Version Control using GIT

Loide Jesus (INdT)

April, 2014

Summary

Conditions

Introduction

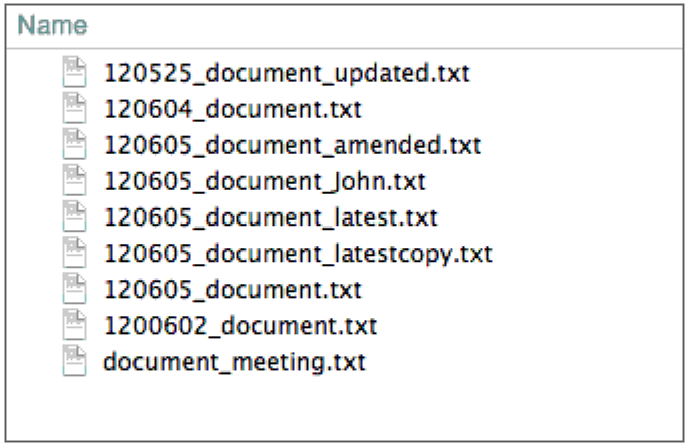
What is and Why should you care about it?

- **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
  - Keep every version of your development project
  - Allows to revert files (or the entire project) back to a previous state
  - Compare changes over time
  - See when and who introduced a issue
  - Recovery easily lost or damaged files
- What do you do when you want to recover past versions of a file?

Introduction

Local Version Control

- The most common way to do it is to save multiple files with different dates and file names.
- This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

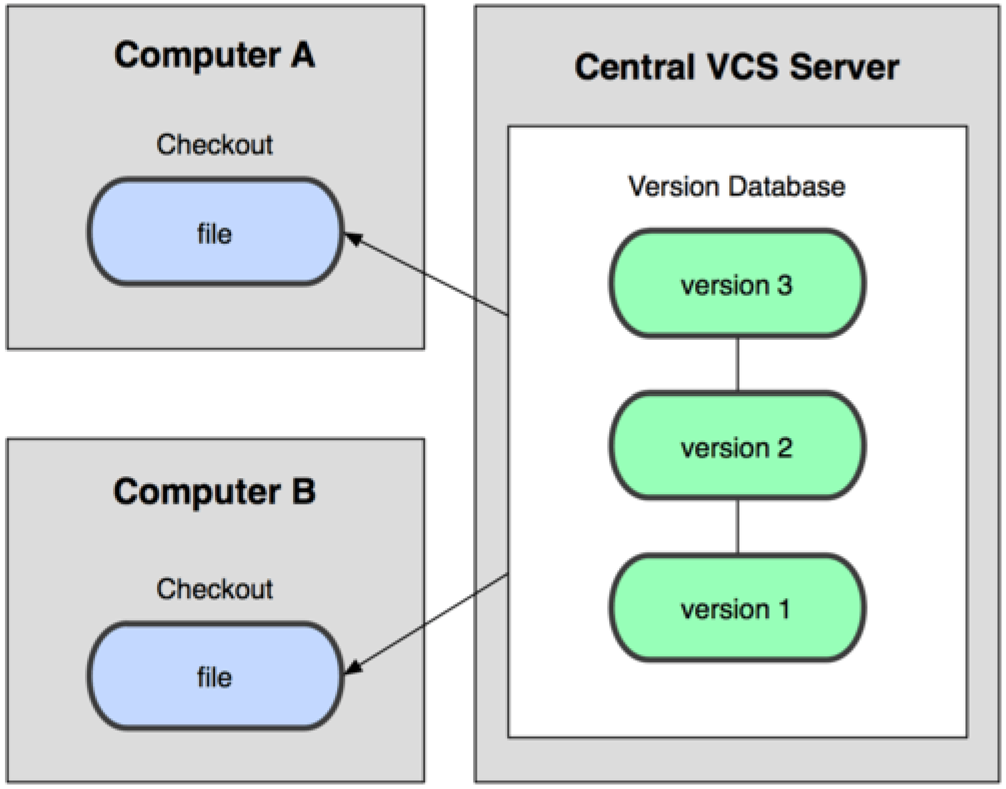


```
Name
120525_document_updated.txt
120604_document.txt
120605_document_amended.txt
120605_document_John.txt
120605_document_latest.txt
120605_document_latestcopy.txt
120605_document.txt
1200602_document.txt
document_meeting.txt
```

