

CS 241: Systems Programming

Lecture 5. Version Control/Git

Fall 2023

Prof. Stephen Checkoway

Version control system (VCS)

Version control system (VCS)

A way to track changes to your files

- What you changed
- Why you changed it

Version control system (VCS)

A way to track changes to your files

- What you changed
- Why you changed it

A way to keep “backups” of older versions

Version control system (VCS)

A way to track changes to your files

- What you changed
- Why you changed it

A way to keep “backups” of older versions

A way to keep track of different versions (branches) of a project

- Development
- Release

Version control system (VCS)

A way to track changes to your files

- What you changed
- Why you changed it

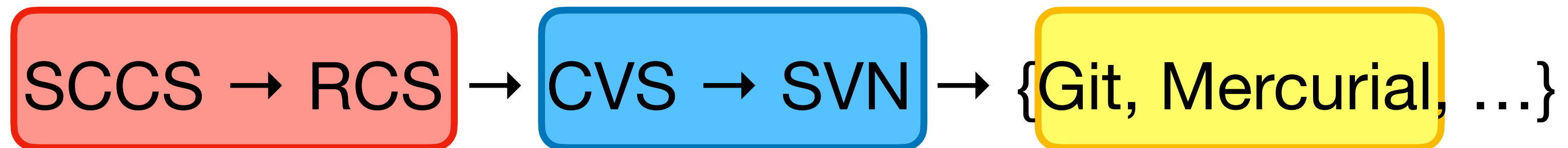
A way to keep “backups” of older versions

A way to keep track of different versions (branches) of a project

- Development
- Release

A way to organize and collaborate on a project

VCS history (abridged)



1972 — Source Code Control System (SCCS)

1985 — Revision Control System (RCS)

- All users on the same system, each with their own checkout of the files

1986 — Concurrent Versioning System (CVS)

- Client/server model

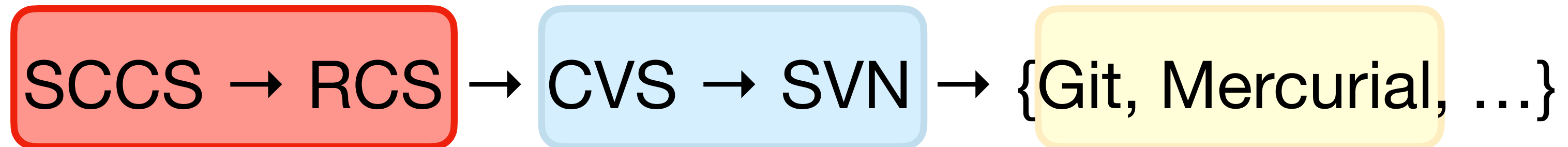
2000 — Subversion (SVN)

- Essentially a better CVS

2005 — Git and Mercurial

- Distributed model: each user has their own copy of the whole repository

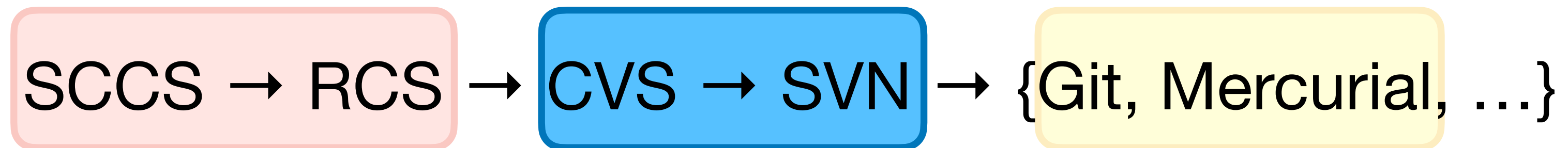
VCS history (abridged)



SCCS/RCS

- ▶ Master repository with all history stored somewhere, e.g.,
/source/program
- ▶ Individual users checkout the current version somewhere else, e.g.,
~/program
- ▶ Modifications can be checked in to the master repo
- ▶ Other users' modifications can be checked out again
- ▶ The history of files and their differences can be shown

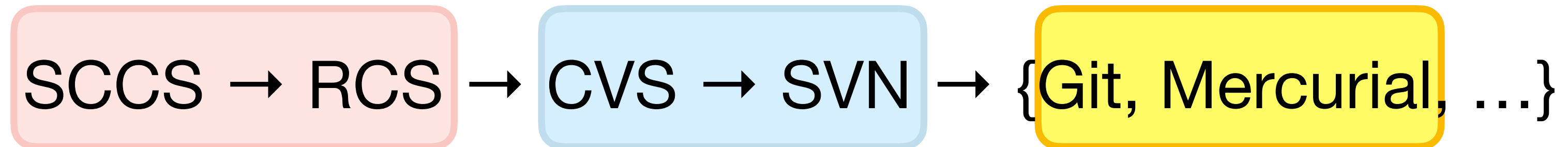
VCS history (abridged)



CVS/SVN

- ▶ Master repo stored on some server, e.g.,
`vcs.oberlin.edu:/vcs/program`
- ▶ Users on many different machines can checkout copies, e.g.,
`clyde.cs.oberlin.edu:~/program`
- ▶ Changes to files are committed to the server which maintains the authoritative copy of the repository history
- ▶ Local copies can be updated with other users' changes from the server
- ▶ Multiple branches, but each with a linear commit history (r1, r2, r3, ...)

VCS history (abridged)



Git/Mercurial

- Decentralized
 - Each user has a full copy of the repo
 - No authoritative version
- Users can push changes to other users or pull changes from others
- Multiple, lightweight branches
- History is not linear, it's a DAG (we'll see what this means shortly)
- Decentralization is hard to deal with: use Github (or similar)

Git

Git

A distributed version control system

- Everyone can act as a “server”
- Everyone mirrors the entire repository

Git

A distributed version control system

- Everyone can act as a “server”
- Everyone mirrors the entire repository

Many local operations

- Quick to add files, commit, create new branches, etc.
- Can have local changes w/o pushing to others

Git

A distributed version control system

- Everyone can act as a “server”
- Everyone mirrors the entire repository

Many local operations

- Quick to add files, commit, create new branches, etc.
- Can have local changes w/o pushing to others

Collaborate with other developers

- “Push” and “pull” code from hosted repositories such as Github

Initial setup

```
$ git config --global user.name 'Stephen Checkoway'
$ git config --global user.email \
    'stephen.checkoway@oberlin.edu'
$ git config --global core.editor vim
```

Global config values are stored in `~/.gitconfig`

Can also have local config settings in `${repo}/.git/config`

Creating a repository

```
$ mkdir project  
$ cd project  
$ git init
```

Creates a `.git` folder in `project`

No files are currently being tracked or managed

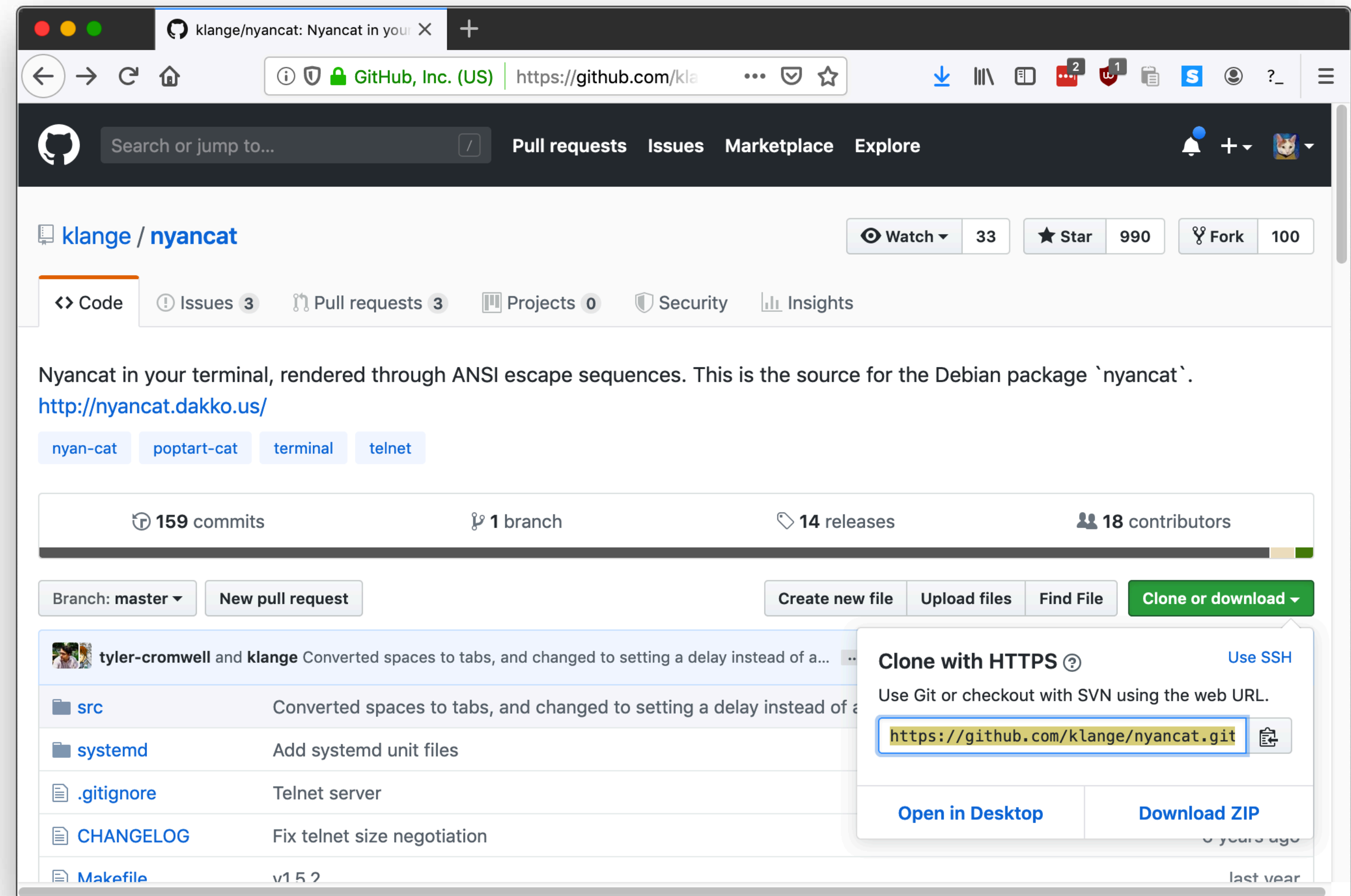
No remote server

Cloning a (remote) repository

```
$ git clone https://github.com/klange/nyancat.git
```

