NAYANA DAVIS

GITHUB FOR TEAMS

TEXT

What does Git do and how does it help us?

GIT

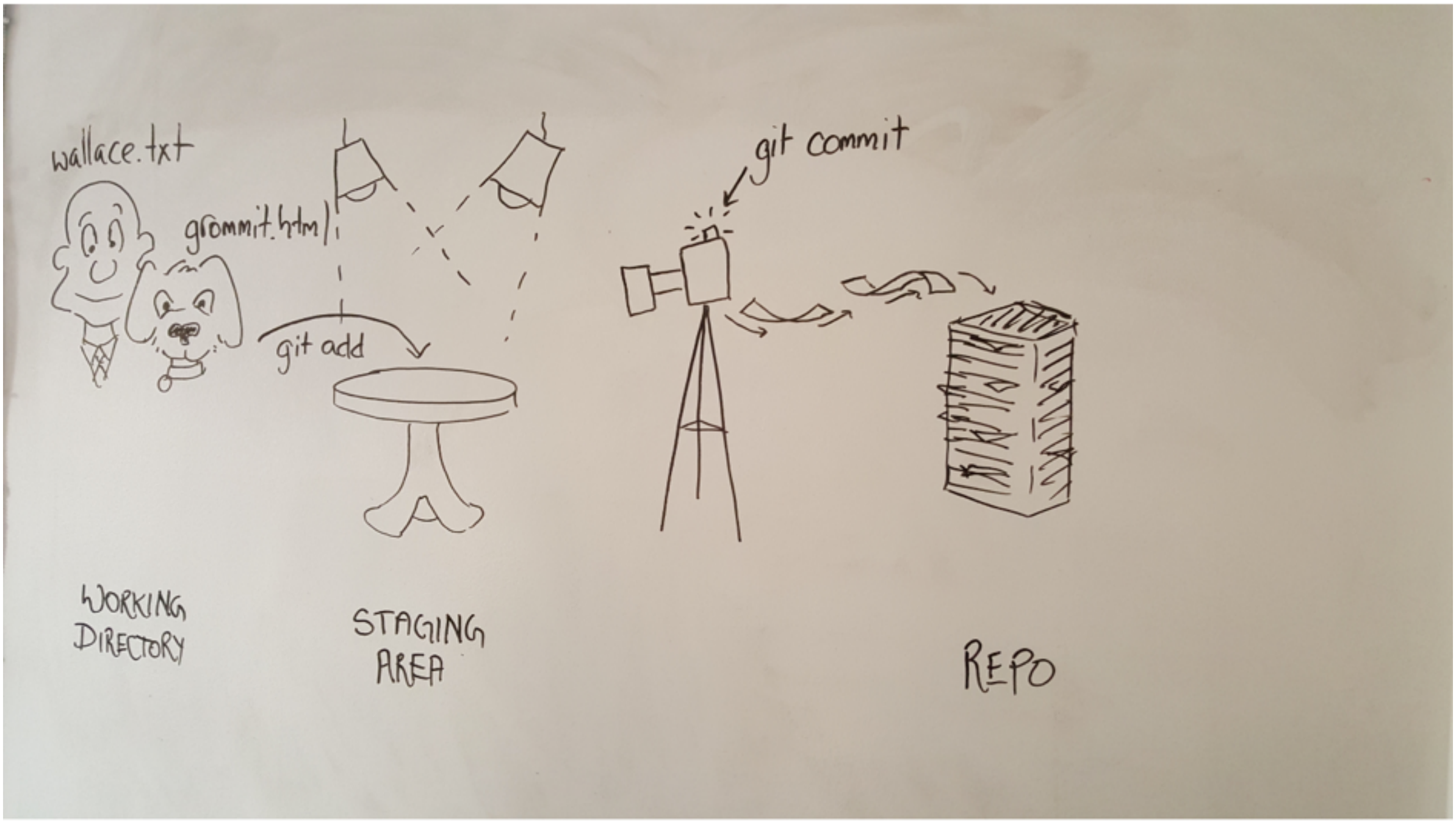
Git is a fast version control system, that allows you to efficiently handle projects large and small. It helps by:

