Git

- Presented by
- Samarjit Adhikari (samarjit.adhikari@gmail.com)

Git

[Agenda]

- Introduction of GIT/History
- How Git is different from other VCS
- Git commands
- How Git works
- Git Configuration (Command Line)
- Git Configuration with GUI
- Git and GITHUB
- Git and SVN integration
- An example with open source project setup.

Git

1 What is it ?

Git

IS …

- Directory content management system
- Tree history storage system
- Stupid content tracker
- A toolkit

[History]

In 2005, the relationship between the open community that developed the Linux kernel and the commercial company that developed BitKeeper broke down → Results git

Git

[git and other VCS]

- git is not subversion!
- A distributed SCM/VCS
- Snapshots, Not Differences
- Nearly every operation is local
- Git has integrity

Git

2 What is Version control System (VCS)

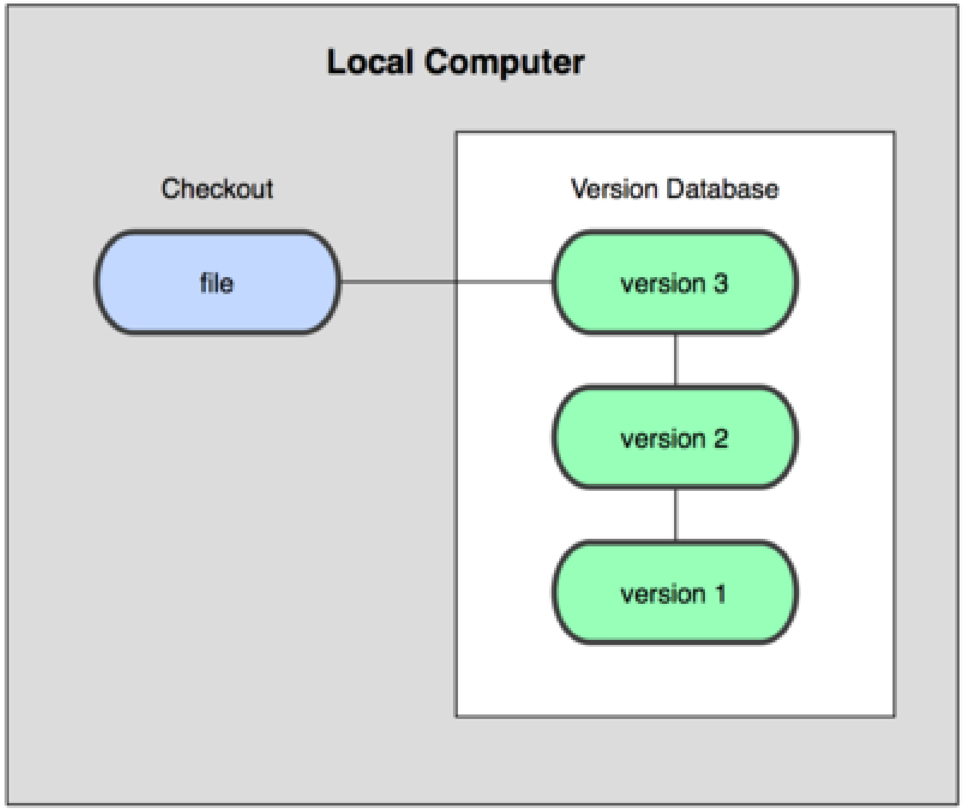
Git

- 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.

- *Local version control system*
- *Centralized version control system*
- *Distributed version control system*

Git

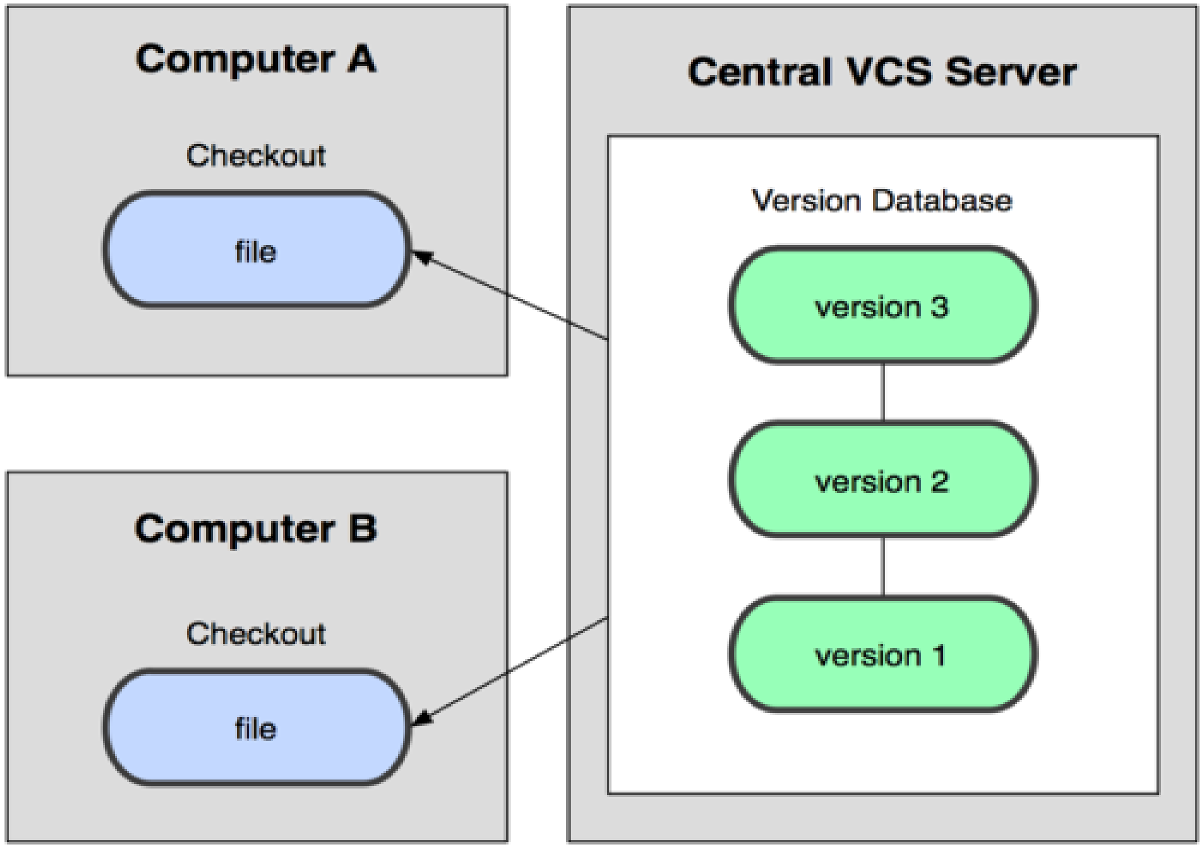
- Local Version Control Systems



e.g. rcs

Git

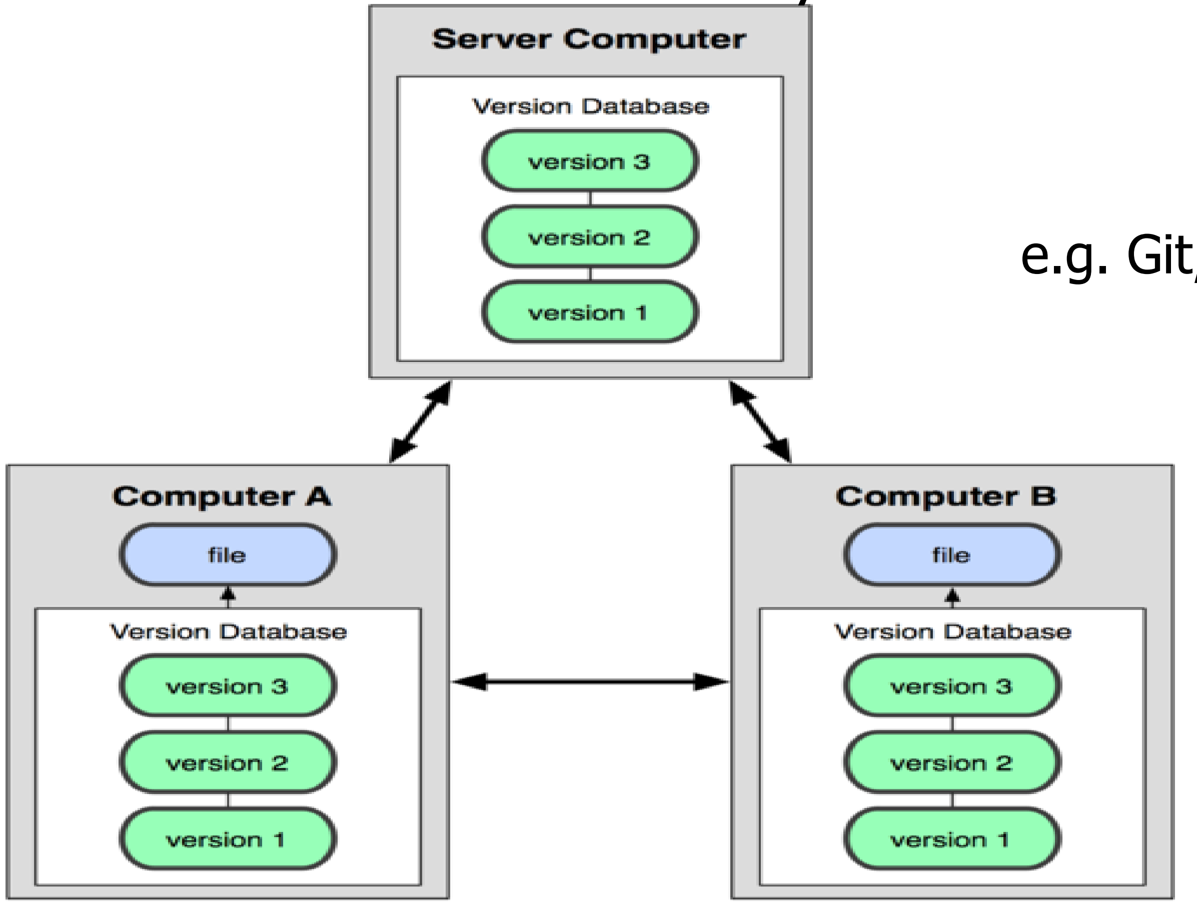
- Centralized Version Control System



e.g. cvs,
Subversion
perforce

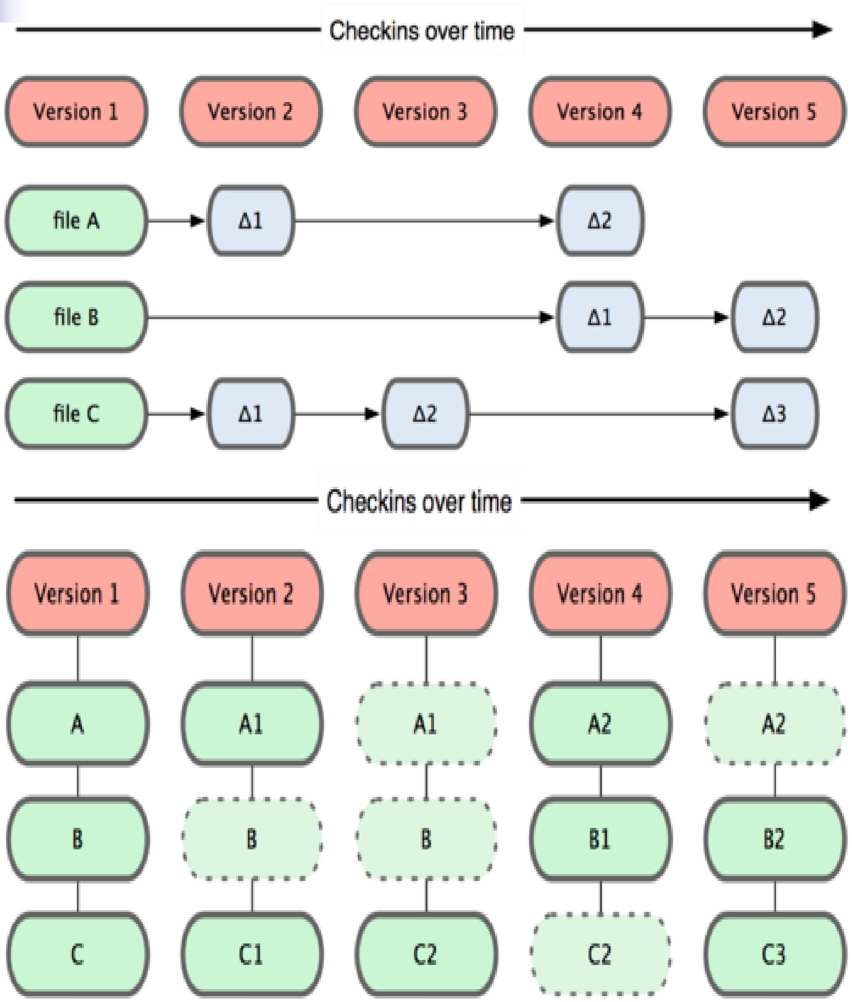
Git

- Distributed Version Control System



e.g. Git, Mercurial

Git



Snapshots, Not Differences

- Git thinks of its data more like a set of snapshots of a miniature filesystem.
- Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
- To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored.
- Git thinks about its data more like a **stream of snapshots**.

Git

Nearly Every operation is local

- Browse the history of the project
- Work offline
- Subversion provides local repositoy but History not being maintained.

Git has integrity

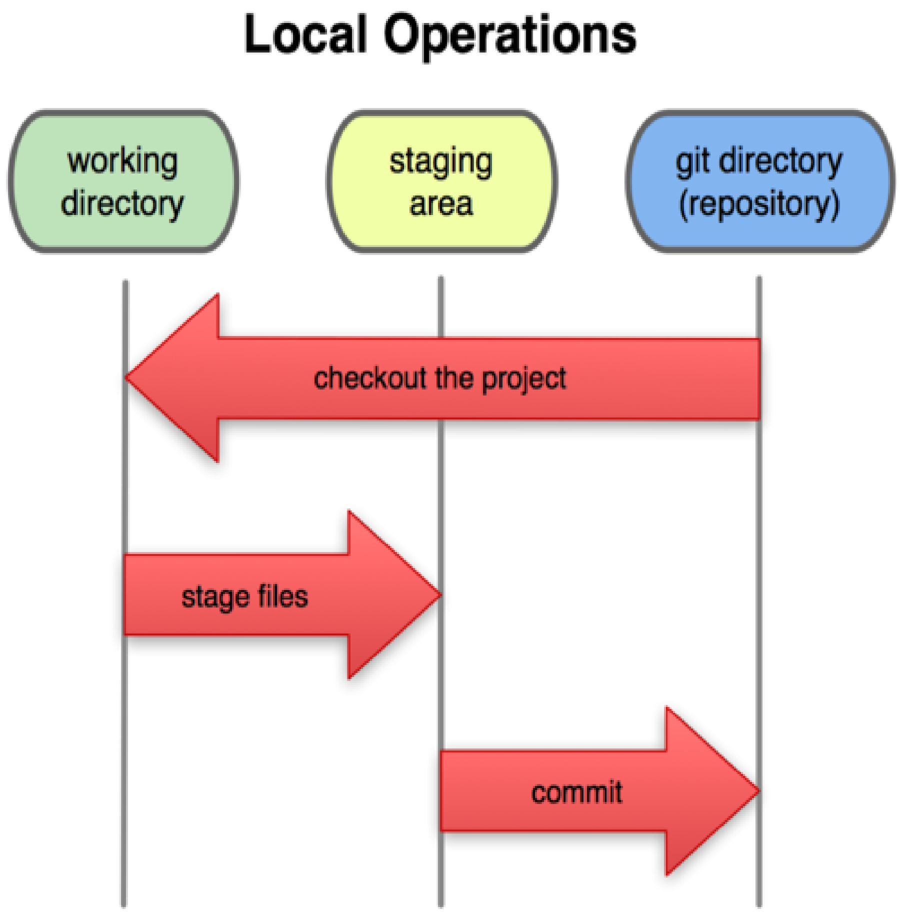
It doesn’t store filename rather stores file content.