Creates a local copy of the repo including the whole history

Associated with a remote server



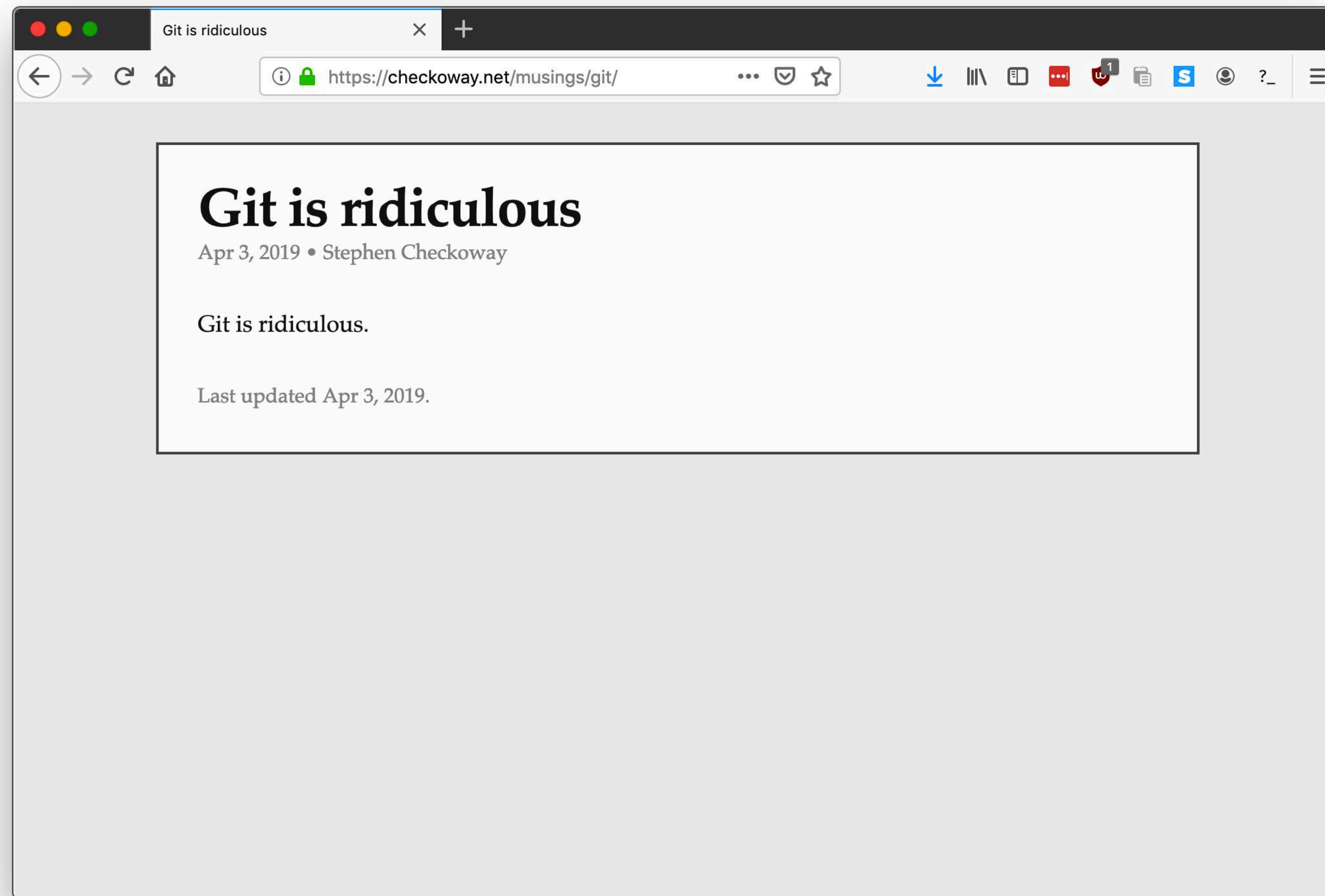
Cloning a (remote) repository

```
steve@clyde:~$ █
```

Cloning a (remote) repository

```
steve@clyde:~$ █
```

Warning: Git is ridiculous



Working dir vs staging vs .git

After `git init` or `git clone`, you have a working directory on the file system

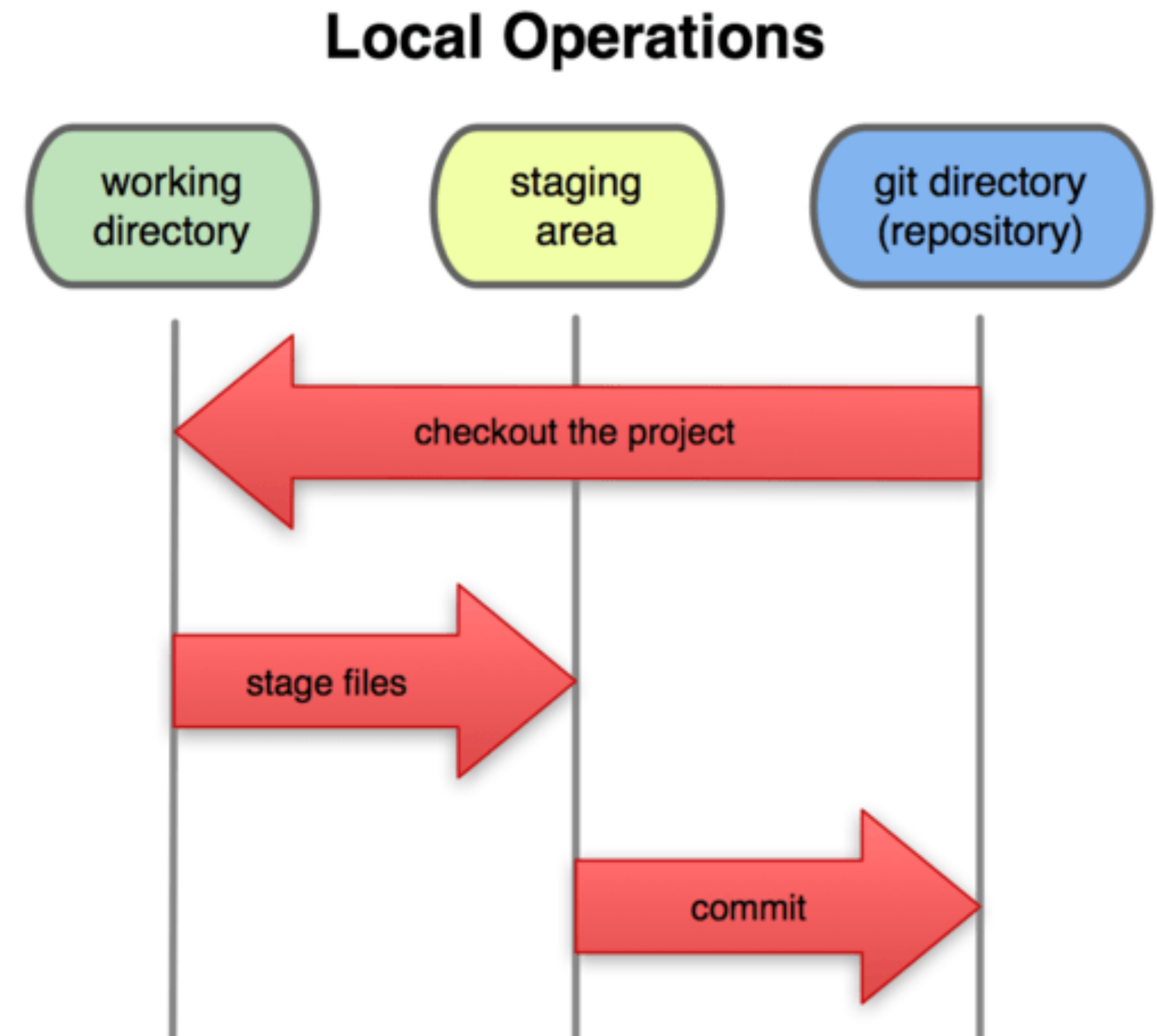
- Holds one version of the files in the repo

Inside it (usually) is a `.git` directory with

- The whole history of the repo (all commits)
- config options, branches, etc.

Conceputional staging area

- Holds files to be committed



Adding and committing

Working directory



The diagram consists of three vertically oriented rounded rectangles arranged horizontally. The first rectangle on the left is green with a dark green border and contains the text 'Working directory'. The middle rectangle is yellow with a dark yellow border and contains the text 'Staging area'. The third rectangle on the right is light blue with a dark blue border and contains the text 'Git directory'.

Staging area

Git directory

Adding and committing

```
$ vim README
```

```
# Create a readme describing the project
```

Working directory



Staging area



Git directory



Adding and committing

```
$ vim README          # Create a readme describing the project
```

Working directory

README

Staging area

Git directory

Adding and committing

```
$ vim README          # Create a readme describing the project  
$ git add README      # Add README to the staging area
```

Working directory

README

Staging area

Git directory

Adding and committing

```
$ vim README      # Create a readme describing the project  
$ git add README  # Add README to the staging area
```

Working directory

README

Staging area

README

Git directory

Adding and committing

```
$ vim README           # Create a readme describing the project
$ git add README       # Add README to the staging area
$ vim hello.py         # Create some code
```

Working directory

README

Staging area

README

Git directory

Adding and committing

```
$ vim README           # Create a readme describing the project
$ git add README       # Add README to the staging area
$ vim hello.py         # Create some code
```

Working directory

README

hello.py

Staging area

README

Git directory

Adding and committing

```
$ vim README          # Create a readme describing the project
$ git add README      # Add README to the staging area
$ vim hello.py        # Create some code
$ git add hello.py    # Add the hello.py to the staging area
```

Working directory

README

hello.py

Staging area

README

Git directory

Adding and committing

```
$ vim README          # Create a readme describing the project
$ git add README      # Add README to the staging area
$ vim hello.py        # Create some code
$ git add hello.py    # Add the hello.py to the staging area
```

Working directory

README
hello.py

Staging area

README
hello.py

Git directory

Adding and committing

```
$ vim README          # Create a readme describing the project
$ git add README      # Add README to the staging area
$ vim hello.py        # Create some code
$ git add hello.py    # Add the hello.py to the staging area
$ git commit          # Commit the files to the repo
```

Working directory

README
hello.py

Staging area

README
hello.py

Git directory

Adding and committing

```
$ vim README           # Create a readme describing the project
$ git add README       # Add README to the staging area
$ vim hello.py         # Create some code
$ git add hello.py     # Add the hello.py to the staging area
$ git commit          # Commit the files to the repo
```

Working directory

README

hello.py

Staging area

Git directory

82F1A6

Commits

Each commit is (in essence) a snapshot of the repository

Commits are named by a hash of their contents, e.g.,
 c37ce054c766b79a3577aba898b296d3557c3d24,
often just the first 7 digits: c37ce05

Each commit links to its parent commit(s)

Adding and committing

Working directory

README

hello.py

Staging area

Git directory

82F1A6

Adding and committing

```
$ vim hello.py          # Modify the code
```

Working directory

README

hello.py

Staging area

Git directory

82F1A6

Adding and committing

```
$ vim hello.py           # Modify the code
$ vim ChangeLog          # Write a change log with changes
```

Working directory

README

hello.py

Staging area

Git directory

82F1A6

Adding and committing

```
$ vim hello.py          # Modify the code
$ vim ChangeLog         # Write a change log with changes
```

Working directory

README

hello.py

ChangeLog

Staging area

Git directory

82F1A6

Adding and committing

```
$ vim hello.py           # Modify the code
$ vim ChangeLog          # Write a change log with changes
$ git add hello.py       # Add the hello.py to the staging area
```

