git commit -m "branch commit"
```

- This time `git status` won’t tell us anything because the remote doesn’t know about our new branch.
- First we need to push the branch up

Git push

- Pushing up a branch for the first time is a little more complicated. We need to tell git where to push changes.
- Run the following command

```
> git push -u origin myname
```

- The `-u` flag tells git to push our commits to the remote ‘`origin`’ and to a branch called ‘`myname`’.

Pushed to Github

- You only need to add the `-u` the first time you push a branch. After that you can make commits and just use **`git push`**.

```
> git rm myname.txt
> git commit -m "branch commit 2"
> git push
```

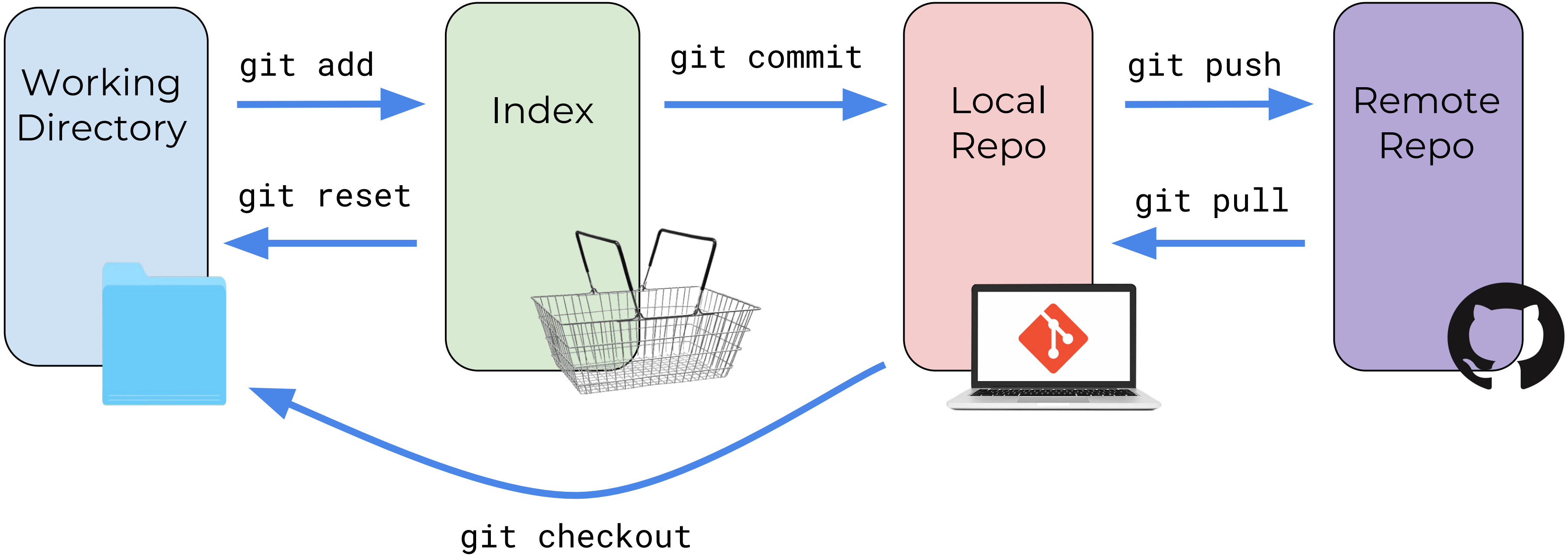
- If we look at github we should now see loads of branches with your names on them.

GitHub

Phew!

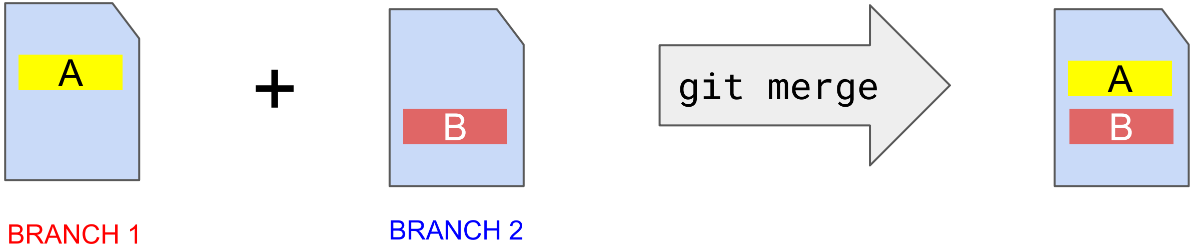