- Reverting to past versions
- Comparing changes to past versions
- Keeping track of what each version 'meant'
- Collaborating / discussing changes (GitHub feature)
- Fearlessness in making changes

KEY GIT TERMS & CONCEPTS

- Demo with DSI Course Materials
- working tree - the folder (and it's files and sub-folders, that are in the repository)
- repository - collection of commits (save points of the working tree)
- commit - a snapshot of the working tree at a giving time (along with a message of what changed)
- the index - a staging area where we list changes we want to commit
- HEAD - what is currently checked out
- status - what files have changed, and what step are they in.

TEXT


wallace.txt
grommit.html
git add
git commit
WORKING DIRECTORY
STAGING AREA
REPO

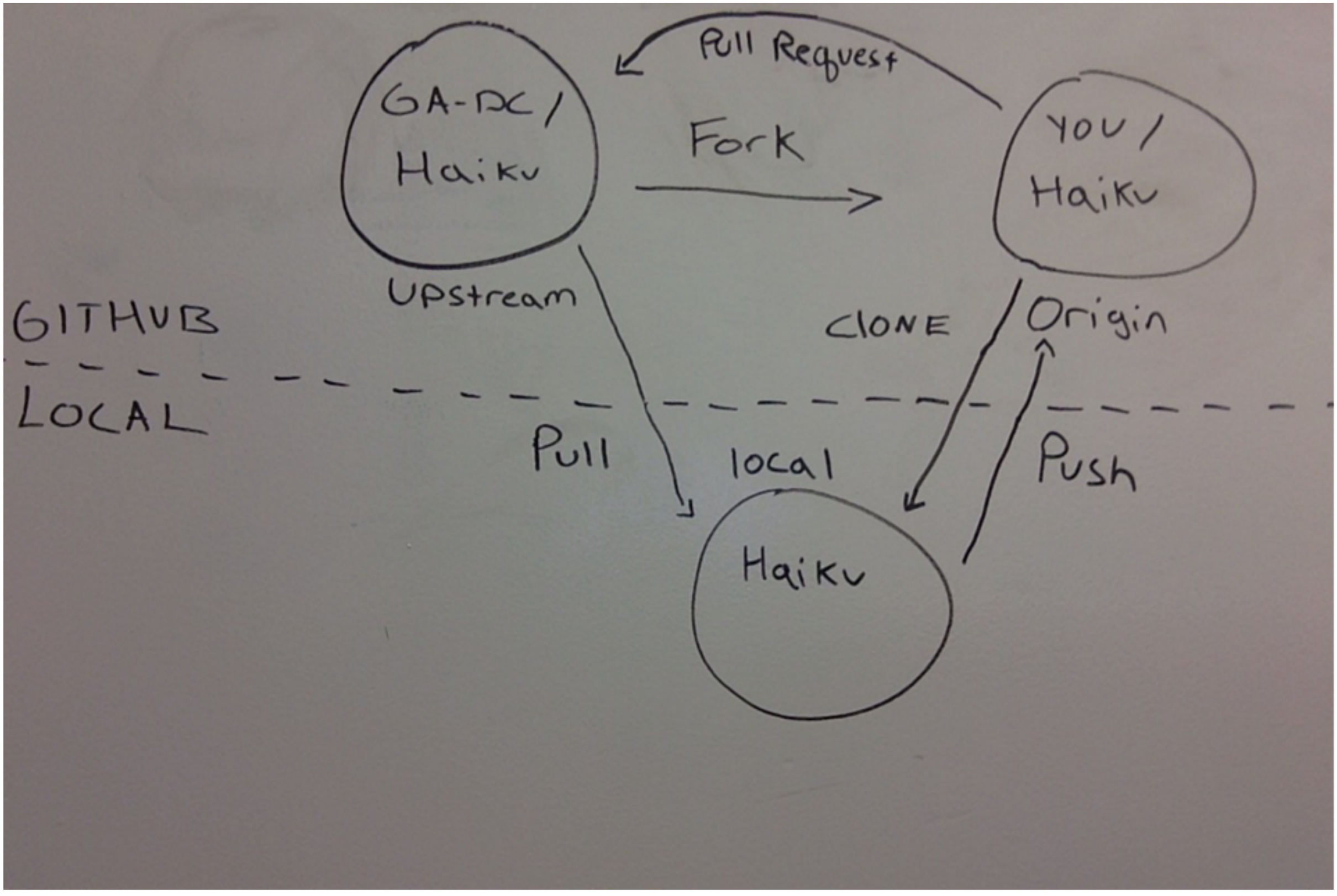
REMOTE REPOSITORIES & GITHUB

- remote - another repository that can be synchronized with a remote