Working directory

README

hello.py

ChangeLog

Staging area

Git directory

82F1A6

Adding and committing

```
$ vim hello.py           # Modify the code
$ vim ChangeLog          # Write a change log with changes
$ git add hello.py       # Add the hello.py to the staging area
```

Working directory

README

hello.py

ChangeLog

Staging area

hello.py

Git directory

82F1A6

Adding and committing

```
$ vim hello.py          # Modify the code
$ vim ChangeLog         # Write a change log with changes
$ git add hello.py      # Add the hello.py to the staging area
$ git add ChangeLog     # Add ChangeLog
```

Working directory

README
hello.py
ChangeLog

Staging area

hello.py

Git directory

82F1A6

Adding and committing

```
$ vim hello.py           # Modify the code
$ vim ChangeLog          # Write a change log with changes
$ git add hello.py       # Add the hello.py to the staging area
$ git add ChangeLog      # Add ChangeLog
```

Working directory

README
hello.py
ChangeLog

Staging area

hello.py
ChangeLog

Git directory

82F1A6

Adding and committing

```
$ vim hello.py          # Modify the code
$ vim ChangeLog         # Write a change log with changes
$ git add hello.py      # Add the hello.py to the staging area
$ git add ChangeLog     # Add ChangeLog
$ git commit            # Commit the files to the repo
```

Working directory

README
hello.py
ChangeLog

Staging area

hello.py
ChangeLog

Git directory

82F1A6

Adding and committing

```
$ vim hello.py           # Modify the code
$ vim ChangeLog          # Write a change log with changes
$ git add hello.py       # Add the hello.py to the staging area
$ git add ChangeLog      # Add ChangeLog
$ git commit             # Commit the files to the repo
```

Working directory

README
hello.py
ChangeLog

Staging area

Git directory

82F1A6

F00D11



You've just cloned a repository from github, cd'd into the repo's directory, and created a new file.

```
$ git clone git@github.com:username/example-project.git
$ cd example-project
$ vim foo
```

What command(s) should you run to commit this new file to the repo?

A. `$ git add foo`

B. `$ git commit foo`

C. `$ git add foo`
`$ git commit`

D. `$ git add foo`
`$ git push`

E. `$ git add --commit foo`

After adding and committing initially, you've been working on `foo` for a while and want to commit again.

What command(s) should you run to commit your changes repo?

A. `$ git add foo`

D. `$ git commit foo`
`$ git push`

B. `$ git commit foo`

E. `$ git add --commit foo`

C. `$ git add foo`
`$ git commit`

Commit Message

When doing a commit, your editor will be opened so you can enter a commit message

- Short summary line
- Blank line
- Longer description

Try to provide enough detail that you can read the message to understand what changes were made (and why)

- Might be easy to remember now, but in 6 months?

Naming commits

Individual commits can have human-readable names

- **HEAD** is the currently checked out commit
- **main** is most recent commit on the default **branch** (which is itself named `main`)
- tags and branches give names to commits

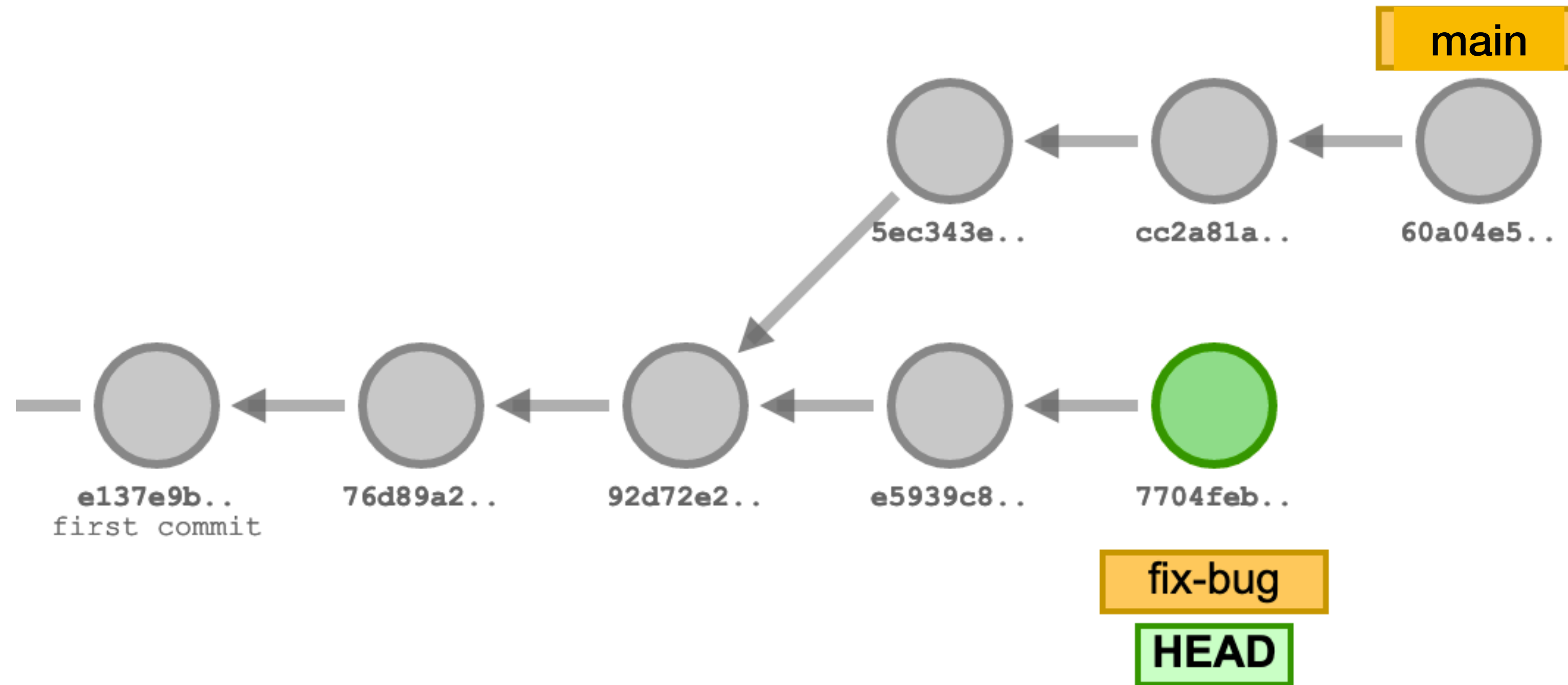
Example



After two commits, `HEAD` and `main` point to the second commit

After a third commit, `HEAD` and `main` point to the third commit

HEAD != main



We can create a new branch `fix-bug` and commit to that branch

We can also keep committing to `main`

`HEAD` points to the branch we have checked out

Pushing to the remote server

```
$ git push
```

Sends to the remote server all of your committed data (it doesn't already have)

Remote servers are called **remotes**

- ▶ When cloning, the remote is named `origin` by default
- ▶ Remotes have their own branches `origin/main` is `origin's` main branch
- ▶ It's possible to have multiple remotes (but we probably won't in this class)