Introduction

Centralized Version Control Systems

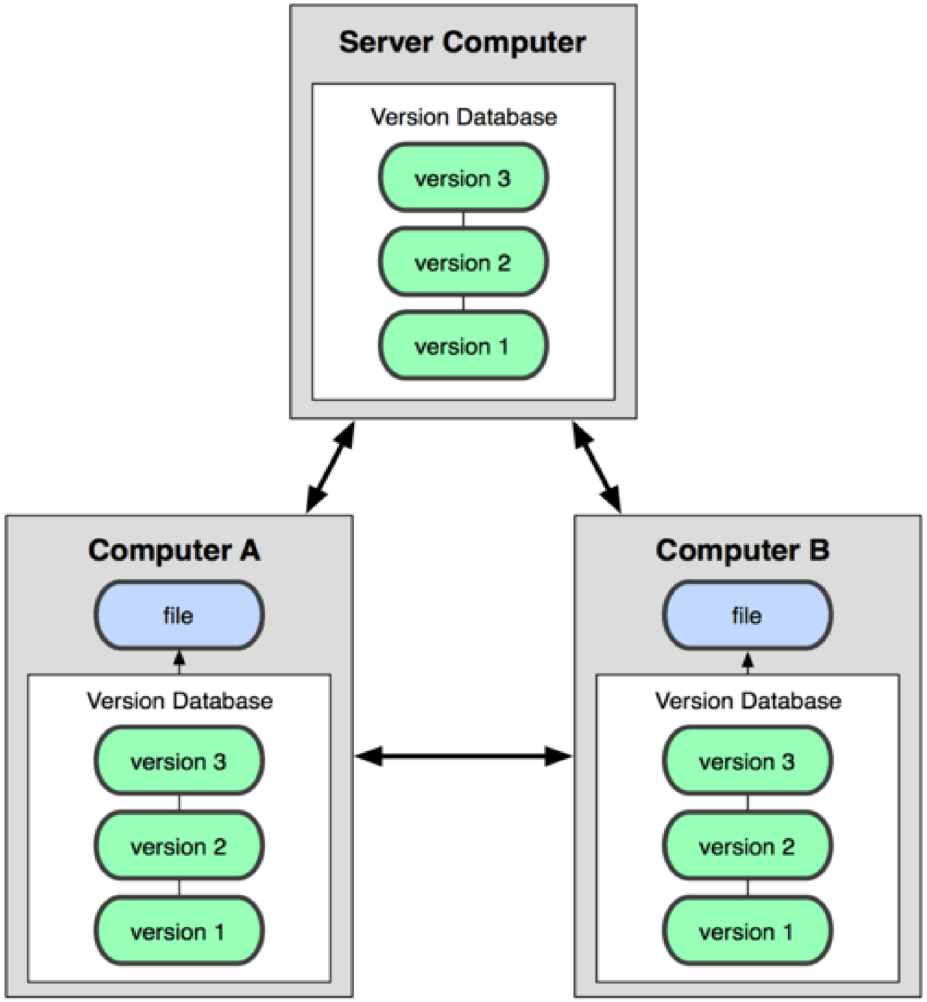
- There are a single server that contains all the versioned files, and a number of clients that check out files from that central place.
- For many years, this has been the standard for version control.
  - Such as CVS, Subversion or Perforce



Introduction

Distributed Version Control Systems

- Clients don't just check out the latest snapshot of the files: they fully mirror the repository.
  - Such as Git, Mercurial, Bazaar or Darcs
- If any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it
- Allows you to collaborate with different groups of people in different ways simultaneously within the same project.



