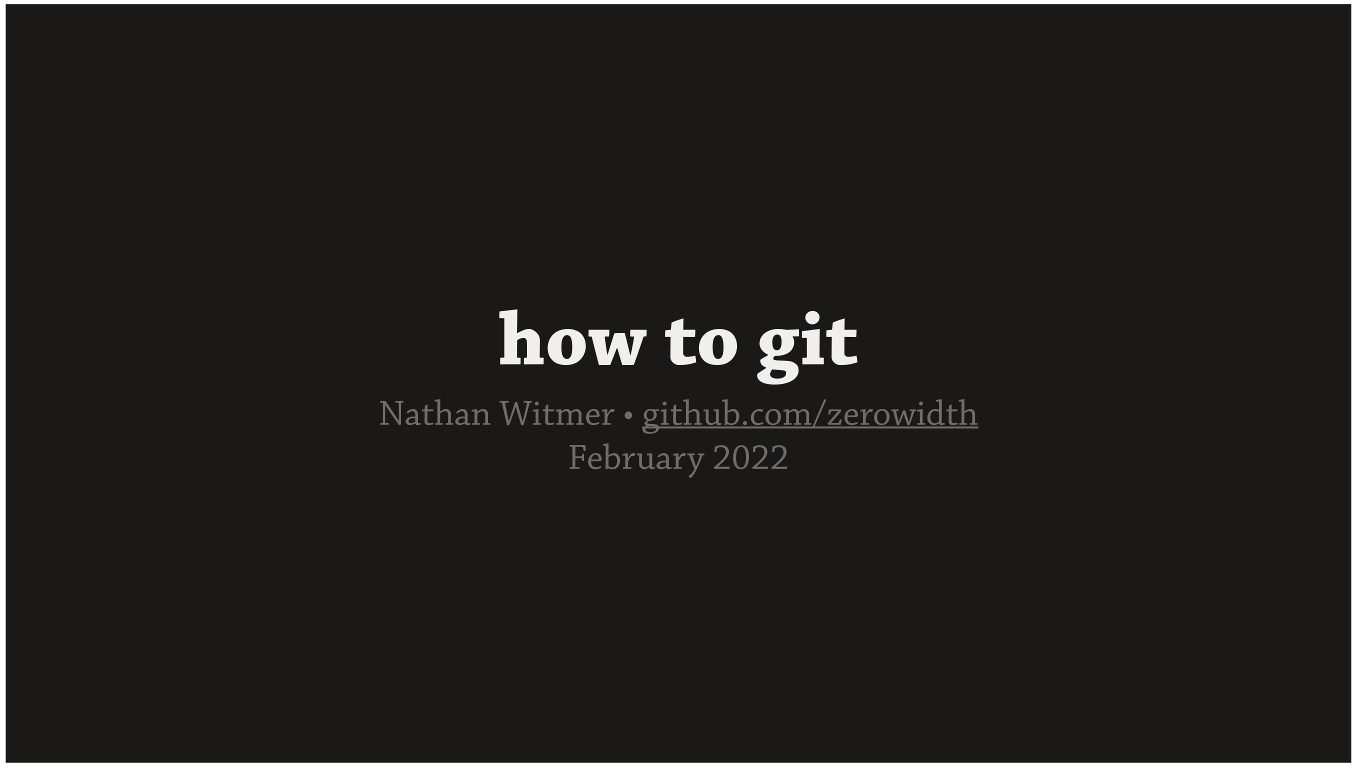


how to git

Nathan Witmer • github.com/zerowidth
February 2022

Intro
Who’s familiar with git? GitHub?
I’m assuming all of you are at least a little familiar with git, but I’ll give a quick overview and then demonstrate visually what git commands do, in hopes that you can understand it better.
I’ll be covering the basics of git, give some examples of what you can do, then show you GitHub, and talk about how we use both to write software.

IBM 2004-2008

- Oracle DBA
- ETL developer (Python)

CollectiveIntellect 2008-2011

- Analytics web app development (Rails)

LivingSocial 2011-2013

- **C**heckout and payment processing (Rails)

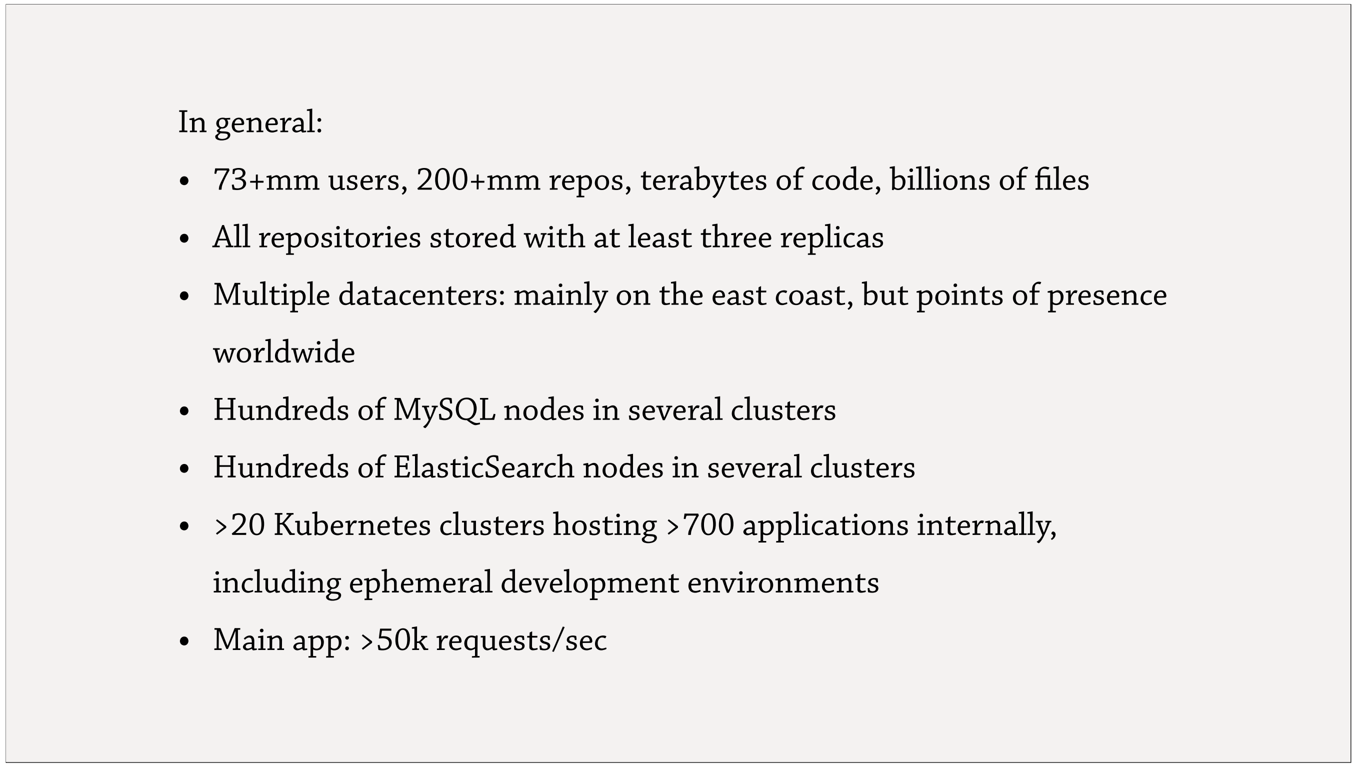
Quick overview of my career

GitHub since 2013

- Back-end application development (Rails), performance tuning (MySQL, etc.), systems refactoring/extraction
- Data infrastructure
    - Kafka cluster operations, schema management, client libraries, stream processors
    - Job processing service, client libraries

My team:

- Kafka operations and developer tools
  - 250k messages/sec: analytics, application events, audit logs, search indexing
- Job queue service: 15k jobs/sec
  - supports github.com, webhooks, notifications, etc.



In general:

- 73+mm users, 200+mm repos, terabytes of code, billions of files
- All repositories stored with at least three replicas
- Multiple datacenters: mainly on the east coast, but points of presence worldwide
- Hundreds of MySQL nodes in several clusters
- Hundreds of ElasticSearch nodes in several clusters
- >20 Kubernetes clusters hosting >700 applications internally, including ephemeral development environments
- Main app: >50k requests/sec

Since this is the “Big Data” class, I’ll talk about the scale of GitHub for a minute.

https://github.blog/category/engineering/

You can read more about some of the projects we’re working on here.

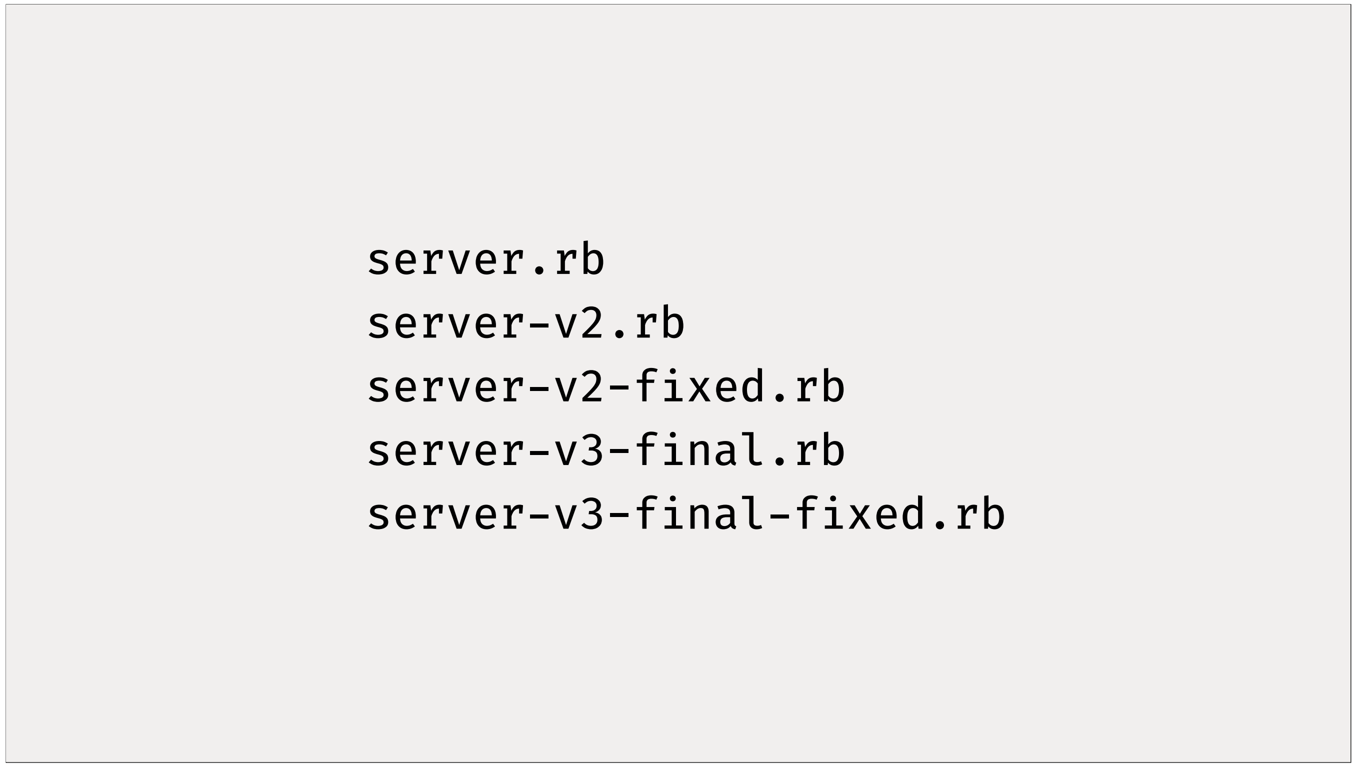
Version Control

But: I’m going to talk about something more fundamental to what we do as a company: version control.

We’d market GitHub as “developer collaboration tools”. But these are the technical bits that the whole thing was built on.

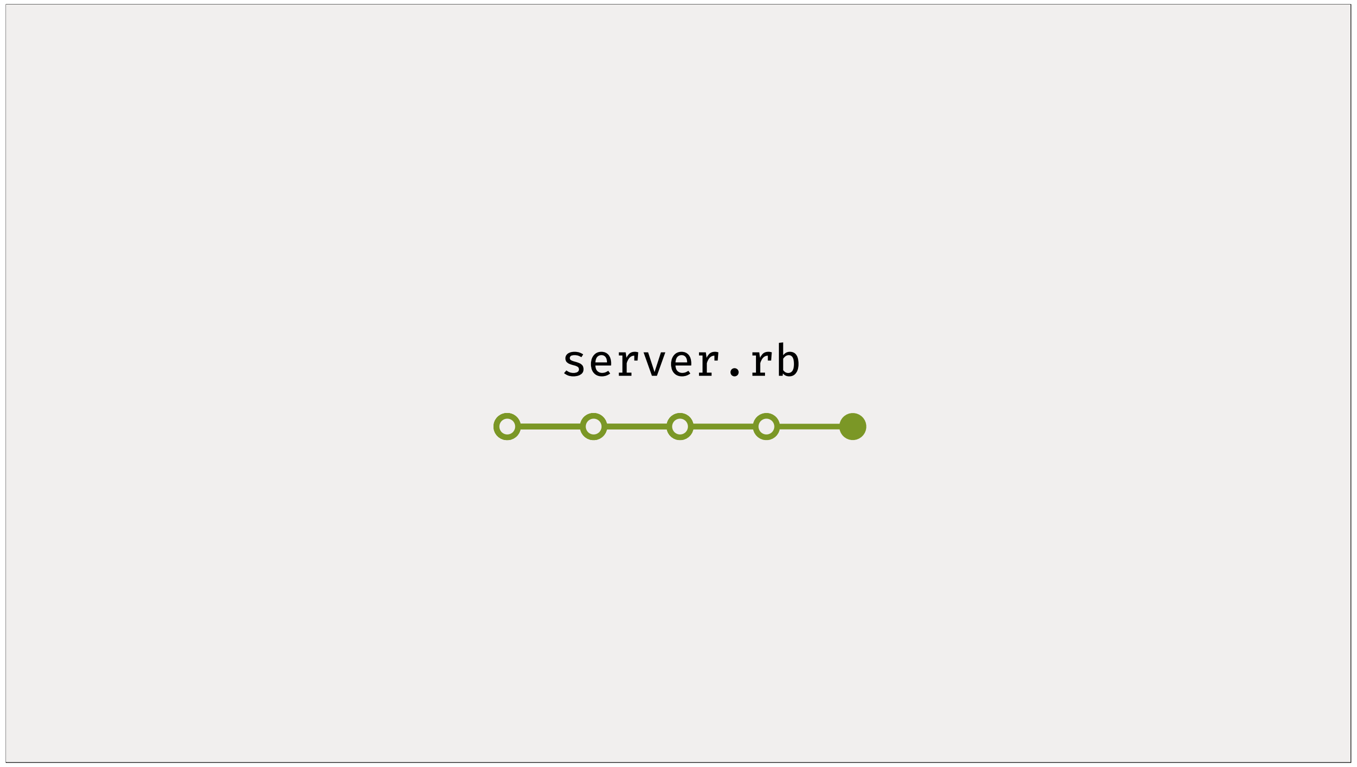
server.rb

Let’s say you have a ruby program that runs a server. You want to have different versions: the official one running in production, but also you want to make changes to it.



```
server.rb
server-v2.rb
server-v2-fixed.rb
server-v3-final.rb
server-v3-final-fixed.rb
```

A really simple way to do this is by just copying and renaming this file. Does this look familiar to anyone?



server.rb

Version control gives us a different way of doing this.
It can track the changes to a file separately from the file itself. We can capture and record snapshots of the file anytime we care to take one.
It gives us a history of the file over time, so you can go back to see what the code was at any point, or roll back to a known good state. Hopefully, too, it can help you find out who did things, and why.

Git

Specifically, I want to talk about git. My goal for this lecture is not that you become git experts, but that you can at least get a general idea of what’s going on under the hood and where to look if something goes wrong.

Distributed version control system

Git is a distributed version control system, created in 2005 to support linux kernel development.
It was rough around the edges for a long time, and the learning curve still isn’t that gentle.

Version control system

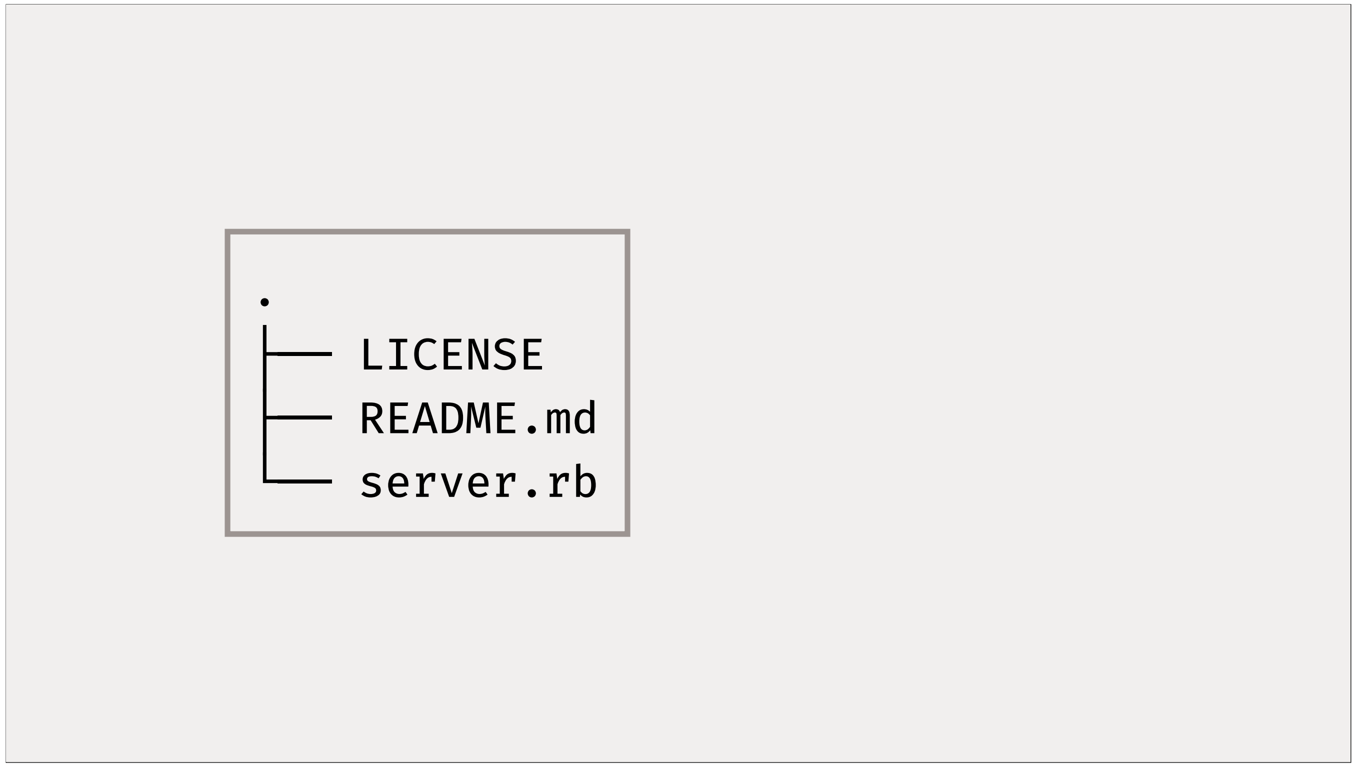
First, it’s a version control system. It tracks the contents of files and changes over time.

Distributed

Secondly, it's distributed. This means there's no central store or server. You don't have to connect to a server running somewhere to do anything. Every copy is self-contained.
(well, except for GitHub… but we do more than just git)

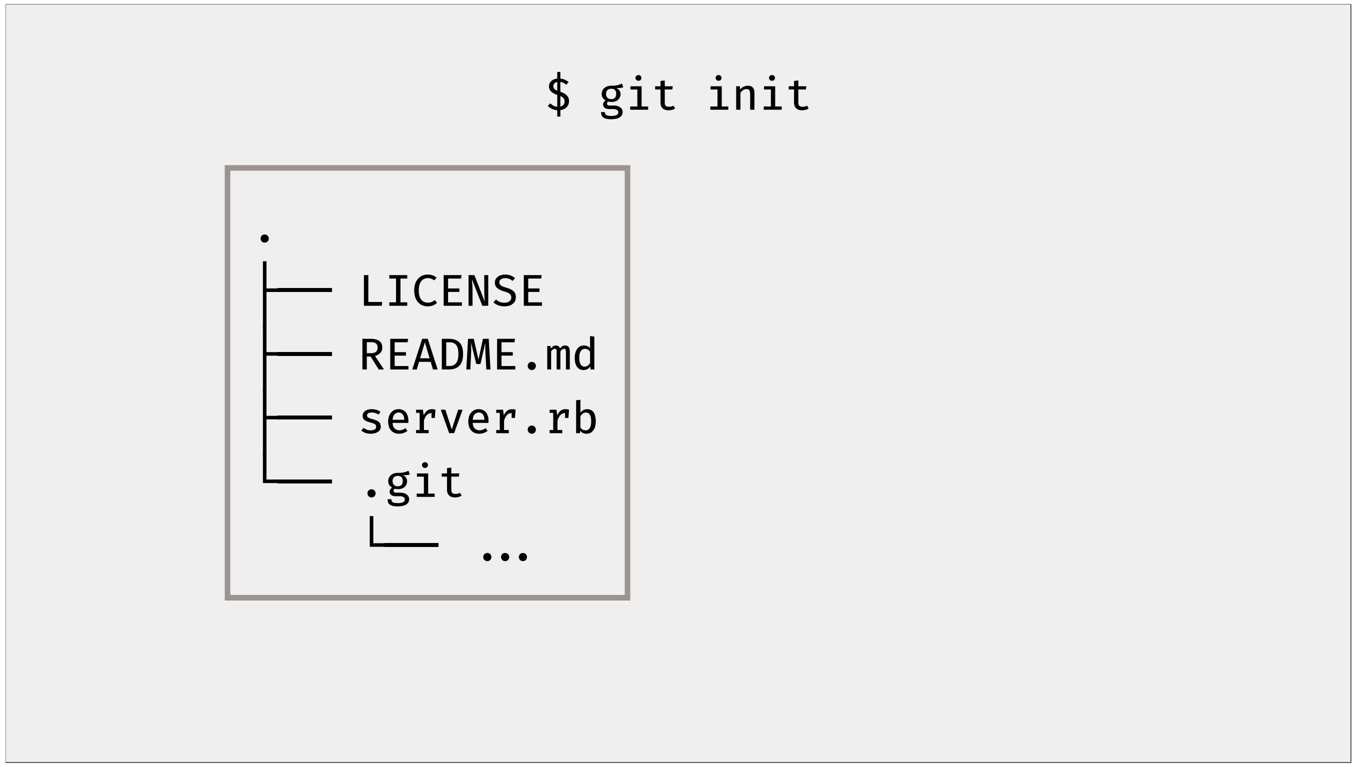
Repository

Git can do a lot for us. Let’s start by talking about repositories.
Repositories are where git tracks the history of a project.



```
.
├── LICENSE
├── README.md
└── server.rb
```

Let’s say we have a project. It’s got a license, a README, and a ruby file.

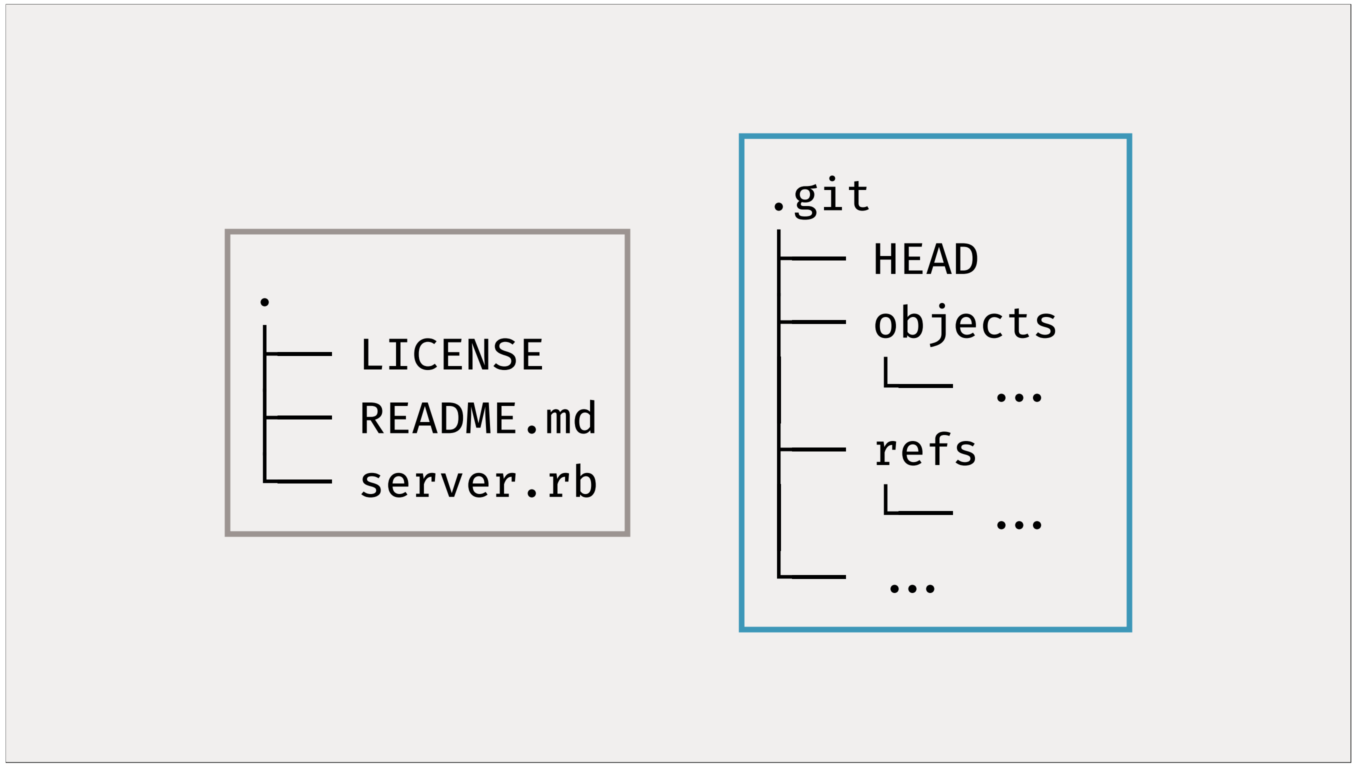


```
$ git init
```

```
.
├── LICENSE
├── README.md
├── server.rb
└── .git
    └──  ...
```

When we initialize a new git repository, we’ll end up with a new directory, `.git`. This is where git stores all its data. You won’t ever need to touch anything in here directly, but that’s where it lives.

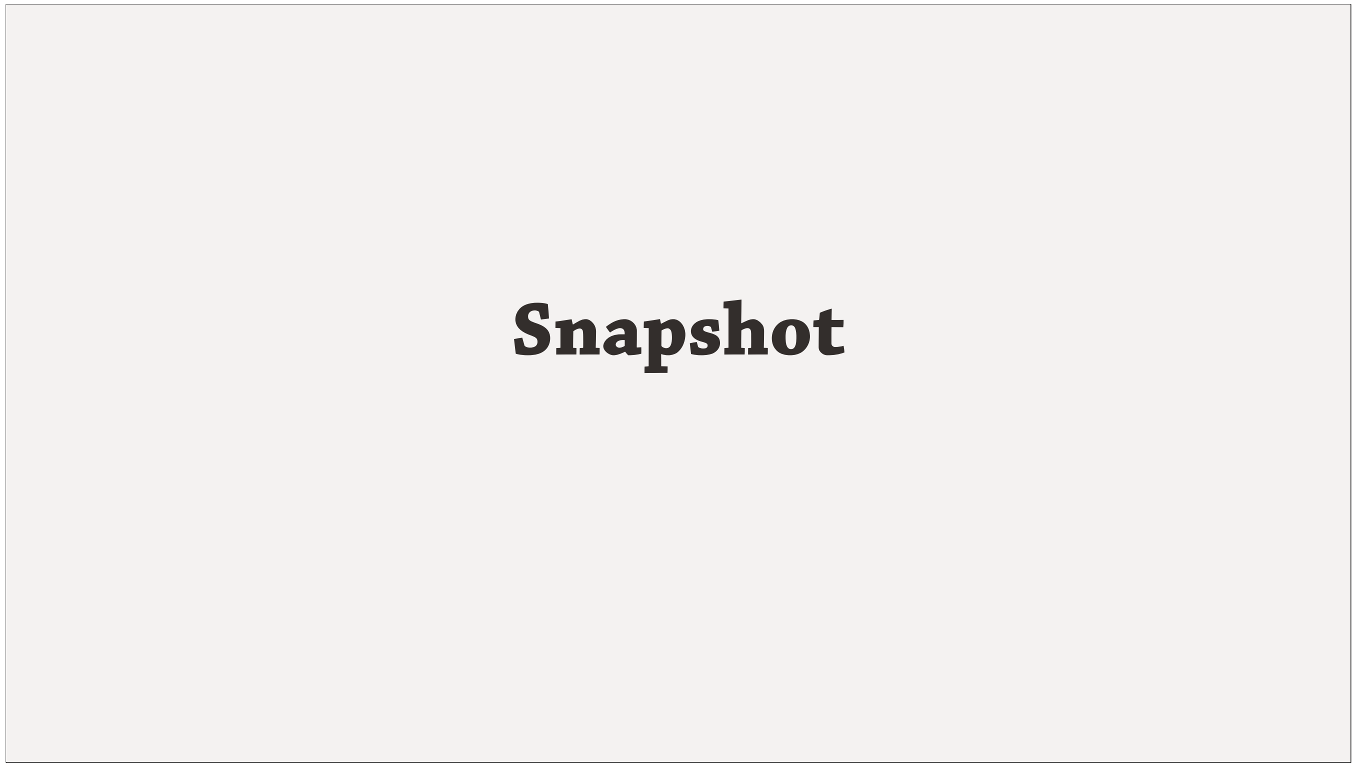
The .git folder is right alongside your code, but generally hidden. I’ll draw it off to the side instead.



```
.
├── LICENSE
├── README.md
└── server.rb
```

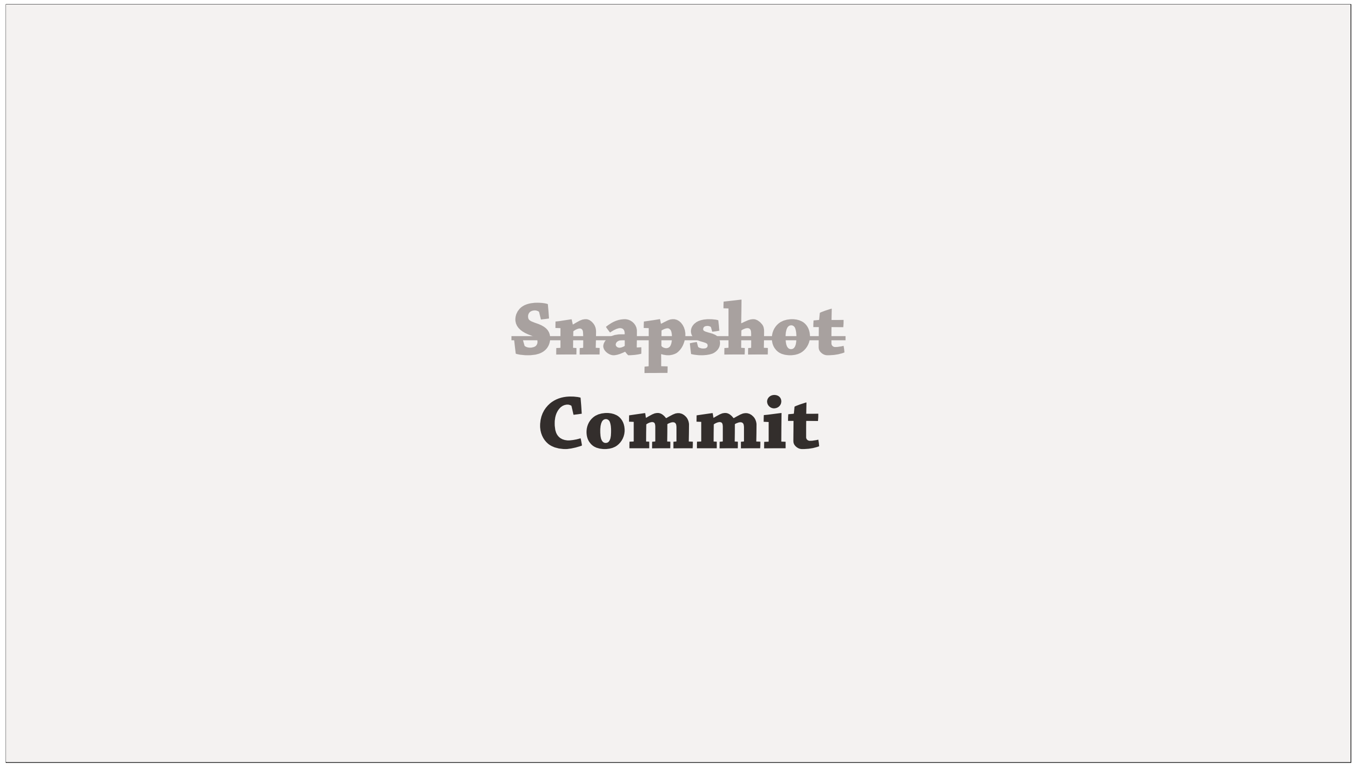
```
.git
├── HEAD
├── objects
│   └──  ...
├── refs
│   └──  ...
└── ...
```

The .git folder is right alongside your code, but generally hidden, so I’ll draw it off to the side instead.



Snapshot

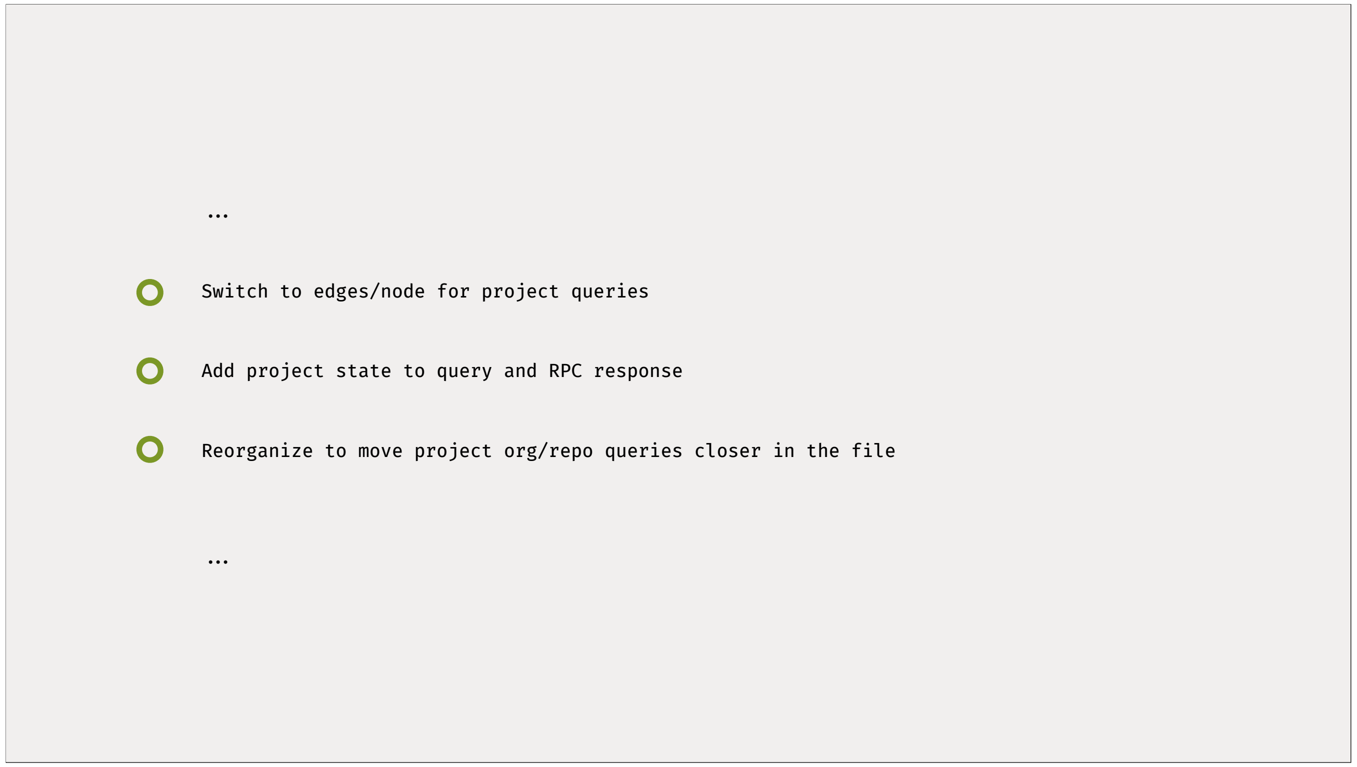
I used the word “snapshot” before to talk about tracking changes to a file.



~~Snapshot~~

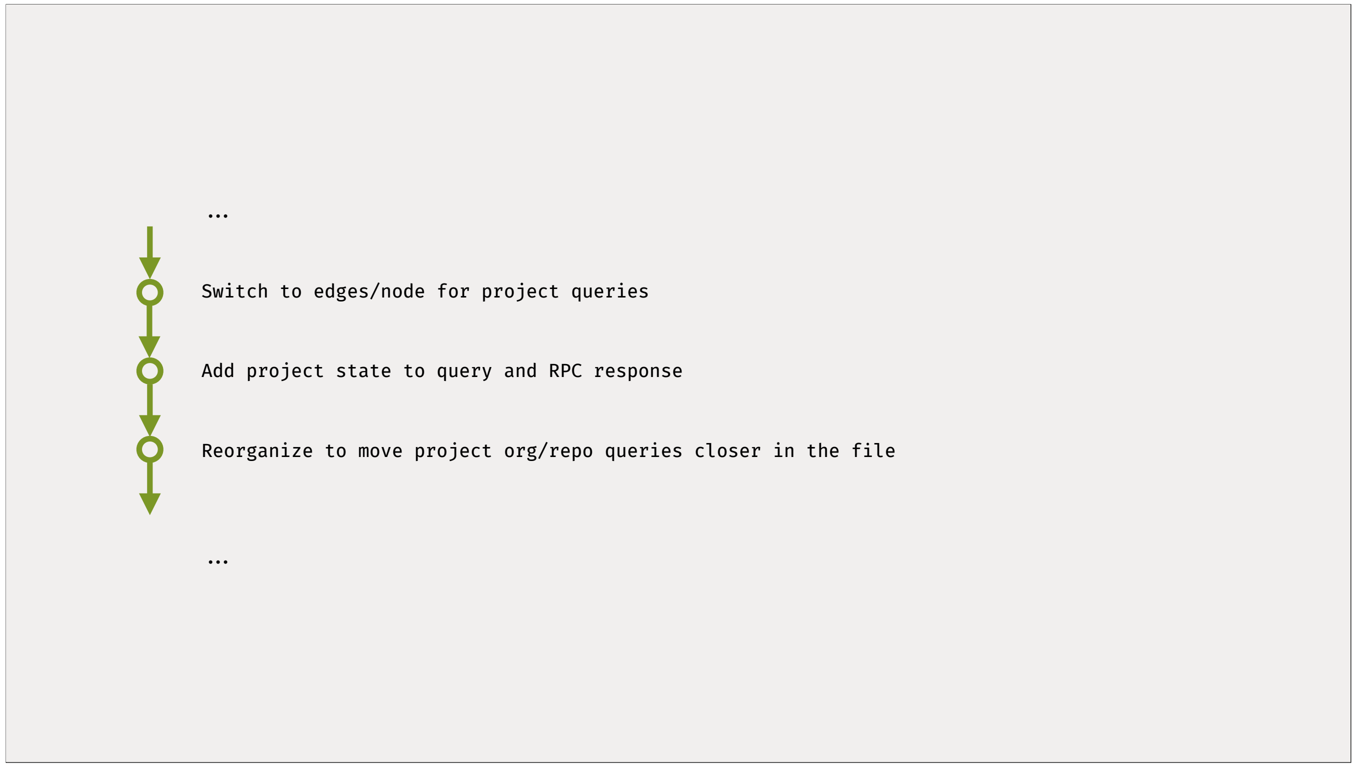
Commit

in git, a snapshot is called a “commit”. Commits are the fundamental building block in git. Let’s look at a commit to see what’s in it.



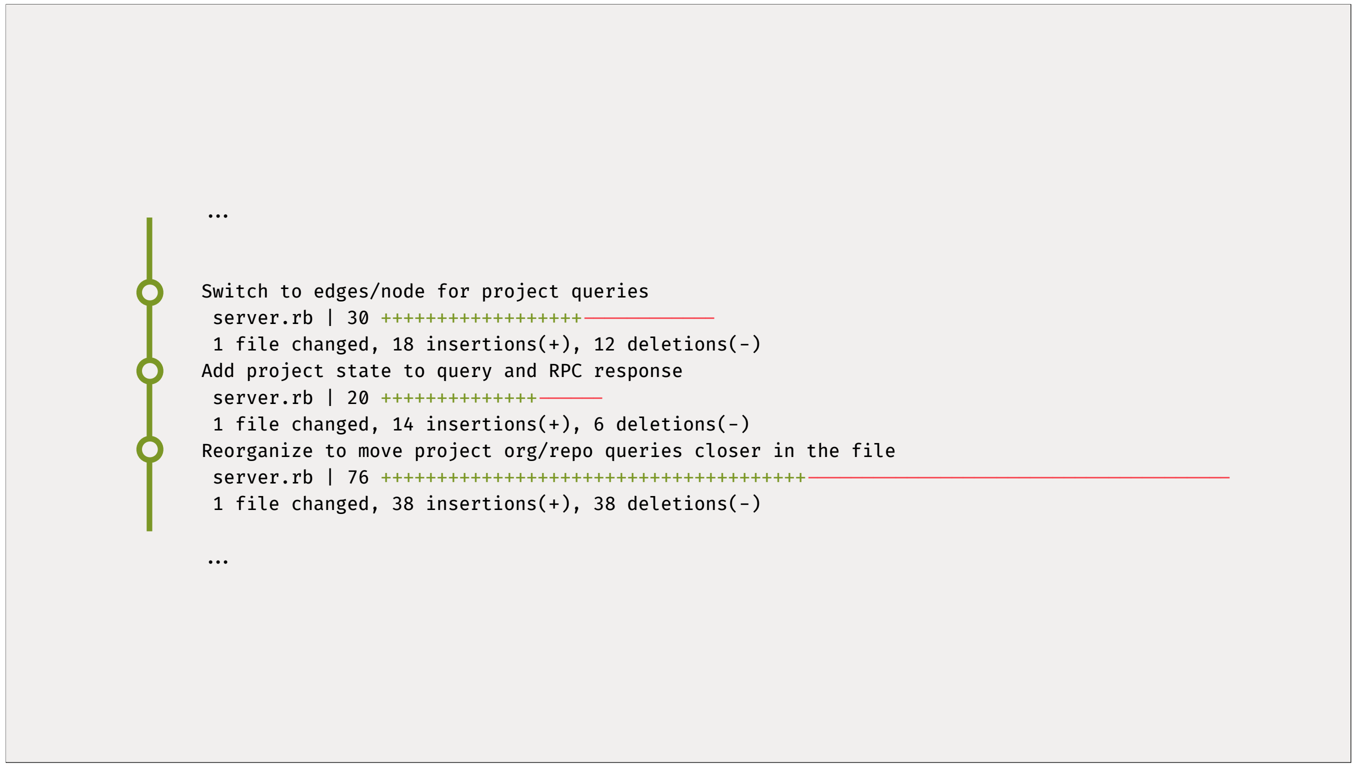
```
...

○ Switch to edges/node for project queries

○ Add project state to query and RPC response

○ Reorganize to move project org/repo queries closer in the file

...
```

Let’s make some commits while we work on a project. This is fine, but it’s not very helpful to just have those hanging around by themselves. It would be a lot more useful if they were linked together.



```
...

Switch to edges/node for project queries

Add project state to query and RPC response

Reorganize to move project org/repo queries closer in the file

...
```

By linking commits together, we can create a history over time.

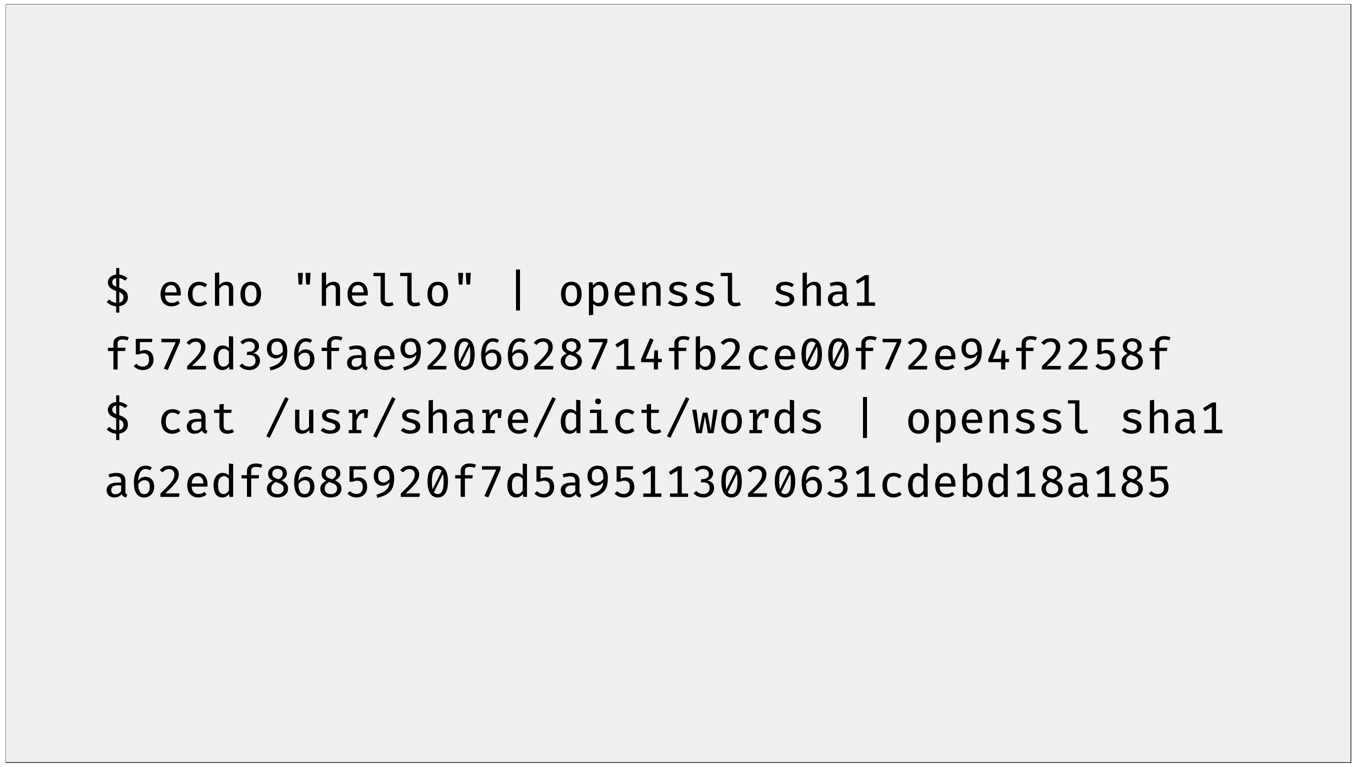


```
...


Switch to edges/node for project queries
 server.rb | 30 ++++++++++++++++++------------
 1 file changed, 18 insertions(+), 12 deletions(-)
Add project state to query and RPC response
 server.rb | 20 ++++++++++++++------
 1 file changed, 14 insertions(+), 6 deletions(-)
Reorganize to move project org/repo queries closer in the file
 server.rb | 76 ++++++++++++++++++++++++++++++++++++++--------------------------------------
 1 file changed, 38 insertions(+), 38 deletions(-)

 ...
```

And since each commit is linked to the previous one, we can also compare the two and see what changed between each one. Here, we have three commits that each changed a single file by adding and removing some lines.
So what’s in a git commit?

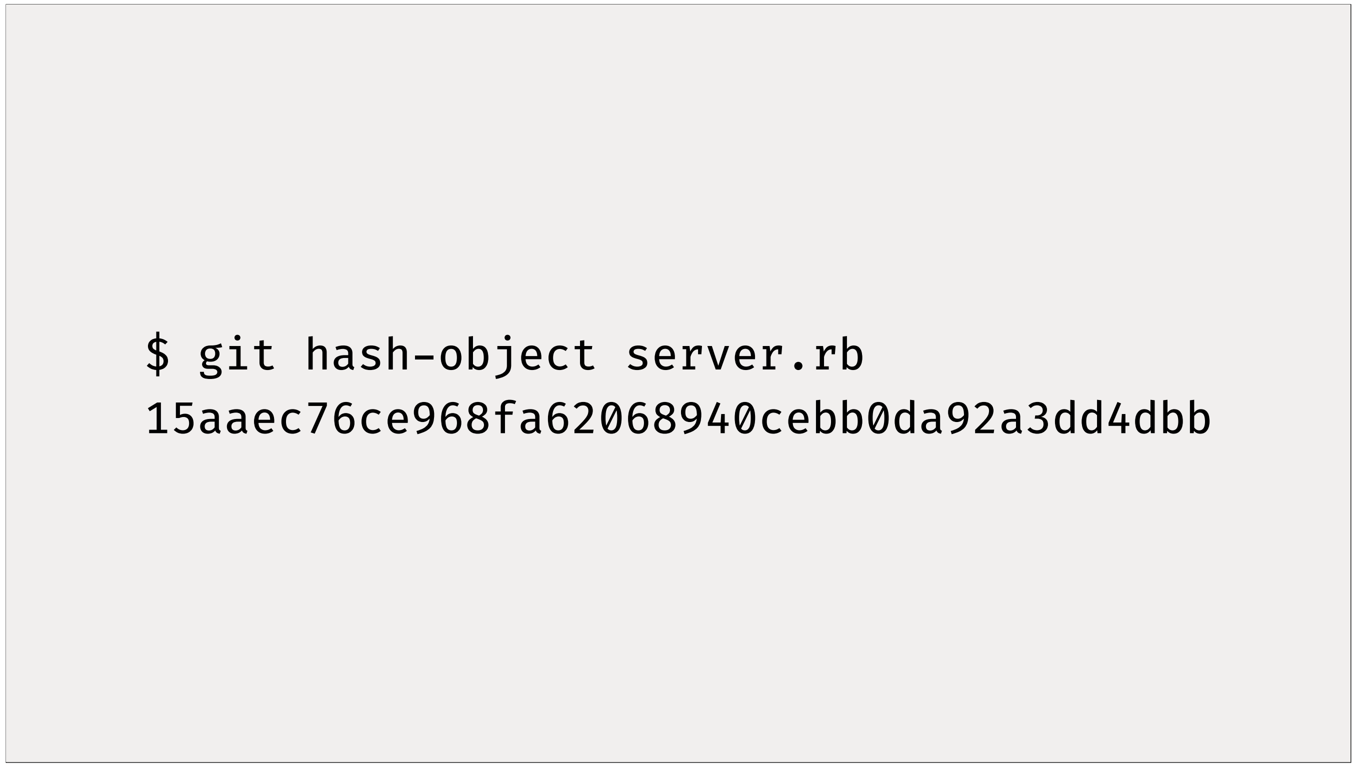
SHA1 Hash

Before I talk about that, let’s talk about the SHA1 hash function.