Example

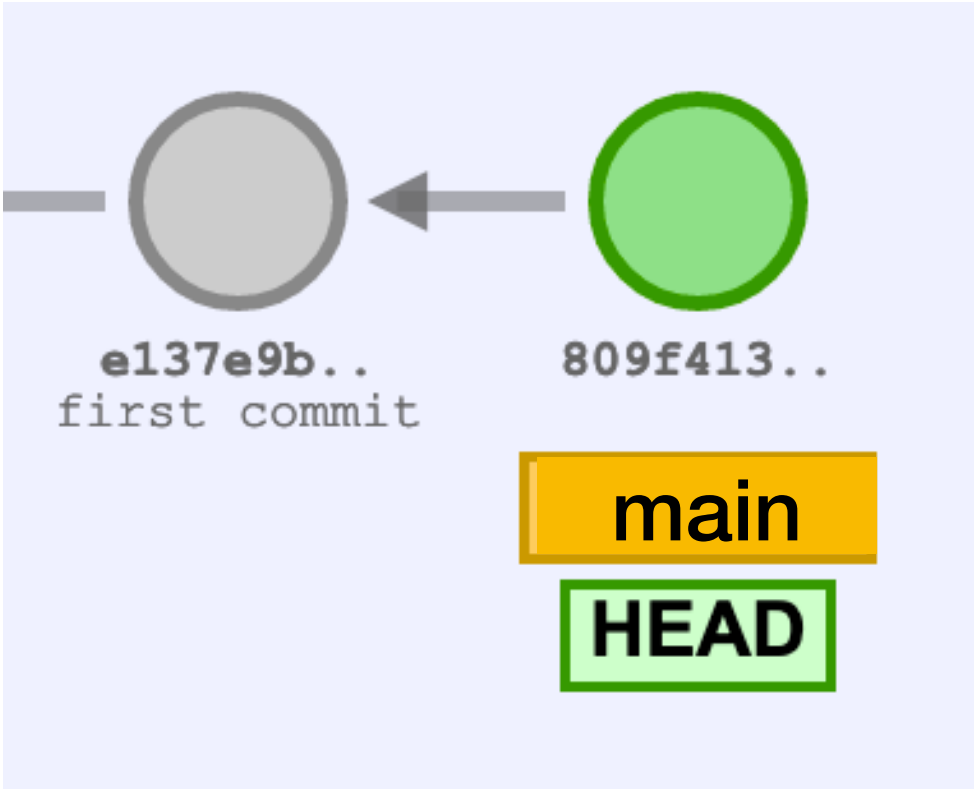
Local repository

main

main

main

Origin



main

main

Example

```
$ git clone ...
```

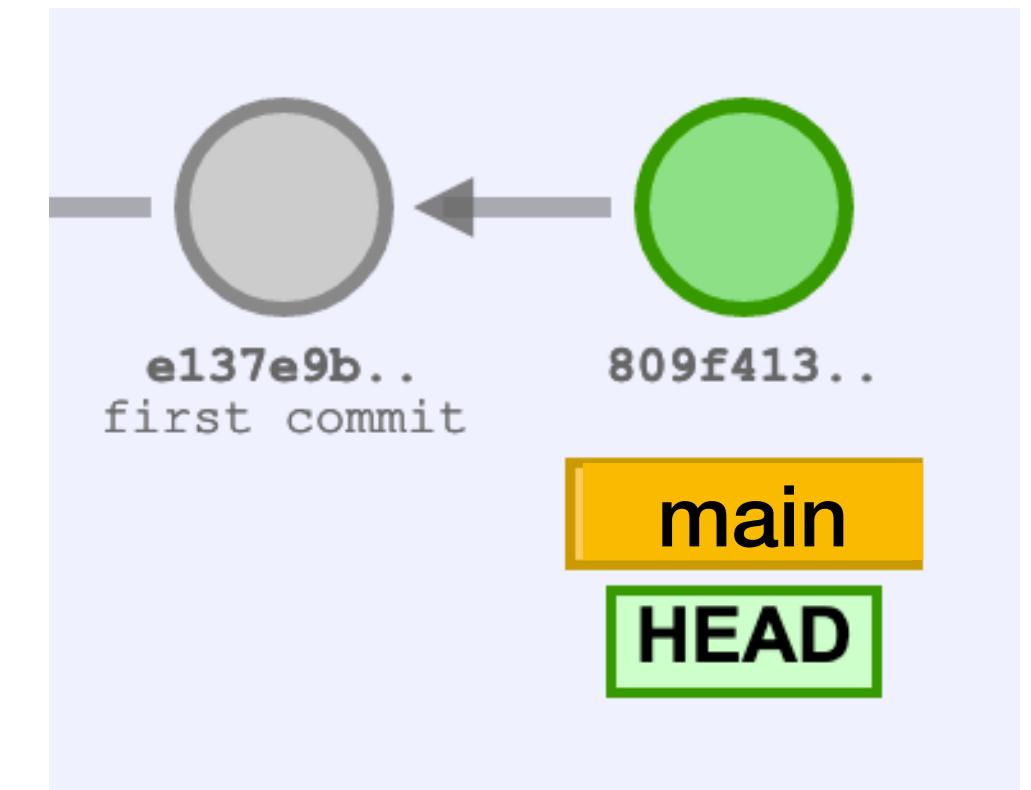
Local repository

main

main

main

Origin



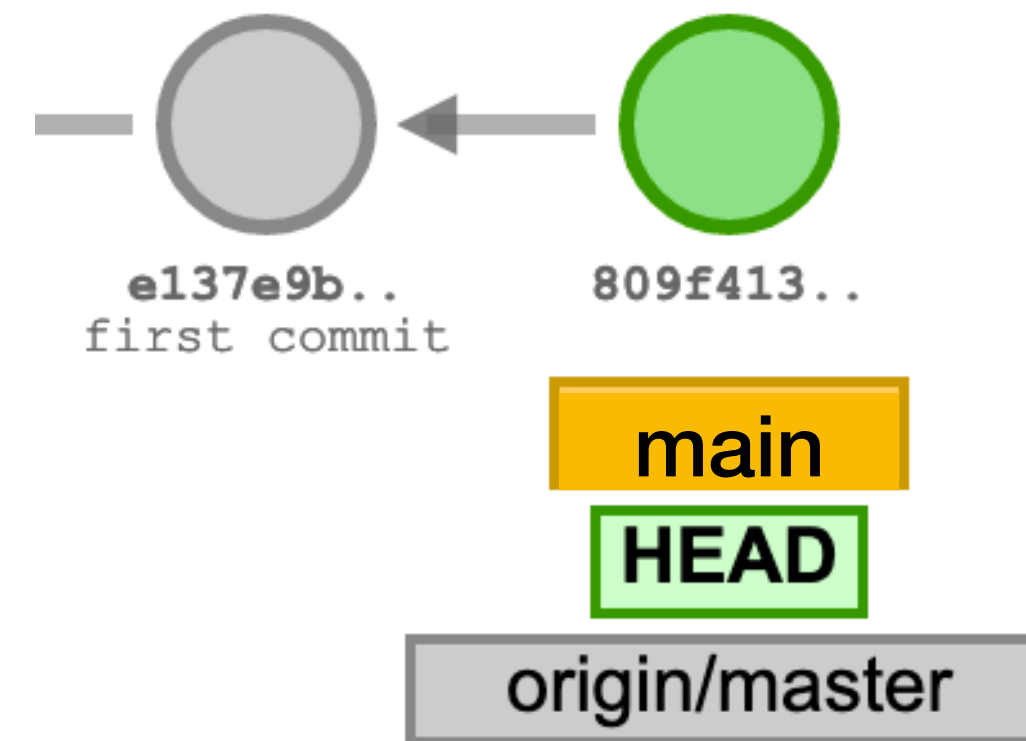
main

main

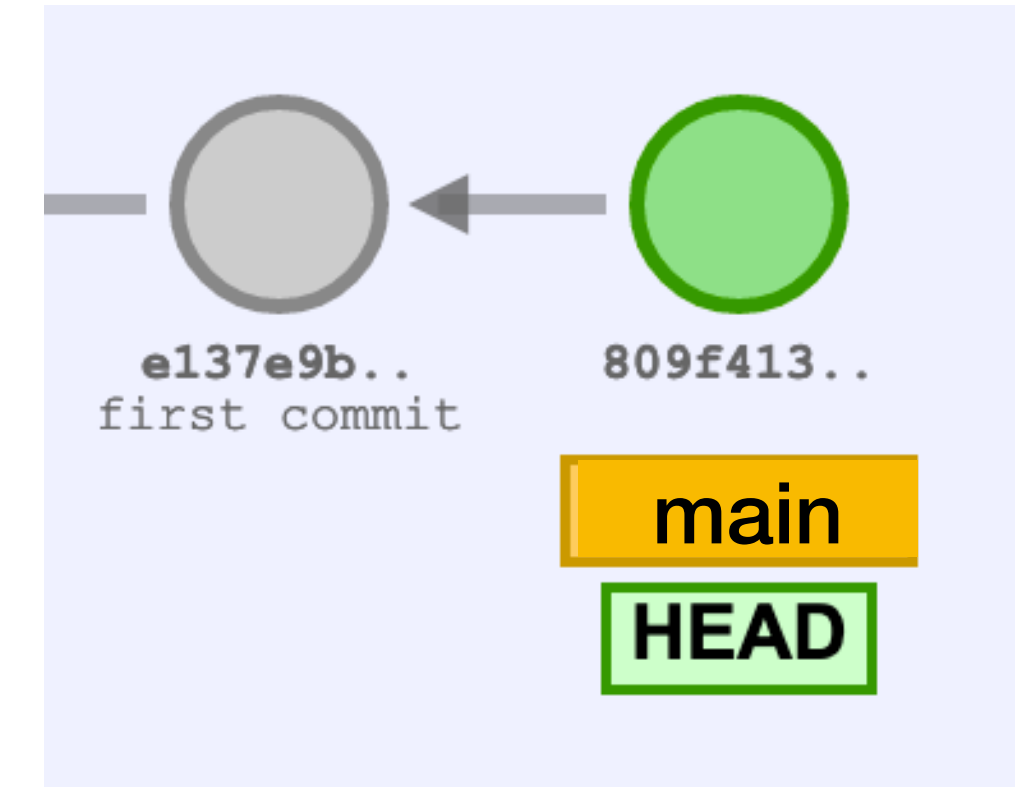
Example

```
$ git clone ...
```

Local repository



Origin



main

main

main

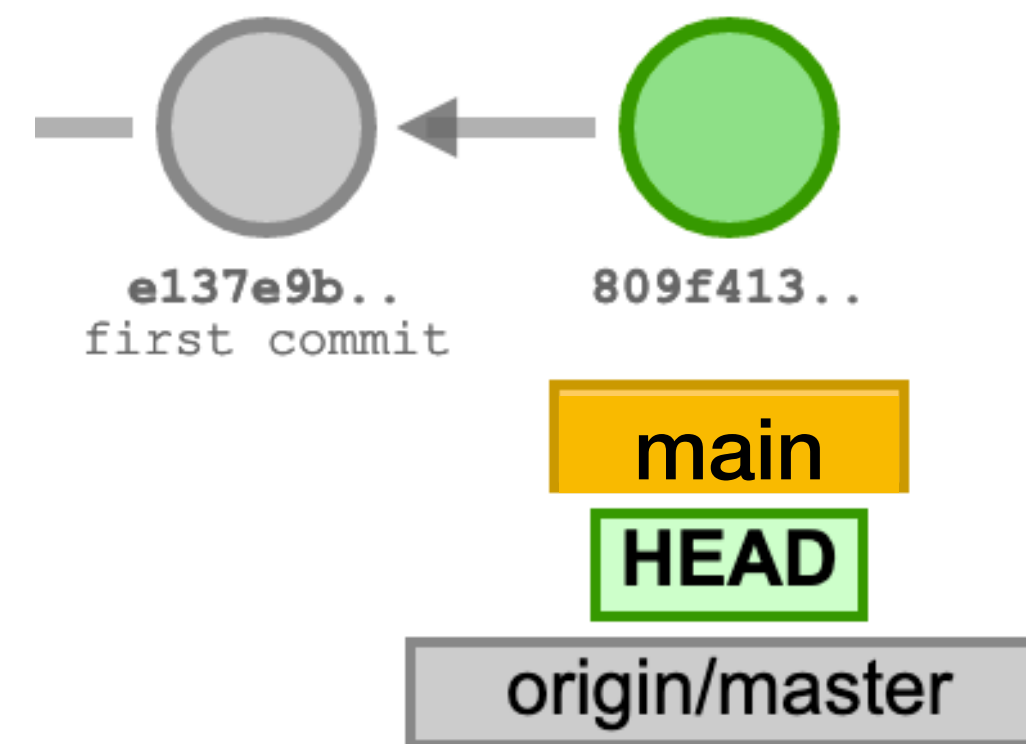
main

