Sources list

ProGit(Book)
Git and GitHub for Beginners - Crash Course
Atlassian - setting up a repo

Todo

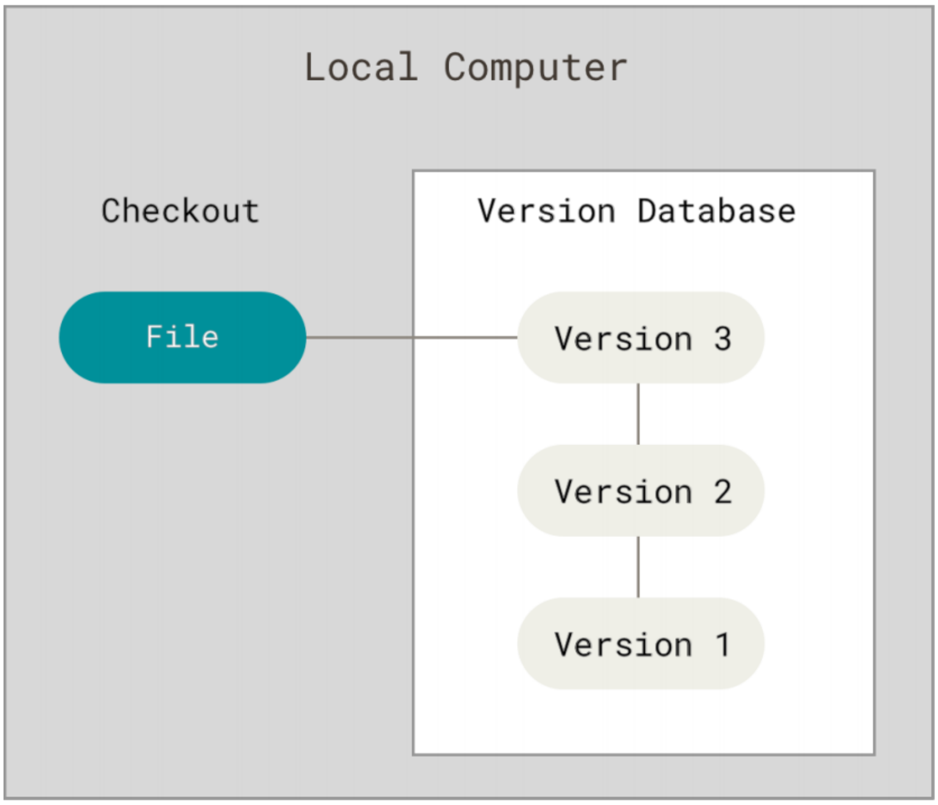
Large File storage
git init --SEPARATE-GIT-DIR=

Pro Git

What is git

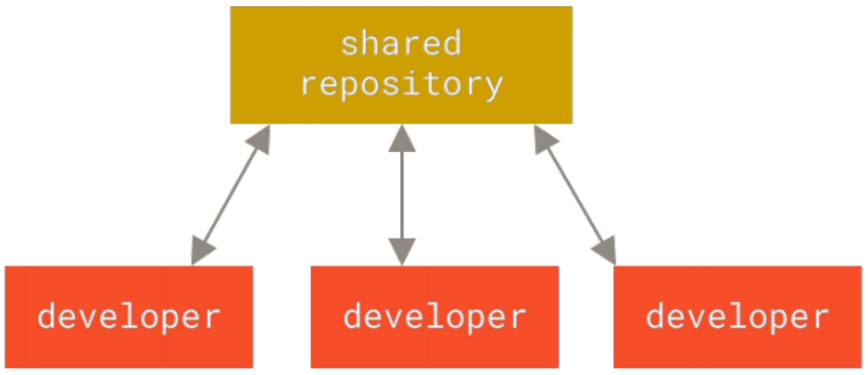
Version control

What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.



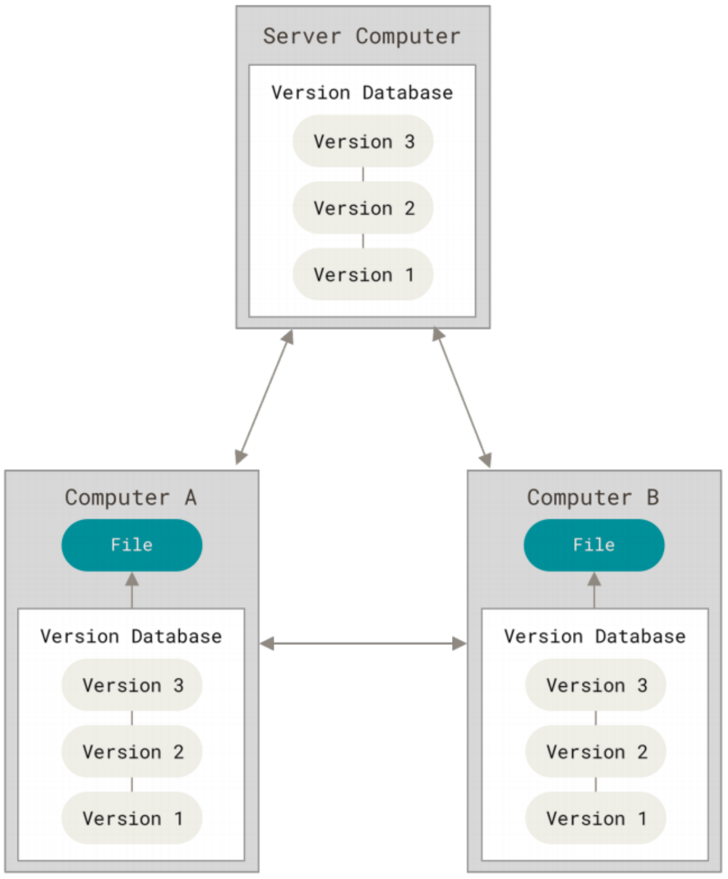
Centralized version control

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.



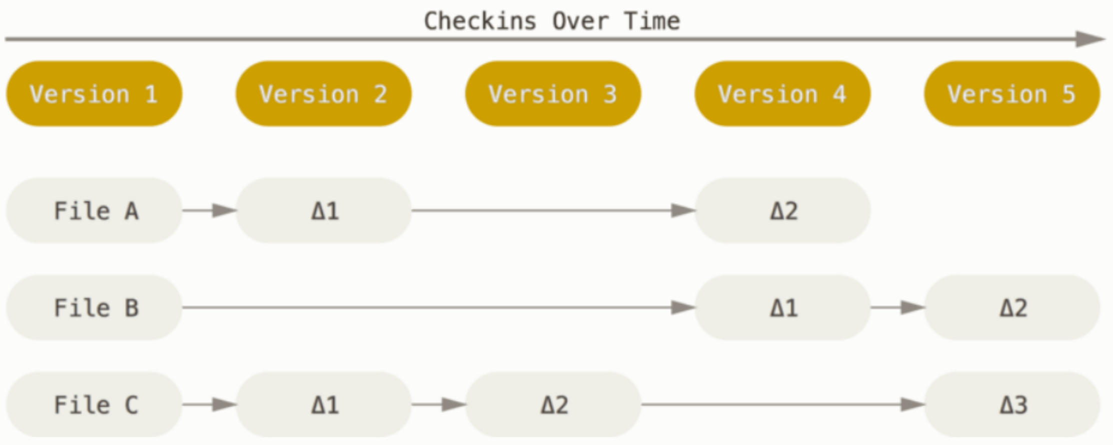
Distributed version control

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history.



Snapshots, Not Differences

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These other systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they store as a set of files and the changes made to each file over time (this is commonly described as **`delta-based`** version control).



file it has already stored. Git thinks about its data more like a **stream of snapshots**.

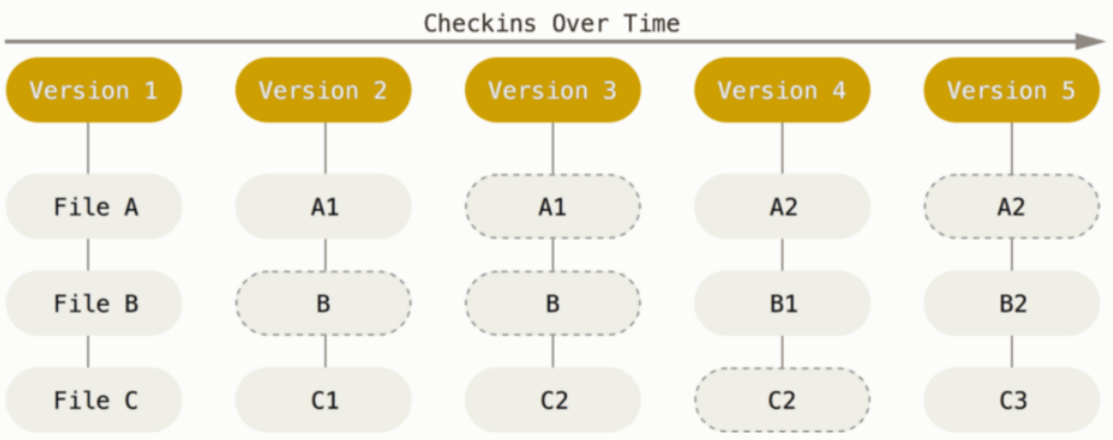


Figure 5. Storing data as snapshots of the project over time

This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS. We'll explore some of the benefits you gain by thinking of your data this way when we cover Git branching in Git Branching.

Nearly Every Operation Is Local

Git Has Integrity

Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't lose information in transit or get file corruption without Git being able to detect it.

A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

The Three States S.15

Pay attention now — here is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: *modified*, *staged*, and *committed*:

- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
- Committed means that the data is safely stored in your local database.

This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory.

