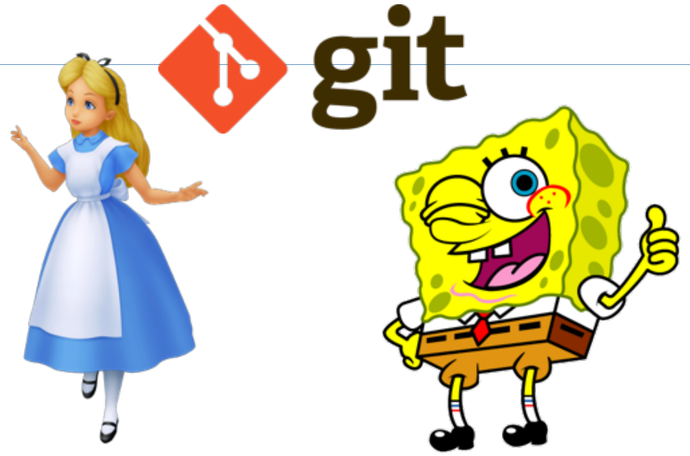
Git Training

Introduction to concepts, commands and collaboration workflows

Agenda



- **Part 1 : 08:00 - 10:00**
  - Introduction
  - Core concepts, data model and commands
  - Git in practice with Alice and Bob

- **Part 2 : 10:00 - 12:00**
  - Collaboration workflows
  - Let’s play music with git and GitHub

Introduction

Objectives

- **Get familiar with git**
  - Understand what happens "behind the scenes" (data model)
  - Present important commands (not all of them!)
- **Present a sandbox environment for individual training**
  - How can I practice without breaking things?
  - How can I practice multi-user scenarios by myself?
- **Get familiar with collaborative workflows and pull-requests**
- **Put the GitHub workflow in practice with TDD and maven**

Objectives

- **Get fam**
  - Unders
  - Preser
- **Prepare**
  - How c
  - How c
- **Get far**
- **Put the GitHub workflow in practice with TDD**

el)

uests

Get ready to shutdown TFS in August!

What do we need to install?

- **For the first part**
  - command line tools (git bash) *https://git-scm.com/download/win*
- **For the second part**
  - JDK
  - maven
  - (NetBeans)
  - A GitHub account

Core concepts and the git object model

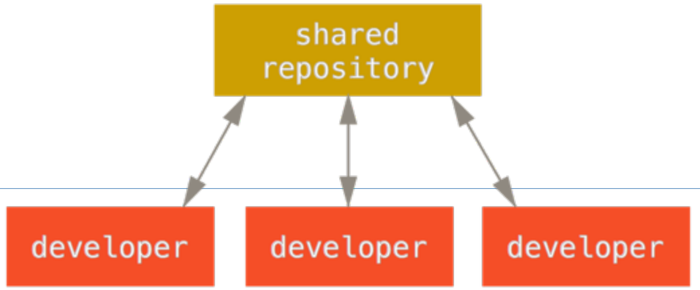
Git vs GitHub/GitLab/BitBucket

- **Git is a Distributed Version Control System (DVCS)**
  - A data model (blobs, trees, commits, branches, tags)
  - Commands to work on the local repository (clone, commit, etc.)
  - Commands (fetch, pull, push) to synchronise remote repositories.
- **GitHub and similar systems are Web-based collaborative environments built on top of Git**
  - Repositories hosted in the cloud
  - Collaborative workflows (forks, pull-requests, etc.)
  - Issues management, etc.

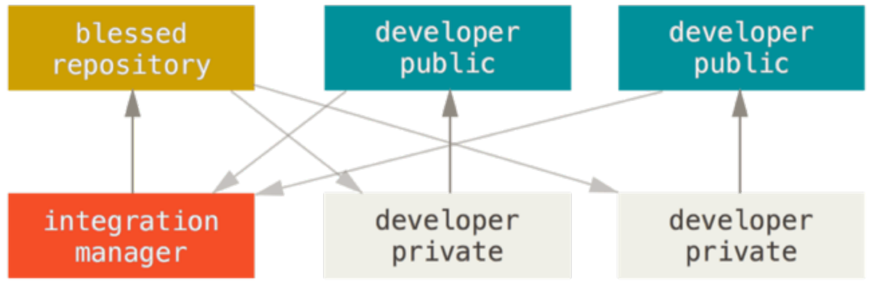
Git a distributed version control system

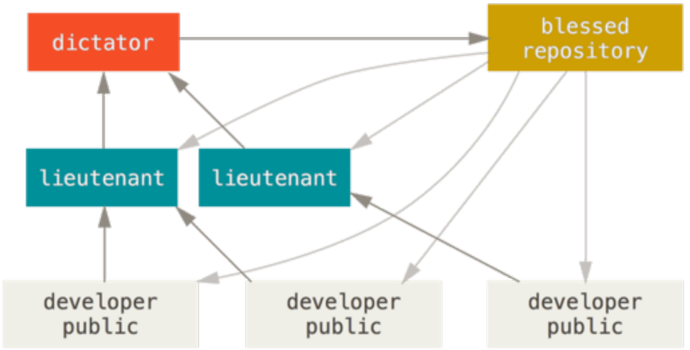
- The **entire repository**, with the **full revision history**, is stored on **every machine**.
- **No need to be connected** to the server to perform operations.
- **No absolute need for a server**: in theory, it is possible to use git on the client side (we will do that in this training).

Examples of topologies



- **All sorts of topologies are possible:**
  - One shared repo +
    one local repo per developer
  - One shared repo +
    one repo per team +
    one local repo per developer
  - etc.
- **You can give access rights**
  - to repositories
  - to branches





https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows

Different workflows operate on top of git

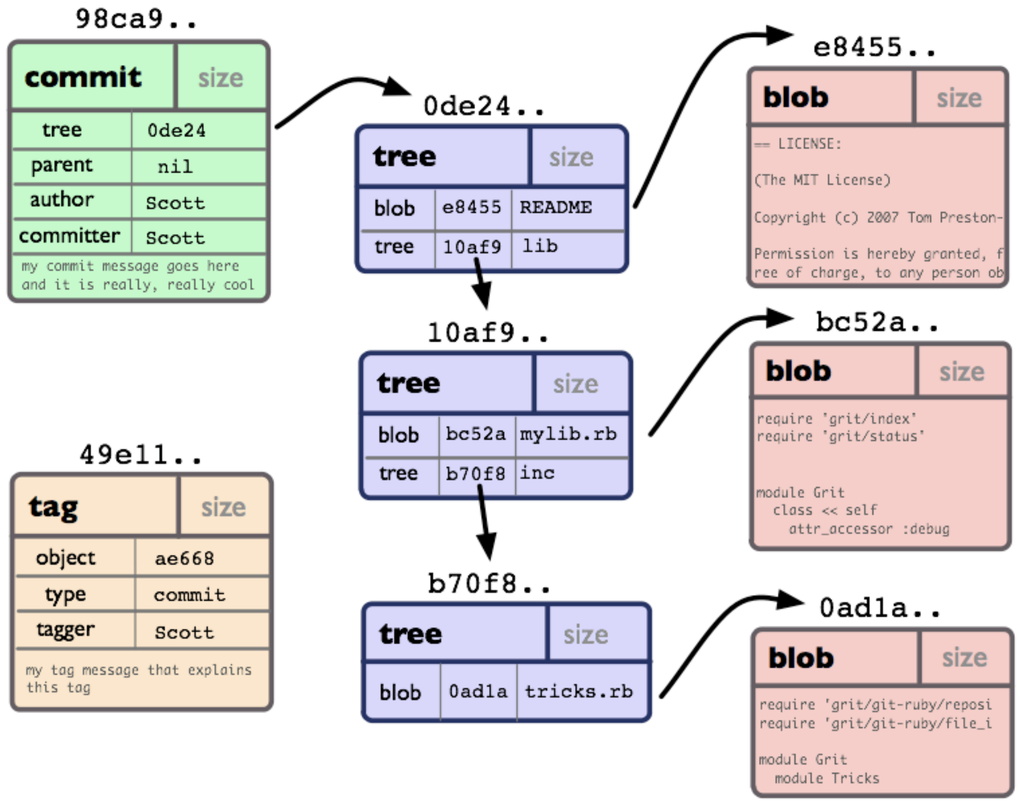
- **The developer defines a personal workflow** for local operations
  - How often do I record a snapshot?
  - What do I include?
  - How do I use private branches?
- **The team agrees on a collaborative workflow** to
  - Share snapshots
  - Review and accept contributions
  - Build a shared history

Git stores full snapshots, not deltas

- **Git is a "content-addressable file system"**
- Git uses a **key-value store**:
  - When you store a file in a repo, git computes a SHA-1 hash of its content.
  - The hash is used as a key to index the file in the store.
  - For this reason, two files with exactly the same content are stored only once in the git repository.
  - Go in the **.git hidden directory**, have a look at the **./objects directory** and you will find this key-value store.

Model: BLOBs, Trees and Commits

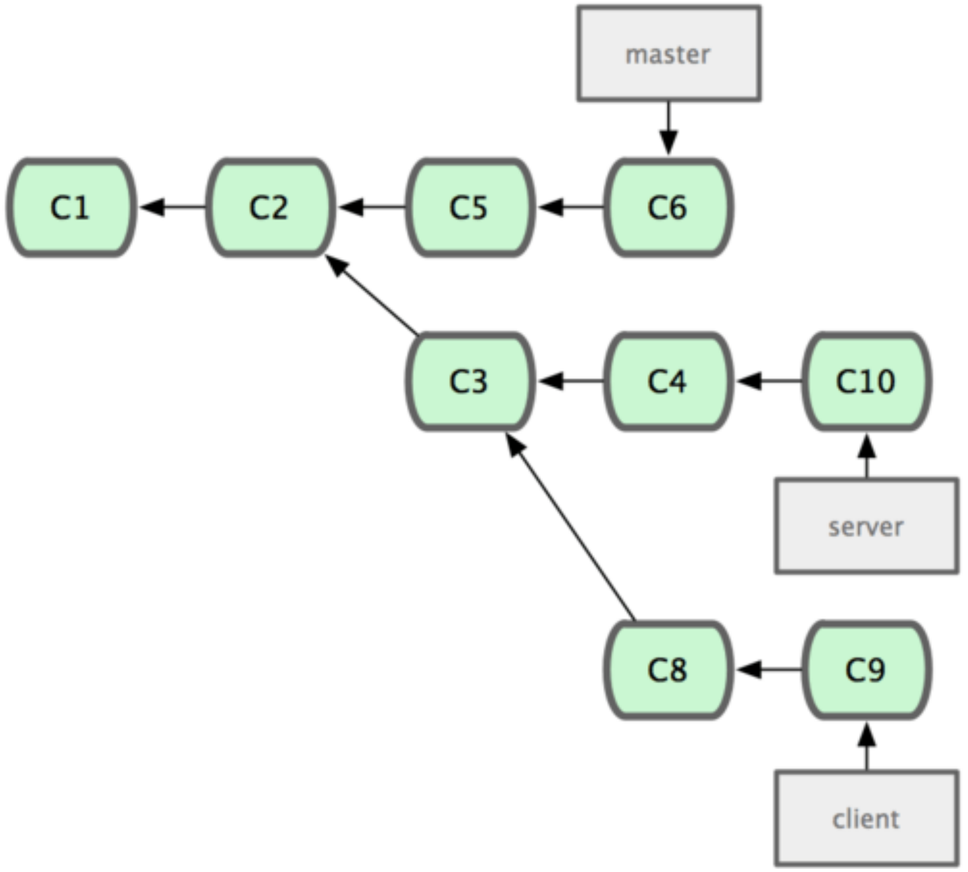
- **BLOBs** store **files**.
- **Trees** to store **directory** structures in the file system. A tree has **pointers** to subtrees and BLOBs.
- **Commits** identify **snapshots** in the history. Every commit points to a version of the **top-level tree**. A commit also stores **metadata**: author, parent, message, etc.
- **Annotated Tags** (created with **git tag -a**) are used to mark releases.
  - *Recommendation: do not use **lightweight tags** (created with git tag, without -a)*