Introduction

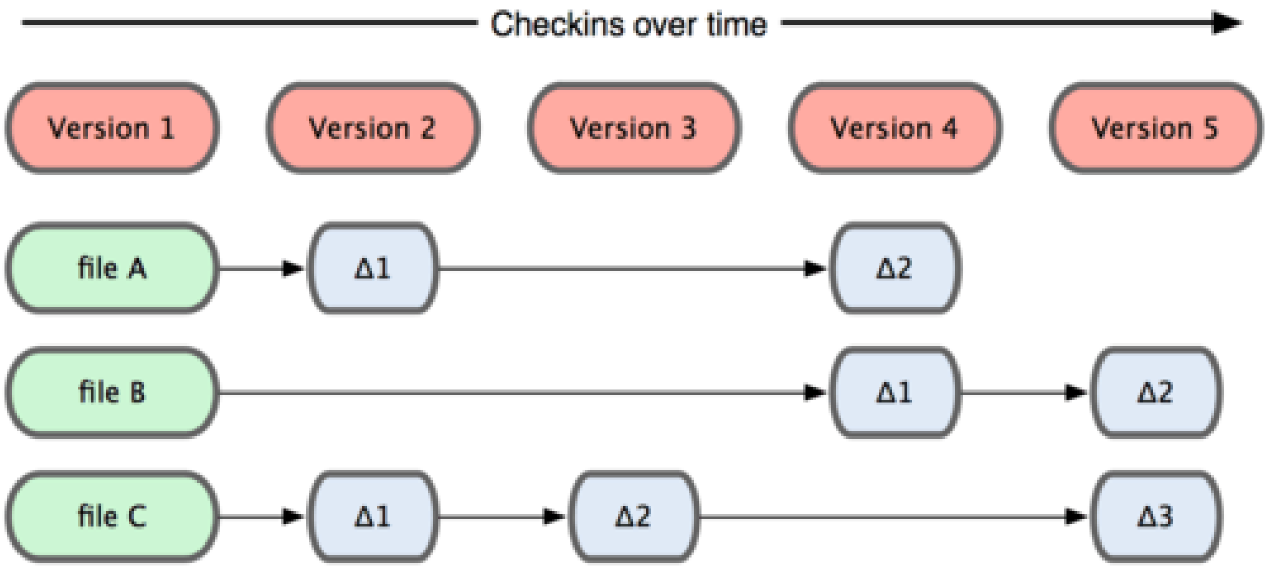
A short history of Git

- A reaction to licensing changes to BitKeeper.
- Git was initially designed and developed by Linus Torvalds for Linux kernel development in 2005.
- Linux development community developed their own tool based on some of the lessons they learned while using BitKeeper:
  - Speed
  - Simple design
  - Strong support for non-linear development (thousand of parallel branches)
  - Fully distributed
  - Ability to handle large projects like the Linux kernel efficiently (speed and data size)

Introduction

Snapshots, not differences

- The major difference between Git and any other VCS is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes.
  - Keep a set of files and the changes made to each file over time.