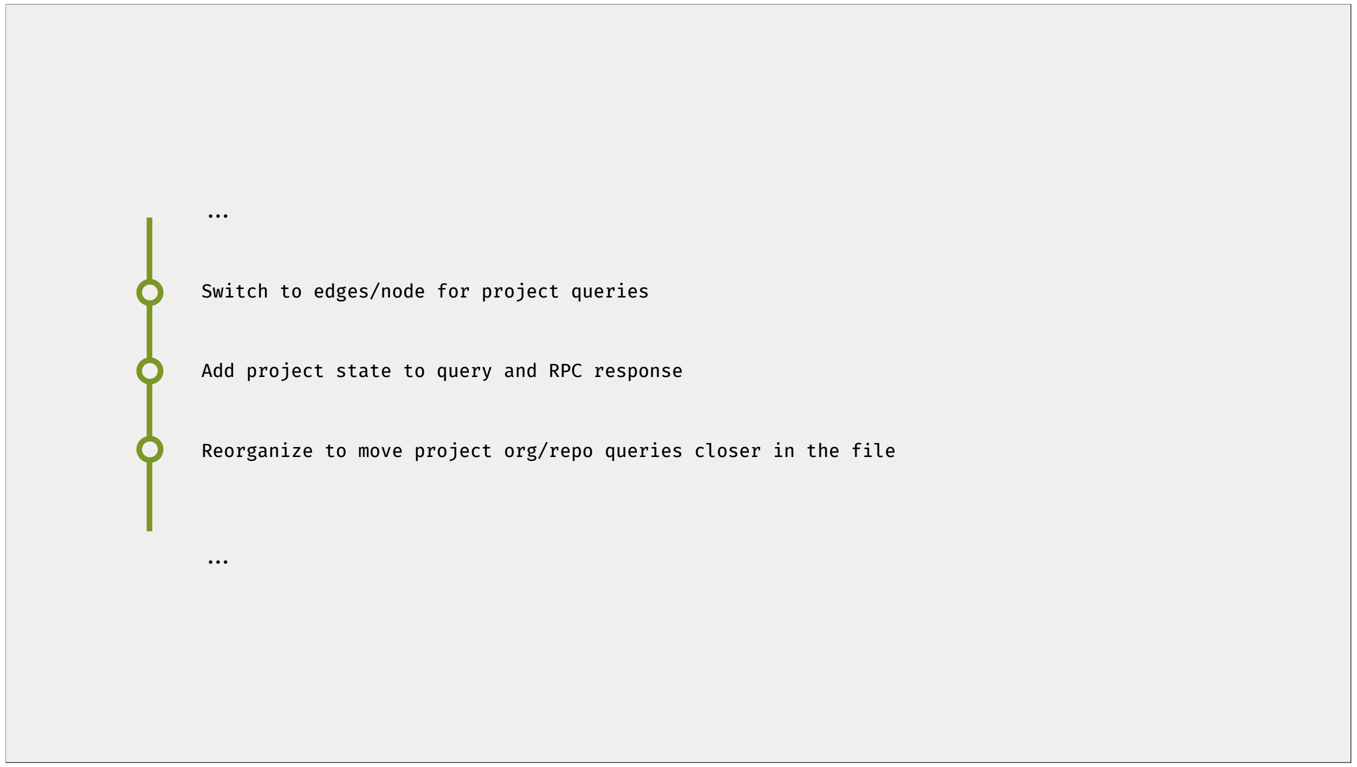
```
$ echo "hello" | openssl sha1
f572d396fae9206628714fb2ce00f72e94f2258f
$ cat /usr/share/dict/words | openssl sha1
a62edf8685920f7d5a95113020631cdebd18a185
```

A hash function takes input and returns a fixed size output. Both a five-byte input and a 2 megabyte file become 40 character strings (representing 20 bytes). This hash function always produces the same output if the input is the same.



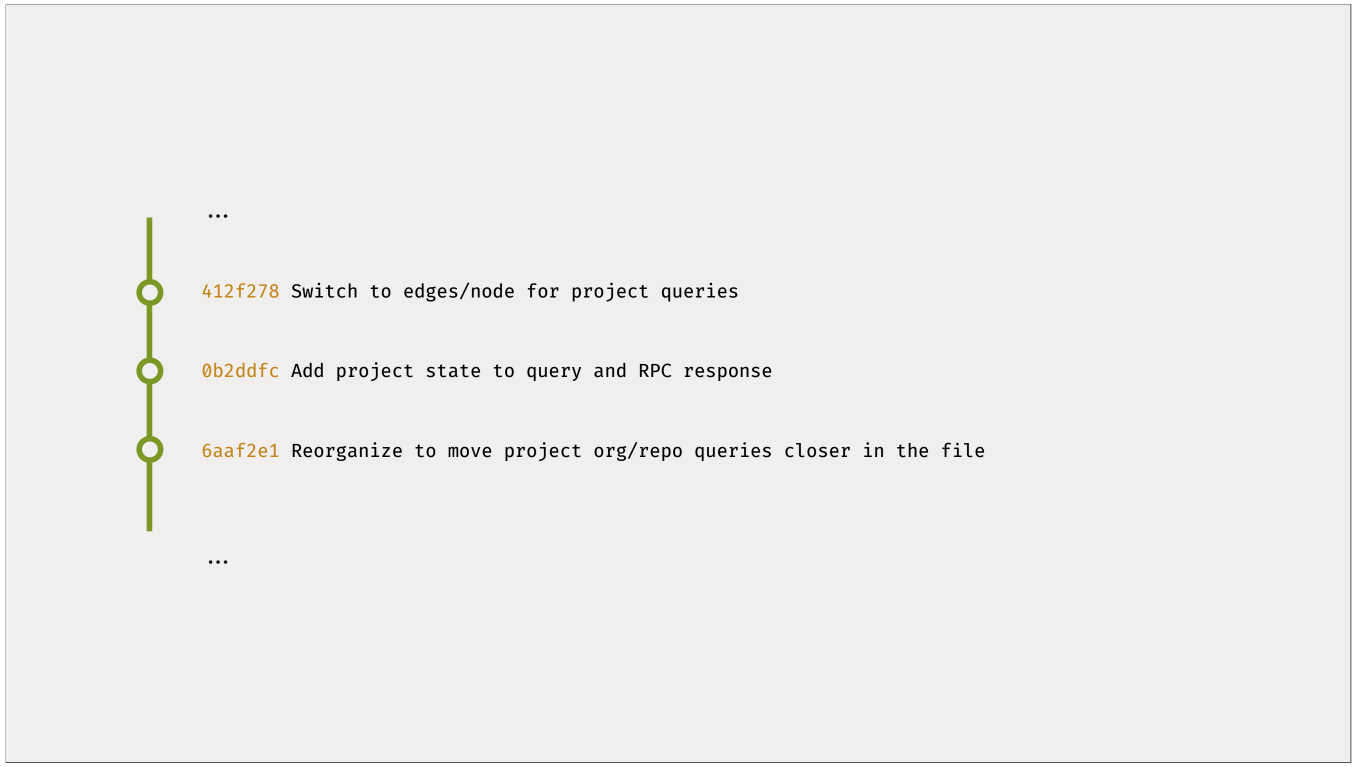
```
$ git hash-object server.rb
15aaec76ce968fa62068940cebb0da92a3dd4dbb
```

Git uses SHA1 hashes to identify everything. Given an input file, it generates a hash for it. If the file changes, so does the hash. It’s nearly impossible for two different inputs to result in the same hash, so this is useful.



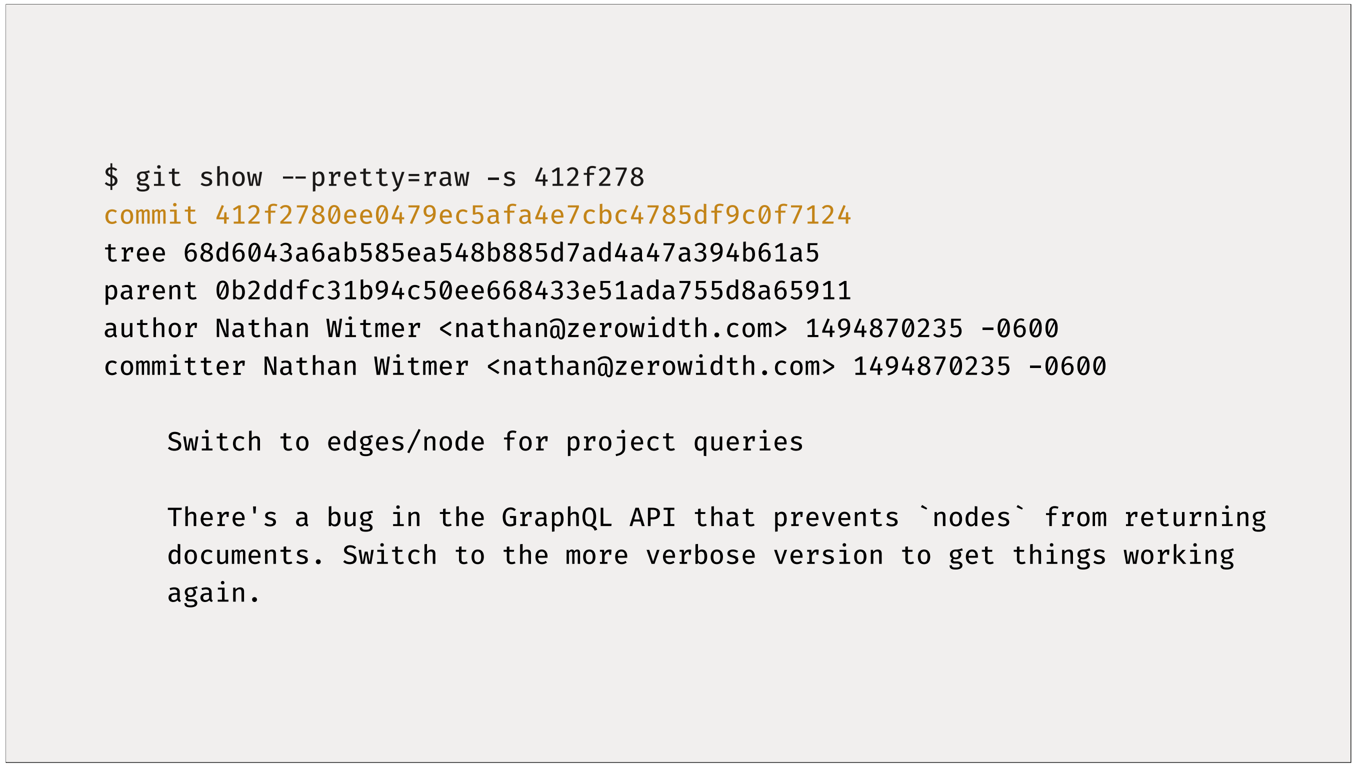
```
...
Switch to edges/node for project queries
Add project state to query and RPC response
Reorganize to move project org/repo queries closer in the file
...
```

Looking back at this history, this is actually from one of my repositories.



```
…

412f278 Switch to edges/node for project queries

0b2ddfc Add project state to query and RPC response

6aaf2e1 Reorganize to move project org/repo queries closer in the file

…
```

And each of these commits is identified by a SHA1. Git usually shows only as much of the hash as necessary to disambiguate.

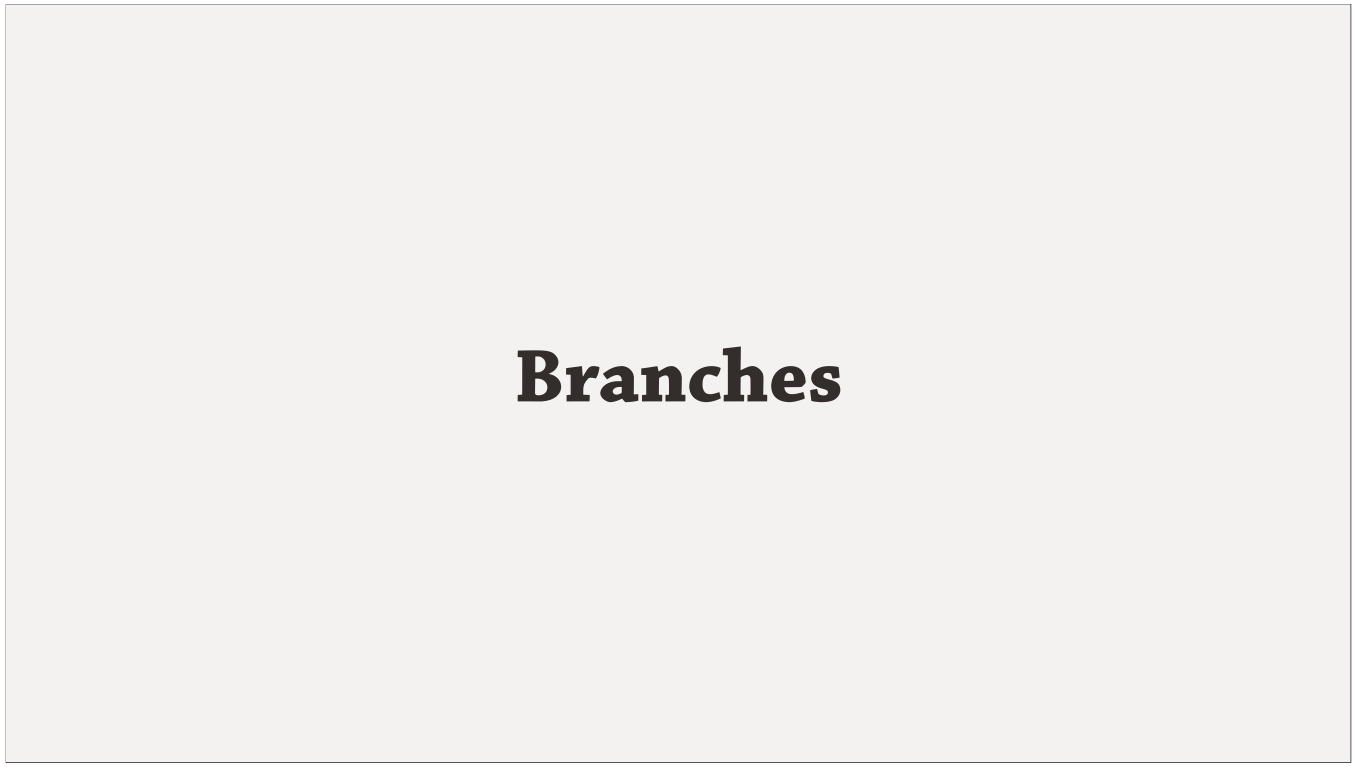


```
$ git show --pretty=raw -s 412f278
commit 412f2780ee0479ec5afa4e7cbc4785df9c0f7124
tree 68d6043a6ab585ea548b885d7ad4a47a394b61a5
parent 0b2ddfc31b94c50ee668433e51ada755d8a65911
author Nathan Witmer <nathan@zerowidth.com> 1494870235 -0600
committer Nathan Witmer <nathan@zerowidth.com> 1494870235 -0600

    Switch to edges/node for project queries

    There's a bug in the GraphQL API that prevents `nodes` from returning
    documents. Switch to the more verbose version to get things working
    again.
```

So now, let’s take a look at one of these commits. I’m going to ask git for basically the raw version it stores, so I can talk about a few parts.
I’m asking for the commit by only giving git the first seven characters, but that’s all it needs.
Here’s the contents of the commit. It’s giving us the full SHA1 of the commit, and then a few other things: the sha1 of the “tree”, that is, the information about the files and paths in this commit, and the sha1 of the parent of this commit, that is, the previous commit in the history of this repositroy.
If we dug into that tree, we’d see the hashes of all the paths in the commit, which includes the hashes of the files. So it’s SHA1 all the way down.

Next some metadata: who wrote this commit, and when.
Finally, a subject and a body: these are like the subject/body of an email: a short message about what’s in the commit, and then an optional body with more explanation.

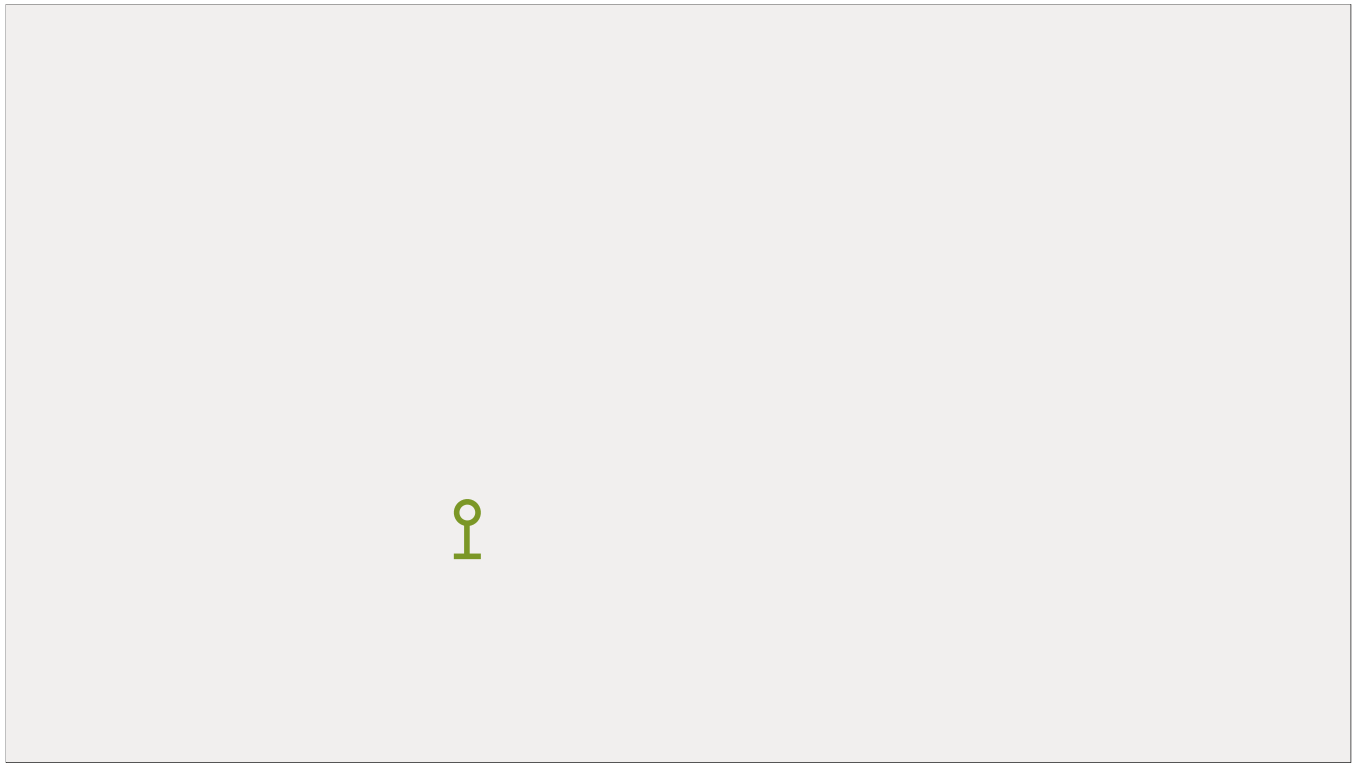
I mentioned it’s sha1 hashes the entire way: every commit has the SHA of its contents and prior commit, which has the hash of its contents, etc. So the whole repository contents and history can be checked for accuracy.



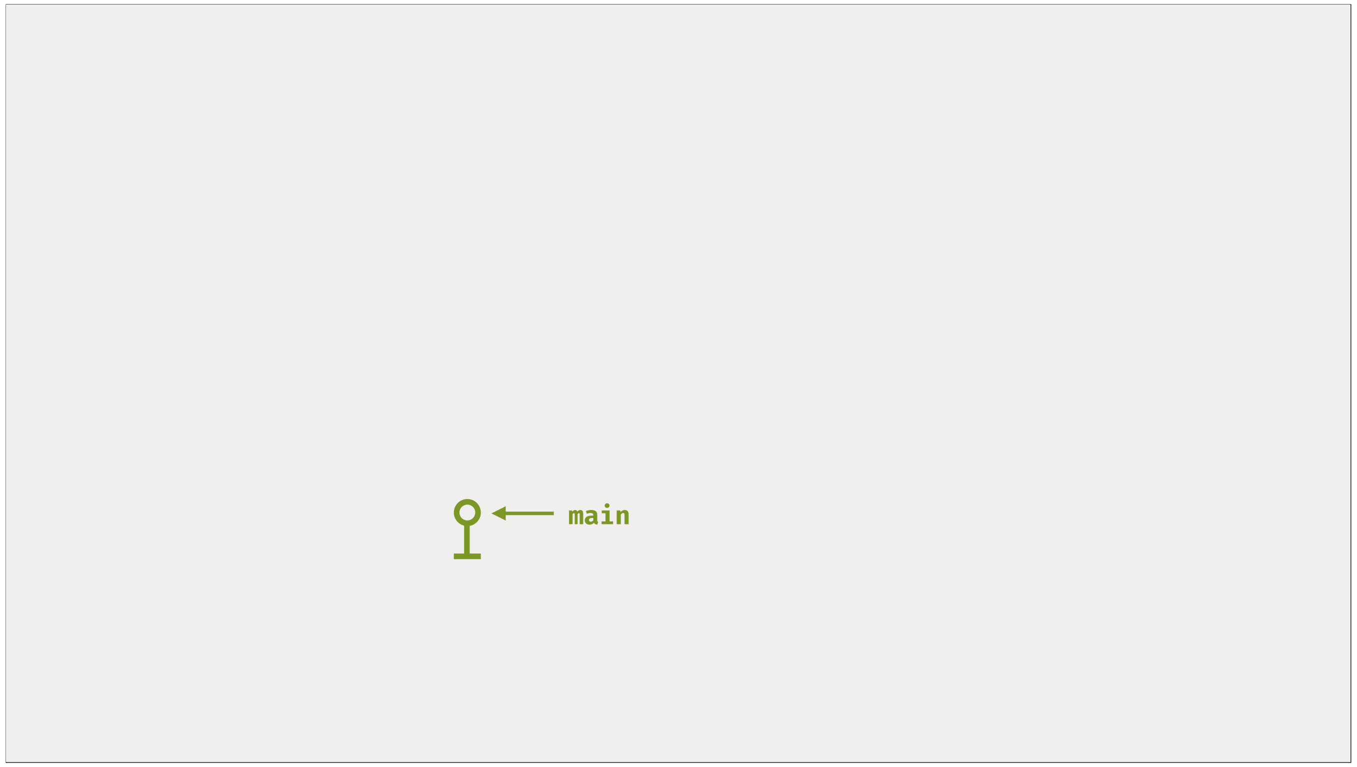
Branches

Next I want to talk about branches in git.

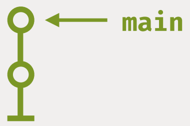
Let’s start with an empty repository. There’s nothing here yet, not really, so there’s just this line to represent that.

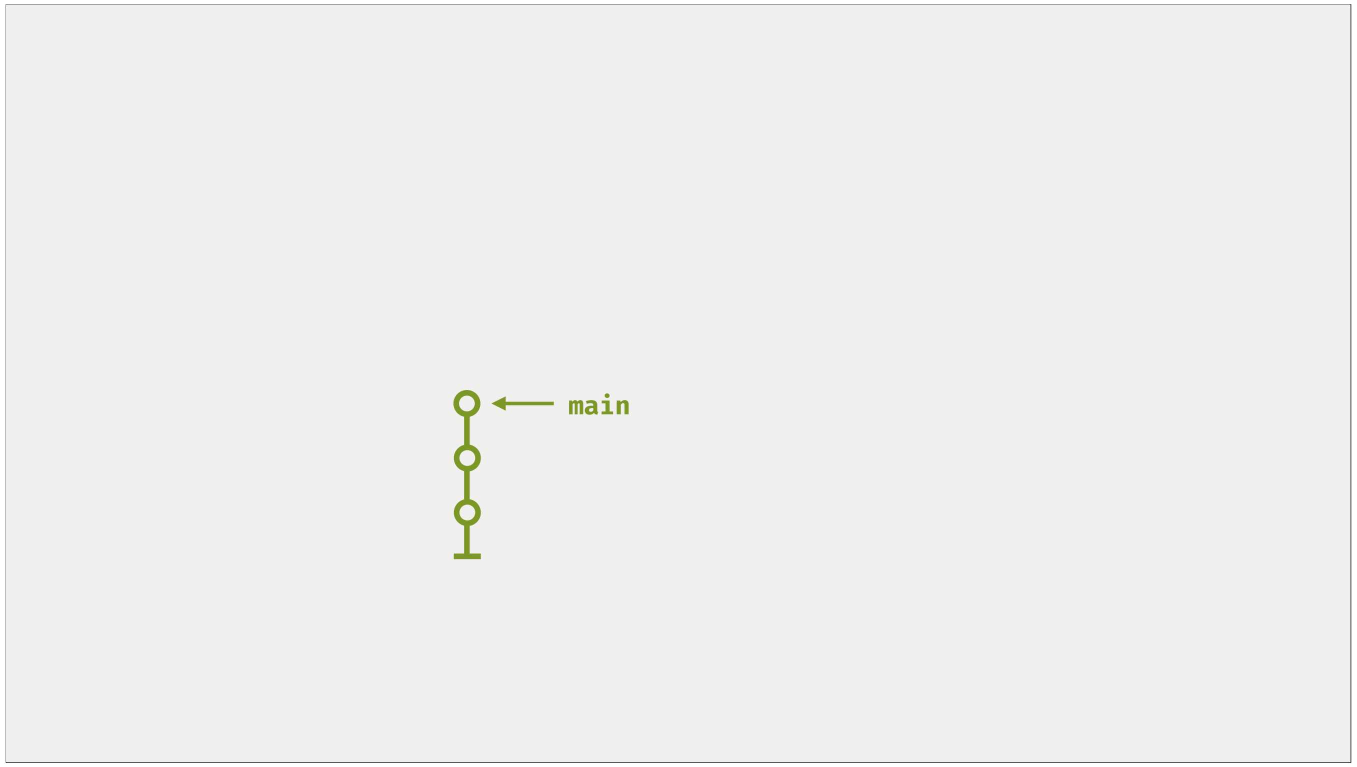


Let’s create a commit. This commit needs to have some kind of name.

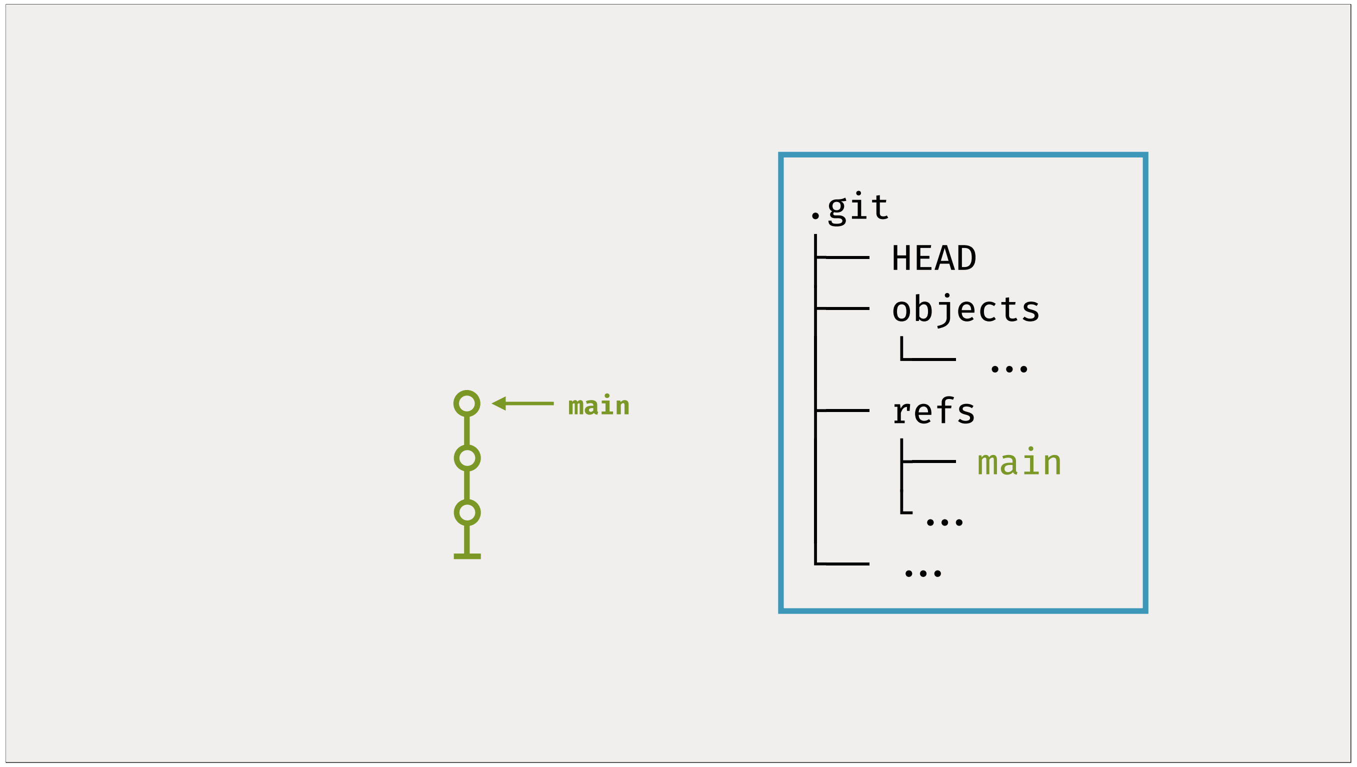


So, we'll give it a name. Or rather, git will. This is called a "branch", and the default branch name is "main". Branches are how we refer to commits in a repository. This is just one commit, so let's add a few more.


main



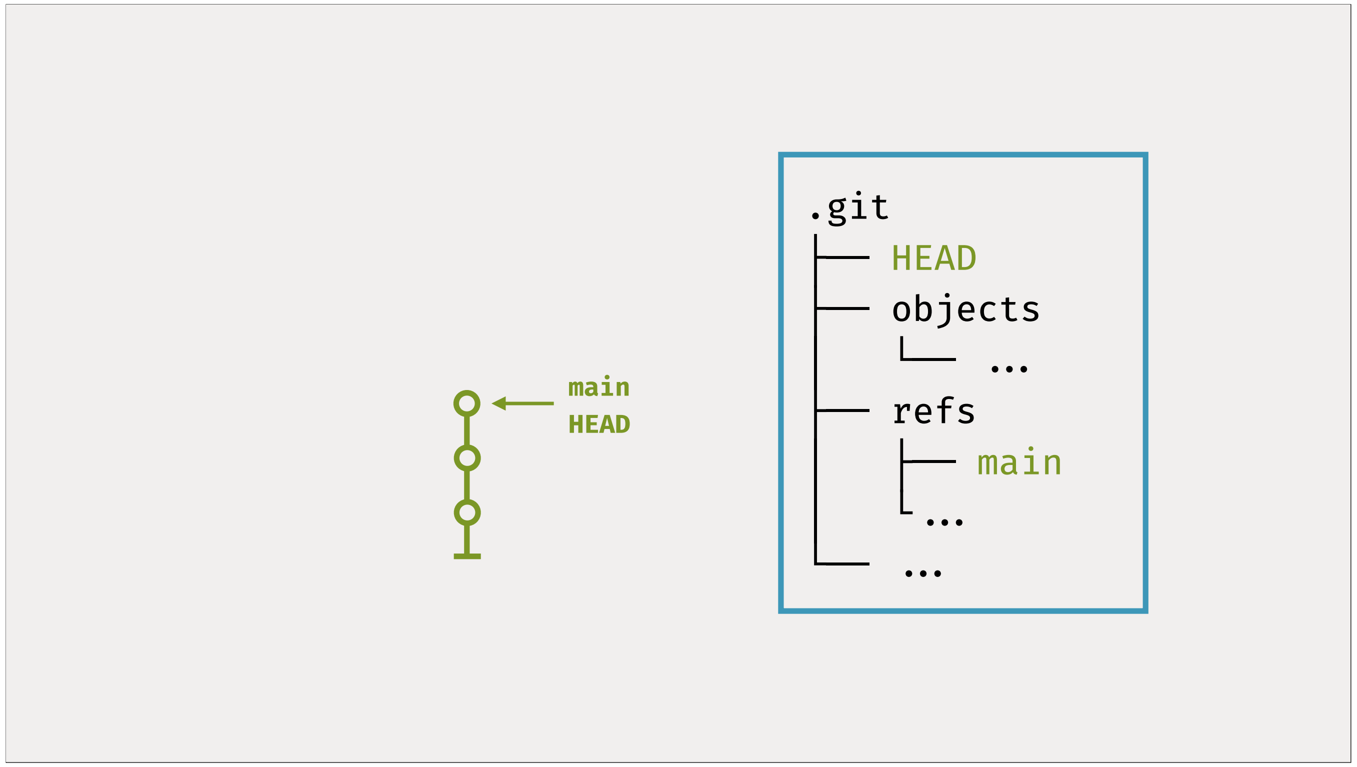
Now we’ve got a history of commits. Each of these is a snapshot of the project at that time.
If you do nothing else but make commits as you change files, this is the simple linear history you’ll end up with.



```
.git
├── HEAD
├── objects
│   └──  ...
├── refs
│   ├── main
│   └ ...
└── ...
```

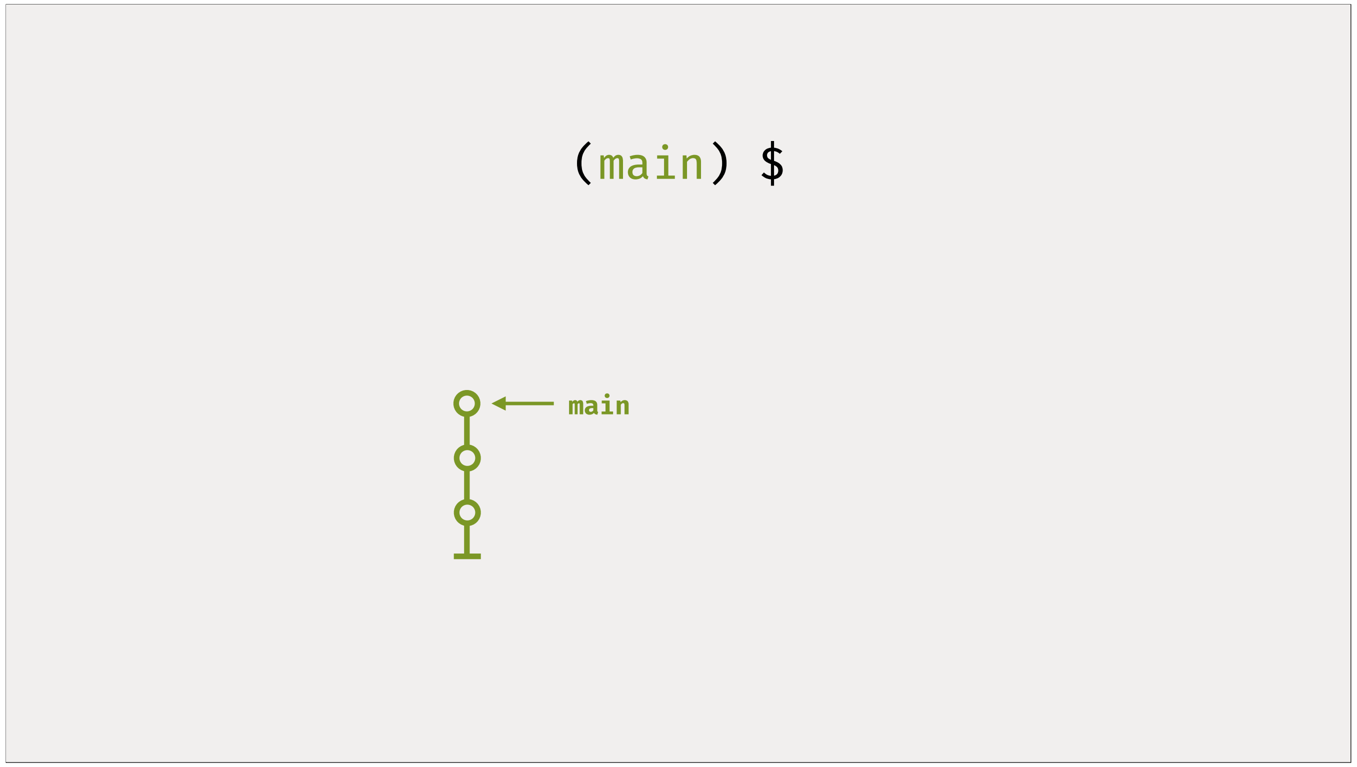
You might have noticed the “main” branch was moving along with each new commit. That’s by design. Think of a branch as a label pointing at a commit, and as you add new commits it follows along.

And that’s what it is. git branches are just simple labels. In the repository data directory, there’s a file called `main` that contains the SHA1 hash of the commit it’s pointing to.



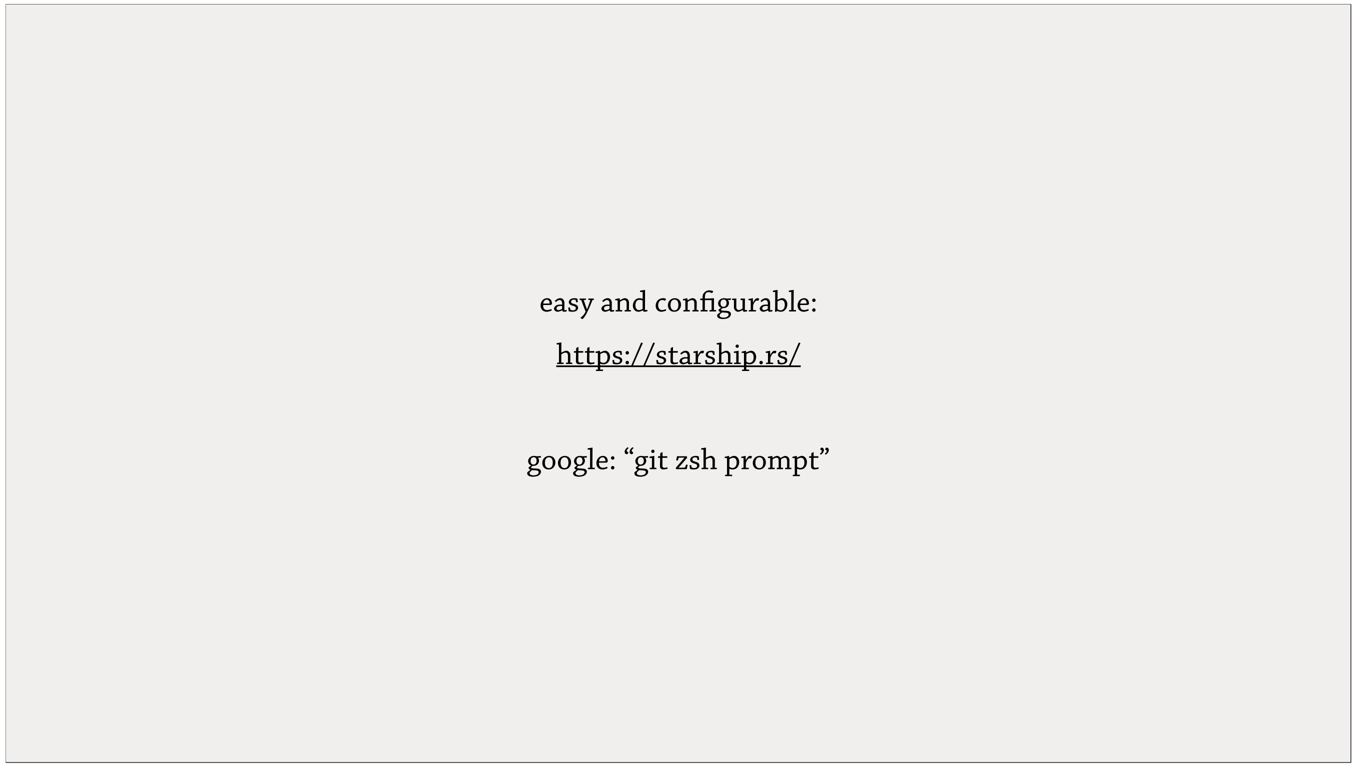
```
.git
├── HEAD
├── objects
│   └── ...
├── refs
│   ├── main
│   └...
└── ...
```

Internally, git also tracks the branch you're working on at any given time. This is called "HEAD". You never need to update this yourself, just know that this is how git knows which branch you're using.



```
(main) $
```

Git knows what branch you’re on, but it’s nice for you to know too. One way of doing that is to configure your shell prompt to show the current branch you’re on, which I’m showing here.

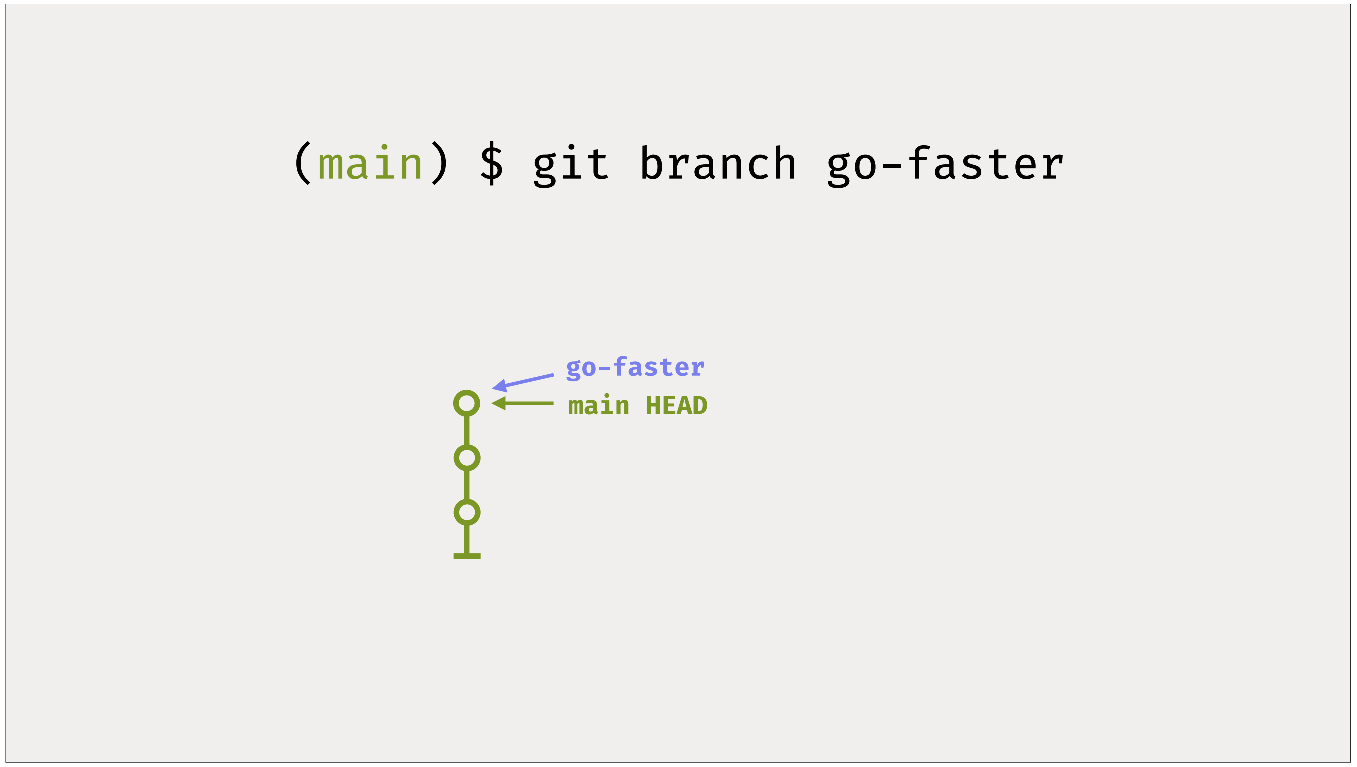


easy and configurable:

https://starship.rs/

google: “git zsh prompt”

A couple links for you to look for if you want to set that up. These will be in the slides emailed later.

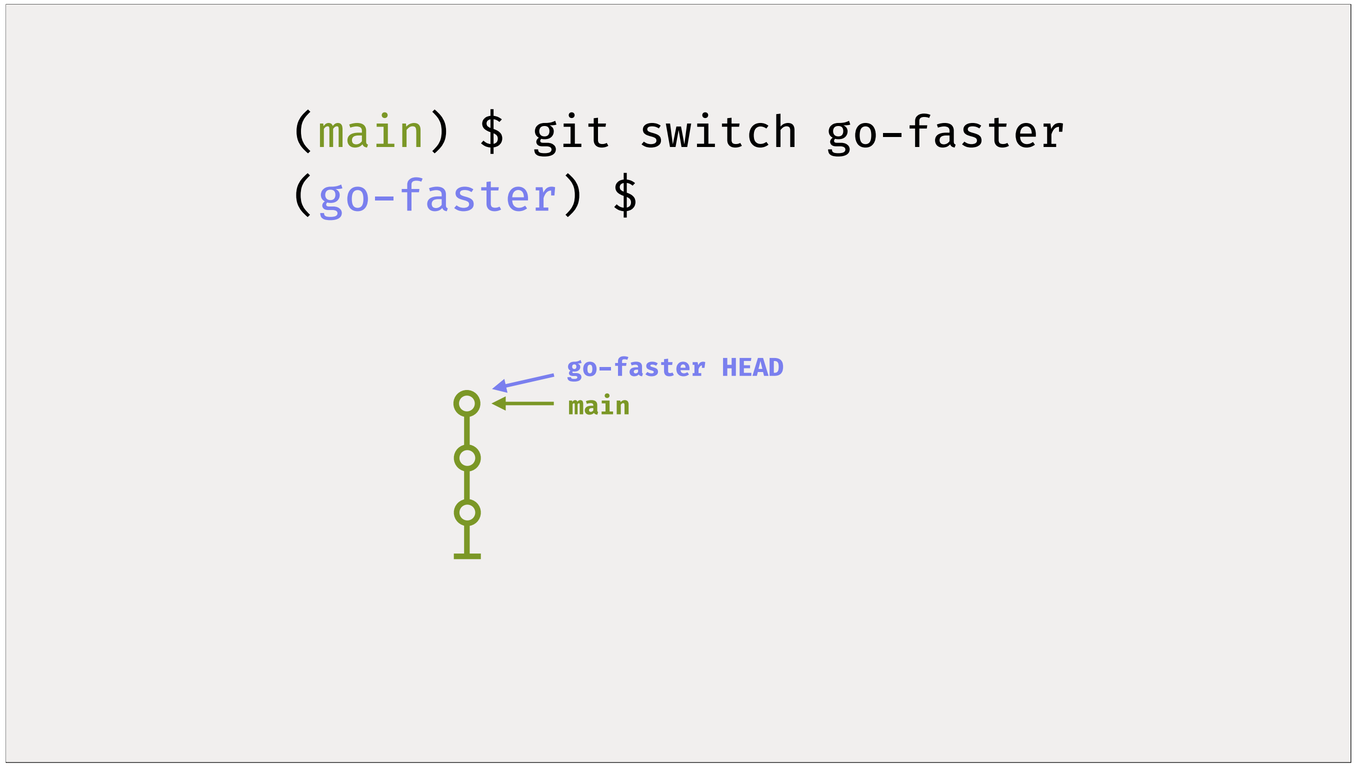


```
(main) $ git branch go-faster
```

And we can create new branches, simply creating a new label pointing at a commit.

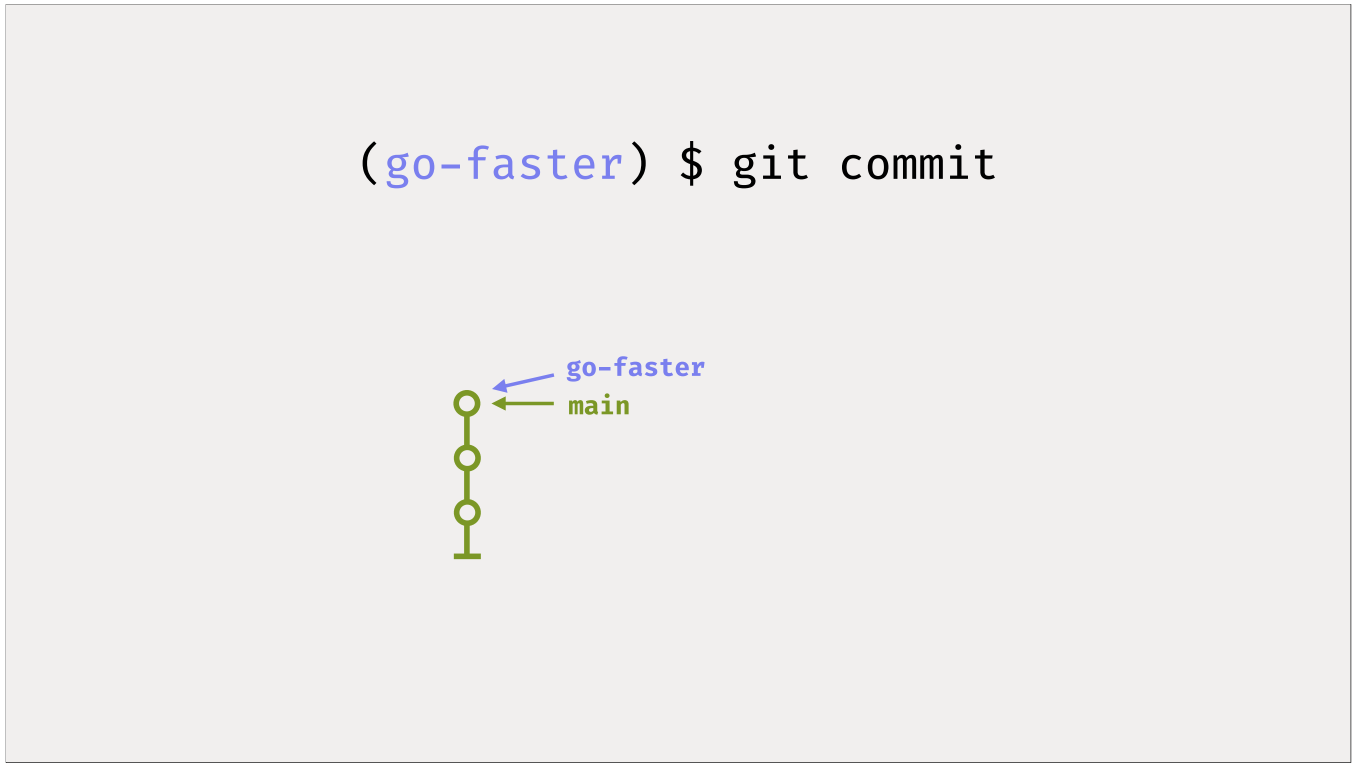
Let’s say you’re going to do some performance tuning. We create a new branch called “go-faster” with this command. It’s just another label, pointing to the same commit as the main branch.

Under the hood, git writes a new 40-byte file in the `.git` directory, and that’s it.



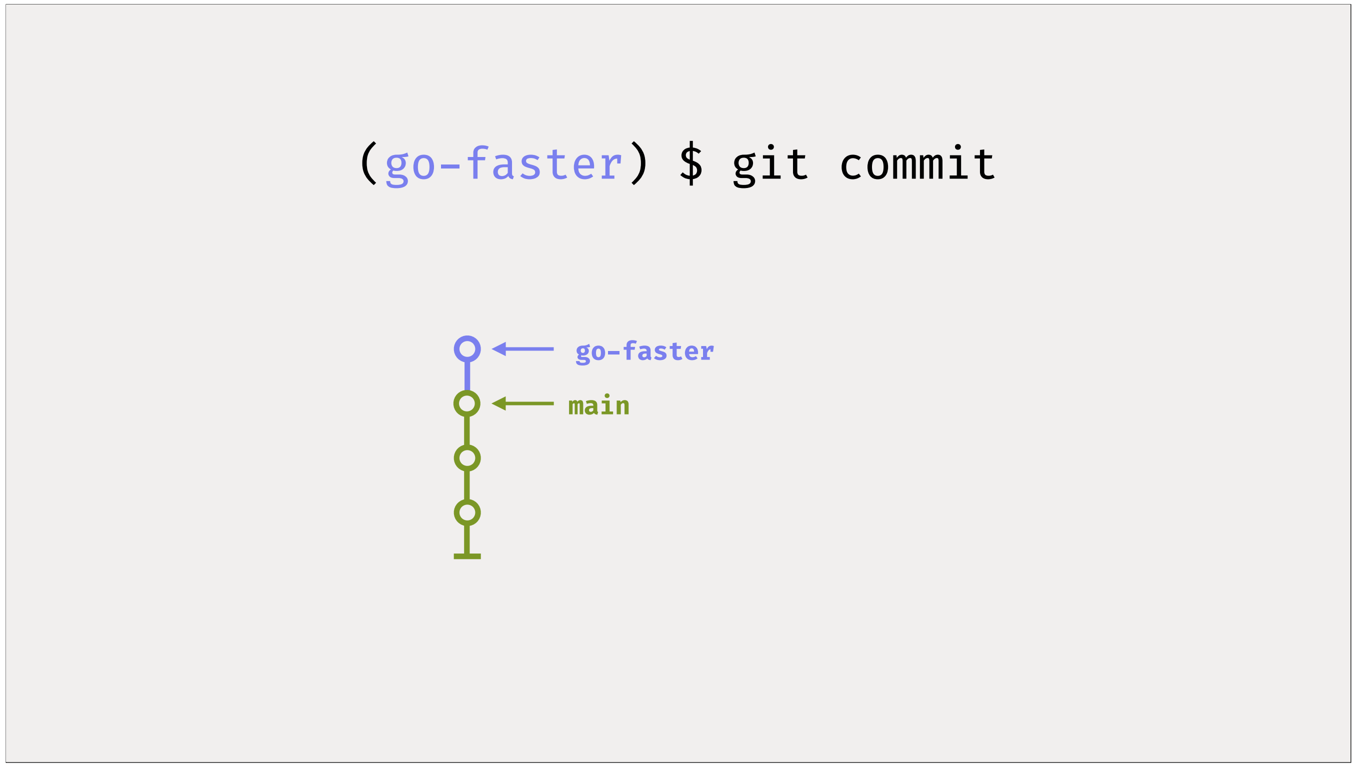
```
(main) $ git switch go-faster
(go-faster) $
```

We can make new commits on this branch. Let’s switch to it first, using the “switch” command.
Notice that HEAD changes, too.