Time for another break. Questions?

Story so far


Working Directory
git add
git reset
Index
git commit
Local Repo
git push
git pull
Remote Repo
git checkout

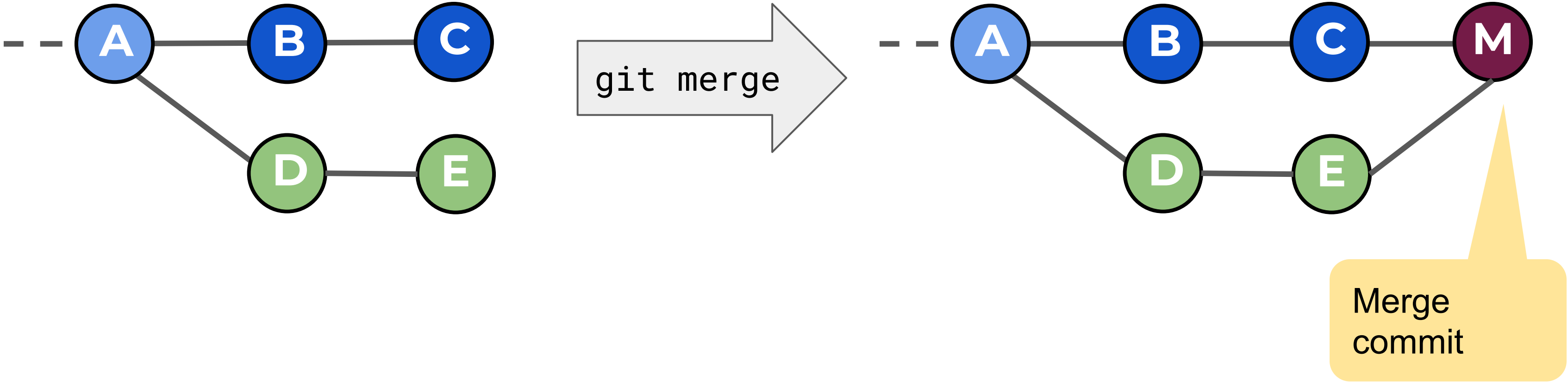
Combining work

- You’re working on a branch, and someone else has done a bug fix on another branch. How can you include their changes in your branch.
- We use the `git merge` command to import changes from one branch into another.
- If the same file has changed in both branches, git will combine the changes together.



Merge

- The command `git merge` will import the changes from one branch **into** another using a special commit called a `merge commit`.
- The source branch will not be changed.



`git merge`

Merge commit

Git merge

- Let’s try it out. Let’s go back to **my-repo** repository and merge one branch into another. First let’s list the branches