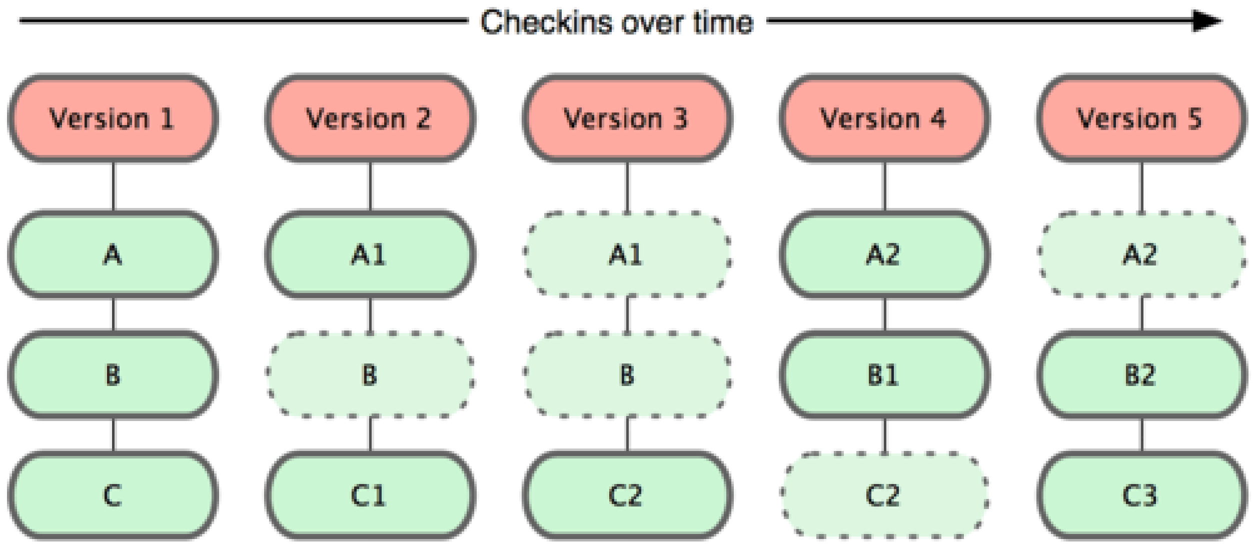


Other systems tend to store data as changes to a base version of each file.

Introduction

Snapshots, not differences

- Git thinks of its data more like a set of snapshots of a mini 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.

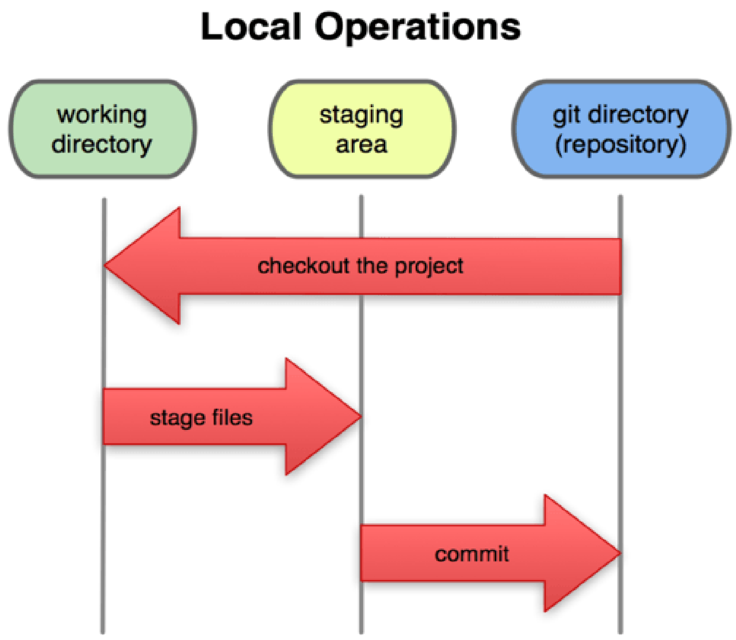


Git stores data as snapshots of the project over time

Introduction

Main thing to remember about Git

- Three main states that your files can reside in:
  - ***commited***: data is safety stored in your local database
  - ***modified***: the file was changed but have not commited it to your database yet
  - ***staged***: the modified file in its current version was marked to go into your next commit snapshot



Git Basics

Getting a Git repository

- There are two ways to create a local repository on your local machine:
  - Create a brand new repository from scratch

```
$ git init
```

  - Cloning an existing remote repository onto your local machine.

```
$ git clone <url>
```

Git Basics

Getting a Git repository

- There are two ways to create a local repository on your local machine:
  - Create a brand new repository from scratch

```
$ git init
```

  - Cloning an existing remote repository onto your local machine.

```
$ git clone gitosis@spearhead.indt.org:/git-course.git

$ git clone gitosis@10.60.100.158:/git-course.git
```