Example

```
$ git clone ...
```

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

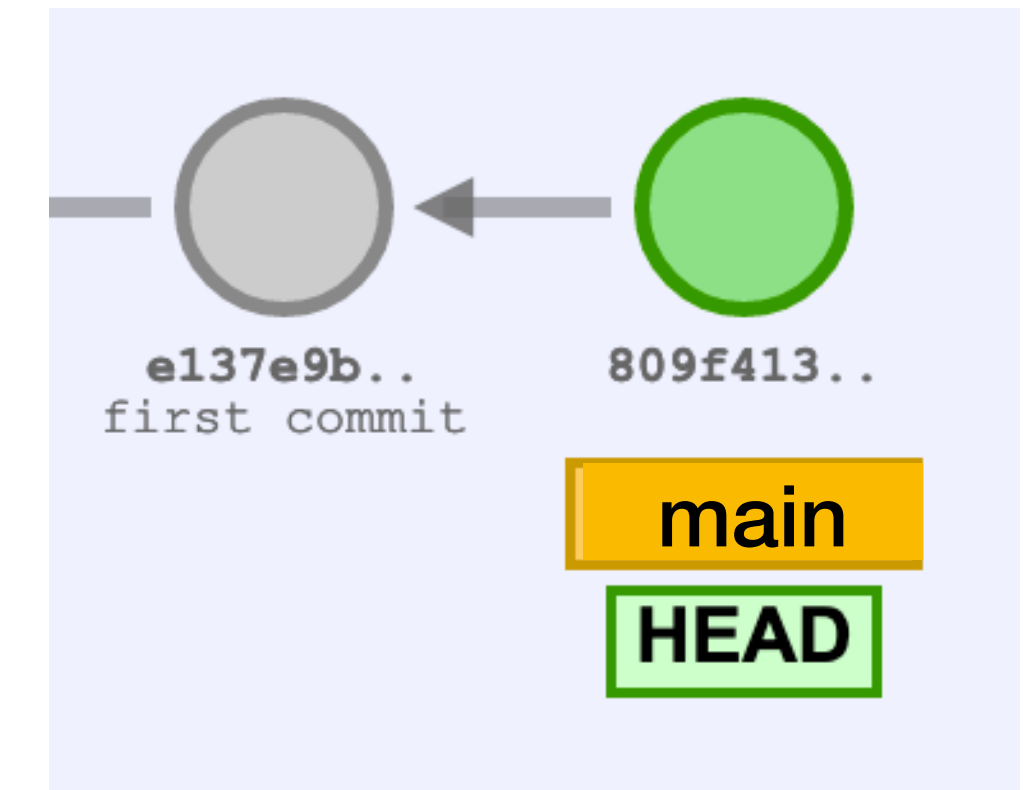
Local repository



main

main

Origin



main

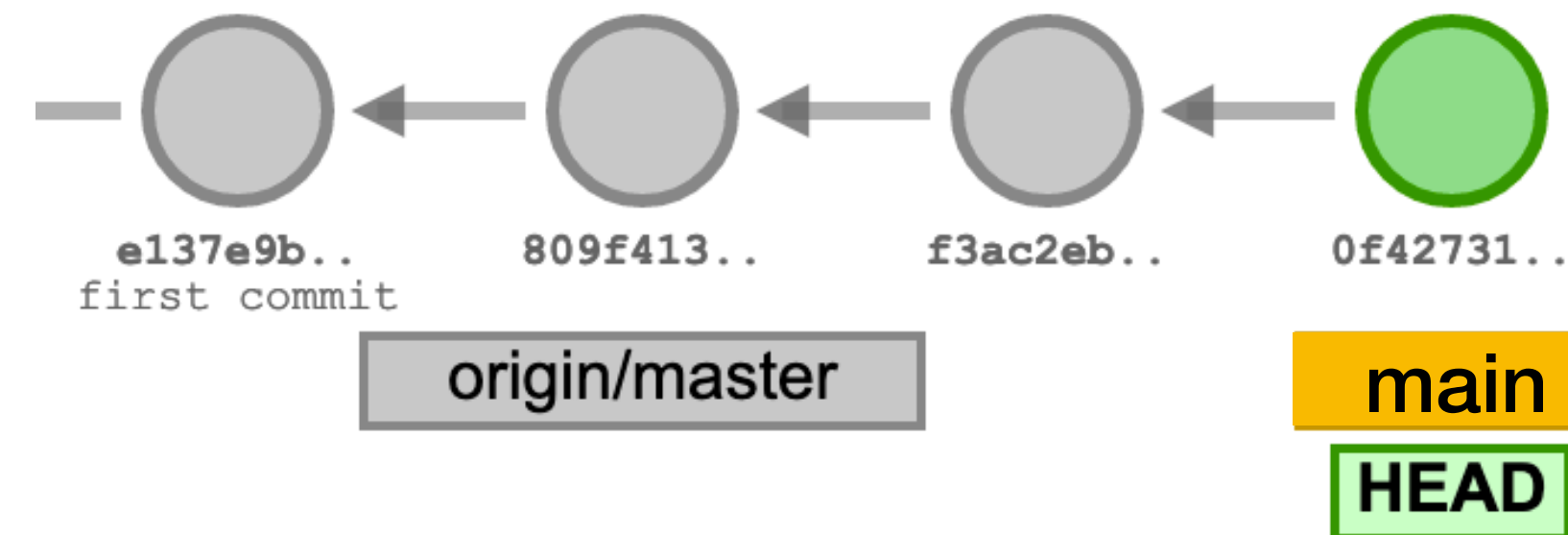
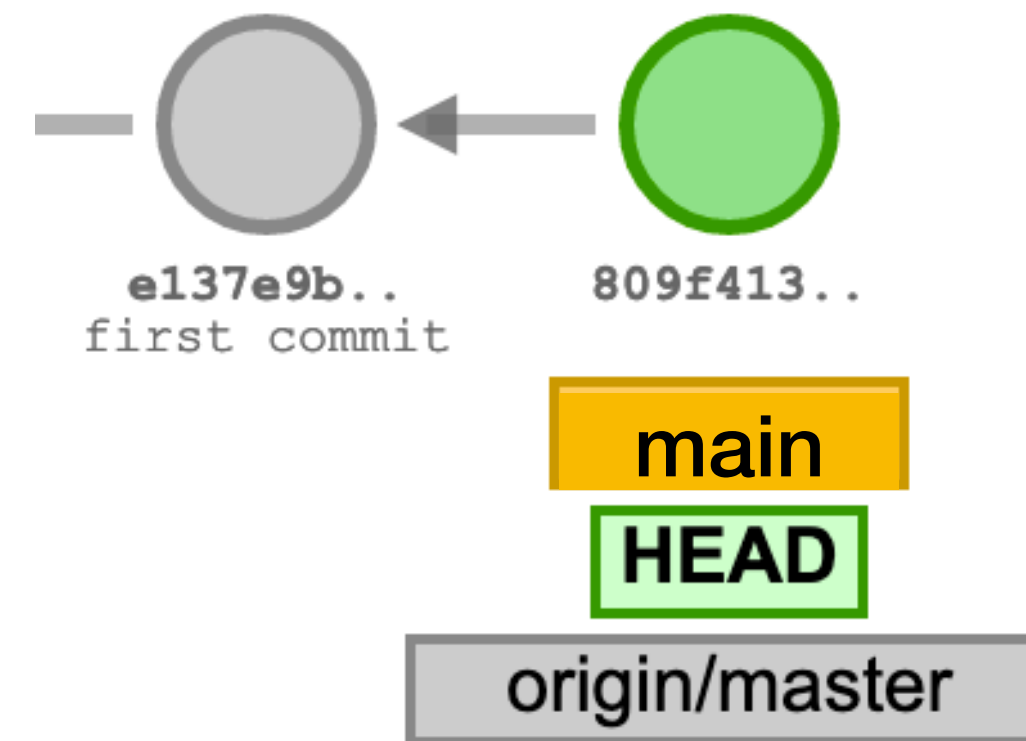
main

Example

```
$ git clone ...
```

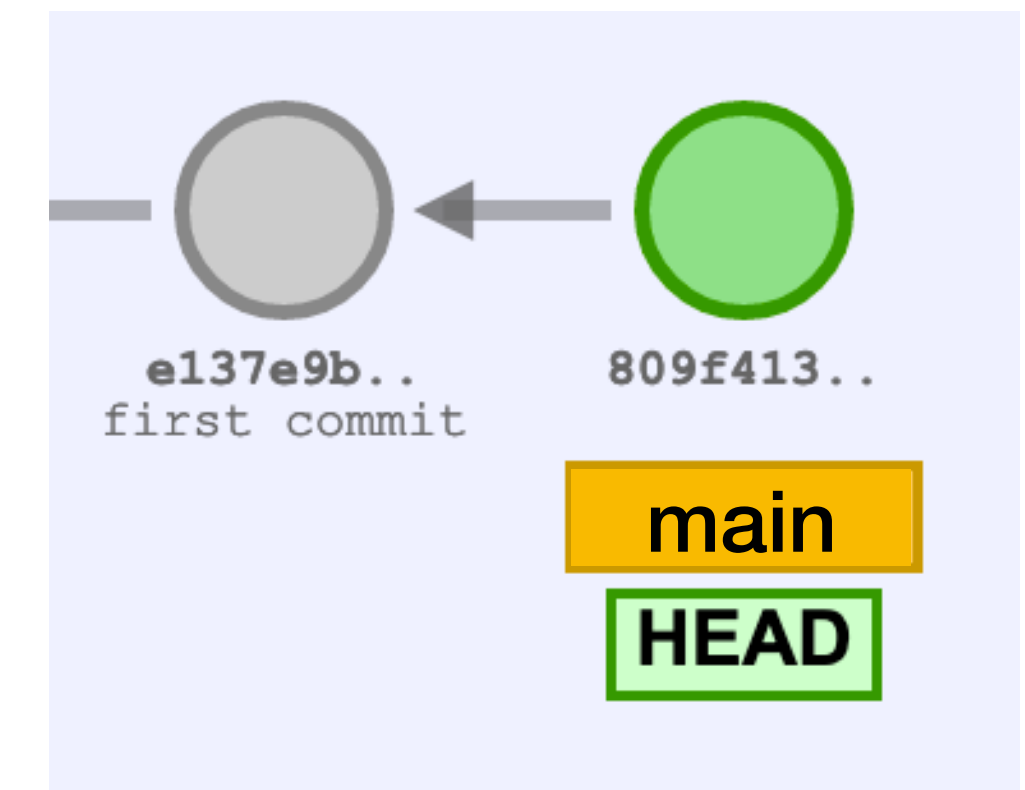
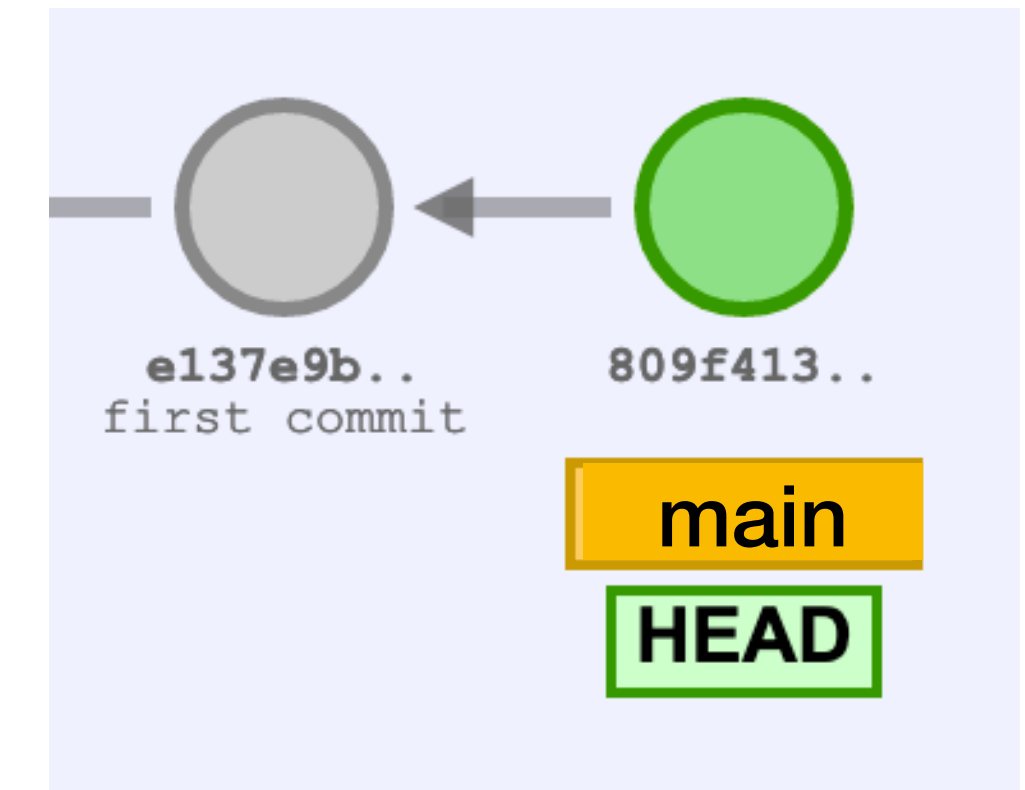
```
$ git add ...  
$ git commit  
$ git add ...  
$ git commit
```