http://shafiulazam.com/gitbook/1_the_git_object_model.html

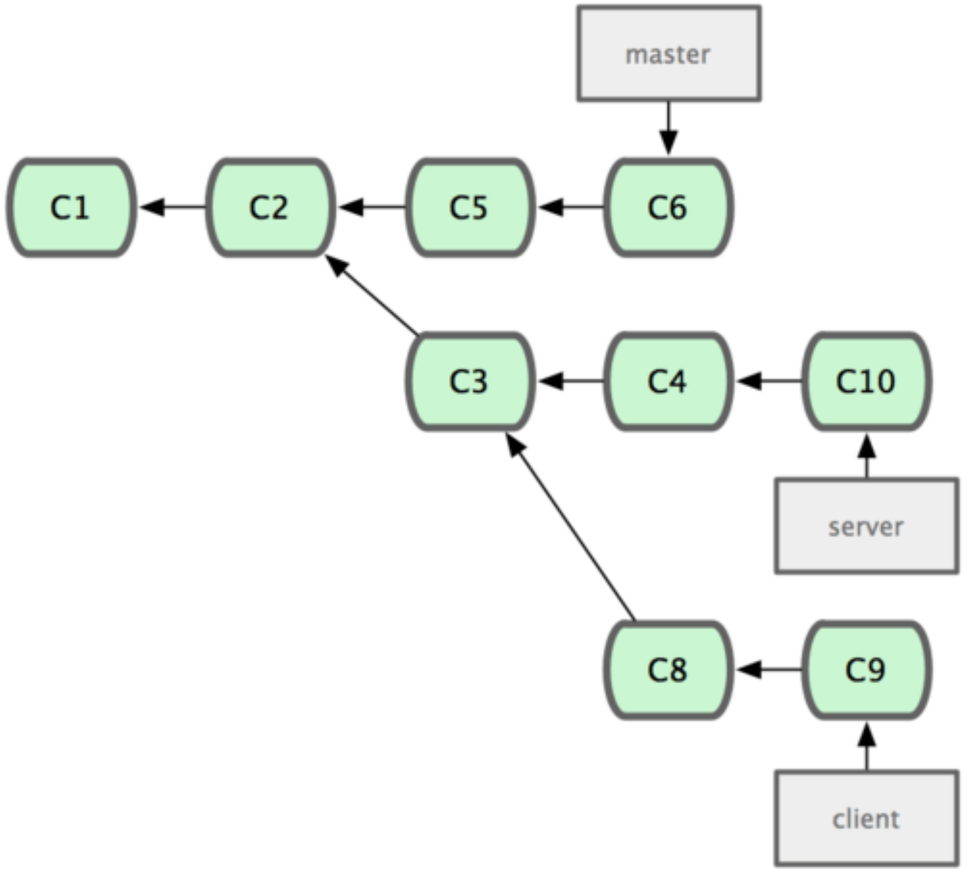
The Git history is a DAG structure

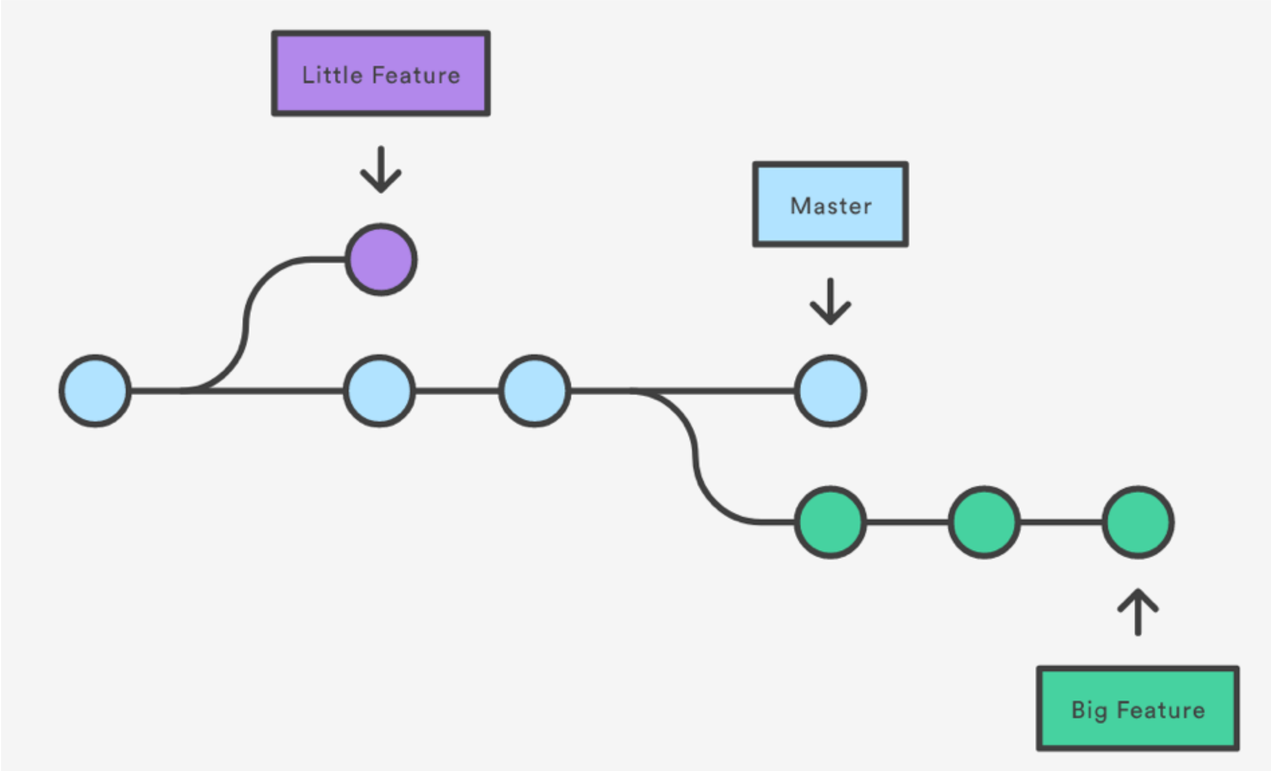
- **A DAG is a Directed Acyclic Graph**
  - Except for the initial commit, every commit has a least one parent.
  - Commits with more than one parent are "merge" commits.
- **In this example:**
  - C1 is the initial commit
  - There are no merge commits
  - master, server and client are branches
  - The branches have never been merged
  - The last commit on the client branch is C9
  - All branches have C1 and C2 as common ancestors



Git branches are simply pointers

- Branches are stored in files located in the **.git/refs/heads** directory.
- **Each branch is stored in a file**, which only contains the hash of a commit.
- What is **HEAD**?
  - HEAD means "**the tip of the current branch**" (last commit of the current branch)
  - When you checkout a branch, HEAD is the name of this branch.
- What is **HEAD^n**?
  - It is a reference to the nth parent of HEAD
  - HEAD^1 means the parent of the last commit
  - HEAD^2 means its grand-parent





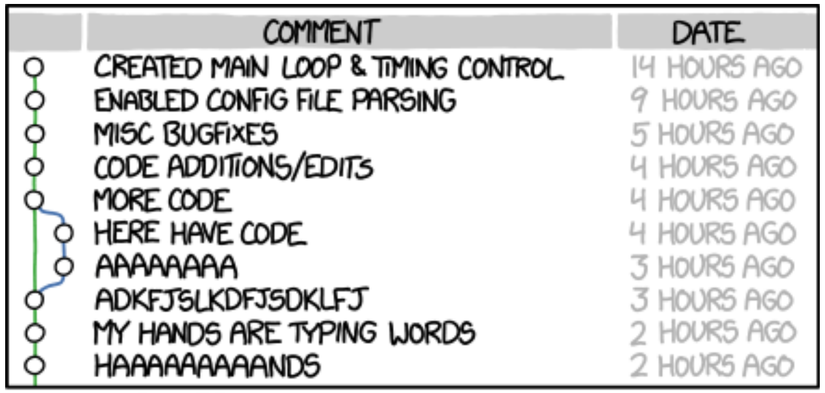
https://www.atlassian.com/git/tutorials/using-branches

Commands: get a local repository

`git init`	You want to **create a new repository** on your machine. This initialises **the .git hidden directory** in the current folder, where all objects are stored. Often, you rather start from an existing repository (on a server).
`git clone URL`	You want to copy a (usually remote) repository in the current folder. After the operation, **the .git hidden** directory contains all revisions of the the remote repository (commits, blobs and trees). After the operation, the **.git/config** file contains a named reference to the copied repository: **origin**.

Commands: record changes in the history

`git add`	You want to **prepare the content of the next snapshot** in the history (the next commit). You do a git add to **include the new and updated files** in the next revision. You "add" the file revisions to the **staging area**, also called index. After the operation, the file revisions are **not yet** in the repository.
`git commit`	You create a snapshot with the content of the staging area. The new commit points to the previous one. **Best practice**: do **not** use **git commit -m "short message"**, but prefer **git commit**, and use the editor to write a detailed message.



https://chris.beams.io/posts/git-commit/

The seven rules of a great Git commit message

> *Keep in mind: This has all been said before.*

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain *what* and *why* vs. *how*

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

Commands: what is going on?

`git status`	You want to know: - what has changed in the **working directory**? - what has been added to the **staging area**? - whether some files are **not tracked yet**? You may also want to know: - if **the current local branch is up-to-date with the corresponding remote branch** (e.g. is master up-to-date with origin/master)
`git log`	You want to inspect the error. There are a lot of options and ways to format the output. Try: **git log --oneline** Try: **git log --oneline --graph**

Commands: branches

`git checkout -b NEW_BRANCH_NAME`	You want to create AND checkout a branch in a single operation
`git checkout BRANCH_NAME`	BRANCH_NAME points to a snapshot in the revision history You want to replace the content of the working directory with this snapshot
`git branch`	You want to know on which branch you are. Pass arguments to create, delete, rename, etc. branches.
`git branch -a`	You want to list all branches in the repository (local and remotes)
`git merge BRANCH_NAME`	You want to merge BRANCH_NAME into the current branch
`git rebase BRANCH_NAME`	Rebase is an alternative to merge, which should be used carefully. We will not cover it in this training. See https://medium.com/@porteneuve/getting-solid-at-git-rebase-vs-merge-4fa1a48c53aa
`git tag -a`	You want to tag a commit, typically with a release number Using **-a** creates an **annotated tag** (vs lightweight), which is good.

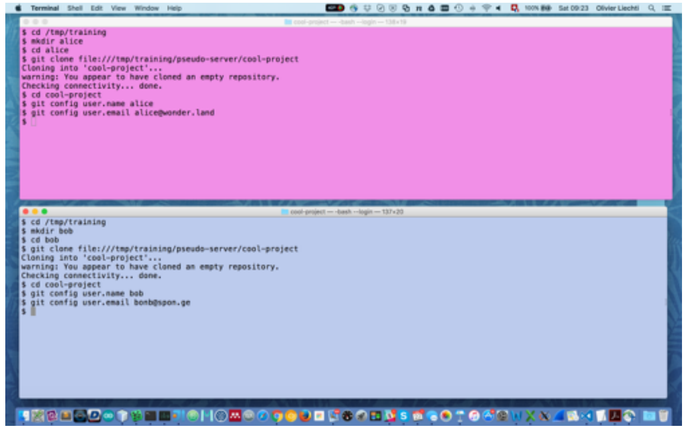
Commands: remote operations

`git push REPO BRANCH_NAME`	You want to send the commits on BRANCH_NAME to a remote REPO e.g. "git push origin master"
`git fetch REPO`	You want to retrieve the commits from a remote REPO, but not modify your local branches immediately.
`git pull --rebase` `git pull`	You want to retrieve the commits from a remote REPO and immediately rebase or master them into the current branch. **Make sure you are on the right branch!**
`git remote ...`	You want to configure a new remote repository (e.g. upstream), or update existing remotes. This command modifies **.git/config**.

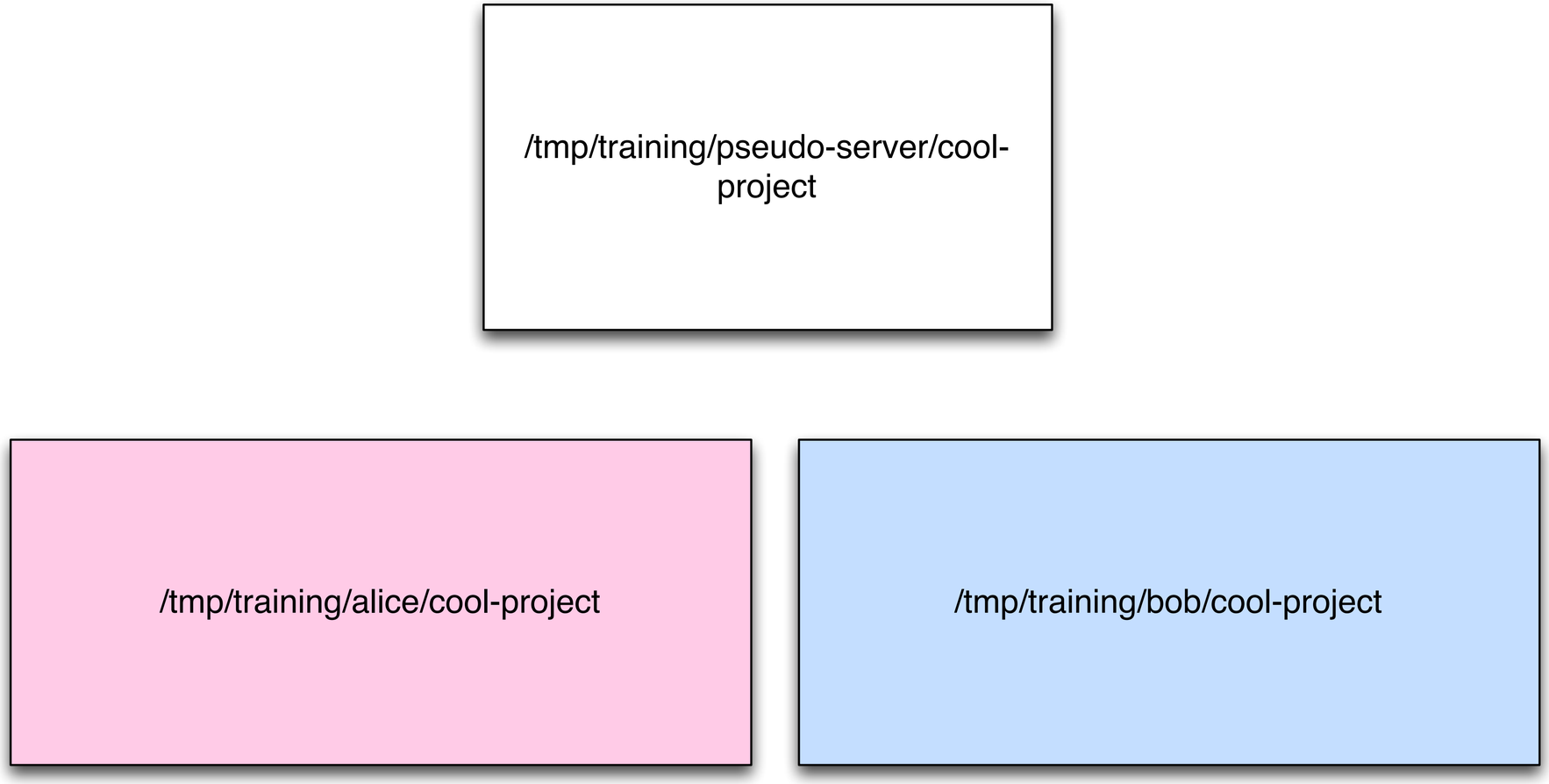
Git in practice with Alice and Bob

You don’t need a server to practice!