Git uses for this checksumming, called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.

A SHA-1 hash looks something like this:

24b9da6552252987aa493b52f8696cd6d3b00373

Git



The Three States

Git has three main states that your files can reside in
: committed, modified, and staged.

- Committed means that the data is safely stored in your local database.
- 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.

Git

3 First-Time Git Setup

Git

- Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

- **/etc/gitconfig** file: Contains values for every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.

- **~/.gitconfig or ~/.config/git/config** file: Specific to your user. You can make Git read and write to this file specifically by passing the --global option.

- **config** file in the Git directory (that is, .git/config) of whatever repository you're currently using: Specific to that single repository.

- Each level overrides values in the previous level, so values in .git/config override those in /etc/gitconfig.

Git

- Set Your Identity

```
$ git config --global user.name "User Name"
$ git config --global user.email username@example.com
```

You need to do this only once if you pass the --global option.

- Set Your Editor

```
$ git config --global core.editor emacs
```

=> On windows x54

```
$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -nosession"
```

- Checking Your Settings

If you want to check your settings, you can use the **git config --list** command to list all the settings Git can find at that point:

```
$ git config --list
user.name=User Name
user.email=username@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
```

Git

4 Git Basics - Getting a Git Repository

Git

There are two main approaches for getting a Git reporsitory

- The first takes an existing project or directory and imports it into Git.
- The second clones an existing Git repository from another server.

=>Initializing a Repository in an Existing Directory

```
$ cd /user/your_repository
```

and type:

```
$ git init
```

This creates a new subdirectory named .git that contains all of necessary repository files. At this point, nothing in your project is tracked yet.

To add your current project files into Git repository ..

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

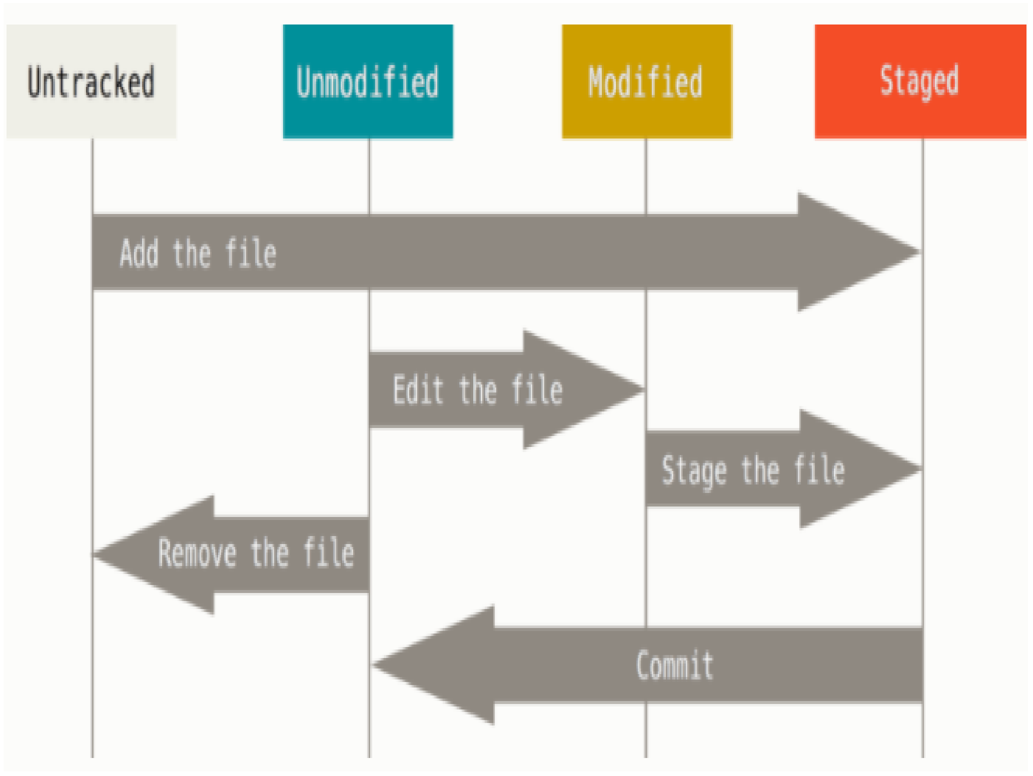
Git

- Cloning an Existing Repository

To get a copy of an existing Git repository use "git clone"
e.g.
$ **git clone** https://github.com/libgit2/libgit2
or
$ **git clone** https://github.com/libgit2/libgit2 mylibgit

Git do supports other protocols like git:// inaddition to http://

Git



- Recording Changes to repository

Each file in working directory can be in one of two states: tracked or untracked.

Tracked files are files that were in the last snapshot, they can be unmodified, modified, or staged.

Untracked files are everything else that were not present in last snapshot and are not present in staging area.

After clone-ing a repository, all of files will be tracked and unmodified because Git just checked them out

Git

A close look on Git Status

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

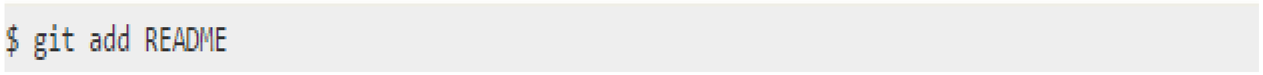
Add a new File

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

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

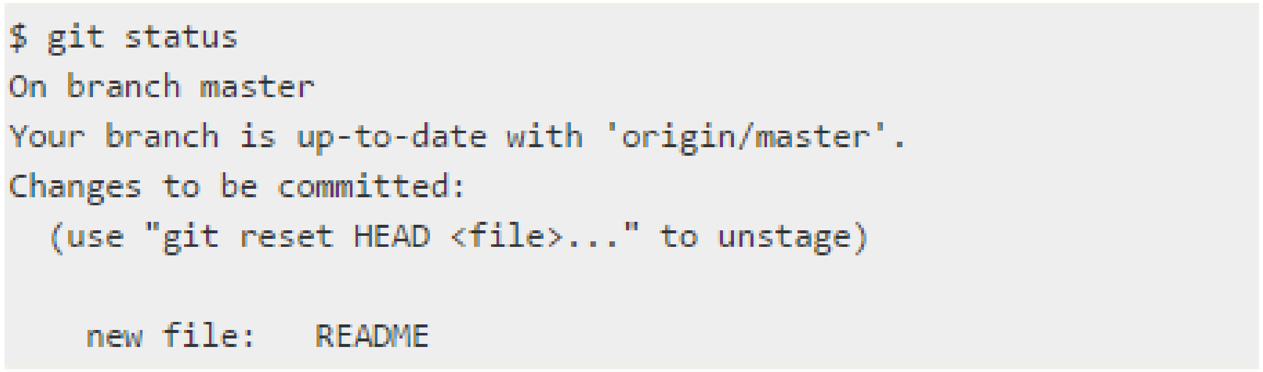
Git

A close look on Git Status

Track a new File



```
$ git add README
```



```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

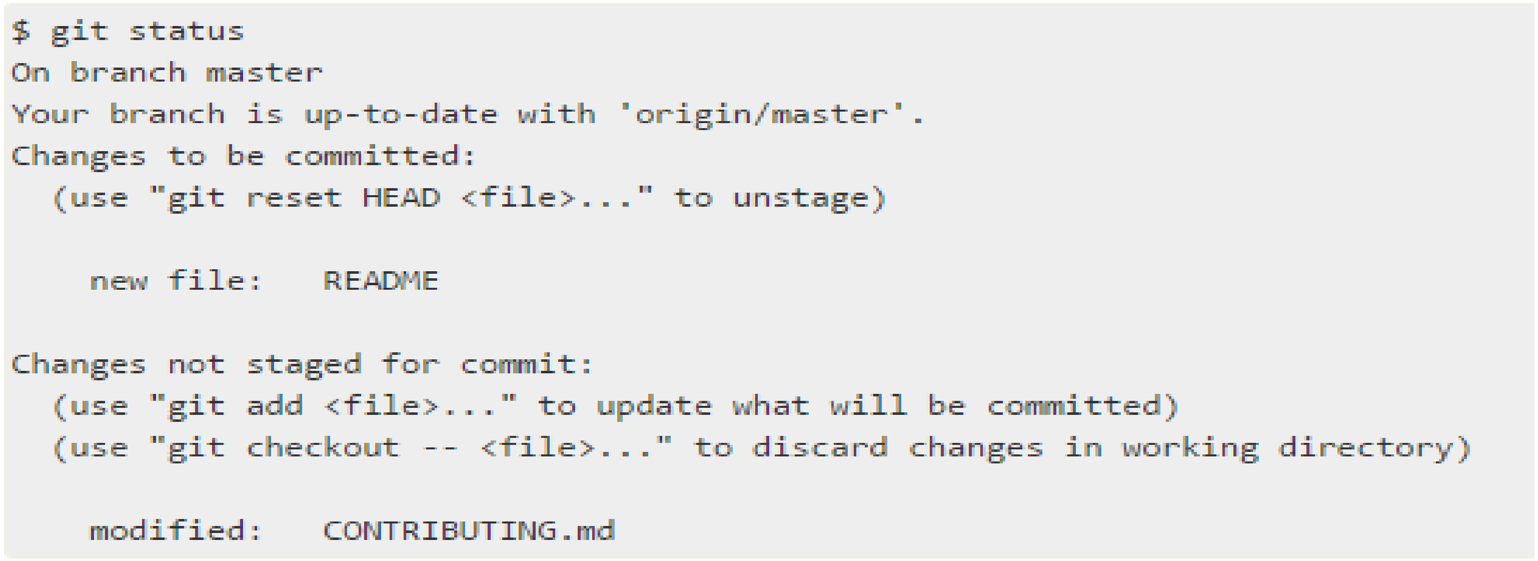
It’s under the “Changes to be committed”

Git

A close look on Git Status

Staging Modified Files

Let’s change a file that was already tracked. If you change a previously tracked file called **CONTRIBUTING.md** and then run your **git status** command again



```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

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

    modified:   CONTRIBUTING.md
```

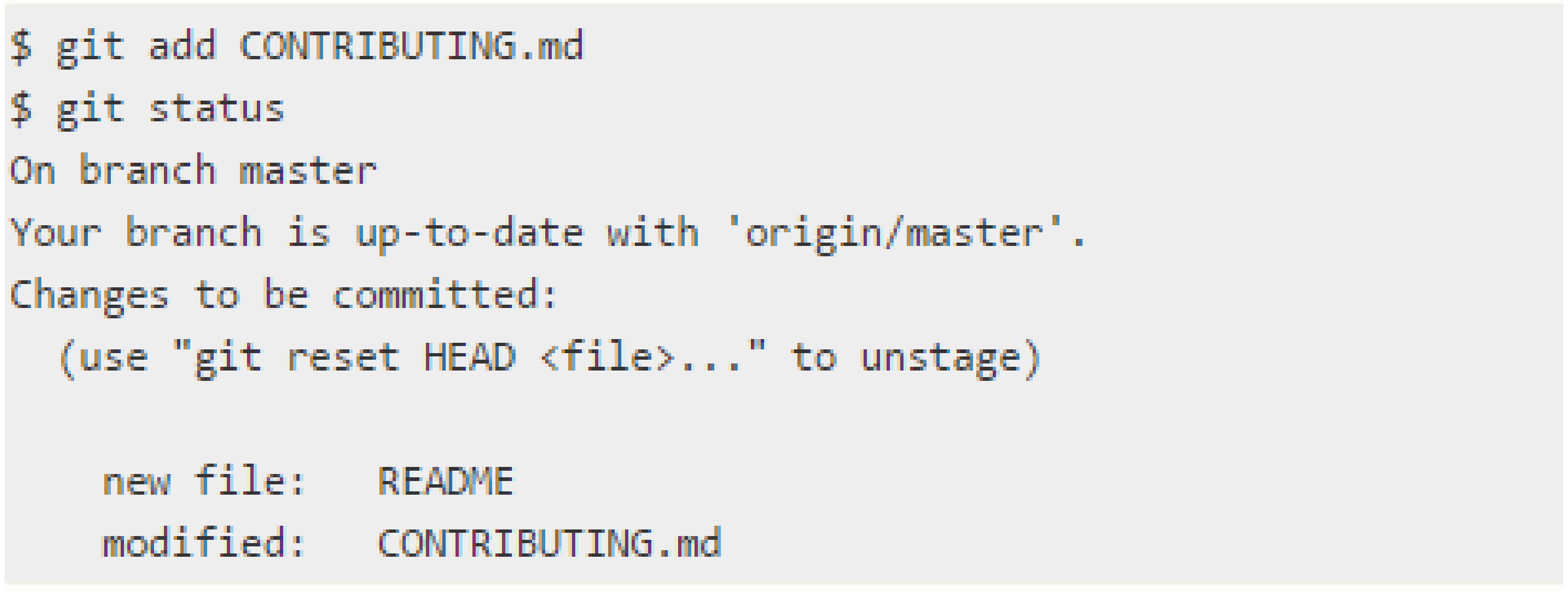
The **CONTRIBUTING.md** file appears under a section named “Changes not staged for commit” – which means that a file that is tracked has been modified in the working directory but not yet staged.

Git

A close look on Git Status

Staging Modified Files

To stage it, you run the git add command. git add is a multipurpose command



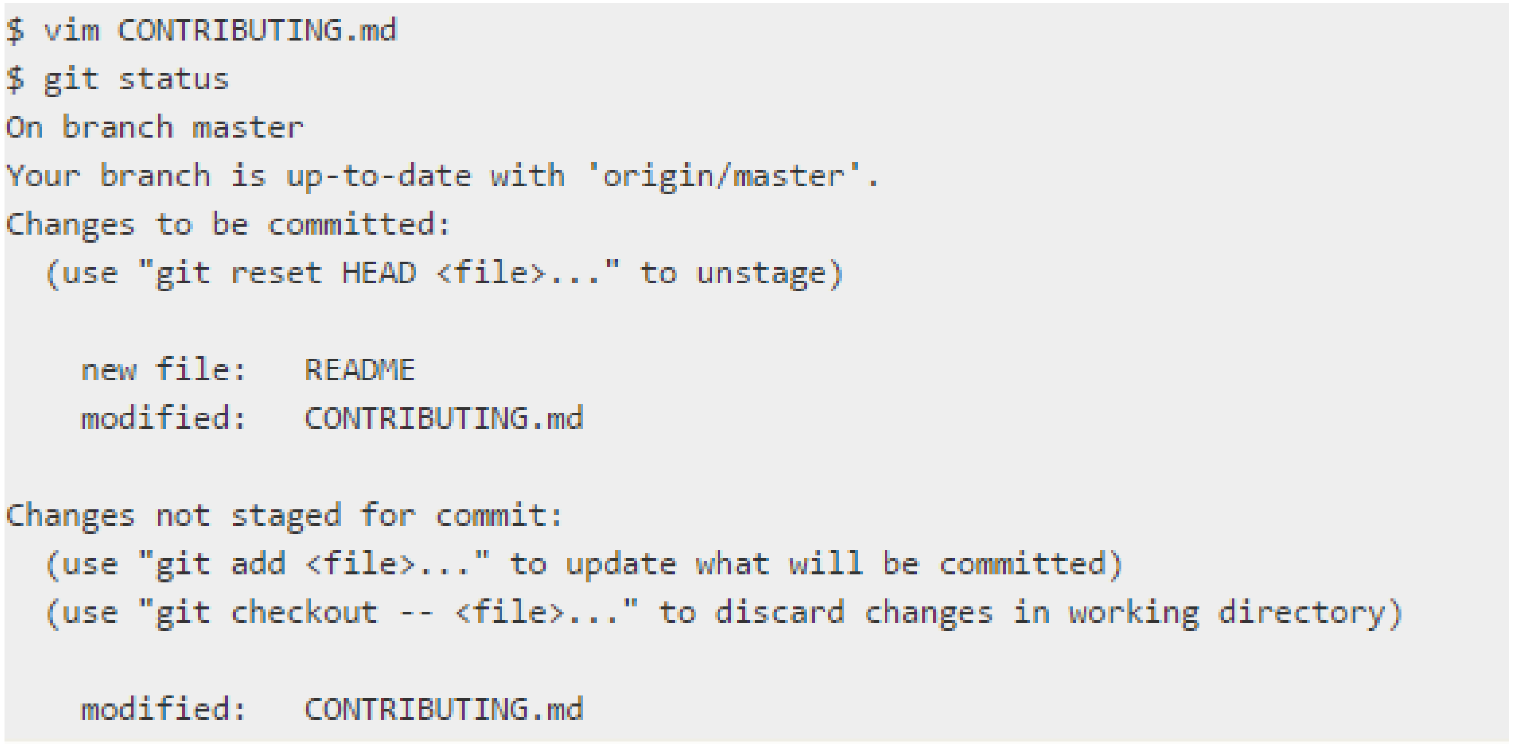
```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Git