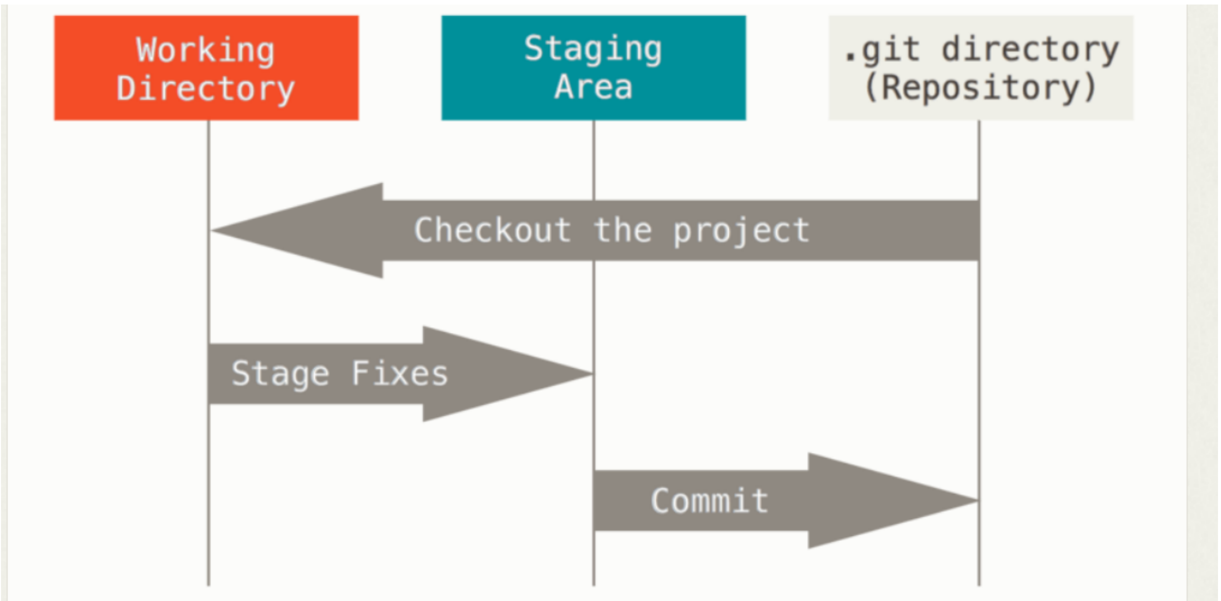


Figure 6. Working tree, staging area, and Git directory
The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

Unstaging a staged file

```
git reset HEAD <file> #or
git restore --staged <file>
```

Reset changes on unstaged file

```
git restore <file>
```

This reset all changes to the file and reverts it to last commit version, be careful if you do this all your changes are gone

Showing remotes

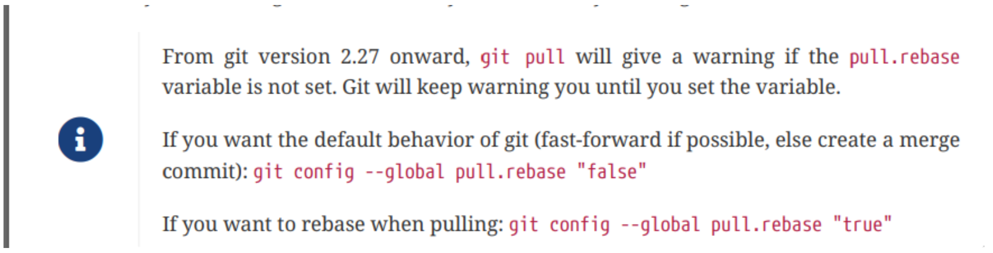
```
git remote
```

`-v` to see the corresponding url's

Getting data from remotes

```
git fetch
```

No automatic merge is done



From git version 2.27 onward, `git pull` will give a warning if the `pull.rebase` variable is not set. Git will keep warning you until you set the variable.

If you want the default behavior of git (fast-forward if possible, else create a merge commit): `git config --global pull.rebase "false"`

If you want to rebase when pulling: `git config --global pull.rebase "true"`

Tagging

Listing tags

Lists tags in alphabetical order

```
git tag
```

Creating Tags

There exist two types of tags *lightweight* and *annotated*

```
git tag -a v1.4 -m "<Message>" #annotated tag
git tag v1.4 #lightweight tag
```

This will tag your current commit

Tagging earlier commits

```
git tag -a v1.2 <commit id>
```

Push tags

```
git push --tags
```

deleting tag names

```
git tag -d <name>
```

Using tags enables you to checkout specific tags

```
git checkout v2.0.0
```

But this put you in a detached head state so you can use

```
git checkout -b <branch-name> <tag-name>
```

to automatically create a new branch

Git branching

What are branches

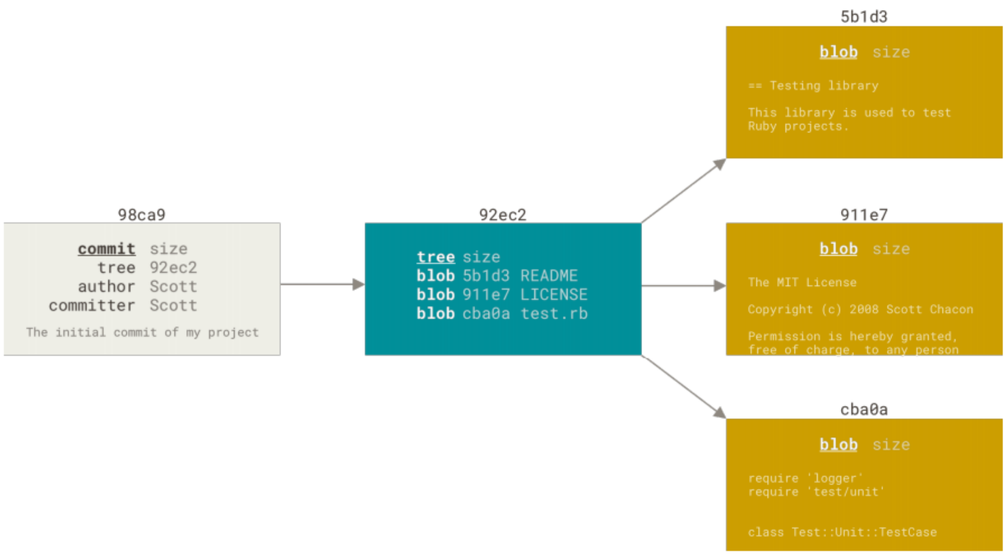
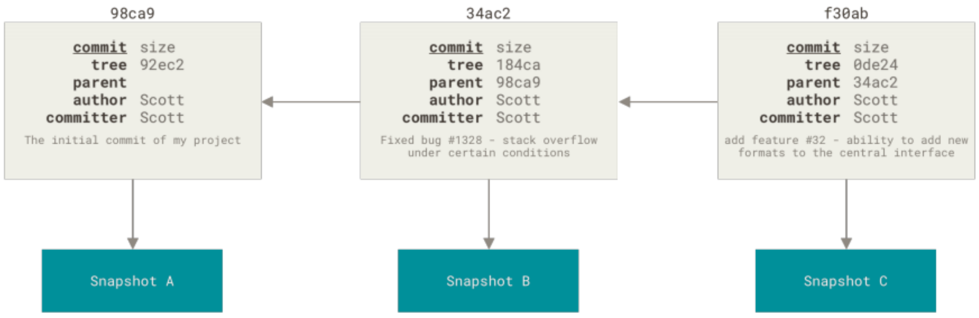
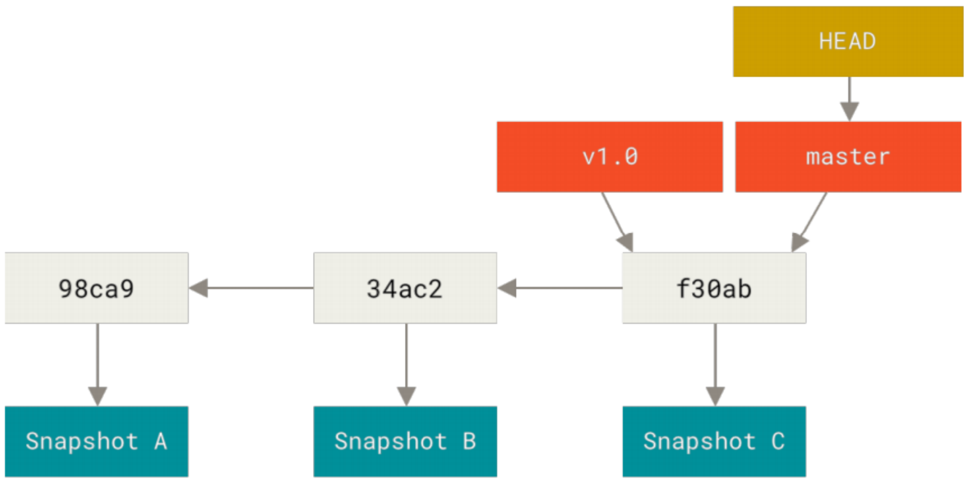
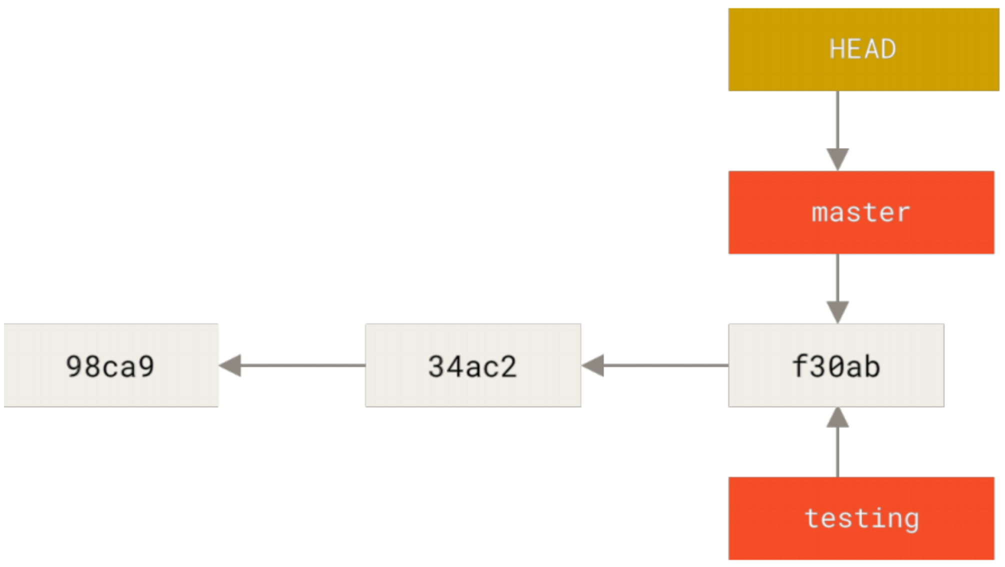


Figure 9. A commit and its tree

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.







A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically
How does Git know what branch you're currently on? It keeps a special pointer called HEAD. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS.