- **Create a “central” repo on your machine ("origin")**
- **Pretend to be “Alice”**
  - Open a terminal window, pick colour background
  - Create a directory on your machine
  - Clone the “central” repo
  - Tell git that you are Alice
- **Pretend to be “Bob”**
  - Open a second terminal, pick another colour background
  - Do the same operations, in another directory
- You have a nice environments to play with commands. **You won’t break anything, you won’t disturb anyone!**

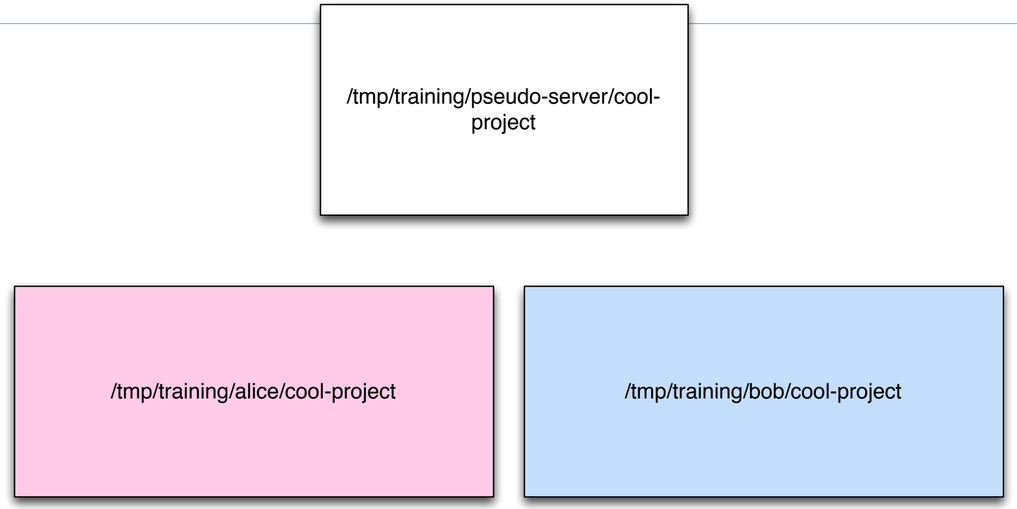


The setup



Step 1: create the “pseudo” central repo

- **In practice**, you will generally start with a repo hosted on GitHub, GitLab, BitBucket or at least on a remote server.
- **Here**, we do not want to depend on any external server. So, we need to create the repo locally.

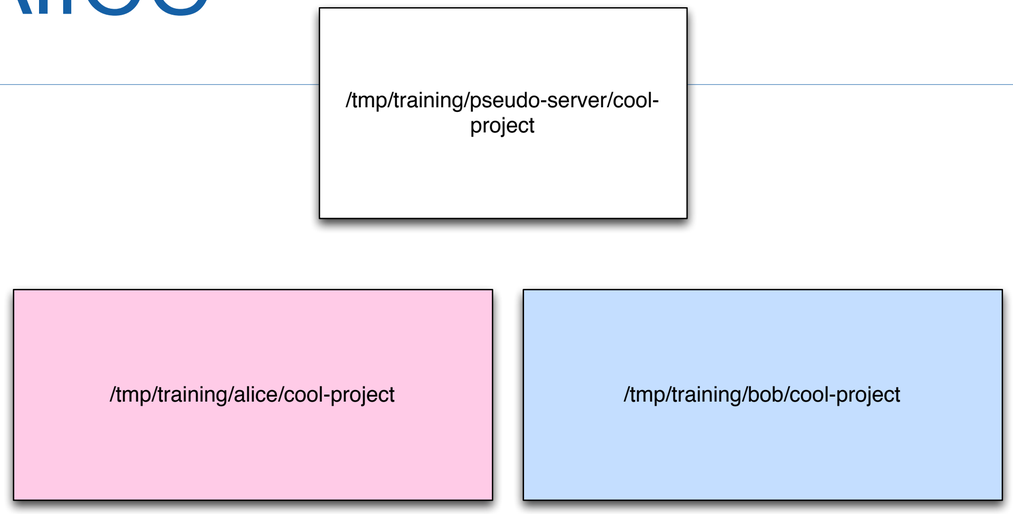


In practice, instead of doing this, you will create the repo on GitHub, GitLab, etc.

```
cd /tmp
mkdir training
cd training
mkdir pseudo-server
cd pseudo-server
mkdir cool-project
cd cool-project
git init --bare
```

Step 2: pretend to be Alice

- **Open a terminal window** and use it for a fake user named “Alice”
- **Pretend** that Alice is cloning the repo on her machine and contributes to the project.



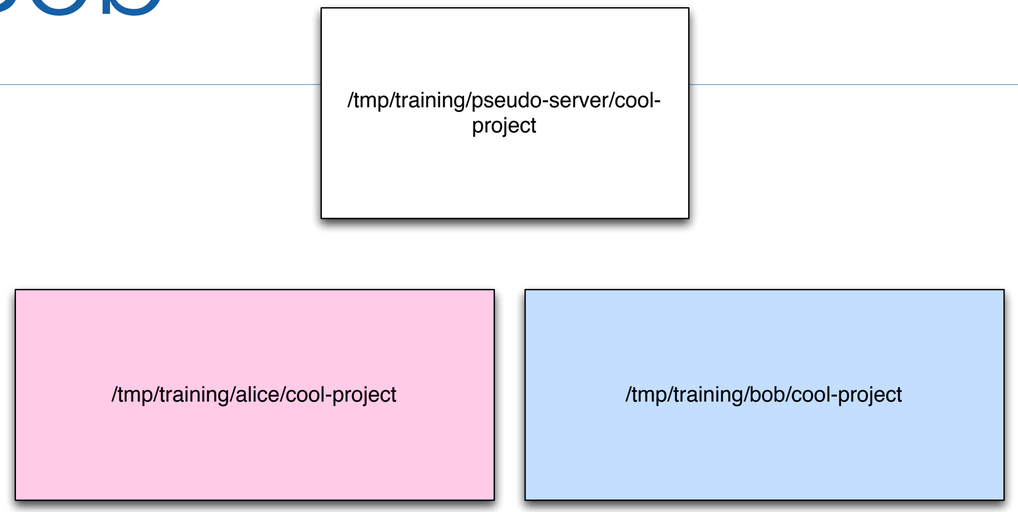
join the project and get a local repo

in this local repo, pretend to be Alice

```
cd /tmp/training
mkdir alice
cd alice
git clone file:///tmp/training/pseudo-server/cool-project
cd cool-project
git config user.name alice
git config user.email alice@wonder.land
```

Step 3: pretend to be Bob

- **Open a terminal window** and use it for a fake user named “Alice”
- **Pretend** that Alice is cloning the repo on her machine and contributes to the project.



join the project and get a local repo

in this local repo, pretend to be Alice

```
cd /tmp/training
mkdir bob
cd bob
git clone file:///tmp/training/pseudo-server/cool-project
cd cool-project
git config user.name bob
git config user.email bob@spon.ge
```

Terminal Shell Edit View Window Help — 100% Sat 09:23 Olivier Liechti

cool-project — -bash --login — 138×19

```
$ cd /tmp/training
$ mkdir alice
$ cd alice
$ git clone file:///tmp/training/pseudo-server/cool-project
Cloning into 'cool-project'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
$ cd cool-project
$ git config user.name alice
$ git config user.email alice@wonder.land
$
```

cool-project — -bash --login — 137×20

```
$ cd /tmp/training
$ mkdir bob
$ cd bob
$ git clone file:///tmp/training/pseudo-server/cool-project
Cloning into 'cool-project'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
$ cd cool-project
$ git config user.name bob
$ git config user.email bonb@spon.ge
$
```

Task 1: Alice **creates** a README.md file

```
echo "Welcome to the Cool Project" > README.md
git status
```

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

   README.md

nothing added to commit but untracked files present (use "git add" to track)
```

At this moment, the new file is not tracked by git yet. If we want to include it in the next commit, we need to **"add" it to the staging area**.

```
git add README.md
git status
```

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

   new file:   README.md
```

Done! If we commit now, the file will be included in the **snapshot** pointed by the commit.

```
git commit -m "Write 1st draft"
git status
```

```
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)
nothing to commit, working directory clean
```

A new commit has been added to the **local** history.

Task 2: Alice **updates** a README.md file

```
echo "Greetings from Alice" >> README.md
git status
```

```
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Git tells us that there is a difference between the last commit and the working directory. If we want to include it in the next commit, we once again need to **add it to the staging area**.

```
git add README.md
git status
```

```
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   modified:   README.md
```

Done! If we commit now, the file will be included in the **snapshot** pointed by the commit.

```
git commit -m "Improve README.md"
git status
```

```
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)
nothing to commit, working directory clean
```

A new commit has been added to the **local** history.

Task 3: Alice and Bob look at the **history**

`git log`

```
commit 3c270b3da8f7ffc86589d15f050fc70c2c546d34
Author: alice <alice@wonder.land>
Date:   Sat Jul 22 11:01:27 2017 -0300

    Improve README.md

commit 2bf2e53cc804419debd5a3c58032f7ad23171c19
Author: alice <alice@wonder.land>
Date:   Sat Jul 22 09:35:29 2017 -0300

    Write 1st draft
```

`git log`

```
fatal: your current branch 'master' does not have
any commits yet
```

git log shows the history of the **local repository**.

Alice has created two snapshots (two commits), but has not shared them with the team yet.

Bob has not done anything yet. His local history is empty.

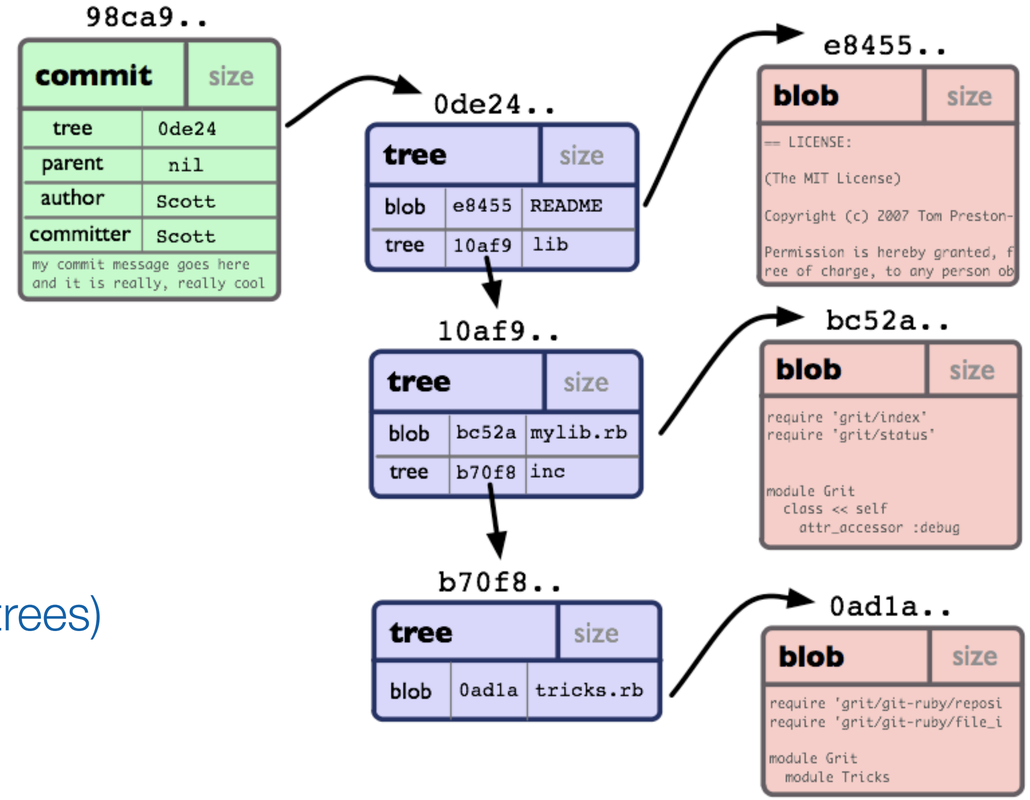
`git log --oneline`

```
3c270b3 Improve README.md
2bf2e53 Write 1st draft
```

`git log --oneline`

```
fatal: your current branch 'master' does not have
any commits yet
```

Behind the scenes



The git object model

Alice has created 6 git objects:

- every file version = 1 BLOB (2 BLOBs)
- every file version = 1 Tree, pointing at the right BLOB (2 trees)
- every snapshot in the history = 1 Commit (2 commits)

Behind the scenes...