A close look on Git Status

Staging Modified Files

After staged , the file CONTRIBUTING is opened and modified before commit



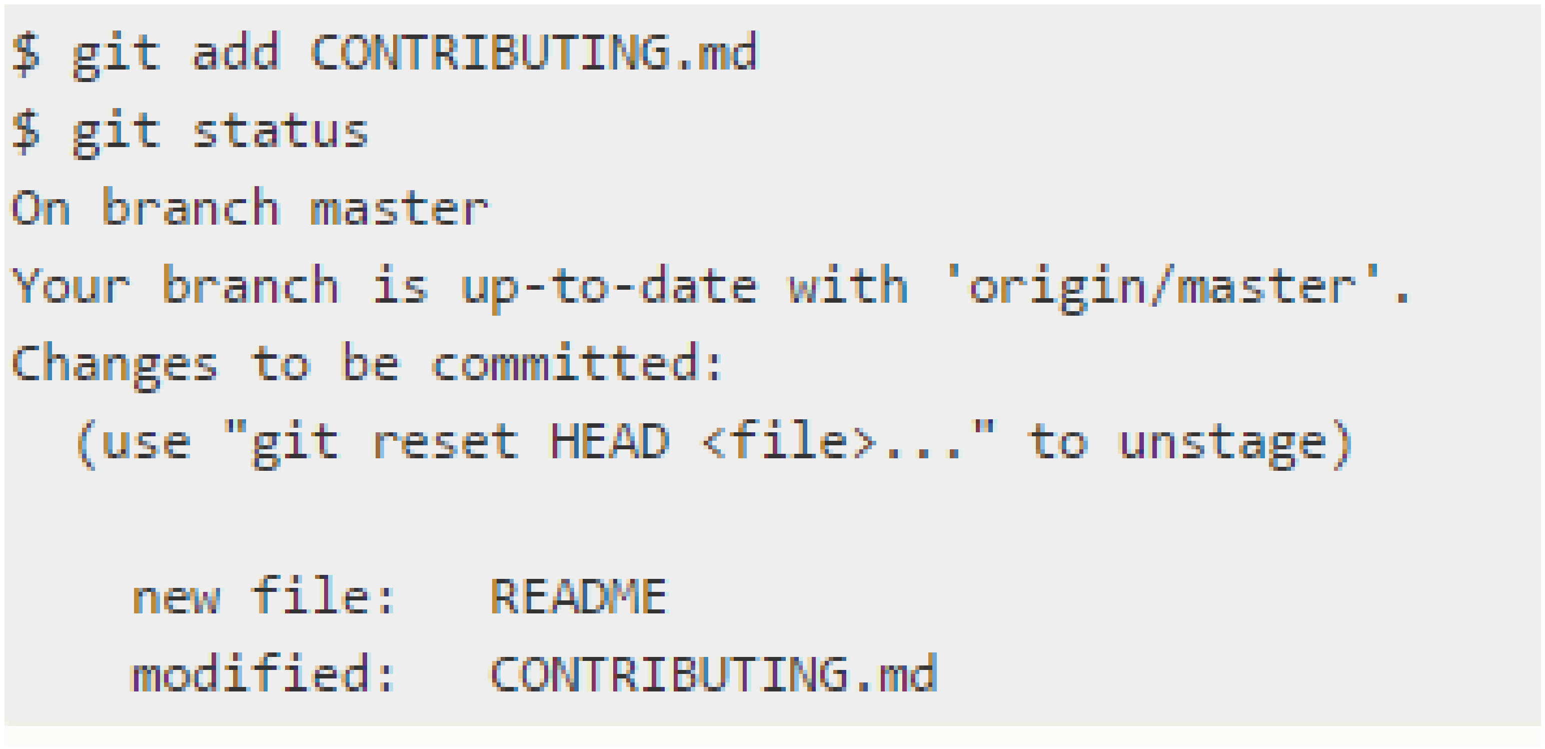
```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

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

    modified:   CONTRIBUTING.md
```

Git

A close look on Git Status

Staging Modified Files

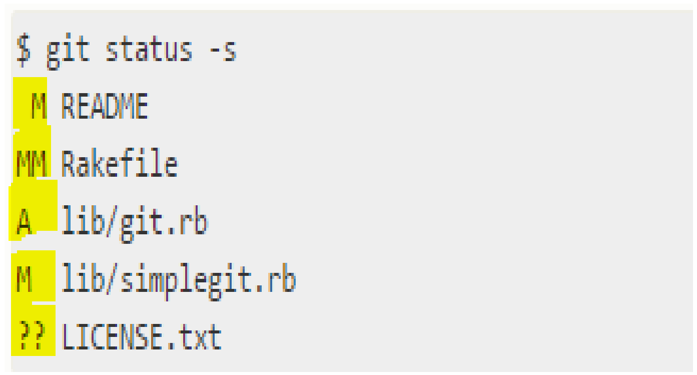
If you modify a file after you run **git add**, you have to run **git add** again to stage the latest version of the file



```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Git

A close look on Git Short Status

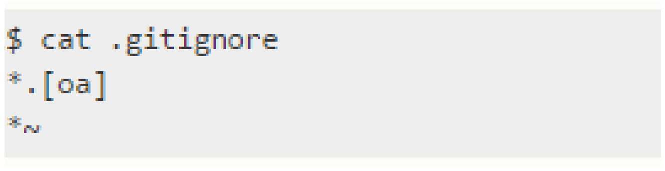


```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

There are two columns to the output –

The left-hand column indicates the status of the staging area

The right-hand column indicates the status of the working tree.



```
$ cat .gitignore
*.[oa]
*~
```

Often, there are class of files that you don't want
Git to automatically add or even show you as being untracked.

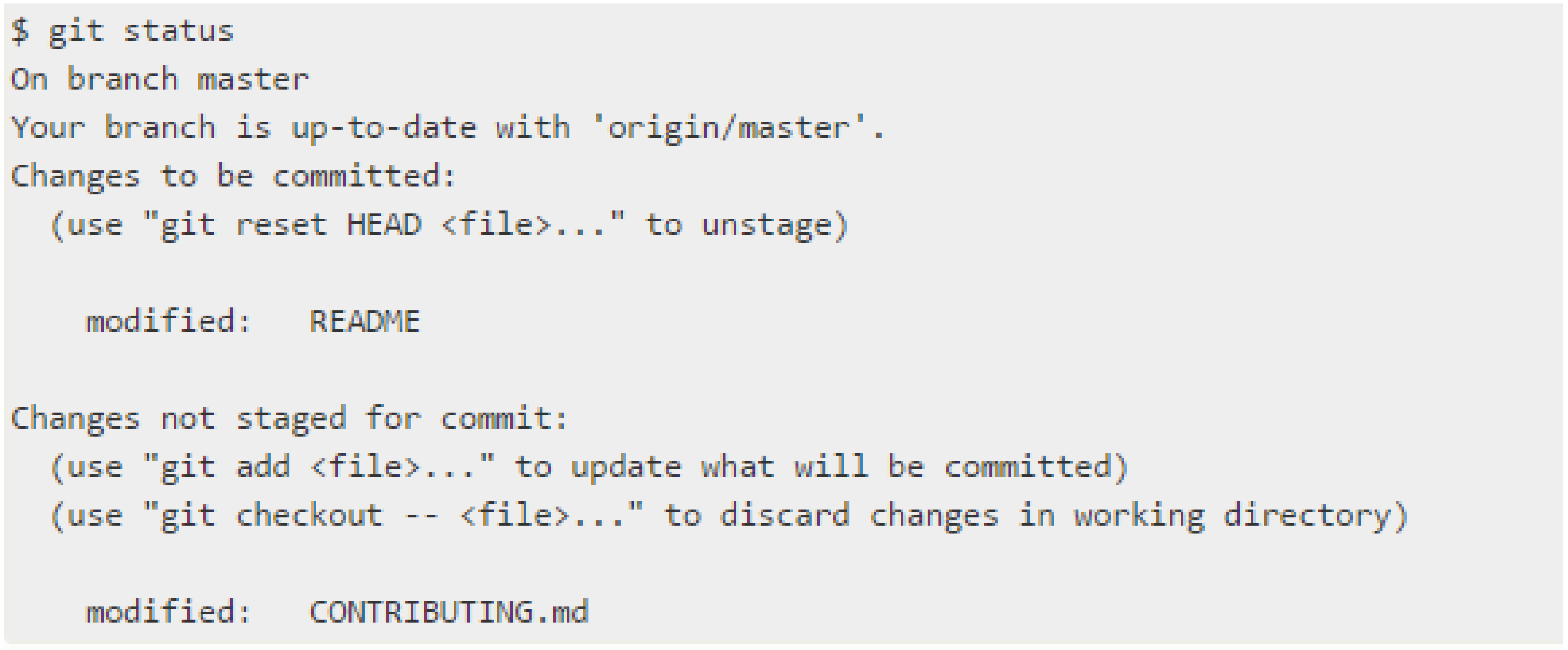
These are generally automatically generated files such as log files or files produced by your build system.

In such cases, you can create a file listing patterns to match them named **.gitignore.**

Git

A close look on Git diff

Let’s say you edit and stage the README file again and then edit the CONTRIBUTING.md file without staging it. If you run your git status command, you once again see something like this



```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README

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

    modified:   CONTRIBUTING.md
```

Git

A close look on Git diff

To see what you’ve changed but not yet staged, type **git diff** with no other arguments:

If you want to see what you’ve staged that will go into your next commit,
you can use **git diff --staged.**
This command compares your staged changes to your last commit:

Committing Your Changes

```
$git commit –m “<Comments>”
```

Skipping the Staging Area

```
$git commit –am “comments”
```

Git

Quick Revisit

- How do you initialize a Git repository?
- How do you determine the current state of the project?
- How to move all your changes since your last commit to the staging area?
- How to store the saved changes in the repository and add a message "first commit"?