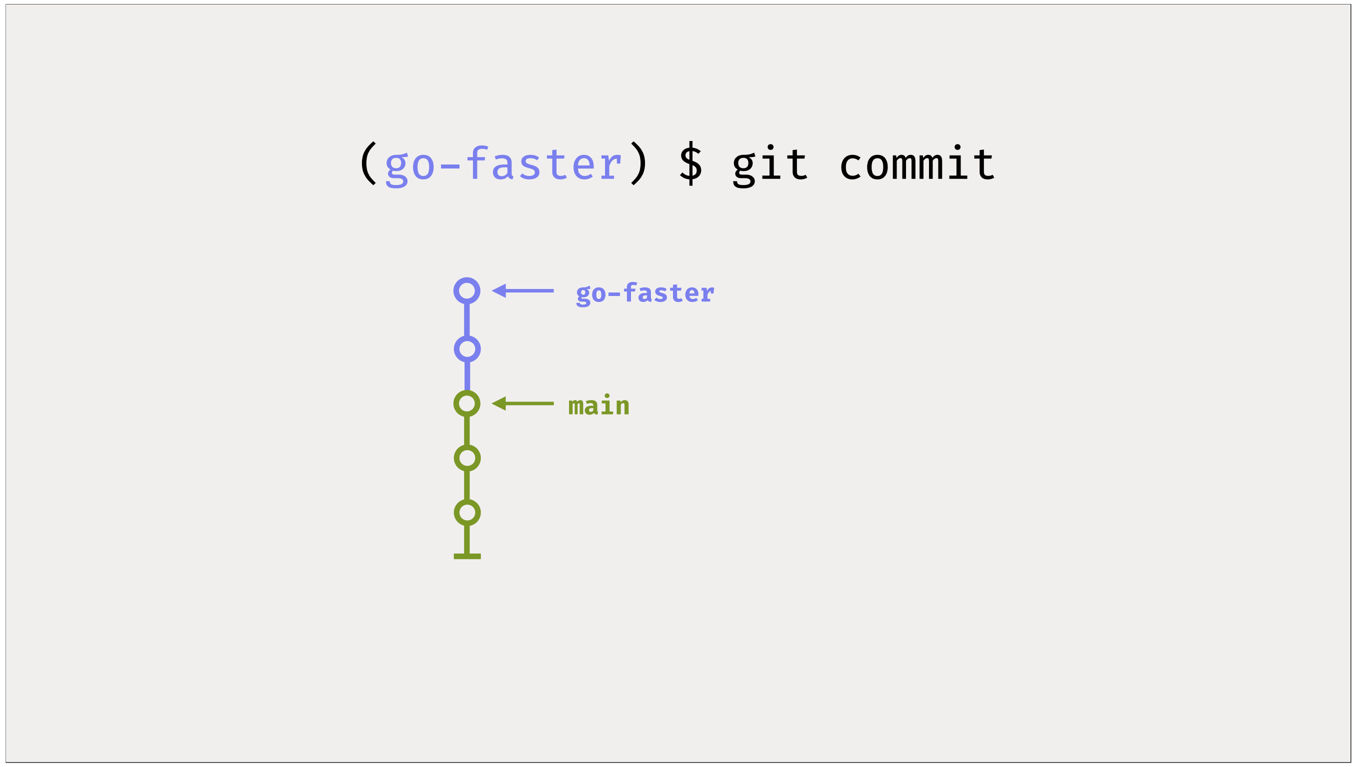
```
(go-faster) $ git commit
```

And now let’s say we made some changes, and we make a commit on this branch



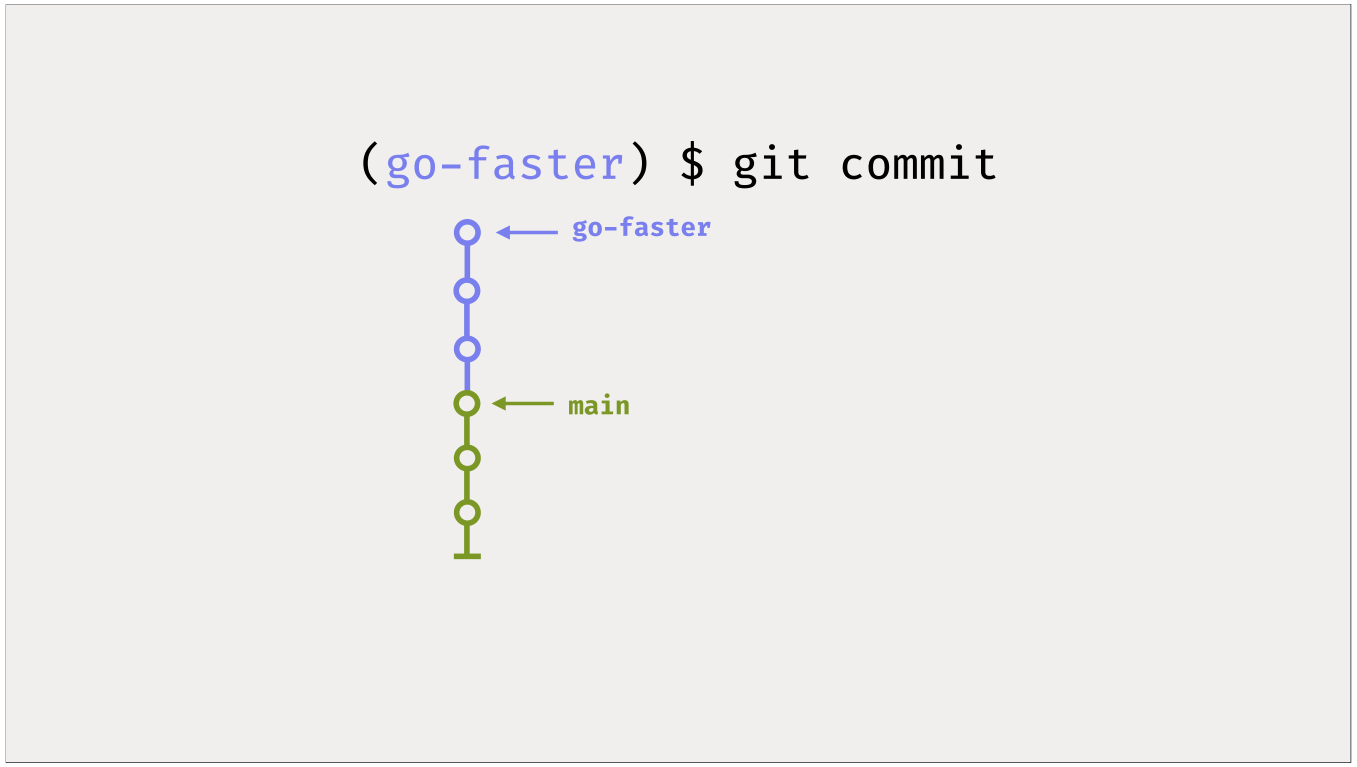
```
(go-faster) $ git commit
```

This creates a new commit and updates where `go-faster` is pointing, but the commit “main” is pointing at doesn’t change.



```
(go-faster) $ git commit
```

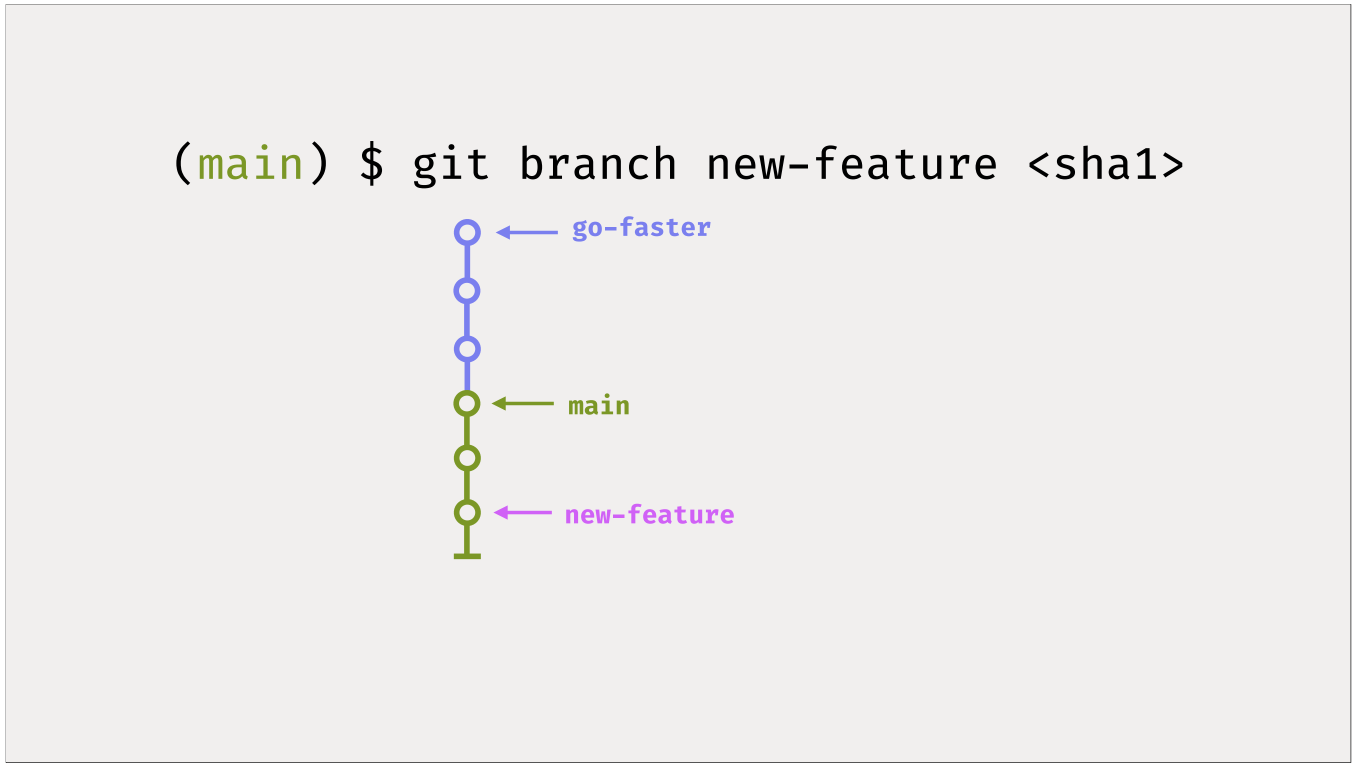
We can make more commits.



```
(go-faster) $ git commit
```

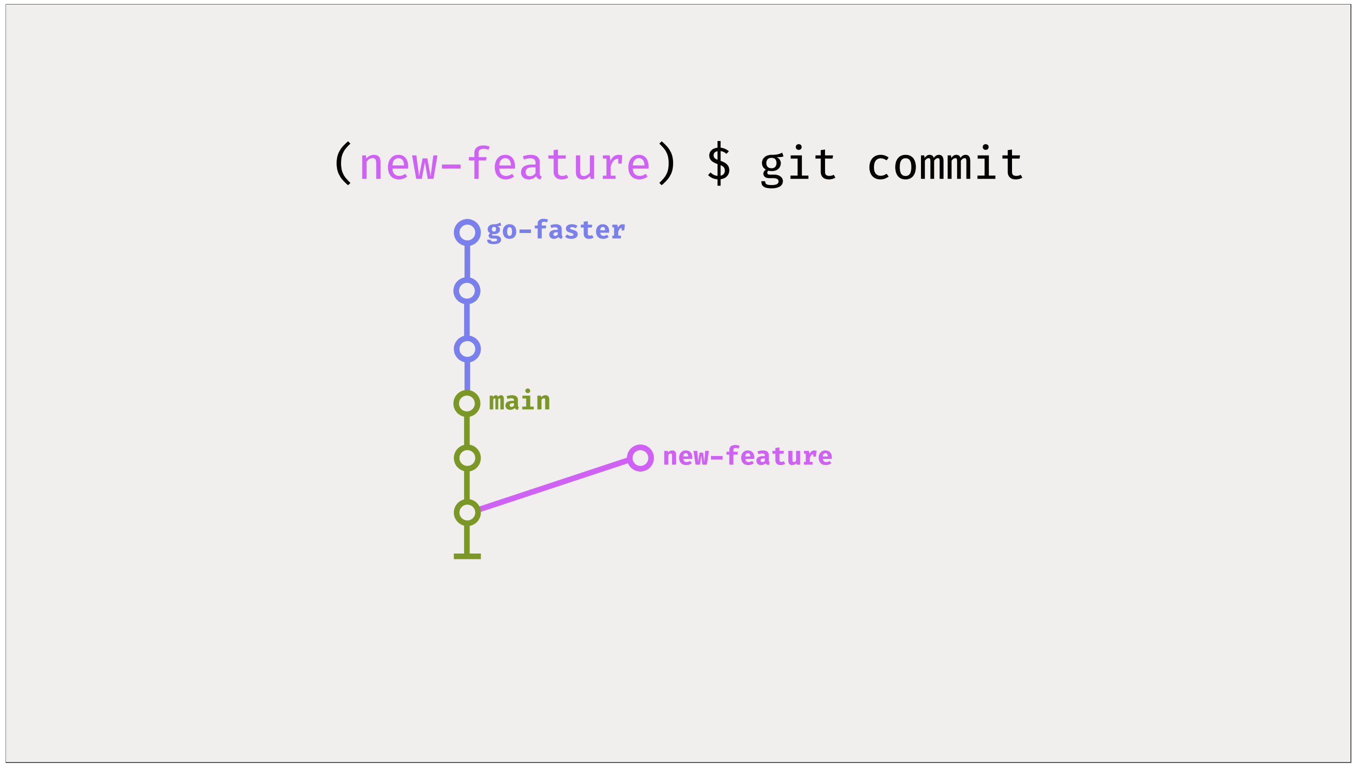
Ok, now we have both main and this new “go-faster” branch.

Branches don’t all have to be created from the same place. We can point them anywhere in history:



```
(main) $ git branch new-feature <sha1>
```

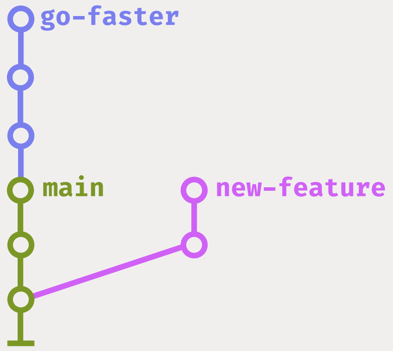
Now we’ve created another branch, “new-feature”, pointing at an older location. Maybe you started work on that new feature awhile back, created the branch, but didn’t get around to doing anything beyond that.



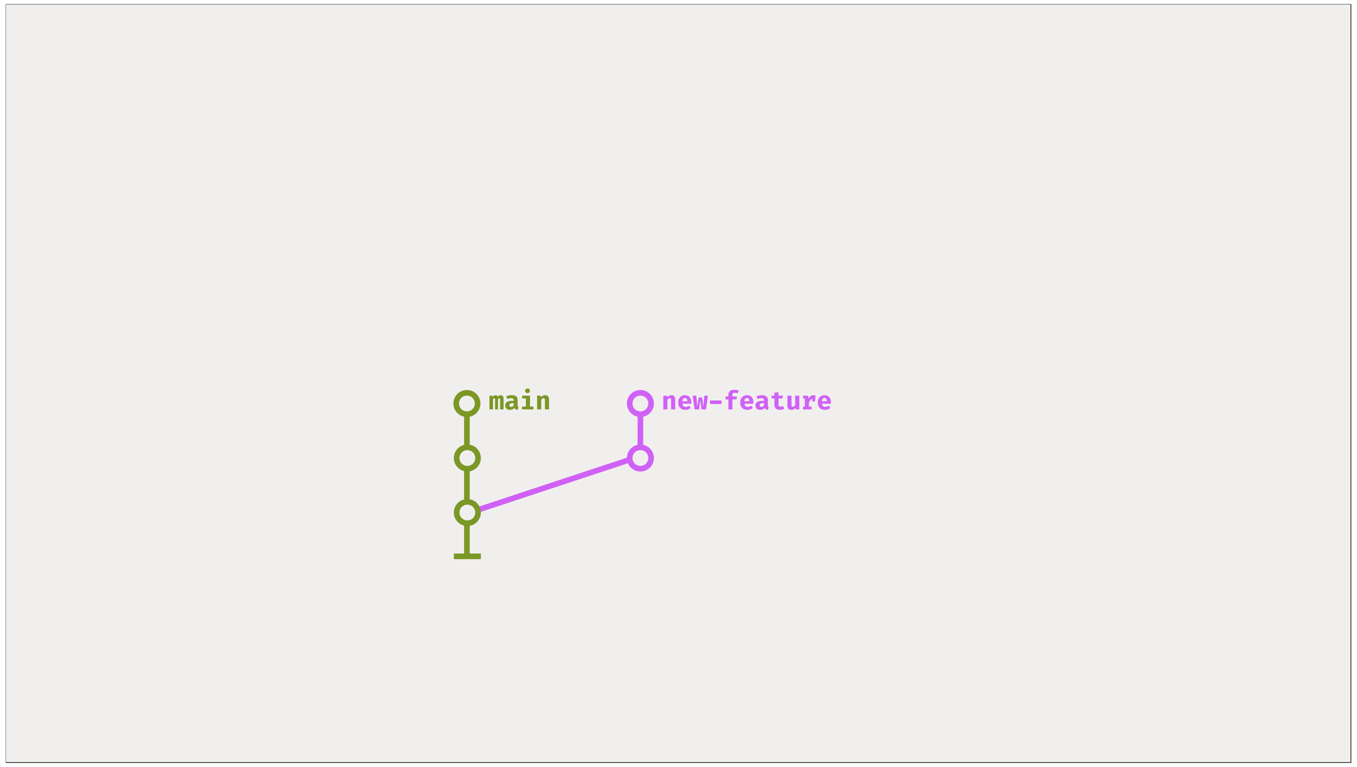
```
(new-feature) $ git commit
```

Let’s create a commit on that branch too.

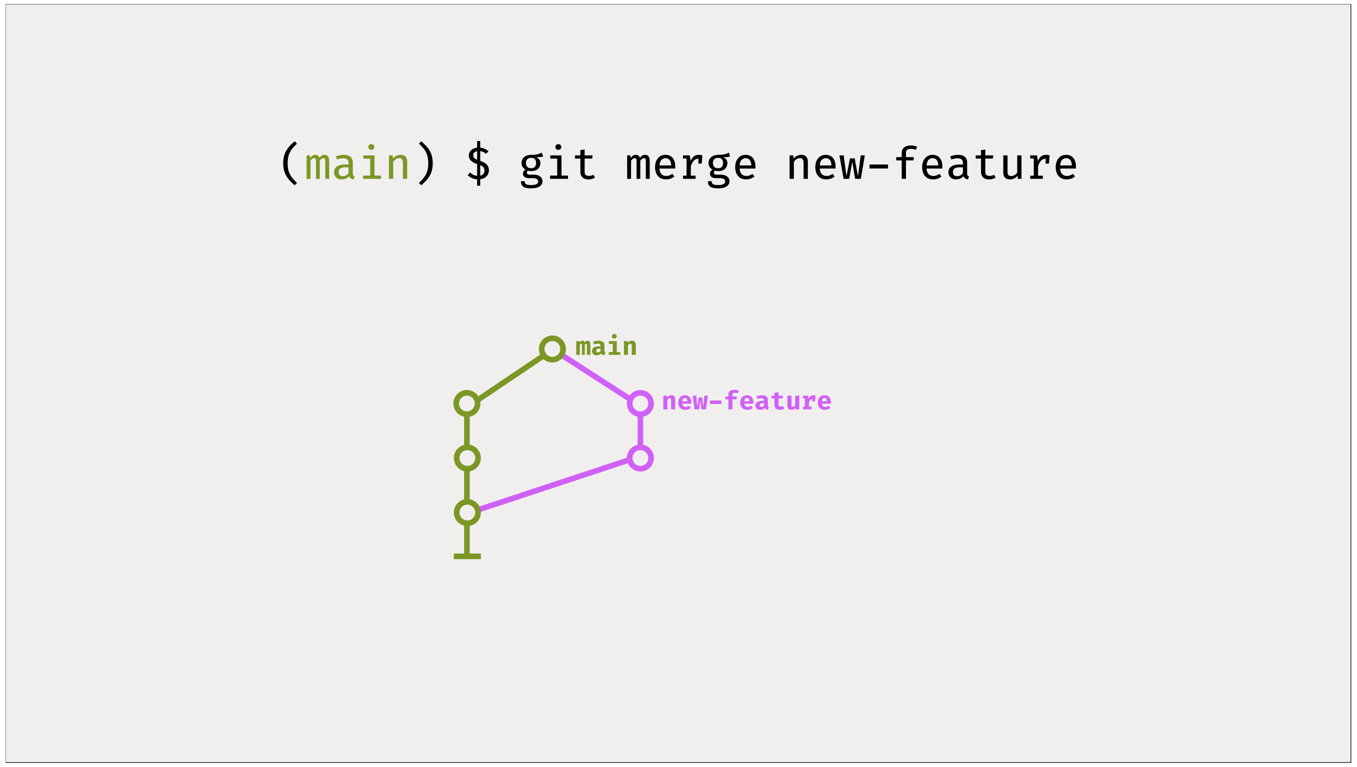
```
(new-feature) $ git commit
```



And another.



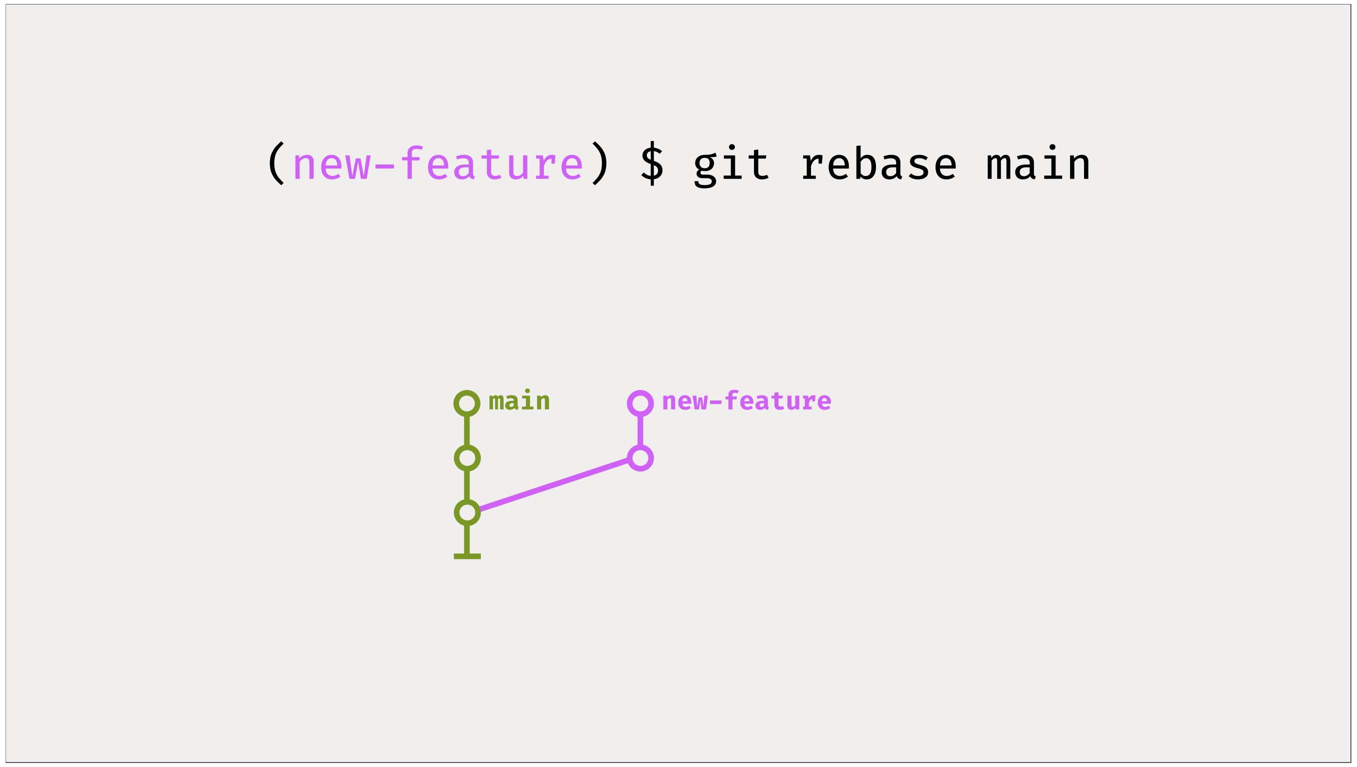
Now let’s say you’re done with the new feature, and you want to combine the work you did there with the work on main, and make that the new main. With git, you create what’s called a “merge” commit.



```
(main) $ git merge new-feature
```

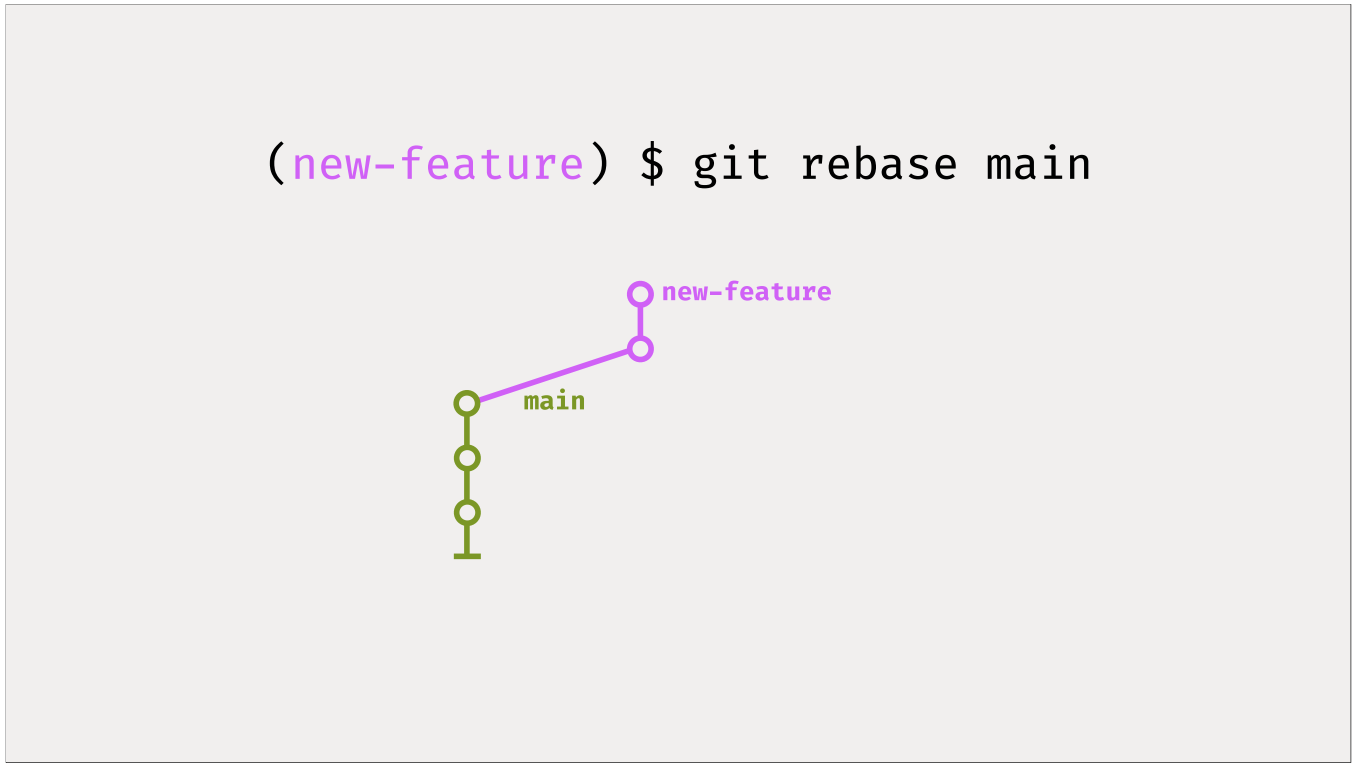
A merge commit combines the two histories together. From the main branch, we say “merge the new-feature branch”. This combines the two histories together, and changes `main` to point at the new merged history.

When I said earlier that commits point at their parents, I didn’t mention that a commit can have any number of parents. The first commit in a repository has no parents. Most commits have one parent, and this new merge commit has two parents.



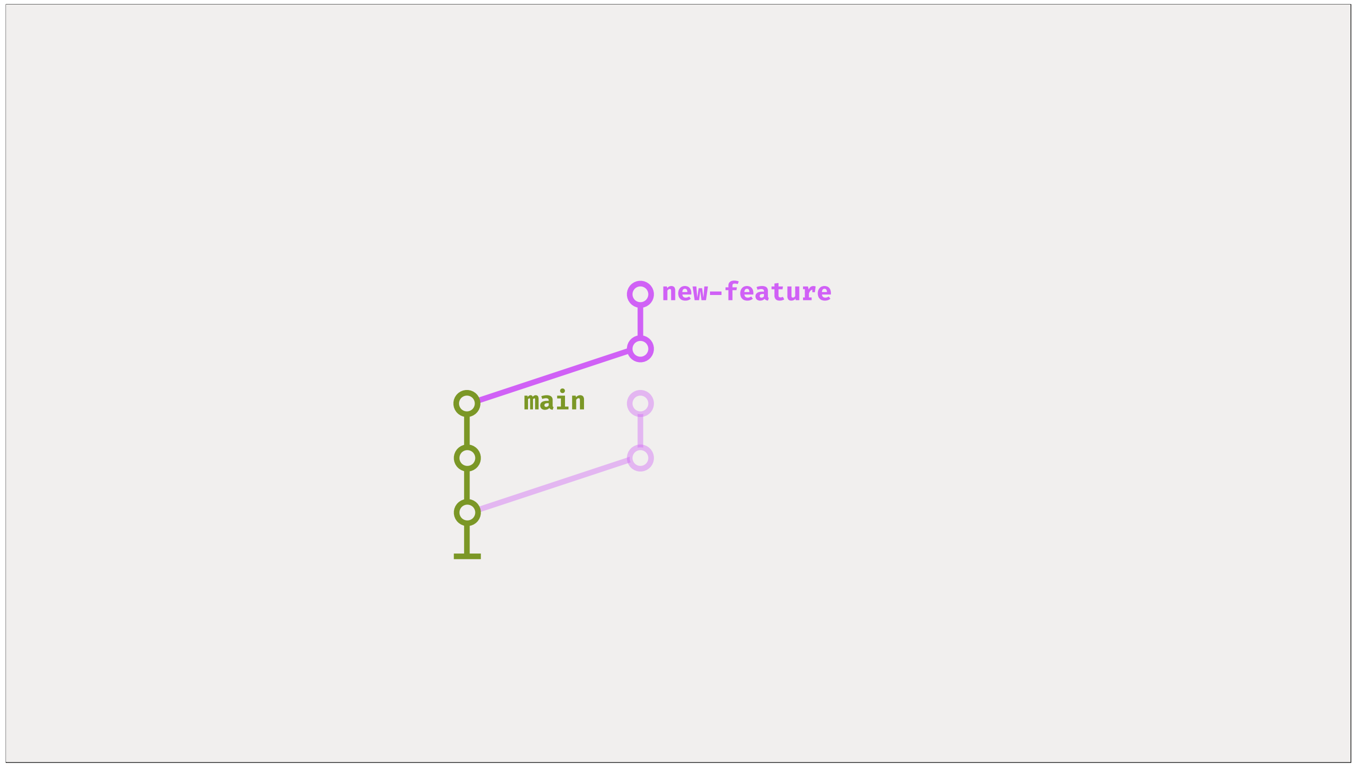
```
(new-feature) $ git rebase main
```

There’s more you can do with git histories, too. History isn’t *exactly* fixed, you can change and rearrange things.



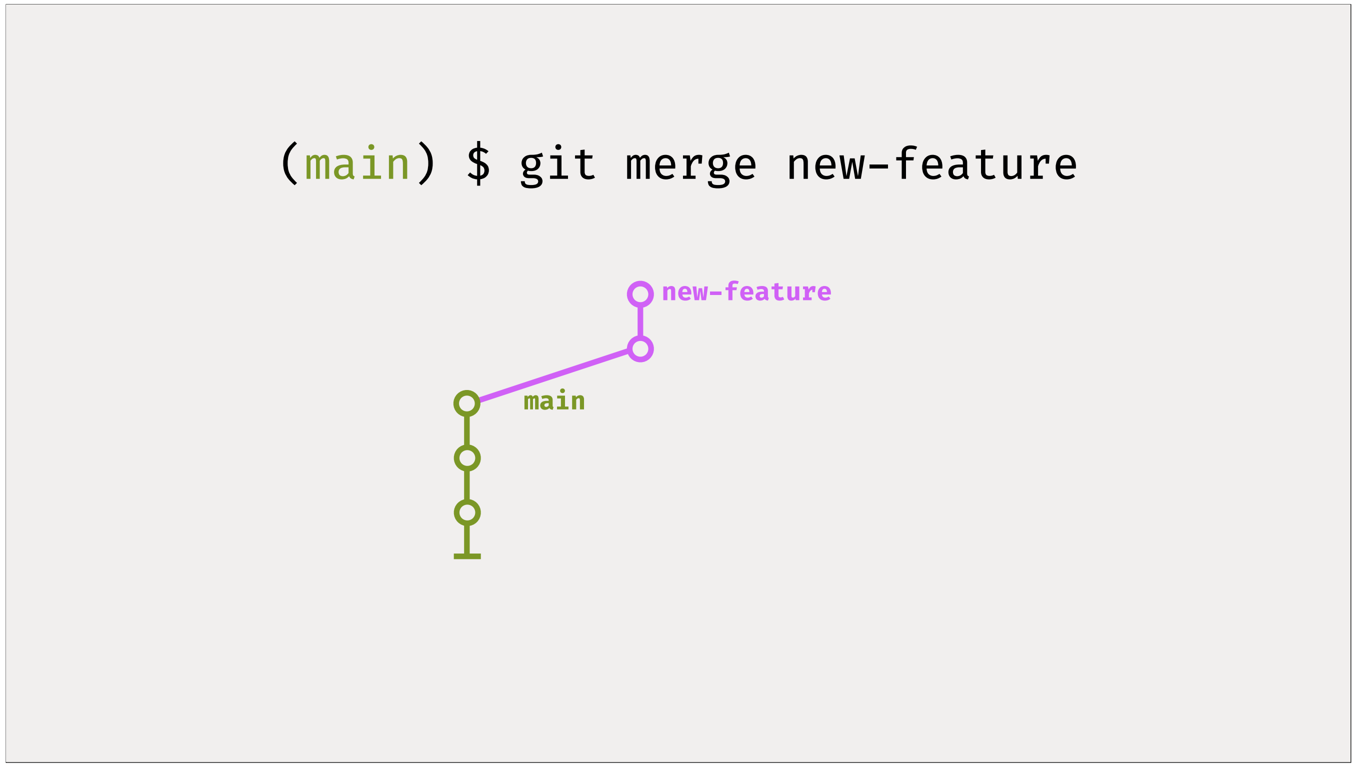
```
(new-feature) $ git rebase main
```

A git operation called “rebase” lets you move the “base” of a branch. Here, we’ve moved the base of the new-feature branch to be on top of the latest “main”.



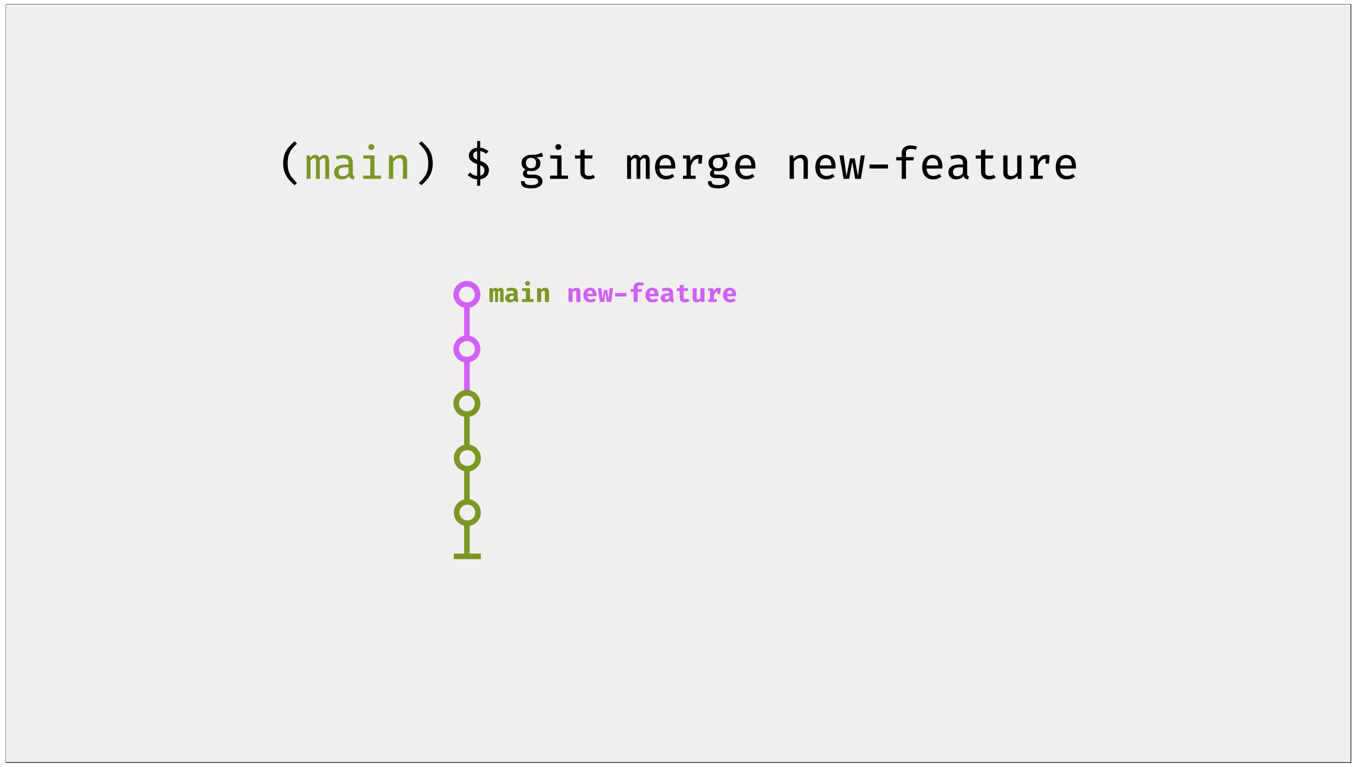
I said history is almost fixed, but remember that commit hashes are calculated with the contents of a commit. That content includes a commit’s parent, so the commits on this rebased branch have different hashes from what they had before. The changes themselves may not be different, and the timestamps and commits message are the same, but even if just the parent changed, the contents have changed, and now have new commit hashes.

This also means the original version of the branch, and those commits, still exist in git, at least internally. It’ll get cleaned up eventually, but that data and those commits are still there, and you can actually get them back if you need to.



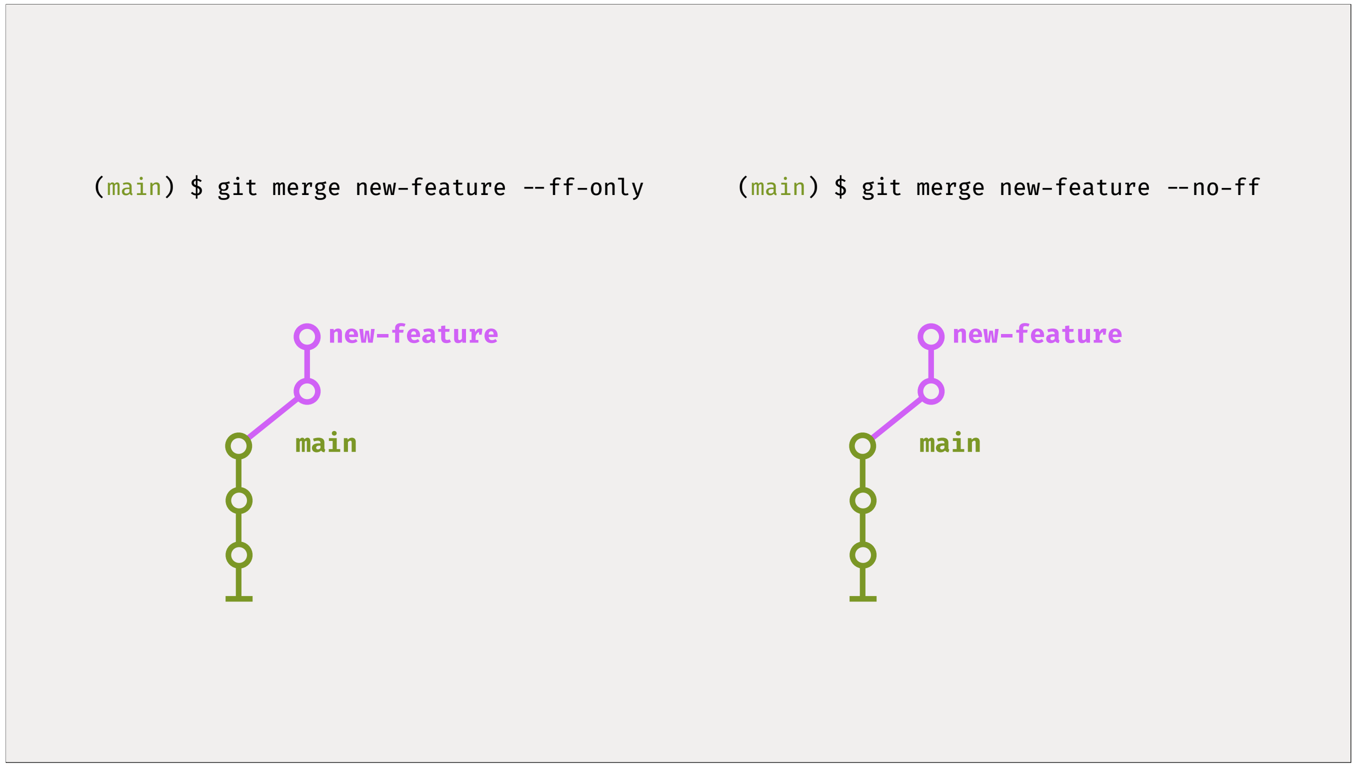
```
(main) $ git merge new-feature
```

Now you’ve rebased this new feature onto main, and you want to merge your changes in. Because there’s no divergence between main and new feature, the histories haven’t split, you can simply move the main label forward.



```
(main) $ git merge new-feature
```

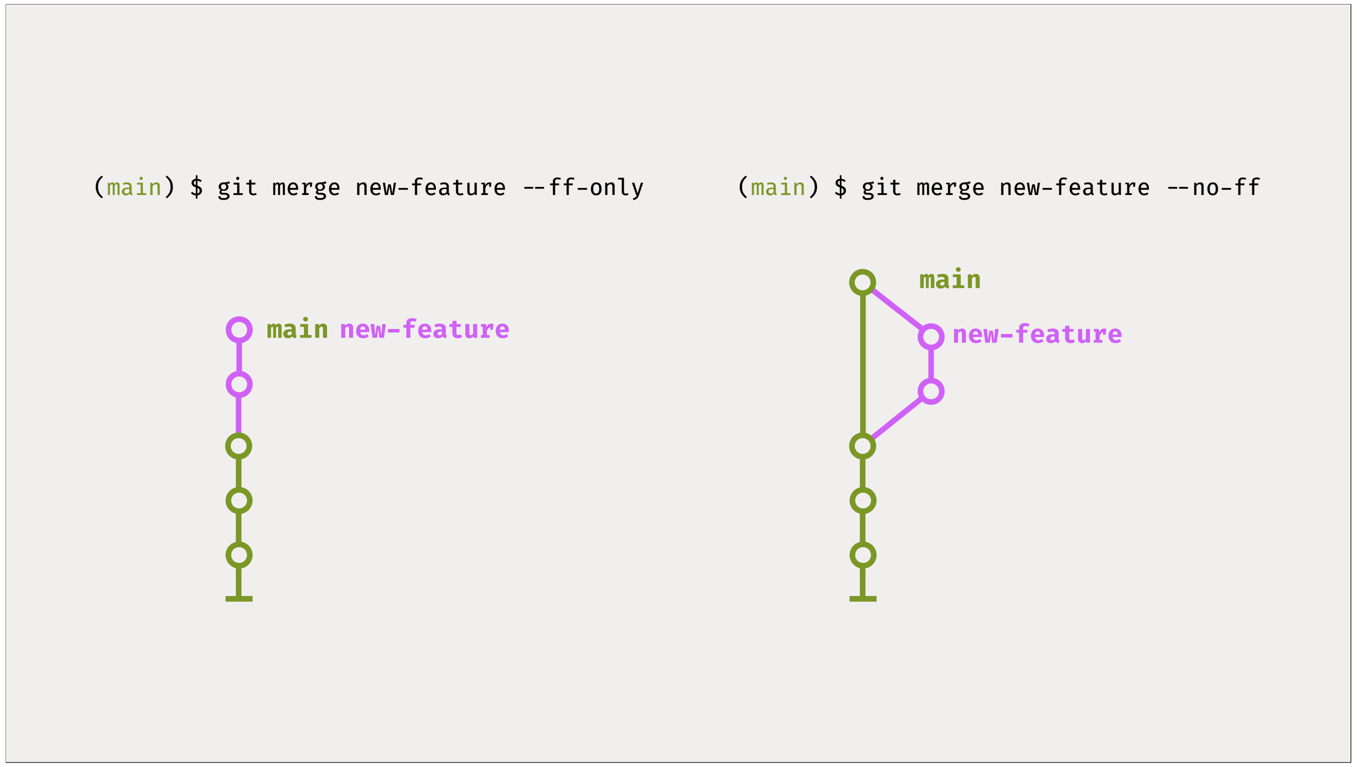
From the main branch, we’ll merge the new-feature branch. Now main is pointing at the same commit as new-feature. This is called a “fast-forward merge”.



```
(main) $ git merge new-feature --ff-only
```

```
(main) $ git merge new-feature --no-ff
```

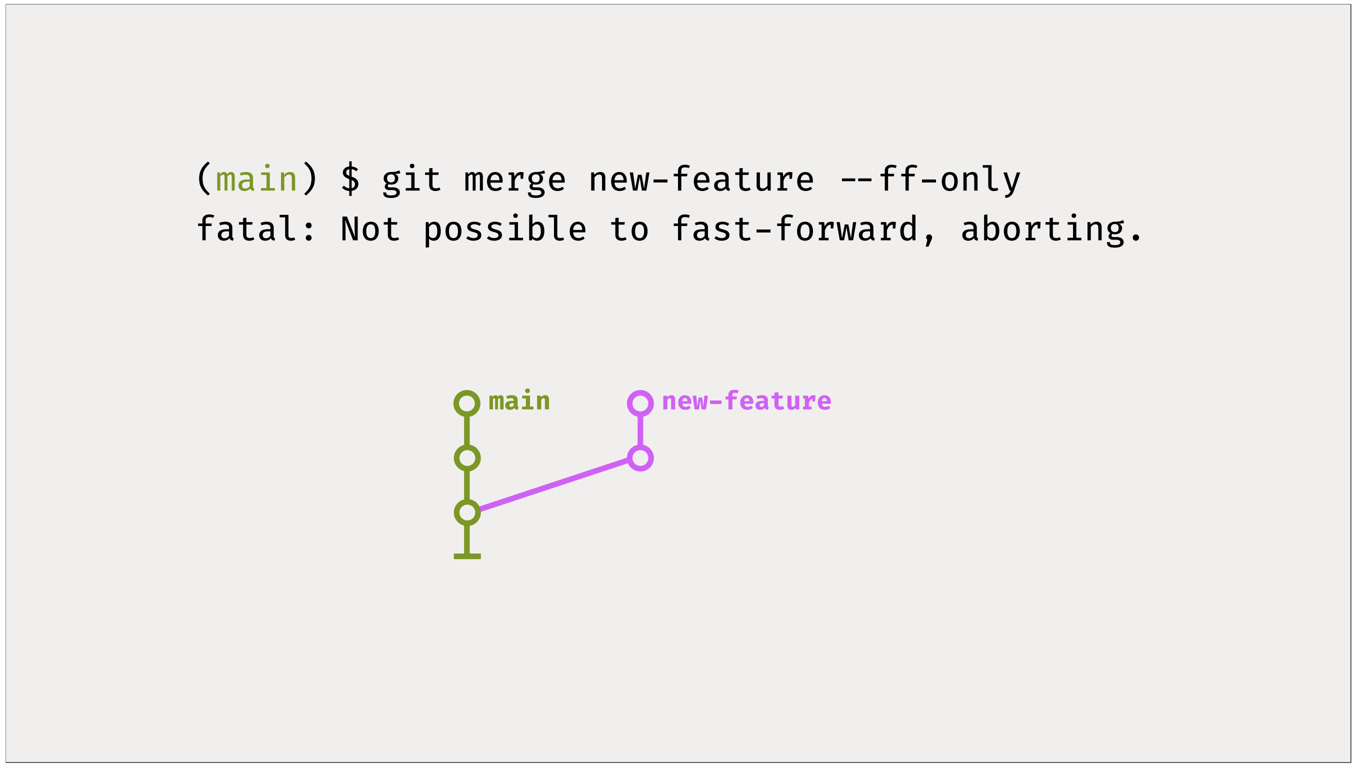
To make this more obvious, let’s do this merge two different ways.
First, fast-forward. We’re going to explicitly ask git to “fast forward” the main branch to match `new-feature`.
Secondly, we’ll ask for _no_ fast forward, that is, we explicitly want to make a merge commit even if we could otherwise fast forward.



```
(main) $ git merge new-feature --ff-only
```

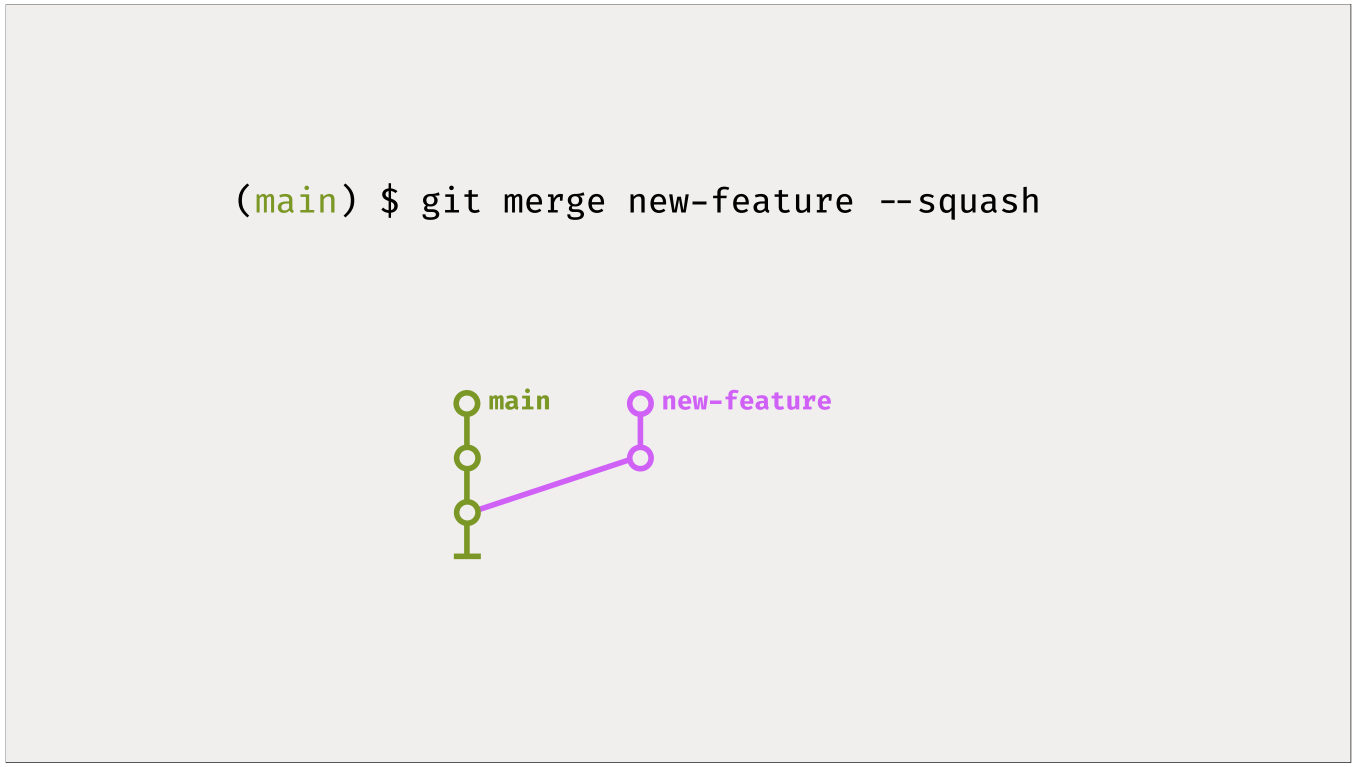
```
(main) $ git merge new-feature --no-ff
```

And that looks like this. On the left, we’ve simply moved the main branch forward. On the right, we’ve created a merge commit combining the two histories, even if there wasn’t any change on the “main” side.