- github - a service that hosts git remote repositories, and provides a web app to interact / collaborate on them
- fetch - downloading the set of changes (commits) from a remote repository
- merge - taking two histories (commits),
- clone - download an entire remote repository, to be used as a local repository
- pull - fetching changes and merging them into the current branch
- push - sending changes to a remote repository and merging them into the specified branch
- merge conflict - when two commits conflict, and thus can't be merged automatically.

FORKING & PULL REQUESTS

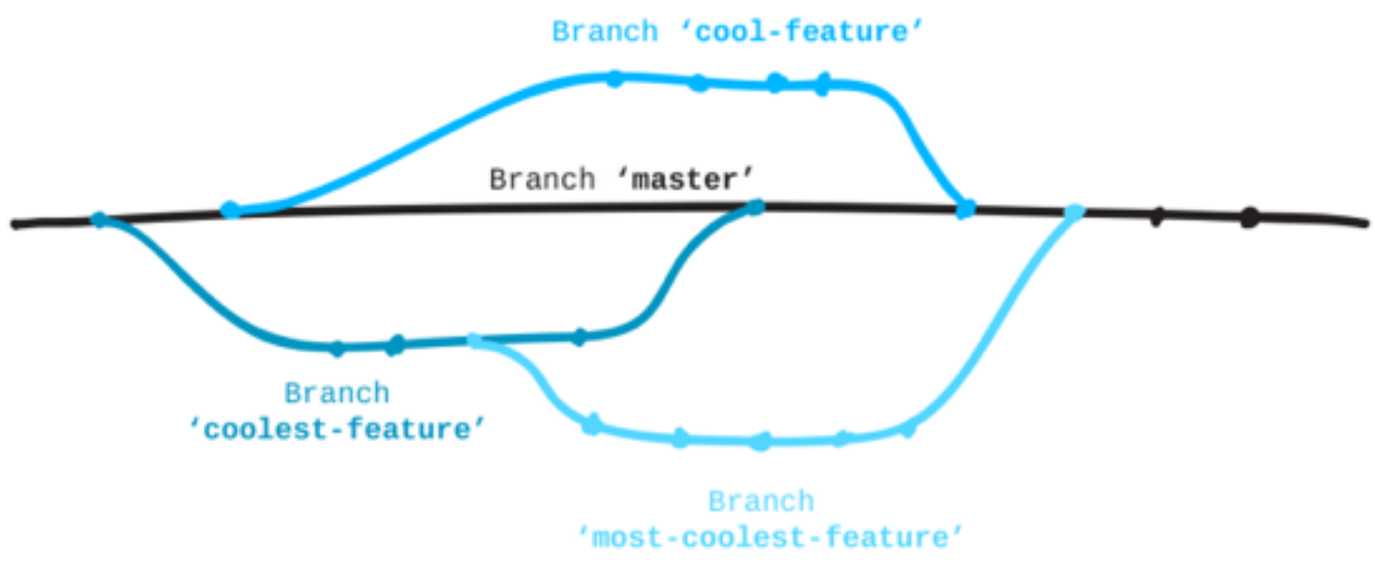
- fork - make a copy of a repo on github under a different account
- pull request - a github feature which allows a user to suggest and discuss changes to a repo they have forked

TEXT


GA-DC /
Haiku
Pull Request
Fork
You /
Haiku
GITHUB
LOCAL
Upstream
Clone
Origin
Pull
local
Push
Haiku

GIT BRANCHING

- When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes.