Git

Quick Revisit

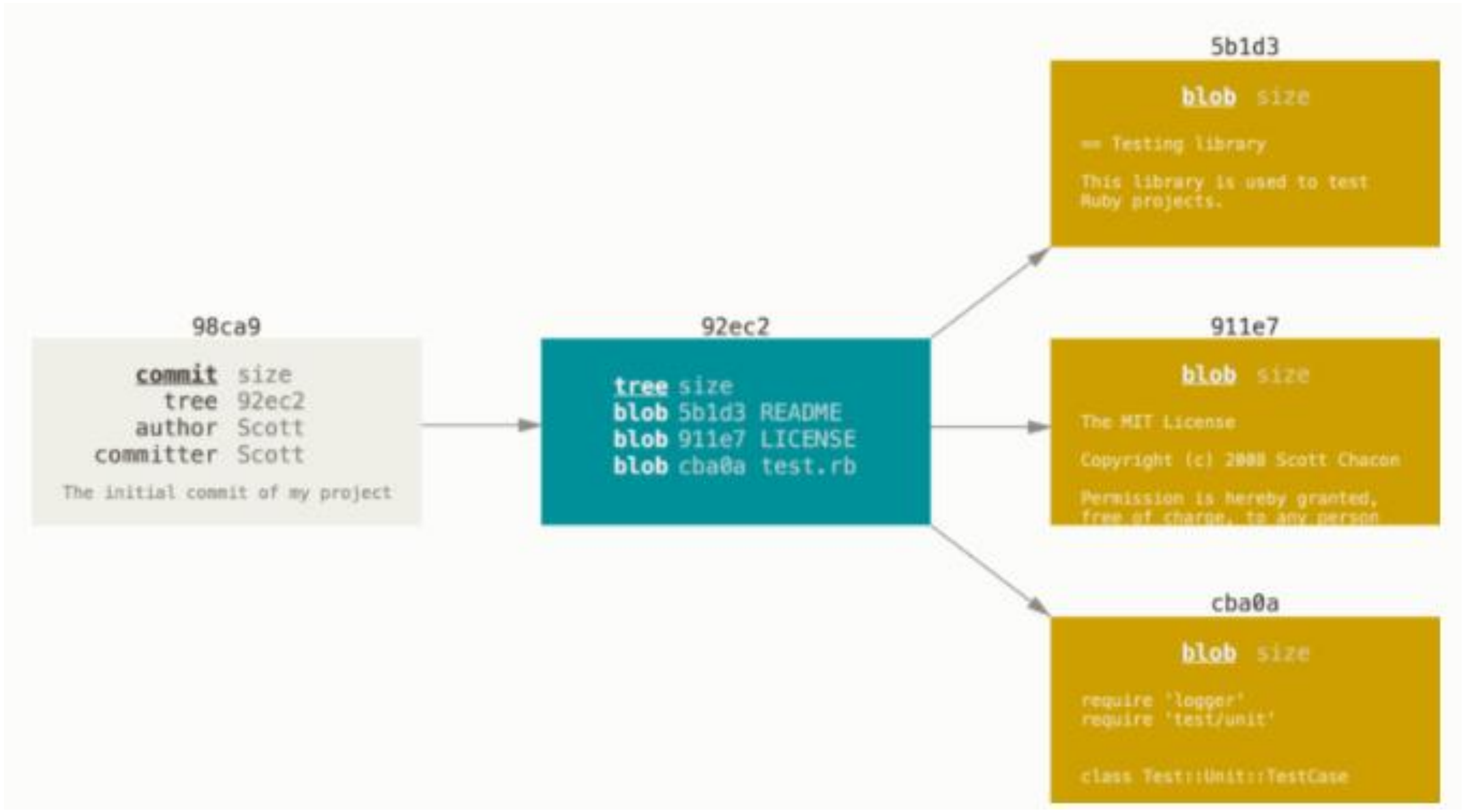
- Q&A

Git

5 Git Branching and Merging

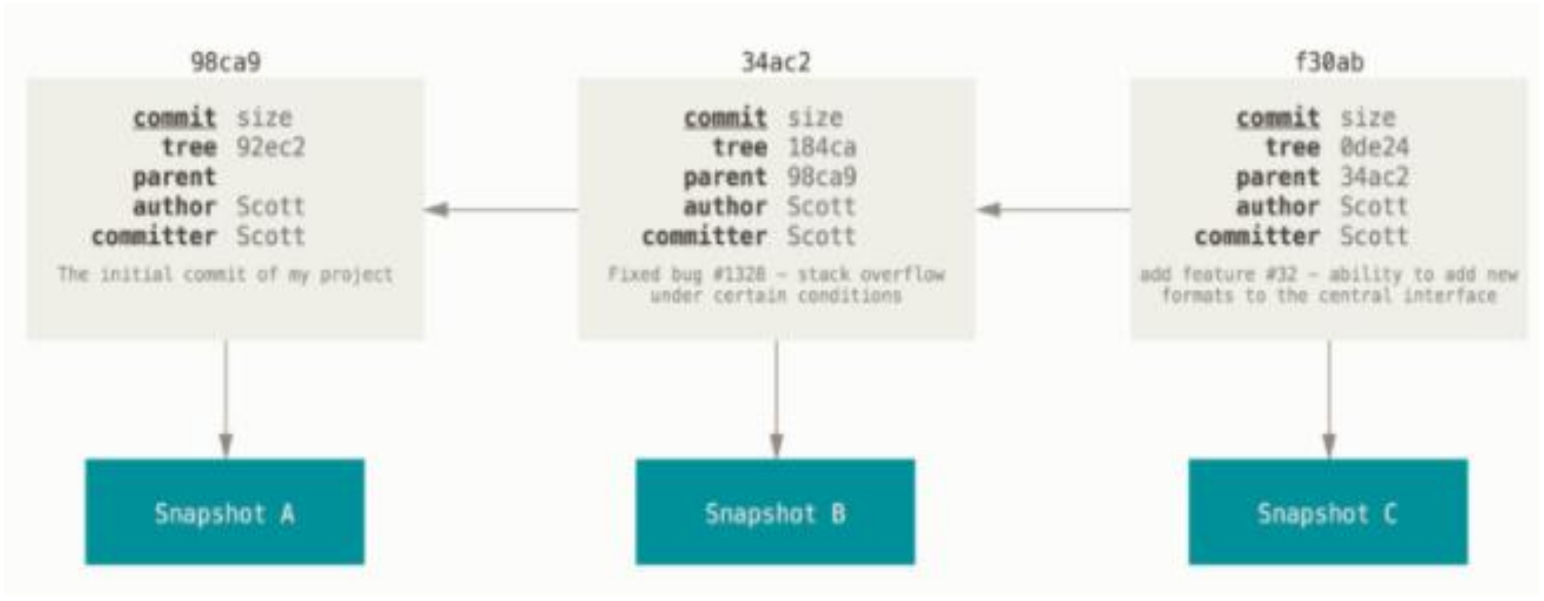
Git

How Git stores Data



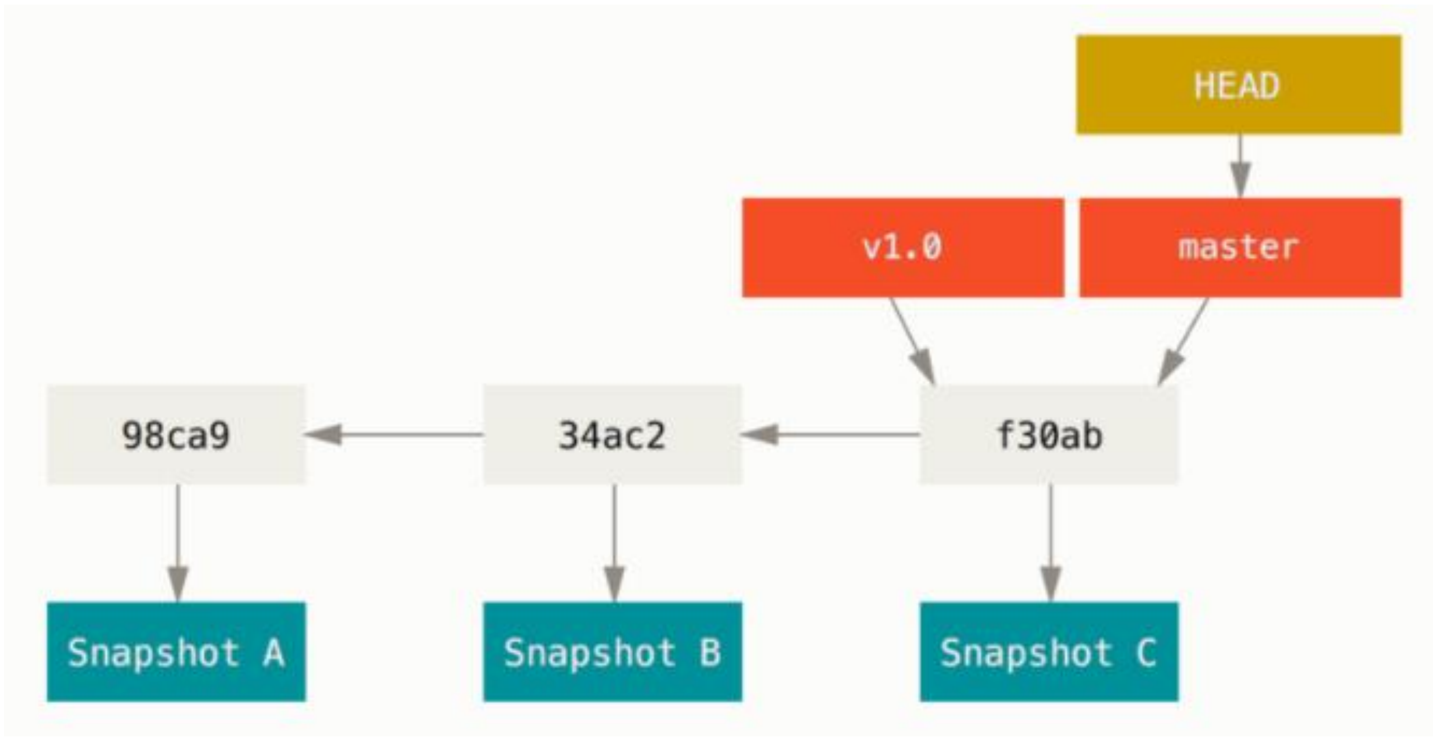
Git

After multiple commits



Git

A branch and its commit history

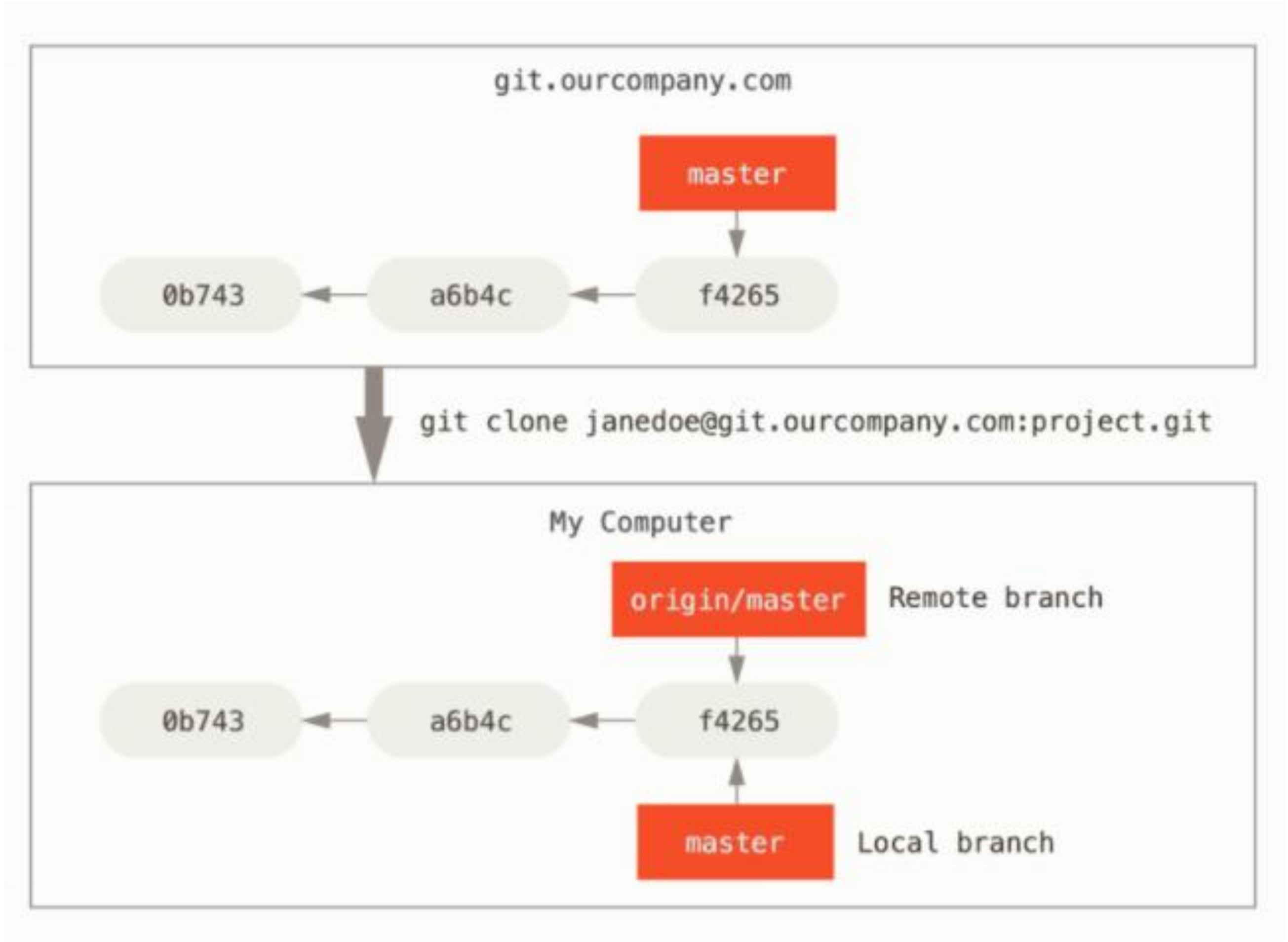


Git

6 Git Branching - Remote Branches

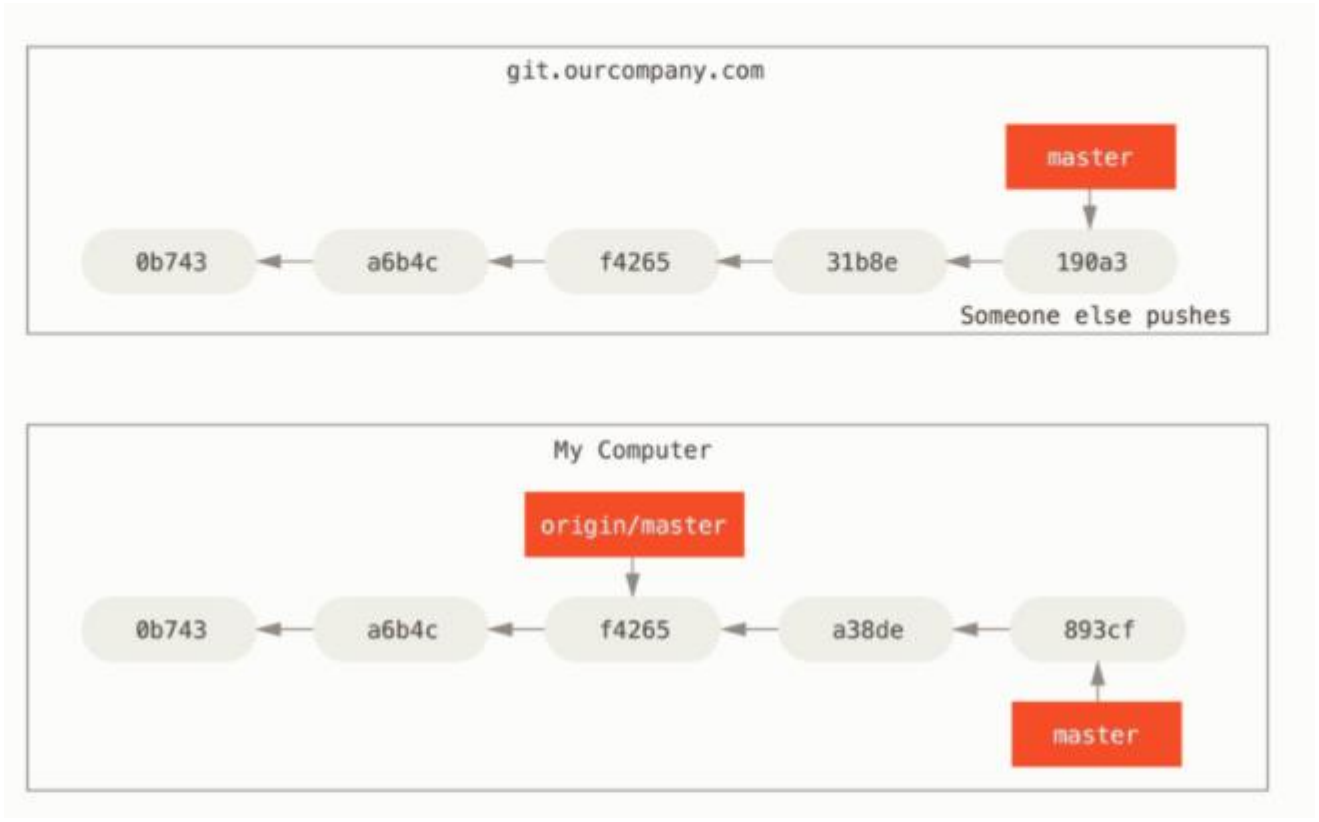
Git

Server and local repository after cloning



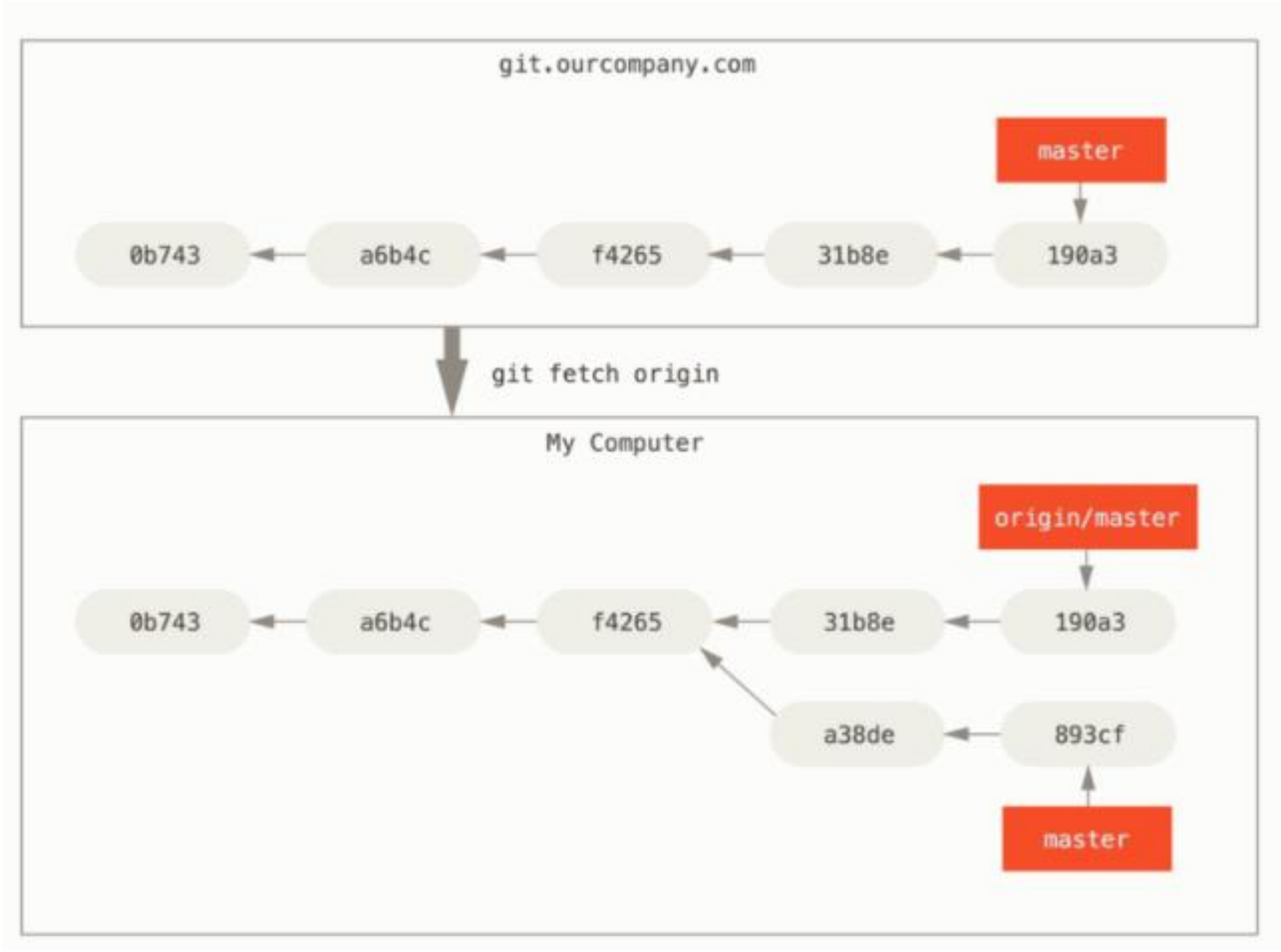
Git

Local and remote work could diverge



Git

Git fetch update remote references



Git

Quick revisit

- What's the git command that downloads your repository from GitHub to your computer?
- What's the opposite of git clone, instead of downloading your code from GitHub, uploads your changes and code back to GitHub?
- How do you check the state of your local git repository since your last commit?
- How do you stage files for a commit?
- What comes first, staging with git add . or committing with git commit?
- To set your email address at the global--or user--level, what command would you type?
- What is the name of the file used to tell Git to ignore certain files?
- Write the command that will display the diff of the README file, comparing the version in the staging area with the latest committed version.