Local repository



main

Origin



main

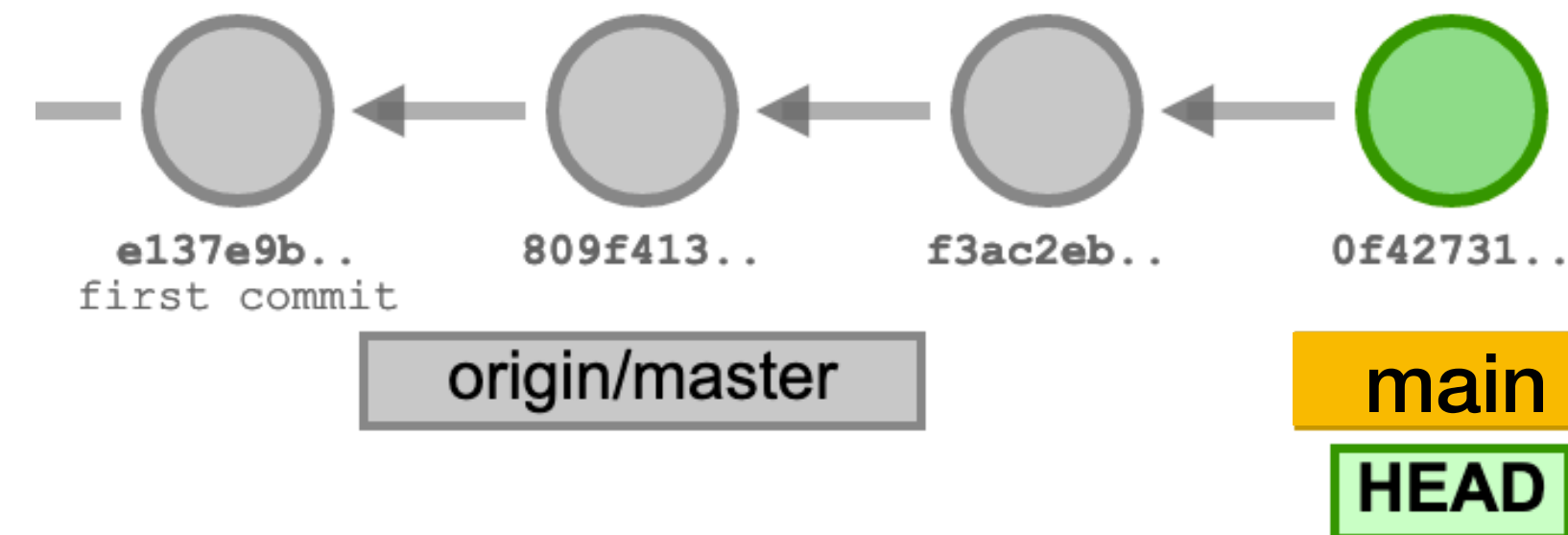
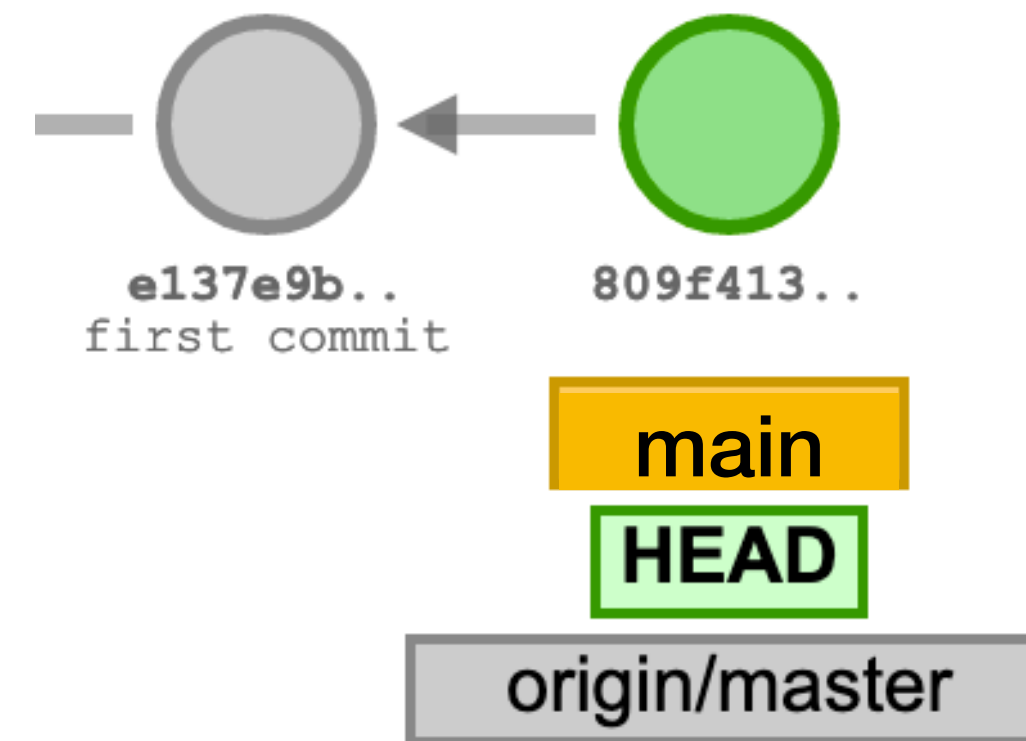
Example

```
$ git clone ...
```

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

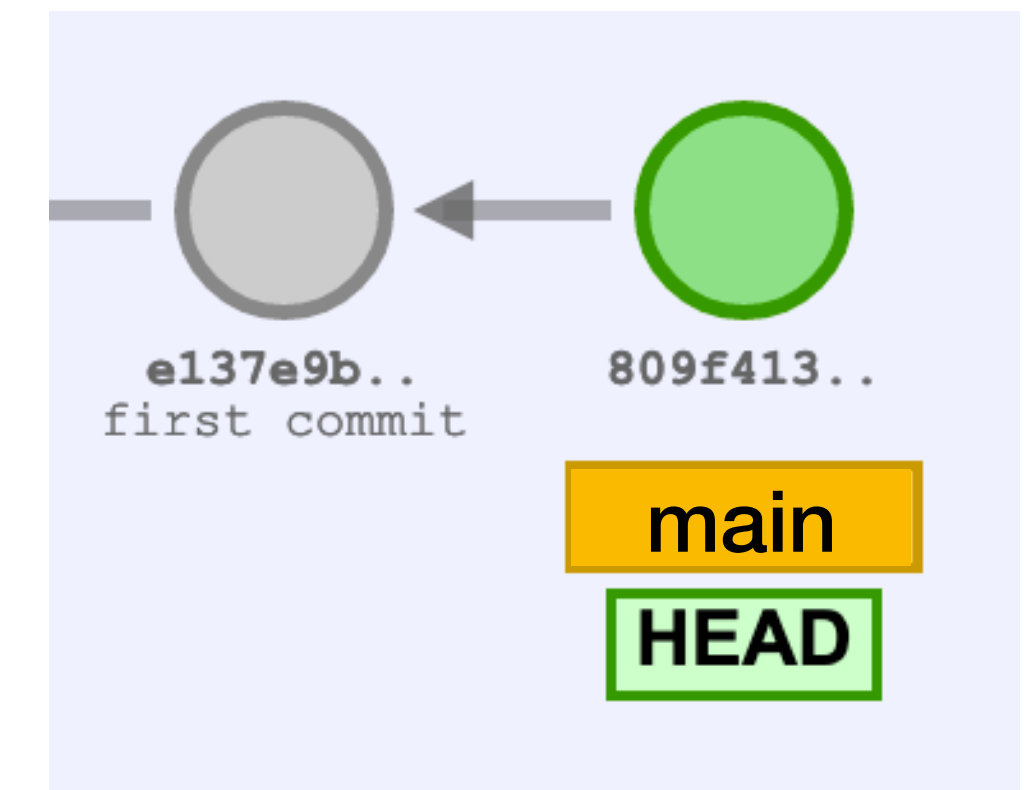
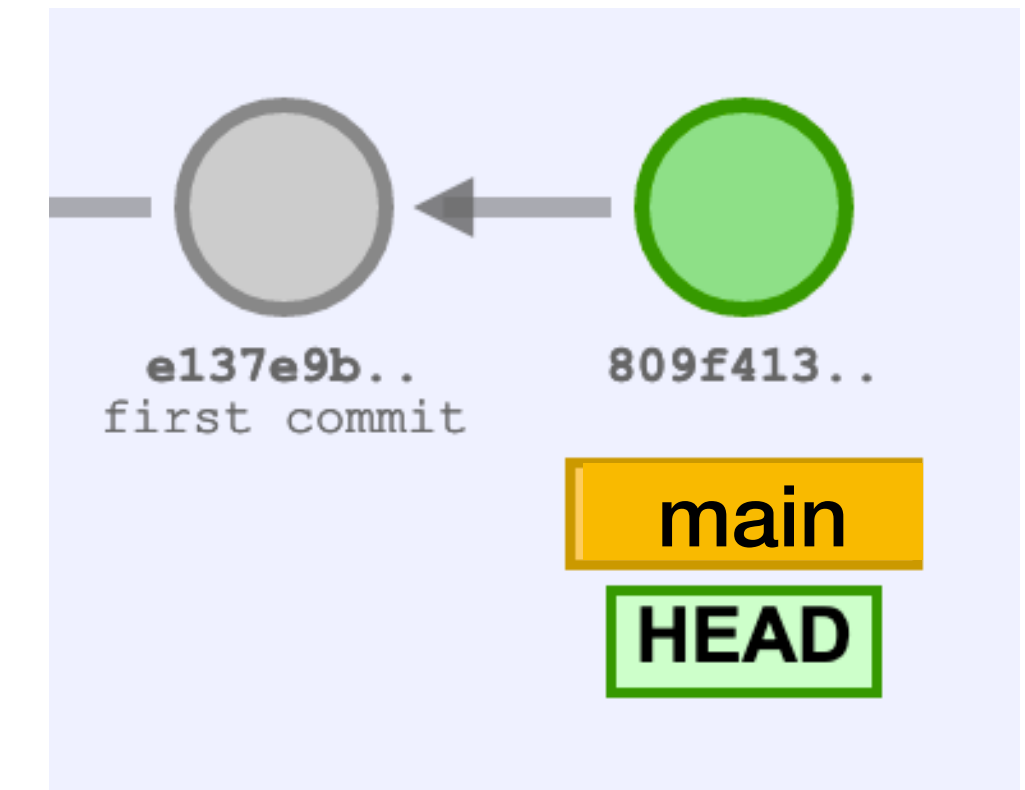
```
$ git push
```