WHY IS BRANCHING AN IMPORTANT PART OF GIT?

- To allow experimentation.
- To allow easy bug fixes on a stable version while features are being developed.
- To allow work to proceed on multiple features (or by multiple people) without interfering. When a feature is complete, it can be merged back into master.
- "Branch Early, Branch Often": Branches are lightweight, there is no additional overhead associated with branches, so it can be a great way to organize our workflow

MERGING

- If our feature branch and work is complete, we need to merge our changes back into our master branch.
- $ git merge <feature_branch_name>
- Then delete branch
- git branch -d <feature_branch_name>

YOU DO: BRANCHING EXERCISE (15 MINUTES)

Do Levels 1-3. Stop at 4: "Rebase Introduction". Take your time: http://learngitbranching.js.org/

- Read all the dialogs. They are part of the tutorial.
- Think about what you want to achieve
- Think about the results you expect before you press enter.

COMMON COMMANDS FOR MANAGING BRANCHES

- git branch <new_branch_name> - create a new branch
- git checkout <branch_name> - switch to a specific branch (checks out tip commit and makes branch active)
- git checkout -b <new_branch_name> - create a new branch and check it out in one step
- git branch - list local branches
- git branch -r list remote branches
- git branch -a list both remote & local branches
- git branch -d <branch_to_delete> - delete a branch
- will not let you delete if branch isn't merged into another branch (i.e. would cause data loss)
- git branch -D <branch_to_delete> - over-rides and deletes a non-merged branch - be careful!
- git merge <branch_name> - merges <branch_name> into the current branch, creating a new merge commit in the process

MERGE CONFLICTS

- When we try to merge two branches (or commits from the same branch from a remote), changes may conflict. In this case, git will stop and ask us to fix the issues manually.
- A ‘conflict’ occurs when the commit that has to be merged has some change in the same place as the current commit.

MERGE CONFLICTS

```
jeff@js4:~/git-example/merge-conflict-example-for-laura [master] $ vim Person.java
jeff@js4:~/git-example/merge-conflict-example-for-laura [master] $ cat Person.java
public class Person {

        private String firstName;
        private String lastName;
        private int age;

        public Person(String firstName, String lastName, int age) {
                this.firstName = firstName;
                this.lastName = lastName;
                this.age = age;
        }

        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }

        public void setLastName(String lastName) {
                this.lastName = lastName;
        }
}
jeff@js4:~/git-example/merge-conflict-example-for-laura [master] $ git add .
jeff@js4:~/git-example/merge-conflict-example-for-laura [master] $ git commit
[master 776e231] Adding age to Person.
 1 files changed, 3 insertions(+), 1 deletions(-)
jeff@js4:~/git-example/merge-conflict-example-for-laura [master] $ git push
To github-jeffshantz:jeffshantz/merge-conflict-example-for-laura.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'github-jeffshantz:jeffshantz/merge-conflict-example-f
or-laura.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
jeff@js4:~/git-example/merge-conflict-example-for-laura [master] $
```

TEXT

```
jeff@js4:~/git-example/merge-conflict-example-for-laura [master] $ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From github-jeffshantz:jeffshantz/merge-conflict-example-for-laura
   9f8702e..af7d61a  master     -> origin/master
Auto-merging Person.java
CONFLICT (content): Merge conflict in Person.java
Automatic merge failed; fix conflicts and then commit the result.
jeff@js4:~/git-example/merge-conflict-example-for-laura [master] $
```