```
tree .git
```

The **.git hidden directory** is the local repository.
.git/config is a text file with config properties (e.g. user.name)
.git/objects is the key-value store for BLOBs, trees, commits

```
git log --oneline
3c270b3 Improve README.md
2bf2e53 Write 1st draft
```

get sha-1 hash of commits

```
git cat-file -p 3c270b3
tree 395ca1e5bb0e756470851c881331c8d4007a0b12
parent 2bf2e53cc804419debd5a3c58032f7ad23171c19
author alice <alice@wonder.land> 1500732087 -0300
committer alice <alice@wonder.land> 1500732087 -0300

Improve README.md
```

display commit object

```
git cat-file -p 395ca1e5bb0e756470851c881331c8d4007a0b12
100644 blob de3a48fbcf6c6866cfc64d522b089ac2e663ca0d README.md
```

display tree object

```
git cat-file -p de3a48fbcf6c6866cfc64d522b089ac2e663ca0d
Welcome to the Cool Project
Greetings from Alice
```

display BLOB object

```
.git/
├── COMMIT_EDITMSG
├── HEAD
├── branches
├── config
├── description
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── prepare-commit-msg.sample
│   └── update.sample
├── index
├── info
│   └── exclude
├── logs
│   ├── HEAD
│   └── refs
│       └── heads
│           └── master
├── objects
│   ├── 2b
│   │   └── f2e53cc804419debd5a3c58032f7ad23171c19
│   ├── 39
│   │   └── 5ca1e5bb0e756470851c881331c8d4007a0b12
│   ├── 3c
│   │   └── 270b3da8f7ffc86589d15f050fc70c2c546d34
│   ├── 62
│   │   └── c853853e3aaf0d67fc6f5042a2497c1661955a
│   ├── d2
│   │   └── 7c3cdf0f05bdd791f8b272010413dbbaefe88f
│   ├── de
│   │   └── 3a48fbcf6c6866cfc64d522b089ac2e663ca0d
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   └── master
    └── tags

18 directories, 24 files
```

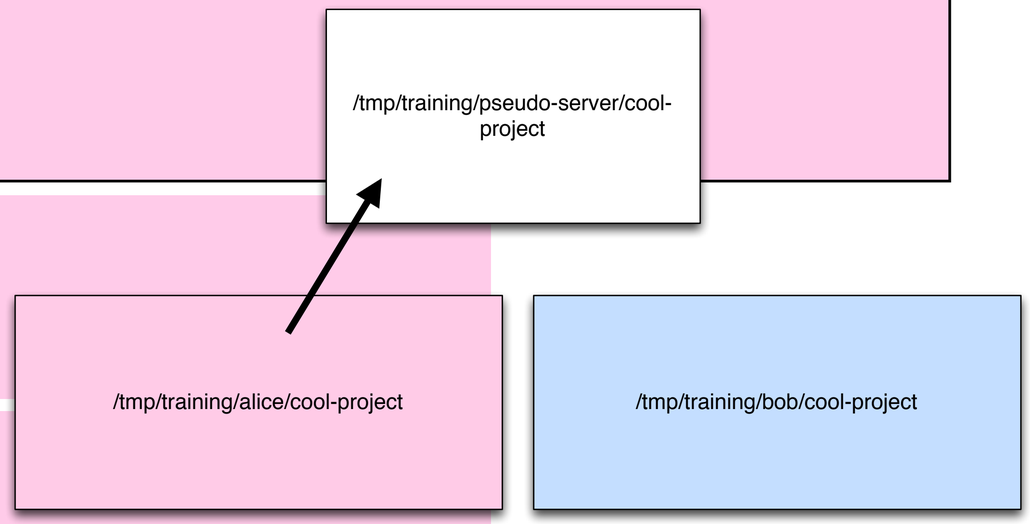
Task 4: Alice **shares** her snapshots (push)

```
git status
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)
nothing to commit, working directory clean
```

`git push origin master`

```
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 500 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To file:///tmp/training/pseudo-server/cool-project
 * [new branch]      master -> master
```

we send the commits on the local master branch to the origin repository (the pseudo-server)




```
git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

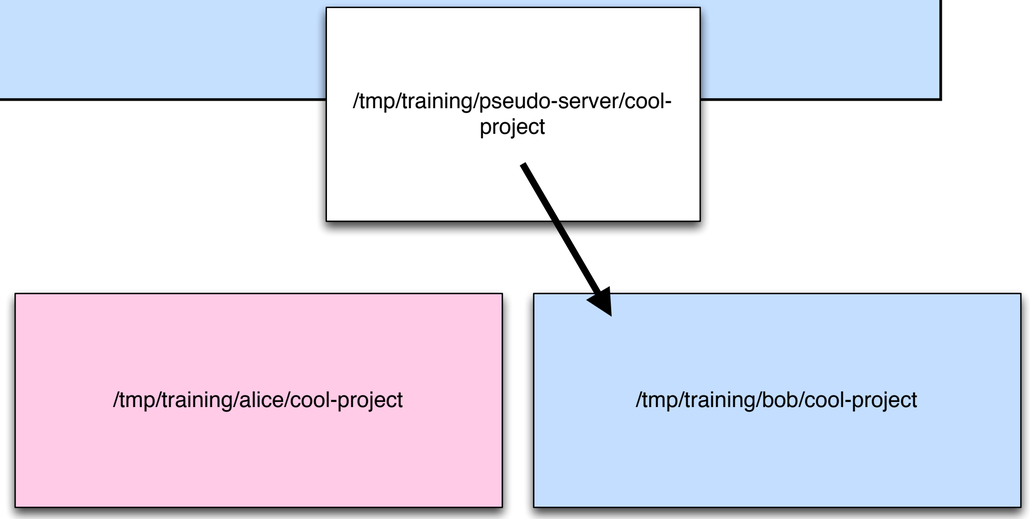
```
git branch -a
* master
  remotes/origin/master
```

Task 5a: Bob **fetches** the commits

```
git status
On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

```
git fetch origin
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From file:///tmp/training/pseudo-server/cool-project
 * [new branch]      master     -> origin/master
```

We do not use git pull. We use git fetch to proceed in 2 steps. In the first step, we only fetch Alice's commits, but we don't merge them on our local branch.




```
git log --oneline
fatal: your current branch 'master' does not have any commits yet
```

```
tree .git/objects/
.git/objects/
├── 2b
│   └── f2e53cc804419debd5a3c58032f7ad23171c19
├── 39
│   └── 5ca1e5bb0e756470851c881331c8d4007a0b12
├── 3c
│   └── 270b3da8f7ffc86589d15f050fc70c2c546d34
├── 62
│   └── c853853e3aaf0d67fc6f5042a2497c1661955a
├── d2
│   └── 7c3cdf0f05bdd791f8b272010413dbbaefe88f
├── de
│   └── 3a48fbcf6c6866cfc64d522b089ac2e663ca0d
├── info
└── pack
```

The sha-1 keys and values in Alice's and Bob's local repositories are the same.

Task 5b: Bob **merges** commits on his branch

```
git merge origin/master
git log --oneline --graph
* 3c270b3 Improve README.md
* 2bf2e53 Write 1st draft
```

This means that we merge the branch named origin/master onto the current branch.

```
cat .git/HEAD
ref: refs/heads/master
```

The **local master** branch now points to the same commit as the **origin/master branch**.

```
cat .git/refs/heads/master
3c270b3da8f7ffc86589d15f050fc70c2c546d34

cat .git/refs/remotes/origin/master
3c270b3da8f7ffc86589d15f050fc70c2c546d34
```

Task 6: Bob makes a **contribution**

```
$ echo "This is the story of..." > jokes.txt
$ echo "Checkout my jokes in the jokes.txt file" >> README.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

   jokes.txt

no changes added to commit (use "git add" and/or "git commit -a")

$ git add README.md
$ git add jokes.txt

$ git commit -m "Add jokes"
[master 2e1786f] Add jokes
 2 files changed, 2 insertions(+)
 create mode 100644 jokes.txt

$ git push origin master
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 373 bytes | 0 bytes/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To file:///tmp/training/pseudo-server/cool-project
   3c270b3..2e1786f  master -> master
```

Bob **modifies in the working directory** (1 file modified and 1 file updated)

Bob **prepares the next commit** by adding the new and modified files in the staging area (index)

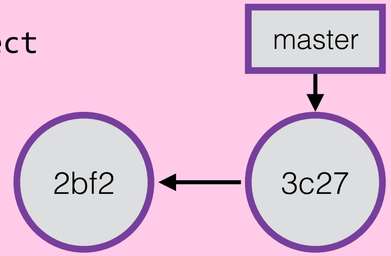
Bob **creates the commit**.

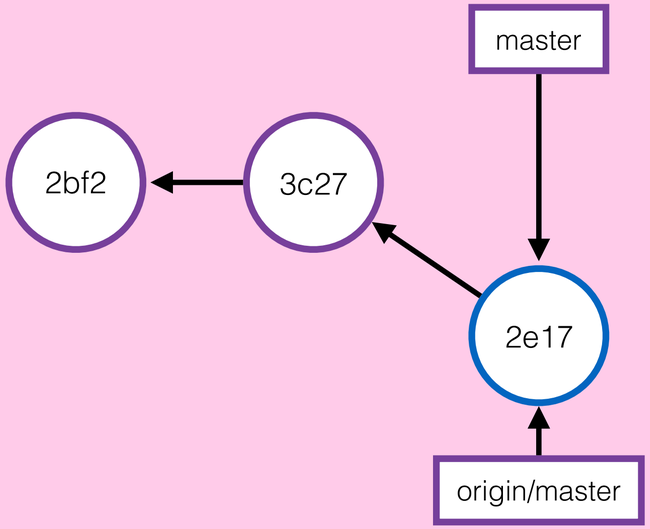
Bob **pushes** the commits on the local master branch to the origin repository.

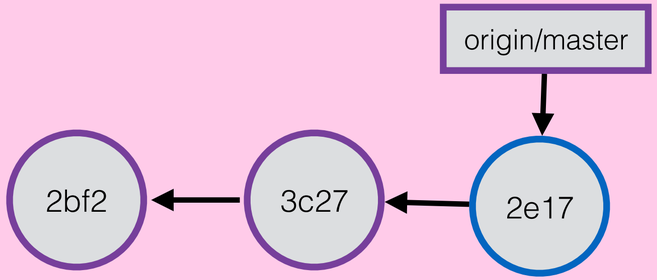
Task 7: Alice gets Bob's contribution

```
$ git fetch
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///tmp/training/pseudo-server/cool-project
   3c270b3..2e1786f  master     -> origin/master

$ git log --oneline
3c270b3 Improve README.md
2bf2e53 Write 1st draft

$ git merge origin/master
Updating 3c270b3..2e1786f
Fast-forward
 README.md | 1 +
 jokes.txt | 1 +
 2 files changed, 2 insertions(+)
 create mode 100644 jokes.txt

$ git log --oneline
2e1786f Add jokes
3c270b3 Improve README.md
2bf2e53 Write 1st draft
```

Alice **fetches the commits**, but does not modify her local branch yet.







Alice **merges origin/master** onto her local master branch. Because she did not work create any commit while Bob was working, git is able to do a "**fast forward merge**".

Task 8: Alice and Bob work in parallel... (1)

```
$ echo "I do this while Bob works" > bob-file.txt
$ git add bob-file.txt
$ git commit -m "Work in parallel on bob-file.txt"
[master a001db2] Work in parallel on bob-file.txt
 1 file changed, 1 insertion(+)
 create mode 100644 bob-file.txt
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 346 bytes | 0 by
Total 3 (delta 0), reused 0 (delta 0)
To file:///tmp/training/pseudo-server/cool-pr
   2e1786f..a001db2  master -> master
```

```
$ echo "I do this while Alice works" > alice-file.txt
$ git add alice-file.txt
$ git commit -m "Work in parallel on alice-file.txt"
[master 99ea7b7] Work in parallel on alice-file.txt
 1 file changed, 1 insertion(+)
 create mode 100644 alice-file.txt
$ git push origin master
To file:///tmp/training/pseudo-server/cool-project
 ! [rejected]        master -> master (fetch first)
                                                    cool-project'
                                                    o
                                                    shing
                                                    es
                                                    s.
```

```
$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From file:///tmp/training/pseudo-server/cool-project
   2e1786f..a001db2  master     -> origin/master
$ git push origin master
To file:///tmp/training/pseudo-server/cool-project
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'file:///tmp/training/pseudo-server/cool-project'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

... but Alice is a bit faster. Bob

Task 8: Alice and Bob work in parallel... (2)