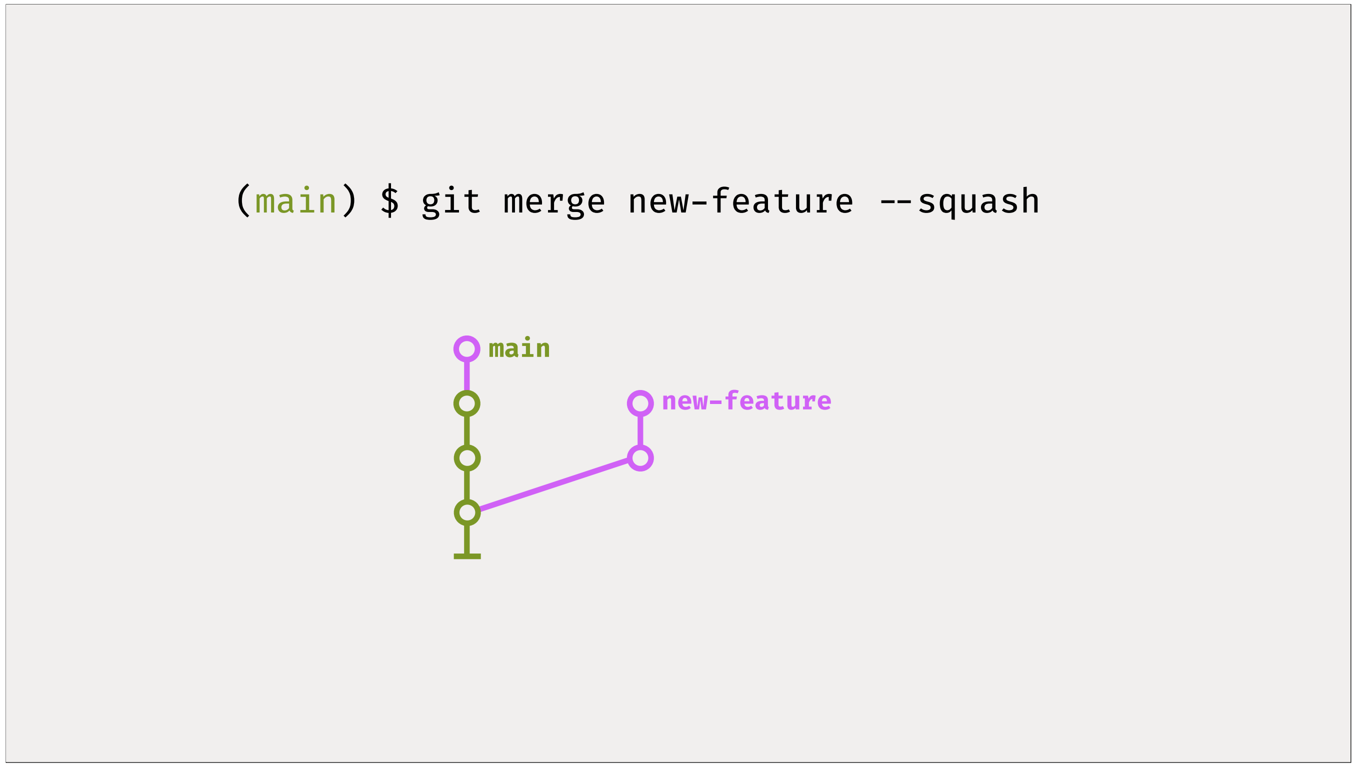
```
(main) $ git merge new-feature --ff-only
fatal: Not possible to fast-forward, aborting.
```

We can only fast-forward if the histories haven’t diverged. If we try to, git won’t do it.



```
(main) $ git merge new-feature --squash
```

And of course there are even more ways to do things with git. For example, maybe you don’t want to see all the separate commits on a feature branch, you just care about getting the combined changes into main. You can do this with what’s called a “squash” merge

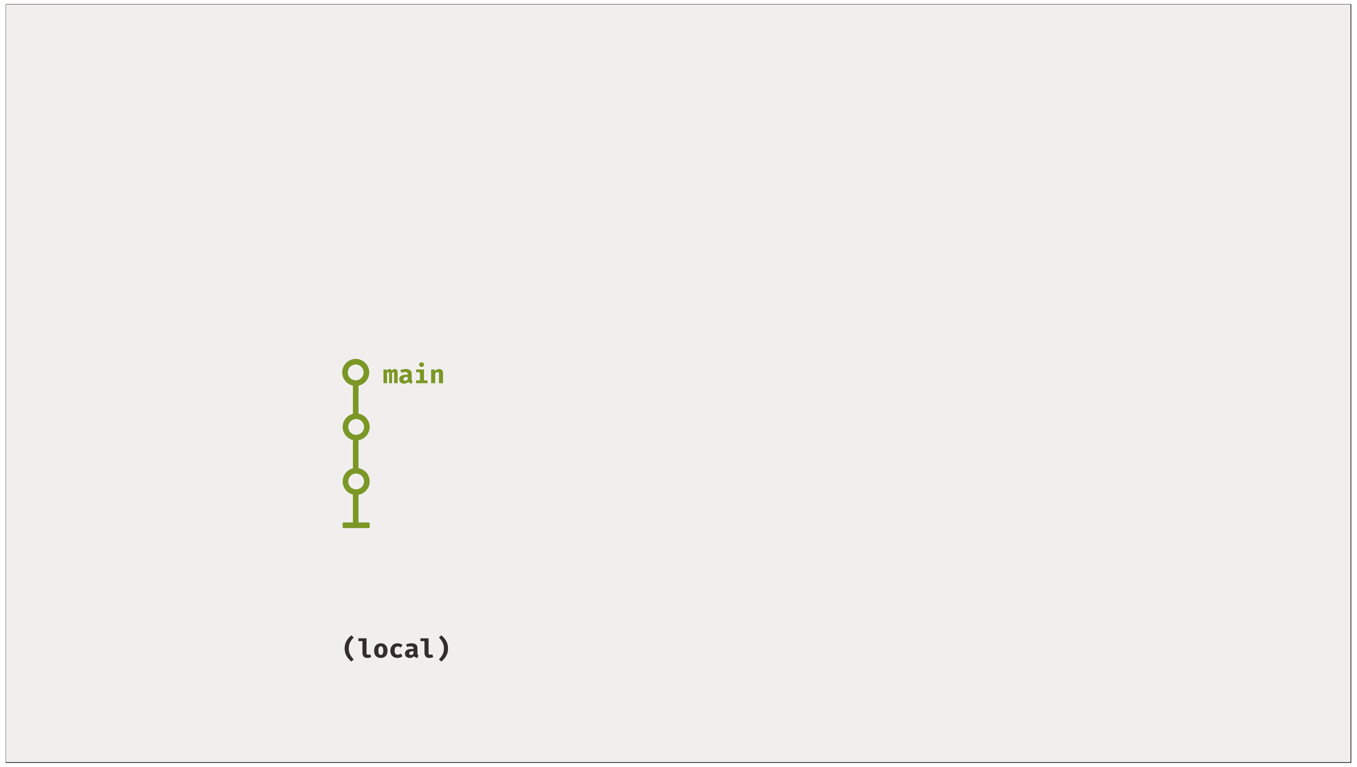


```
(main) $ git merge new-feature --squash
```

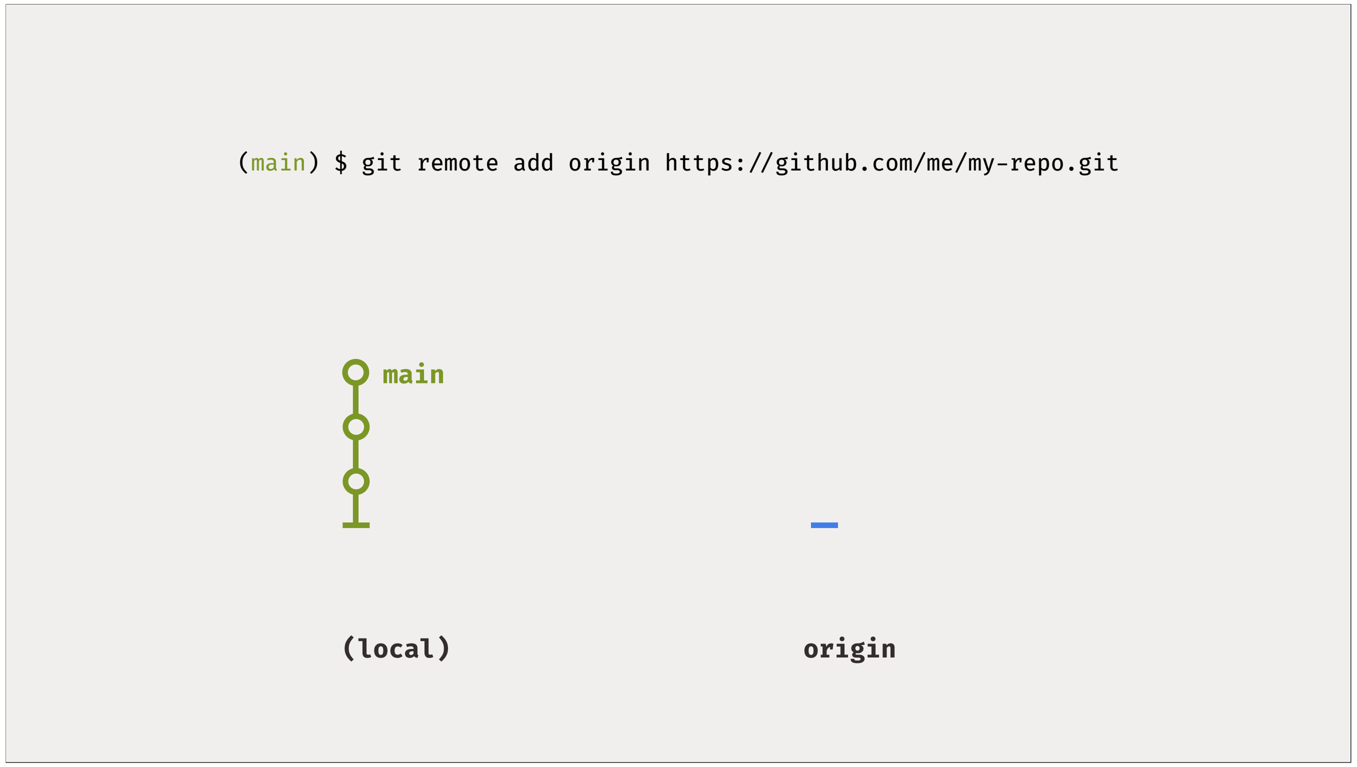
This takes all the changes from the `new-feature` branch and squashes them together into a single change that you can commit. The branch is still there, but now all its changes have been added to the main branch together as a single commit.

Remotes

So far all everything I’ve shown you involves the repository sitting on your computer next to your code. But you probably want to store a copy somewhere else, like, say, GitHub. You can keep that copy to yourself or share it and work with others, but we have to get it there first.
In git, other copies of a repository are called “remotes”.

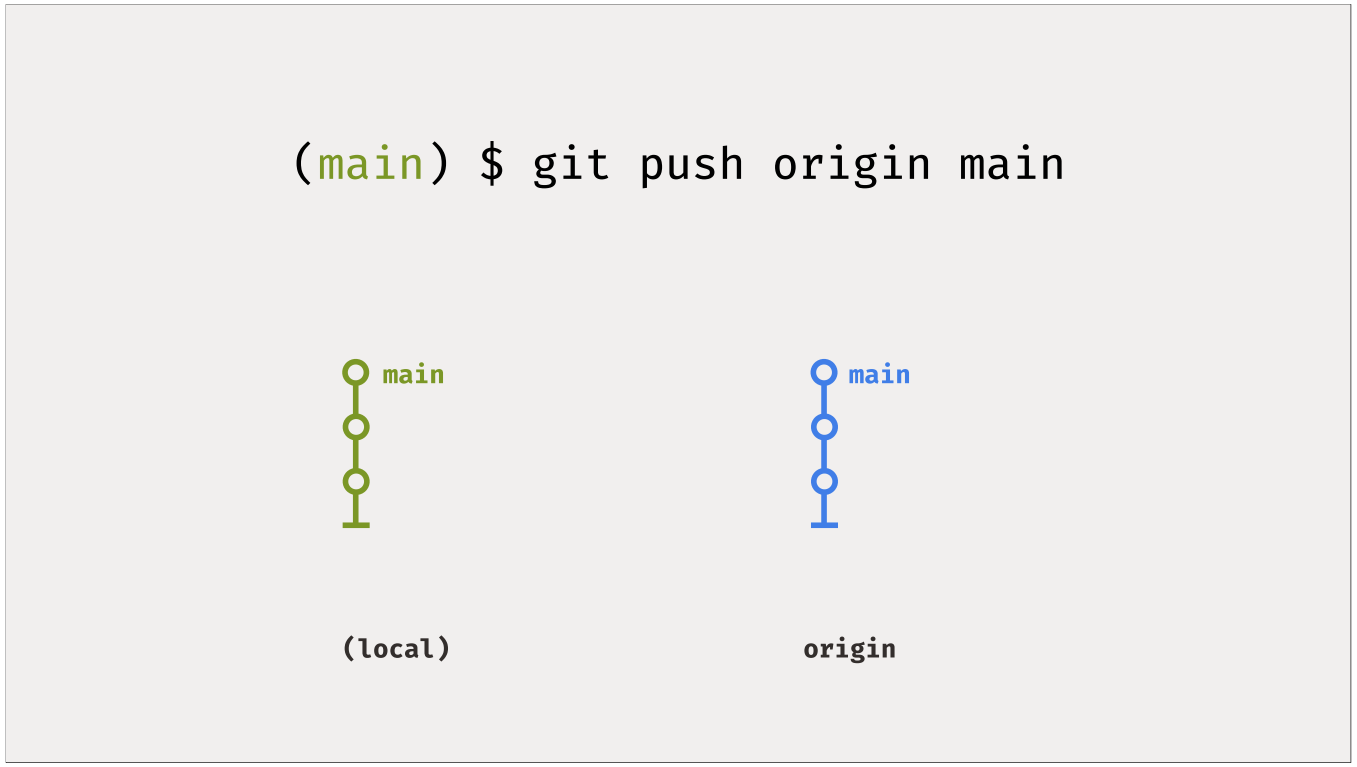


So let’s start with your repository. You’ve got a few commits, and you want to push them somewhere.



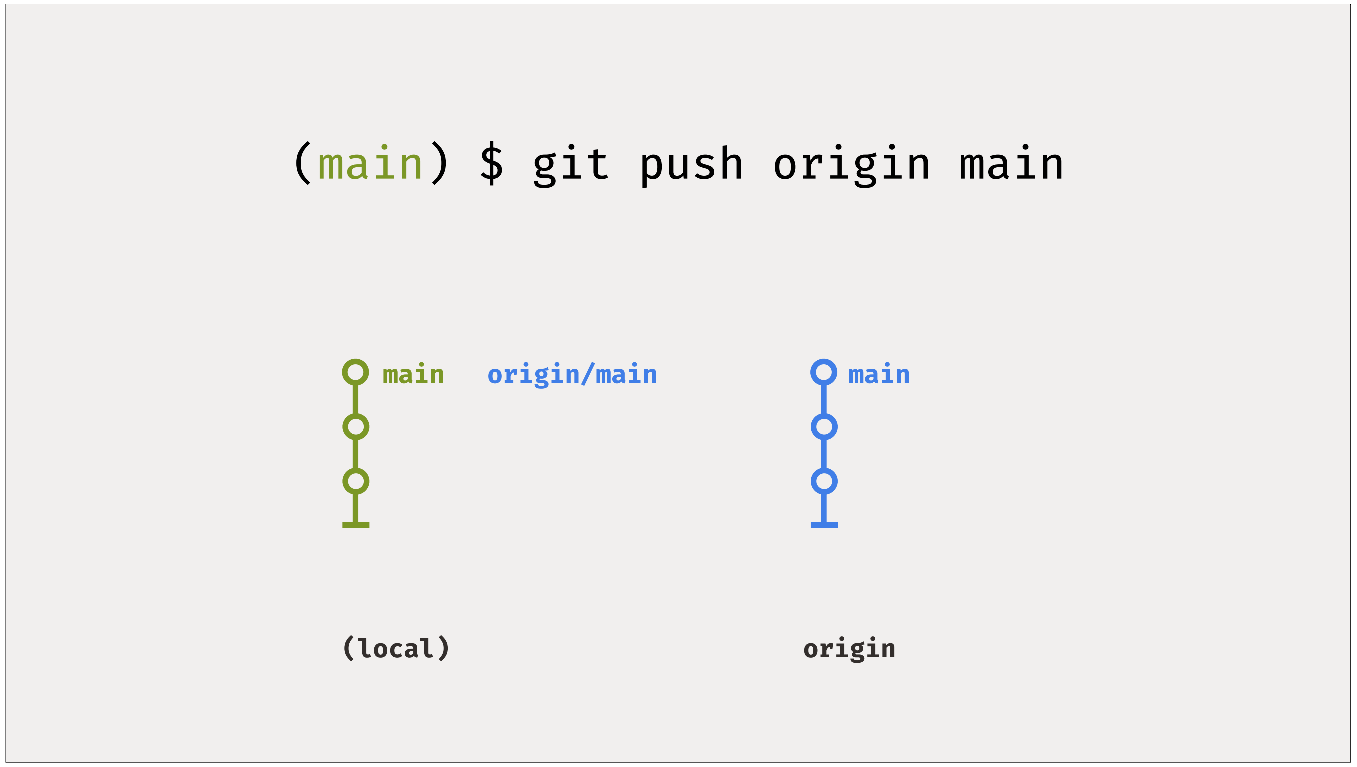
```
(main) $ git remote add origin https://github.com/me/my-repo.git
```

And let’s say you’ve created a new, empty repository on GitHub. You can tell git about it by adding a new remote. The default name is “origin”, so that’s what we’ll call it.



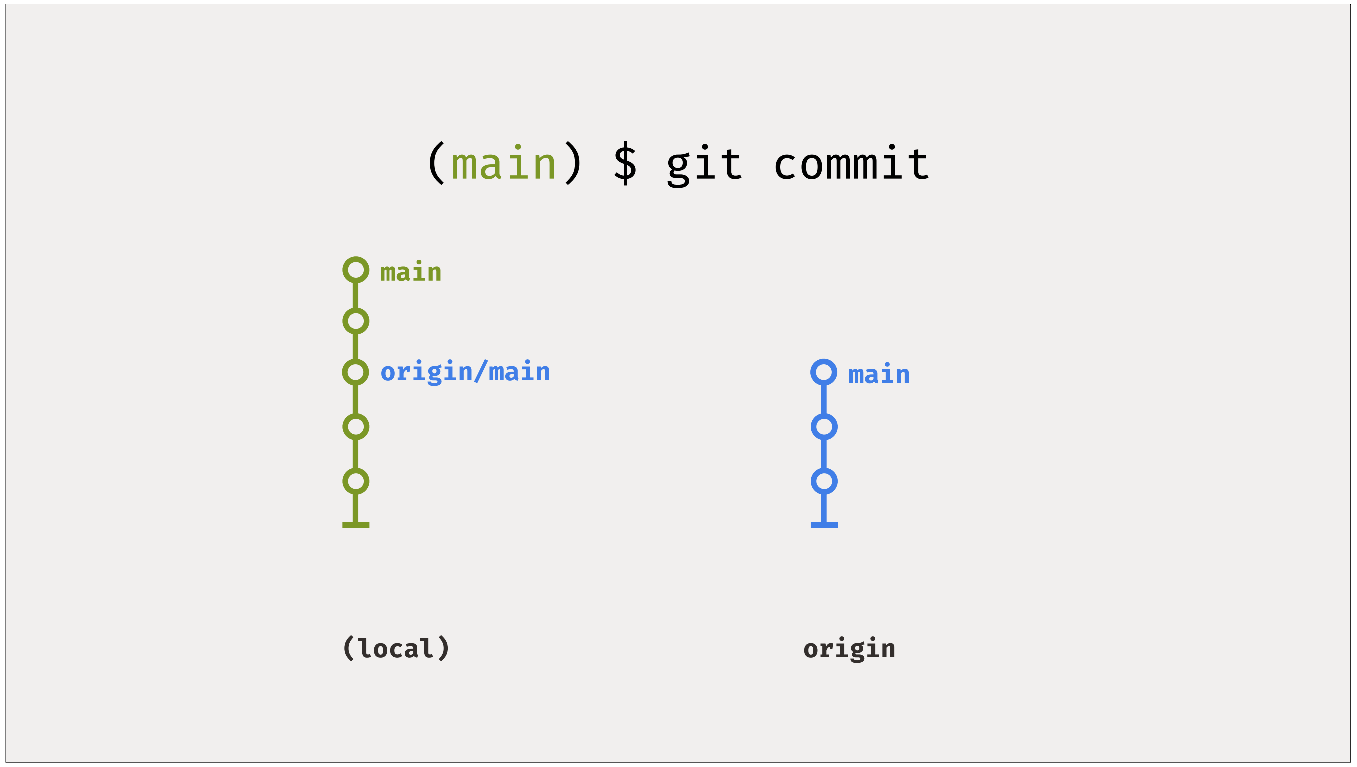
```
(main) $ git push origin main
```

And then we'll push our changes. We're pushing the main branch to origin. This pushes all of our commits.



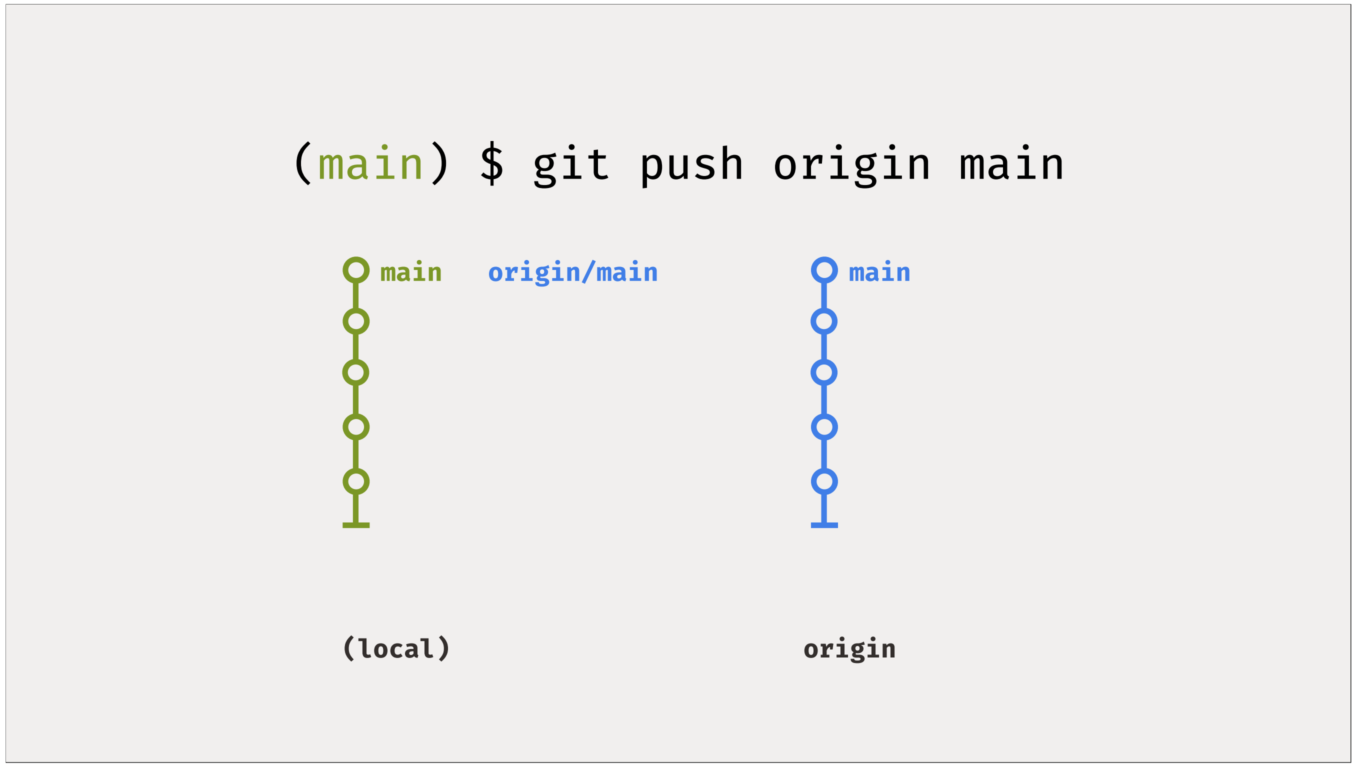
```
(main) $ git push origin main
```

The remote side now has a “main” branch, but git also adds a new label, or branch, to the local repository named after the remote branch so we can keep track of what we know about the remote repository.



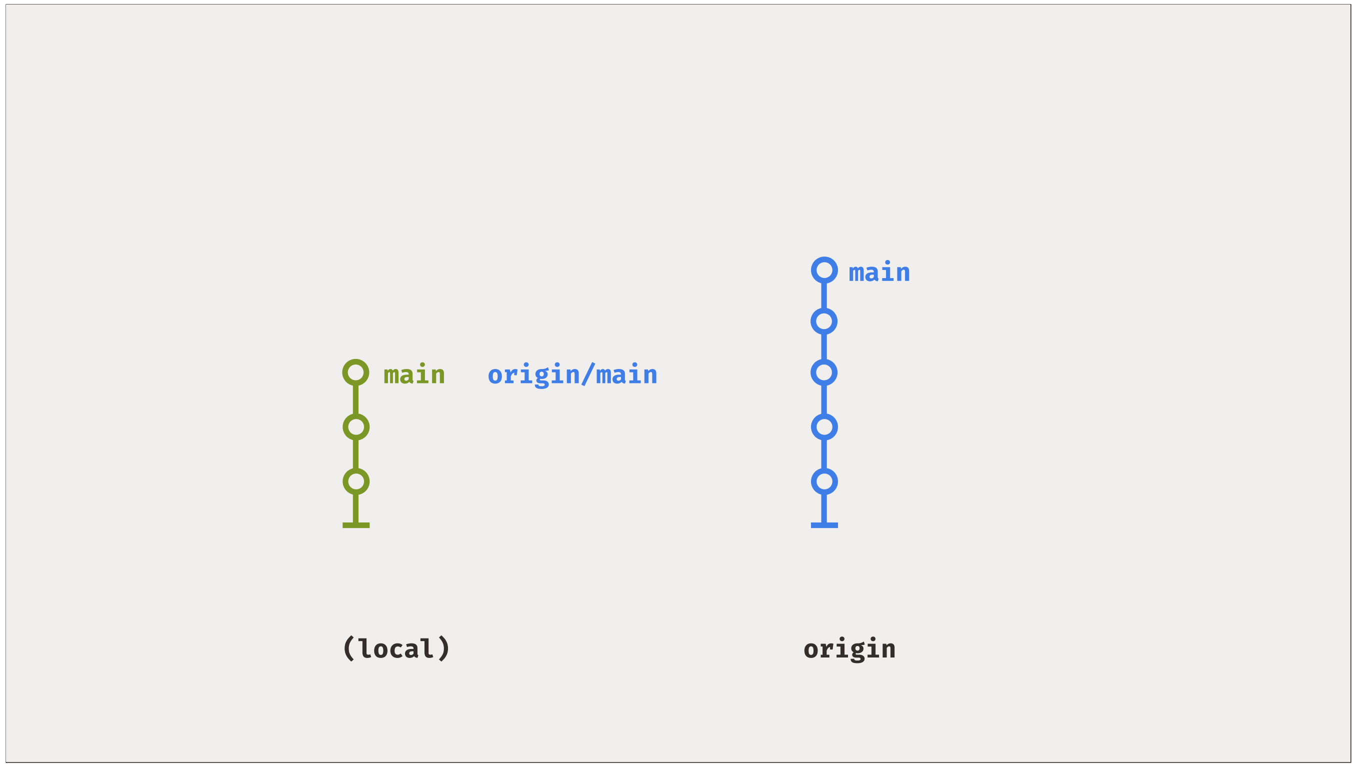
```
(main) $ git commit
```

Now we'll make some commits locally. Our local main branch changes, but the remote hasn't, at least as far as we know. The origin/main branch that we're tracking locally remains where it was.

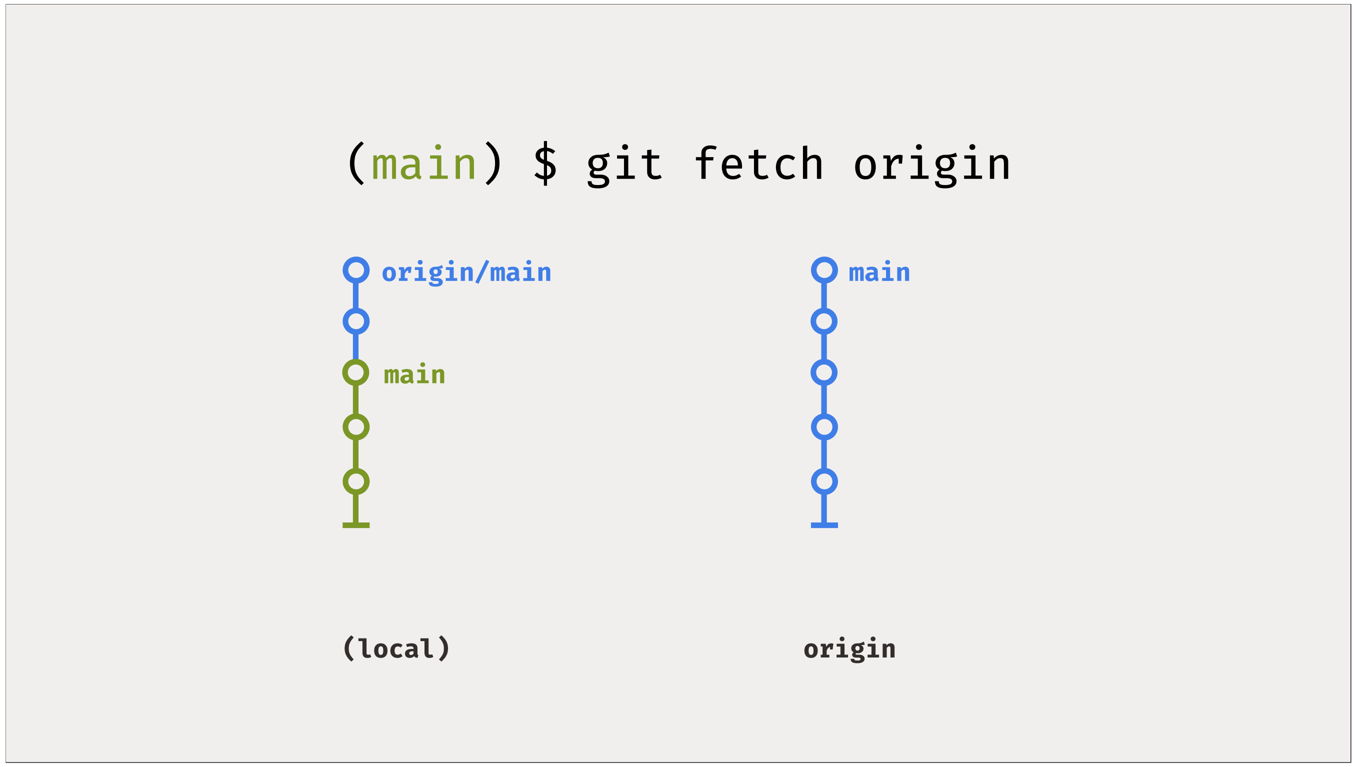


```
(main) $ git push origin main
```

And we can push the new changes on our main to the origin repository. The new commits get pushed, and all the branch information is updated both locally and in the remote repository.

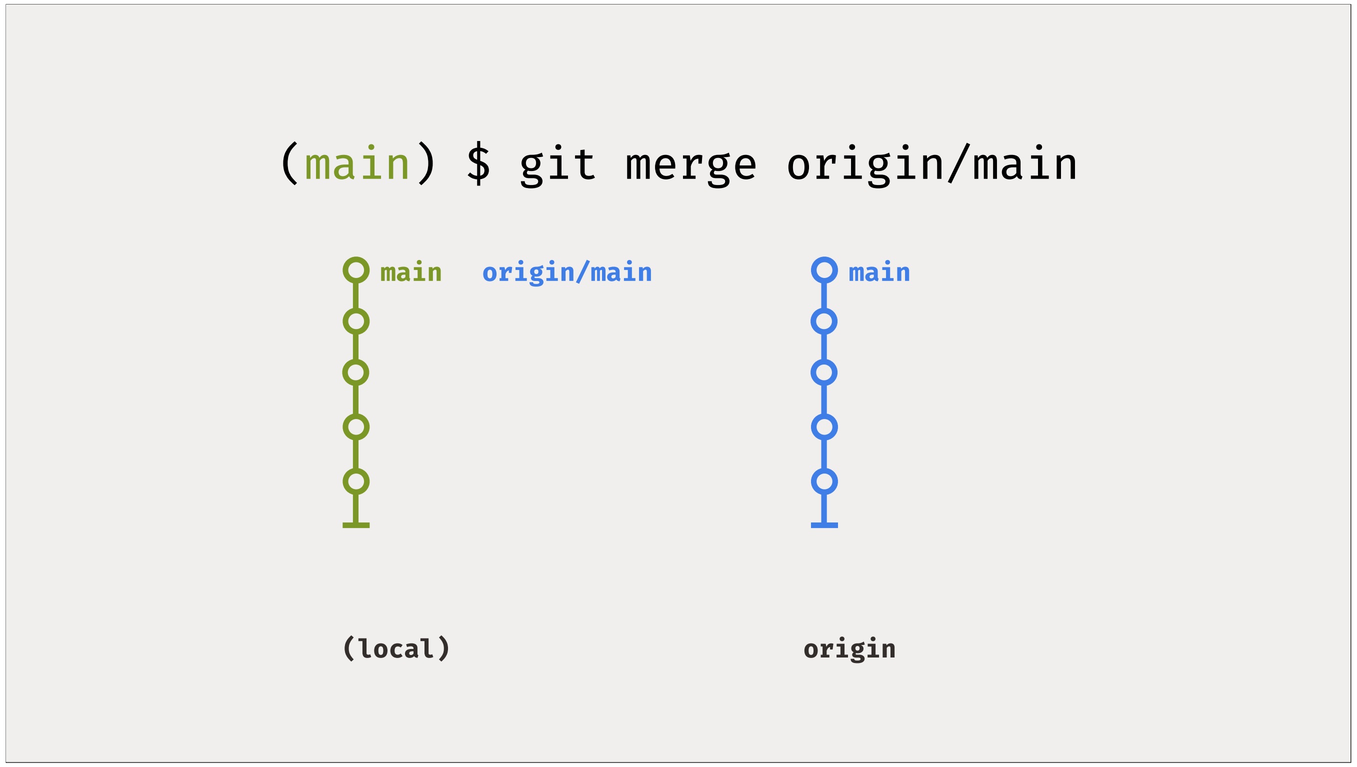


What about the other way? Let’s say the remote repository has changes that we don’t have yet.
Remember git is distributed - the only time our repository syncs with the remote repository is when we tell it to. Here, the remote repository has been updated, but our local repository hasn’t. So let’s update that.



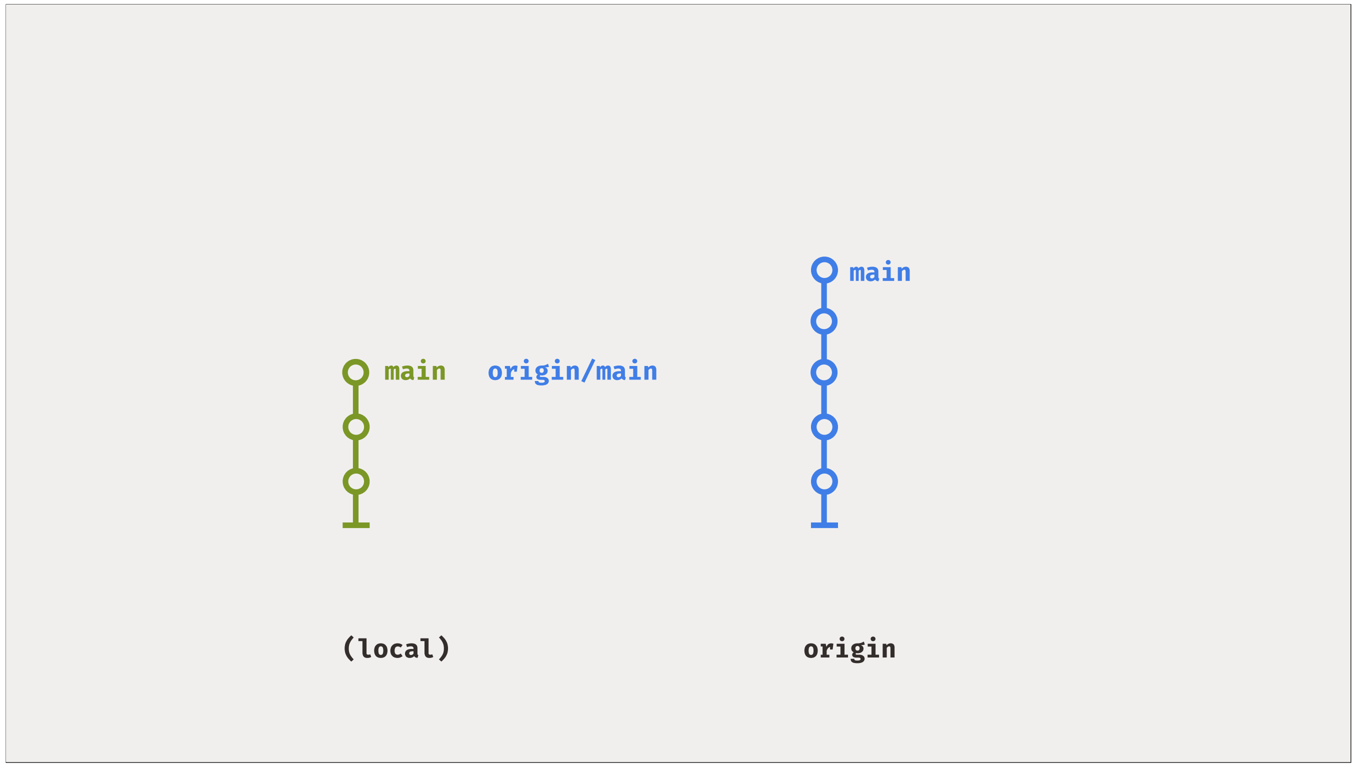
```
(main) $ git fetch origin
```

To sync up histories, we'll start with a "fetch": we ask the remote repository what it's got, and store that locally. We haven't touched our main branch yet, we've only updated what we know about the main branch in the origin repository.

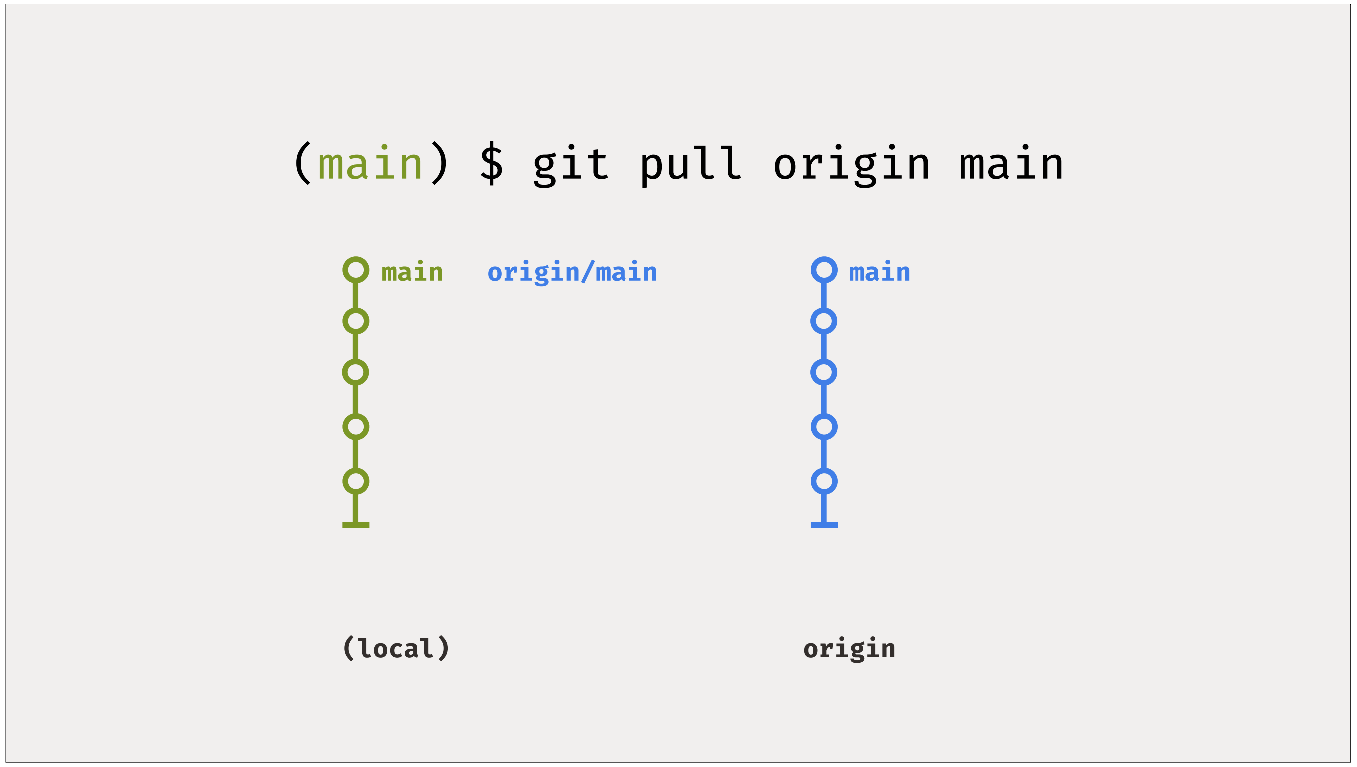


```
(main) $ git merge origin/main
```

And now, to incorporate those changes, we'll merge the updated origin main branch into our main branch.

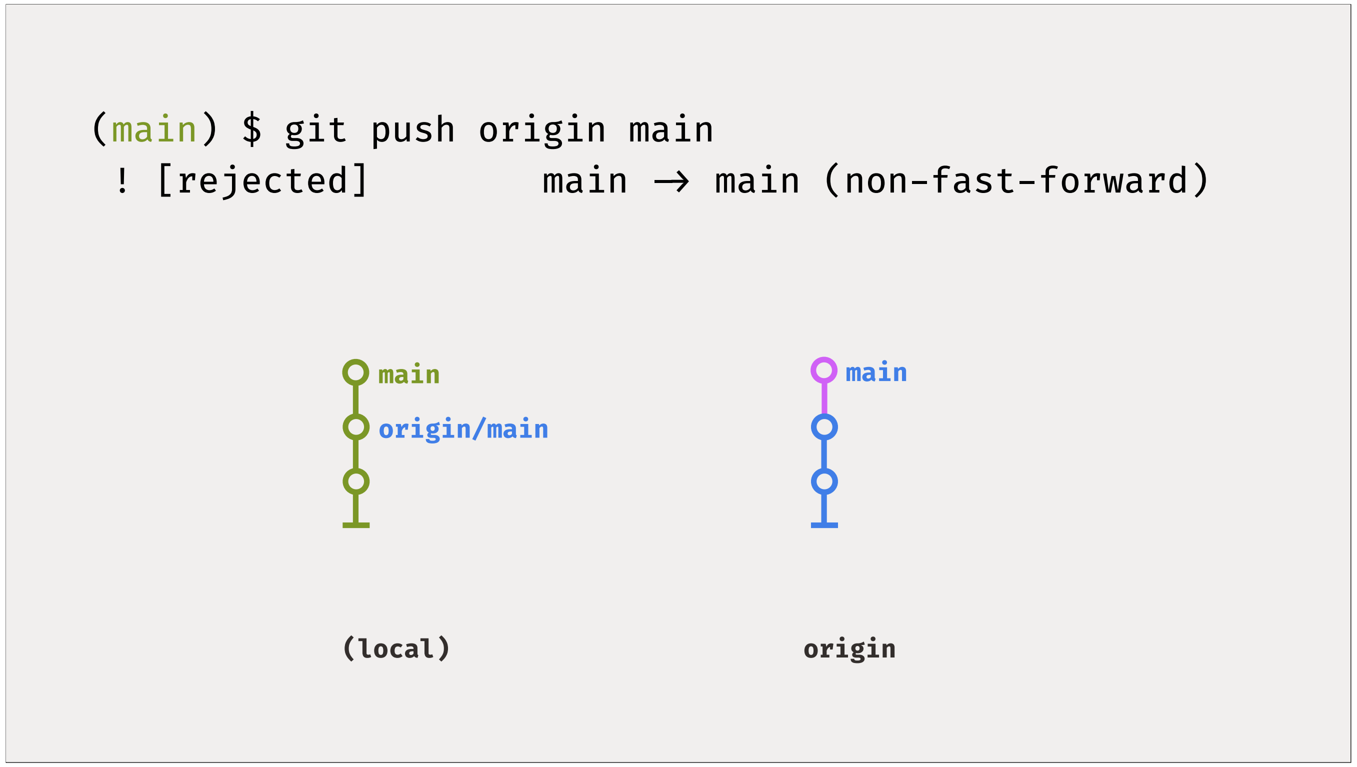


Ok, that's a two-step process, which is a bit of a pain. There's another command that does both steps together, first a fetch then a merge.



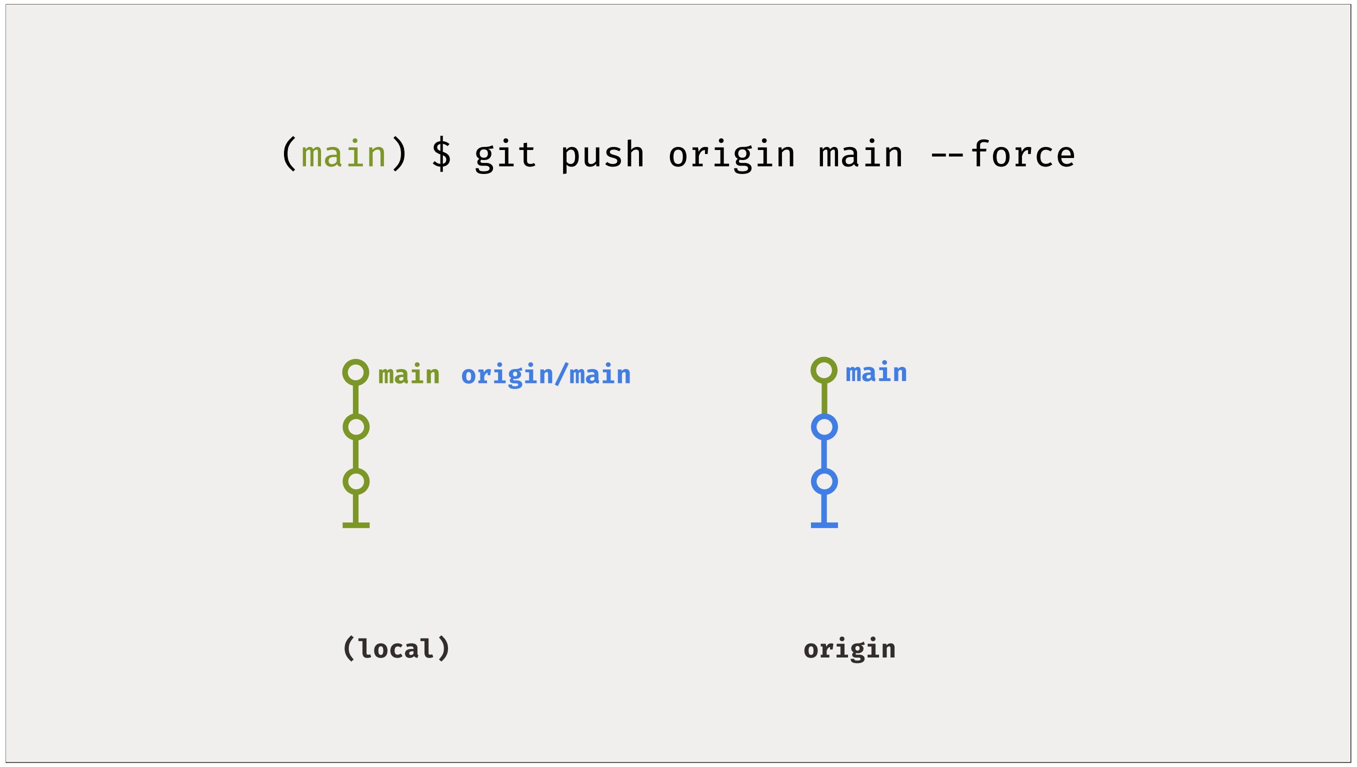
```
(main) $ git pull origin main
```

That command is “pull”: This fetches all the information from the origin, then applies it to our local repository with a merge, in one command.
If you’re ever confused about what happened to your repository after a pull, remember that it’s those two steps underneath: first a fetch, then a merge.



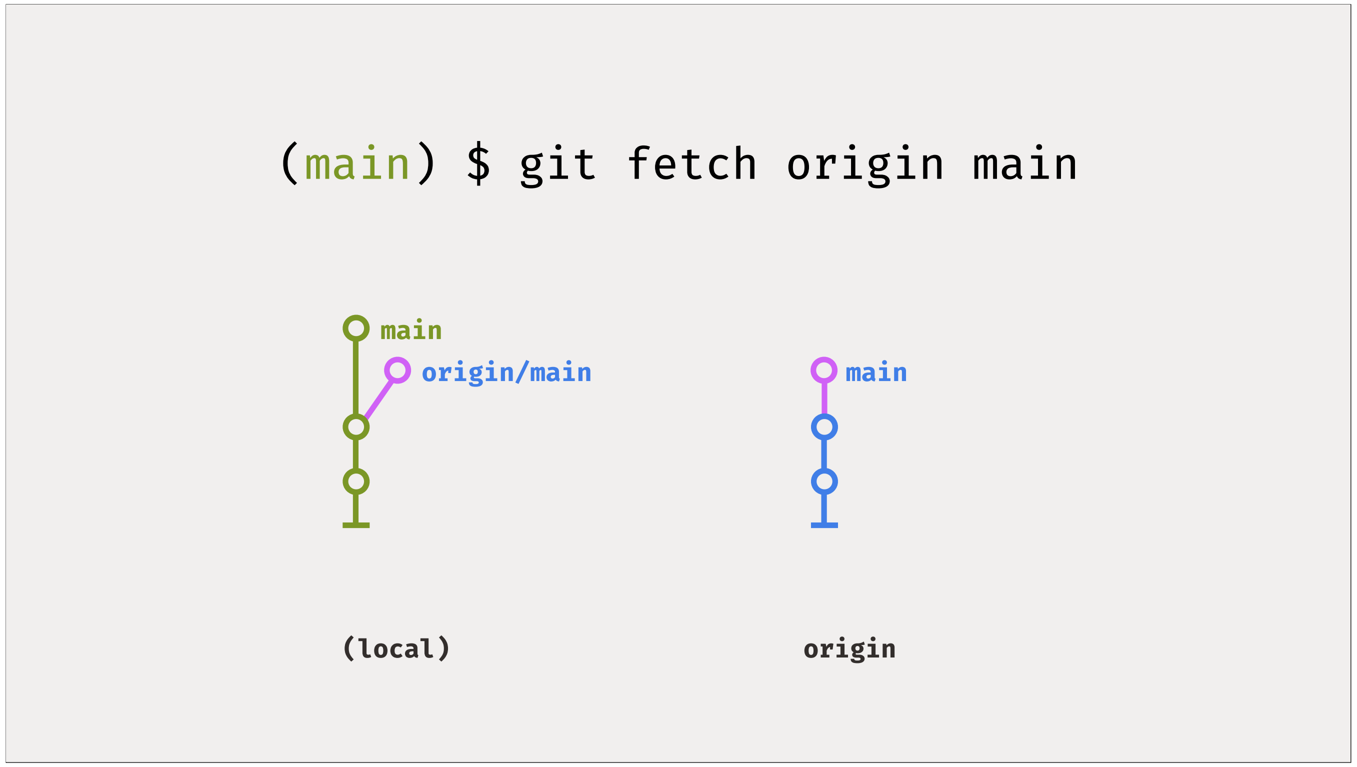
```
(main) $ git push origin main
 ! [rejected]        main → main (non-fast-forward)
```

One more scenario: You’ve made changes to your local branch, and there are already different changes in the remote branch.
If you try and push your changes, it will fail: git will say “sorry, my version of main is different from yours!”
They’ve got a common history, but there’s a new but different commit on each side.