Git Basics

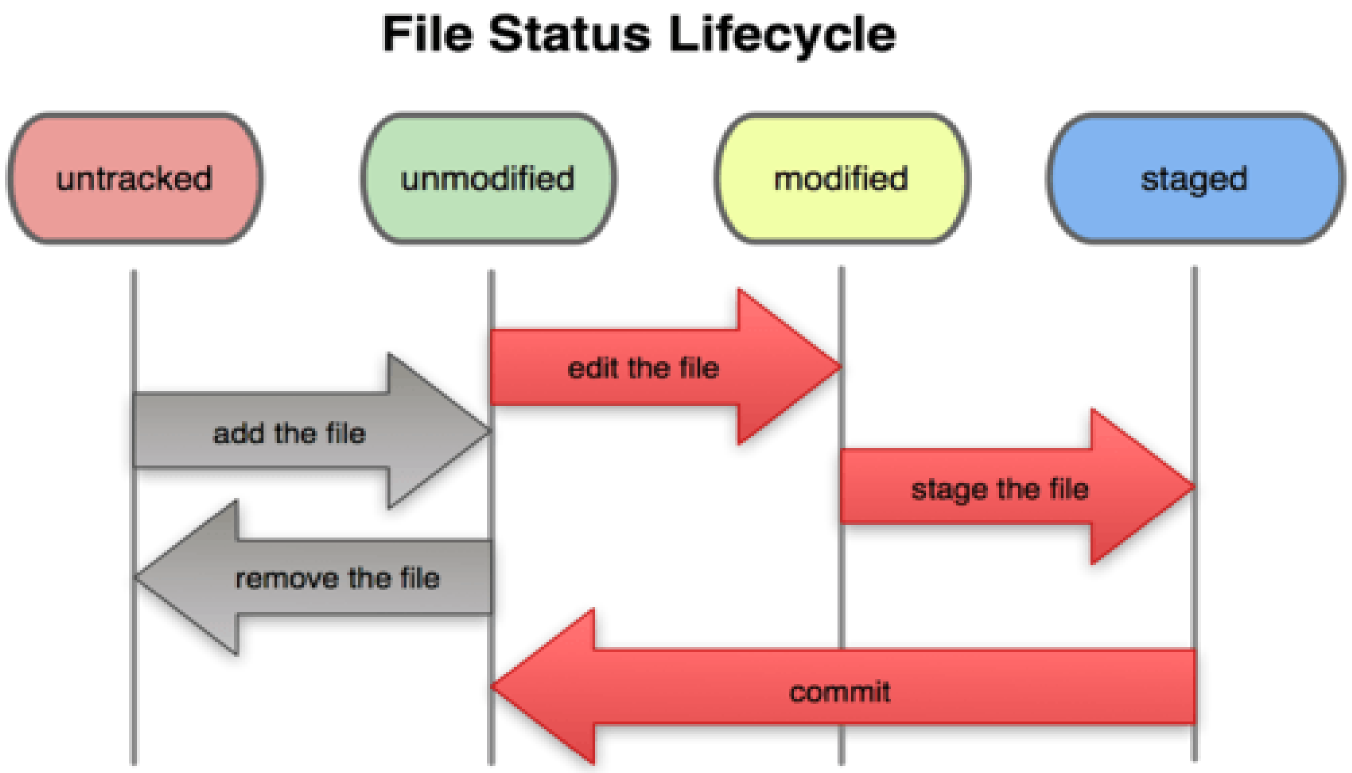
Introduce yourself

- Define everything from user info to preferences to the behavior of a repository

```
$ git config --global user.name <name>

$ git config --global user.email <email>
```

Git Basics

Recording changes to the repository



Git Basics

Checking the status of your file

- Determine which files are in which state

```
$ git status
```

Git Basics

Checking the status of your file

- Determine which files are in which state

```
      $ git status

       On branch master

       Your branch is up-to-date with 'origin/master'.

       nothing to commit (working directory clean)
```

Git Basics

Checking the status of your file

- Change a file that was already tracked then run your status command again.

```
$ git status
 On branch master
 Your branch is up-to-date with 'origin/master'.

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

        modified: README
  no changes added to commit (use “git add” and/or git commit -a)
```