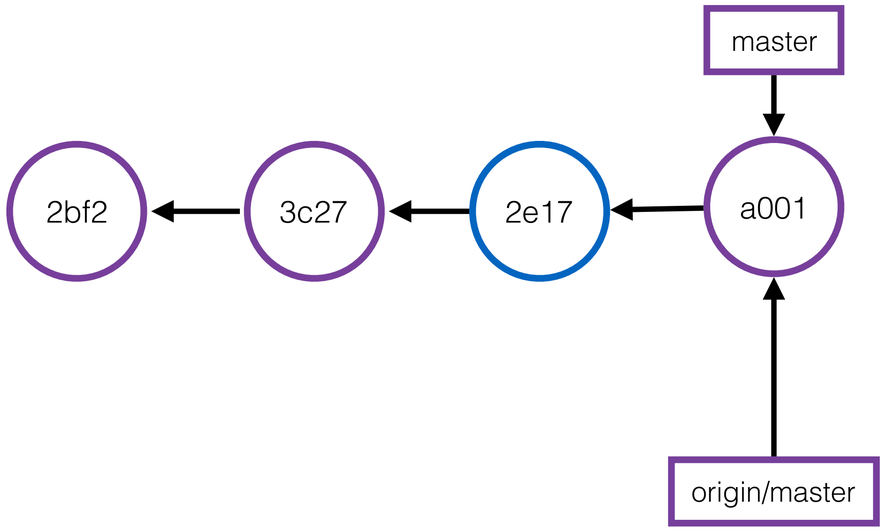
```
$ echo "I do this while Bob works" > bob-file.txt
$ git add bob-file.txt
$ git commit -m "Work in parallel on bob-file.txt"
[master a001db2] Work in parallel on bob-file.txt
 1 file changed, 1 insertion(+)
 create mode 100644 bob-file.txt
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 346 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To file:///tmp/training/pseudo-server/cool-project
   2e1786f..a001db2  master -> master
```

```
$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From file:///tmp/training/pseudo-server/cool-project
   2e1786f..a001db2  master     -> origin/master
$ git push origin master
To file:///tmp/training/pseudo-server/cool-project
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'file:///tmp/training/pseudo-server/cool-project'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
nothing to commit, working directory clean
```

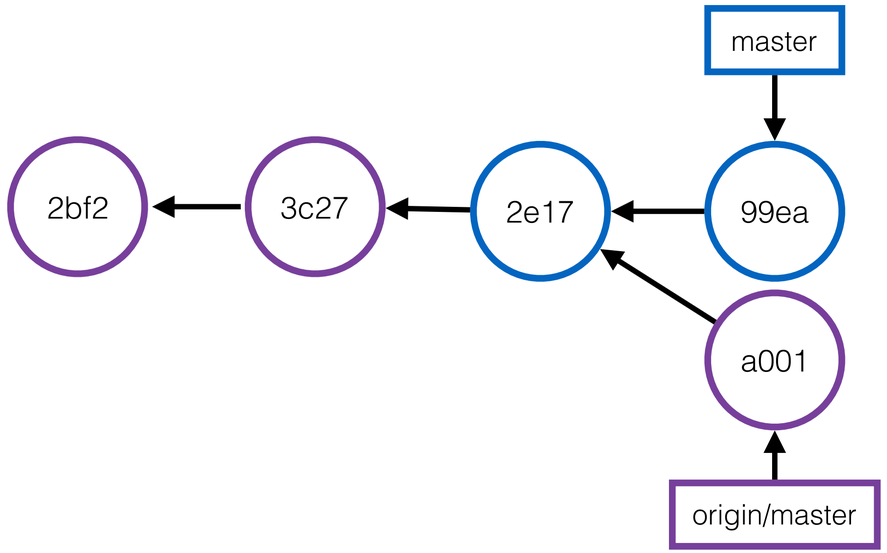
A **git fetch is not enough.** Git wants us to merge.

Task 8: Alice and Bob work in parallel... (2)

```
$ git log --oneline --graph
* a001db2 Work in parallel on bob-file.txt
* 2e1786f Add jokes
* 3c270b3 Improve README.md
* 2bf2e53 Write 1st draft
```




```
$ git log --oneline --graph
* 99ea7b7 Work in parallel on alice-file.txt
* 2e1786f Add jokes
* 3c270b3 Improve README.md
* 2bf2e53 Write 1st draft
```



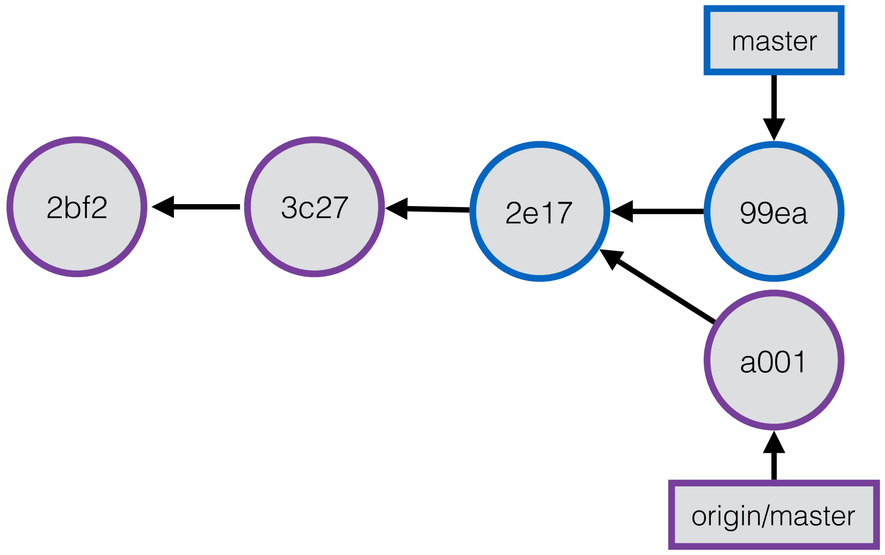
A **git fetch is not enough.** Git wants us to merge.

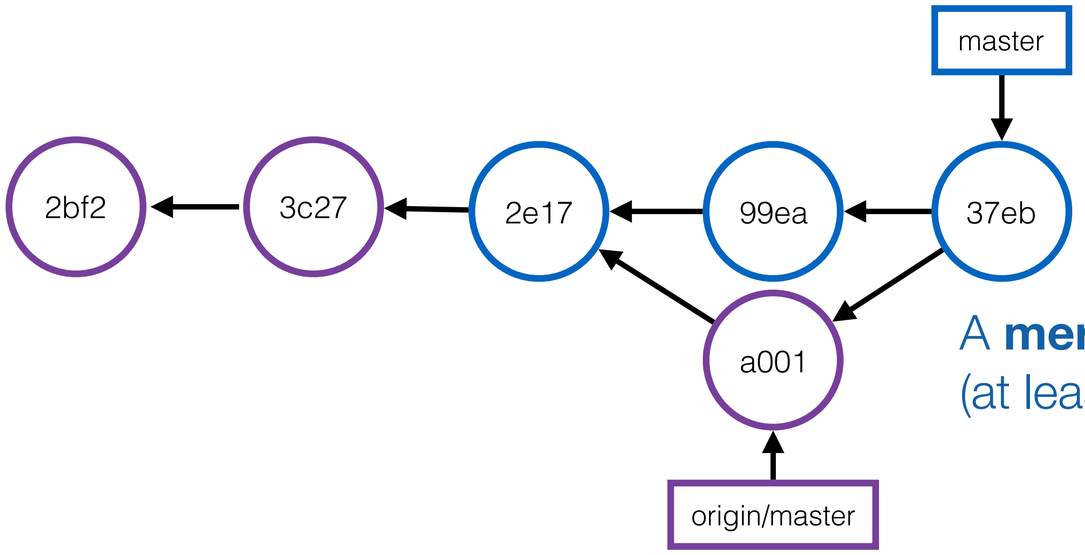
Task 8: Alice and Bob work in parallel... (3)

```
$ git merge origin/master
Merge made by the 'recursive' strategy.
 bob-file.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 bob-file.txt
```

```
$ git log --oneline --graph
*   37eb68d Merge remote-tracking branch 'origin/master'
|\
| * a001db2 Work in parallel on bob-file.txt
* | 99ea7b7 Work in parallel on alice-file.txt
|/
* 2e1786f Add jokes
* 3c270b3 Improve README.md
* 2bf2e53 Write 1st draft
```

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
```

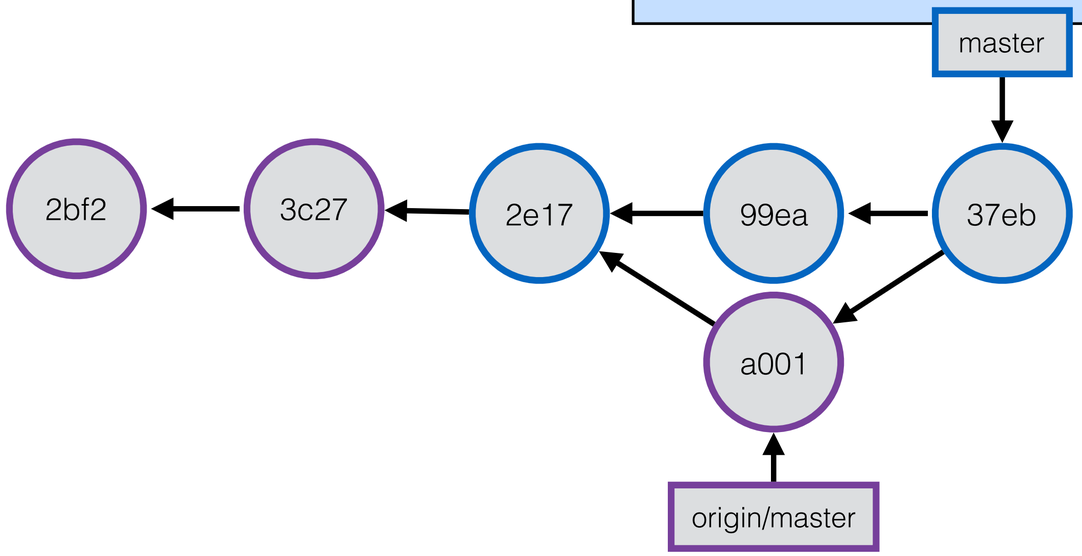


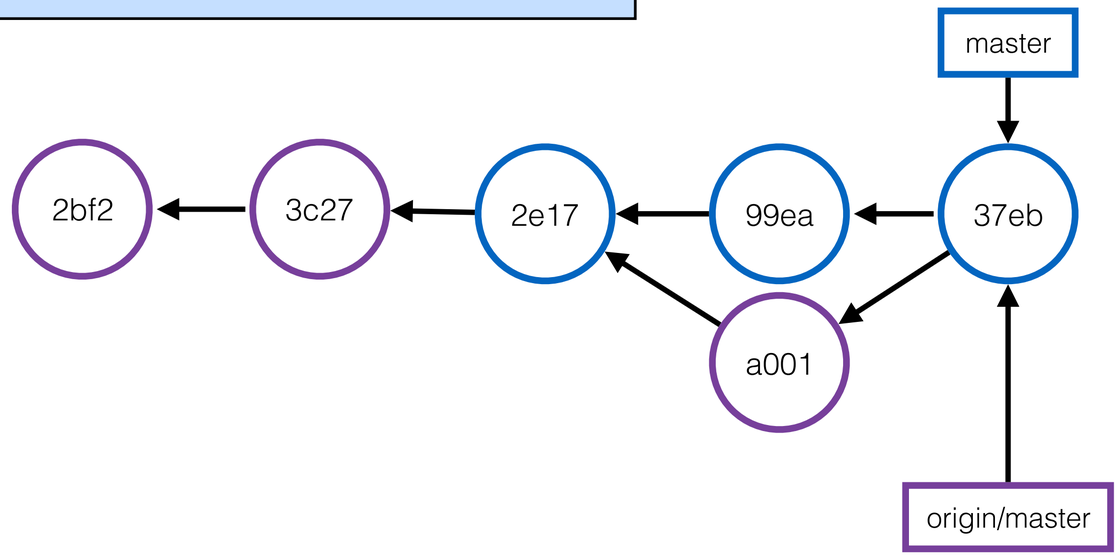


A **merge commit** has (at least) 2 parents

Task 8: Alice and Bob work in parallel... (4)

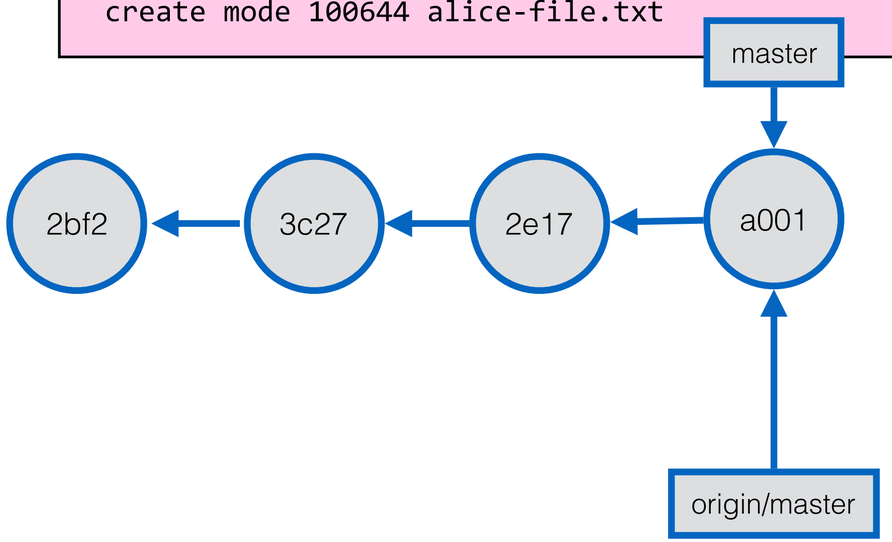
```
$ git push origin master
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 587 bytes | 0 bytes/s, done.
Total 5 (delta 1), reused 0 (delta 0)
To file:///tmp/training/pseudo-server/cool-project
   a001db2..37eb68d  master -> master
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