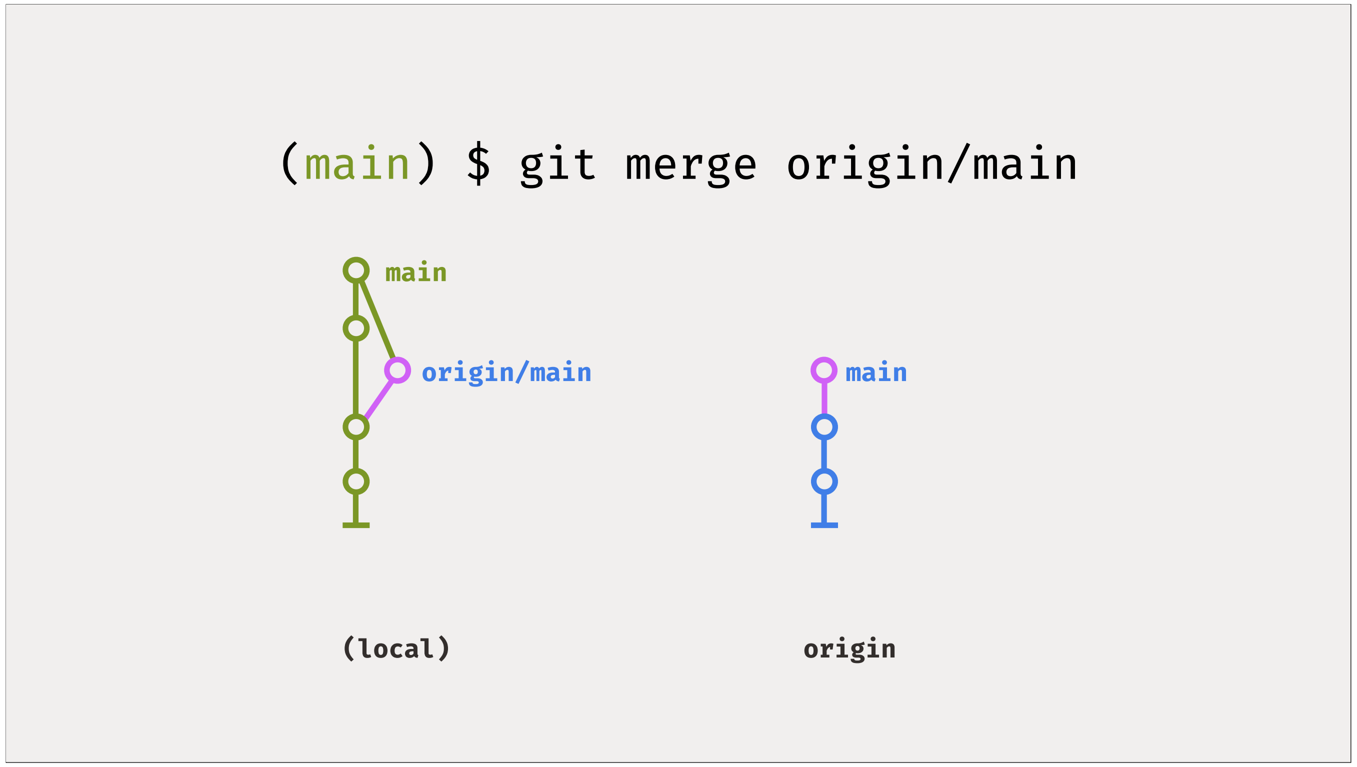
```
(main) $ git push origin main --force
```

Now, you could just force it. Forget what’s there, overwrite it. Not a great idea, though.



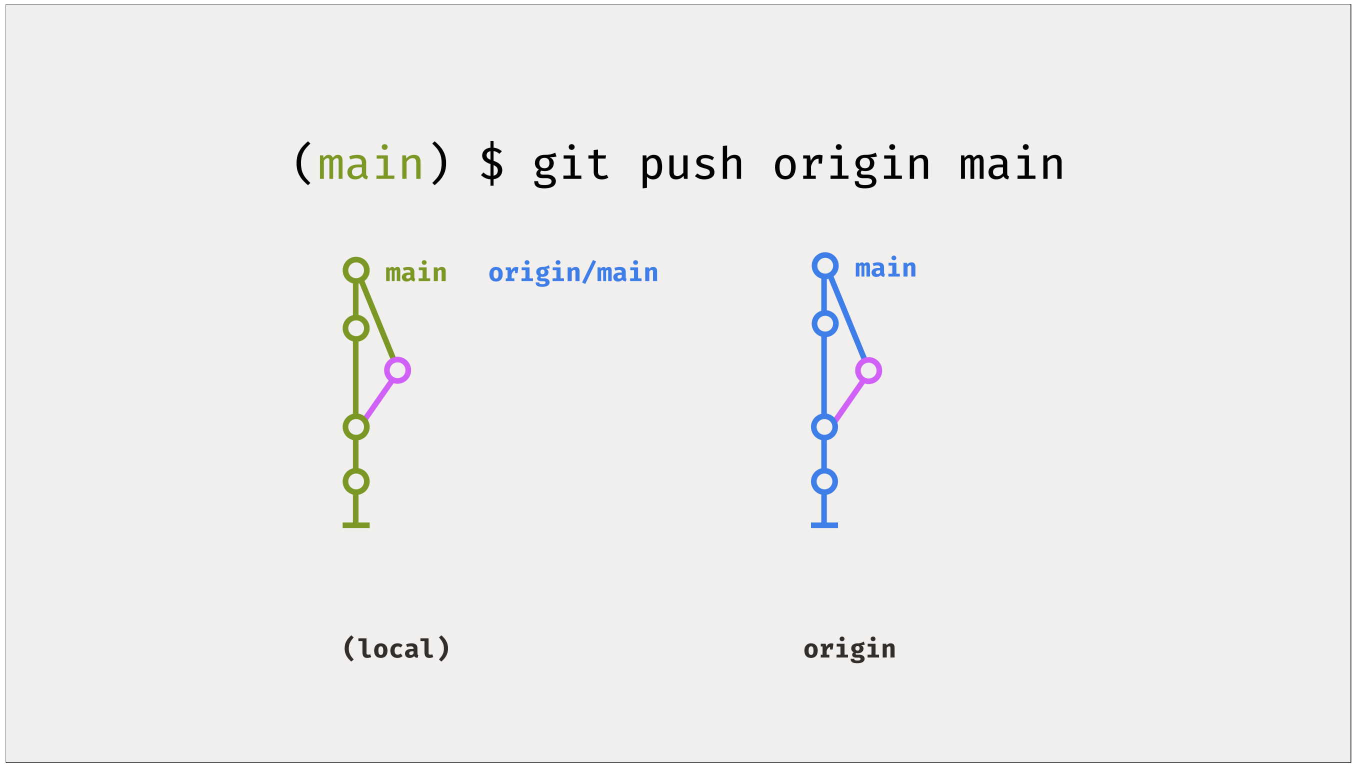
```
(main) $ git fetch origin main
```

The way to resolve this is by first pulling the remote changes in to our repository to incorporate them. I’m going to show you the two-step fetch-and-merge instead of a pull, so you can see exactly what happens. First, we fetch the changes, updating our local repository.



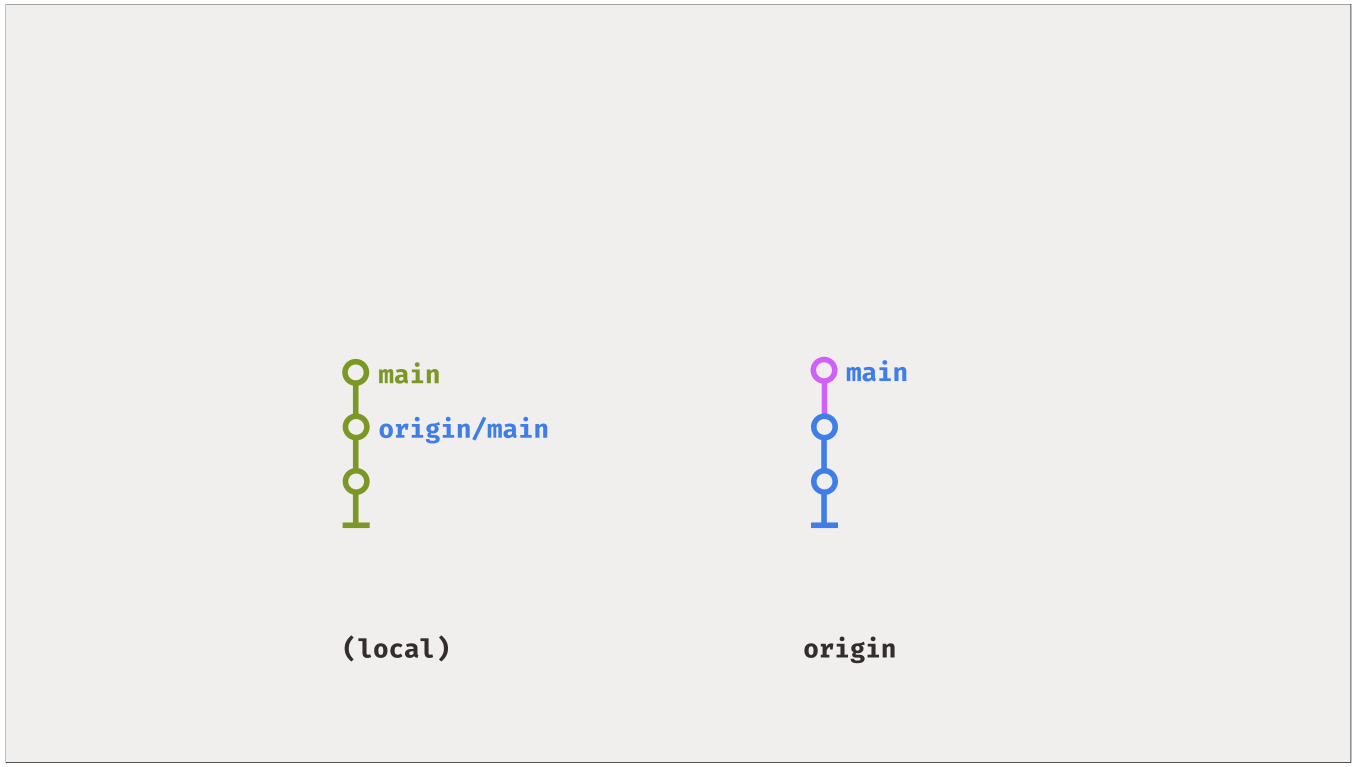
```
(main) $ git merge origin/main
```

We merge the changes into our branch.

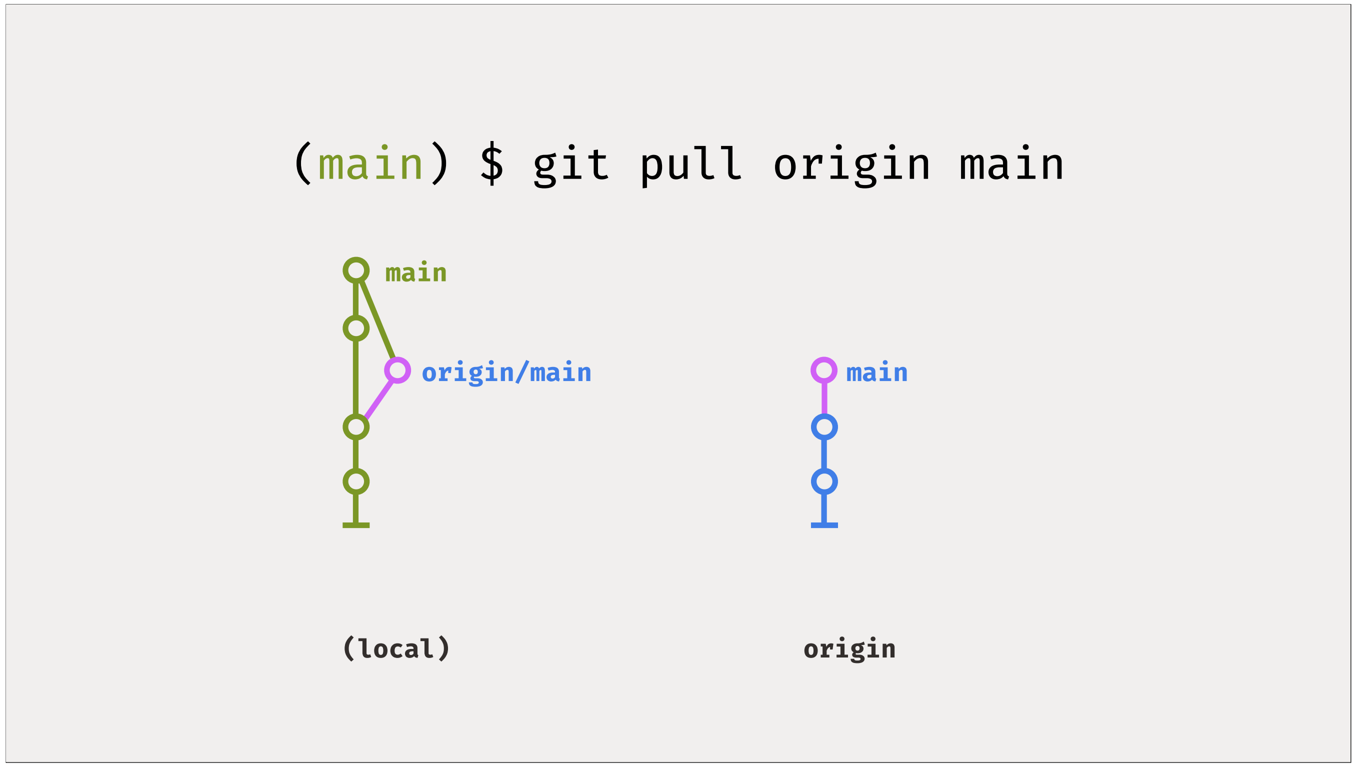


```
(main) $ git push origin main
```

And finally, we can push our changes to the remote repository.

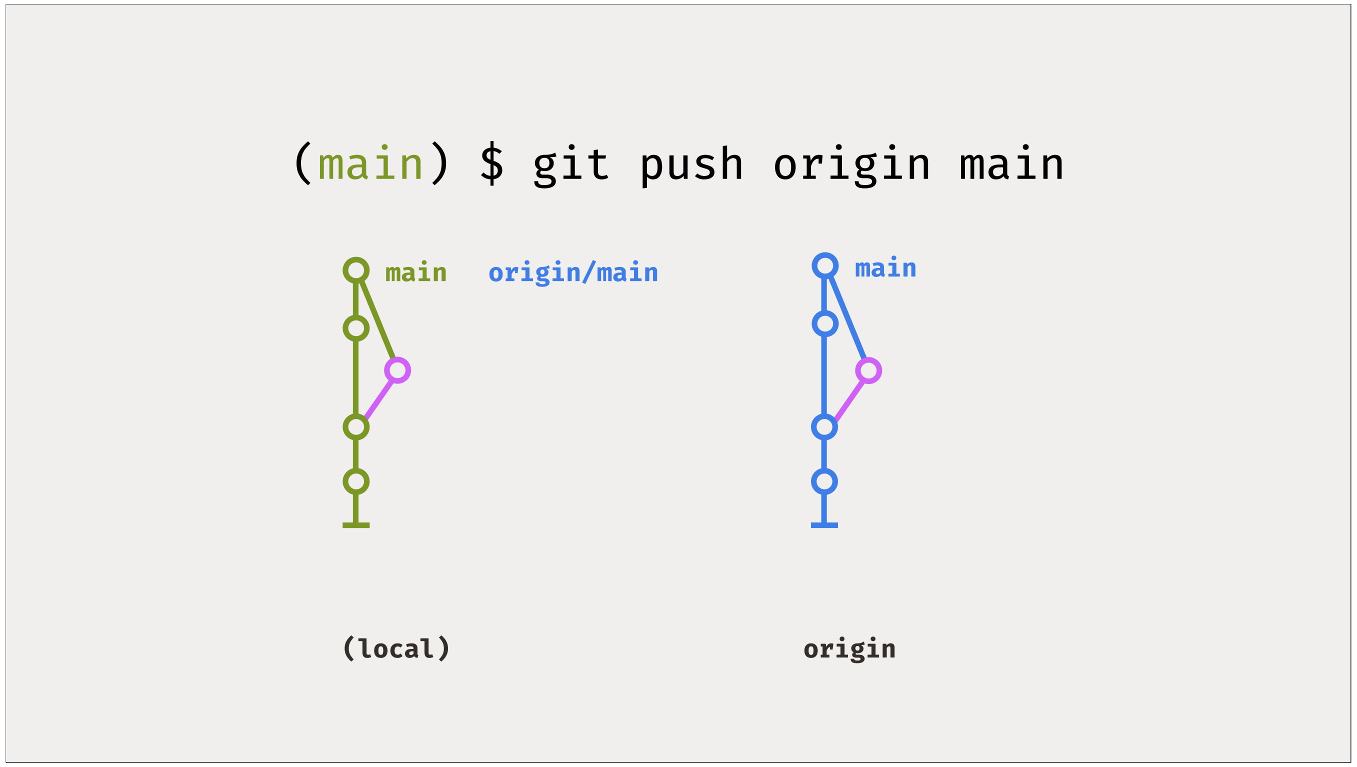


Showing this once more, using pull as one step:



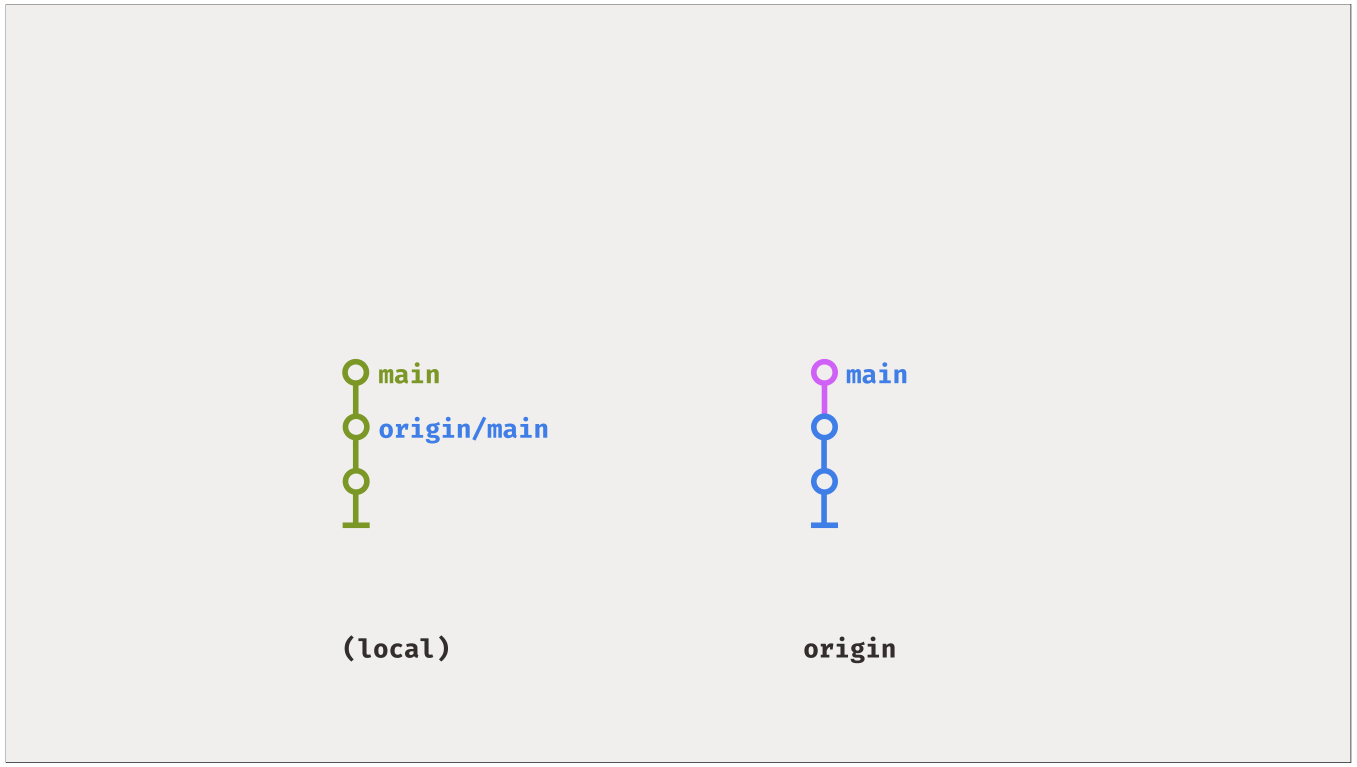
```
(main) $ git pull origin main
```

We pull the changes, automatically merging them in.

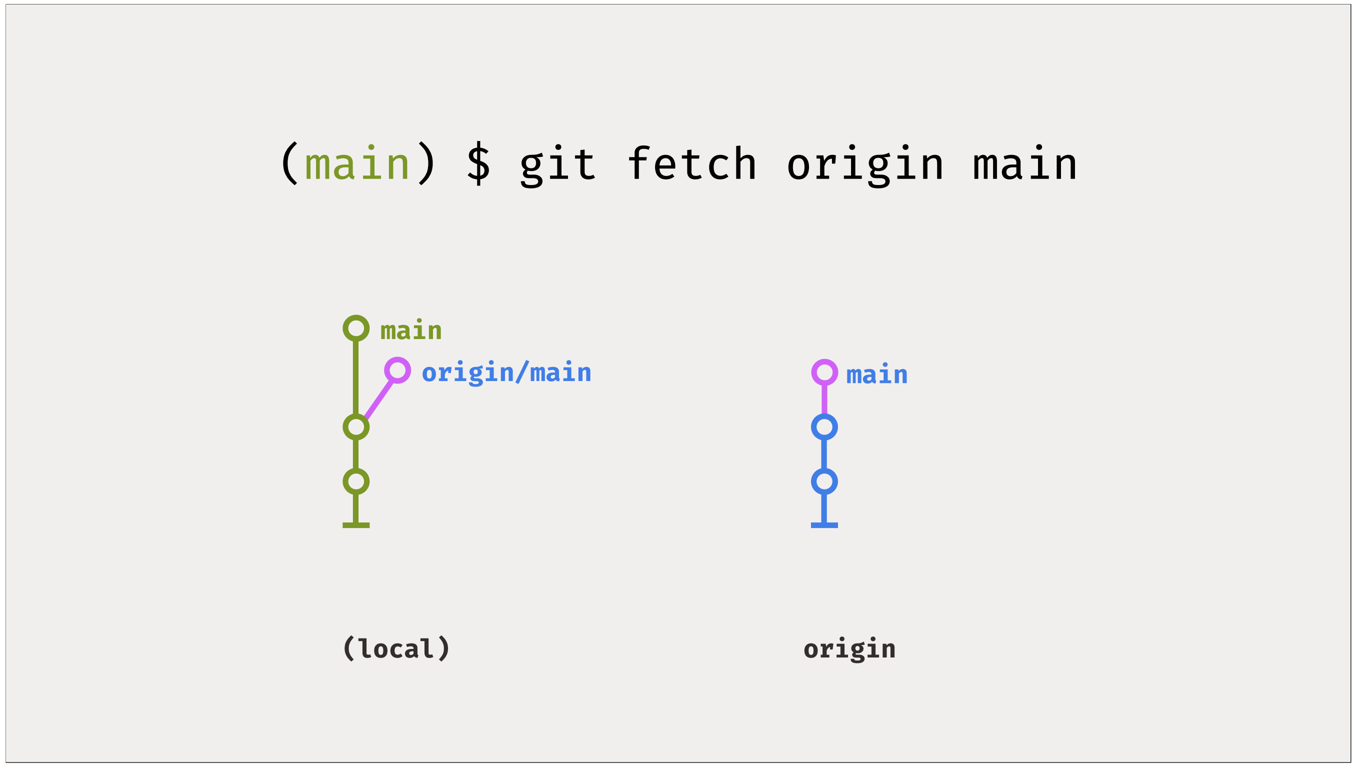


```
(main) $ git push origin main
```

And push them to the remote.
This way of resolving the different histories creates a merge commit.

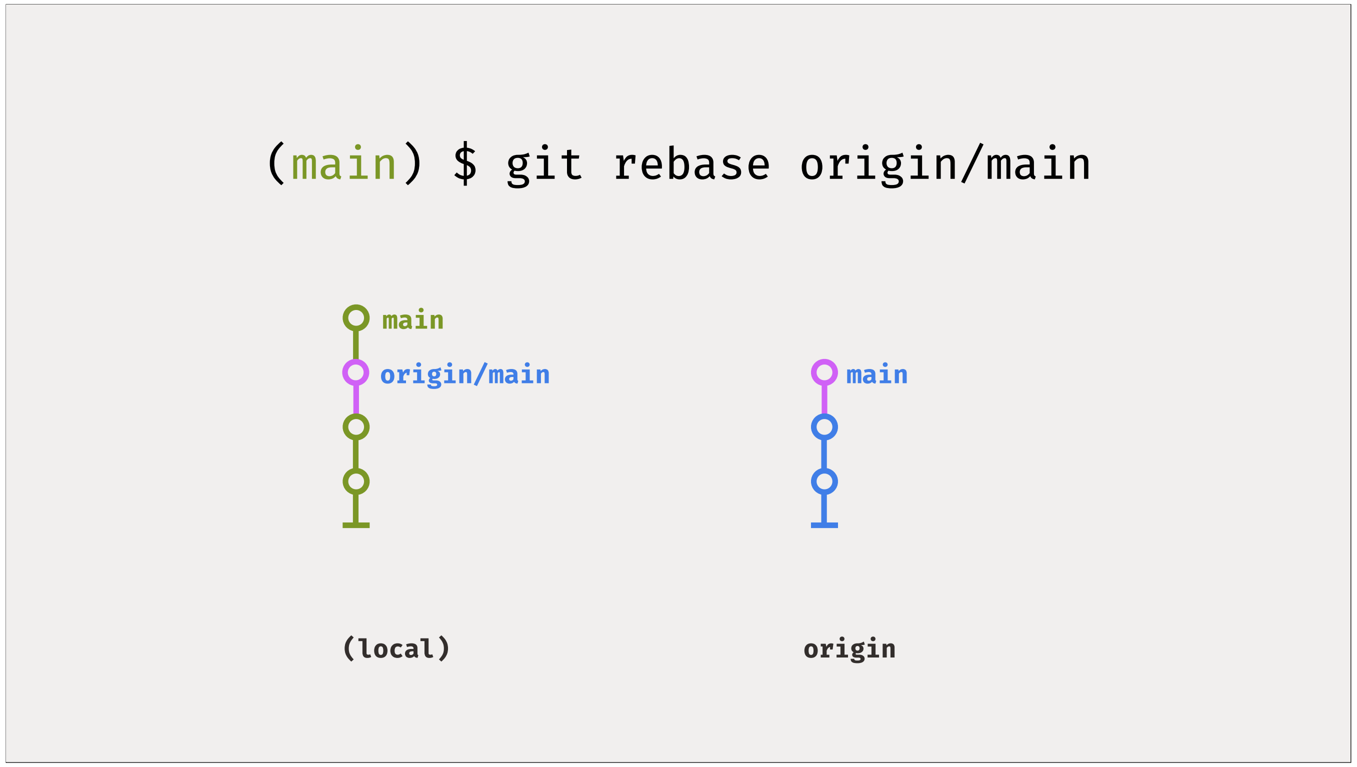


One final example. Let's say we don't want to create a merge commit. Remember the rebase operation I showed you earlier? You can make rebase a part of pull if you want.



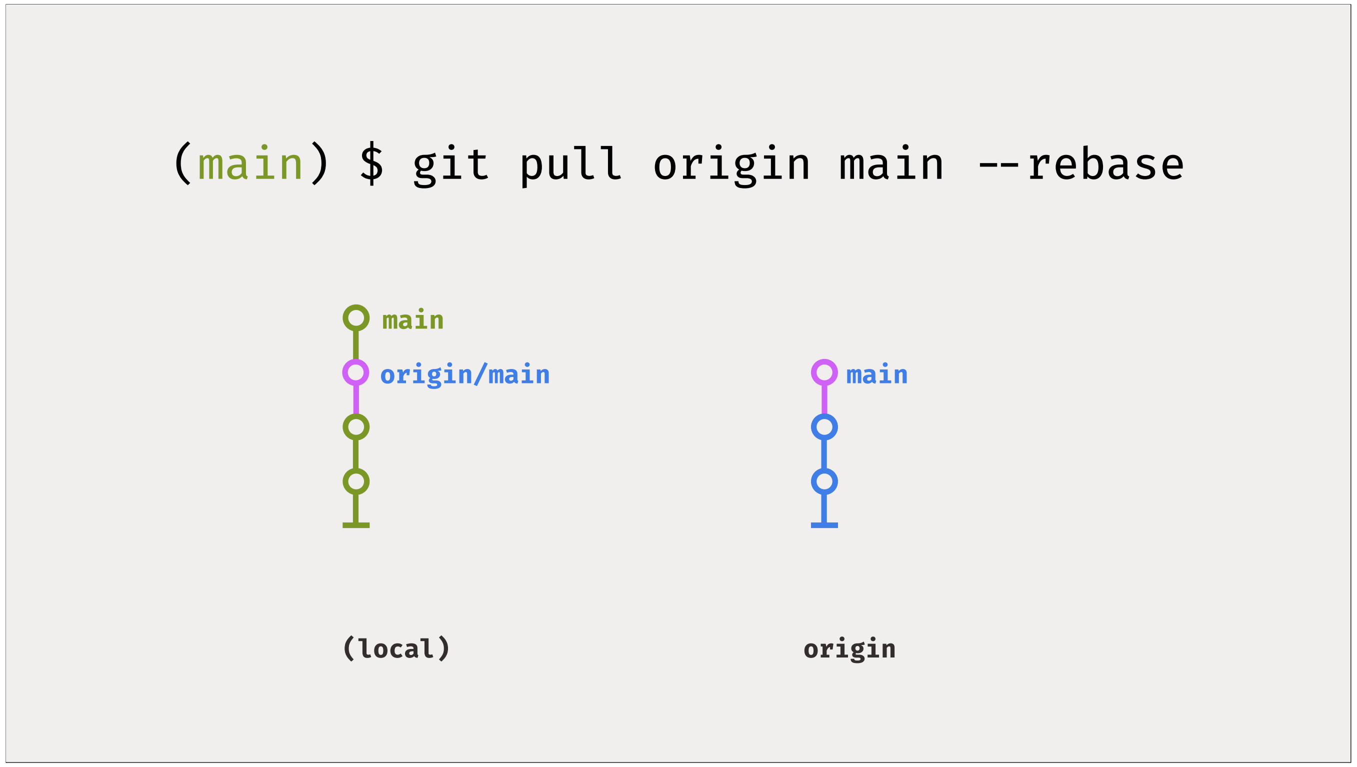
```
(main) $ git fetch origin main
```

Again, we'll do it with two steps. First we fetch the changes.



```
(main) $ git rebase origin/main
```

And then we rebase our changes onto the origin main branch.



```
(main) $ git pull origin main --rebase
```

main

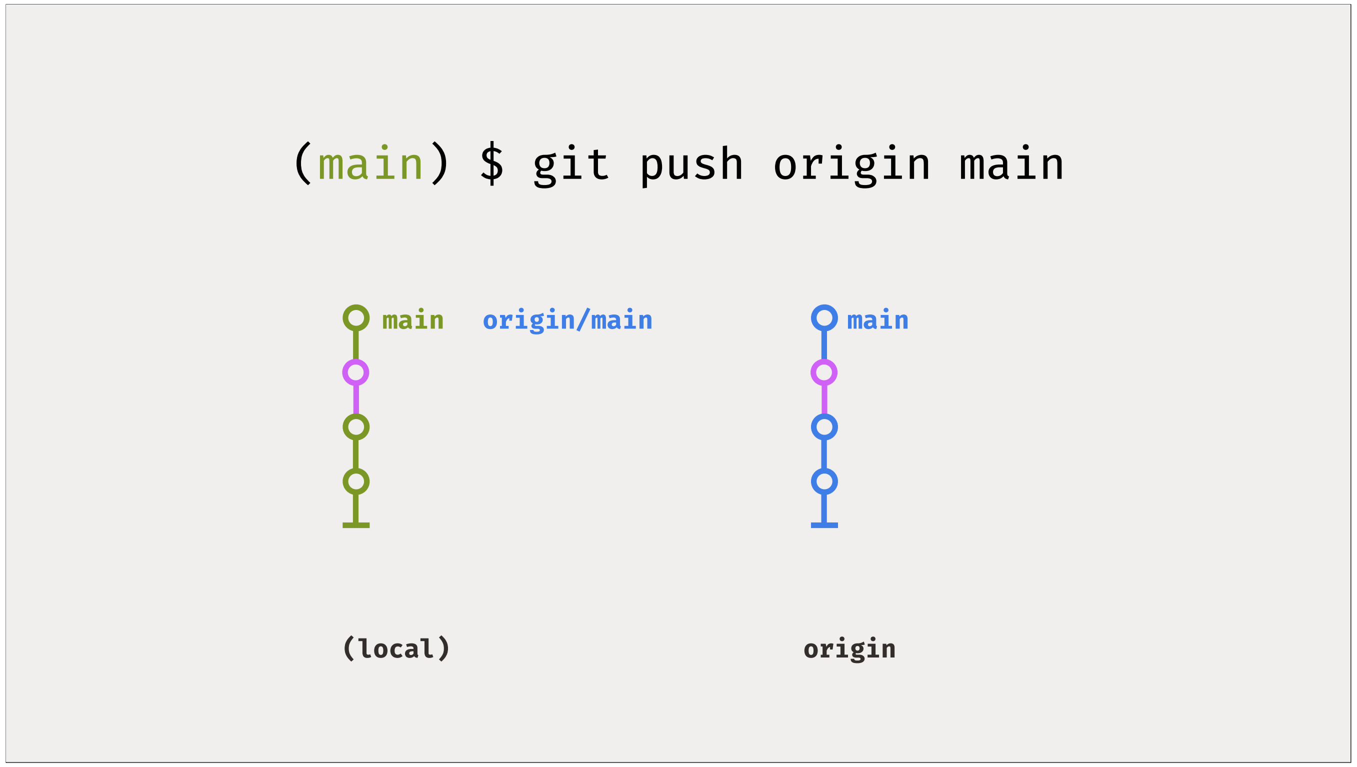
origin/main

main

(local)

origin

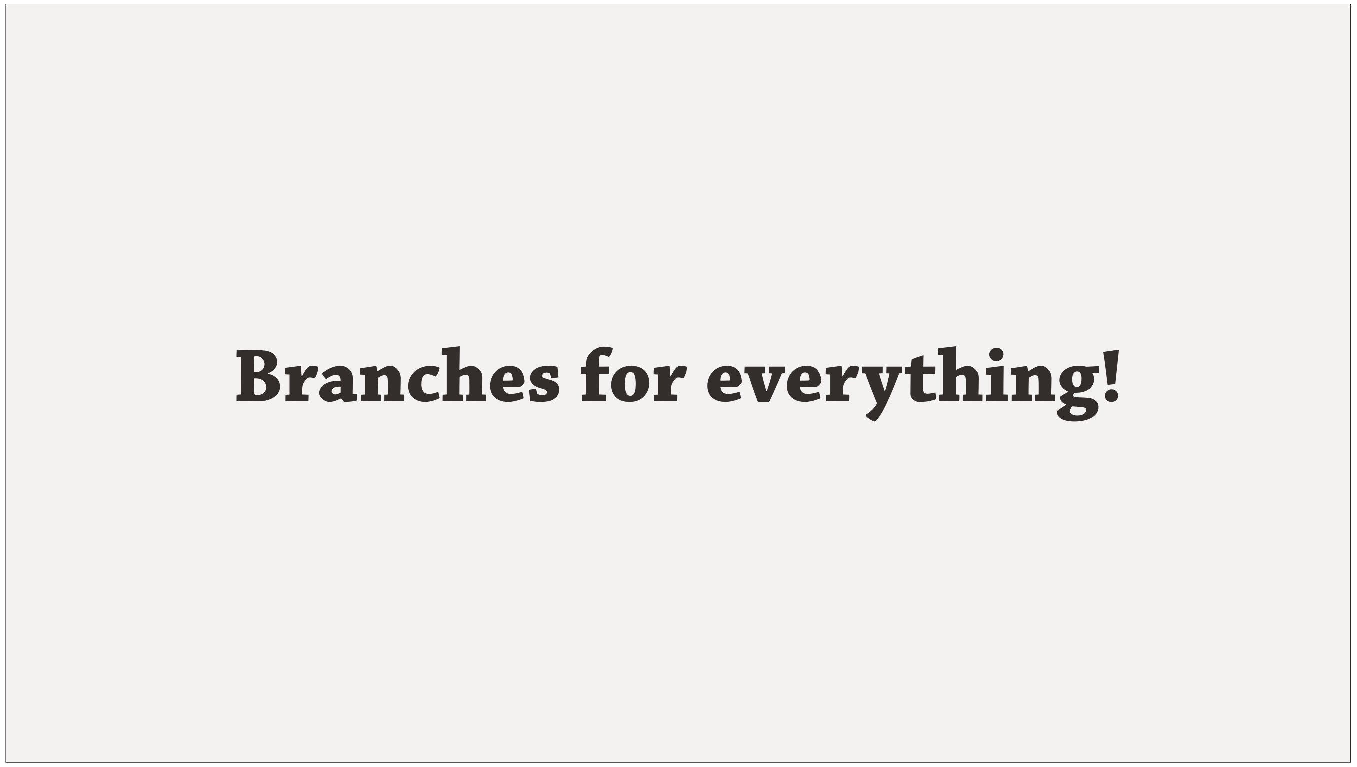
And again, we can do this together with the pull command and the rebase flag.



```
(main) $ git push origin main
```

And finally, we can push our changes.

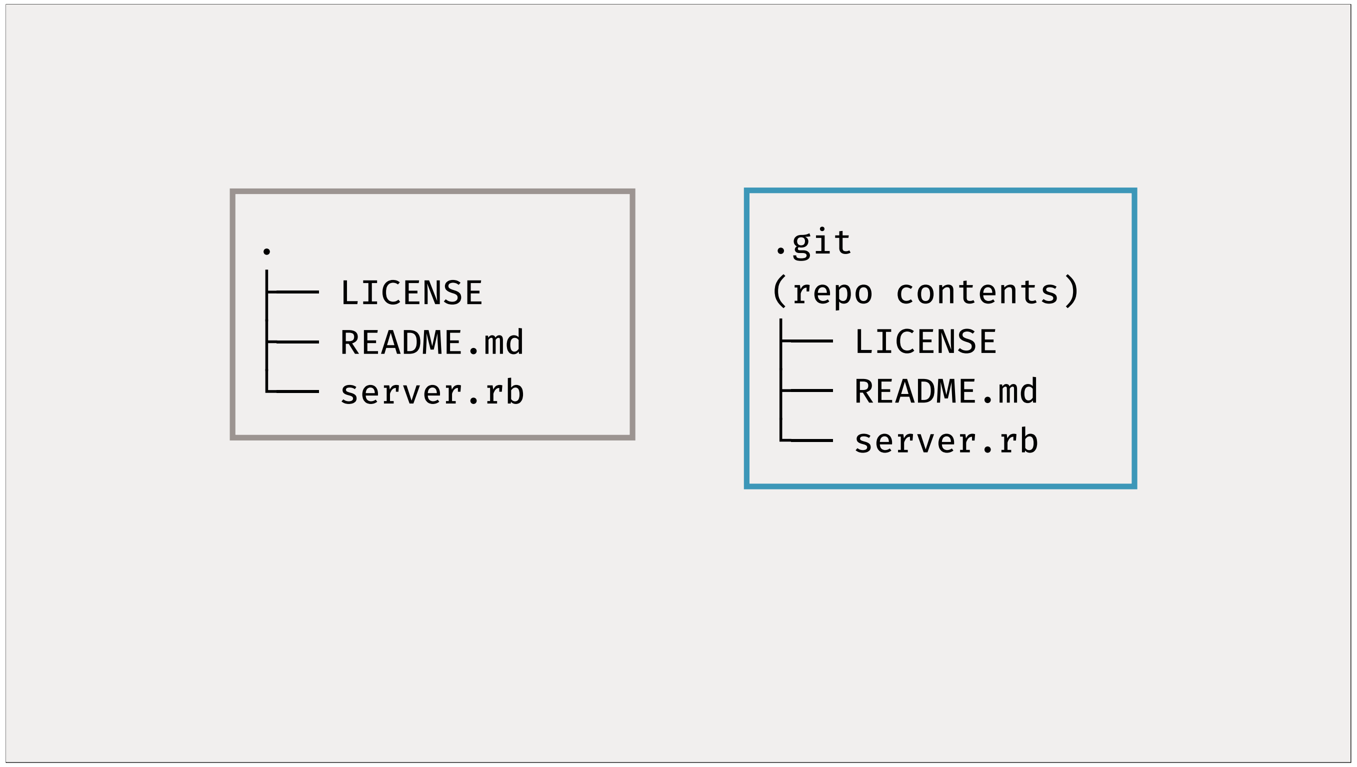
When should you use branches?



Branches for everything!

Use branches for everything! For new features, for bug fixes, for experiments. This lets you make new commits on code you’re working on, and then easily switch back to do something else.

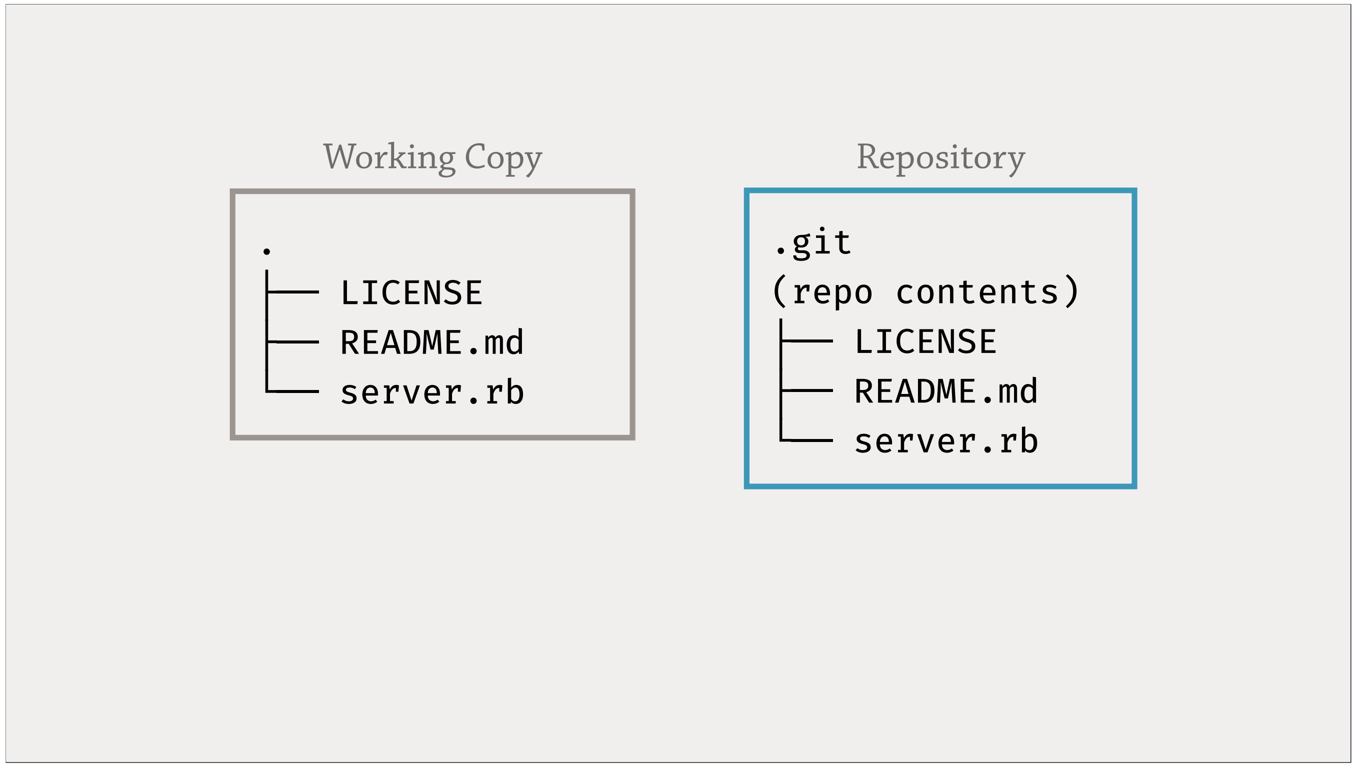
Staging



```
.
├── LICENSE
├── README.md
└── server.rb
```

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Let’s back to this diagram, where we have code living alongside a repository. I want to point out a few things to help you work with repositories locally.



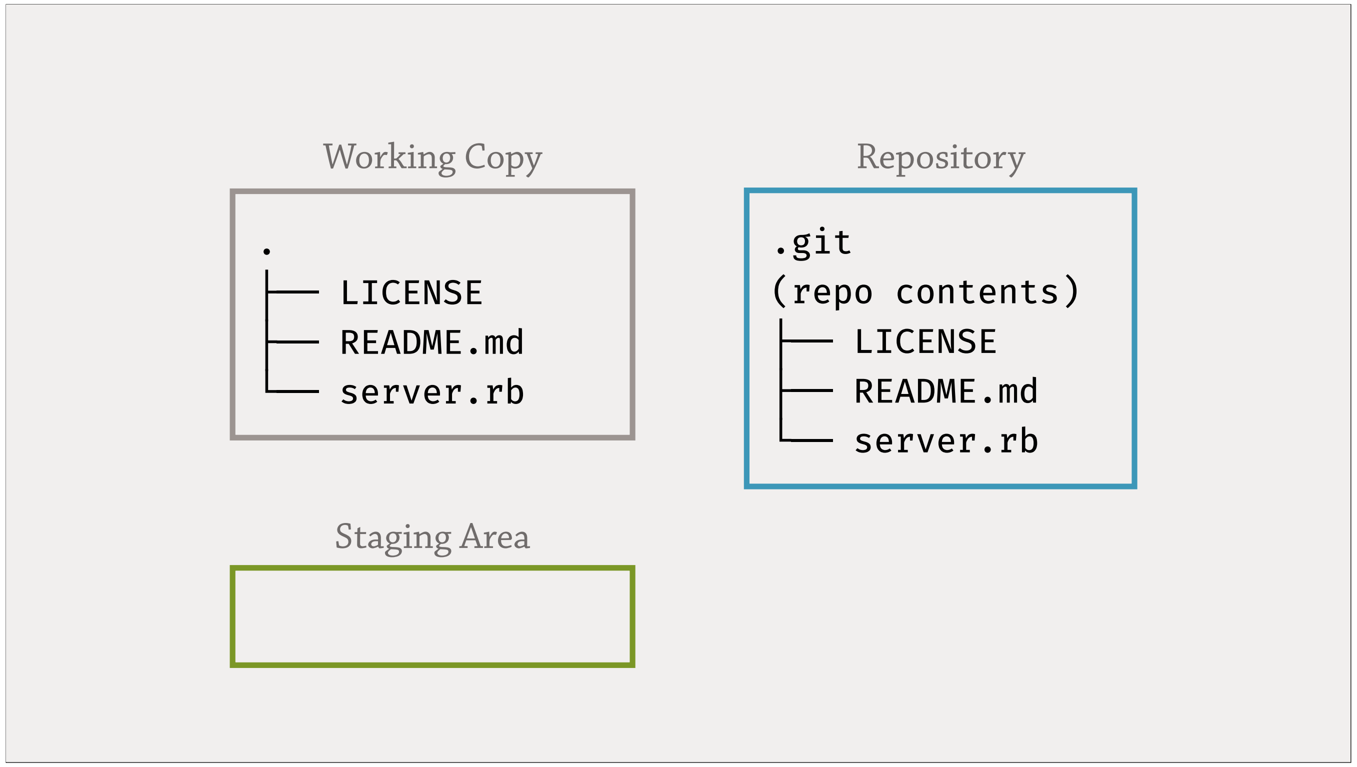
Working Copy

```
.
├── LICENSE
├── README.md
└── server.rb
```

Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

First, let’s give these boxes names. The files on disk that you look at and edit are called your “working copy”. The git directory is called your “repository”.