```
> cd ..
> cd my-repo
> git branch
```

- Two branches are listed, `master` and `mybranch`.
- Let’s merge `mybranch` into `master`

Git merge

- The command `git merge branchname` will merge the branch `branchname` into the current branch.
- So to merge `mybranch` into `master` we first have to checkout onto `master` and then run `git merge mybranch`.
- We add a flag `--no-edit` so that we don't have to provide a commit message for the merge commit.

```
> git checkout master

> git merge mybranch --no-edit.
```

Merged branch

- If you look in the folder now you will see that the files `mybranch.txt` and `master.txt` are both present. Git has merged the changes from both the branches into the master branch.
- If you run git log:

```
> git log
```

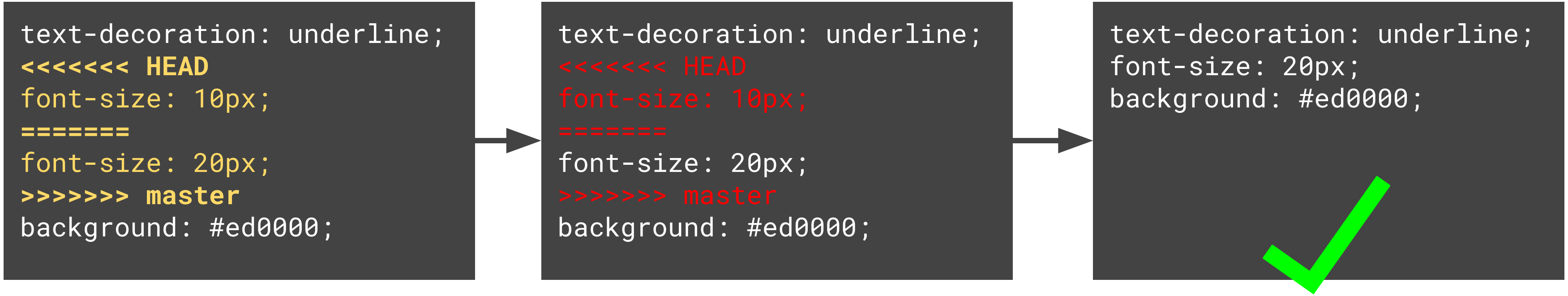
- You will see the commits, “`commit on mybranch`” and “`commit on master`” are both there, as well as a merge commit “`Merge branch ‘mybranch’`”

Conflicts

- Most of the time, git is smart enough to figure out how to merge changes, but sometimes it needs help.
- A conflict occurs when git cannot resolve multiple changes to the same part of a file.
- Conflicts are shown by **CONFLICT** written next to the file when you run `git merge.`
- We need to **resolve** the conflict and manually choose the changes we want to keep.

Resolving Conflicts

- Inside a conflicted file, you'll see some strange notation… these are called **conflict markers**.
- The changes from the two conflicted sources are shown between each set of markers.
- To resolve a conflict, pick the change you want to keep by removing the markers and the change you do not want.



```
text-decoration: underline;
<<<<<<< HEAD
font-size: 10px;
=======
font-size: 20px;
>>>>>>> master
background: #ed0000;
```

```
text-decoration: underline;
<<<<<<< HEAD
font-size: 10px;
=======
font-size: 20px;
>>>>>>> master
background: #ed0000;
```