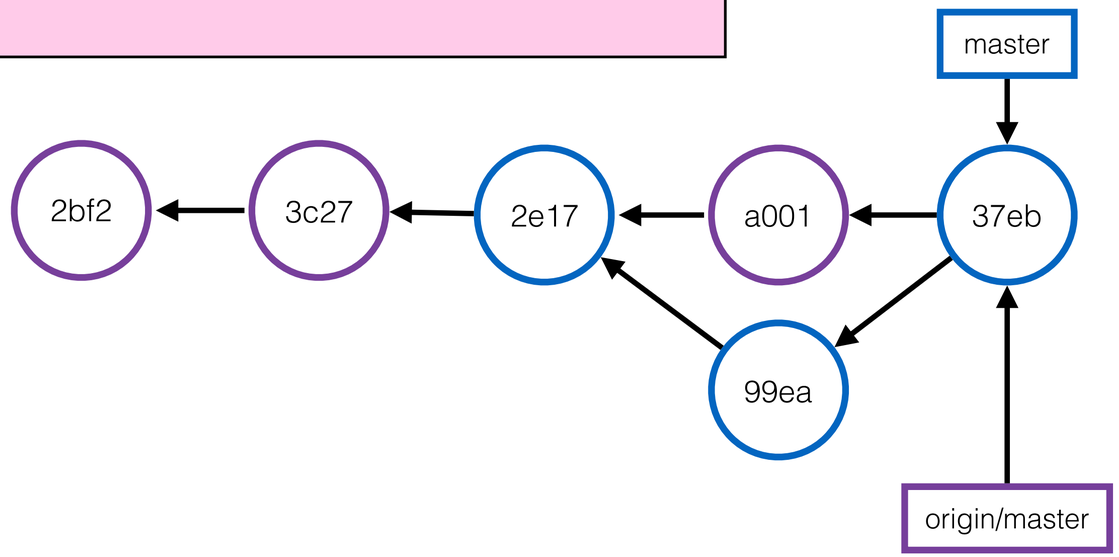




Task 8: Alice and Bob work in parallel... (4)

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From file:///tmp/training/pseudo-server/cool-project
   a001db2..37eb68d  master     -> origin/master
$ git merge origin/master
Updating a001db2..37eb68d
Fast-forward
 alice-file.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 alice-file.txt
```





Task 9: parallel updates & conflict resolution (1)

```
echo "aaa" > shared.md
git add shared.md
git commit -m "A v1"
git push origin master
```

```
git pull
echo "bbb" >> shared.md
git add shared.md
git commit -m "B v2a"
git push origin master
```

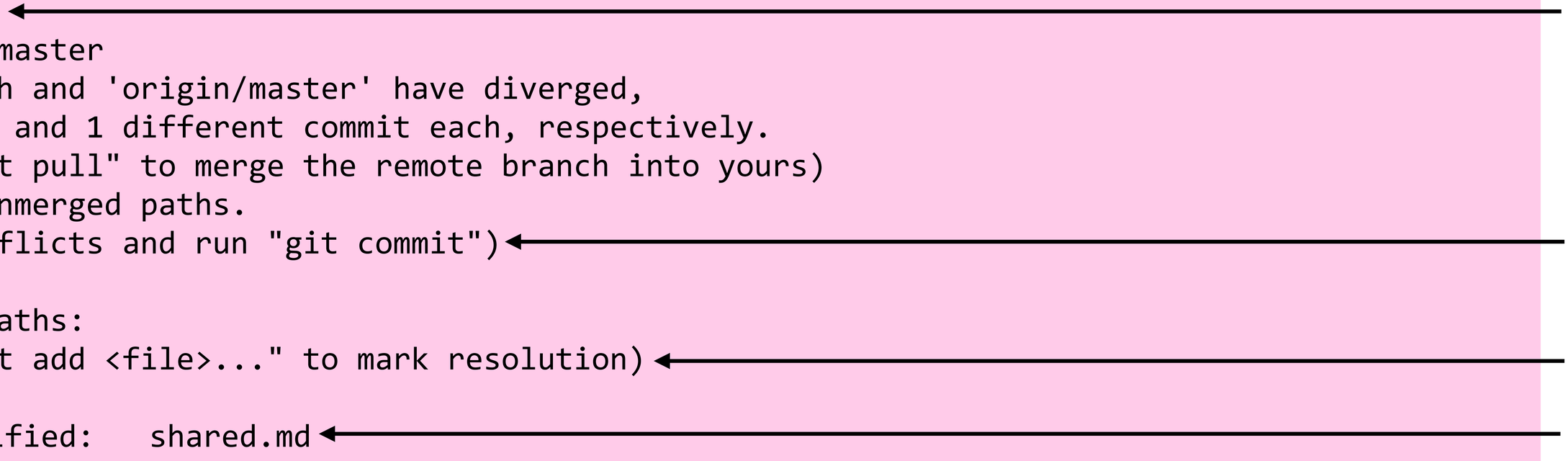
```
echo "ccc" >> shared.md
git add shared.md
git commit -m "A v2b"
```

```
git push origin master
git fetch origin
git merge origin/master
```

```
Auto-merging shared.md
CONFLICT (content): Merge conflict in shared.md
Automatic merge failed; fix conflicts and then commit the result.
```

Task 9: parallel updates & conflict resolution (2)

```
Auto-merging shared.md
CONFLICT (content): Merge conflict in shared.md
Automatic merge failed; fix conflicts and then commit the result.

git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

  both modified:   shared.md

no changes added to commit (use "git add" and/or "git commit -a")
```



Task 9: parallel updates & conflict resolution (2)

```
vi shared.md

aaa
<<<<<<< HEAD
ccc
=======
bbb
>>>>>>> origin/master
```

```
git add shared.md
commit
git push origin master
```

Task 10: Alice develops a feature on a branch

```
git branch

git checkout -b "feature-branch-001"
git branch

echo "hello" > Feature1.java
git add Feature1.java
git commit -m "Implement my feature"