- What command do you run to view the commit history of your repository?
- To create a branch, you run git branch . How can you create a branch and switch to it in one command?
- Write the command that would merge the awesome_feature branch with the master branch, assuming you are on the master branch.
- Which command will add a remote branch to the repository?
- Assuming your repo has a remote called "production", what command would send all local committed changes on the master branch to that remote?

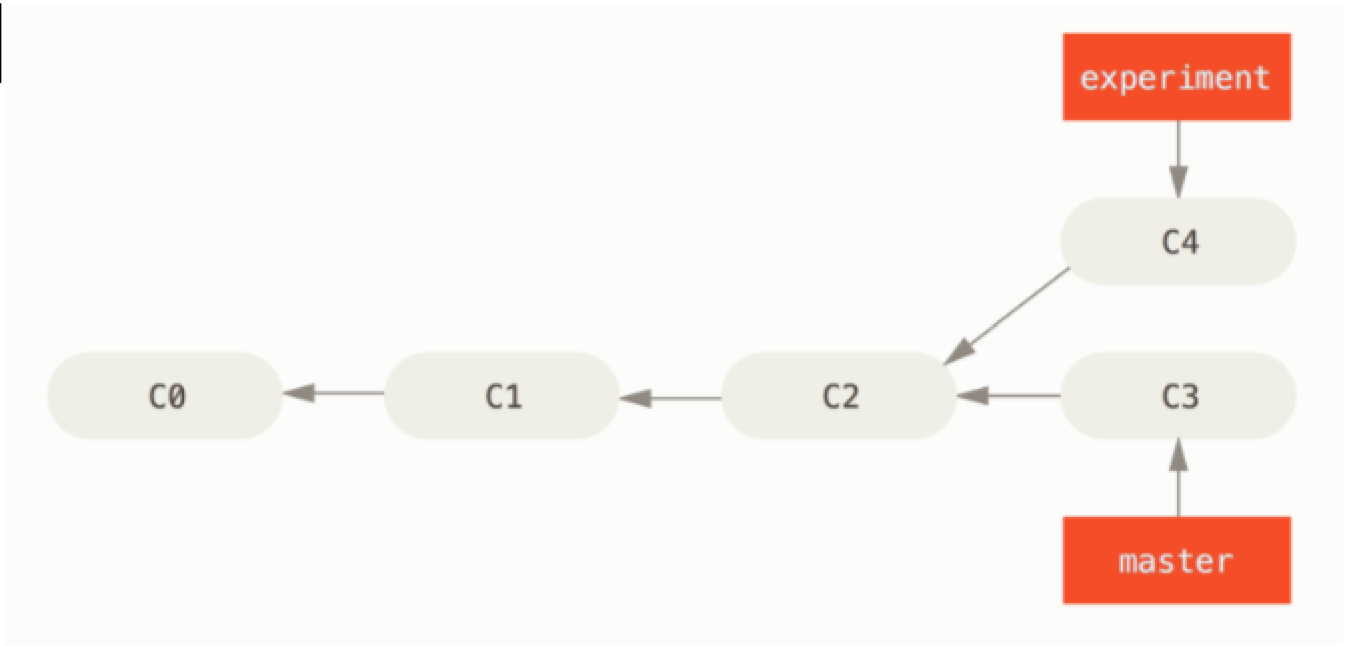
Git

7 Git Branching - Rebasing

Git

In Git, there are two main ways to integrate changes from one branch into another: the **merge** and the **rebase**

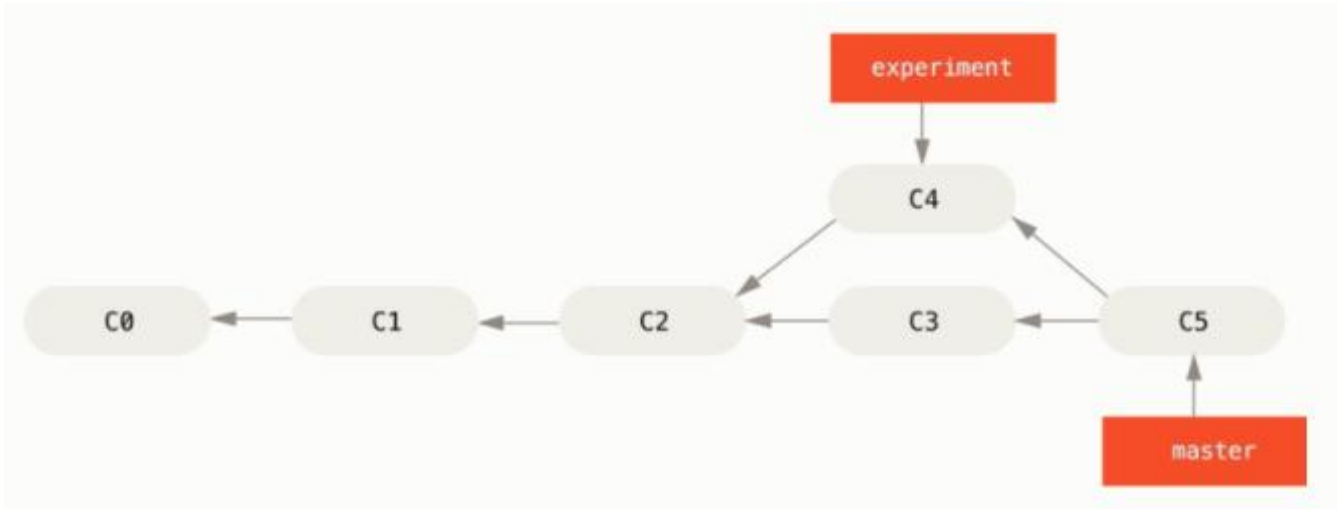
Basic Merging



The easiest way to integrate the branches, as we've already covered, is the **merge** command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit).

Git

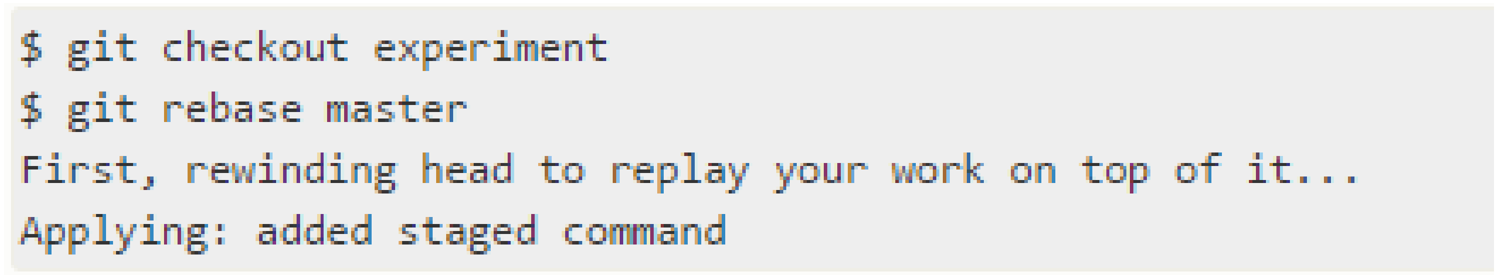
Basic Merging



However, there is another way: you can take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git, this is called rebasing. With the rebase command, you can take all the changes that were committed on one branch and replay them on another one