```
text-decoration: underline;
font-size: 20px;
background: #ed0000;
```

Workflow

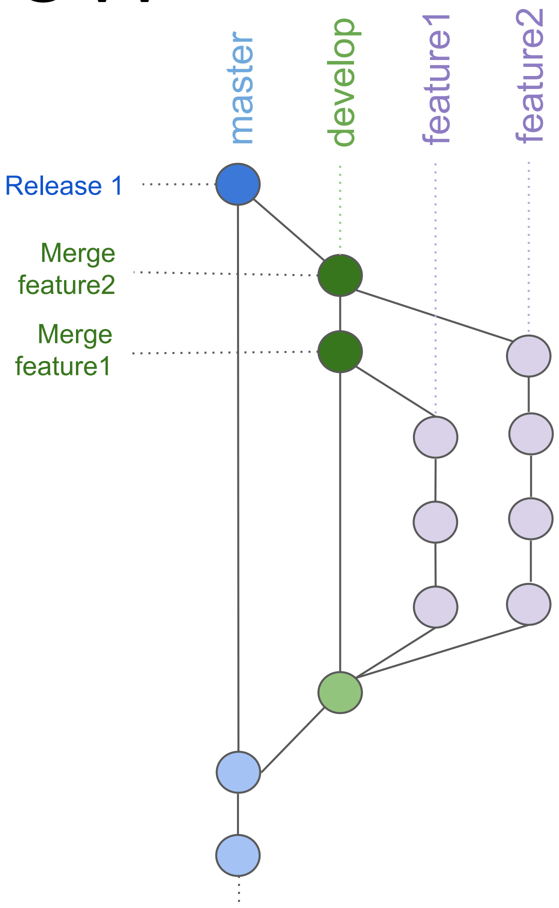
- When multiple people are working on a project, it can be hard to juggle who is doing what, and when to deploy code

- A git project can contain:
  - A version of the code that is `live`
  - New features being developed
  - A next release being tested
  - Bug fixes for the live version

- We can manage this chaos with conventions for naming and merging branches. We call this a `workflow`

Typical Workflow

- `Master` - represents the latest public version (live)
- `Develop` - represents the current state of progress on the next release
- `Feature` - represents the work in progress on a specific feature, with a descriptive branch name
- When a feature is finished it is merged into `develop`. When a release is ready it is merged into `master`.



Quality control

- How can you control the quality of code going into a release?

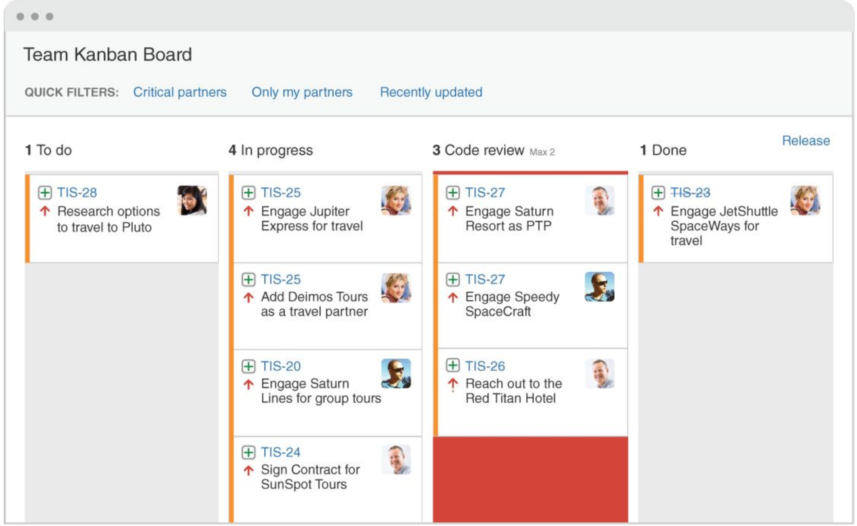
- **Github** and **Bitbucket** have a tool which makes this easier, called a `pull request`
- When a developer has finished with a feature and is ready to merge, they make a `pull request`
- Another developer can then review the code changes on the branch, and decide whether it should be merged

Pull request

- You can create a `pull request` (PR) from a branch on github, giving it a destination branch you want to merge into (eg master)
- You can invite collaborators to look at your PR.
- The PR page shows the commits, and the changes from your branch.
- Collaborators can make comments and approve or reject the PR.
- If approved the pull request can be `merged`, which will merge the branch into the destination branch.
- Let’s make a pull request on `github`…

Divide and conquer

- Successful software development in a team relies on knowing:
    - **Size**: How big a piece of work is. If a piece of work is too large, it should be broken down.
    - **Assignee**: The person responsible for completing a piece of work.



Team Kanban Board