git log --oneline

git checkout master
git log --oneline

git merge --no-ff feature-branch-001 -m "Merge ready-to-go feature"
git log --oneline --graph
```

Create a new branch, where we can work in isolation

Commit the new files on the feature branch

Move back to master branch

Merge feature into master

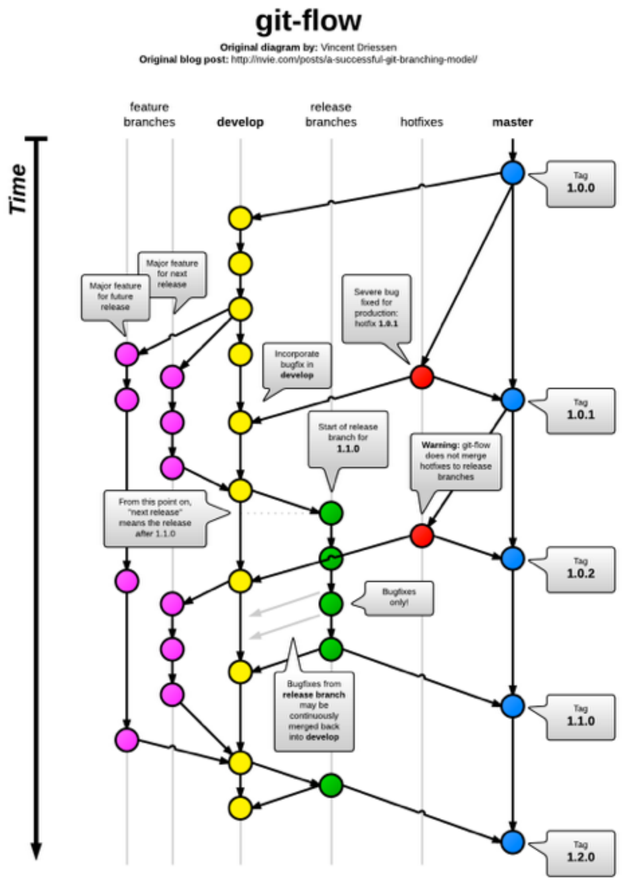
--no-ff will show the parallel work in the history (more on this later)

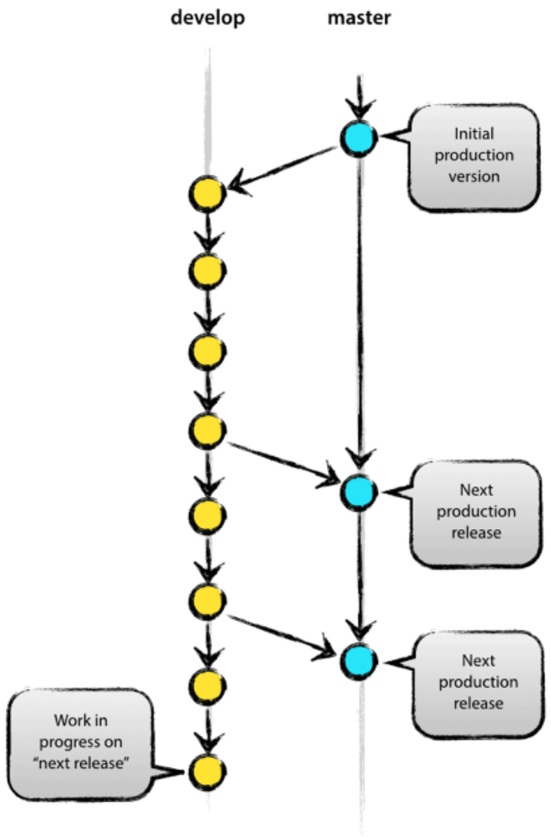
Collaboration workflows

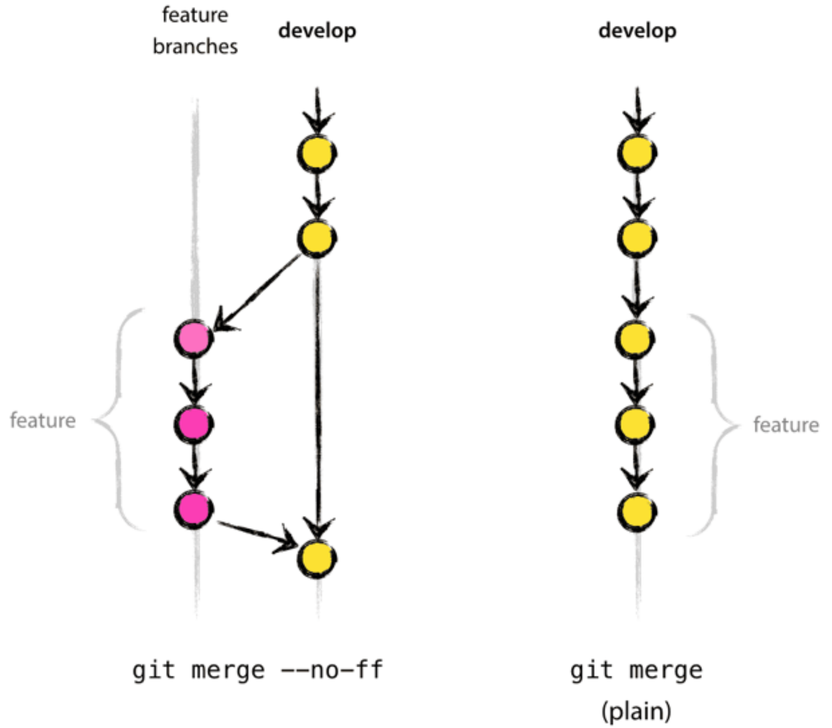
Why do we need collaborative workflows?

- **In the previous examples:**
  - We have used only one branch (master)
  - We have assumed that Alice and Bob had write access to the repo
- **In projects:**
  - Developers work in parallel and don't want to disturb each other
  - Contributions need to be reviewed before being integrated
- **We need to agree on some rules:**
  - How do do use branches, how do we communicate, who is merging.

Examples



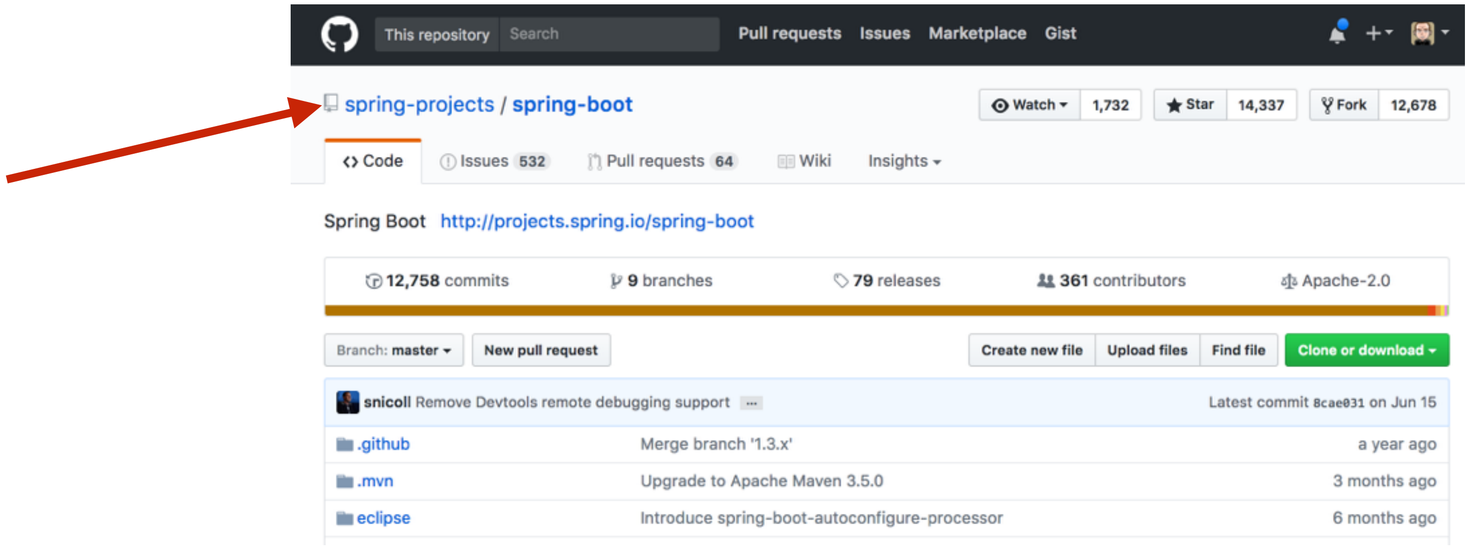




http://nvie.com/posts/a-successful-git-branching-model/

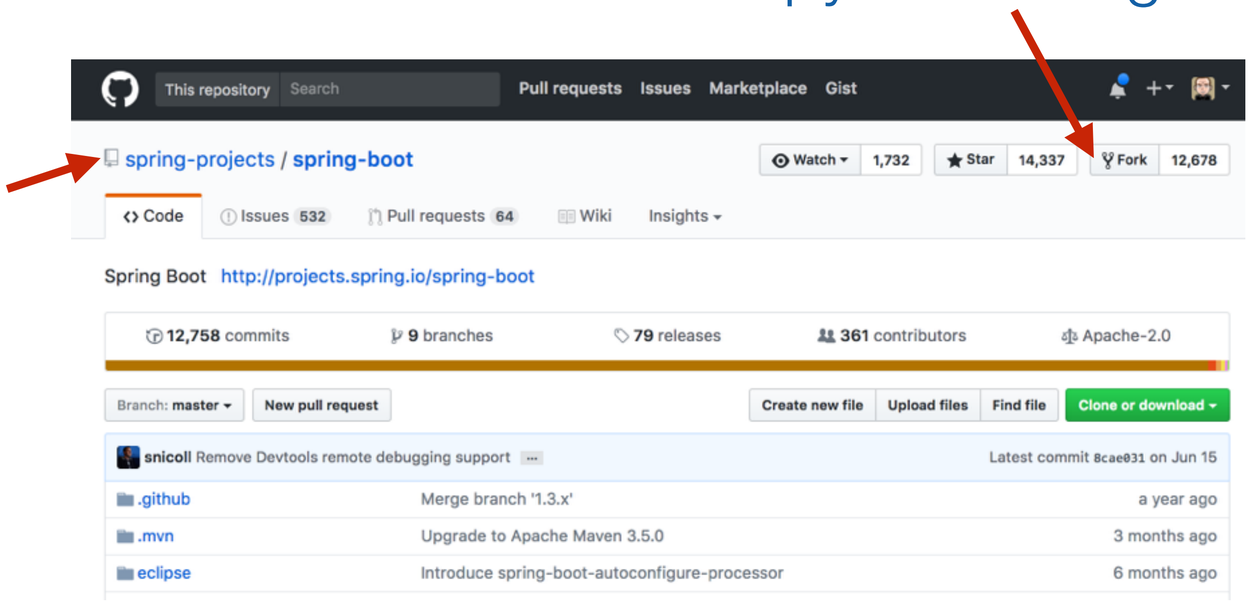
The GitHub workflow

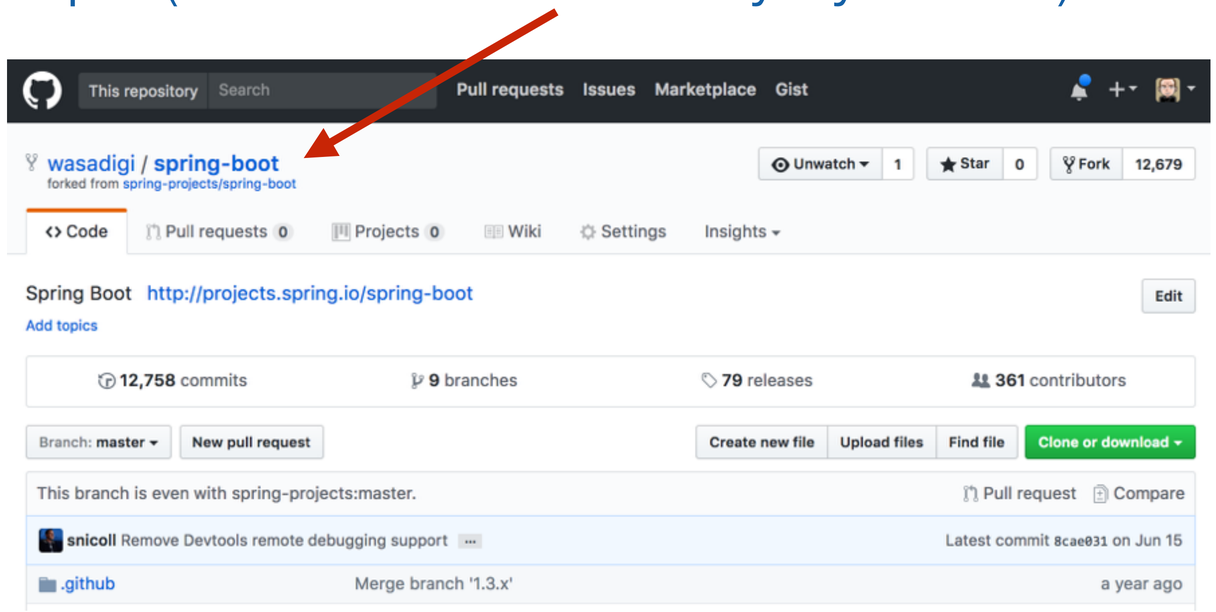
- **There core team creates a repository**
  - Only a few people have the right to write (commit directly)
  - Most people can clone the repo, work locally, but cannot push



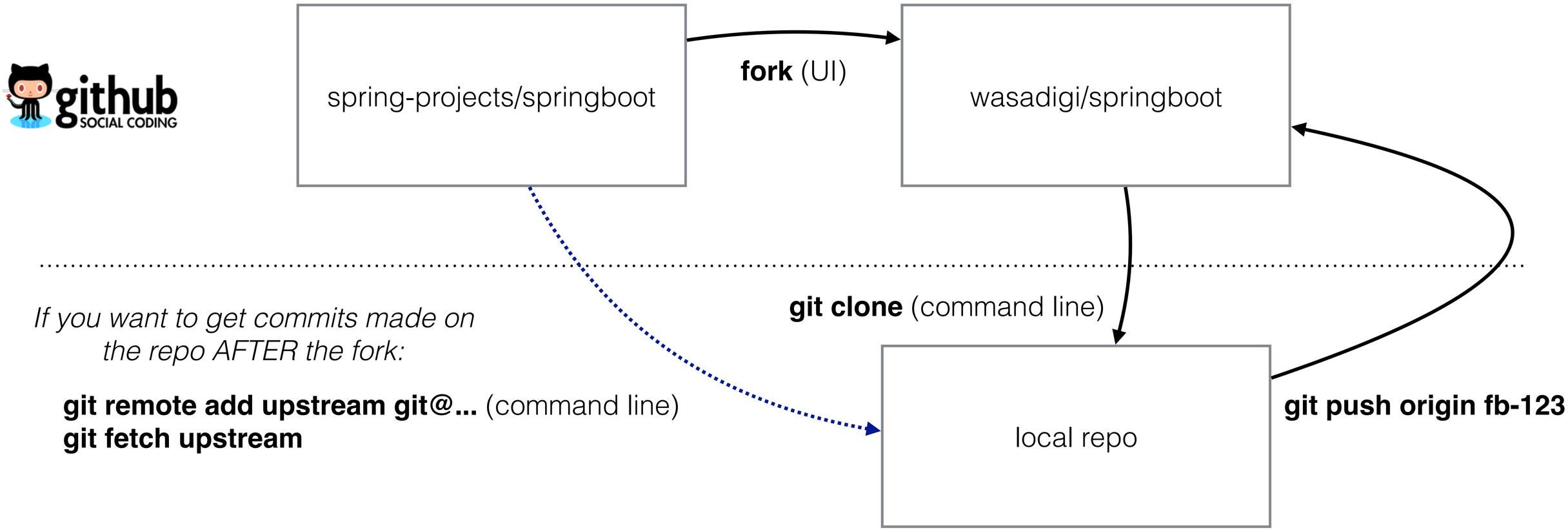
The GitHub workflow

- **A developer wants to contribute**
  - Via the GitHub UI, he forks the repository
  - He now owns a copy of the original repo (it is not automatically synched!)



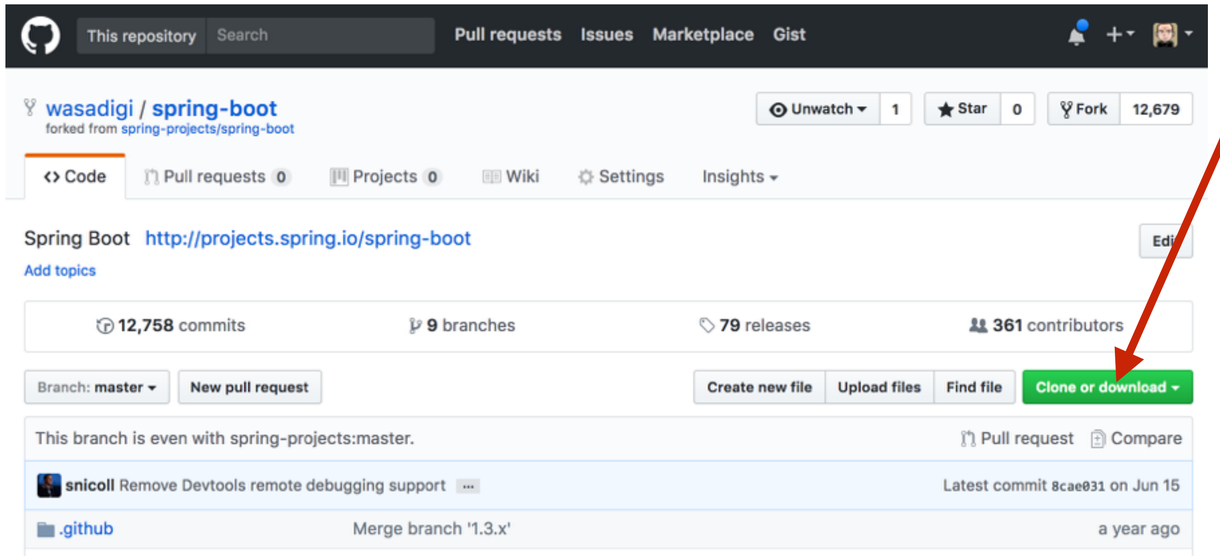


The GitHub workflow



The GitHub workflow

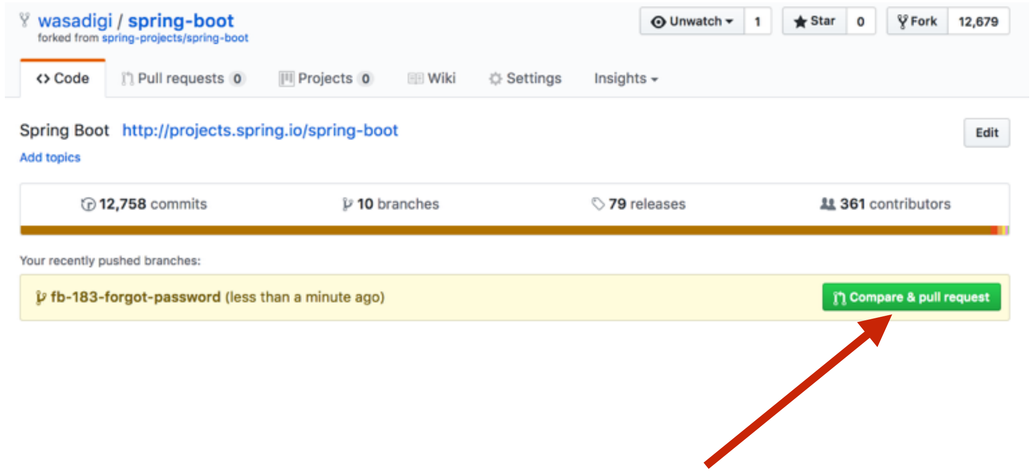
- **The developer works on his local repo**
  - He clones his fork on his machine
  - He then creates a “feature branch” to make his contribution
  - From time to time, he pushes his commits to his clone

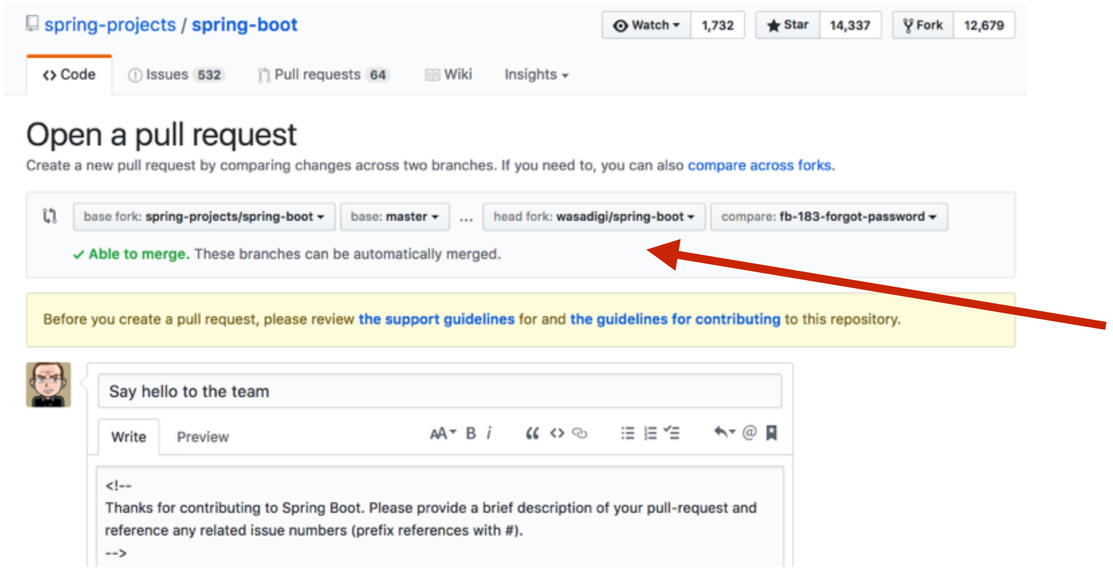



```
$ git clone git@github.com:wasadigi/spring-boot.git
$ git checkout -b "fb-183-forgot-password"
...