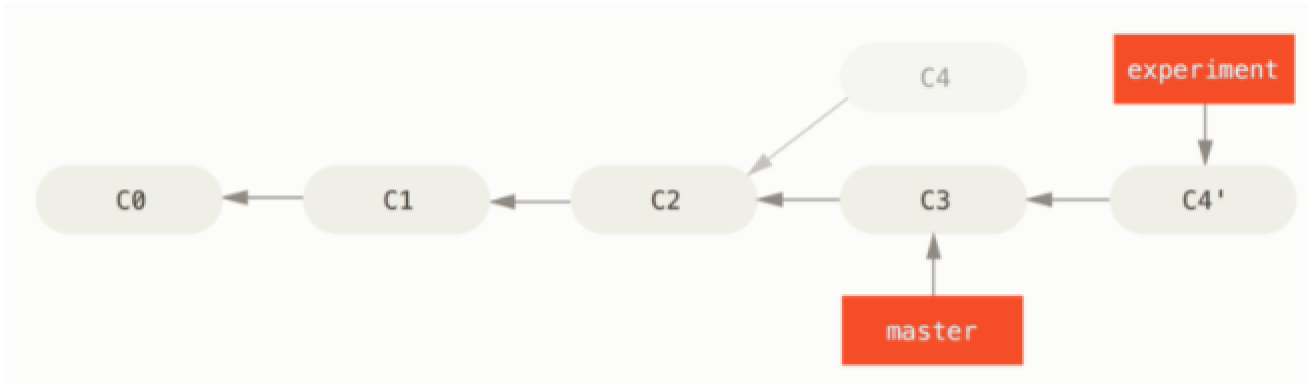
Git

Running command …



```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Git

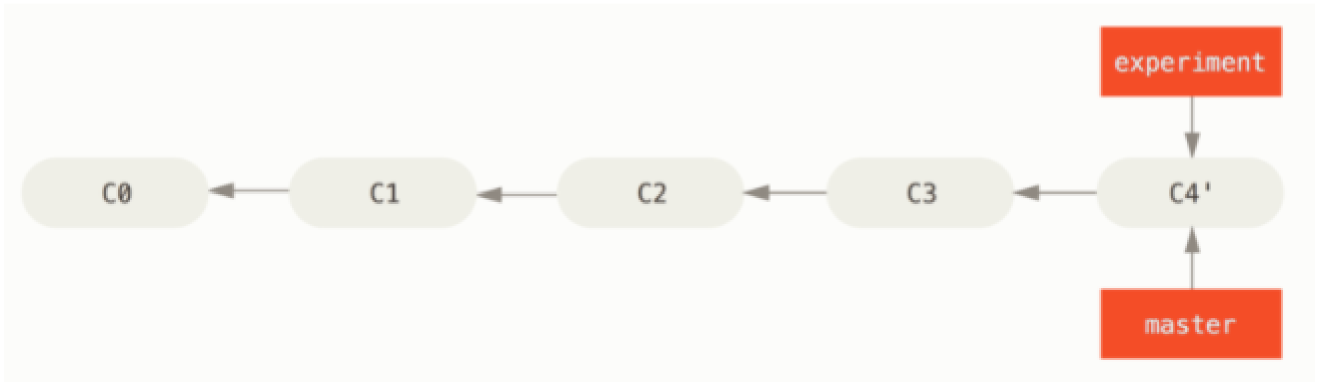


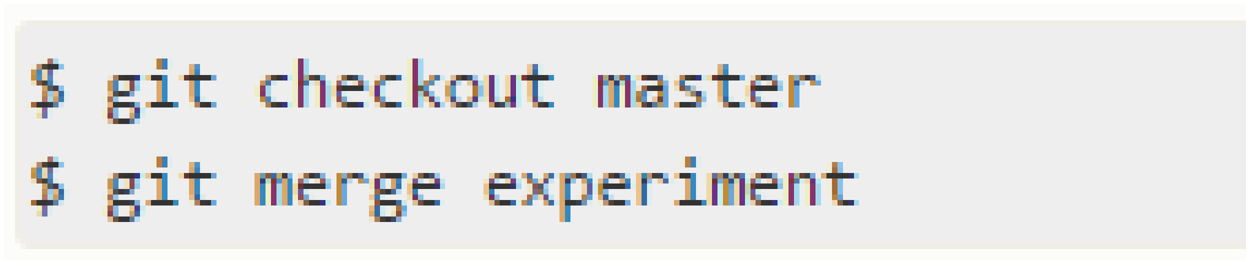
It works by going to the common ancestor of the two branches.

1. getting the diff introduced by each commit of the branch you’re on,
2. saving those diffs to temporary files,
3. resetting the current branch to the same commit as the branch you are rebasing
4. finally applying each change in turn

Git

At this point, you can go back to the master branch and do a fast-forward merge.

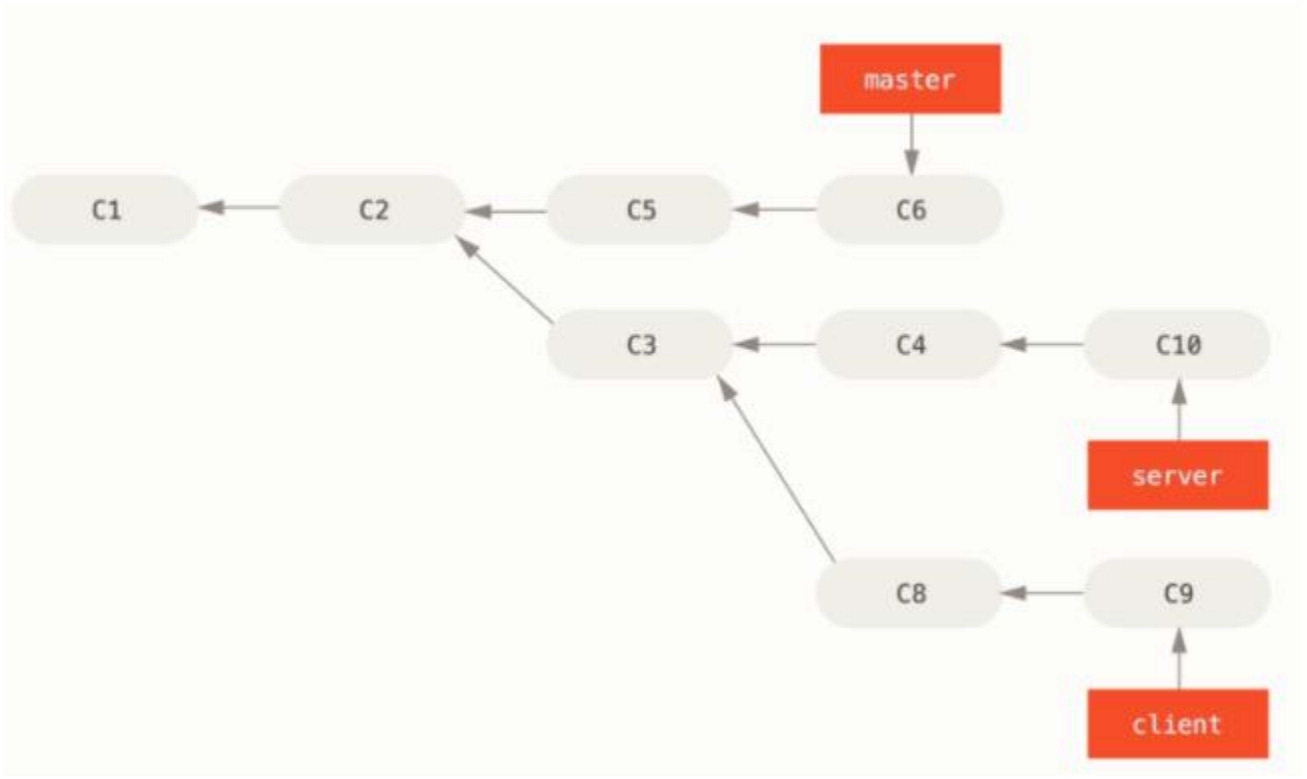




```
$ git checkout master
$ git merge experiment
```

Git

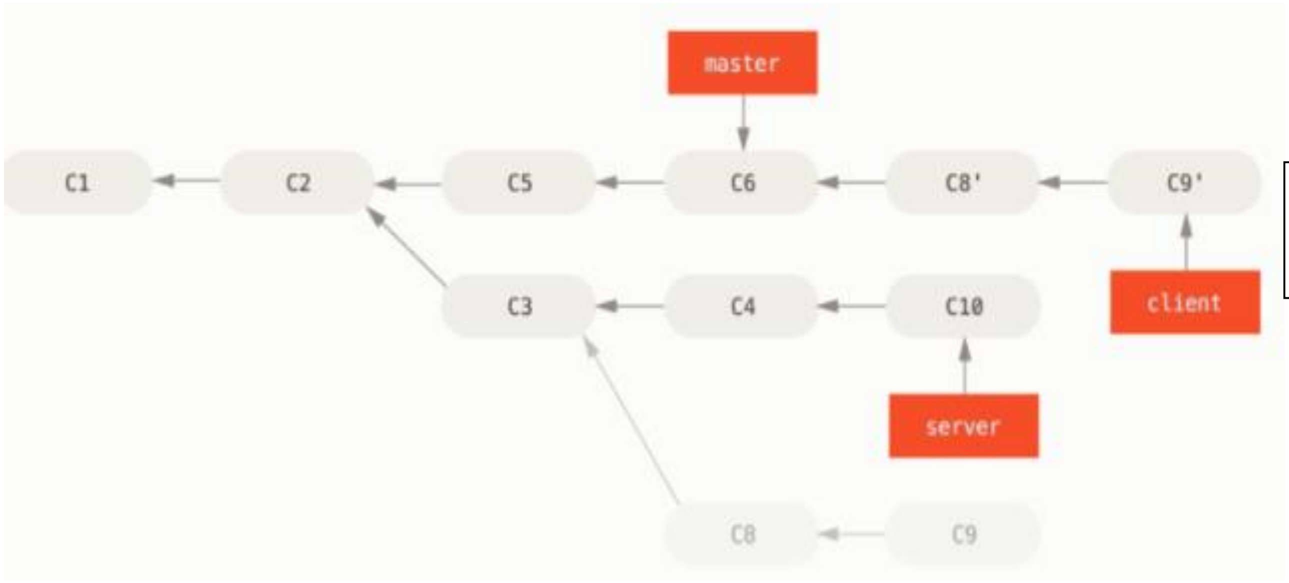
More Interesting Rebases



Git

Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on
the server-side changes until it's tested further.

```
$git rebase –onto master server client
```

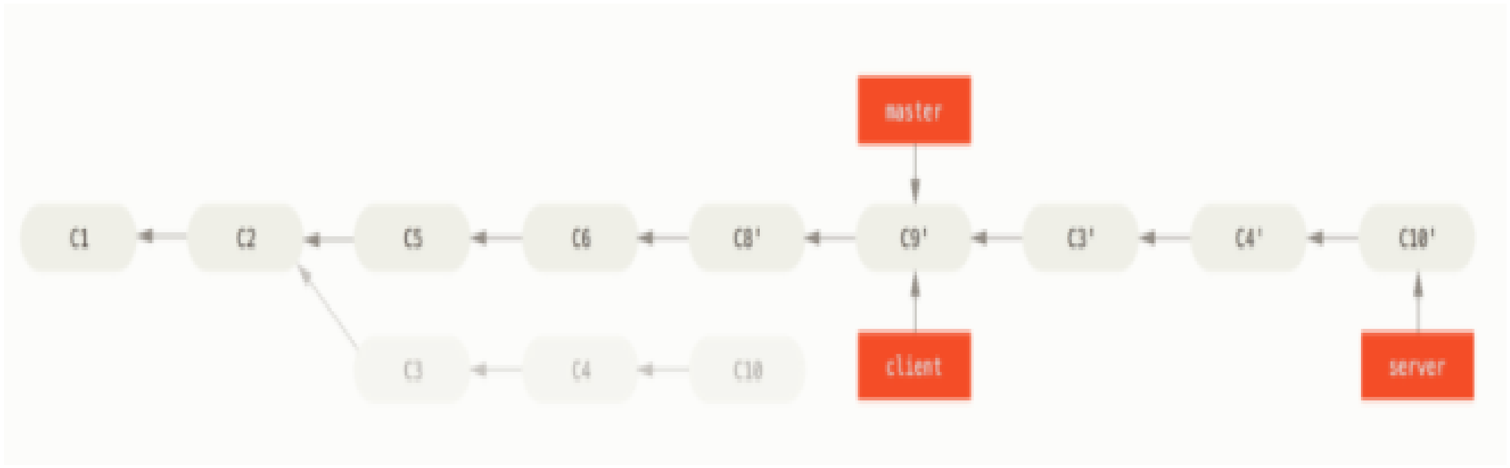


```
$git checkout master
$git merge client
```

Git

Let's say you decide to pull in your server branch as well.