Working Copy

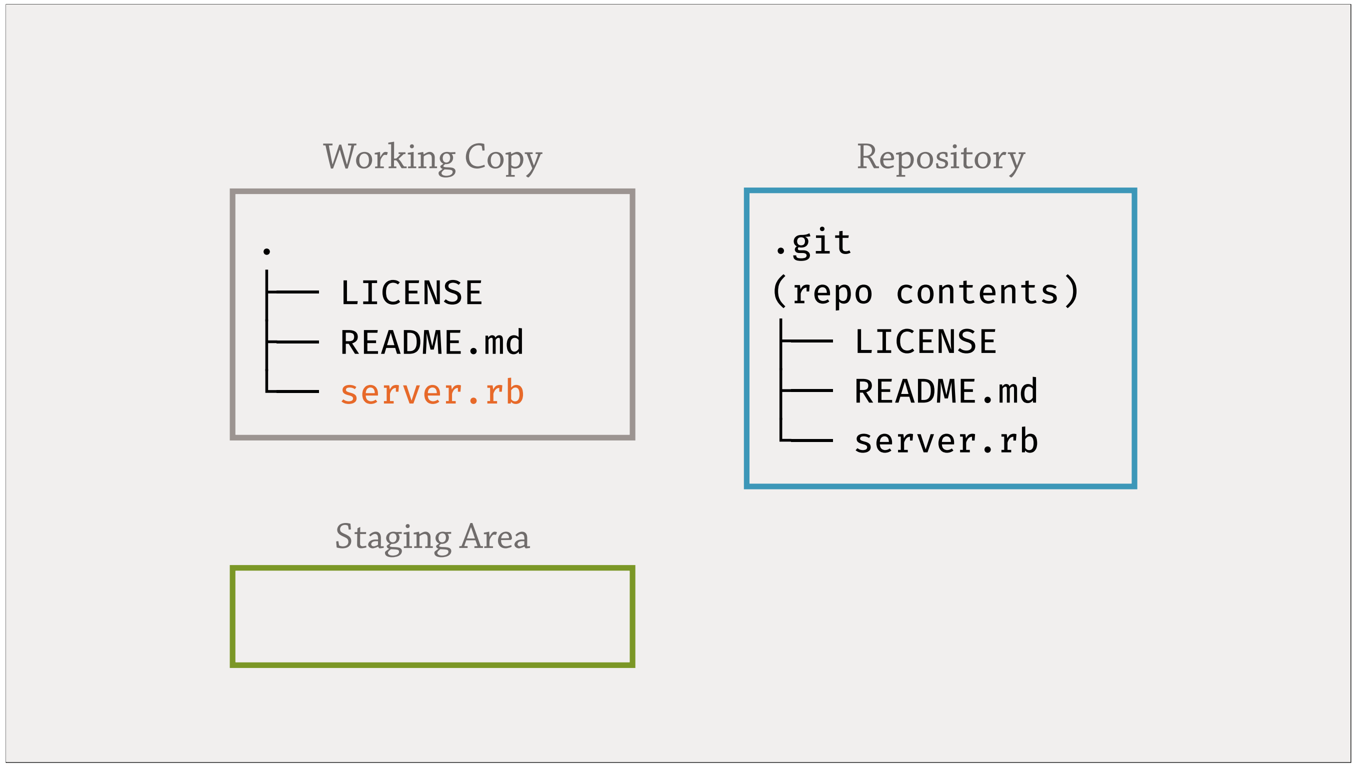
```
.
├── LICENSE
├── README.md
└── server.rb
```

Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Staging Area

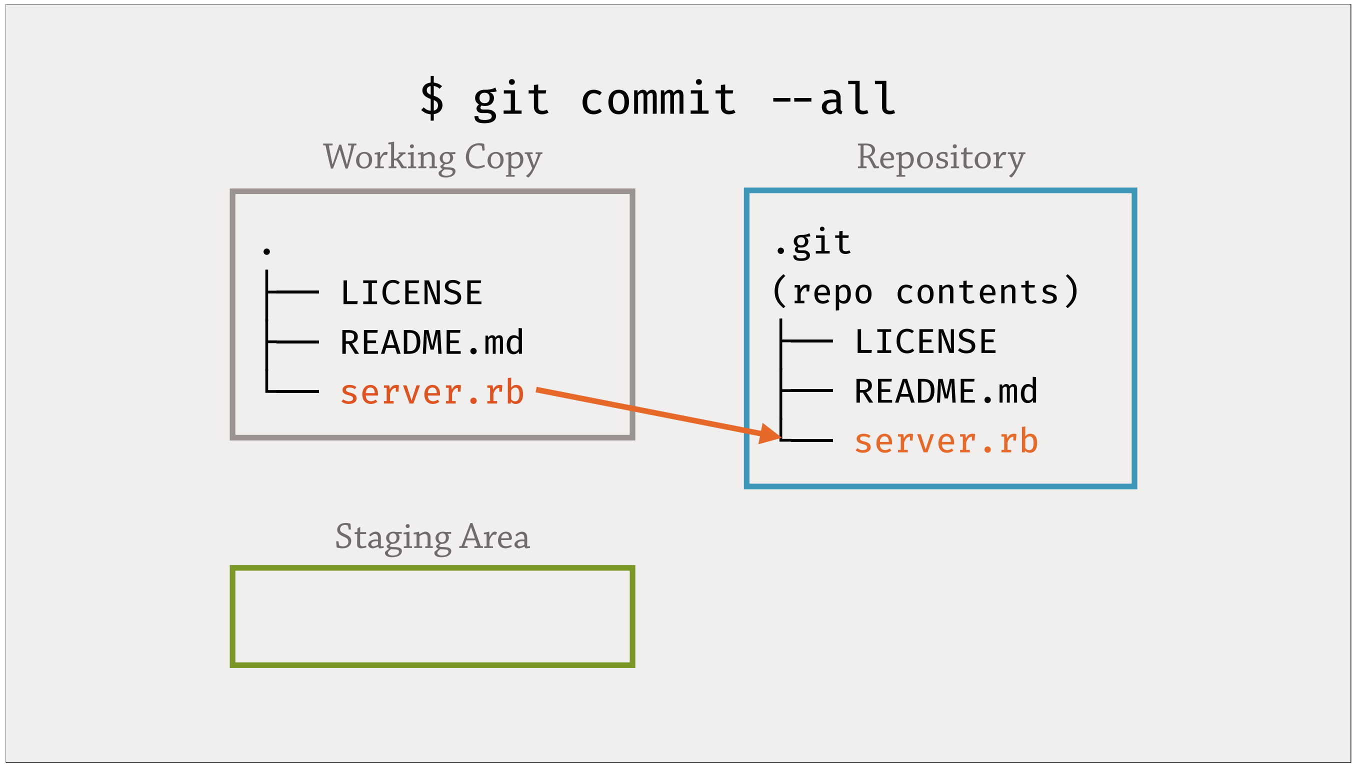
When it comes to committing code to a repository, there’s one more missing piece: the staging area.
When you make changes to your code, or add a file, or delete a file, you have to tell git that you want to include something in a commit. Git tracks the contents of files, but you have to tell git about what you’e changed so that it can be included in a new commit.



```
.
├── LICENSE
├── README.md
└── server.rb
```

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Let’s say you’ve edited your server file.



```
$ git commit --all
```

Working Copy

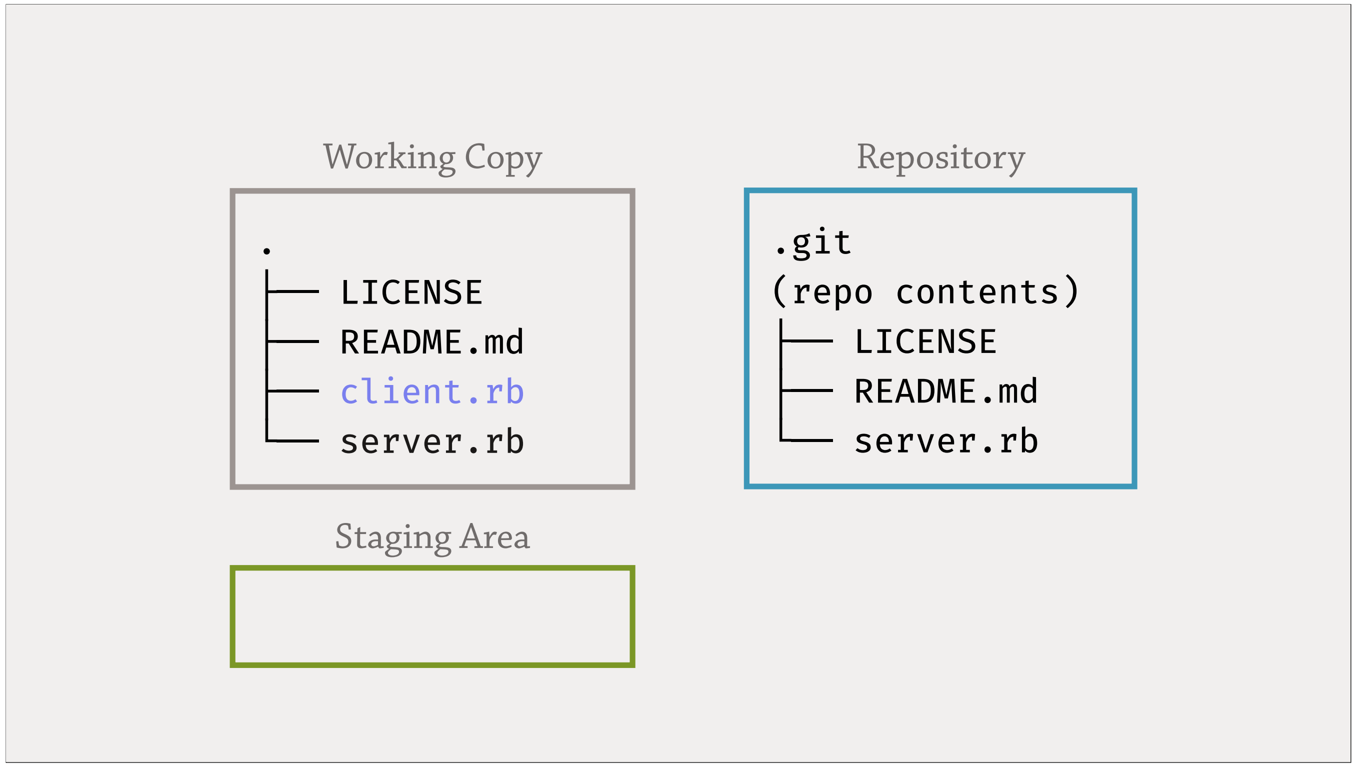
```
.
├── LICENSE
├── README.md
└── server.rb
```

Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Staging Area

The repository is already tracking server.rb, so here you can skip staging and just tell git to commit everything that's changed that it knows about. That's the "-a" flag. And the changes are committed to the repository.



Working Copy

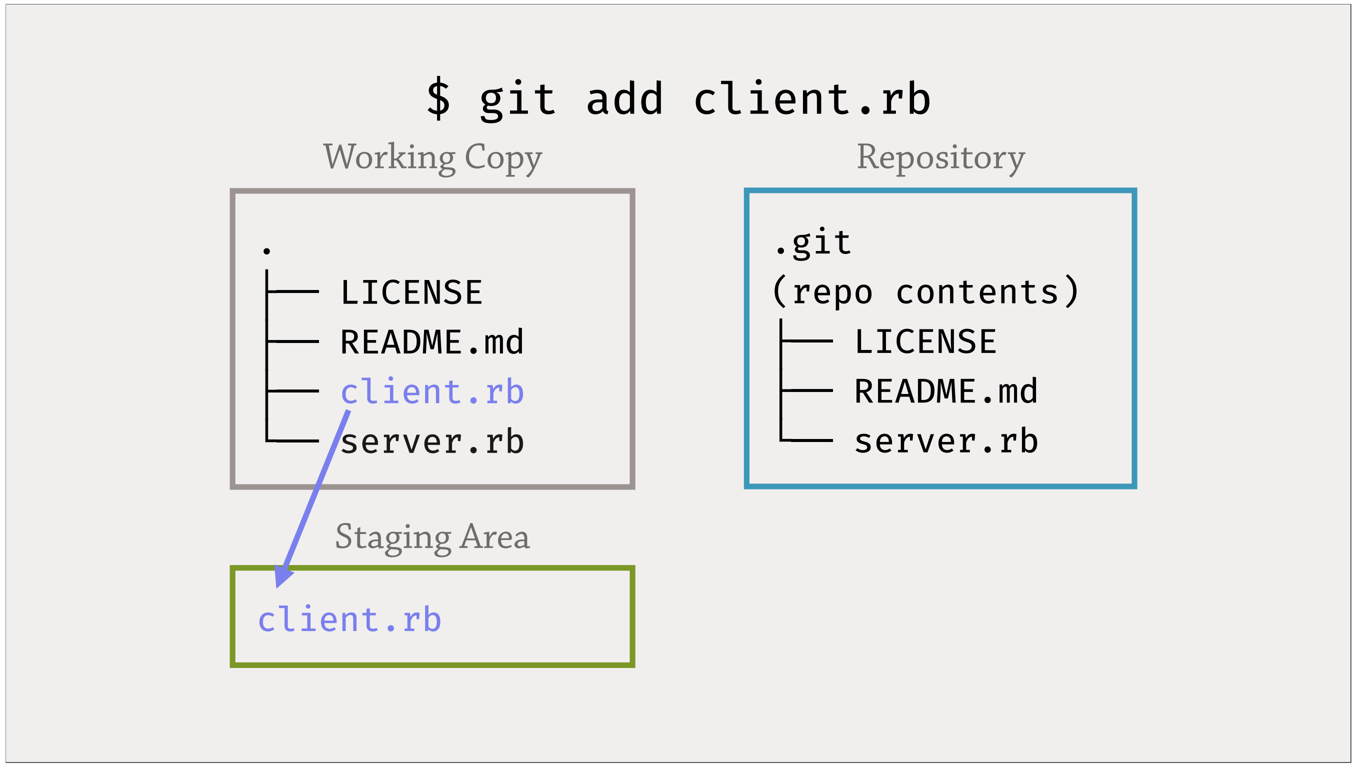
```
.
├── LICENSE
├── README.md
├── client.rb
└── server.rb
```

Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Staging Area

But let’s say you’ve added a new file. Git doesn’t know about it, so you can’t just commit those changes — git doesn’t think anything’s changed, because it doesn’t know about the new client.rb file.



```
$ git add client.rb
```

Working Copy

```
.
├── LICENSE
├── README.md
├── client.rb
└── server.rb
```

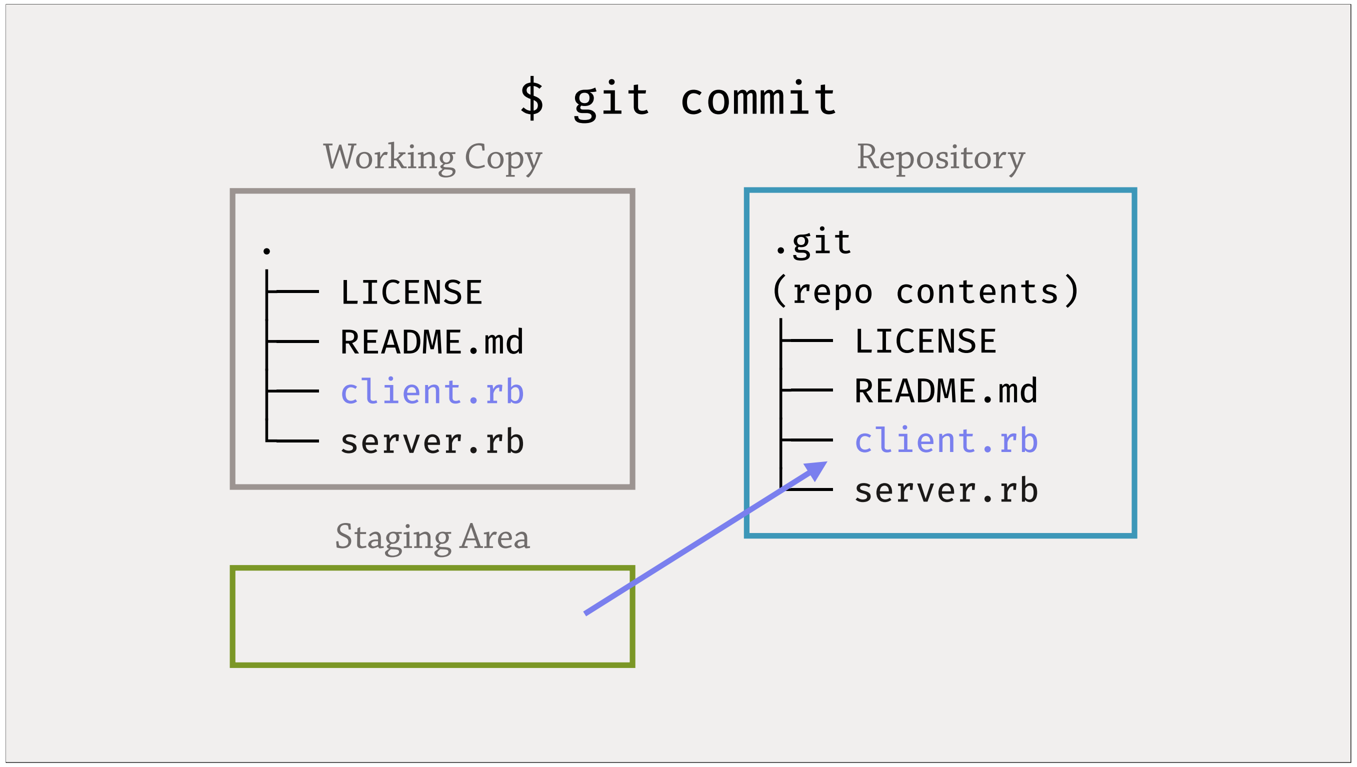
Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Staging Area

```
client.rb
```

We use the `git add` command to tell git about the file, which places it in the staging area.



```
$ git commit
```

Working Copy

```
.
├── LICENSE
├── README.md
├── client.rb
└── server.rb
```

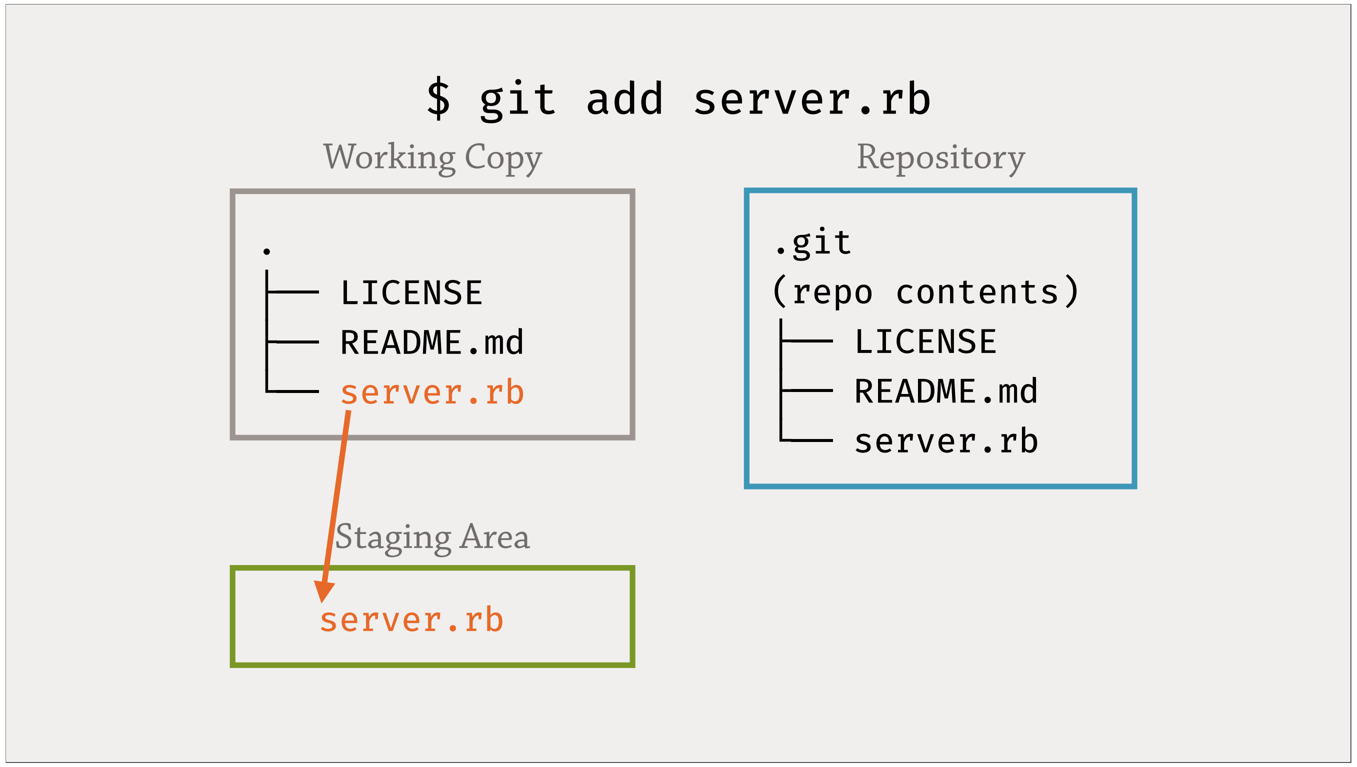
Staging Area

Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
├── client.rb
└── server.rb
```

And when we commit, it’ll add it to the repository. And because you’ve now committed the changes, the staging area is empty again.

This staging area doesn’t really exist anywhere physically, it’s just internal bookkeeping, but you can use it to select what to commit and when.



```
$ git add server.rb
```

Working Copy

```
.
├── LICENSE
├── README.md
└── server.rb
```

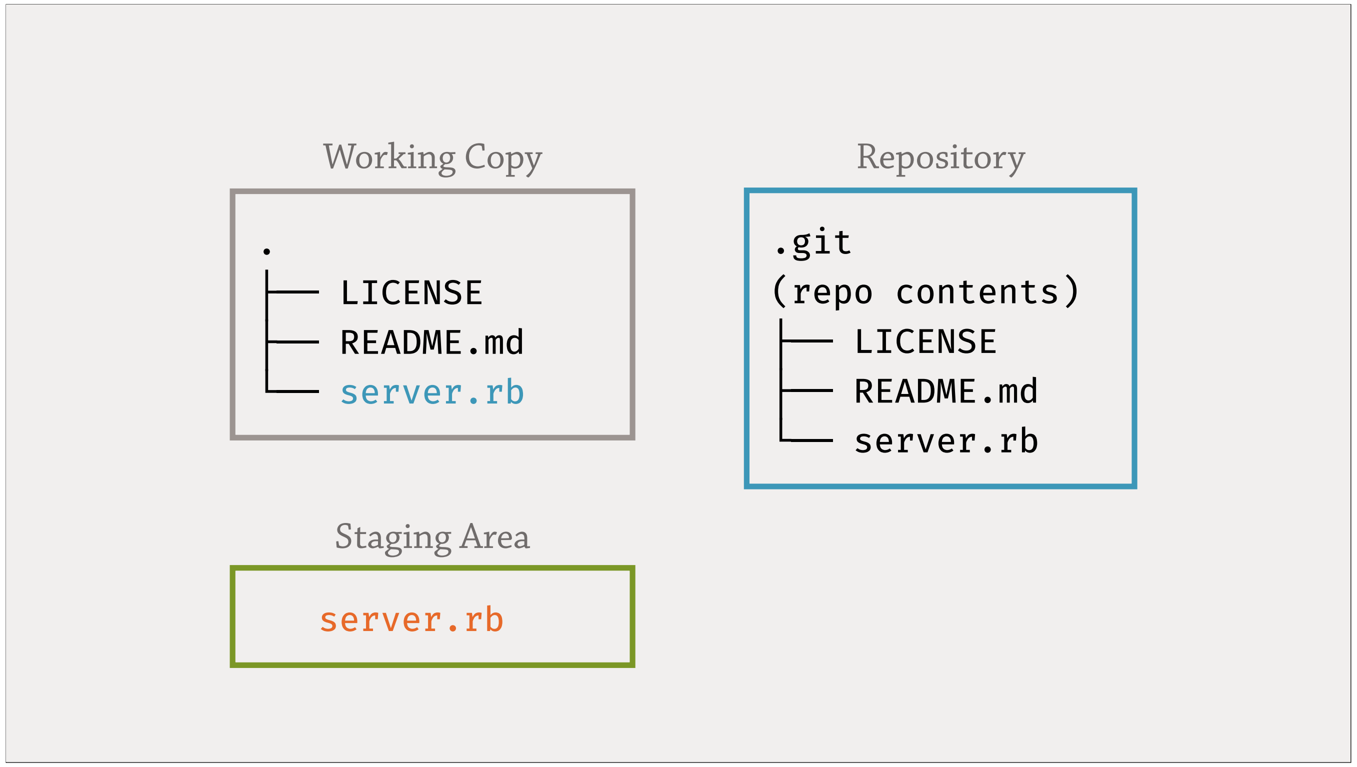
Staging Area

```
server.rb
```

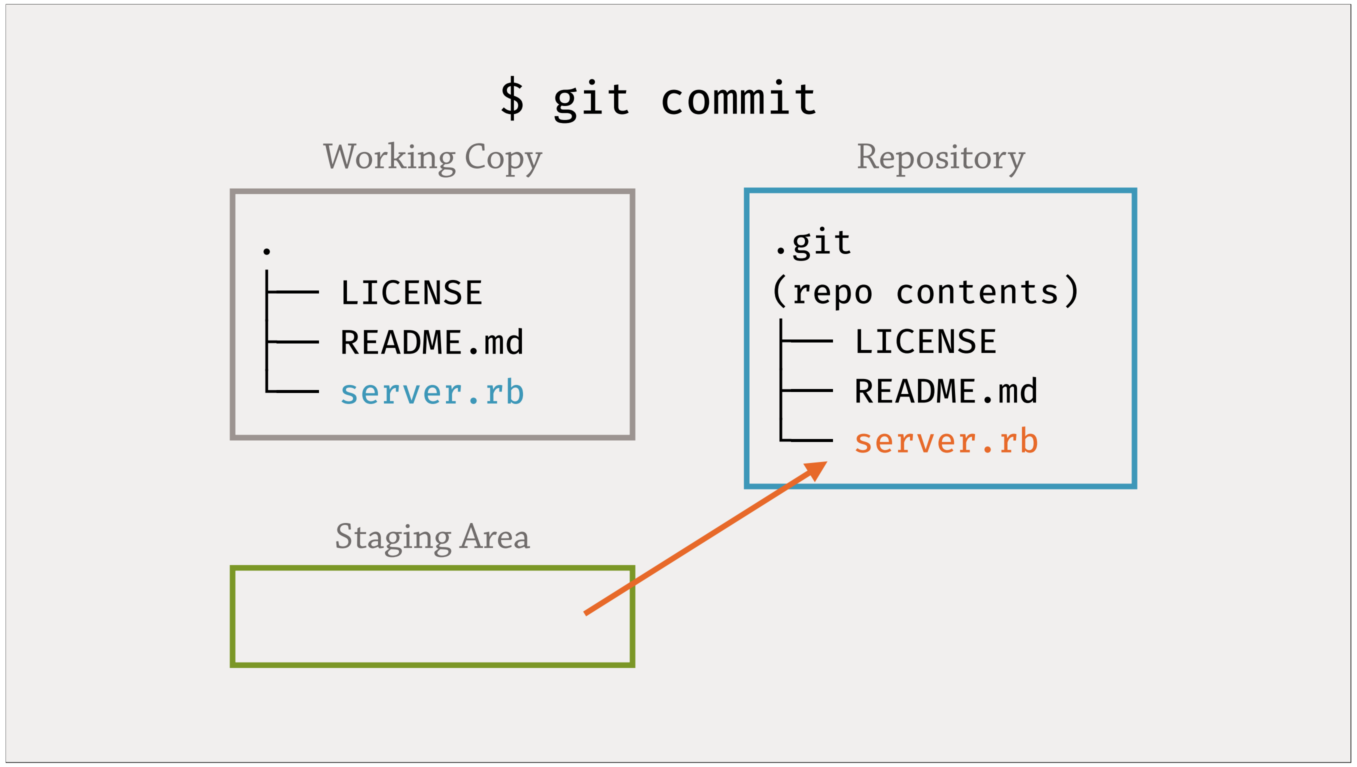
Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Same thing with an edited file. We can stage it with `git add`



Once a version of a file is staged, we can make more changes to the file without affecting what's staged.



```
$ git commit
```

Working Copy

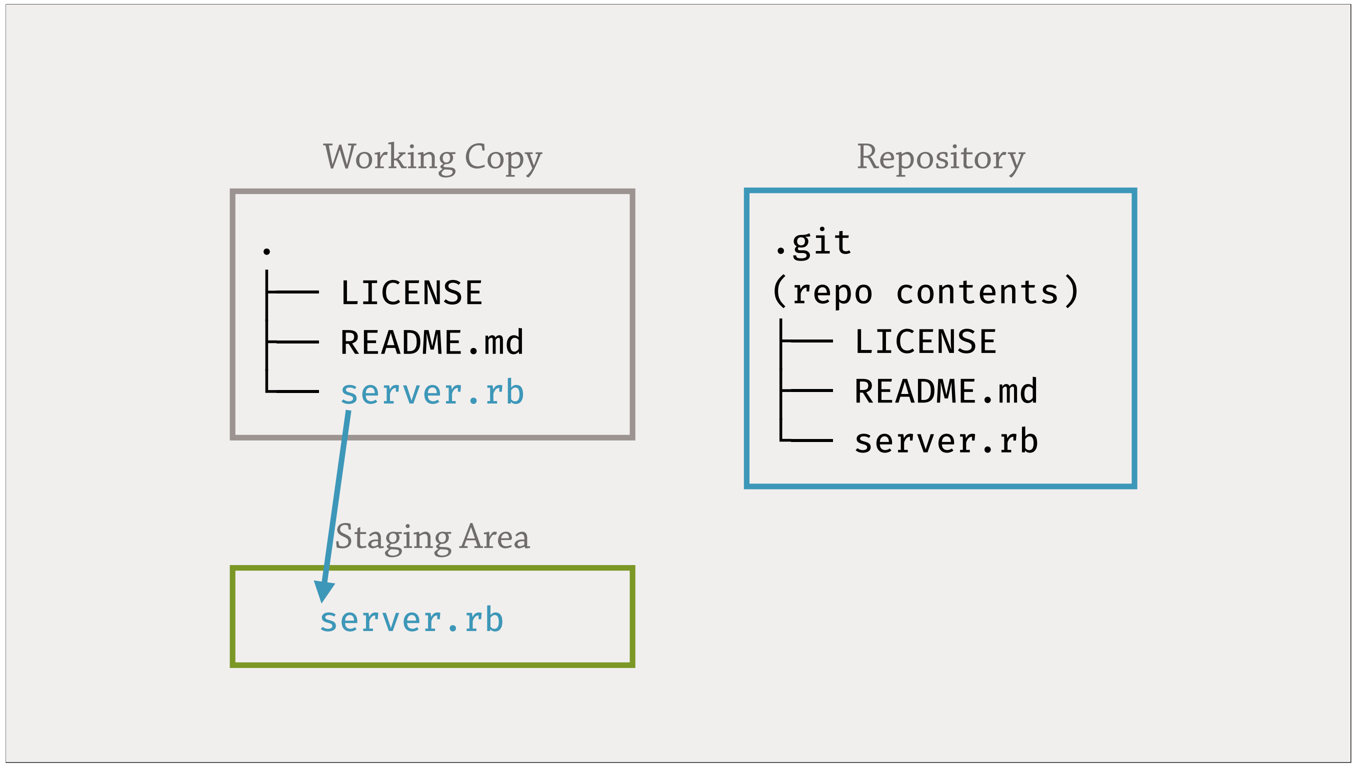
```
.
├── LICENSE
├── README.md
└── server.rb
```

Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Staging Area

But the next commit will only include what's staged. Unless we say commit all the changes, anyway.



Working Copy

```
.
├── LICENSE
├── README.md
└── server.rb
```

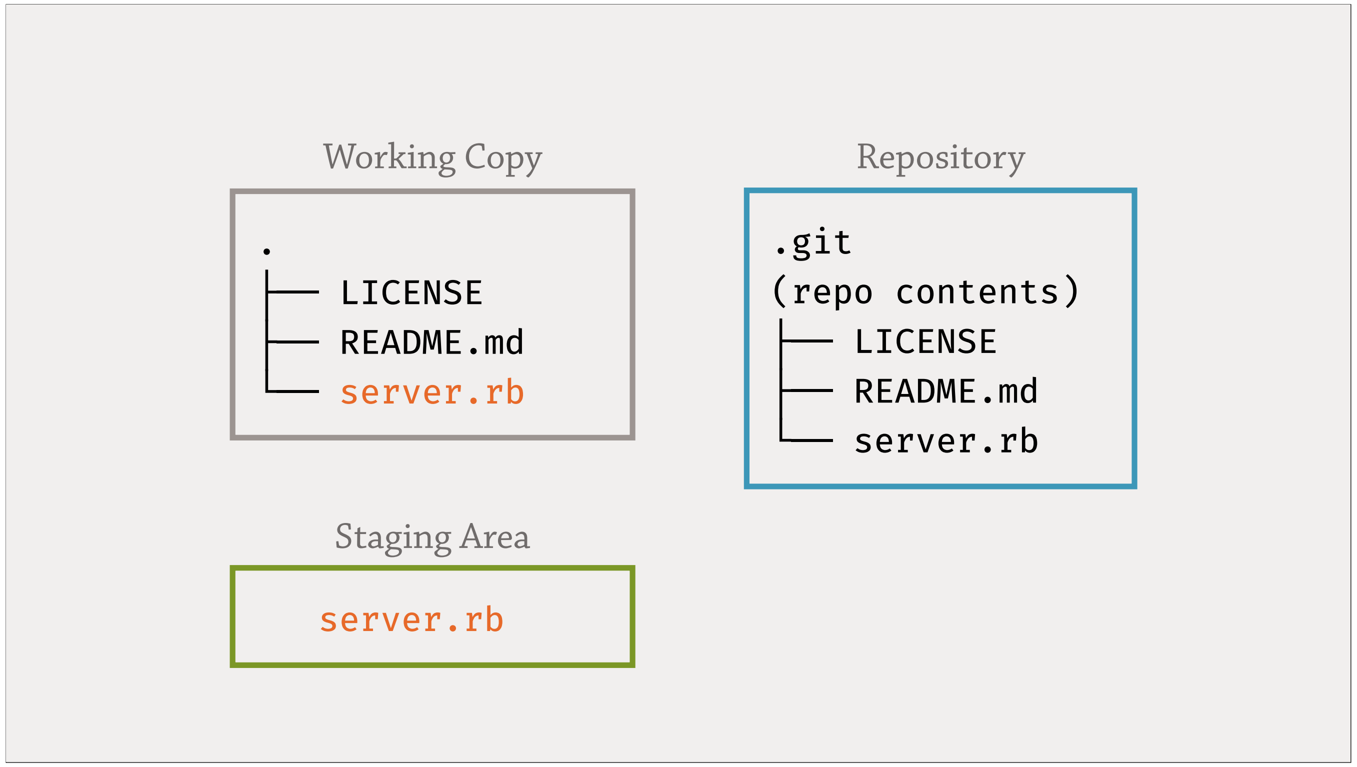
Staging Area

```
server.rb
```

Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Or, we can stage those new changes before committing.



Working Copy

```
.
├── LICENSE
├── README.md
└── server.rb
```

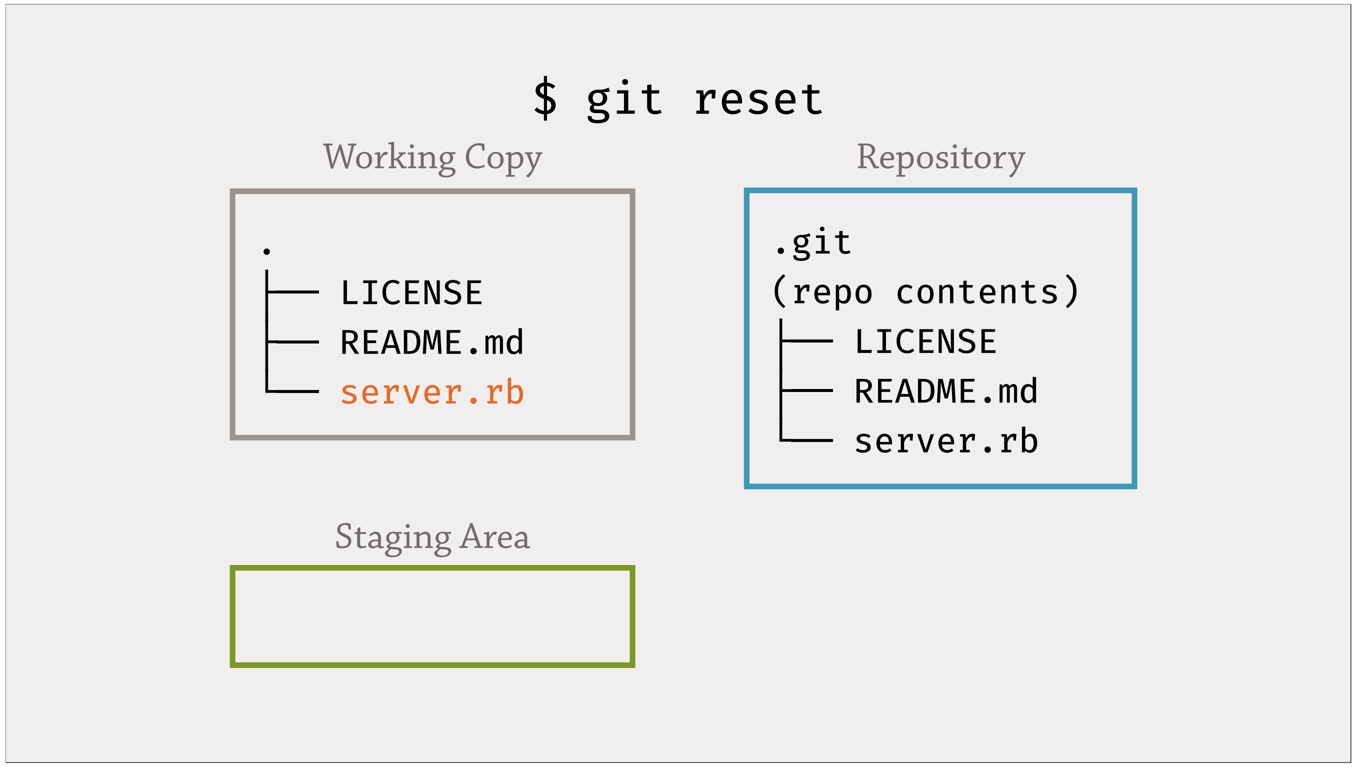
Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Staging Area

```
server.rb
```

Now, let’s say we’ve staged the server file for commit, but we want to back out. We can get things _out_ of the staging area with another command, `git reset`.

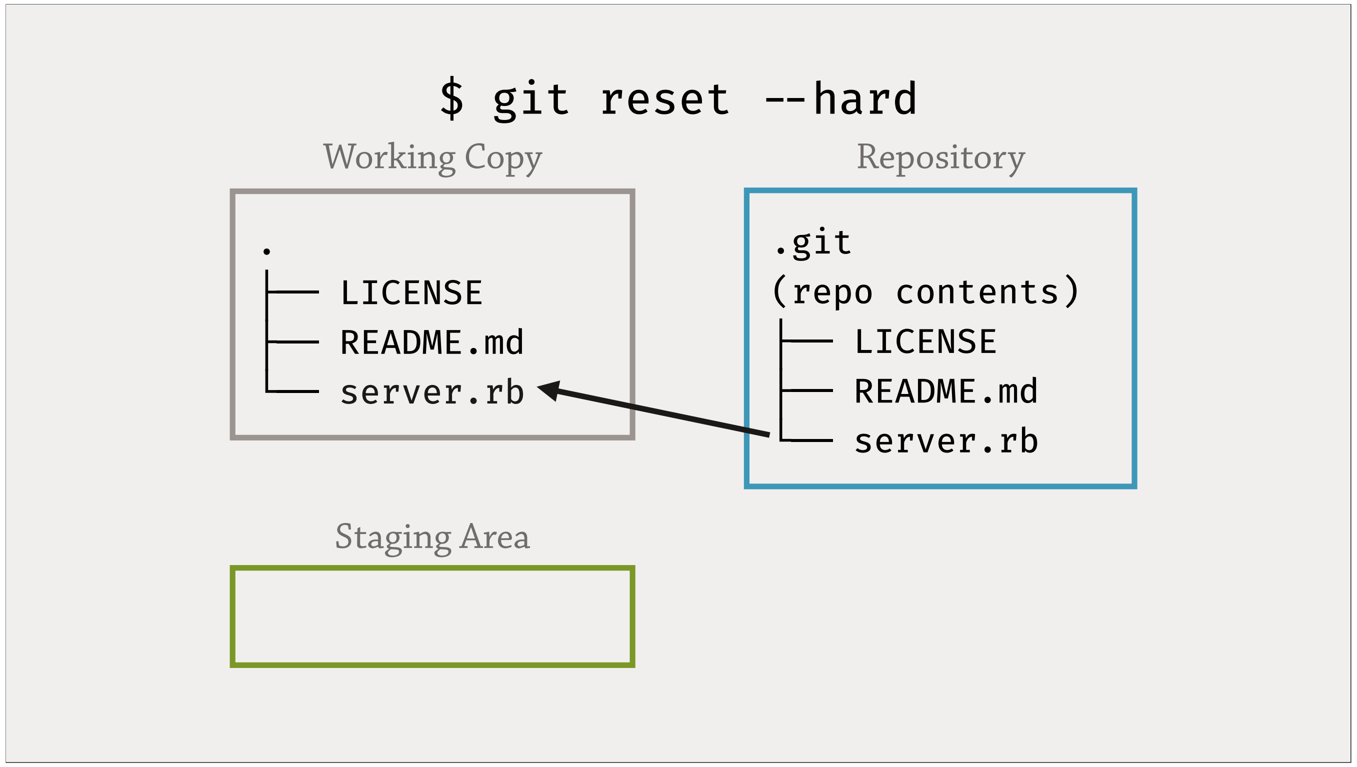


```
$ git reset
```

```
.
├── LICENSE
├── README.md
└── server.rb
```

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

That resets the staging area.



```
$ git reset --hard
```

Working Copy

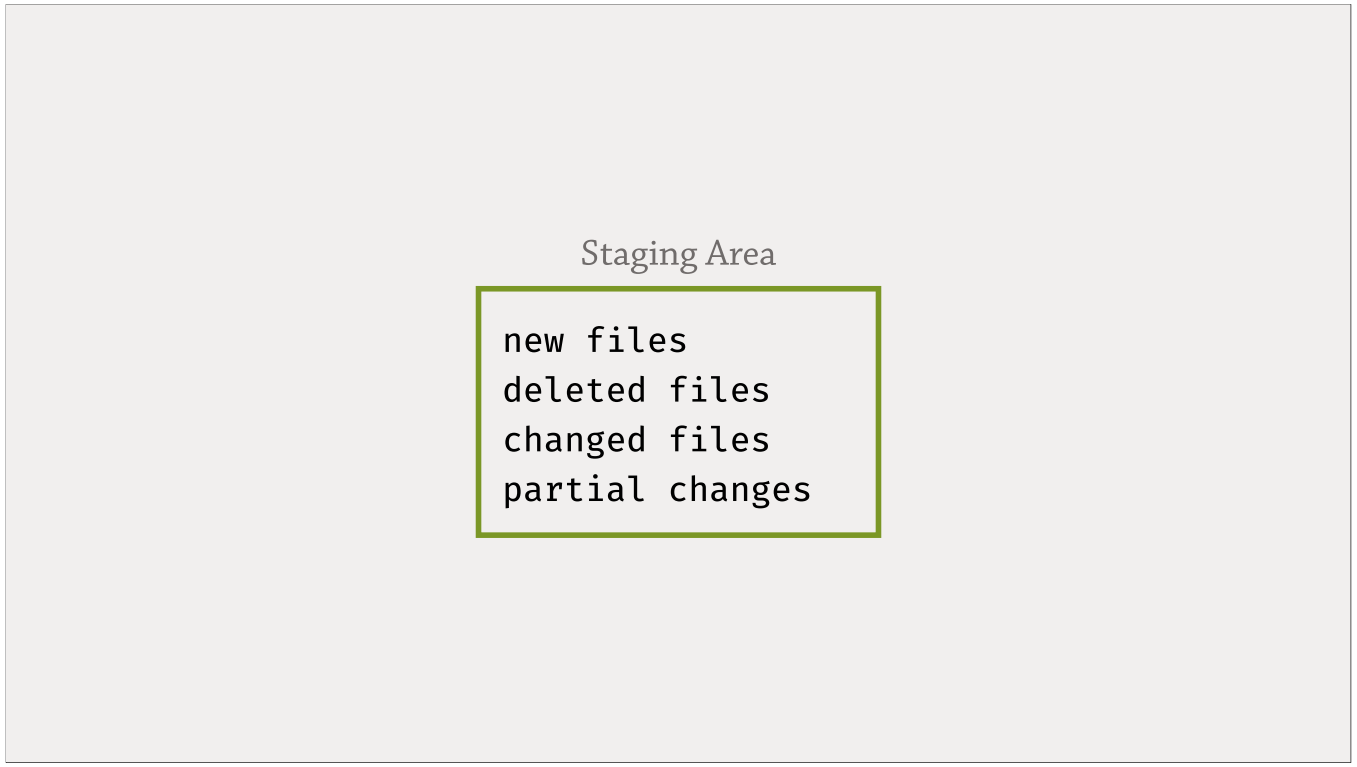
```
.
├── LICENSE
├── README.md
└── server.rb
```

Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Staging Area

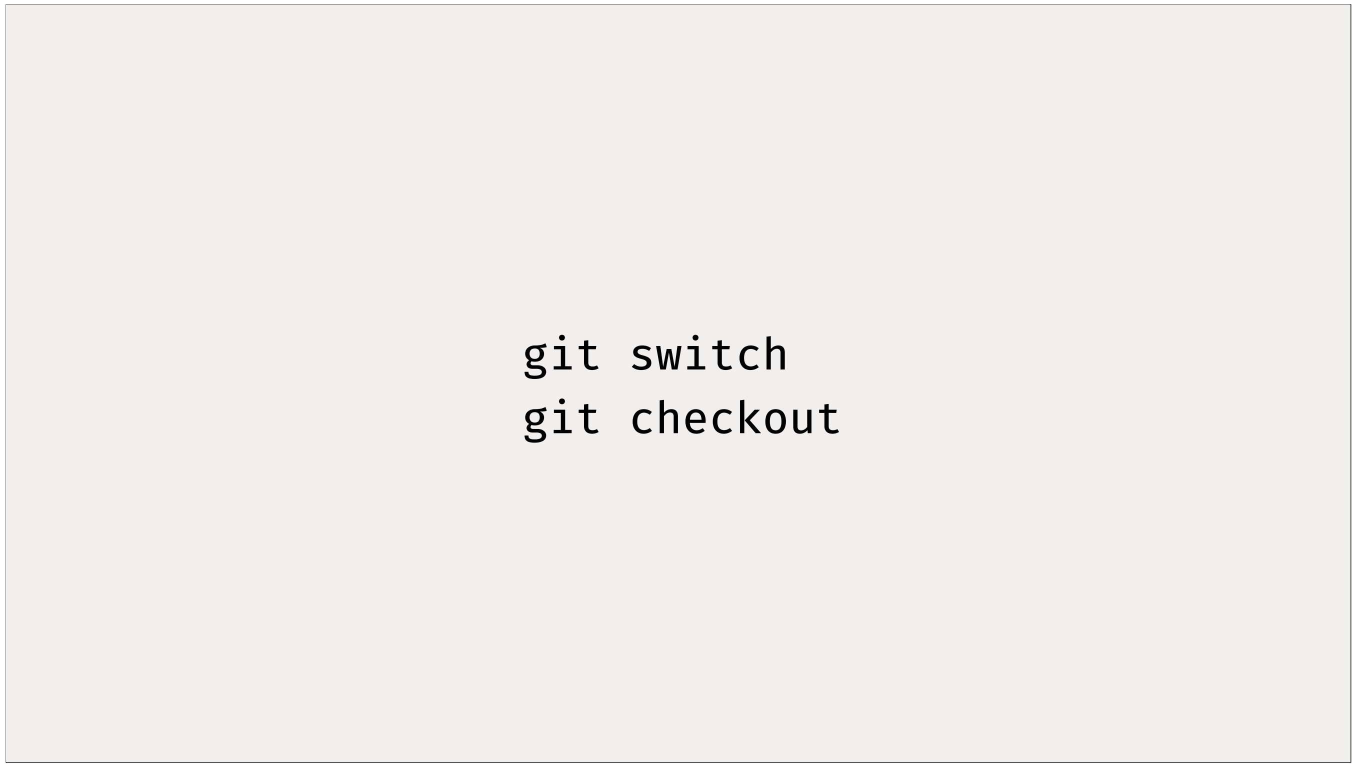
If we ask using the dangerous `--hard` flag, we can reset the working copy, too.



Staging Area

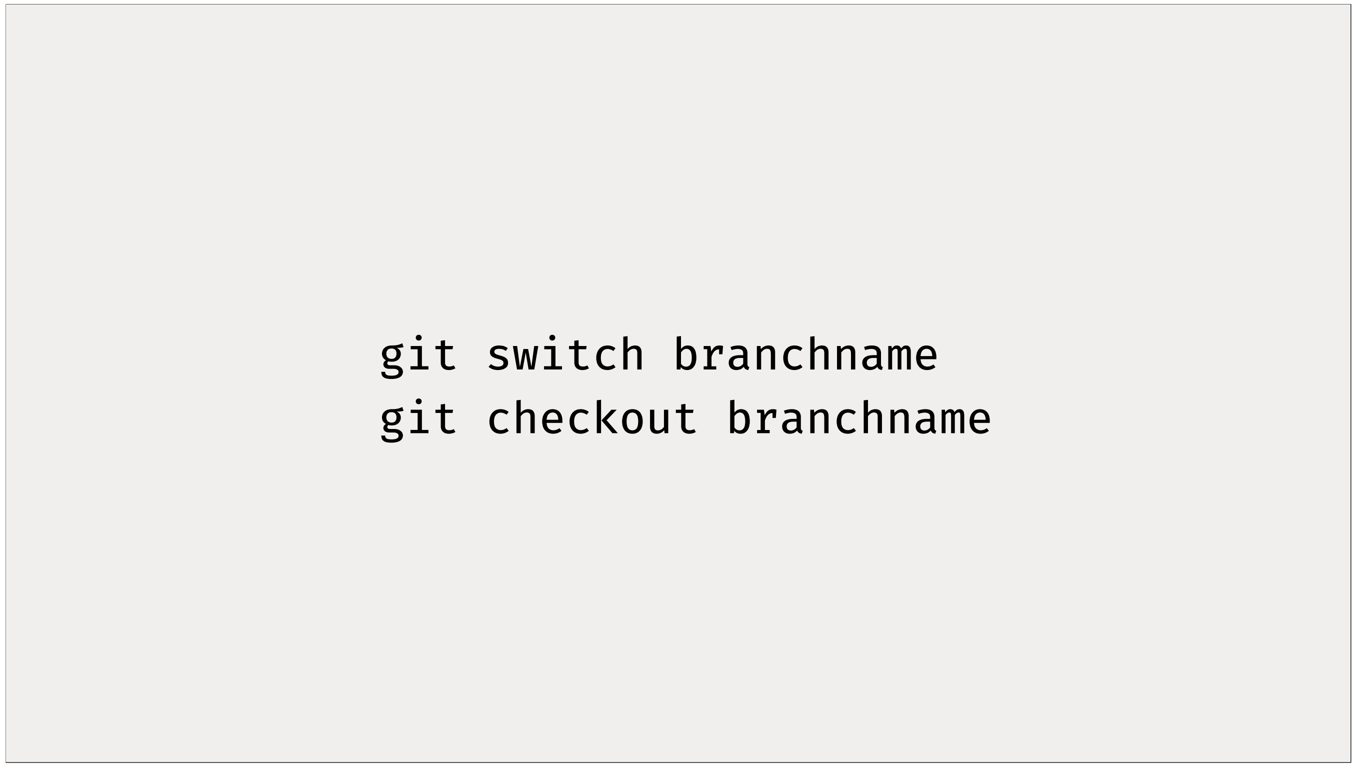
```
new files
deleted files
changed files
partial changes
```

The staging area is useful for more than just adding files. You can stage new files, deleted files, changed files… but also you can stage *parts* of files. For example, if you made a bug fix and added a new feature, you can stage just the changes for the bug fix and commit that, then stage and commit the feature separately. This is the `--patch` flag, something for you to look up later.



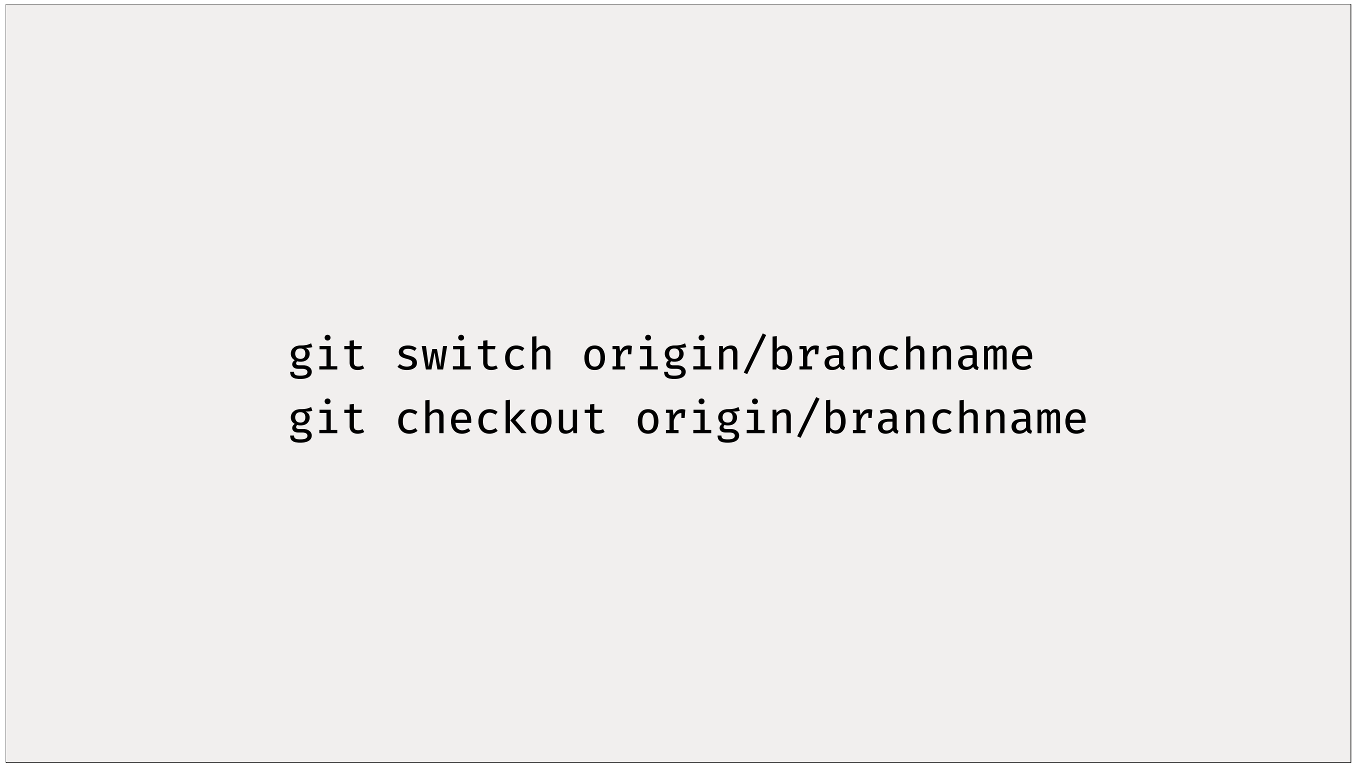
```
git switch
git checkout
```

Let’s talk about git switch and git checkout.



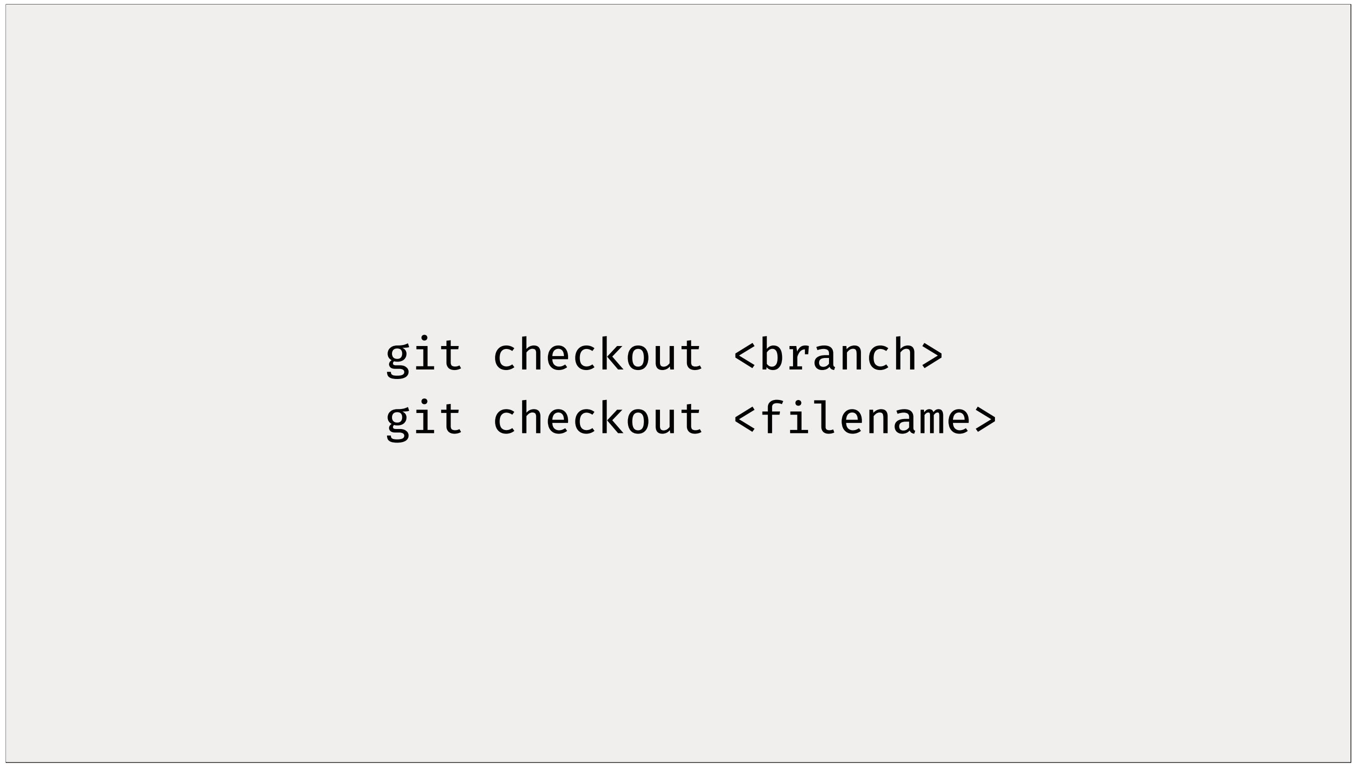
```
git switch branchname
git checkout branchname
```

They both do one thing in common: switching between branches. This updates your working copy to whatever’s in that branch.
Note that this will not overwrite any modifications you’ve made. If it would, git will complain and not let you.



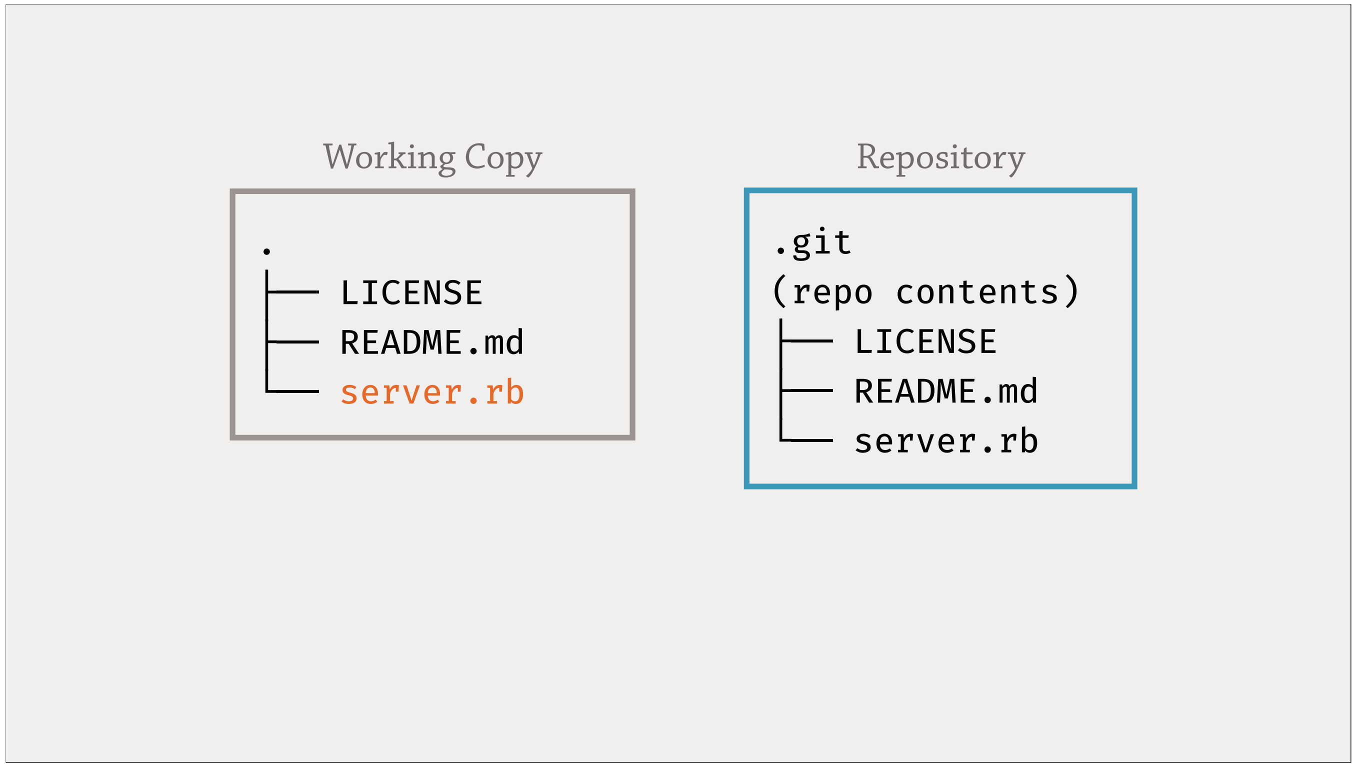
```
git switch origin/branchname
git checkout origin/branchname
```

You can also check out remote branches.