- What's between HEAD is local, the section below is from Github

```
1 public class Person {
2
3         private String firstName;
4         private String lastName;
5         private int age;
6
7 <<<<<< HEAD
8         public Person(String firstName, String lastName, int age) {
9                 this.firstName = firstName;
10                this.lastName = lastName;
11                this.age = age;
12 =======
13        public Person(String firstName, String lastName) {
14                this.setFirstName(firstName);
15                this.setLastName(lastName);
16 >>>>>>> af7d61af856f43aa3e276d57313efe4a498dac6d
17        }
18
19        public void setFirstName(String firstName) throws IllegalArgumentException {
20                if (firstName.isEmpty())
21                        throw new IllegalArgumentException("Can't pass empty first na
   me");
22                this.firstName = firstName;
23        }
24
25        public void setLastName(String lastName) {
26                if (lastName.isEmpty())
27                        throw new IllegalArgumentException("Can't pass empty last nam
   e");
28                this.lastName = lastName;
29        }
30 }
~
~
~
```

MERGE CONFLICTS

- Has older version, doesn’t do pull and starts editing
- then tries to push and gets error message
- so do a git pull and git tries to automerge changes
- open in text editor, manually fix to how you want and delete markup from conflict message
- Add, commit, push

TEXT

YOU DO: MERGE CONFLICTS (15 MINUTES)

1. Pair up with someone.

2. Pick someone as the 'primary', and the 'secondary'.

3. Create a New Repo

Primary Student Instructions:

- Create a new directory named merge-conflicts.

- Initialize merge-conflicts as a git repository and create an index.html file

- Work with the Secondary student to fill out the basic structure for the index.html file. -Include in the index.html file an h1 tag with the content "Merge Conflicts", and a p tag with something new you learned about today.

- Create a New Repo on Github called merge-conflicts and add this repo locally as a remote repo for your merge-conflicts directory.

- Make sure to save and commit local changes and push up to the Remote Repo

- Add the Secondary student as a Collaborator (search github for how to do this)

TEXT

Secondary Students Instructions: After they are added as a Collaborator, they should clone the same repo. Do not fork the Repo.

4. Both the Primary and Secondary should make changes locally on the same "master" branch

–Modify the index.html, including both changing the h1 and p elements

–Add and Commit Changes Locally.

5. Merging commits:

- The Primary Student should push up their changes first
- Then, the Secondary Student should do the same and try pushing up their changes

6. When the Secondary Student tries to push their commits, there should be merge conflicts.

The Secondary Student should work locally (with the Primary) to resolve the merge conflicts.

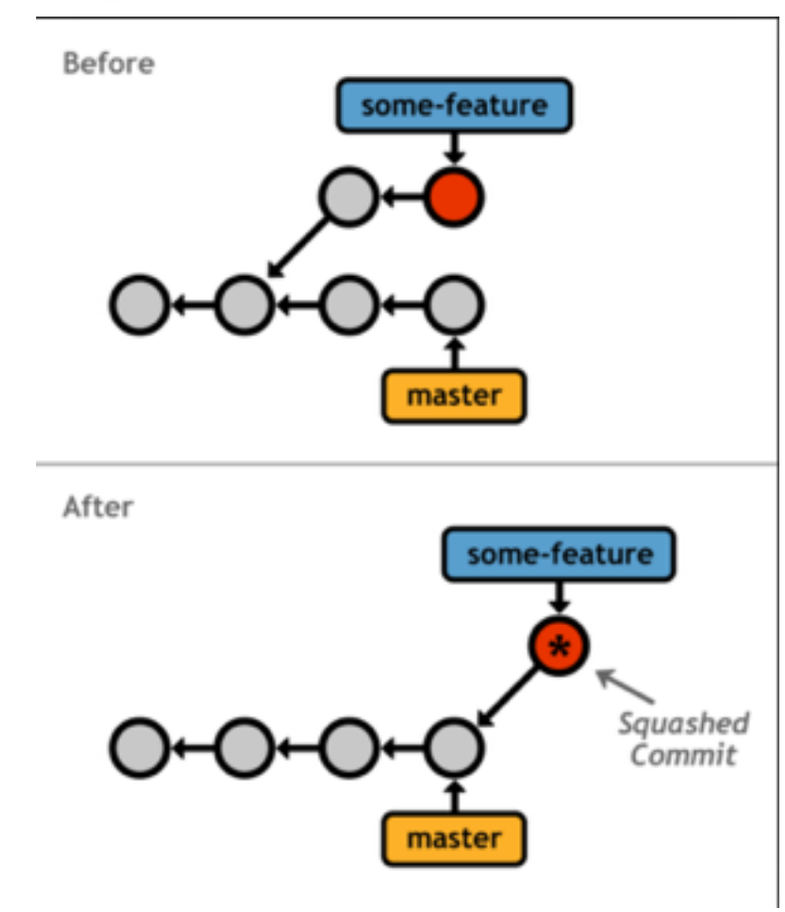
Once completed, commit and push up changes to the remote repo