```
$git rebase master server
```

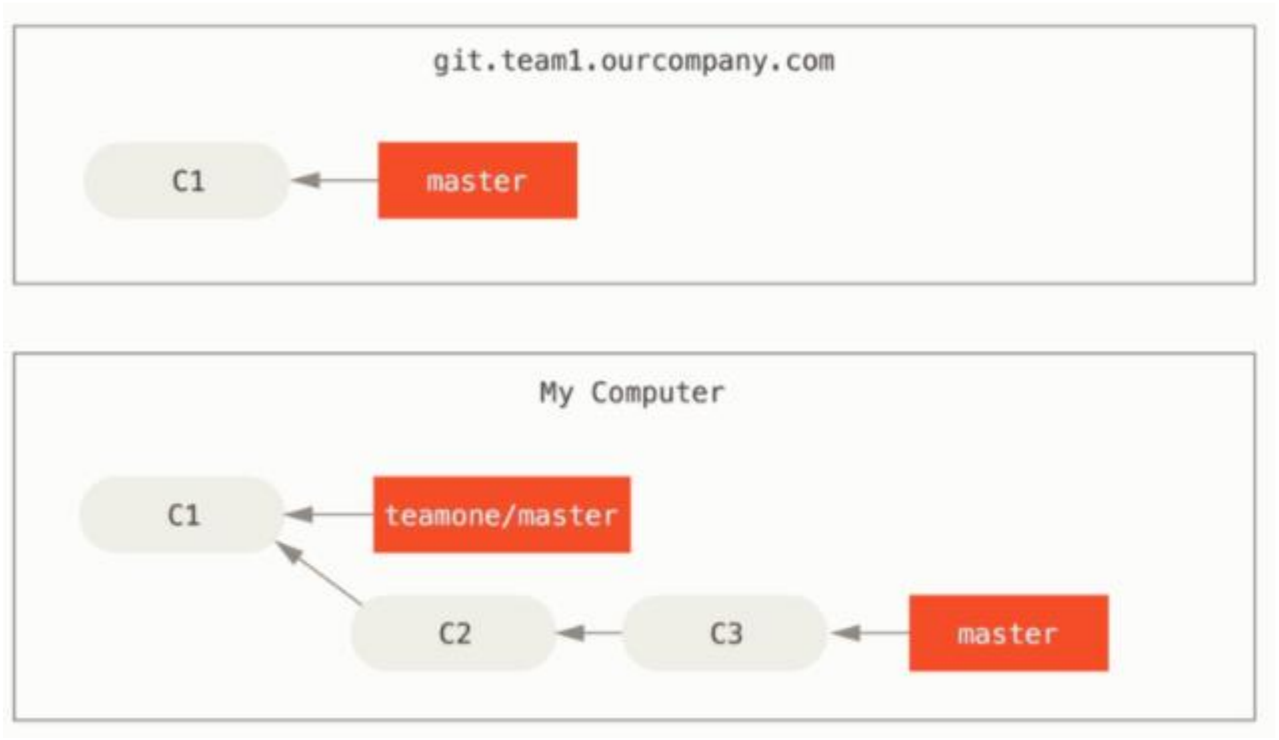


```
$git checkout master
$git merge server
```

Git

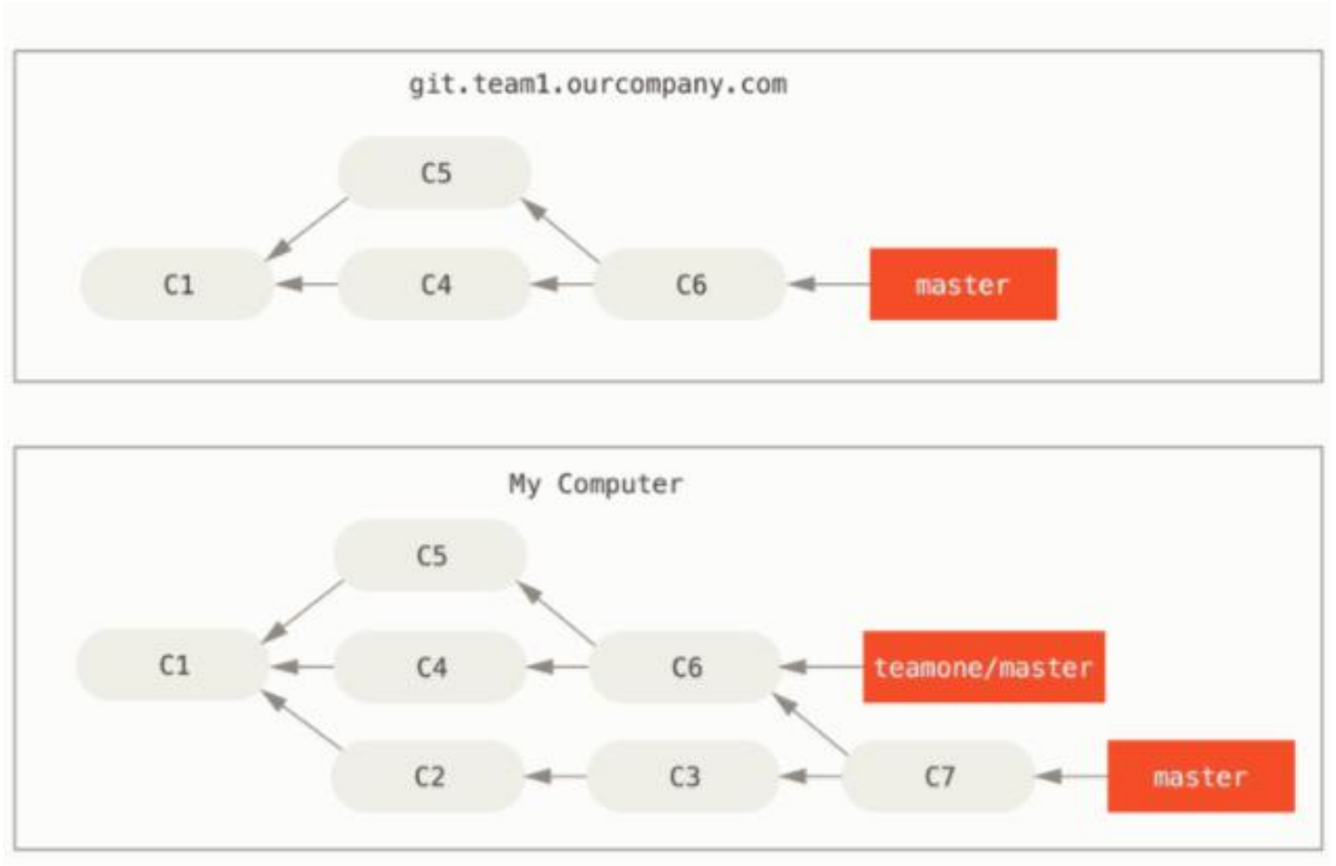
When we should use rebasing and merging?

Lets discuss when not to use rebasing!



Git

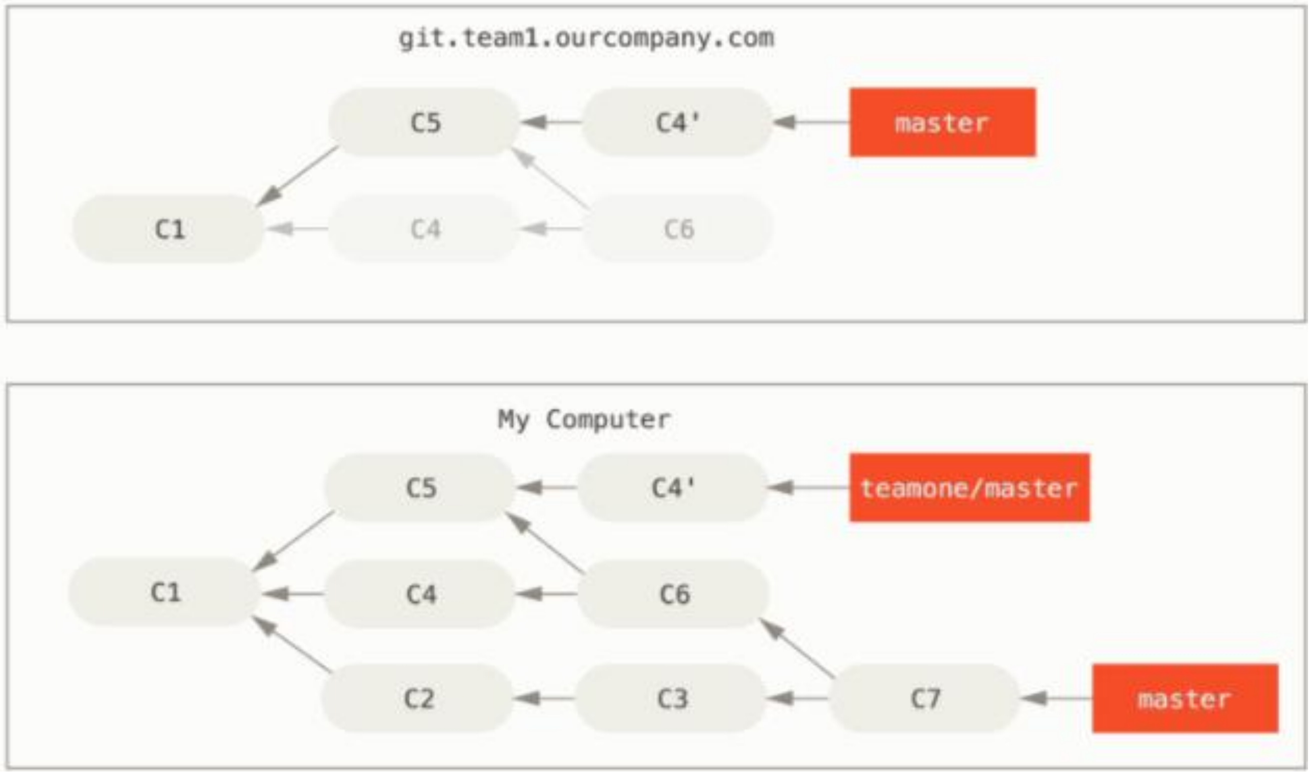
Lets discuss when not to use rebasing! …



Git

Lets discuss when not to use rebasing! …

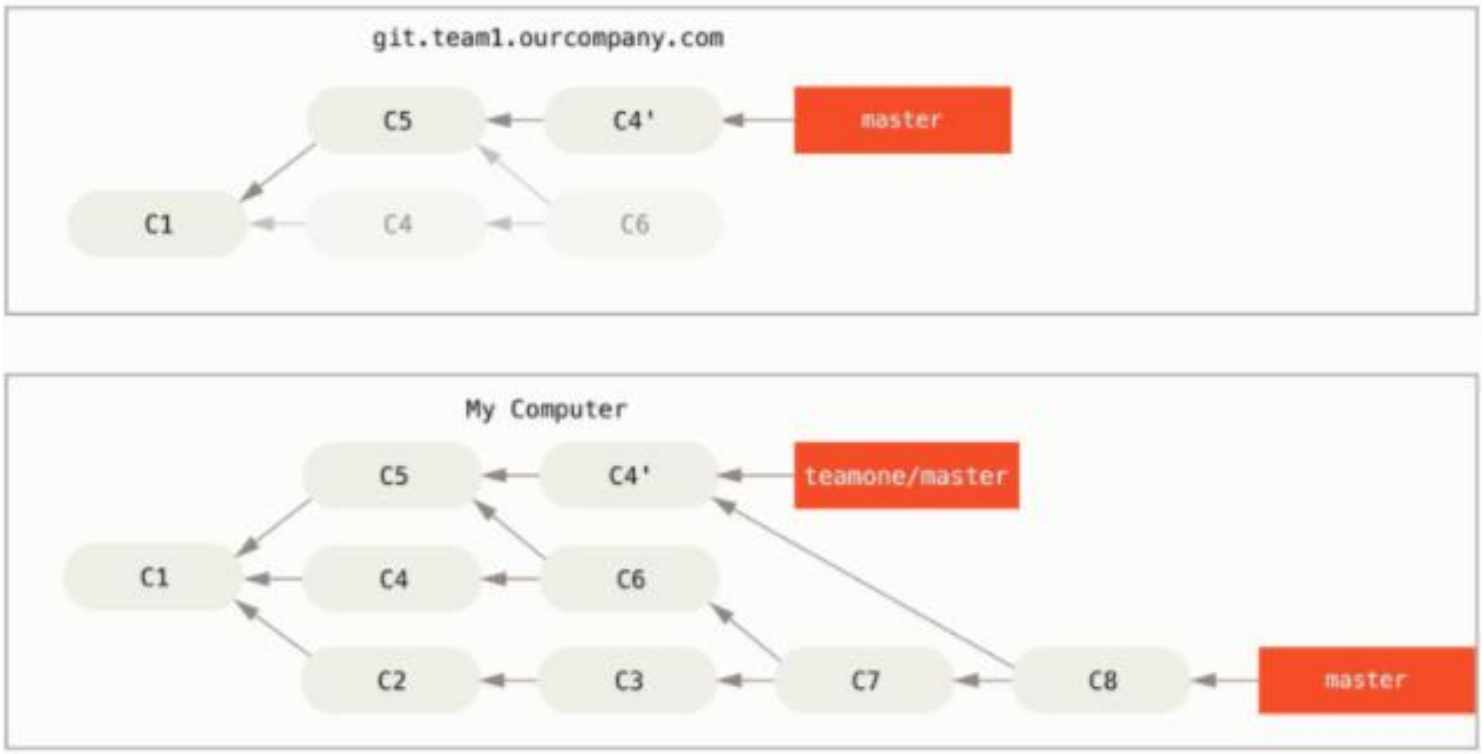
Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a **git push --force** to overwrite the history on the server. You then **fetch** from that server, bringing down the new commits.



Git

Lets discuss when not to use rebasing! …

If you do a **git pull**, you'll create a merge commit which includes both lines of history and your repository will look like this



Git

In Nutshell ..

Do not run rebasing after pushing the result to remote branch

Run $git pull –rebase instead of only git pull

OR

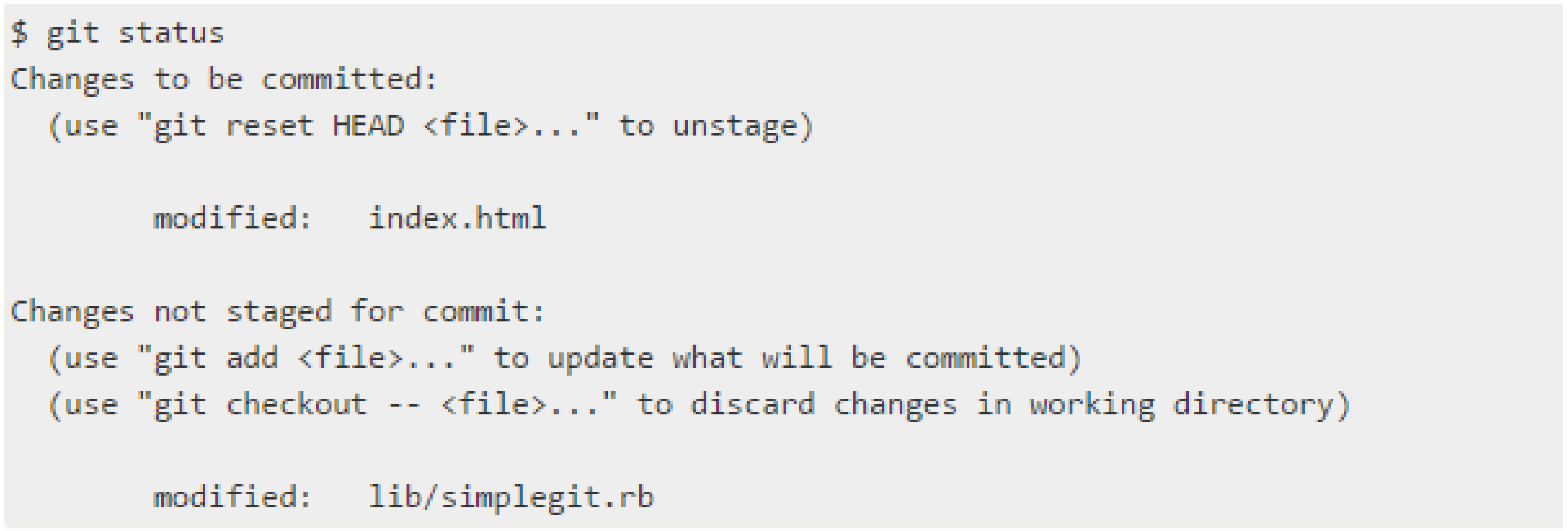
$**git fetch** followed by a $**git rebase teamone/master**

Git

8 Git Tools - Stashing

Git

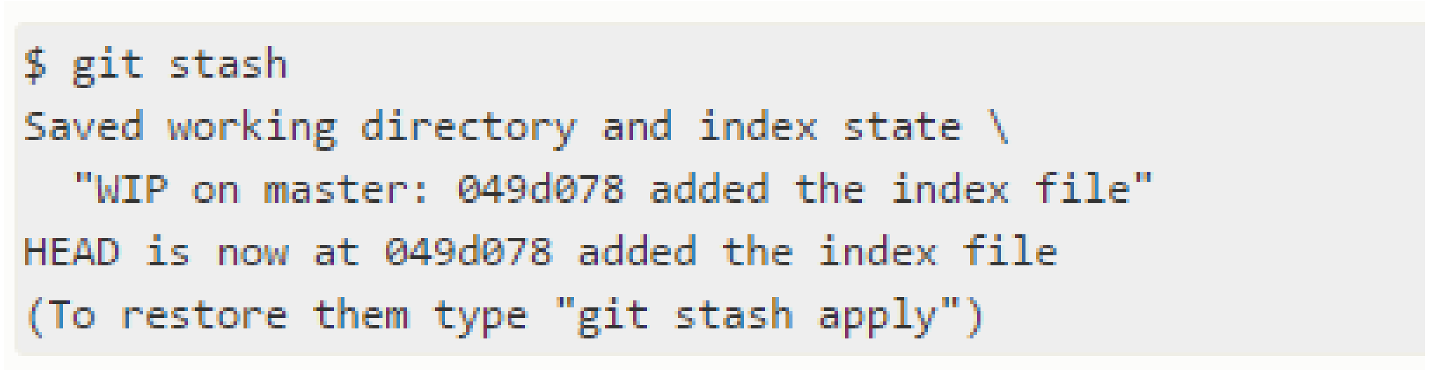
When you’ve been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else.
The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the **git stash** command.



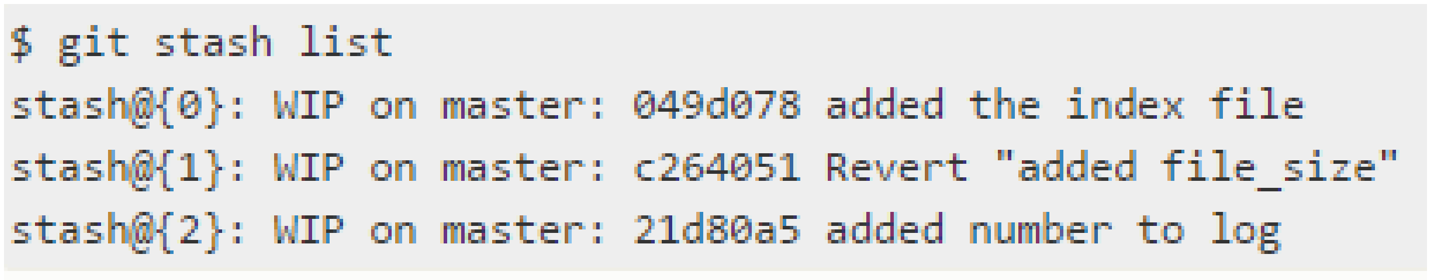
```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html