```
git checkout <branch>
git checkout <filename>
```

Checkout can switch your working copy between different branches, but it can also retrieve the contents of files.



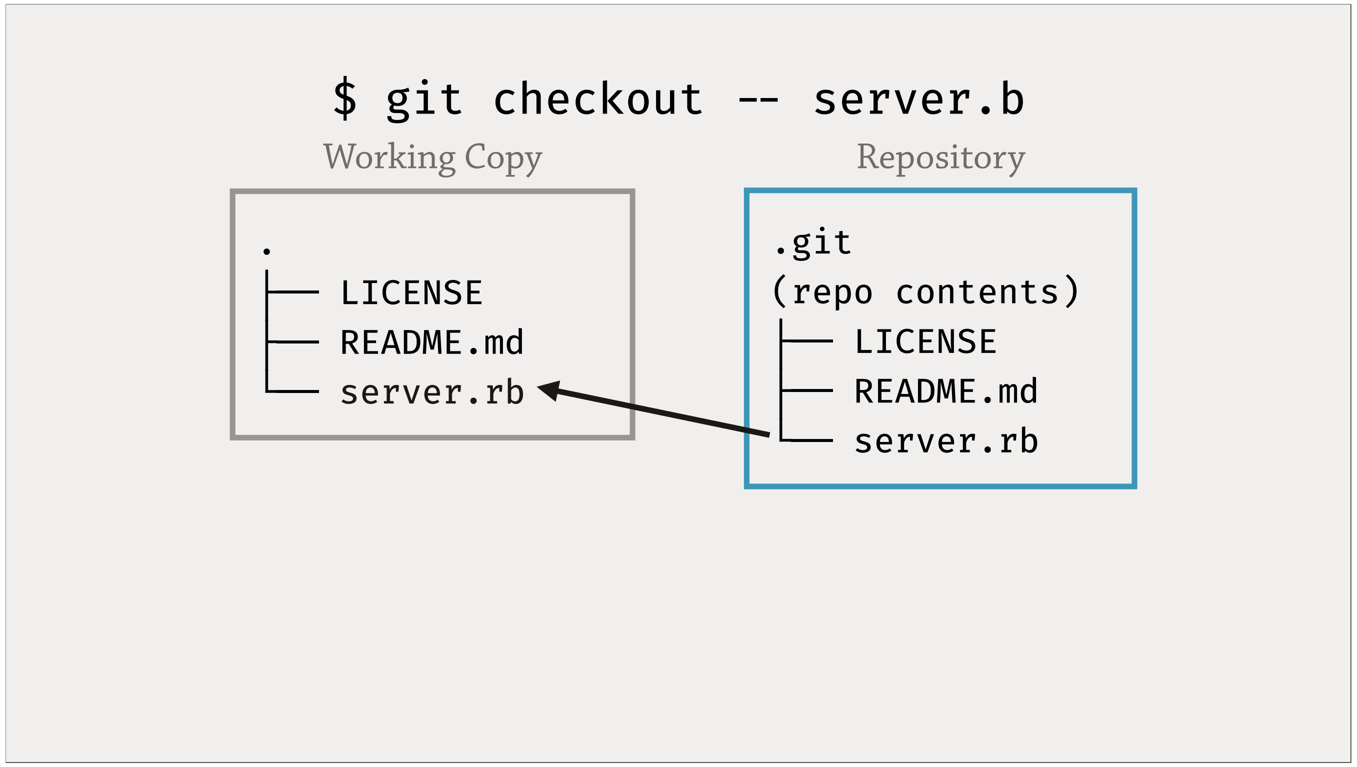
Working Copy

```
.
├── LICENSE
├── README.md
└── server.rb
```

Repository

```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

Let’s say you changed a file, but want to get the original back.



```
$ git checkout -- server.b
```

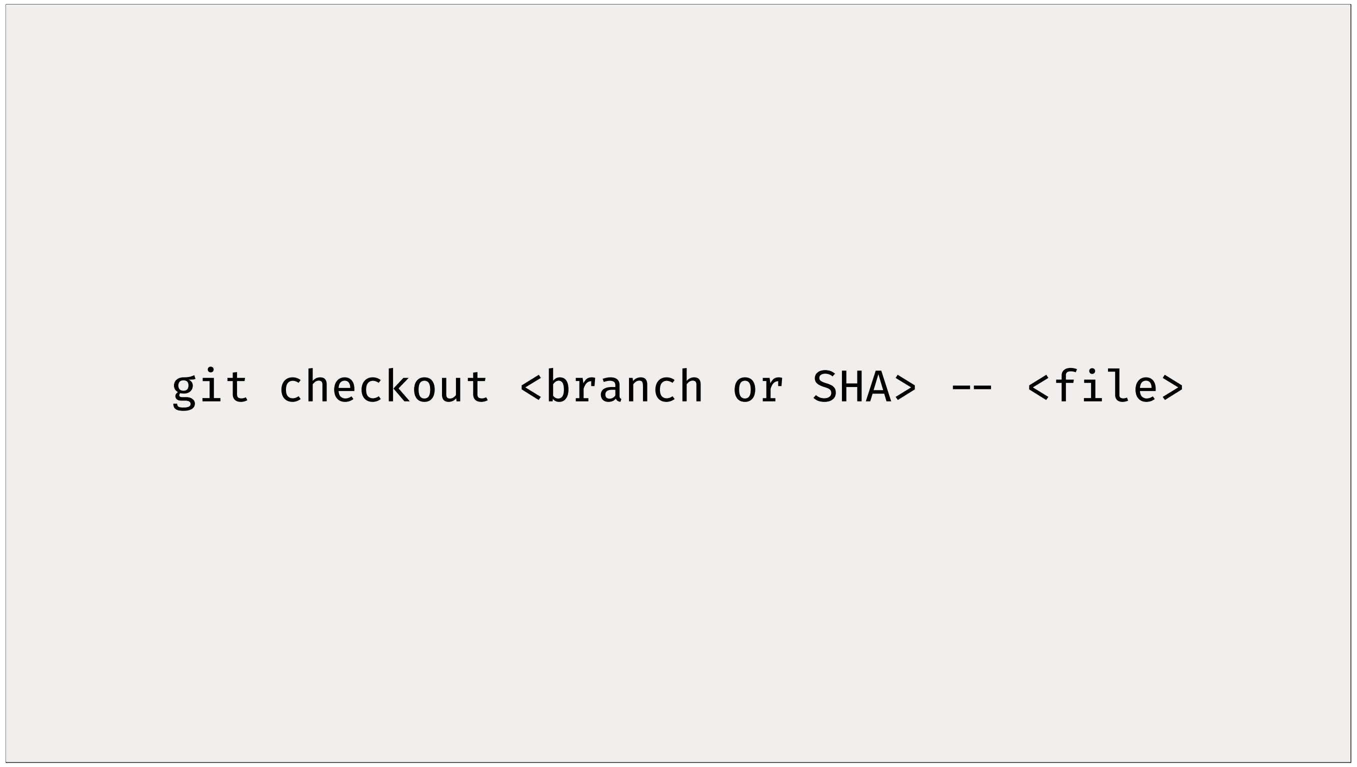
Working Copy

```
.
├── LICENSE
├── README.md
└── server.rb
```

Repository

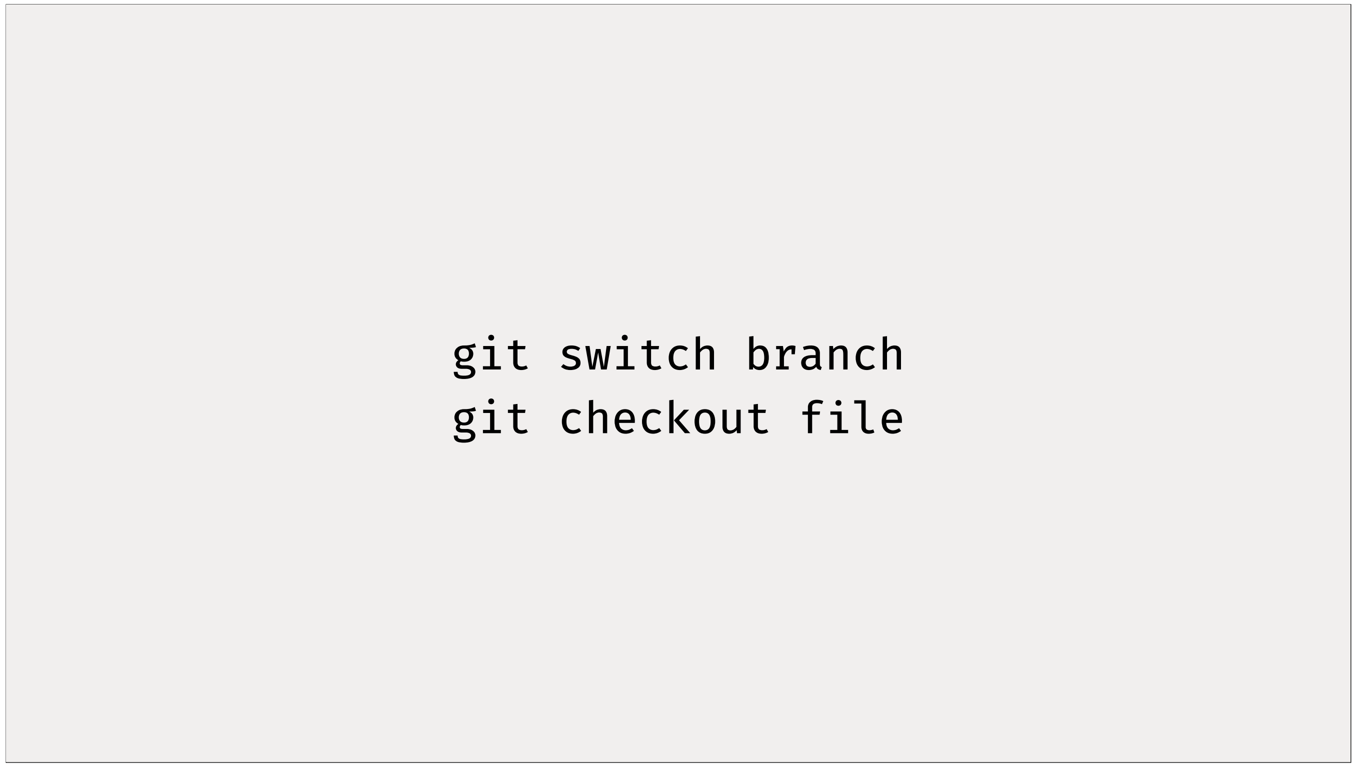
```
.git
(repo contents)
├── LICENSE
├── README.md
└── server.rb
```

When you say “git checkout”, you’re asking git to get the most recently committed version of that file and overwrite what’s in your working copy. This is probably the easiest way to lose work if you’re not paying attention, but it’s also an easy way to get back to a clean slate. The double dash isn’t required but is an extra signal to git that you mean to specify a file path, not a branch. This works a lot like reset --hard.



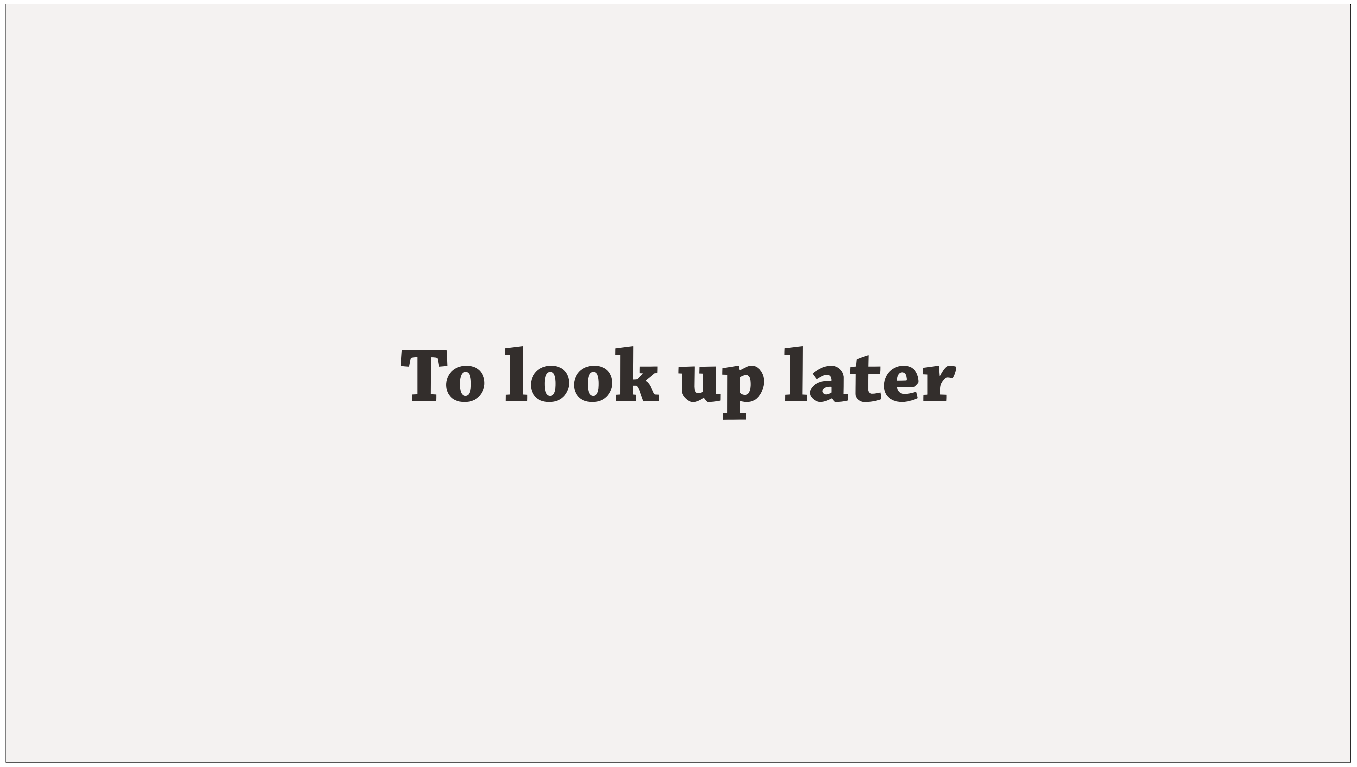
```
git checkout <branch or SHA> -- <file>
```

You can get the contents of any committed file from any branch or commit this way too.



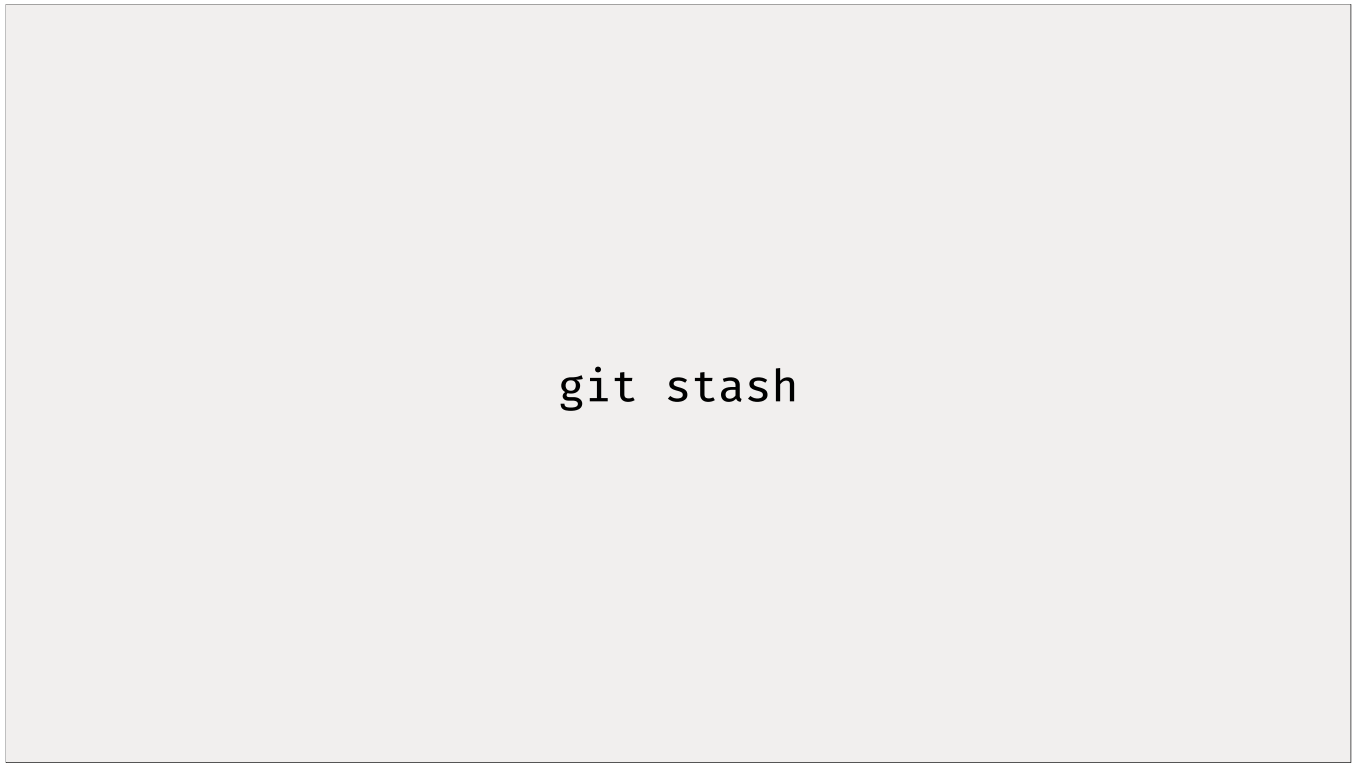
```
git switch branch
git checkout file
```

You’ll see “git checkout” used in a lot of older git tutorials for switching branches. That’s because the switch command was only added recently. Just to reduce confusion, I recommend using switch for branches and checkout for files.



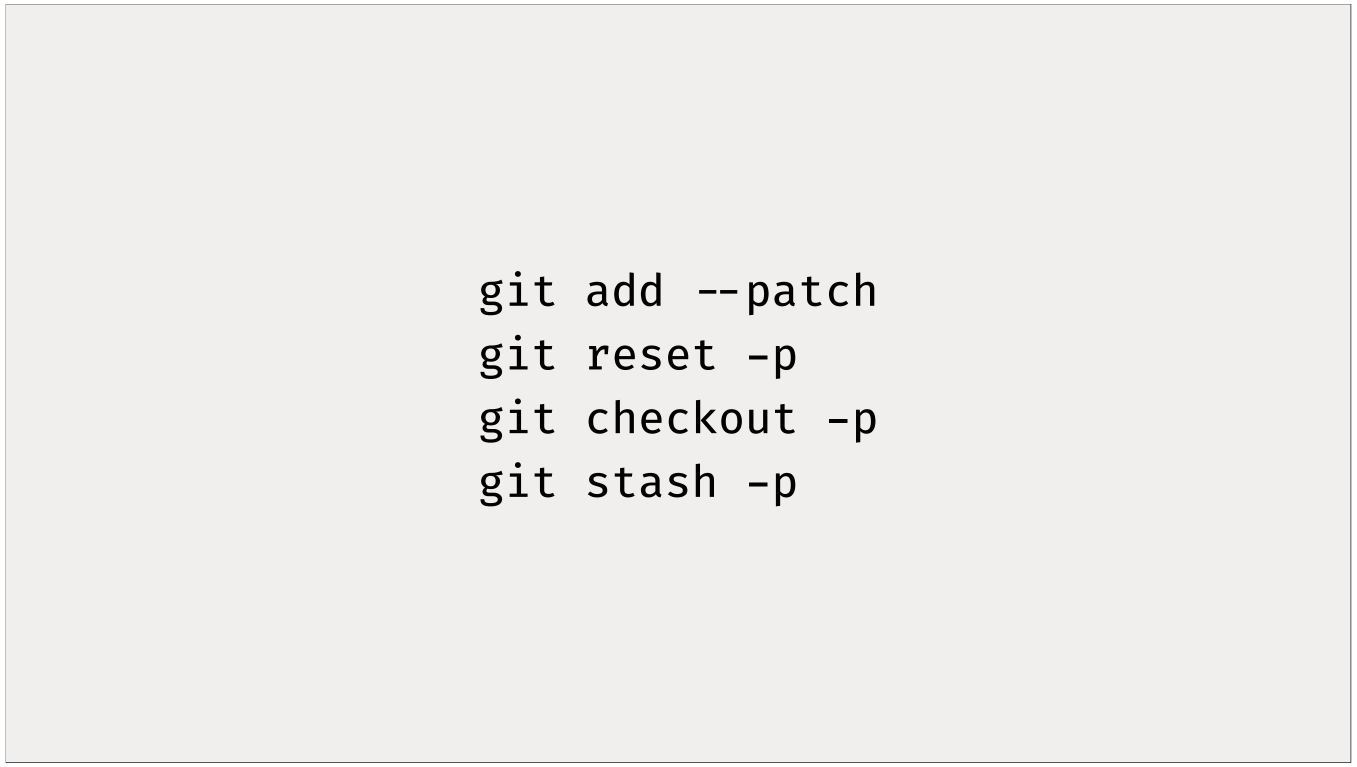
To look up later

And here’s a couple more things to look up later. I’ll give a one-line overview but you can look up documentation and examples if you want.



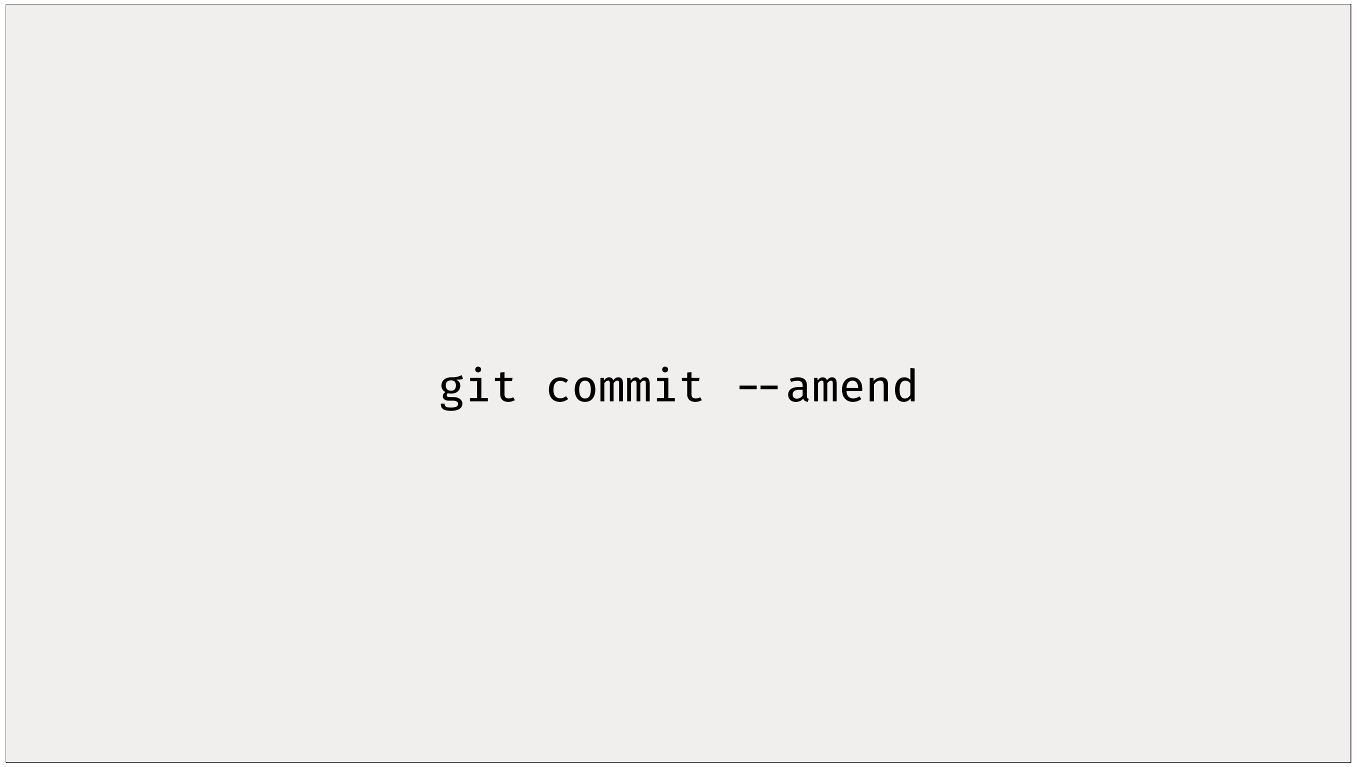
```
git stash
```

The `git stash` is a way for setting aside work in progress that you haven’t committed yet. I’ll demo that in a minute.



```
git add --patch
git reset -p
git checkout -p
git stash -p
```

the -p or patch flag for add, reset, and stash. This lets you pick line by line what to add, reset, checkout, or stash.



```
git commit --amend
```

You can amend a commit, essentially redoing it

```
git rebase --interactive
```

And you can do an “interactive” rebase, which lets you pick and choose which commits to keep, modify, combine, or discard during a rebase.

```
git reflog
```

And one more thing: the reflog. Git keeps track of every change to a branch that you make and tracks that in a log. You can use this log to undo changes or sometimes even get work back. It’s an advanced feature, gets into the git internals, but can be helpful as a fallback in case things get really messed up in your repository.

GitHub

Now let’s talk about GitHub.

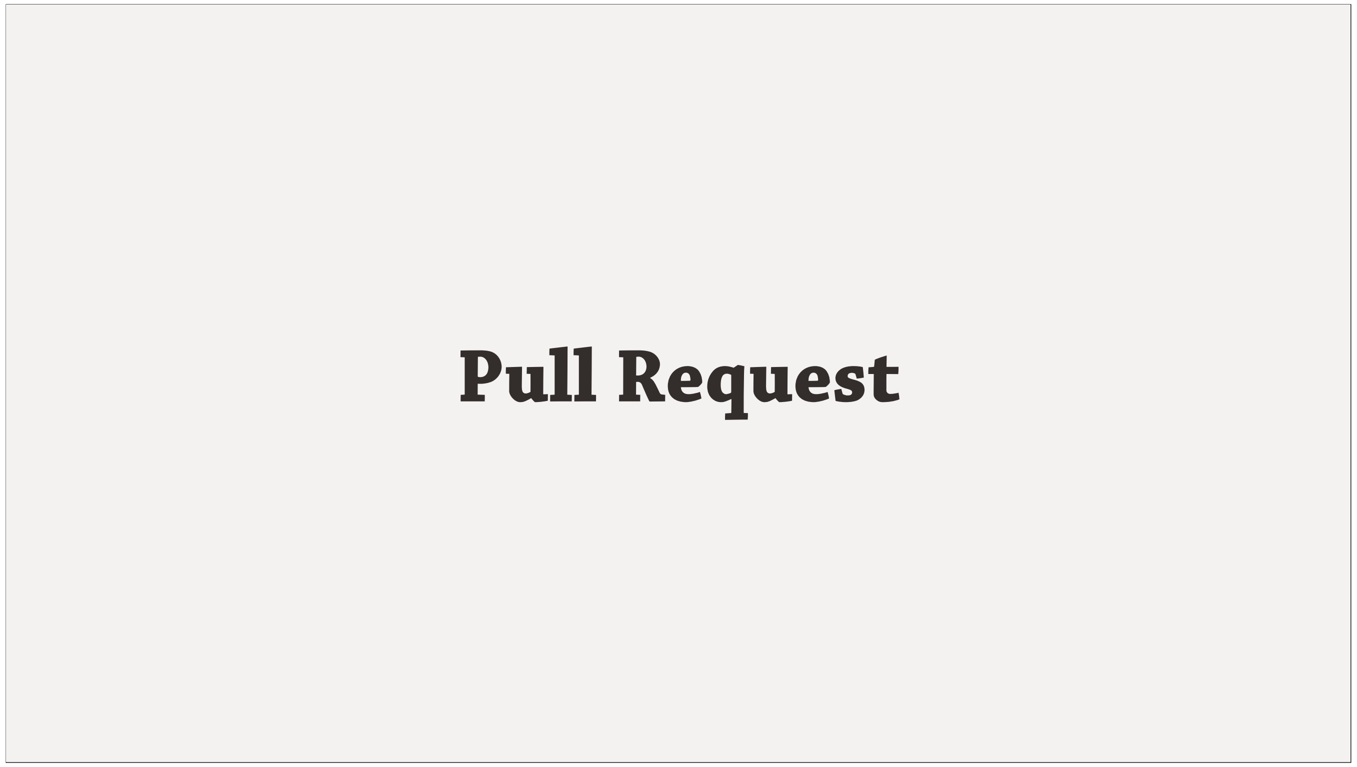
Hosting

GitHub, first, is a place to put your code. Public or private, repositories are free.

Collaboration

But what GitHub really does well is collaboration. It gives you the tools on top of git repositories to discuss, review, and work on code together.

* Issues for filing bugs
* Pull requests for proposing and discussing changes to the code
* Wikis for community-edited documentation
* Project boards for kanban-style project tracking
* Organizations for managing teams of contributors and repository permissions



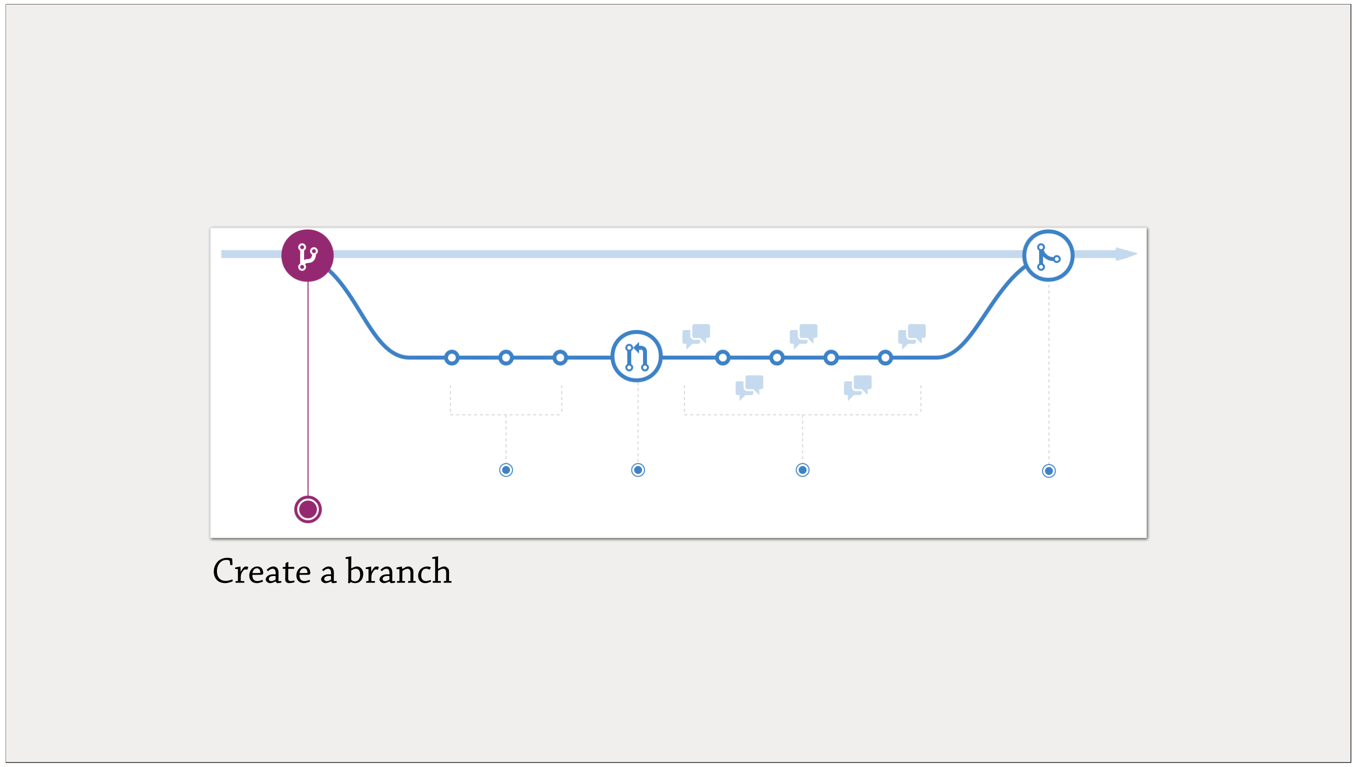
Pull Request

In my opinion, the central defining feature of GitHub is the pull request. This is a way to propose, discuss, and integrate changes into your codebase

Branching Strategies

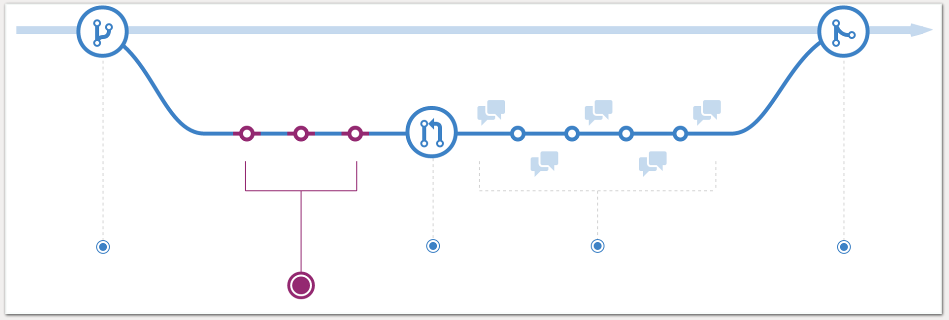
GitHub Flow

I want to talk about what we call the “GitHub Flow”. This is the basic process we use for developing everything at GitHub.



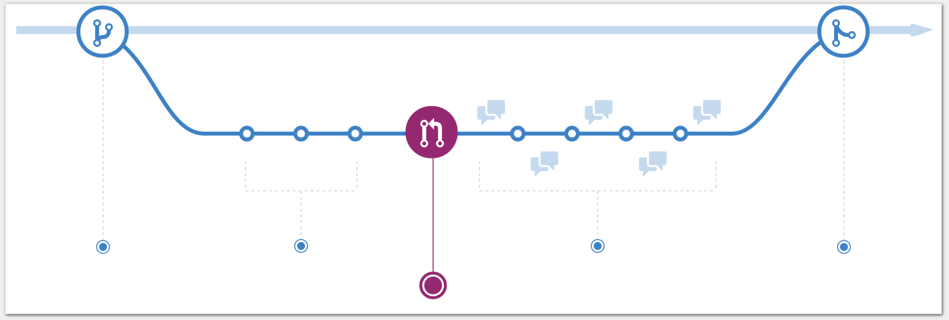
Create a branch

It starts with a branch. You create a branch to work on some new feature, or a bug fix, or whatever that may be.



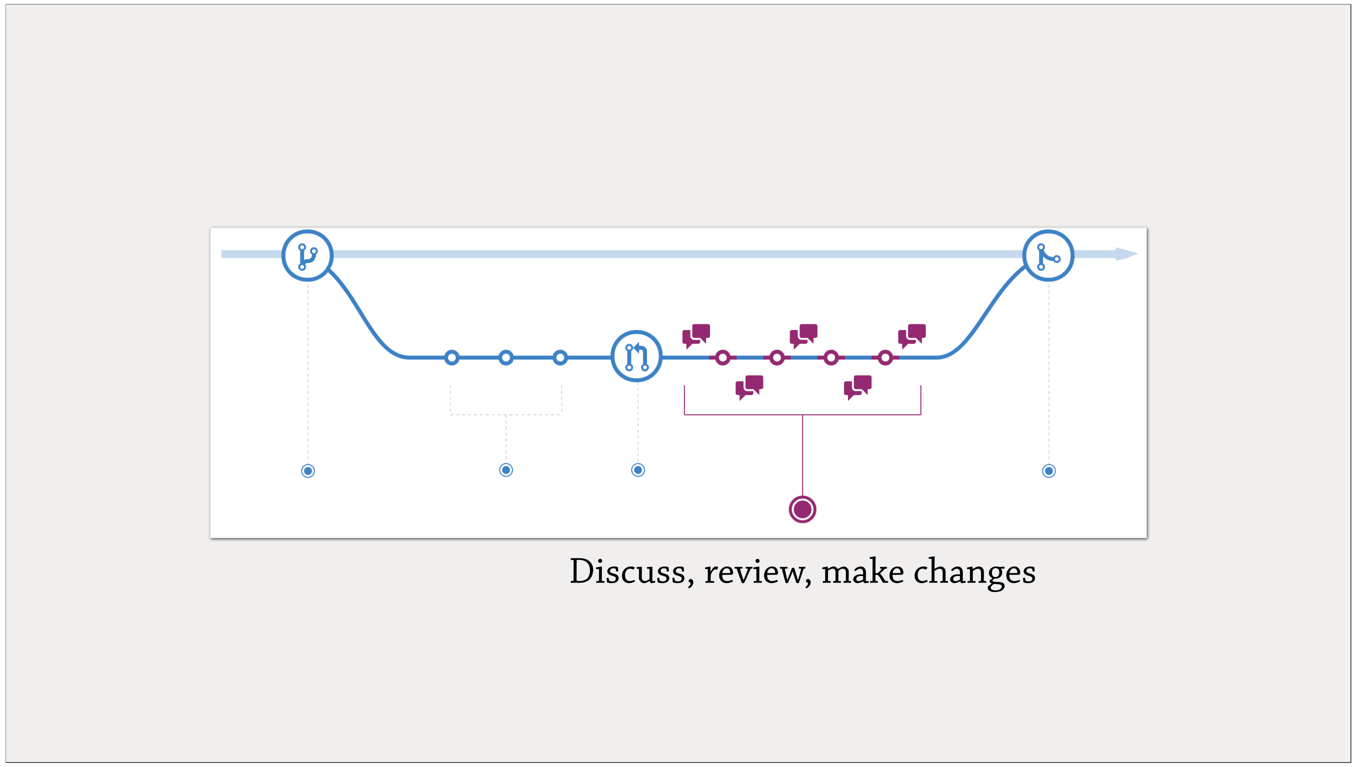
Add commits

Then you'll do some work, add a feature, fix a bug, whatever.



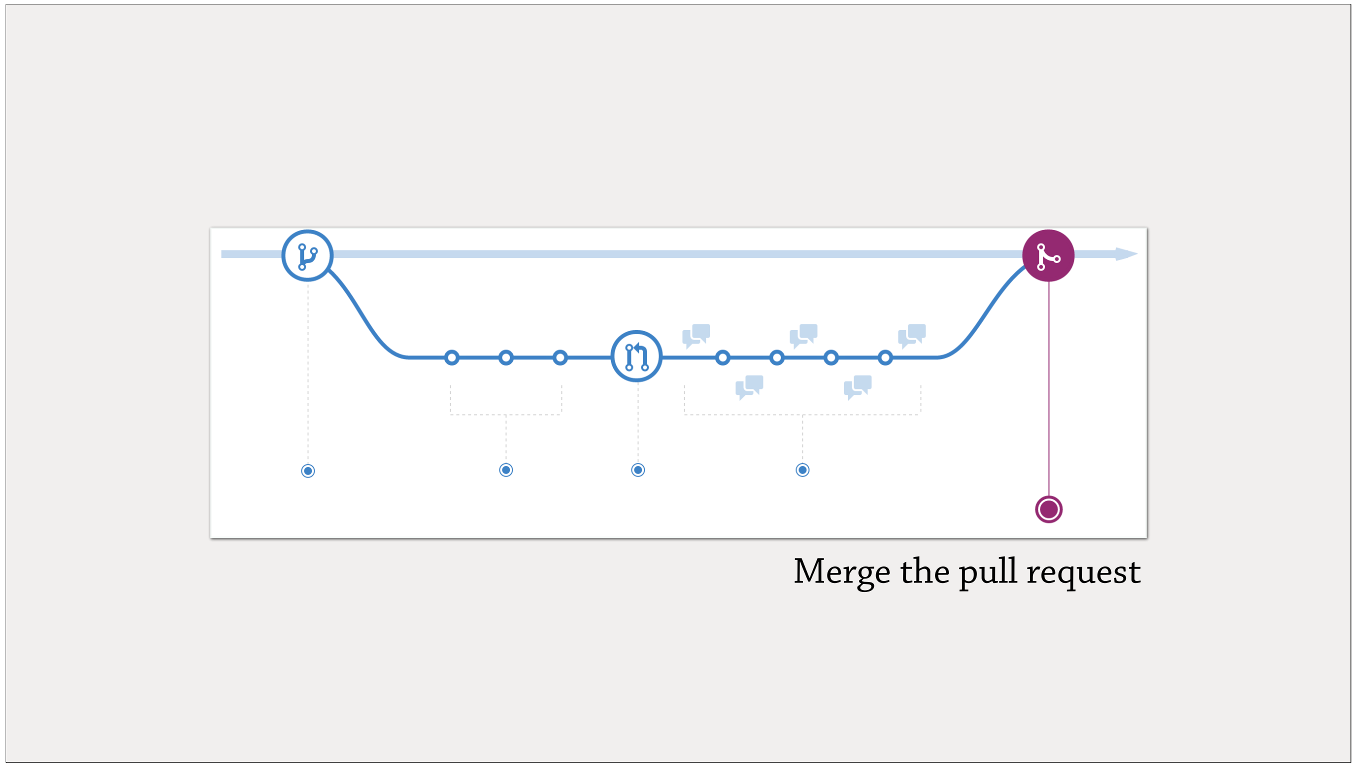
Open a pull request

And then, you'll open a pull request. This says to everyone you're working with: "here are some changes, I want to merge these into the codebase"



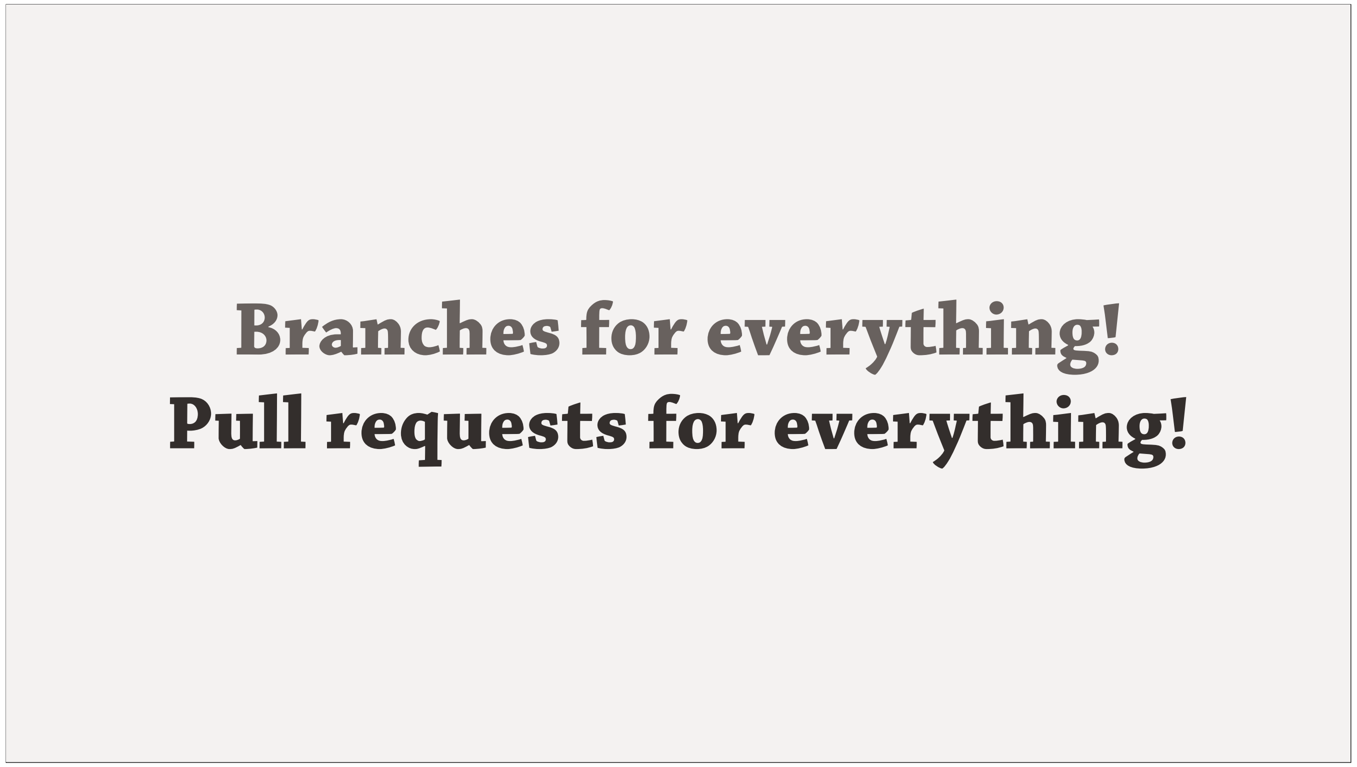
Discuss, review, make changes

Now comes the important part. Everyone can review and discuss the code. You can talk about it and make changes based on feedback until you and your team and your robots (like automated build tools) are happy with it.



Merge the pull request

Then, when it’s ready, you can merge in the code.



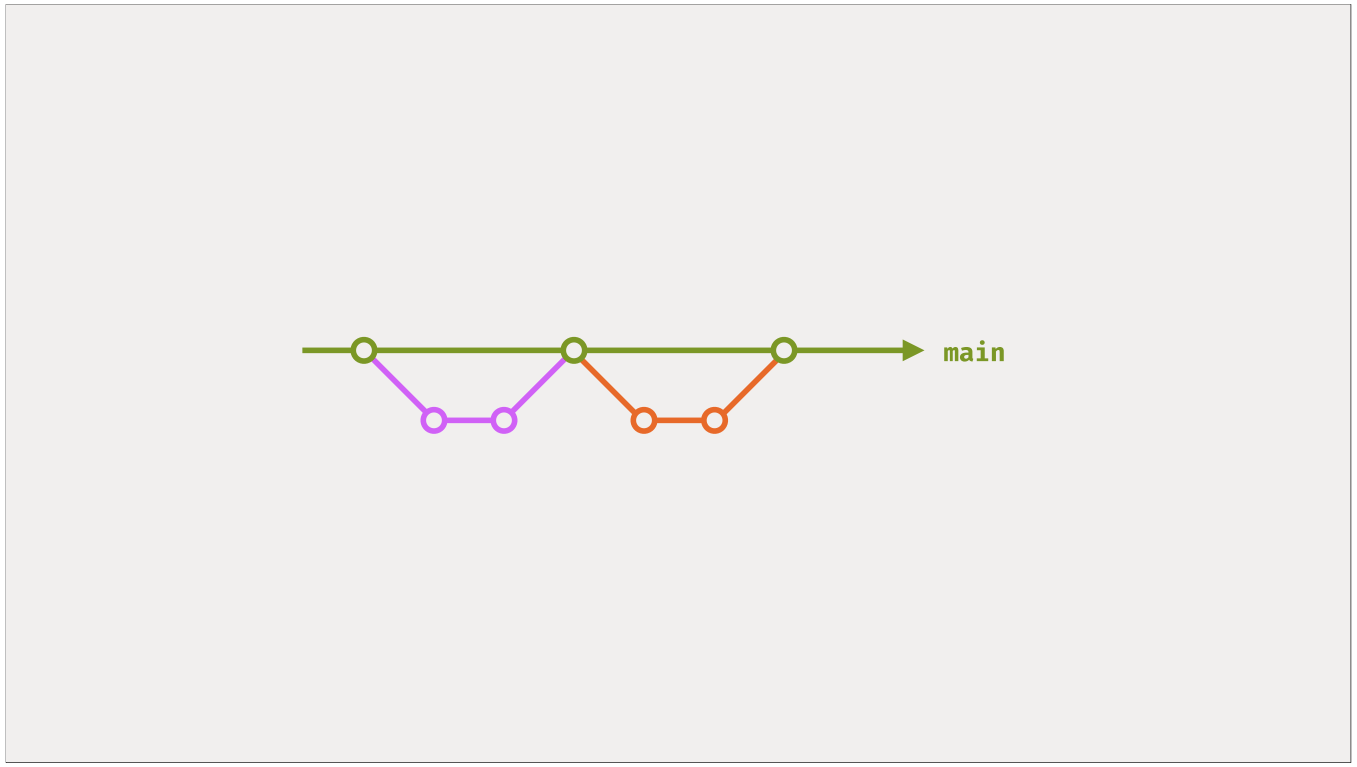
Branches for everything!