Local repository



main

Origin



main

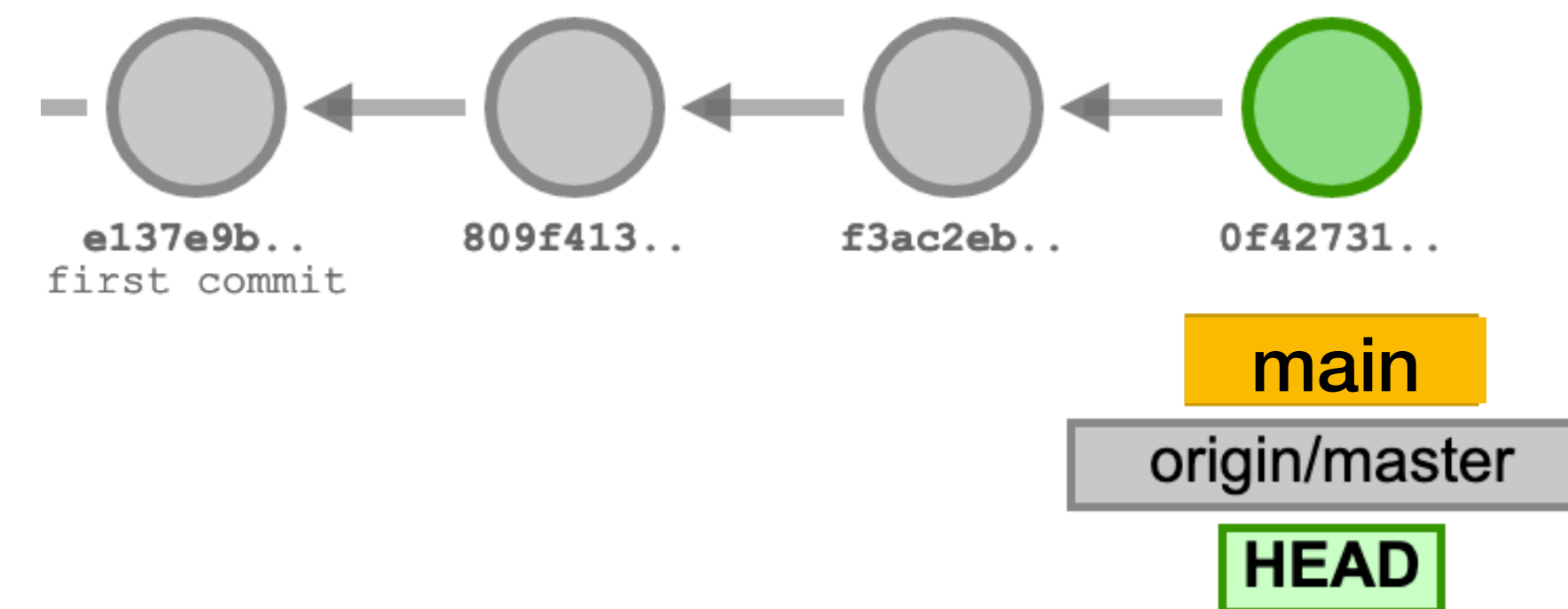
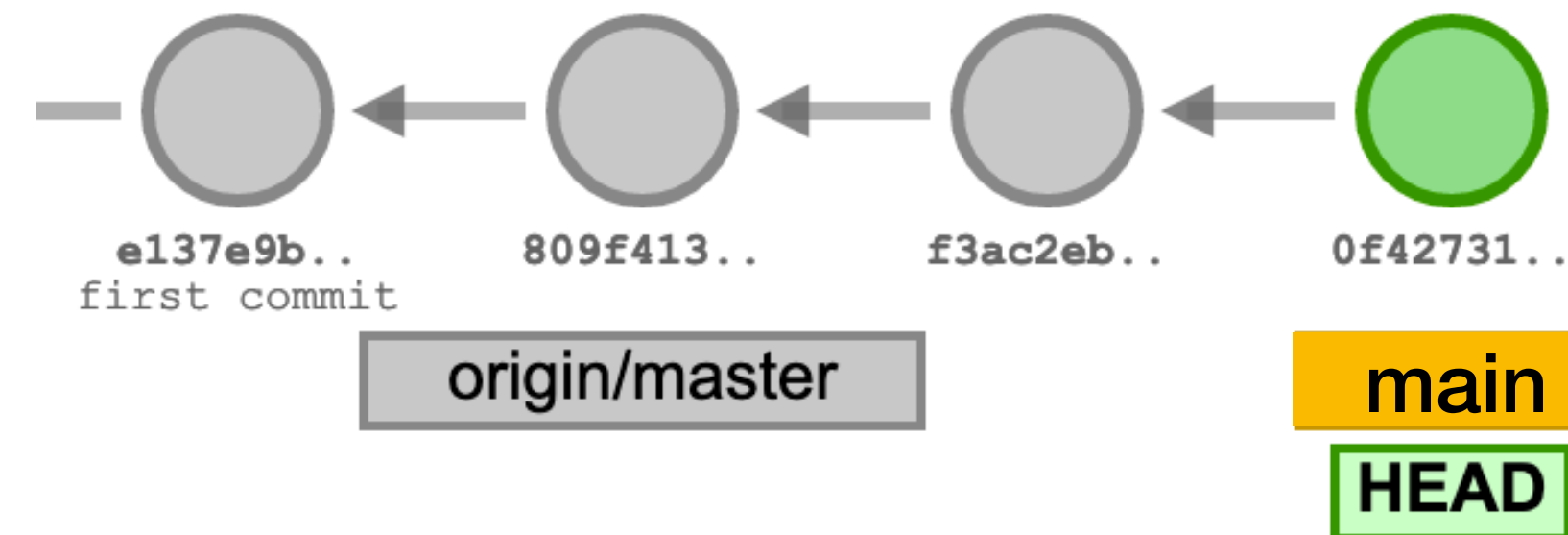
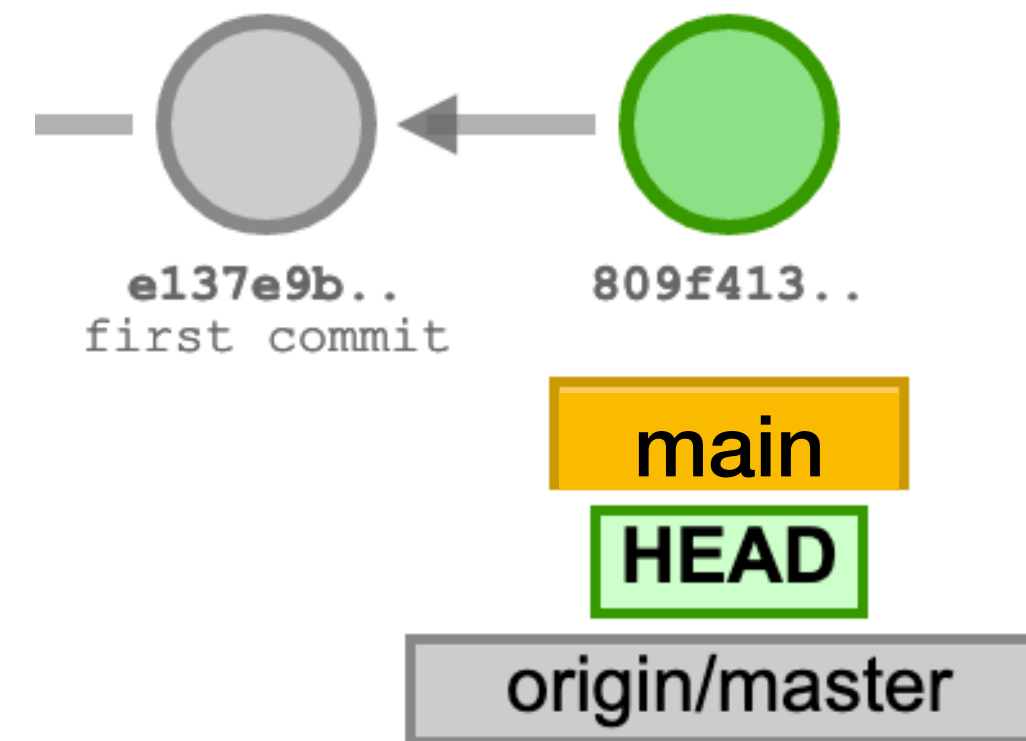
Example

```
$ git clone ...
```