$ git add ...
$ git commit ...
...

$ git push origin fb-183-forgot-password
```

The GitHub workflow

- **When the developer is done...**
  - He asks the core team to merge his contribution in the original repo
  - To do that, he creates a “Pull Request” (I am requesting you to pull my changes)





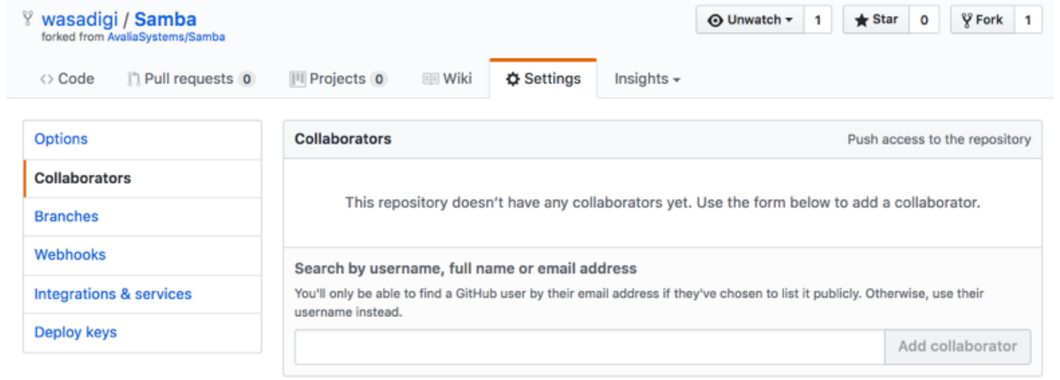
The GitHub workflow

- **When the developer is done...**
  - The core team **reviews the changes**, often with the help of **automated tools** that give feedback about the quality. The developer may be asked to improve the code.
  - At the end of the process, the **Pull Request is either accepted or refused**. If it is accepted, the developer branch is merged into a branch of the original repo (usually master).

Let’s play music with git and GitHub

Organization setup

- **Avalia Systems owns the upstream repo (AvaliaSystems/Samba)**
  - You can fork and clone the repo, but not push to it
  - In coming days, some of you will join as collaborators
- **You work in 9 teams (the Sienge teams)**
  - Every team will own its own fork
  - One person in every team
    - forks the upstream repo
    - invites other team members as collaborators
  - You will play different roles (analyst, developer) and take turns



github SOCIAL CODING

AvaliaSystems/Samba

fork (UI)

teamMemberID/Samba

Collaborators:
all team members

every team creates 1 fork

every team member does a **git clone**

local repo for *team member 1*

local repo for *team member 3*

local repo for *team member 2*

local repo for *team member n*

If you want to get commits made on the repo AFTER the fork:

git remote add upstream git@... (command line)
git fetch upstream

Workflow: new features

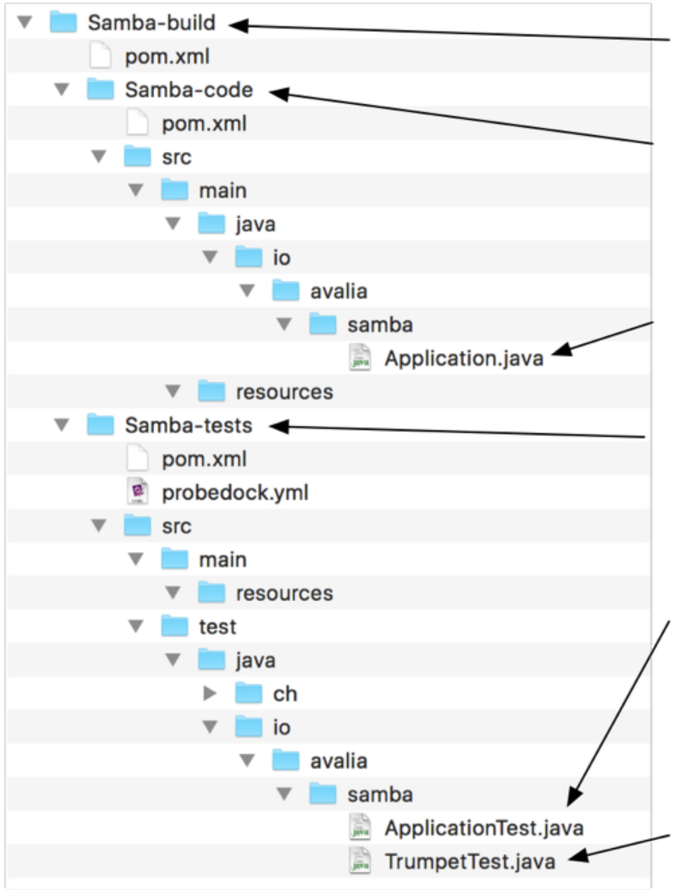
- The **analyst**
  - creates an issue in the **upstream repository** (e.g. #28 on AvaliaSystems/Samba)
  - creates a feature branch in his **local repo** (e.g. fb-28-xxx)
  - specifies the expected behaviour with automated tests
  - pushes the feature branch to his **team repo**
- The **developer**
  - fetches from his team repo, checks out the feature branch (local)
  - implements the code so that tests turn green
  - commits and pushes to his **team repo**
  - issues a pull request on the **upstream repo** (includes “closes #28” in the message)
- The **owner on upstream** on decides if the contribution can be merged into upstream/master

Workflow: bug fixes

- The **analyst**
  - creates an issue in the upstream repository (e.g. #28)
  - creates a feature branch (e.g. fb-28-xxx)
  - reproduces the bug with an automated test
  - pushes the feature branch to the team repo
- The **developer**
  - checks out the feature branch
  - implements the bug fix so that test turn green
  - commits and pushes to the team repo
  - issues a pull request (includes “closes #28” in the message)
- The **owner on upstream** decides if the contribution can be merged into upstream/master

Project

- We develop a **Java** application
- We use **JUnit** to write and execute the specifications
- We use **maven** to manage dependencies and build the project
- **Executable specifications** are in a different module (BDD + automation)



This is a "parent" project, which contains the two others. It is what we use to build and test our application. You can open this project in Netbeans.

This is a project that you can open in Netbeans. It is the project that contains the sources for your application.

We give you a partial implementation. At the beginning, there is only one source file, with one bug. ***You will add code here.***

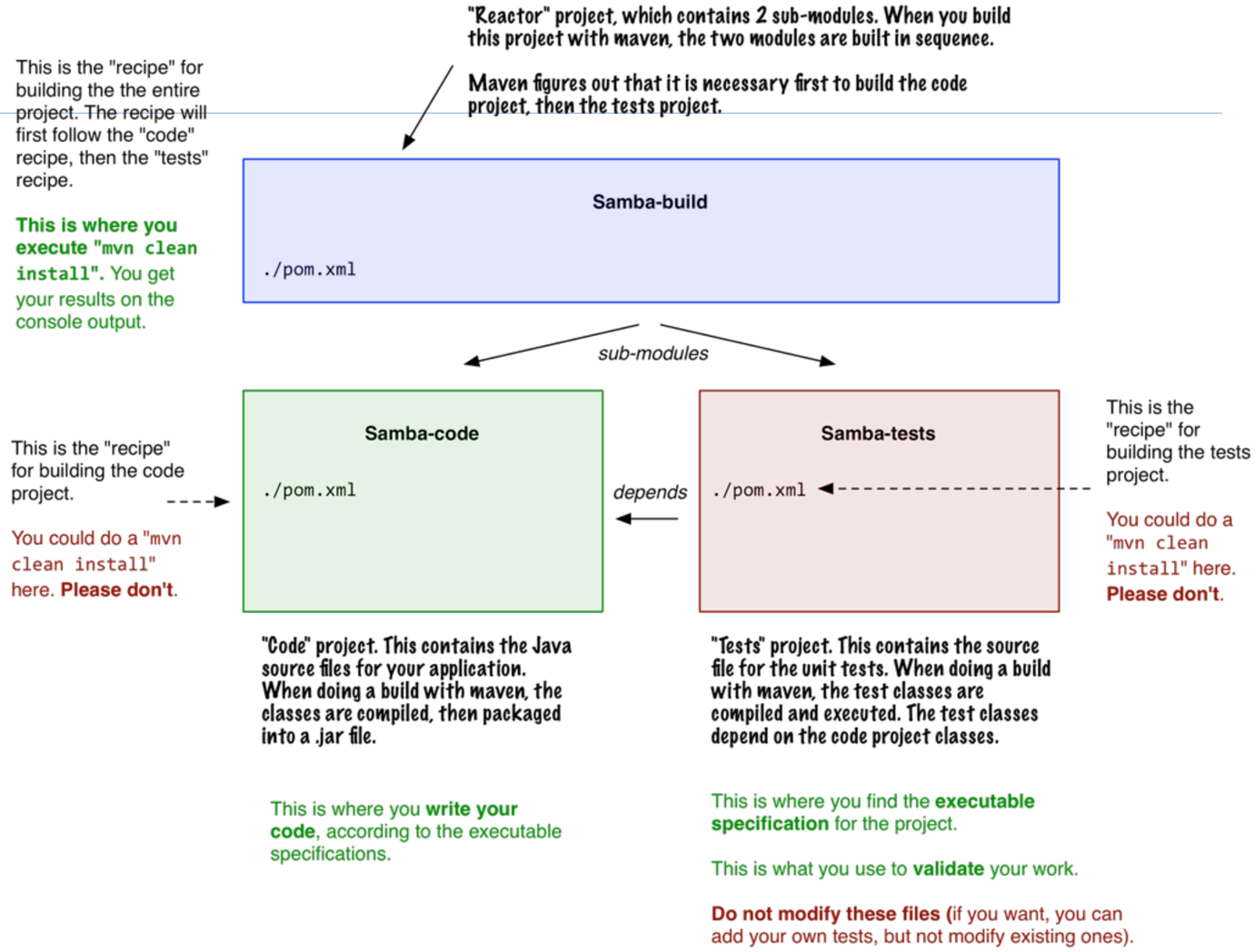
This is also a project that you can open in Netbeans. It contains the unit tests that specify what your application should do. The tests are used to validate your code.

This class contains unit tests to specify and validate the behavior of Application.java. At the beginning, one unit test fails (because we have one bug in Application.java).

This class contain unit tests for code that you have to write. The tests are commented so that you can compile the project after the clone.

Project

- You can build and validate the project from the command line (**mvn clean install**).
- Be careful to run this command from the **correct directory**.
- You can also open the project in **Netbeans**.



Executable specifications

- We write **very, very simple code**: the goal is to experiment with the workflow.
- We start with example features. Up to you to come up with **other ideas and have fun**.
- Pay attention to the name of the test method: it describes the intended **behaviour** (“should do”).

```
@Test
public void aTrumpetShouldMakePouet() {
  IInstrument trumpet = new Trumpet();
  String sound = trumpet.play();
  Assert.assertEquals("pouet", sound);
}
```

```
a Trumpet Should Make Pouet
```

Run #0

- **No team repo in this run**, everyone goes through the process
- Everyone....
  - forks the repo
  - clones his fork
  - fixes the bug and implement the features already in the code
  - submits a pull request (PR)
- I will then review (some of the) PRs, accept one and merge it on master.
- Everyone will then fetch the update from “upstream” and be in sync.

Run #1: new feature

- **In every team**:
  - Player 1: create the issue, specify the behaviour
  - Player 2: implement the feature, submit the PR
  - Teacher: review and accept the PR

Q&A

What is .gitignore?	There are **files that you don't want to store in the repo**. For instance, your IDE settings, the .class files, etc. You use .gitignore to tell git about them.
Can I break things?	Read output of git commands carefully - they often provide guidance. Be careful with “**git reset**” and “**git rebase**” (safe with commits you have not shared) Do not use **--force flag**, unless you know exactly what you are doing.
How do I name branches?	You have to **define your own conventions** and **stick to them**. For instance, you can use “fb-nnn-short-desc” for new features and “fix-nnn-short-desc” for bug fixes. nnn is the number of the issue (in your issue tracker)
What is the difference between origin and upstream?	**Origin** is the repo that you cloned. **Upstream** is the repo from which origin was forked. If people make modifications on upstream, you need to fetch them (command line). GitHub will not keep upstream and origin in sync.