See on which commits the branches are

```
git log --oneline --decorate
```

So checkout just moves the HEAD to point the new branch

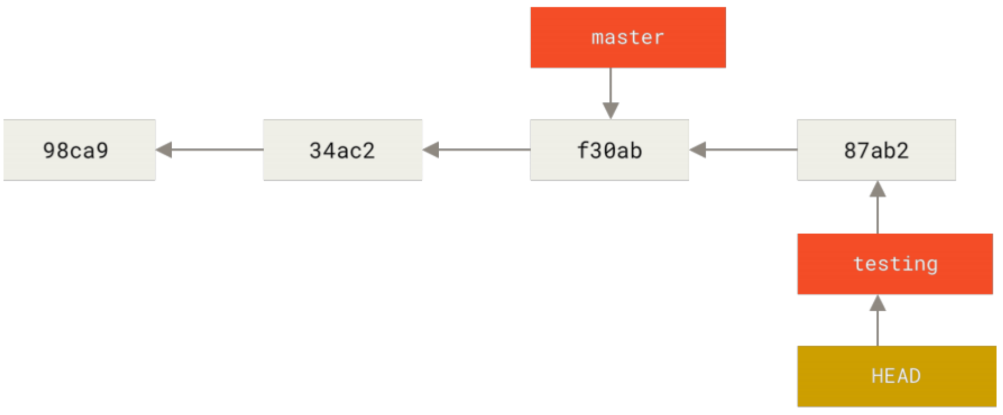
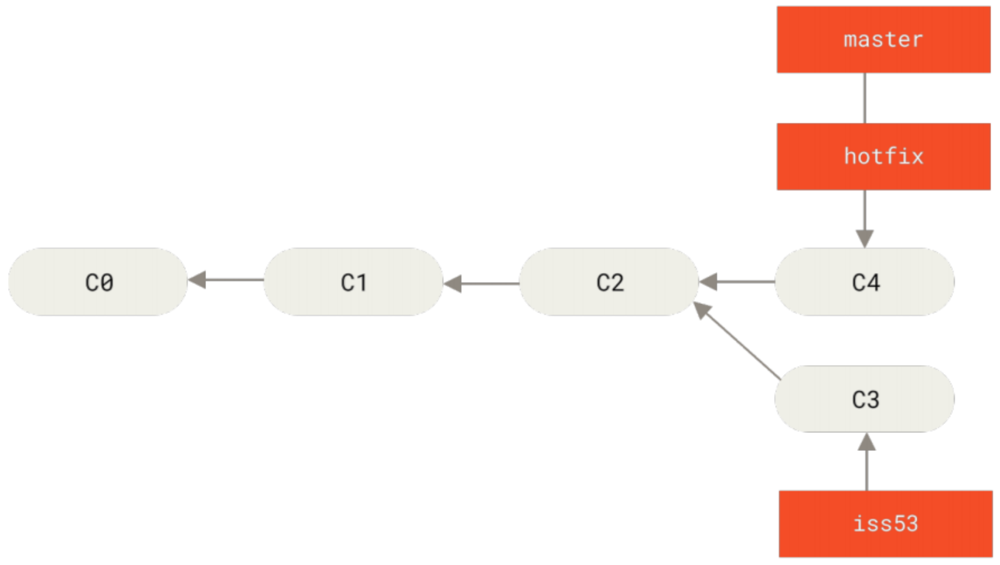
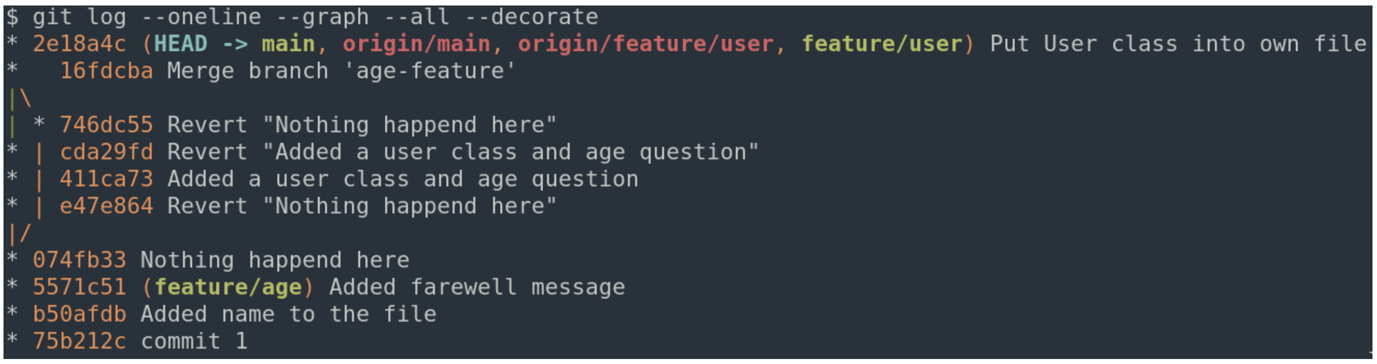


Figure 15. The HEAD branch moves forward when a commit is made

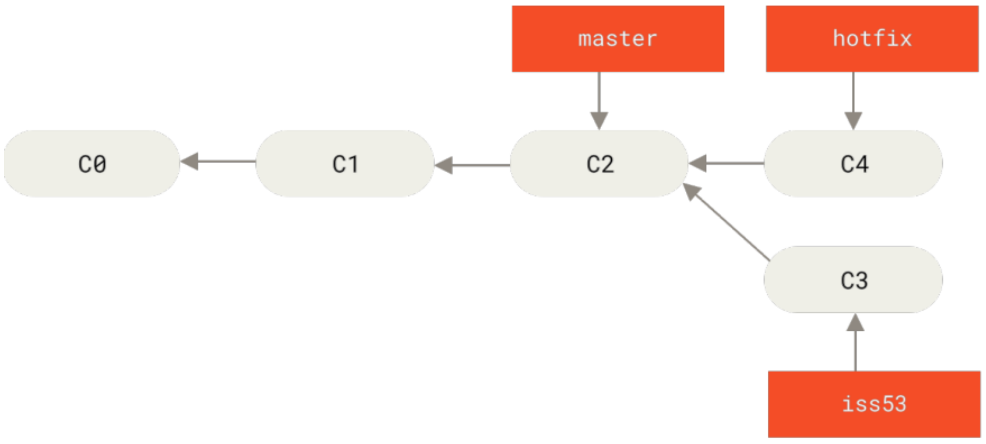


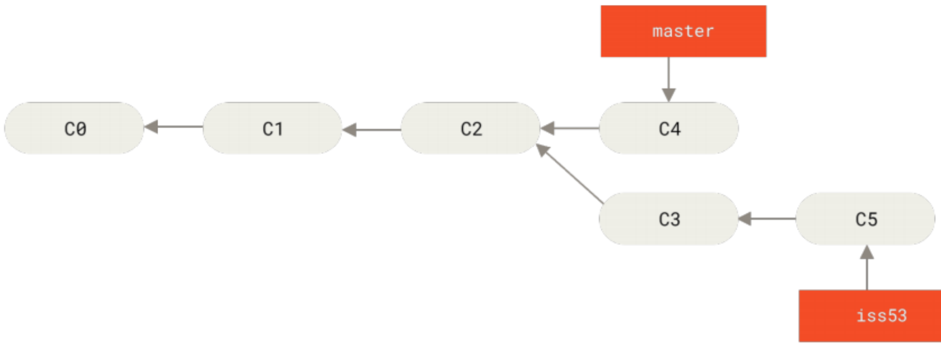
```
git log --oneline --graph --all --decorate
```



```
$ git log --oneline --graph --all --decorate
* 2e18a4c (HEAD -> main, origin/main, origin/feature/user, feature/user) Put User class into own file
*   16fdcba Merge branch 'age-feature'
|\
| * 746dc55 Revert "Nothing happend here"
* | cda29fd Revert "Added a user class and age question"
* | 411ca73 Added a user class and age question
* | e47e864 Revert "Nothing happend here"
|/
* 074fb33 Nothing happend here
* 5571c51 (feature/age) Added farewell message
* b50afdb Added name to the file
* 75b212c commit 1
```

Because a branch in Git is actually a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).



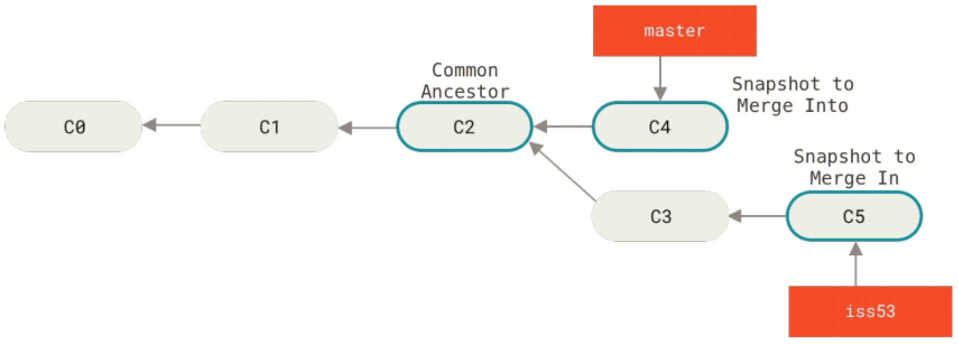


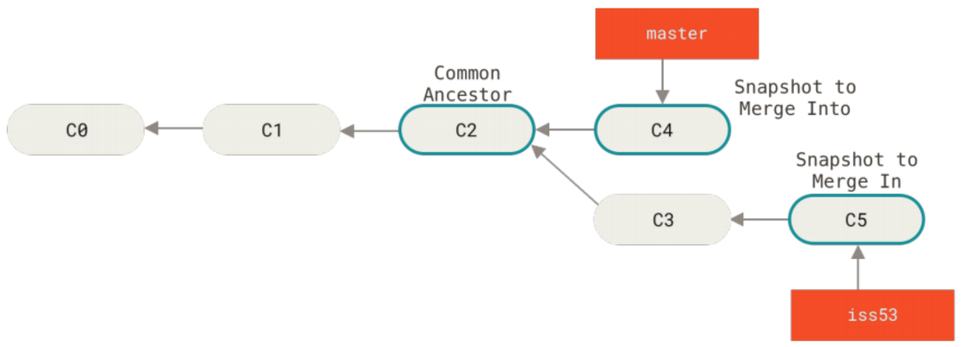
Basic merging

State: your are on iss53
Merge changes into your branch

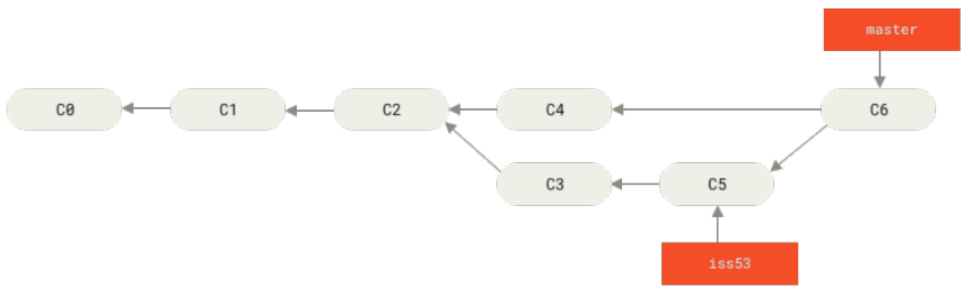
```
git checkout master
git merge iss53
```

Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.





Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.



G

Branching workflows

Long running branches

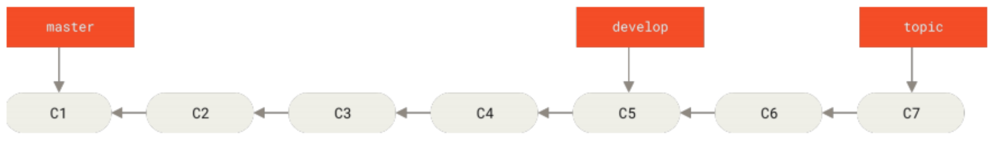
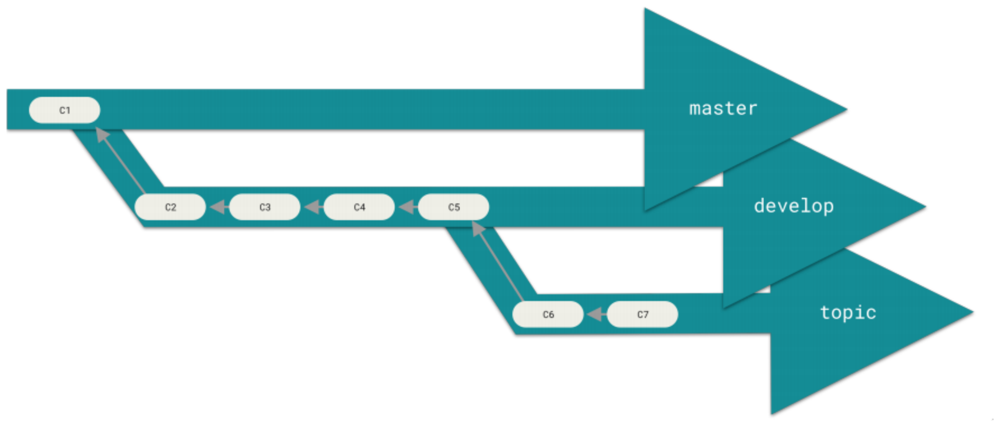
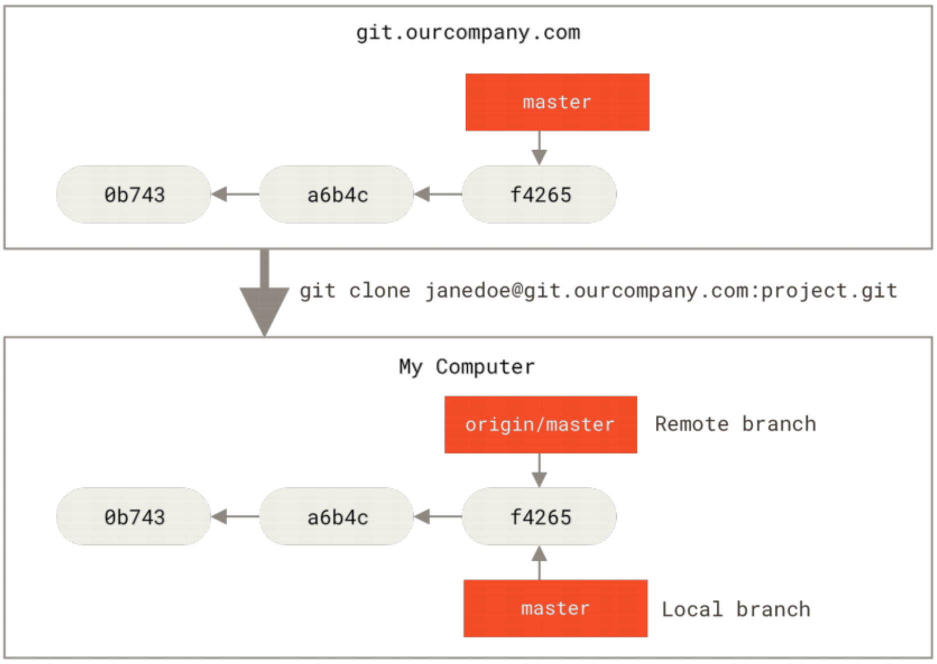


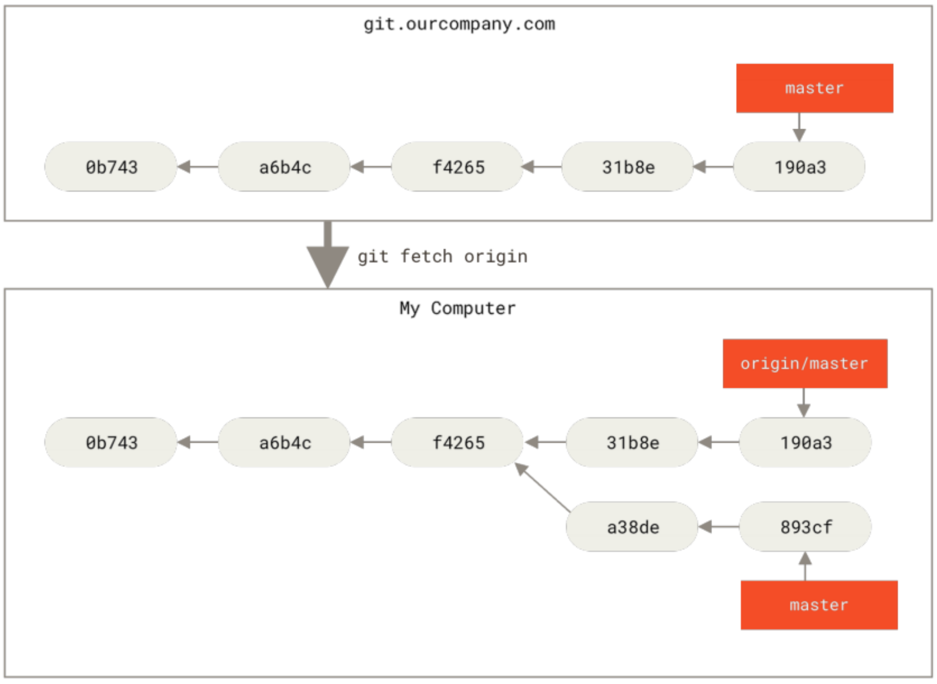
Figure 26. A linear view of progressive-stability branching

It's generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they're fully tested.



origin/master





You can also do git push origin serverfix:serverfix, which does the same thing — it says, "Take my serverfix and make it the remote's serverfix." You can use this format to push a local branch into a remote branch that is named differently. If you didn't want it to be called serverfix on the remote, you could instead run git push origin serverfix:awesomebranch to push your local serverfix branch to the awesomebranch branch on the remote projec

Rebasing

102
220-340

First time setup

https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

Git basics

Init empty git repository with `git init`

```
git init
```

Clone existing(possibly local) repo with `git clone`

git commands

Git basics

git init

git clone

git clone <repo-name> creates a directory called <repo-name and copies the necessary files into it.

git clone <repo-name> <new-name> creates a directory called <new-name> and copies the necessary files into it.

git add

git commit

Pro Git: https://git-scm.com/

Git and GitHub for Beginners - Crash Course

Git workflow

First step(Configure git)

Working with git

Initializing a new git repository(a repository is basically another for a project) i.e. in our case a folder

```
git init <repo_name>
or
git init . #init repo in existing directory
```

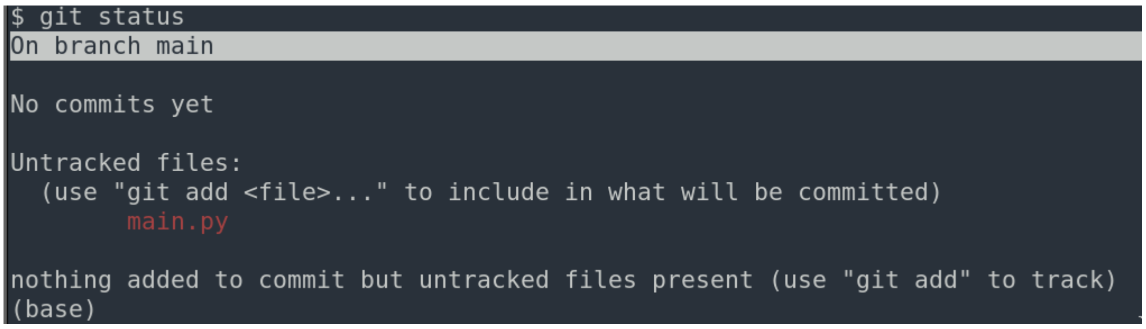
or cloning(copying) an already existing git repository

```
git clone <url>
```

TODO:ssh vs https

TODO How to get help i.e. -h and man pages

TODO: git status
Shows branch and other things we will get into later.



```
$ git status
On branch main

No commits yet

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

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

```
touch main.py
```

```
#!/usr/bin/python3
def main():
        print("hello from python")


if __name__ == "__main__":
        main()
```

```
chmod +x main.py
```

Now this file is in your working directory but its not checked by git. To make this file know to git you have to add it

```
git add main.py
```

Now the file is at the staging are of git(!The file is not stored in git yet)

```
git status
```

TODO: 3 STAGES OF GIT

The next thing to do is commit the file so git really knows about the file and later also its changes

```
git commit -am "message"
or
git commit main.py -m "message" #If the file is already know to git
or
git commit
```

Useful flags for commit

```
git commit -v
git commit -a
```

TODO: What should be the content of a single commit(semantically different changes)
TODO: Amend

Lets add a new feature to our python program

```
#!usr/bin/python3
def main():
        name = input("Please enter your name: ")
        print(f"hello from python {name}")


if  __name__ == "__main__":
        main()
```

```
#!usr/bin/python3
def main():
        name = readline()
        print("hello from python {name}")


def sayGoodBye(name):
        print(f"Goodbye {name}")

if  __name__ == "__main__":
        name = main()
        sayGoodBye(name)
```

Two semantically different things i.e. changes that should not be mixed in a commit