QUICK FILTERS: Critical partners Only my partners Recently updated

Release

1 To do	4 In progress	3 Code review Max 2	1 Done
TIS-28 Research options to travel to Pluto	TIS-25 Engage Jupiter Express for travel	TIS-27 Engage Saturn Resort as PTP	~~TIS-23~~ Engage JetShuttle SpaceWays for travel
	TIS-25 Add Deimos Tours as a travel partner	TIS-27 Engage Speedy SpaceCraft	
	TIS-20 Engage Saturn Lines for group tours	TIS-26 Reach out to the Red Titan Hotel	
	TIS-24 Sign Contract for SunSpot Tours		

- We often use a **ticketing system** to help in dividing up and conquering pieces of work and tracking the progress of a project.

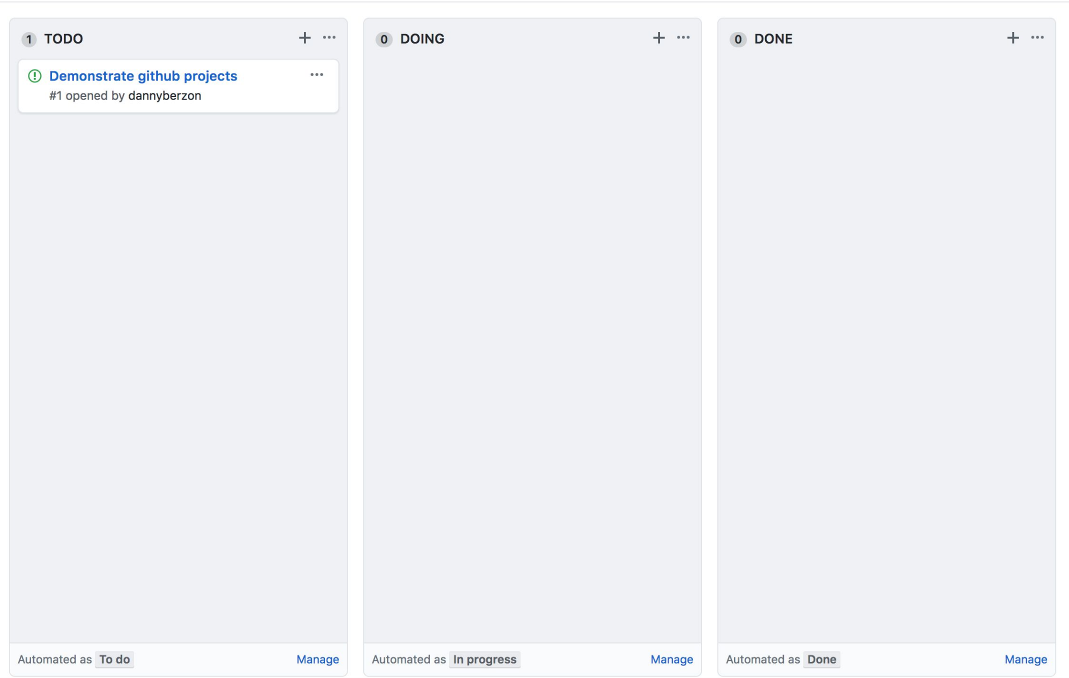
Methodologies

There are also many different ways of approaching a project, we won’t discuss them in detail, but today we’ll practise something close to a methodology known as **Scrum.** In short, this involves:

- Teams spend a set period of time (a **sprint**) working towards a goal, at the end they deliver something and review their progress.
- They use **estimations** to calculate the size of work to be done and whether it can be completed in a sprint.
- This allows for effective planning and an agile way of working with fast results and turnaround.

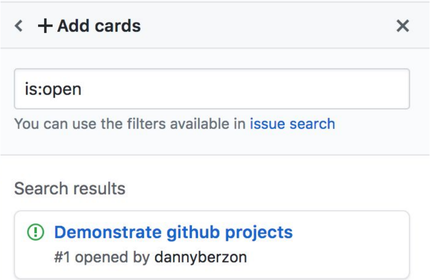
GitHub Boards

- GitHub has a simple ticketing system which we'll use today. This can be found in the **Projects** tab.
- For simplicity, we'll simply have TODO, DOING and DONE columns to track our work.
- Simply drag your ticket across as you work.



GitHub Boards

1. Drag your ticket into TODO, you can do this using **Add Cards**.
2. Once you start work, drag your ticket into DOING.
3. Once it's been completed and your pull request has been merged, move it to DONE.

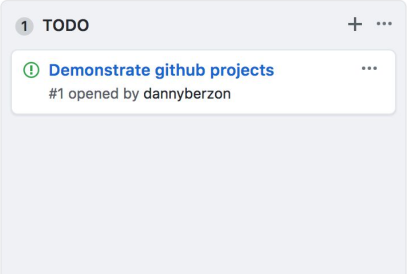


is:open

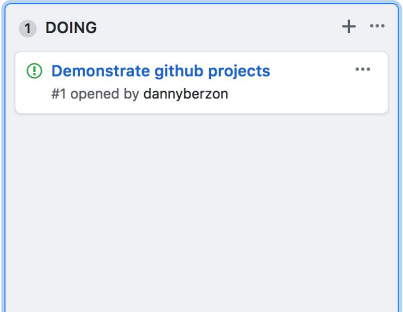
You can use the filters available in issue search

Search results

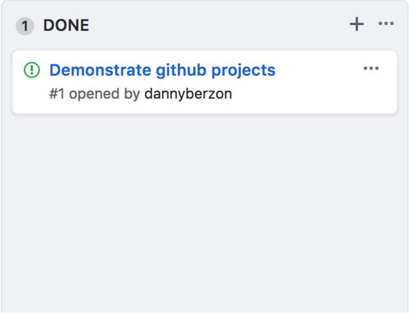
Demonstrate github projects
#1 opened by dannyberzon



Demonstrate github projects
#1 opened by dannyberzon



Demonstrate github projects