Pull requests for everything!

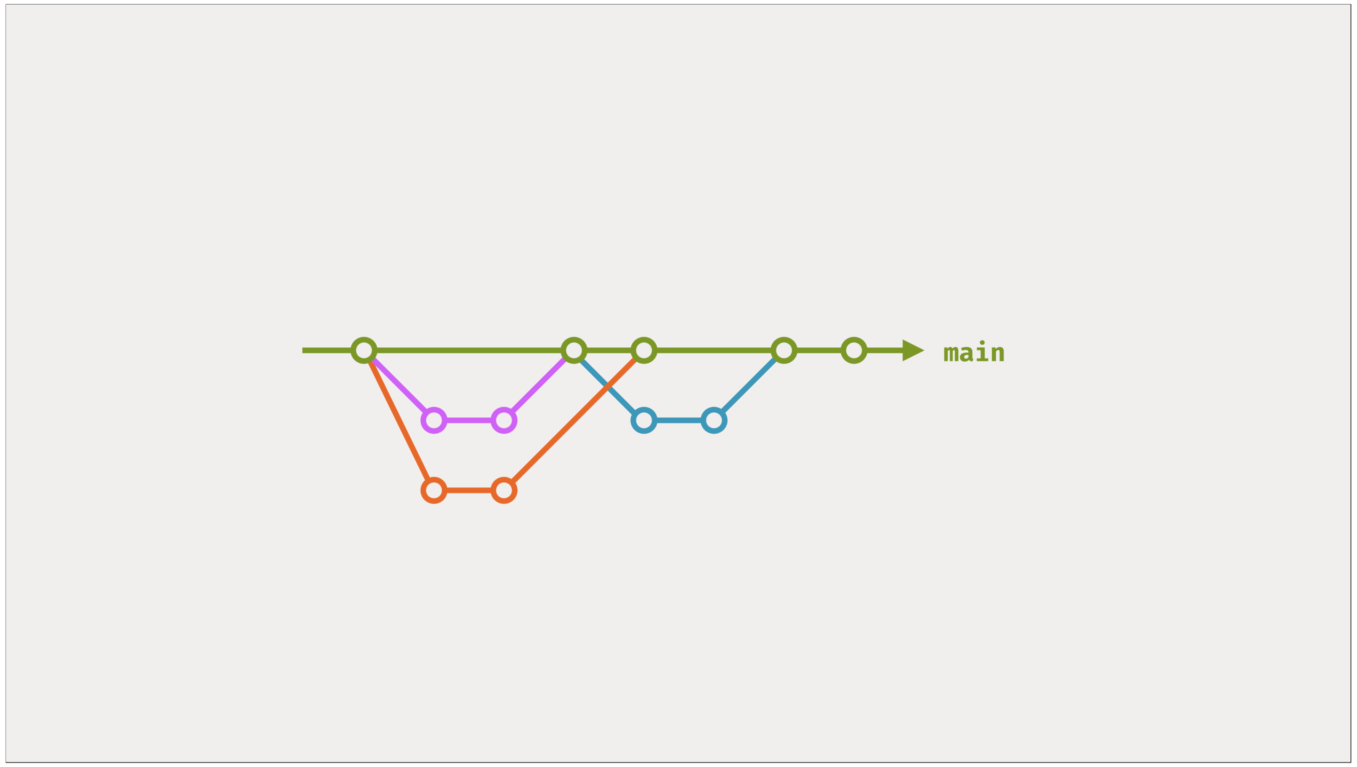
Just like we can and should use branches for everything, everything we change in our codebases is done through a pull request.

Strategies

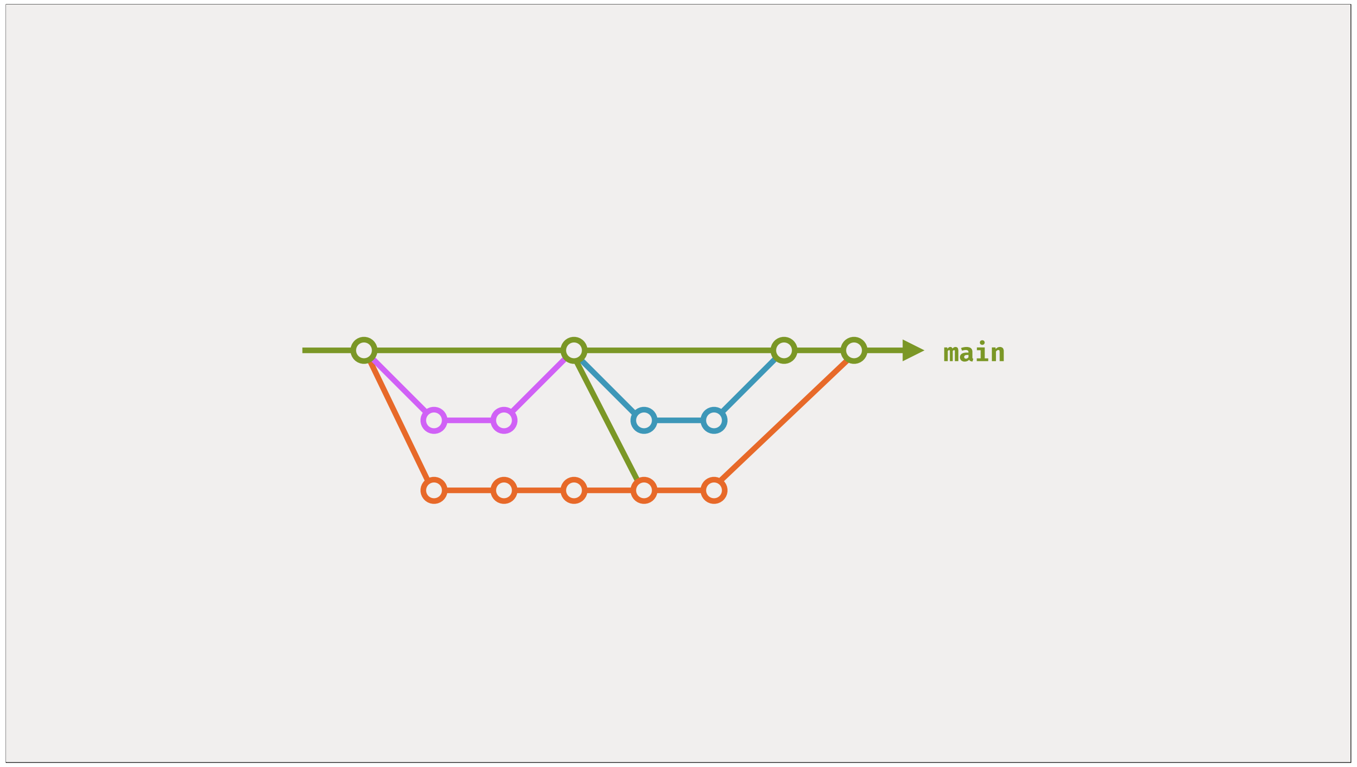
Git doesn’t place any restrictions on how you use branches, it’s entirely up to you and the people you work with. This becomes much more a matter of opinion and tradeoffs with complexity, whether your deployment or test tools support a workflow, etc.



I just mentioned the GitHub flow. This is based on a single main branch, with feature branches merged back in when they’re done. No one ever commits directly to the main branch.



Of course the history isn't nearly so linear and clean, branches overlap, there can be dozens of branches at any given time, etc.



Longer-lived branches might merge the latest changes from the main into themselves first, before ultimately being merged back! Visually this can be a bit of a mess.

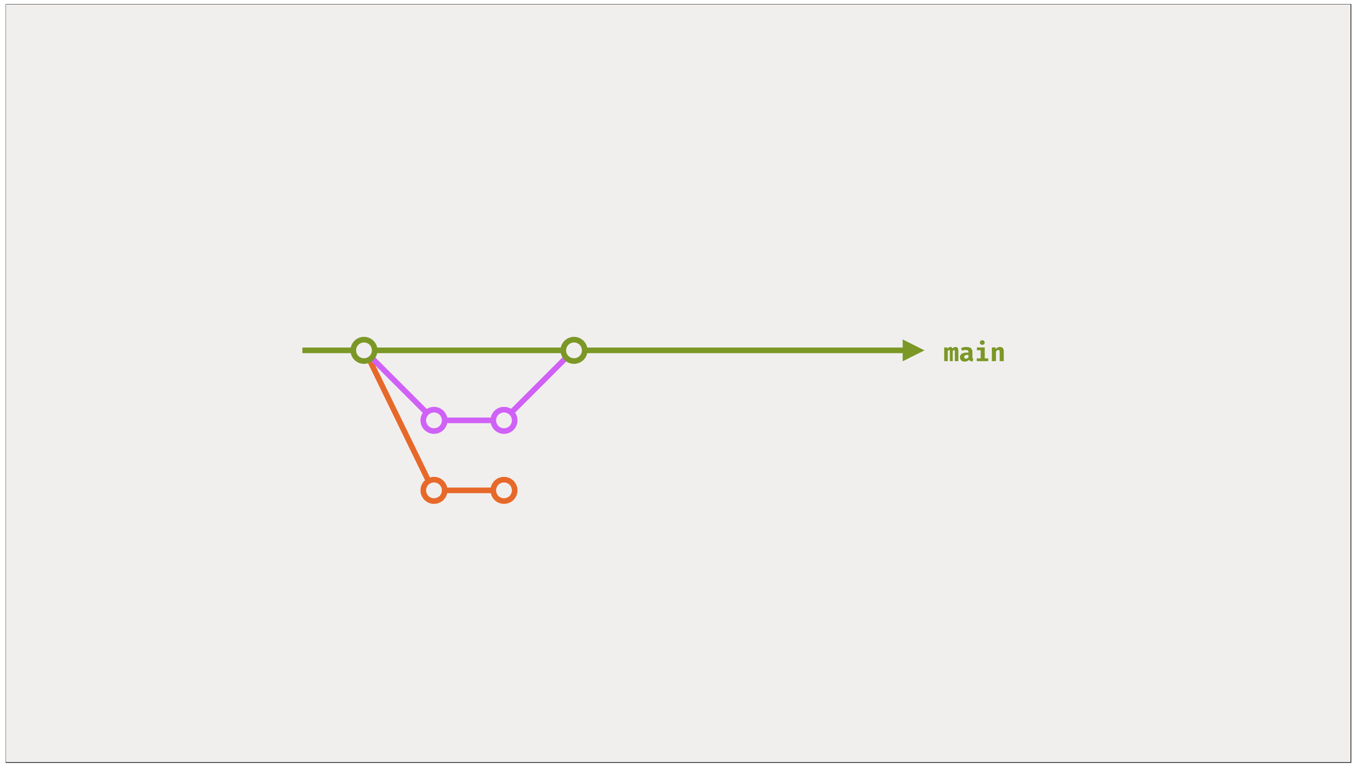
Unit of change

https://zachholman.com/posts/git-commit-history/

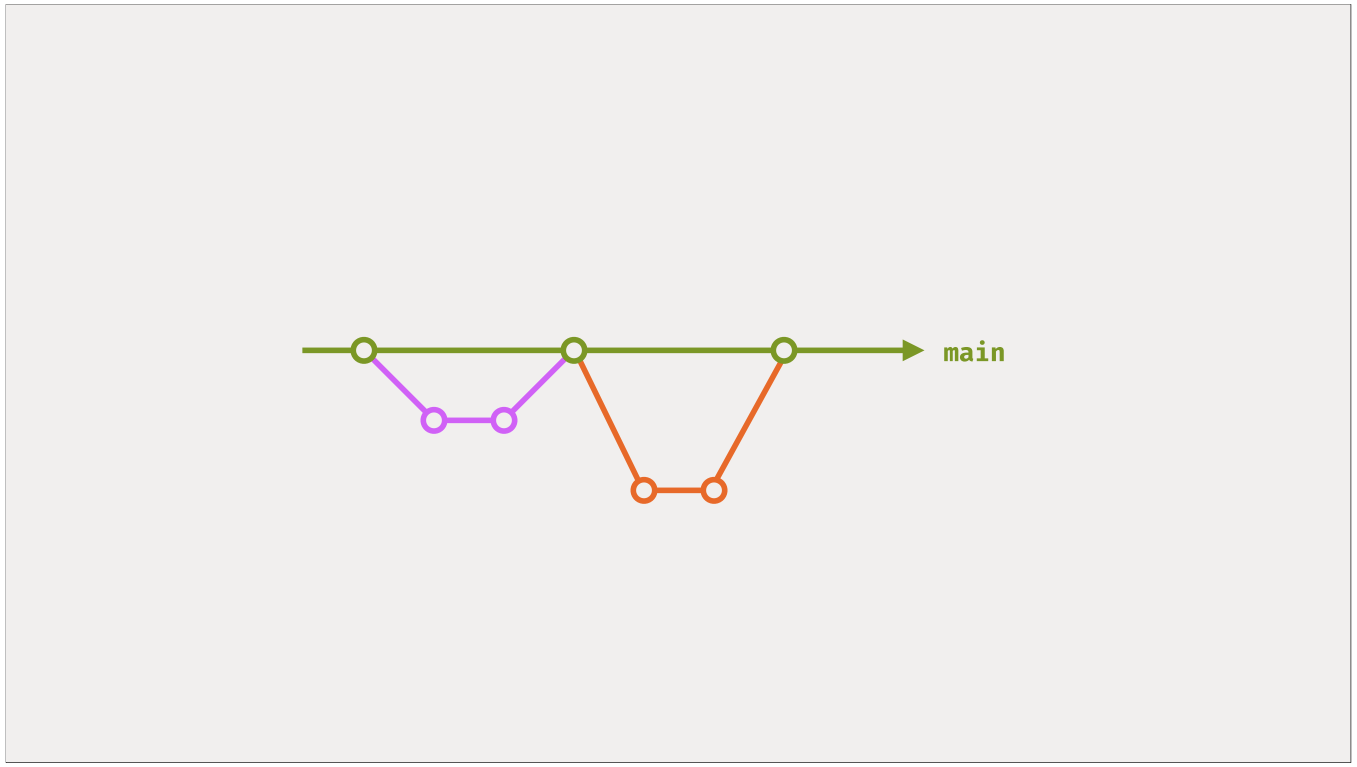
One way to think about this is your “unit of change” Think of commit messages as the details of a story, and pull requests as chapters.

git-core uses commits, and GitHub leans toward pull requests. In my work I’m much more likely to find information about changes in a PR writeup than individual commits.

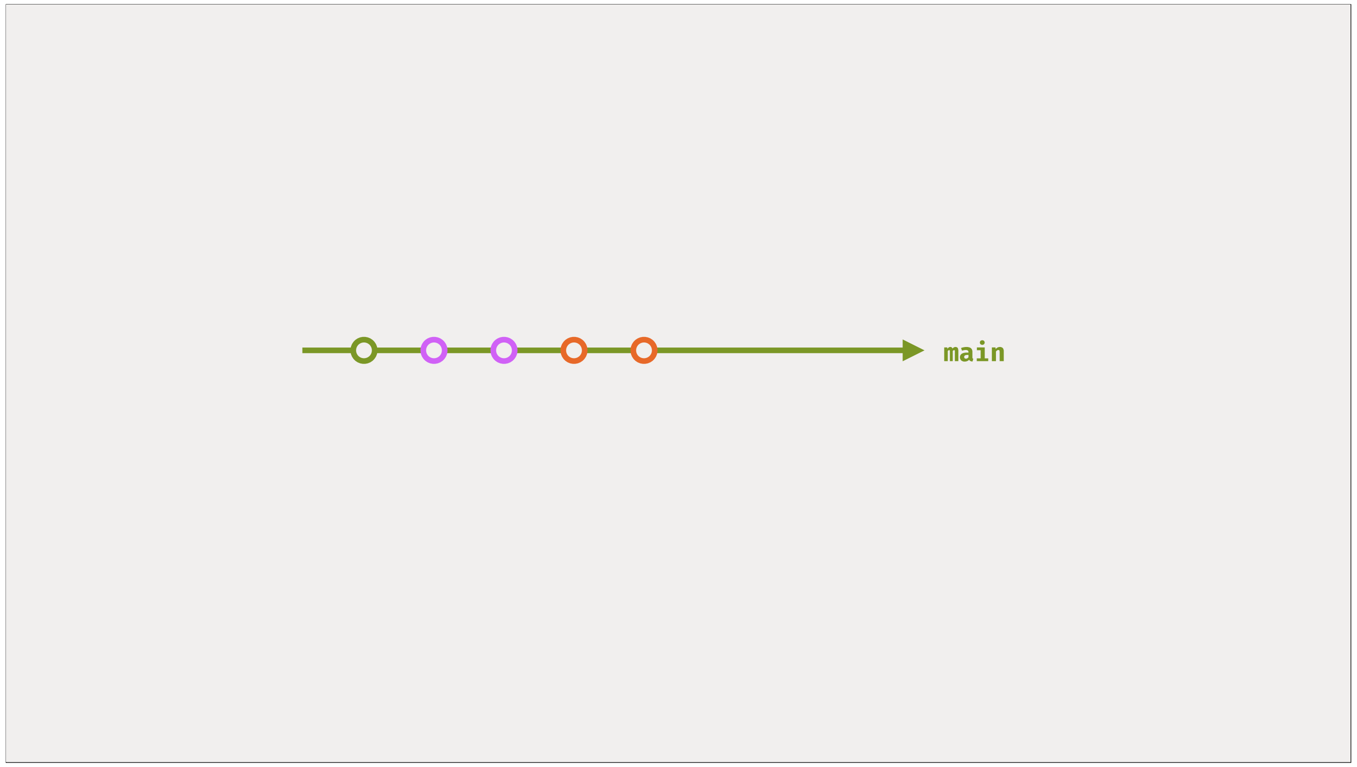
There are a lot of different workflows you can use for development on a project, and there’s not a correct answer either. But they all share the goal of producing an understandable history of software artifacts



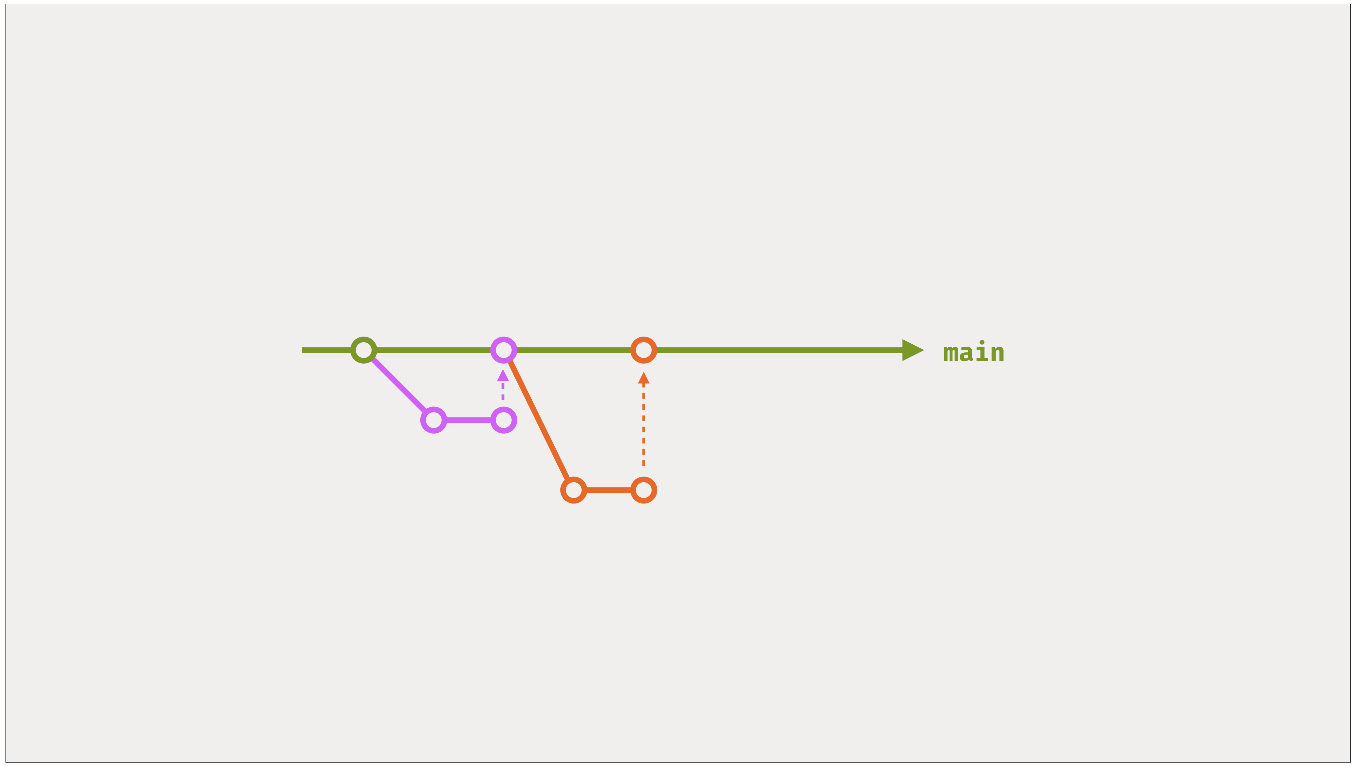
So let’s say you’ve got a feature branch, but in the meantime another one got merged. Instead of just merging, some teams decide that they want that cleaner history.



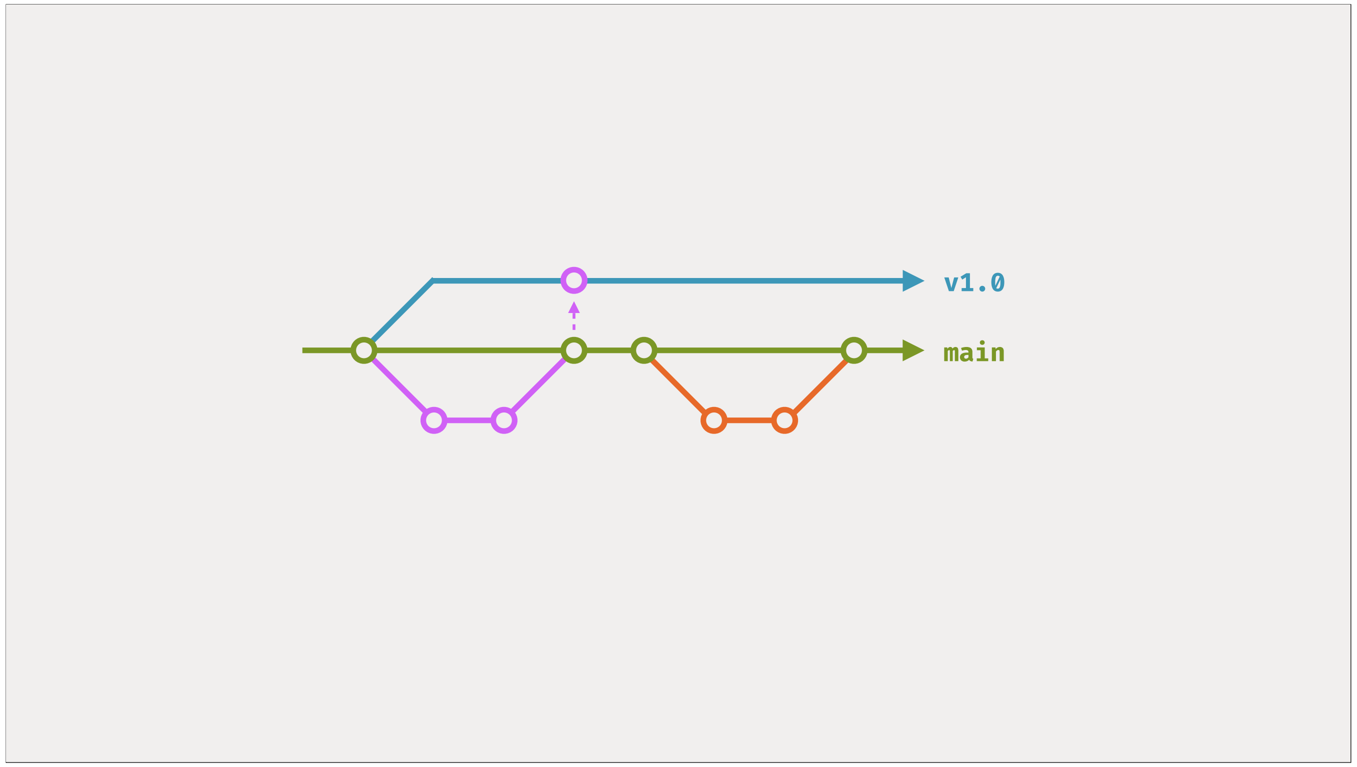
So their practice is, always rebase before merging. This keeps the history more “clean” and linear.



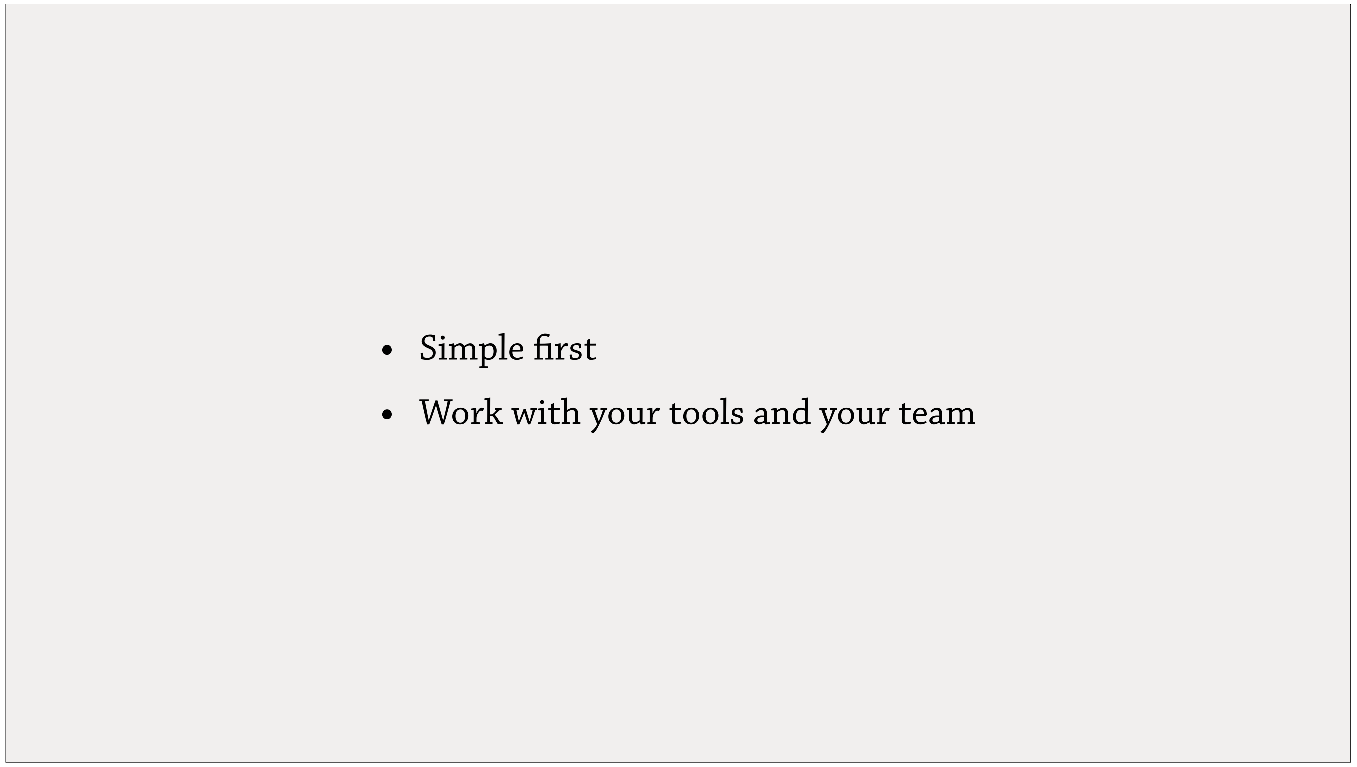
You could even do rebase and fast-forward only! Although this loses the branch entirely so it’s harder to see what part of work an individual commit came from.



Or, you can use squash merges, taking the contents of each feature branch and make them a single commit on the main branch.



You can have long-running branches for released versions, and integrate bug fixes from the main development branch as needed.



- Simple first
- Work with your tools and your team

… and more. My advice would be, keep it simple and work with your tools. At GitHub obviously we build everything around pull requests. We don’t worry too much about a clean linear history because everything’s in the pull requests. And our deployment and testing tools work with that. Whatever team you end up working with will probably have their own approach.

Demos

Now I’d like to show you all of these things we’ve talked about.

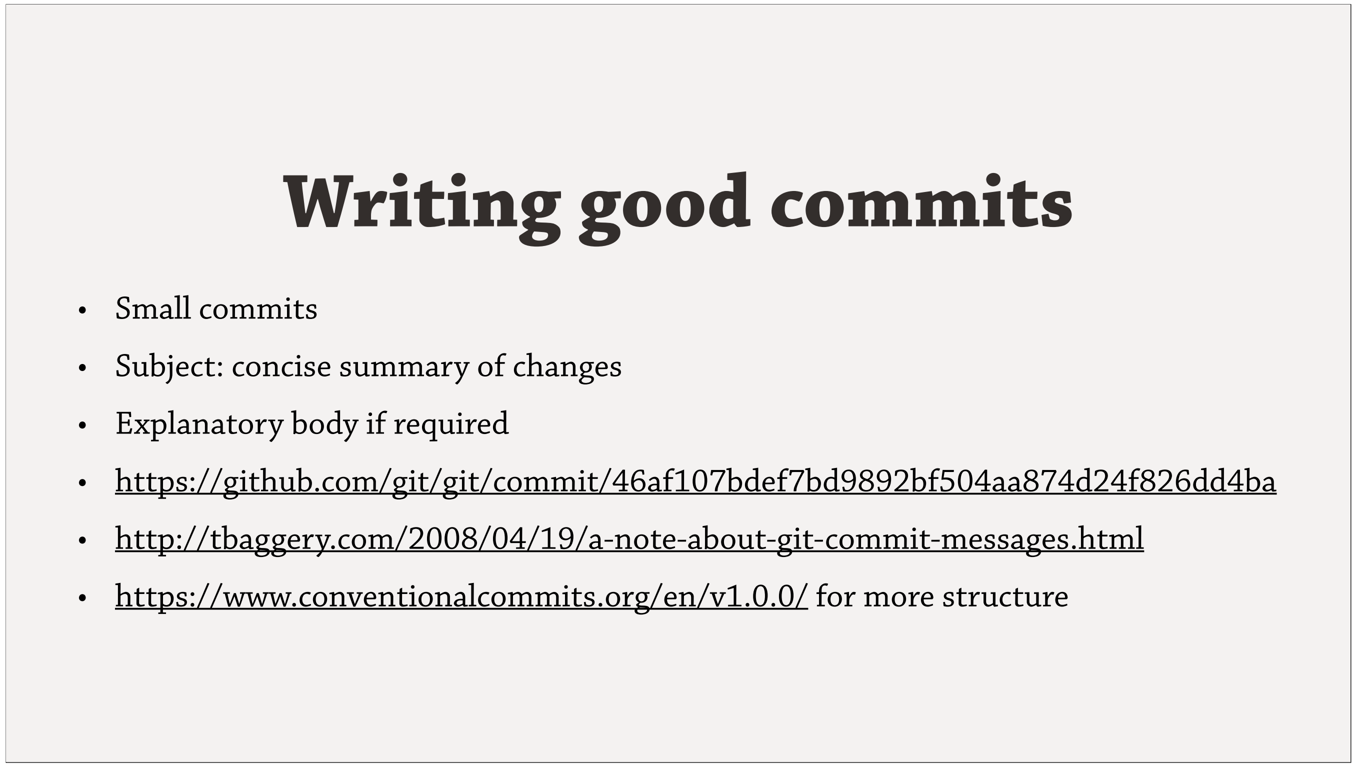
* create repo locally
* create file, make commit
* create new branch, switch to it
* add a feature.
* switch back to main branch, update with a new feature
* create remote repo
* push main to remote
* create new competing feature
* push feature to remote, open PR
* commit PR
* switch back to main, pull changes
* now try to merge original conflicting branch
* resolve conflict, commit merge, push main.

Advice

I have some advice about how to work well with git and GitHub.

Writing

Something that surprised me when I started at GitHub was how much writing I ended up doing. Not just chat in Slack or whatever, but in commits, issues, and pull requests. Here are some tips for better writing when using git and GitHub.

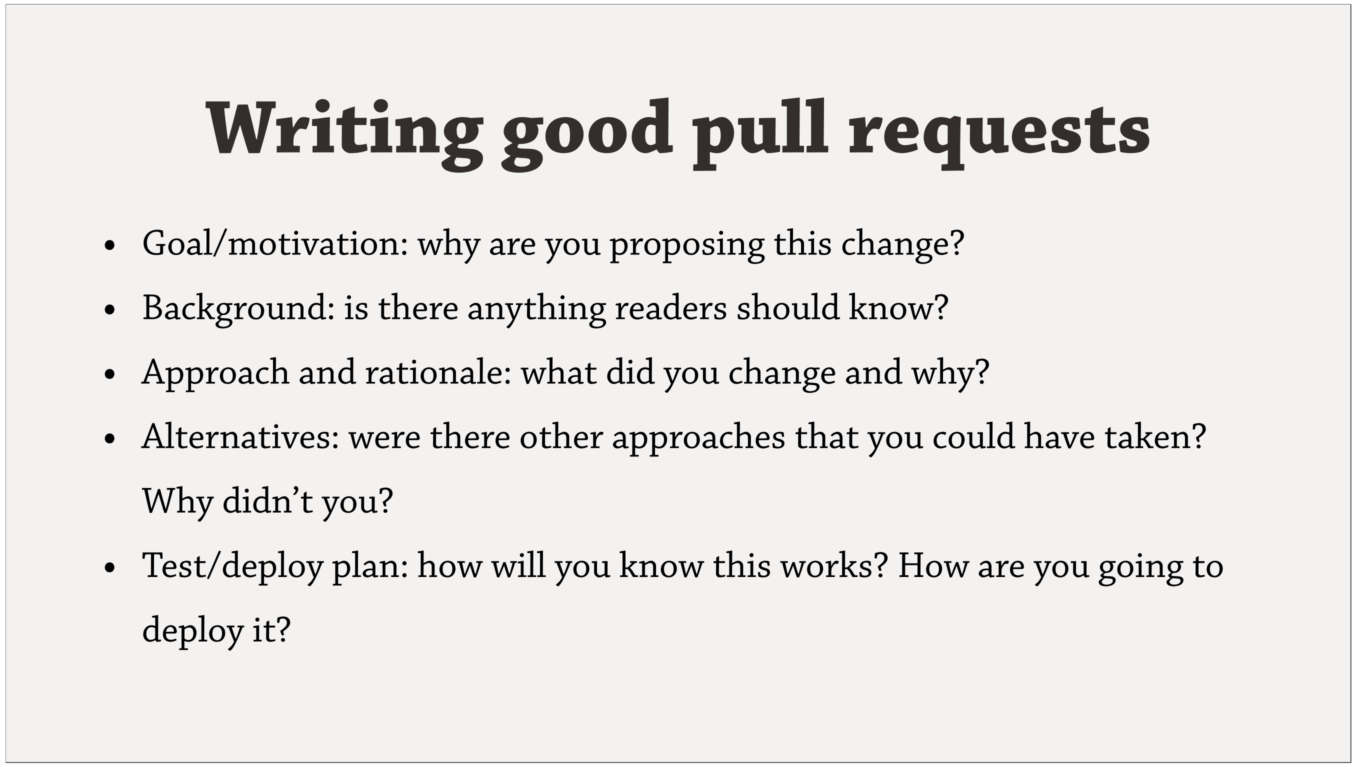


Writing good commits

- Small commits
- Subject: concise summary of changes
- Explanatory body if required
- https://github.com/git/git/commit/46af107bdef7bd9892bf504aa874d24f826dd4ba
- http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html
- https://www.conventionalcommits.org/en/v1.0.0/ for more structure

When you write a commit, you want it to be sufficiently explanatory so someone looking at it can understand why the changes were made.
First, keep them small.
Sometimes this is a simple "here's what changed", but you can also add more explanation if needed.
Don't make them too big. Change one thing at a time — with partial staging you can change a whole bunch of things and then commit each piece separately.

An extreme (but also very good) example who writes good commit messages: the git core team. https://github.com/git/git/commit/46af107bdef7bd9892bf504aa874d24f826dd4ba



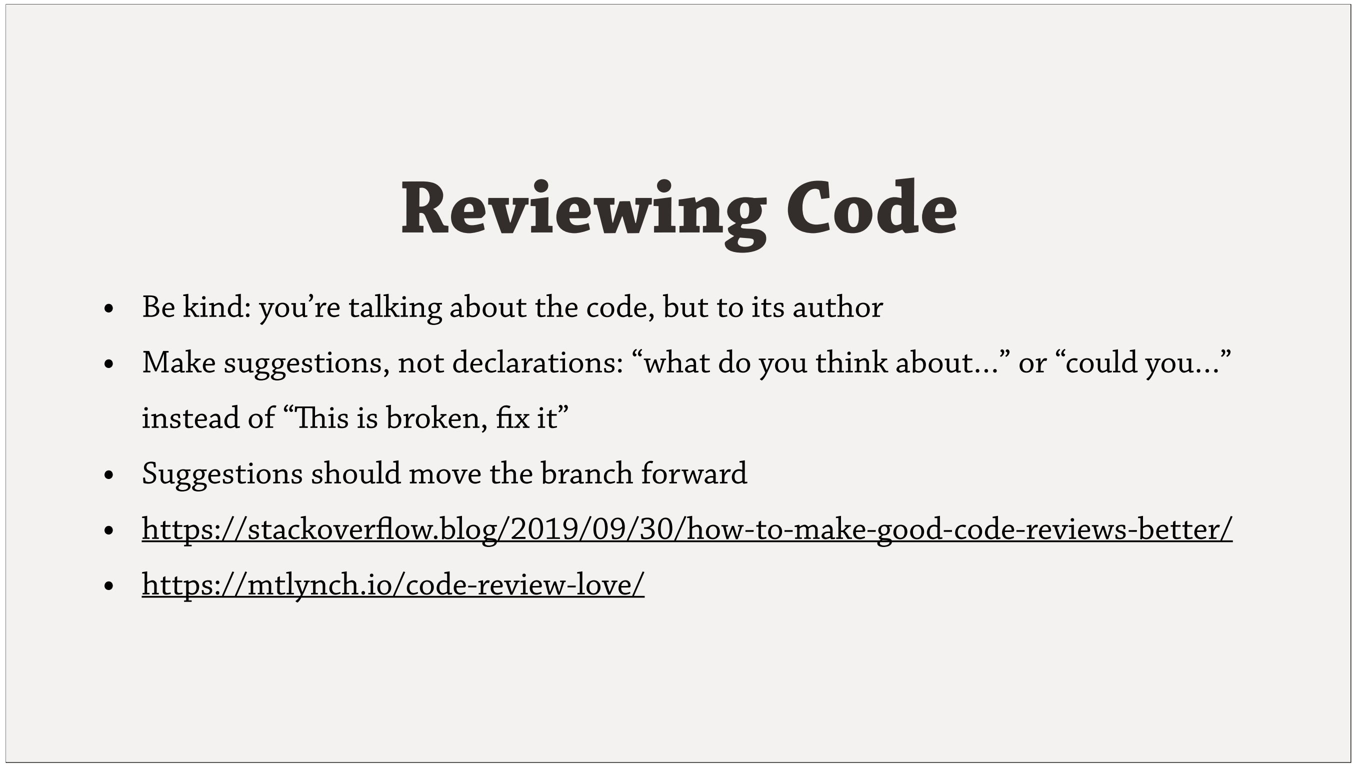
Writing good pull requests

- Goal/motivation: why are you proposing this change?
- Background: is there anything readers should know?
- Approach and rationale: what did you change and why?
- Alternatives: were there other approaches that you could have taken? Why didn’t you?
- Test/deploy plan: how will you know this works? How are you going to deploy it?

Much of my writing happens in pull requests.
Writing for two audiences: first, those who are reviewing your code now, and second, developers in the future wanting to find out what happened.
An example: https://github.com/github/scientist/pull/18 and a *great* example for opening an issue on an open-source project: https://github.com/github/scientist/issues/95

Writing a history

Above all, remember that you’re writing a history. Even if it’s messy, the idea is that in the future if you wanted to know why a piece of code is the way it is, you can figure it out — either from reading commits, or pull requests, or whatever.

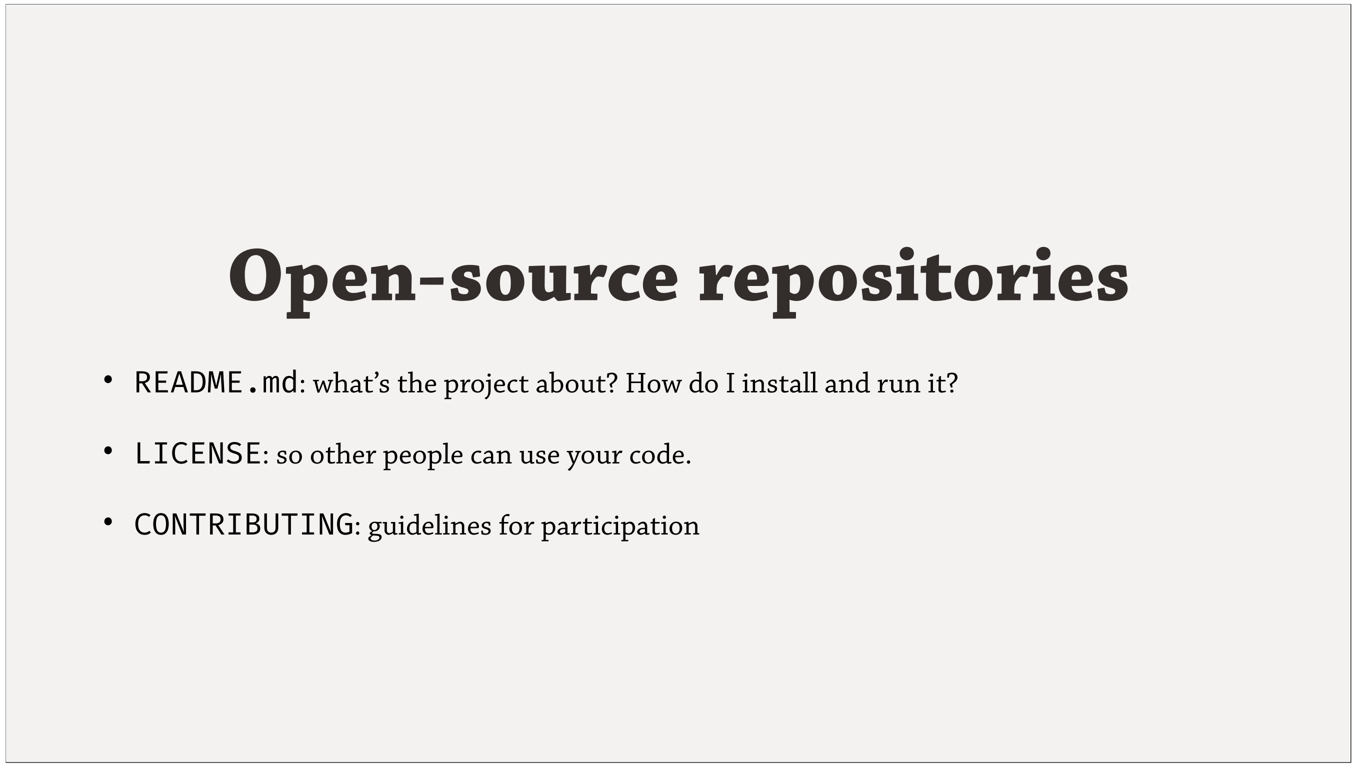


Reviewing Code

- Be kind: you're talking about the code, but to its author
- Make suggestions, not declarations: "what do you think about…" or "could you…" instead of "This is broken, fix it"
- Suggestions should move the branch forward
- https://stackoverflow.blog/2019/09/30/how-to-make-good-code-reviews-better/
- https://mtlynch.io/code-review-love/

One more thing. Writing a pull request isn't the last thing. Others may be opening pull requests too, and you're going to be reviewing code. Here are some things to think about when doing those reviews.

Examples: https://github.com/github/scientist/pull/79 https://github.com/github/scientist/pull/87 https://github.com/github/github-ds/pull/47



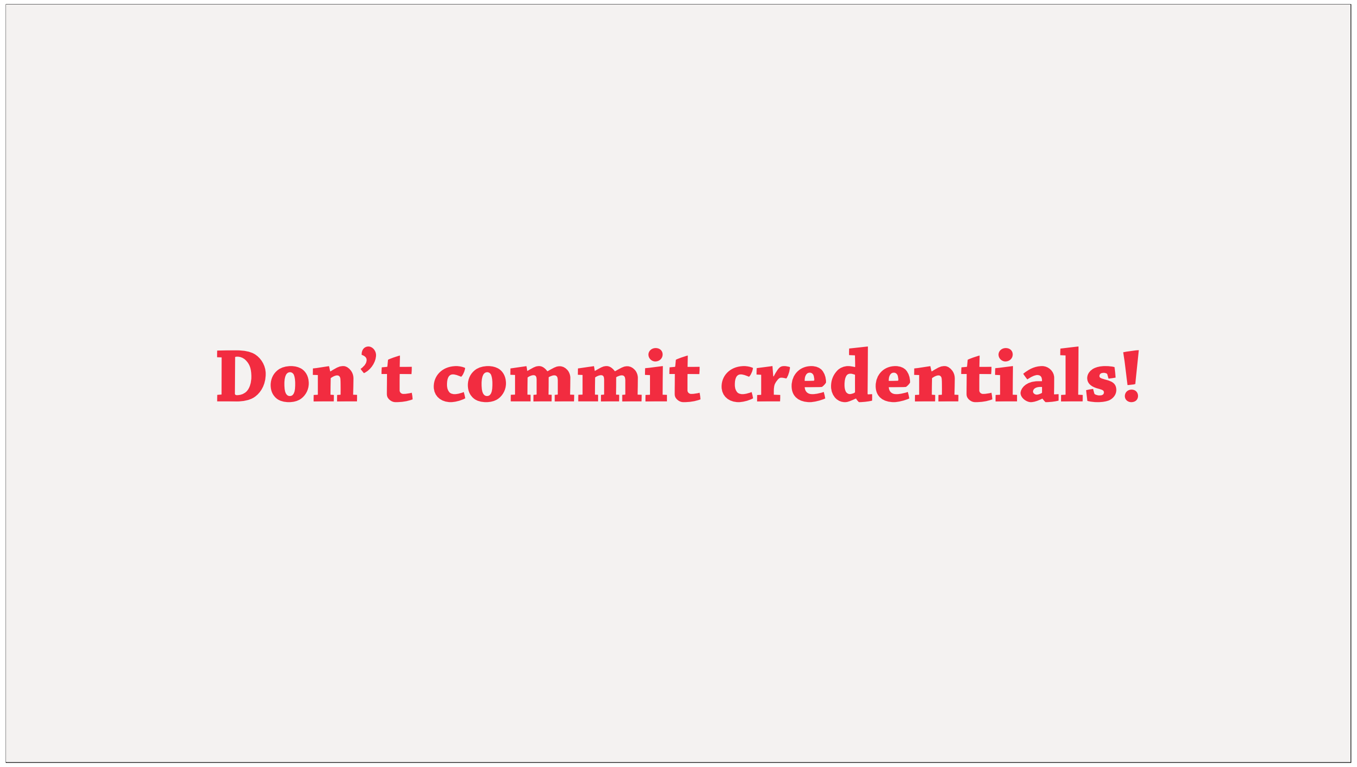
Open-source repositories

- `README.md`: what's the project about? How do I install and run it?
- `LICENSE`: so other people can use your code.
- `CONTRIBUTING`: guidelines for participation

A few things about posting your code publicly.
Show https://github.com/github/scientist : has both a README and a LICENSE

What shouldn’t you commit?

What shouldn’t go in your repository?



Don’t commit credentials!

Since you’ll be working with AWS and web services and things during this class, you’ll have lots of API keys and credentials. Don’t commit these!, but people watch for public repos containing credentials, will steal them, and mine bitcoins using your amazon account. We try and catch things like this before they leak, but still!

git is a sharp tool

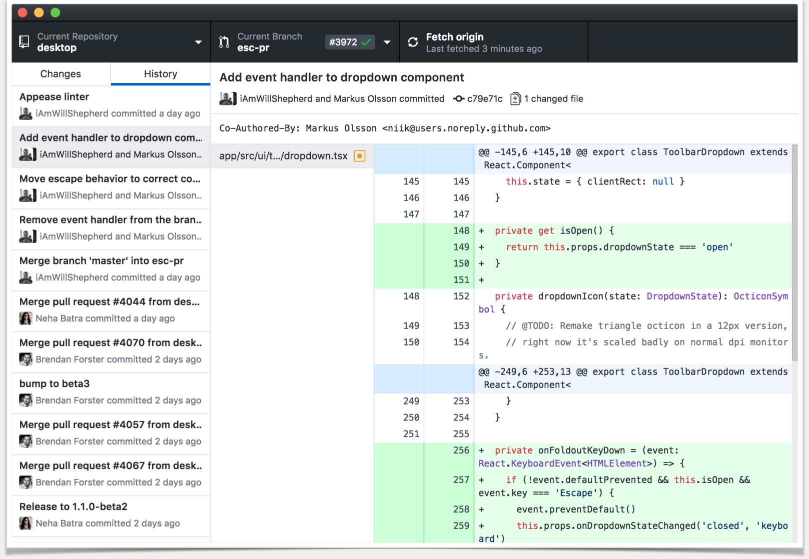
git is a sharp tool, and it lets you do a lot of things. You can change your history, rewrite commits, and more. If you’re not careful, you can get into confusing states.

…but it tries

But git also does its best to protect your data. Something to keep in mind: if git knows about your data — that is, if you’ve committed something — it’s hard to lose it permanently. If you *haven’t* committed something, then it’s a lot easier to lose!

Resources

- Oops! https://ohshitgit.com
- Learning game: https://ohmygit.org
- Documentation: https://docs.github.com/en/get-started
- Book: https://git-scm.com/book/en/v2
- Learning courses: https://lab.github.com/
- Cheat sheets and manuals: https://training.github.com/
- Git internals: https://jwiegley.github.io/git-from-the-bottom-up/



Appease linter
iAmWillShepherd committed a day ago

Add event handler to dropdown com...
iAmWillShepherd and Markus Olsson...

Move escape behavior to correct co...
iAmWillShepherd and Markus Olsson...

Remove event handler from the bran...
iAmWillShepherd and Markus Olsson...

Merge branch 'master' into esc-pr
iAmWillShepherd committed a day ago

Merge pull request #4044 from des...
Neha Batra committed a day ago

Merge pull request #4070 from desk..
Brendan Forster committed 2 days ago

bump to beta3
Brendan Forster committed 2 days ago

Merge pull request #4057 from desk..
Brendan Forster committed 2 days ago

Merge pull request #4067 from desk..
Brendan Forster committed 2 days ago

Release to 1.1.0-beta2
Neha Batra committed 2 days ago

Add event handler to dropdown component

iAmWillShepherd and Markus Olsson committed c79e71c 1 changed file

```
Co-Authored-By: Markus Olsson <niik@users.noreply.github.com>
```

app/src/ui/t.../dropdown.tsx

```
@@ -145,6 +145,10 @@ export class ToolbarDropdown extends
 React.Component<
     this.state = { clientRect: null }
   }

+  private get isOpen() {
+    return this.props.dropdownState === 'open'
+  }
+
   private dropdownIcon(state: DropdownState): OcticonSym
bol {
     // @TODO: Remake triangle octicon in a 12px version,
     // right now it's scaled badly on normal dpi monitor
s.
@@ -249,6 +253,13 @@ export class ToolbarDropdown extends
 React.Component<
     }
   }

+  private onFoldoutKeyDown = (event:
React.KeyboardEvent<HTMLElement>) => {
+    if (!event.defaultPrevented && this.isOpen &&
event.key === 'Escape') {
+      event.preventDefault()
+      this.props.onDropdownStateChanged('closed', 'keybo
ard')
```

https://desktop.github.com/

Everything I showed you today was using the command line. It’s where I work and I’m most comfortable. We do, however, have a GUI desktop app for multiple platforms that lets you use git without having to remember what to type. Check this out especially if you’re not as comfortable on the command line.

http://education.github.com/pack

Free stuff for students! Private repositories, AWS credits, and a lot more.

https://internships.github.com/

https://github.com/about/careers

Internships: it looks like they’re currently not available, but if you visit here you can sign up for emails when they are.
And our general careers site as well.

Questions?

Any questions before we wrap up?

Thank you!

That’s it!

zerowidth@github.com
https://github.com/zerowidth

This is here for reference when the slides get emailed to you later.