```
git add -p
use e to get into editing mode
git commit -m "Added function to read in name"



git add -p
git commit -m "Added functionality to say Goodbye"
```

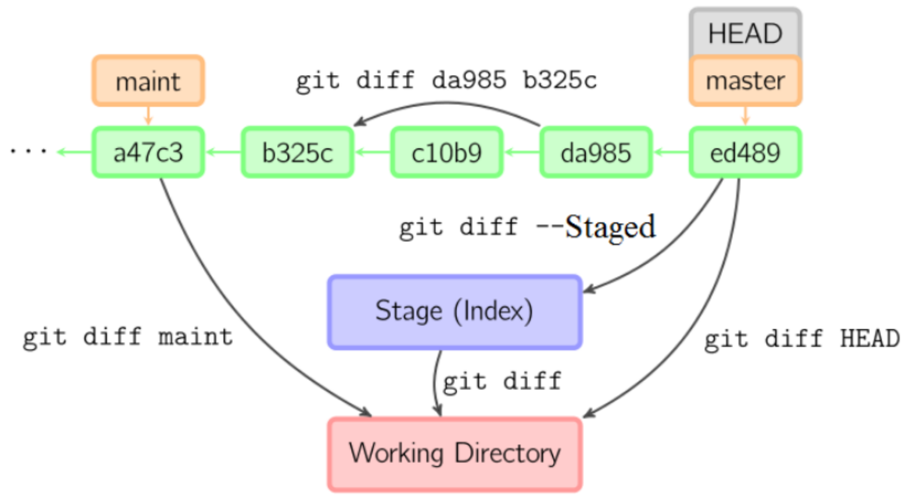
Enables you to stage different hunks of a given file.
This will present you with a chunk of changes and prompt you for a command. Use `y` to stage the chunk, `n` to ignore the chunk, `s` to split it into smaller chunks, `e` to manually edit the chunk, and `q` to exit.

- y stage this hunk for the next commit
- n do not stage this hunk for the next commit
- q quit; do not stage this hunk or any of the remaining hunks
- a stage this hunk and all later hunks in the file
- d do not stage this hunk or any of the later hunks in the file
- g select a hunk to go to
- / search for a hunk matching the given regex
- j leave this hunk undecided, see next undecided hunk
- J leave this hunk undecided, see next hunk
- k leave this hunk undecided, see previous undecided hunk
- K leave this hunk undecided, see previous hunk
- s split the current hunk into smaller hunks
- e manually edit the current hunk
- ? print hunk help

This is quite cumbersome, we will later mention a few tools to make this easier, but I think its important to know the terminal commands
No lets review what we did, which brings use to the next git command `git log` and `git diff`
`git log.`
TODO HEAD HASH IDs
Shows you the history of the TODO branch or repo?

`git diff`

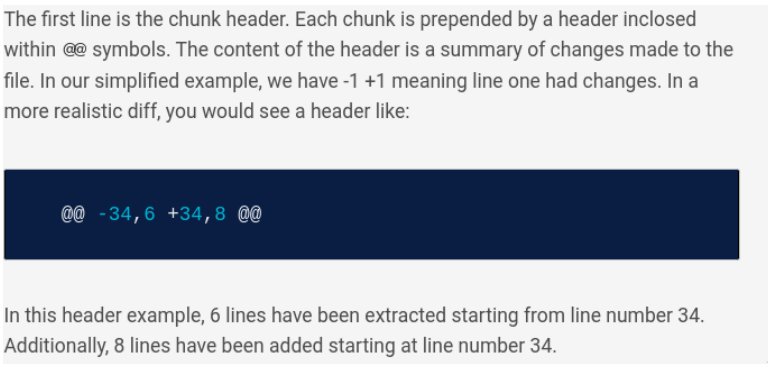
- `git diff` View difference between Stage and Working Directory
- `git diff --staged` View difference between HEAD and Stage
- `git diff HEAD` View difference between HEAD and Working Directory



But lets say you want to look at the difference between two files(later more uses)(only considers non-staged files)
how to diff staged file `git diff --stages`
TODO include example

```
git diff <file-1> <file-2>
```

This creates a diff of the current aka. latest comited version against the one version you specified i.e. every line with a `-` does not exist in your current but in the old, and every line that starts with a `+` does exist in the current but not the old version. Explain this correct
https://www.atlassian.com/git/tutorials/saving-changes/git-diff


The first line is the chunk header. Each chunk is prepended by a header inclosed within @@ symbols. The content of the header is a summary of changes made to the file. In our simplified example, we have -1 +1 meaning line one had changes. In a more realistic diff, you would see a header like:

```
@@ -34,6 +34,8 @@
```

In this header example, 6 lines have been extracted starting from line number 34. Additionally, 8 lines have been added starting at line number 34.

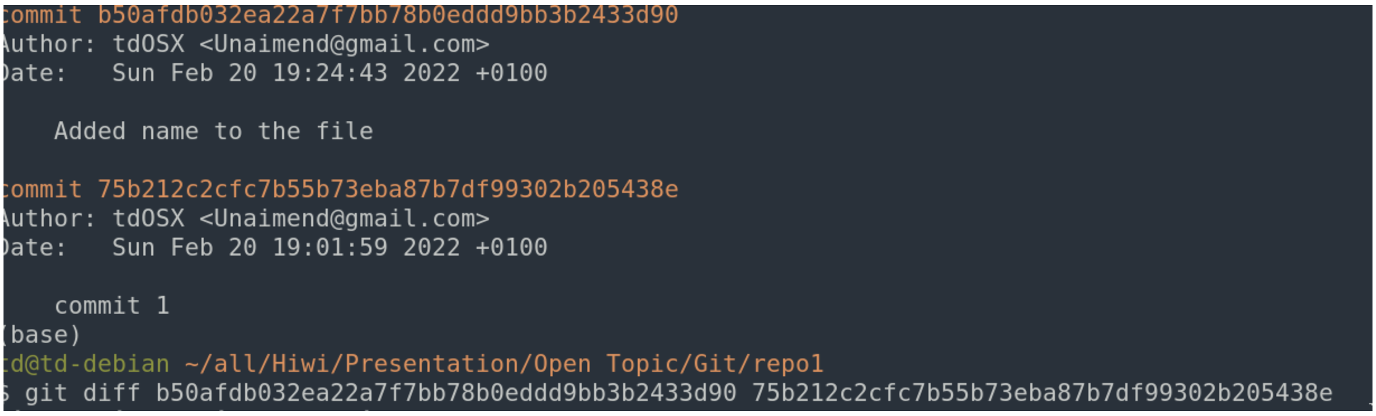
TODO git with HEAD
TODO git with uncommited changed

By default i.e. if you call `git diff` without any arguments it will show you all uncommited changes since the last commit.

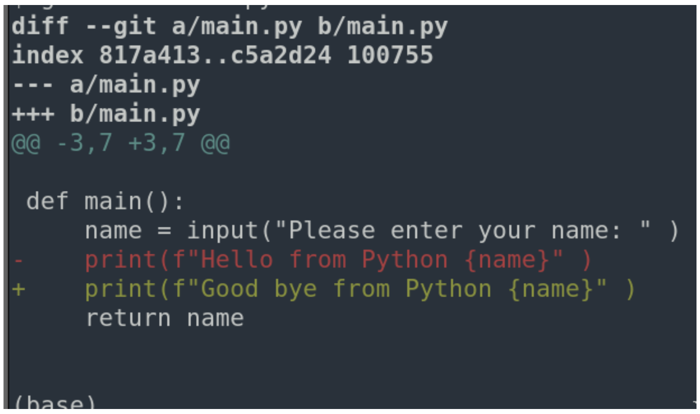
I you want to look at all the commits you did you can call git log

```
git log
```

Here you can also see the specific hashes of each commit, to for example look at a diff of two specific commits(will only show commits of the current branch, we will cover branches in a few minutes)

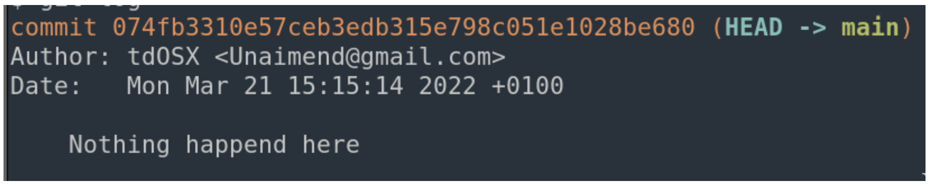

```
ommit b50afdb032ea22a7f7bb78b0eddd9bb3b2433d90
uthor: tdOSX <Unaimend@gmail.com>
ate:   Sun Feb 20 19:24:43 2022 +0100

    Added name to the file

ommit 75b212c2cfc7b55b73eba87b7df99302b205438e
uthor: tdOSX <Unaimend@gmail.com>
ate:   Sun Feb 20 19:01:59 2022 +0100

    commit 1
base)
d@td-debian ~/all/Hiwi/Presentation/Open Topic/Git/repo1
 git diff b50afdb032ea22a7f7bb78b0eddd9bb3b2433d90 75b212c2cfc7b55b73eba87b7df99302b205438e
```

But what happens if you make a mistake i.e. you want to delete changes to the files or maybe even undo whole commits or you just want to revisit an old commit, maybe because you found some weird behavior in the behavior of the program you are working on at the moment
Lets say you changed the line down below from `Hello` to `Goodbye` but you are not sure if that was always the case, and you also don't know which commit might have changed the behavior of the program



```
diff --git a/main.py b/main.py
index 817a413..c5a2d24 100755
--- a/main.py
+++ b/main.py
@@ -3,7 +3,7 @@

 def main():
     name = input("Please enter your name: " )
-    print(f"Hello from Python {name}" )
+    print(f"Good bye from Python {name}" )
     return name


(base)
```

You look at the last message(remember `git log`) but the author clearly stated that nothing happened in this commit
so your try a random earlier commit or one for which you remember that the program was still working there(maybe you even wrote some test)

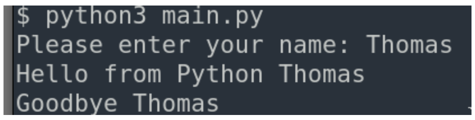


```
commit 074fb3310e57ceb3edb315e798c051e1028be680 (HEAD -> main)
Author: tdOSX <Unaimend@gmail.com>
Date:   Mon Mar 21 15:15:14 2022 +0100

    Nothing happend here
```

So you do

```
git checkout 5571c51f41733de093ed178ce4291fd1d5471810
```

You try out the program and everything is correct, so lets do a diff



```
$ python3 main.py
Please enter your name: Thomas
Hello from Python Thomas
Goodbye Thomas
```

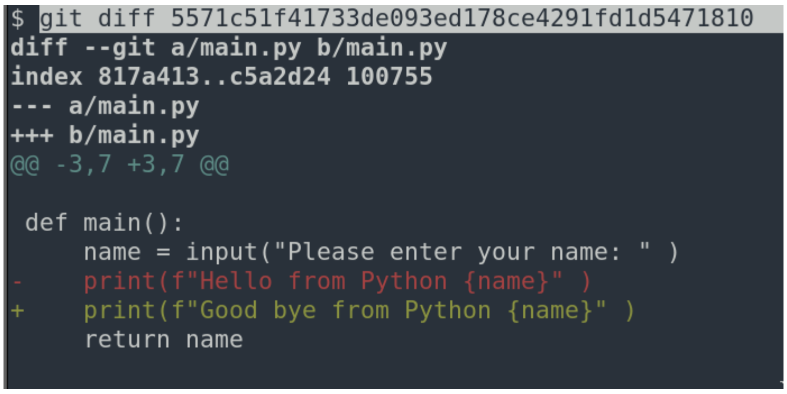
You go back to main