7. Pulling Changes:

Now, the Primary student should pull down the changes from the remote repo and work to resolve any merge conflicts

REBASING

Rebasing allows us to rearrange and effectively rewrite our git commit history! Rather than combining the finished data from two different branches via a single commit, it combines the two branches themselves, rearranging them and, effectively, re-writing history.



REBASING

- rebase makes commits look like they happened along the same line of work--it doesn't look like the work was done on a branch
- Say you are in branch test, git rebase master --tells git which branch to rebase against
- if there are conflicts, will stop rebasing
- then checkout master
- then merge testing ---should merge without conflicts

REBASING

- Rebase is extremely useful for cleaning up your commit history, but it also carries risk; when you rebase, you are in fact discarding your old commits and replacing them with new (though admittedly, similar) commits, and this can seriously screw up a collaborator if you're working in a shared repo. The golden rule for git rebase is "Only rebase before sharing your code, never after."

SINGLE REMOTE WORKFLOWS

One thing all of these approaches have in common is the necessity of staying on top of changes within a single shared repository.

This is usually accomplished by running git fetch, which pulls updates from origin, and merging those updates; alternatively, you could use git pull to do both at once.

1. CENTRALIZED WORKFLOW

How It Works: The remote repo has one single branch on it, master. All collaborators have separate clones of this repo. They can each work independently on separate things. However, before they push, they need to run git fetch/git pull (with the --rebase flag) to make sure that their master branch isn't out of date.

(+) Very simple

(-) Collaboration is kind of clunky.

2. FEATURE BRANCH WORKFLOW

How It Works: This workflow is very similar to the 'Centralized' workflow. The biggest difference is that there are branches (which helps to keep feature-related commits isolated), and that instead of pushing changes up directly, collaborators: (a) push up changes to a new remote branch rather than master, and (b) submit a pull request to ask for them to be added to the remote repo's master branch.

(+) Better isolation than Centralized model, but sharing is still easy. Very flexible.

(-) Sometimes it's too flexible - it doesn't meaningfully distinguish between different branches, and that lack of structure can cause problems on larger projects.

GITFLOW WORKFLOW

How It Works: Similar to the Feature Branch workflows, but with more rigidly-defined branches. For example:

- Historical Branches : master stores official releases, while develop serves as a living 'integration branch' that ties together all the standalone features.
- Release Branches : 'release' branches might exist for any given release, to keep all of those materials together.
- Feature Branches : pretty much the same as in the prior model.
- Maintenance/'Hotfix' Branches : branches used to quickly patch issues with production code.

(+) Highly structured - works well for large projects.

(-) Sometimes overkill for something small.

DISTRIBUTED WORKFLOWS

These approaches all use multiple remote repos; typically, everyone has their own fork of the 'original' project (the version of the repo that's publicly visible and is managed by the project maintainer), and changes are submitted via pull request.

INTEGRATION MANAGER WORKFLOW

How It Works: One collaborator plays the role of 'Integration Manager'. This means that they are responsible for managing the official repository and either accepting or rejecting pull requests as they come in.

(+) One person integrates all changes, so there's consistency.

(-) Could get overwhelming for large projects.

DICTATOR/LIEUTENANTS WORKFLOW

How It Works: This workflow is very similar to the Integration Manager Workflow. The biggest difference is that rather than submitting all pull requests to a single integration manager, pull requests are funneled through 'Lieutenants', who all report to the 'Dictator'. Only the Dictator has write access to the official repo.

(+) Could get overwhelming for large projects.

(-) Only one person has final write access, so there's consistency but also a single point of failure.