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

        modified:   lib/simplegit.rb
```

Git

you want to switch branches, but you don’t want to commit what you’ve been working on yet; so you’ll stash the changes.
To push a new stash onto your stack, run **git stash or git stash save**:



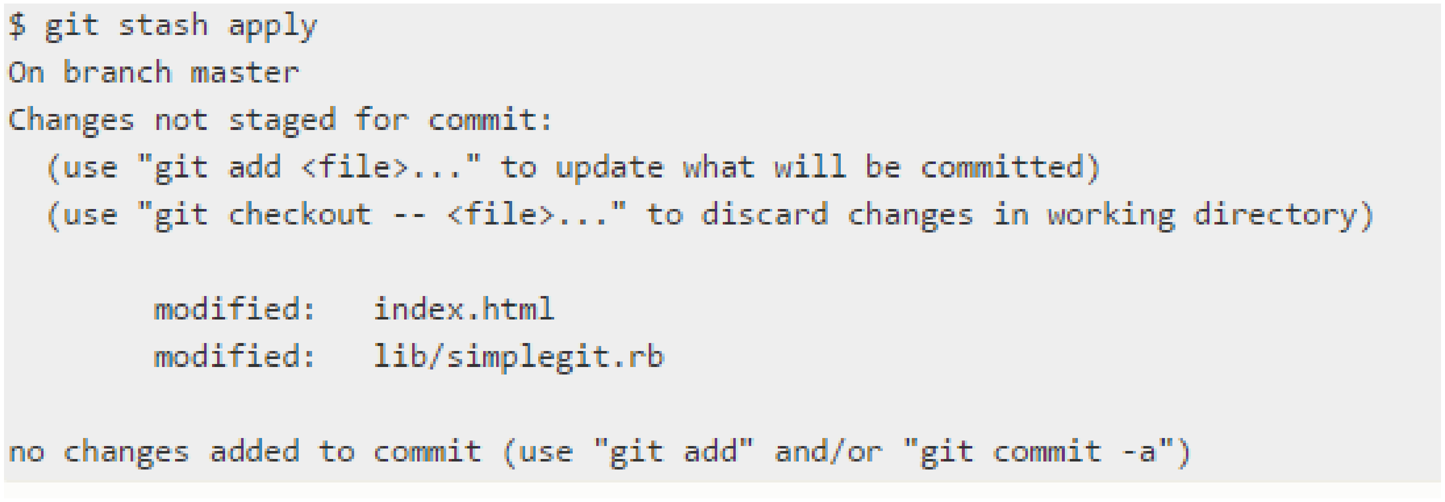
```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```



```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

Git

How to apply stashes?



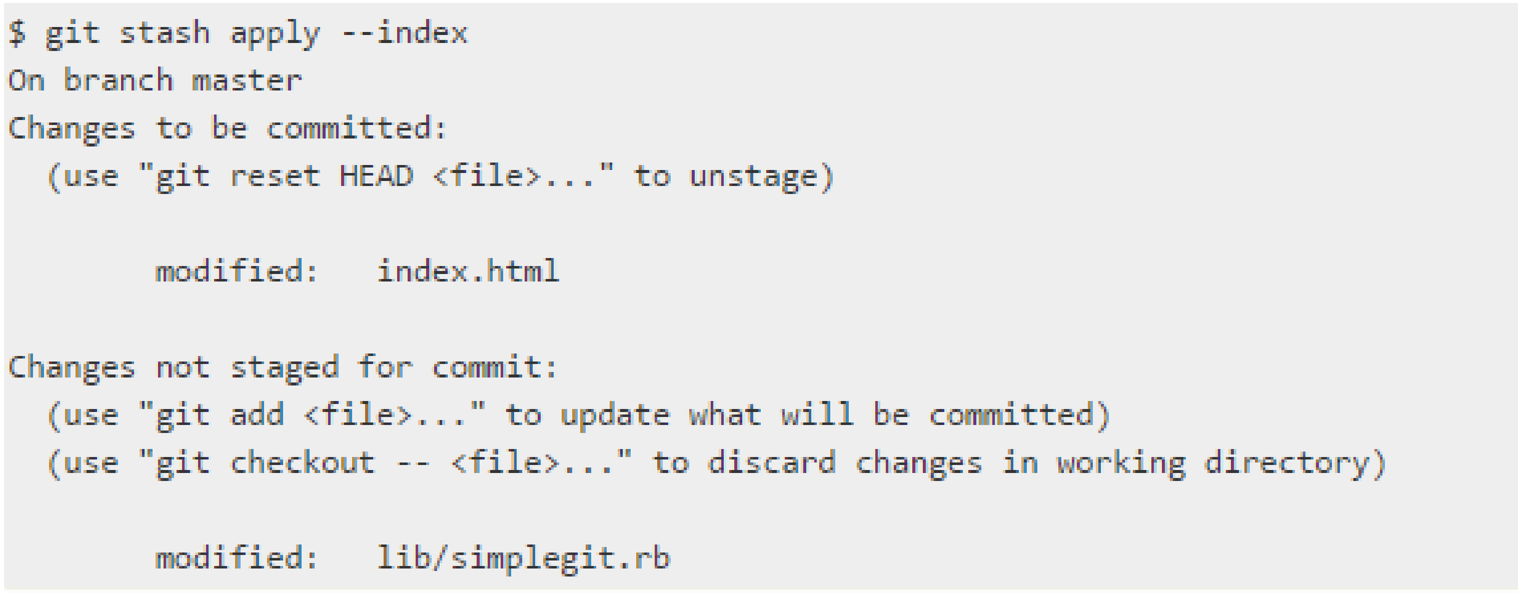
```
$ git stash apply
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   index.html
        modified:   lib/simplegit.rb

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

```
$git stash apply stash@{2}
```

Git

By default the changes to your files were reapplied, but the file you staged before wasn’t restaged.

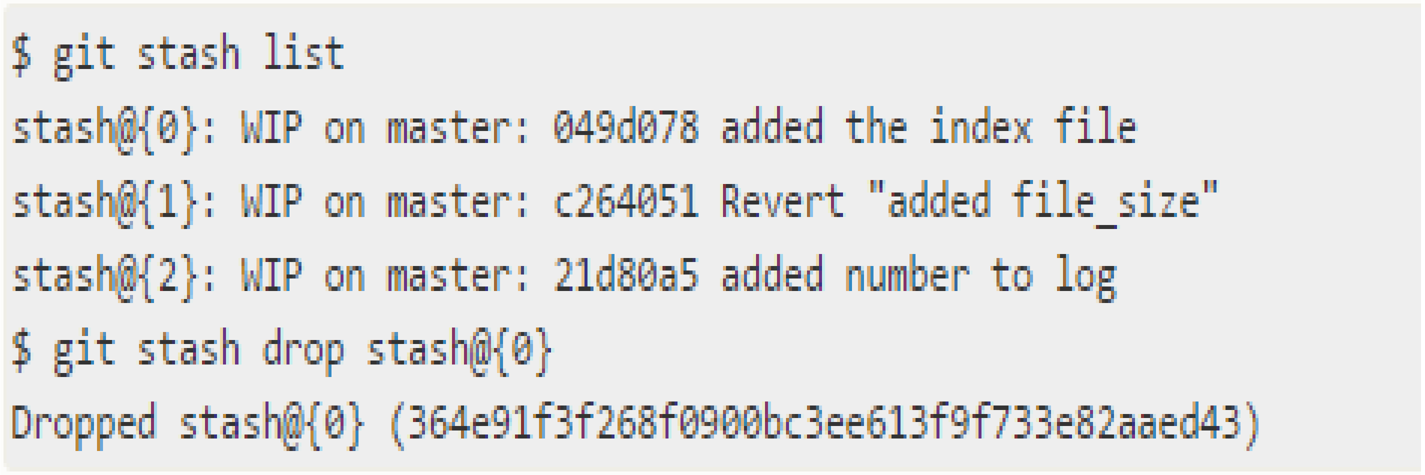
To do that, you must run the git stash apply command with a --index option to tell the command to try to reapply the staged changes.



```
$ git stash apply --index
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html

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

        modified:   lib/simplegit.rb
```

Git

The apply option only tries to apply the stashed work.
You continue to have it on your stack.
To remove it, you can run **git stash drop** with the name of the stash to remove

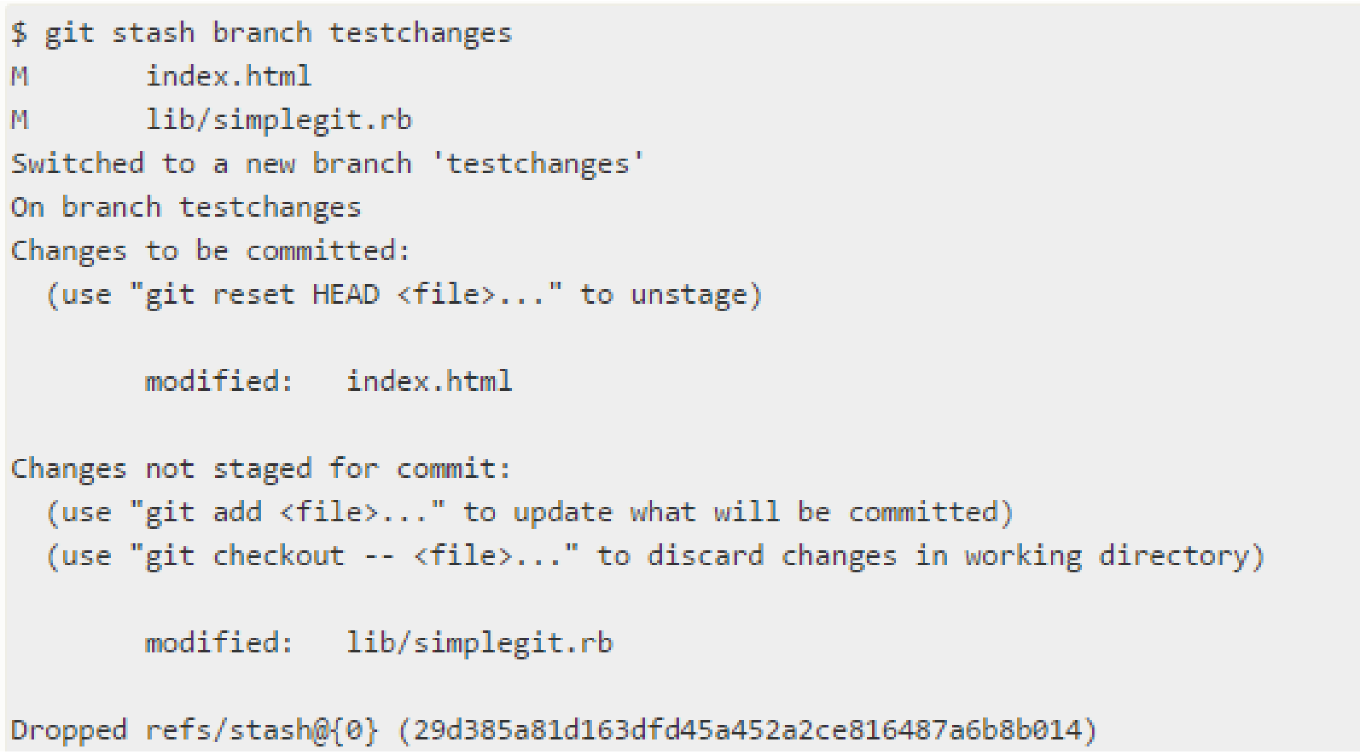


```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Git

Creating a Branch from a Stash!

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work

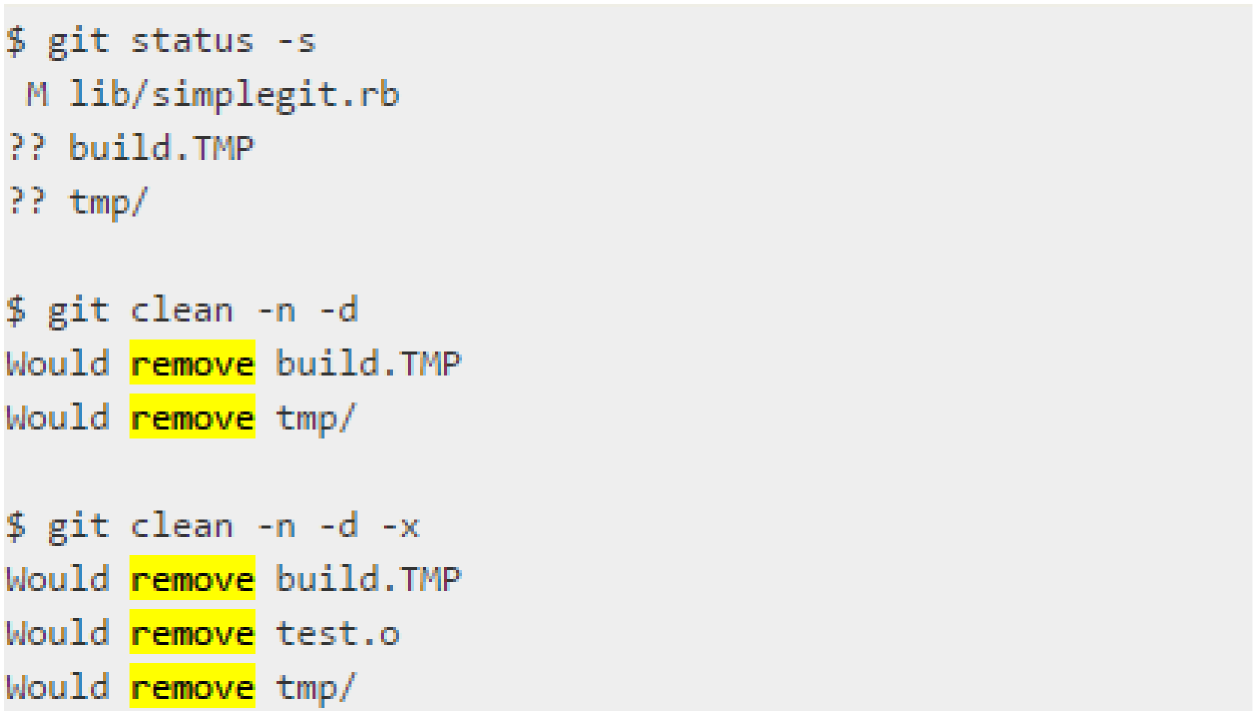


```
$ git stash branch testchanges
M       index.html
M       lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html

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

        modified:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

Git

Cleaning your Working Directory

By default, the **git clean** command will only remove untracked files that are not ignored, any file that matches a pattern in your **.gitignore** or other ignore files will not be removed



```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

Git

Git example…..

Q&A