```
git checkout main
```

and you dou

```
git diff 5571c51f41733de093ed178ce4291fd1d5471810
```

And you easily found the mistake, so it was the last commit after all, but how we get rid of the last commit



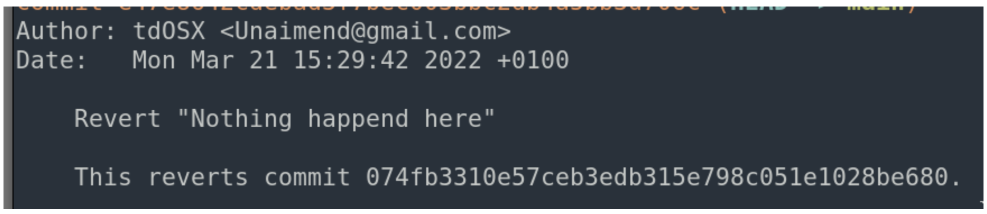
```
$ git diff 5571c51f41733de093ed178ce4291fd1d5471810
diff --git a/main.py b/main.py
index 817a413..c5a2d24 100755
--- a/main.py
+++ b/main.py
@@ -3,7 +3,7 @@

 def main():
     name = input("Please enter your name: " )
-    print(f"Hello from Python {name}" )
+    print(f"Good bye from Python {name}" )
     return name
```

There three ways of doing this `git checkout`, `git revert`, `git reset`

Lets start with `git revert`, to revert the last commit just write

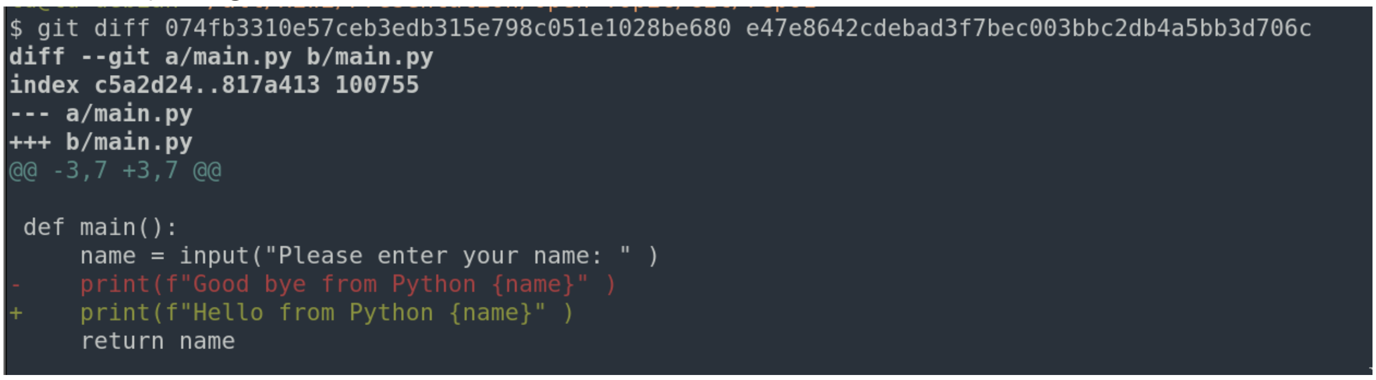
```
git revert HEAD
```

We will talk about HEAD later.



```
Author: tdOSX <Unaimend@gmail.com>
Date:   Mon Mar 21 15:29:42 2022 +0100

    Revert "Nothing happend here"

    This reverts commit 074fb3310e57ceb3edb315e798c051e1028be680.
```

And everything is back in order



```
$ git diff 074fb3310e57ceb3edb315e798c051e1028be680 e47e8642cdebad3f7bec003bbc2db4a5bb3d706c
diff --git a/main.py b/main.py
index c5a2d24..817a413 100755
--- a/main.py
+++ b/main.py
@@ -3,7 +3,7 @@

 def main():
     name = input("Please enter your name: " )
-    print(f"Good bye from Python {name}" )
+    print(f"Hello from Python {name}" )
     return name
```

TODO git clean
TODO git reset

1. `git reset --soft HEAD^1`
   TODO git rm
   TODO Undoing uncomited changes
   https://www.atlassian.com/git/tutorials/undoing-changes

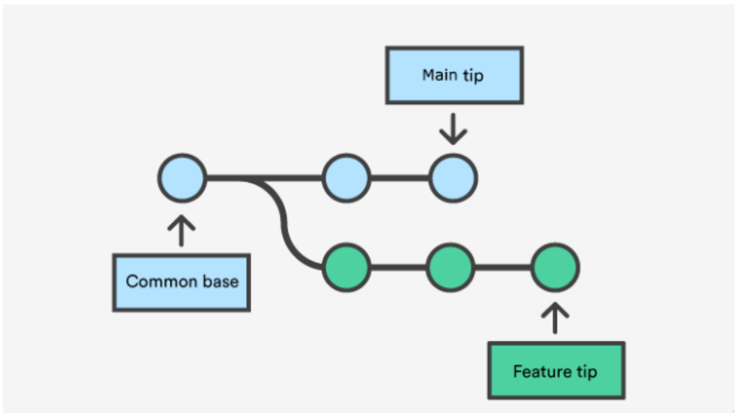
Branching

Lets say you want to implement a new feature which will take a while and might leave your program in an "unusable" state, but you still want to be able to run the current version of your program. There ofc is a feature for that called branching `git branch \<branch name>`

This creates a new branch called <branch-name>

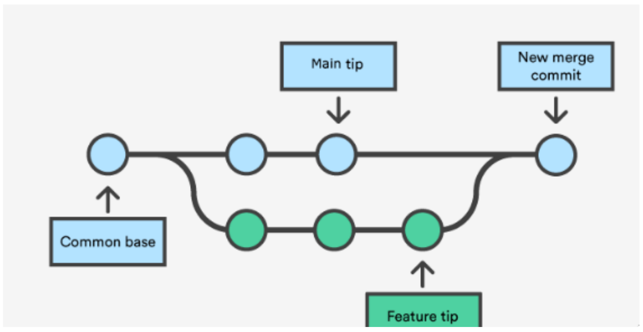
Lets implement a class to store information about the user as well as ask for their age to do this we switch to a new branch

TODO Picture about tree



```
git branch feature-age #this only creates a branch, but we are still on main.
see git status
git checkout feature-age
```

```python
#!/usr/bin/python3

class User():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def toString(self):
        return f"Hello {self.name} you are {self.age} years old"

def main():
    name = input("Please enter your name: " )
    age = input("Please enter your age: " )
    user = User(name, age)

    print(user.toString())

    return name

def sayGoodBye(name):
    print(f"Goodbye {name}")


if __name__ ==  "__main__":
    name = main()
    sayGoodBye(name)
```

After doing implementing our new feature, we of course want to bring the changes to the main working branch
this brings us to `git merge`
`Git merge` will combine multiple sequences of commits into one unified history. In the most frequent use cases, `git merge` is used to combine two branches.


Execute `git status` to ensure that `HEAD` is pointing to the correct merge-receiving branch.

```
git checkout main
```

```
git merge age-feature
```

Now the changed made on the *age-feature* branch are merged onto the new and you can delete the *feature branch*

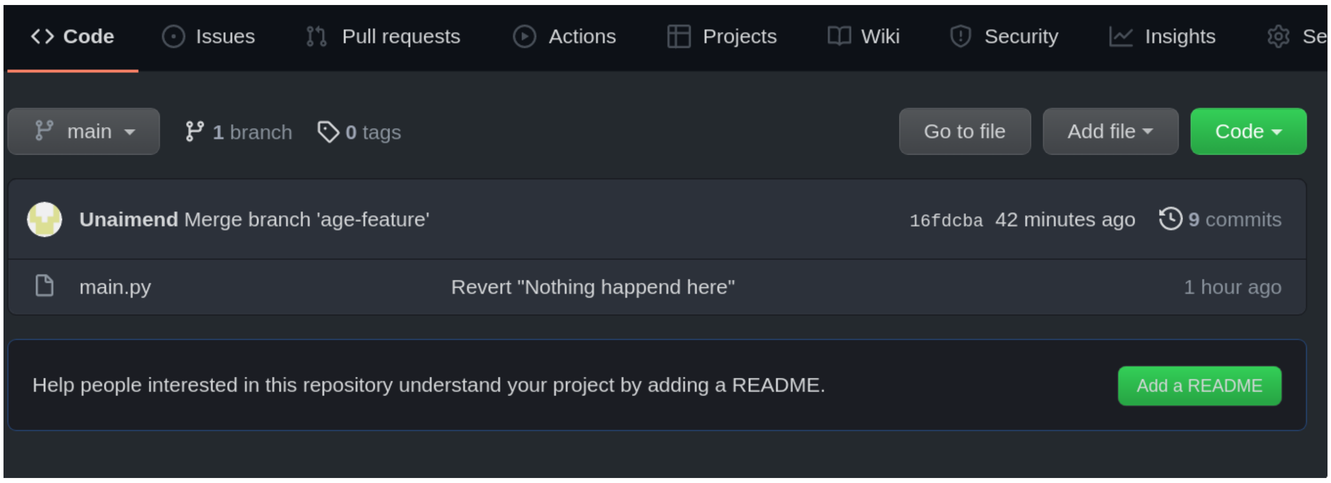
```
git branch -d age-feature
```

-d is safe delete i.e. if there are unmerged changed on the feature branch the delete will fail, if you want to delete the branch anyway there is *-D*

Git collaboration

We know how to

- execute basic git commands
- create/delete branches
  Those are the basics of working locally. But you are probably asking yourself: Whats up with GitLab and GitHub and how can I backup my files,
  because atm all of our files are still only stored on our local computer i.e. if the drive fails all of our data is gone. So lets talk about GitLab



Unaimend Merge branch 'age-feature' 16fdcba 42 minutes ago 9 commits

main.py Revert "Nothing happend here" 1 hour ago

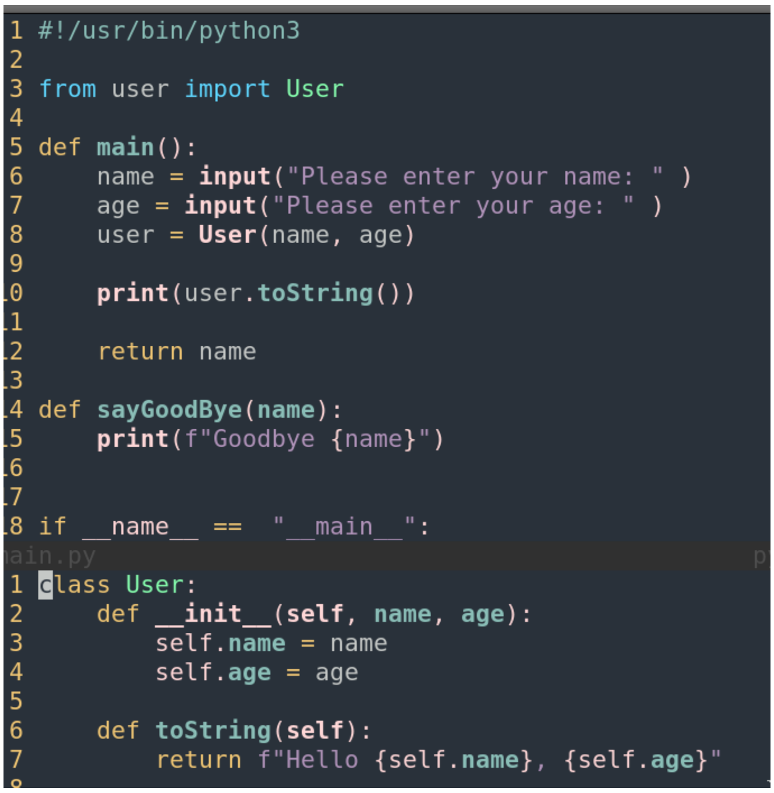
Help people interested in this repository understand your project by adding a README.

Now lets clone the repo into a new folder to simulate another person which whom you are collaborating
We now have the same files/same repo is the two folders *repo1* and *repo2*