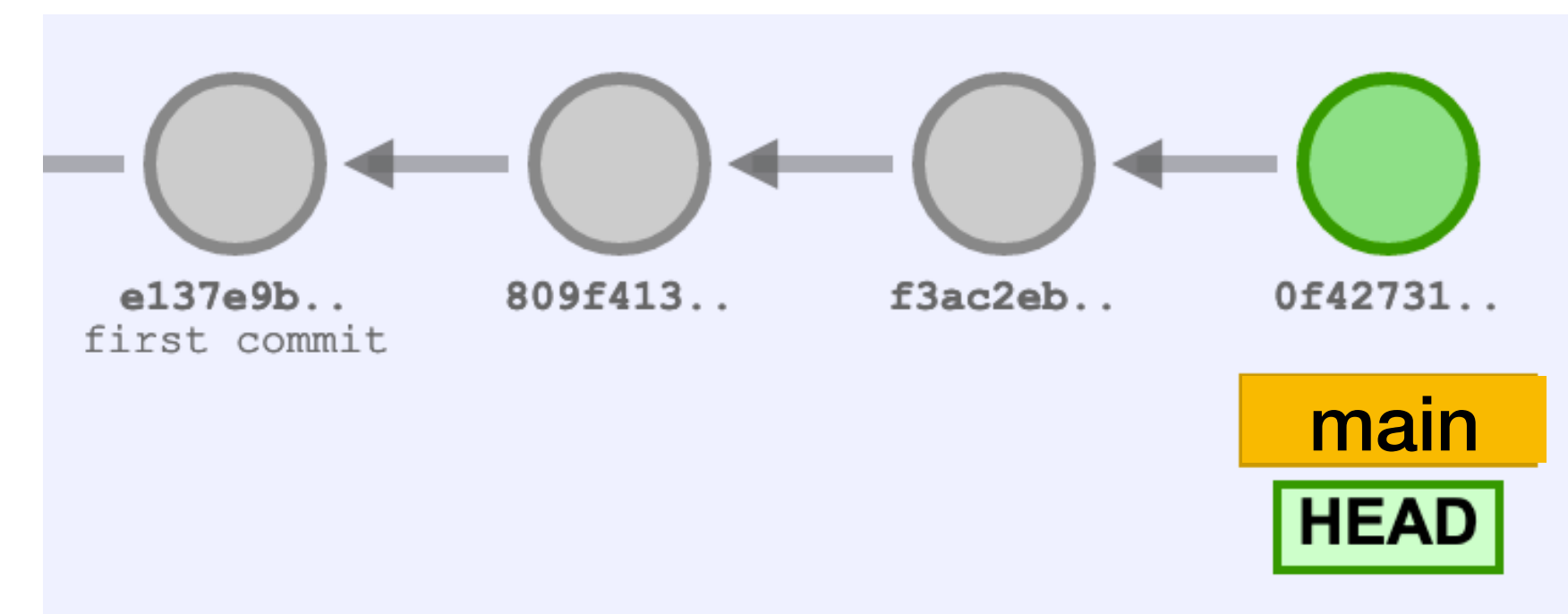
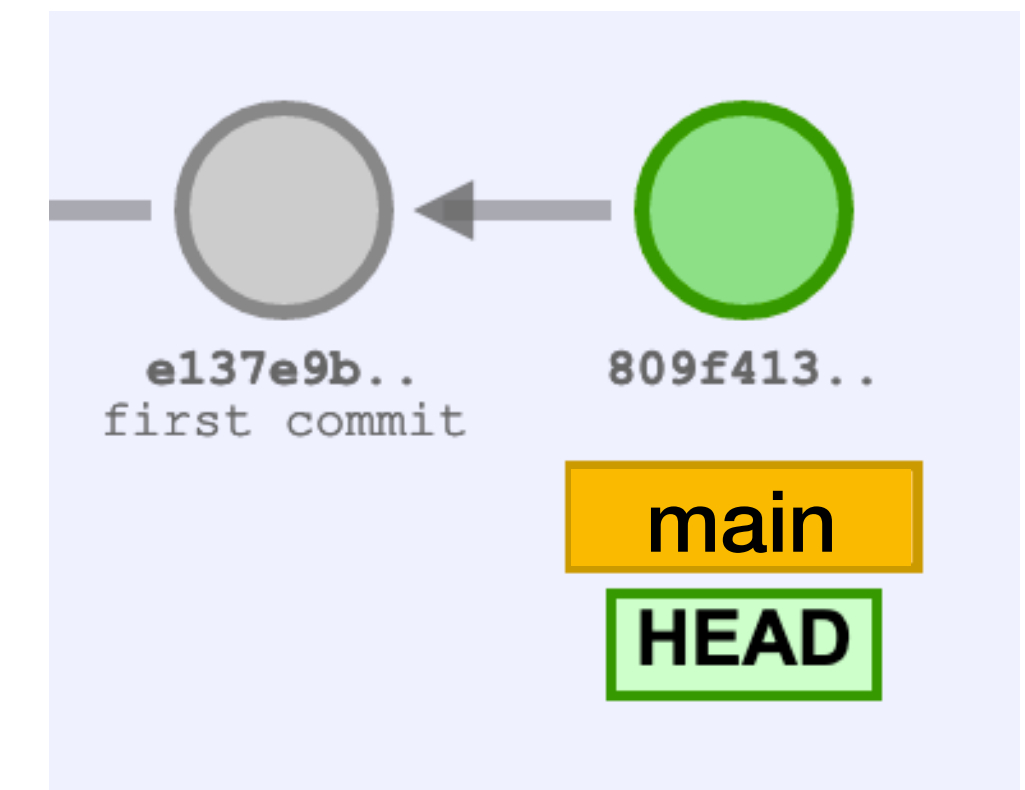
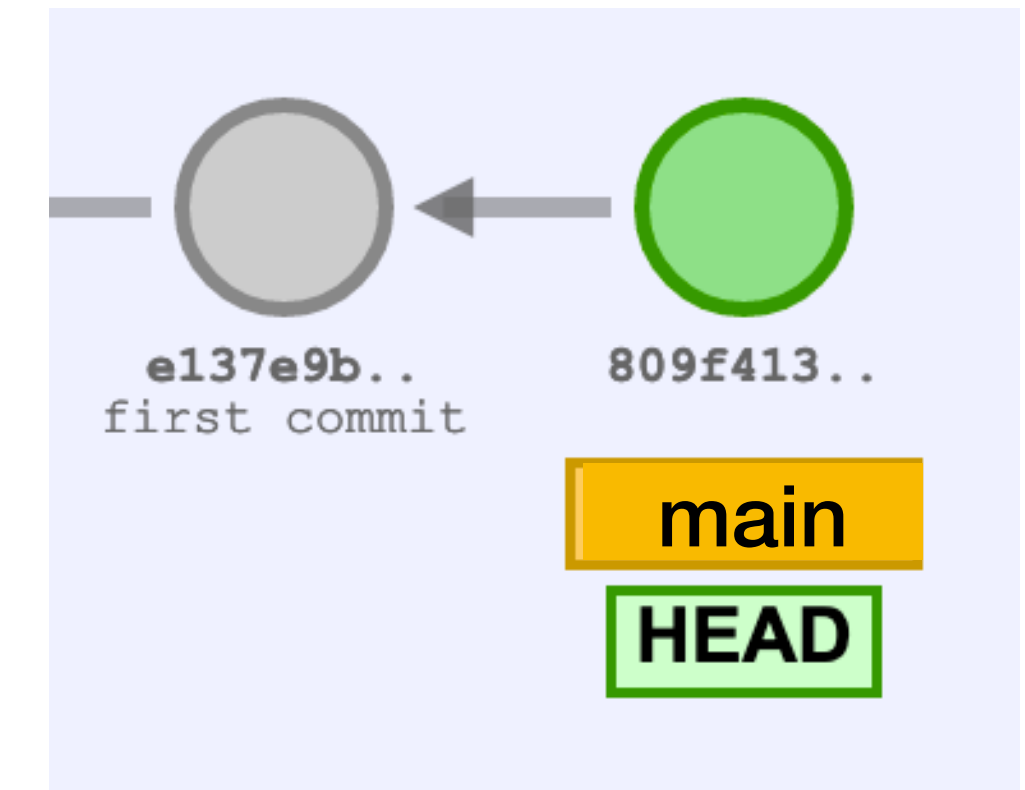
```
$ git add ...  
$ git commit  
$ git add ...  
$ git commit
```

```
$ git push
```

Local repository



Origin



Pulling from the remote server

```
$ git pull
```

Pulls changes from the remote server to the local repo and **merges** with the local changes

```
$ git pull --rebase
```

Pulls changes from the remote server to the local repo and **rebases** local commits on top of remote commits

Pulling with merging

Commits from the remote will be added to the local repository

If there are local commits, git tries to merge them by creating a new commit

```
      A---B---C main on origin
      /
D---E---F---G main
      ^
      origin/main in your repository
```

```
      A---B---C origin/main
      /           \
D---E---F---G---H main
```

Pulling with rebasing

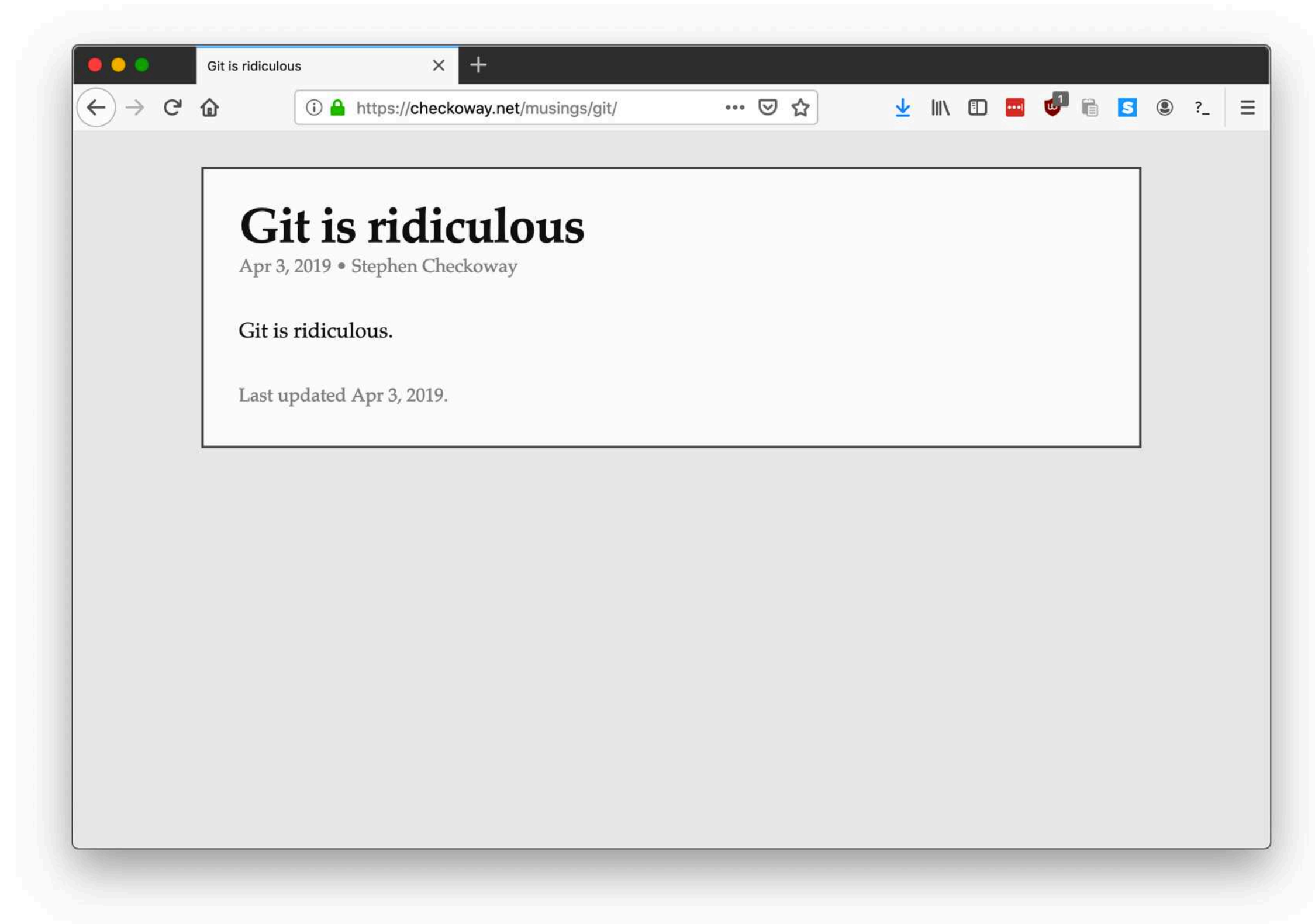
Commits from the remote will be added to the local repository
If there are local commits, git replays them on top of the new commits

```
      A---B---C  main on origin
      /
D---E---F---G  main
      ^
      origin/main in your repository
```

```
                        origin/main
                        v
D---E---A---B---C---F'--G'  main
```

Reminder: Git is ridiculous

Warning: Git is ridiculous



Gitting help

```
$ git --help
```

```
$ git init --help
```

```
$ git clone --help
```

```
$ git add --help
```

```
$ git commit --help
```

```
$ git push --help
```

```
$ git pull --help
```

Basic Workflow

Basic Workflow

Create the repository by clicking on the link in the homework

Basic Workflow

Create the repository by clicking on the link in the homework

Clone the repository on lab machines using `$ git clone <url>`

Basic Workflow

Create the repository by clicking on the link in the homework

Clone the repository on lab machines using `$ git clone <url>`

Add files to be committed with `$ git add <filename>`

Basic Workflow

Create the repository by clicking on the link in the homework

Clone the repository on lab machines using `$ git clone <url>`

Add files to be committed with `$ git add <filename>`

Create a commit (snapshot) of added files using `$ git commit`

Basic Workflow

Create the repository by clicking on the link in the homework

Clone the repository on lab machines using `$ git clone <url>`

Add files to be committed with `$ git add <filename>`

Create a commit (snapshot) of added files using `$ git commit`

Push files to the server using `$ git push`

Basic Workflow

Create the repository by clicking on the link in the homework

Clone the repository on lab machines using `$ git clone <url>`

Add files to be committed with `$ git add <filename>`

Create a commit (snapshot) of added files using `$ git commit`

Push files to the server using `$ git push`

See the current state of the files using `$ git status`

Commit often

Commits are cheap, commit often

Commits can be reverted by `git revert`

- Makes a new commit that undoes the old commit
- `$ git revert <commit_hash>`

Commits that haven't been pushed can be undone completely by `git reset`

- `$ git reset --hard <commit_hash>`

Demo at <https://git-school.github.io/visualizing-git/#free-remote>