#1 opened by dannyberzon



Demonstrate github projects
#1 opened by dannyberzon

And relax…

Let’s take another break. Questions?

Your mission…

- **The Codebar Git Workshop website is in need of a facelift**.
- You're going to do that for us! You'll only need basic HTML and CSS to do this, ask your team or a coach if you need help at any point.
- Firstly, split into teams of 5.
- We've created a set of tasks (tickets) for the work that needs doing. We'll use the "sprint planning" to assign tickets to a team.

The process

1. One person in the team will be submitting a **pull request** for a **ticket,** but we'll do a few of these, so don't forget to rotate and collaborate.
2. Read the **ticket**, check you understand the task. If you're stuck, please ask! Move it into DOING on the board.
3. Make a **branch** with the number of ticket (eg. ticket-12).
4. Write some code and **commit**. Don't forget to **push** them after each one too.
5. When you're ready, go to GitHub and create a **Pull Request.**
6. We'll take a look and review it. When we're happy, we'll **merge** it. When this happens. Don't forget to move the ticket to DONE.

Git command cheat sheet at http://gitforthe.win/commands#afternoon

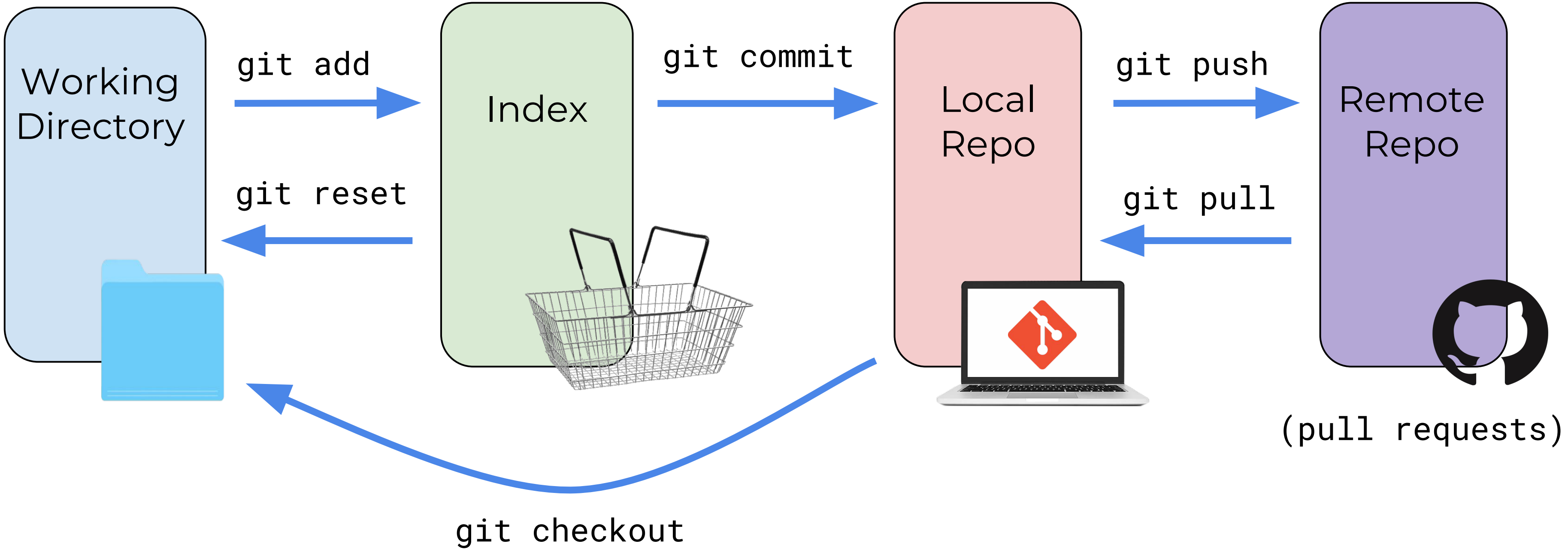
Retrospective

So… how was that?

👍 👎

Wow, we learned a lot...

Summary


Working Directory
git add
git reset
Index
git commit
Local Repo
git push
git pull
Remote Repo
(pull requests)
git checkout

Thank you!

OCASTA

codebar

The Skiff

Reset

- What do you do if you accidentally **add** a file? Use reset.
- Reset removes files from the index.

```
> touch hello.txt
> git add hello.txt
> git status
> git reset hello.txt
> git status
> rm hello.txt
```

Checkout

- What about if you want to discard the changes to a file since your last commit?
- `Checkout` retrieves files from your history. This can be useful for reverting a file to the state of your last commit.
- Make some changes to **README.txt** in your **my-repo** folder and then run:

```
> git status
> git checkout README.txt
> git status
```