```
git remove add git@github.com:Unaimend/test.git
```

```
git pull -u origin man
```

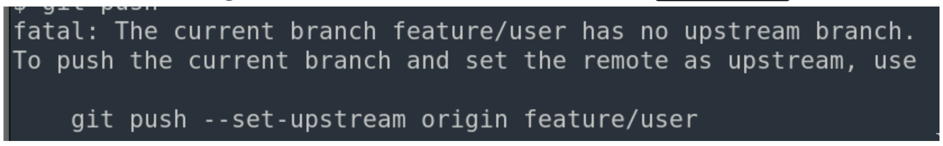
Now image the other persons thinks it would be a good idea to move the user class to a seperate file



```
#!/usr/bin/python3

from user import User

def main():
    name = input("Please enter your name: " )
    age = input("Please enter your age: " )
    user = User(name, age)

    print(user.toString())

    return name

def sayGoodBye(name):
    print(f"Goodbye {name}")


if __name__ == "__main__":
```

```
class User:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def toString(self):
        return f"Hello {self.name}, {self.age}"
```

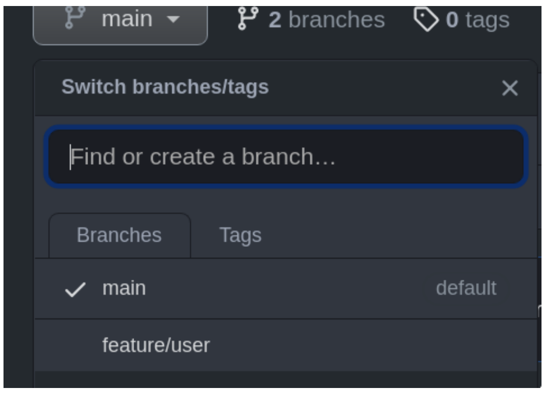
Further imagine your collaborater wants to have lunch, but before this he wants to save his changes online, for this he uses `git push`



```
fatal: The current branch feature/user has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature/user
```

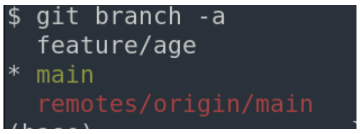
```
git push
```

Now it is possible to see the new changes online and everything is safe, so he can go to lunch



But if you look at your own directory

```
git branch -a #also shows remote branches
```



```
$ git branch -a
  feature/age
* main
  remotes/origin/main
```

you can see that you currently do not have the user branch in your repository, also there is no user.py and your main.py looks like before

```
git pull
```

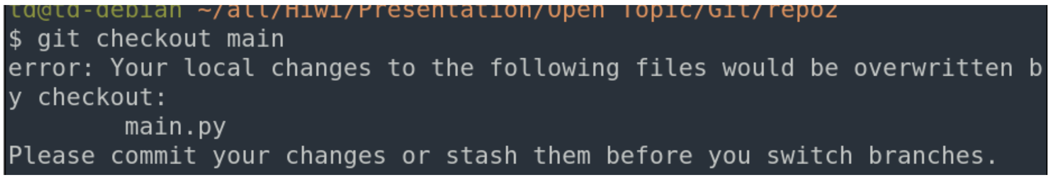
Executing git pull now synchs your local copy with the online repository and you can checkout the user branch
Now you now the basics of collaborating with other people via git(hub)

We will now talk about a few more advanced features

git stash

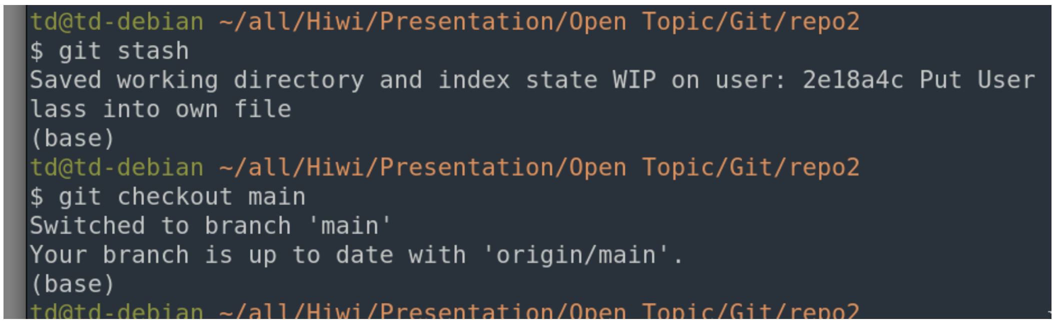
Lets imagine your are working on some feature and a colleague asks you to look at what he did.
So you pull down his changes with *git pull*
And try to switch to his branch and you see this, git asks to you to commit or stash(?)

your changes, your of couse do not wan't to make a commit be cause you did not name any meaning full change



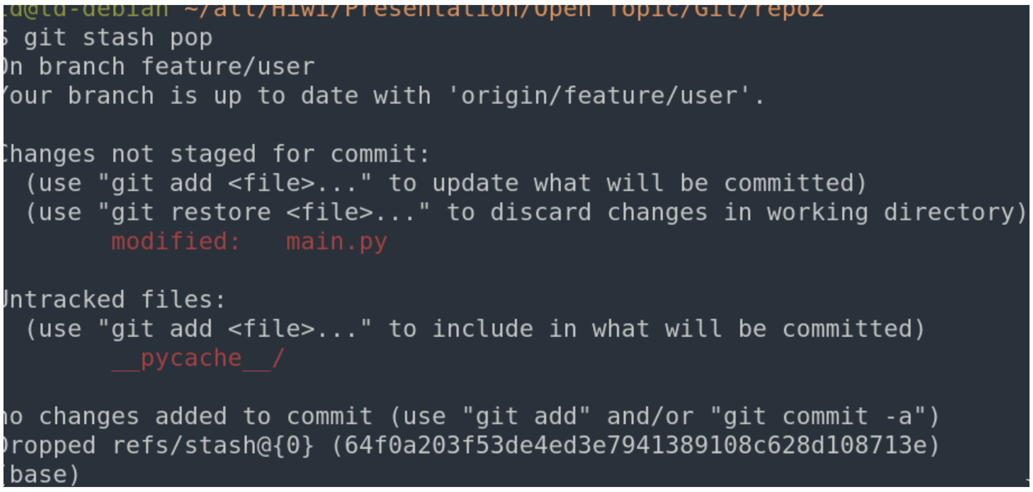
```
td@td-debian ~/all/Hiwi/Presentation/Open Topic/Git/repo2
$ git checkout main
error: Your local changes to the following files would be overwritten b
y checkout:
        main.py
Please commit your changes or stash them before you switch branches.
```

The `git stash` command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy. Now you are able to checkout his changes.



```
td@td-debian ~/all/Hiwi/Presentation/Open Topic/Git/repo2
$ git stash
Saved working directory and index state WIP on user: 2e18a4c Put User
lass into own file
(base)
td@td-debian ~/all/Hiwi/Presentation/Open Topic/Git/repo2
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
(base)
td@td-debian ~/all/Hiwi/Presentation/Open Topic/Git/repo2
```

And go back to your branch and continue to work as if nothing hapend



```
td@td-debian ~/all/Hiwi/Presentation/Open Topic/Git/repo2
$ git stash pop
On branch feature/user
Your branch is up to date with 'origin/feature/user'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   main.py

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

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (64f0a203f53de4ed3e7941389108c628d108713e)
(base)
```

Lets revert the changes with

```
git restore main.py
```

By default, running `git stash` will stash:

- changes that have been added to your index (staged changes)
- changes made to files that are currently tracked by Git (unstaged changes)

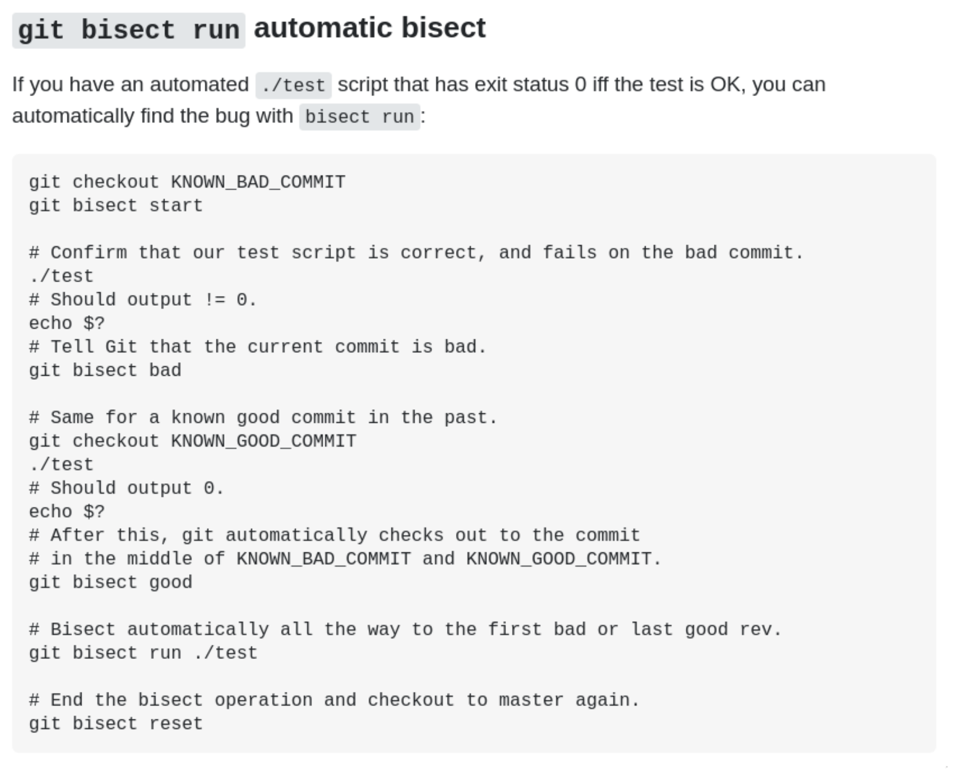
But it will **not** stash:

- new files in your working copy that have not yet been staged
- files that have been ignored

For more on git stash see https://www.atlassian.com/git/tutorials/saving-changes/git-stash

git bisect

Automatic git bisect



`git bisect run` automatic bisect

If you have an automated `./test` script that has exit status 0 iff the test is OK, you can automatically find the bug with `bisect run`:

```
git checkout KNOWN_BAD_COMMIT
git bisect start

# Confirm that our test script is correct, and fails on the bad commit.
./test
# Should output != 0.
echo $?
# Tell Git that the current commit is bad.
git bisect bad

# Same for a known good commit in the past.
git checkout KNOWN_GOOD_COMMIT
./test
# Should output 0.
echo $?
# After this, git automatically checks out to the commit
# in the middle of KNOWN_BAD_COMMIT and KNOWN_GOOD_COMMIT.
git bisect good

# Bisect automatically all the way to the first bad or last good rev.
git bisect run ./test

# End the bisect operation and checkout to master again.
git bisect reset
```

git difftool

merge conflicts

git merge in-depth

TODO Merge conflicts
https://www.atlassian.com/git/tutorials/using-branches/git-merge
https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts
https://www.atlassian.com/git/tutorials/using-branches/merge-strategy

TODO: Tools
GitKraken
GitHubDesktop
SourceTree
Magit(Emacs)

Cheatsheets

"SWTM-2088_Atlassian-Git-Cheatsheet.pdf#page=1" is not created yet. Click to create.

```
git init <repo_name2>
```

To watch
Minute 30
https://www.youtube.com/watch?v=RGOj5yH7evk&ab_channel=freeCodeCamp.org
Not started
https://www.youtube.com/watch?v=Uszj_k0DGsg&ab_channel=freeCodeCamp.org
Not started

Reading List
https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent
https://www.atlassian.com/git/tutorials/advanced-overview
https://www.atlassian.com/git/articles/core-concept-workflows-and-tips
https://www.atlassian.com/git/articles/alternatives-to-git-submodule-git-subtree
https://www.atlassian.com/git/articles/git-team-workflows-merge-or-rebase
https://www.atlassian.com/git/articles/simple-git-workflow-is-simple
https://www.atlassian.com/git/articles/simple-git-workflow-is-simple
https://www.atlassian.com/git/tutorials/comparing-workflows
https://docs.dolthub.com/introduction/what-is-dolt

Jan

TODO git tag

Semver

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards compatible manner, and
3. PATCH version when you make backwards compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

For this system to work, you first need to declare a public API. This may consist of documentation or be enforced by the code itself. Regardless, it is important that this API be clear and precise. Once you identify your public API, you communicate changes to it with specific increments to your version number.

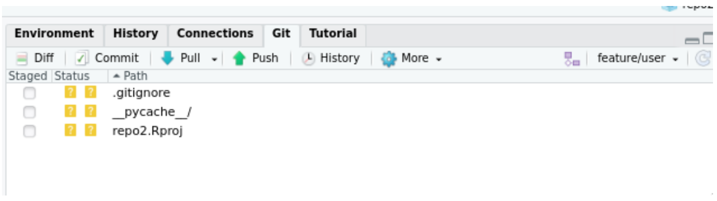
Bug fixes not affecting the API increment the patch version, backwards compatible API additions/changes increment the minor version, and backwards incompatible API changes increment the major version.

https://semver.org/

Lena

Work with git and Rstudio

You need to create a **Project** in R, and then at the Tools label you can select version control, but the R git support is quite bad



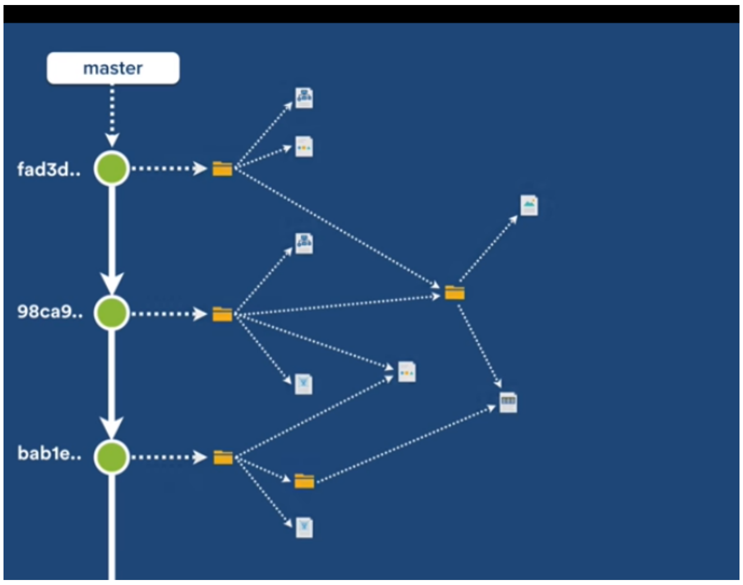
```
.gitignore
__pycache__/
repo2.Rproj
```

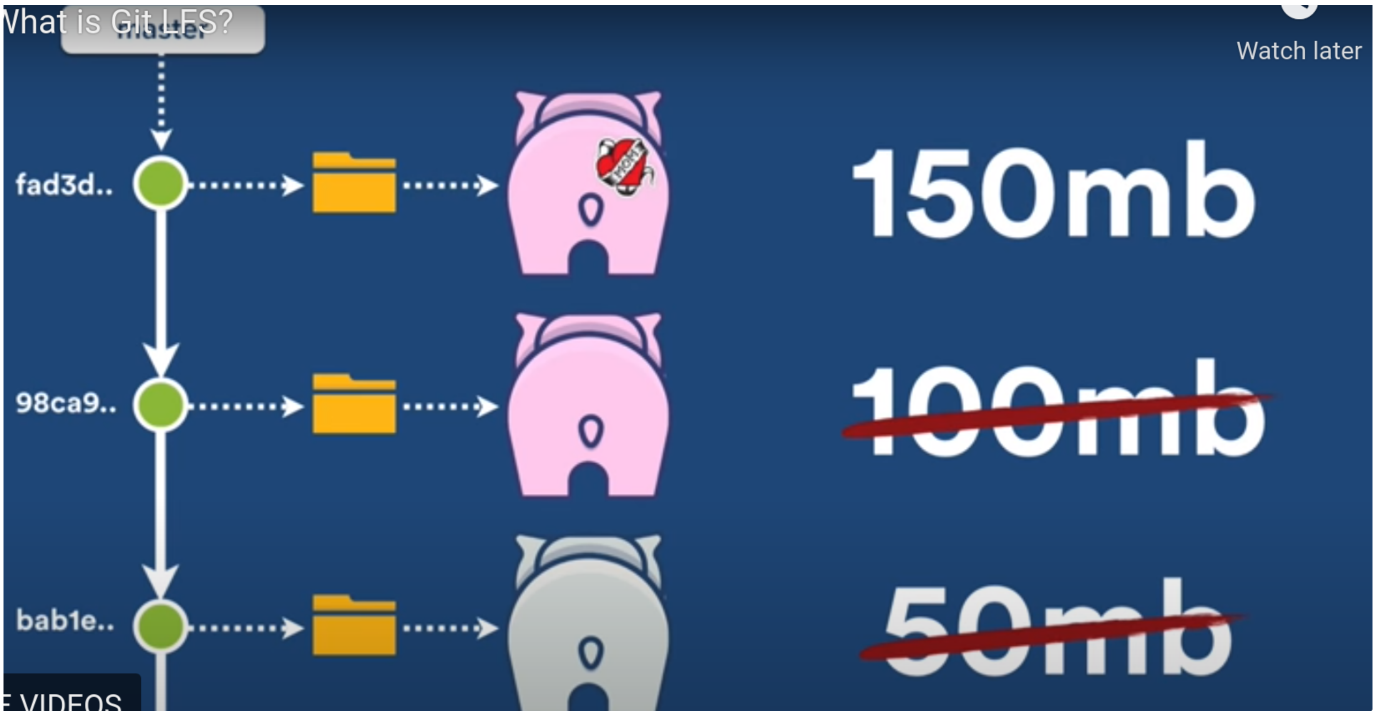
What kind of data to put in to

You can basically put everything in git, but I would be careful if you bug files which are quite big >100mb
Also uploading binary files does not make much sense, you will for example not get meaningful diffs for binary files

git LFS

Git is a *distributed* version control system, meaning the entire history of the repository is transferred to the client during the cloning process. For projects containing large files, particularly large files that are modified regularly, this initial clone can take a huge amount of time, as every version of every file has to be downloaded by the client.





But to be hones I don't think it is that useful for use because we do not work with regular changing big files

Stefano

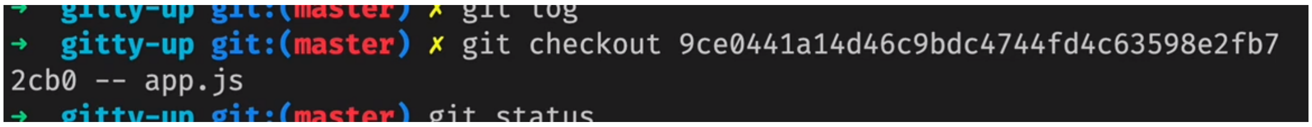
CI/CD
Project management/issues

Git Local Advanced

TODO HEAD
TODO git mv
git rm --cached README
Another useful thing you may want to do is to keep the file in your working tree but

remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files. To do this, use the --cached option:
TODO git diff branch
TODO git rebase
TODO git workflows
TODO git cherrypick https://www.atlassian.com/git/tutorials/cherry-pick
TODO merge vs rebase https://www.atlassian.com/git/tutorials/merging-vs-rebasing
TODO git reflog
TODO .gitignore
TODO git bisect https://git-scm.com/docs/git-bisect
git alias
g git config --global credential.helper cache.

Revert single file



```
→  gitty-up git:(master) ✗ git log
→  gitty-up git:(master) ✗ git checkout 9ce0441a14d46c9bdc4744fd4c63598e2fb7
2cb0 -- app.js
→  gitty-up git:(master) git status
```