Git Basics

Staging modified files

```
$ git add README
$ git status
 On branch master
 Your branch is up-to-date with 'origin/master'.

 Changes to be commited:
    (use “git reset HEAD <file>..” to unstage)

        modified: README
```

Git Basics

Commiting your changes

```
$ git commit -m “Updating README”
[master 463dc4f] Updating README
 1 file changed, 2 insertions(+)

$ git log
```

Git Basics

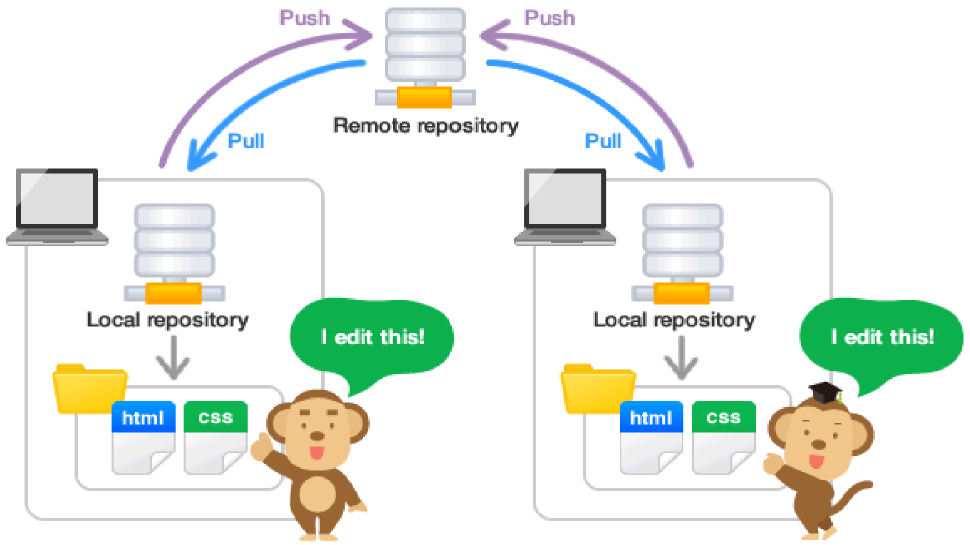
Pulling and pushing

- Pulls down all the data from that remote project that you don't have yet:

```
$ git pull [remote-name]

$ git pull --rebase [remote]
```

- When you want to share your project, you have to push it upstream:

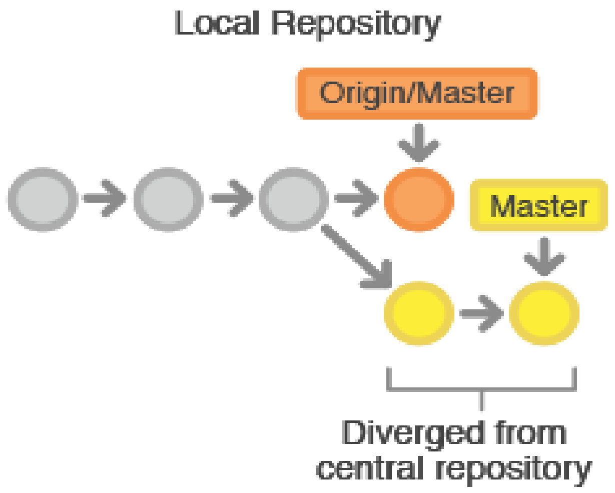
```
$ git push [remote-name] [branch-name]

$ git push origin master
```



Git Basics

Managing conflicts

- The remote repository represents the official project, so its commit history should be treated as immutable.
- If a developer's local commits diverge from the central repository, Git will refuse to push their changes because this would overwrite official commits.



Git Basics

Managing conflits

```
$ git pull origin master
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From http://github.com/git/loide/teste.git
*branch master -> FETCH_HEAD
Auto-merging README.txt
CONFLICT (content): Merge conflict in README.txt
Automatic merge failed; fix conflicts and then commit the result
```

Git Basics

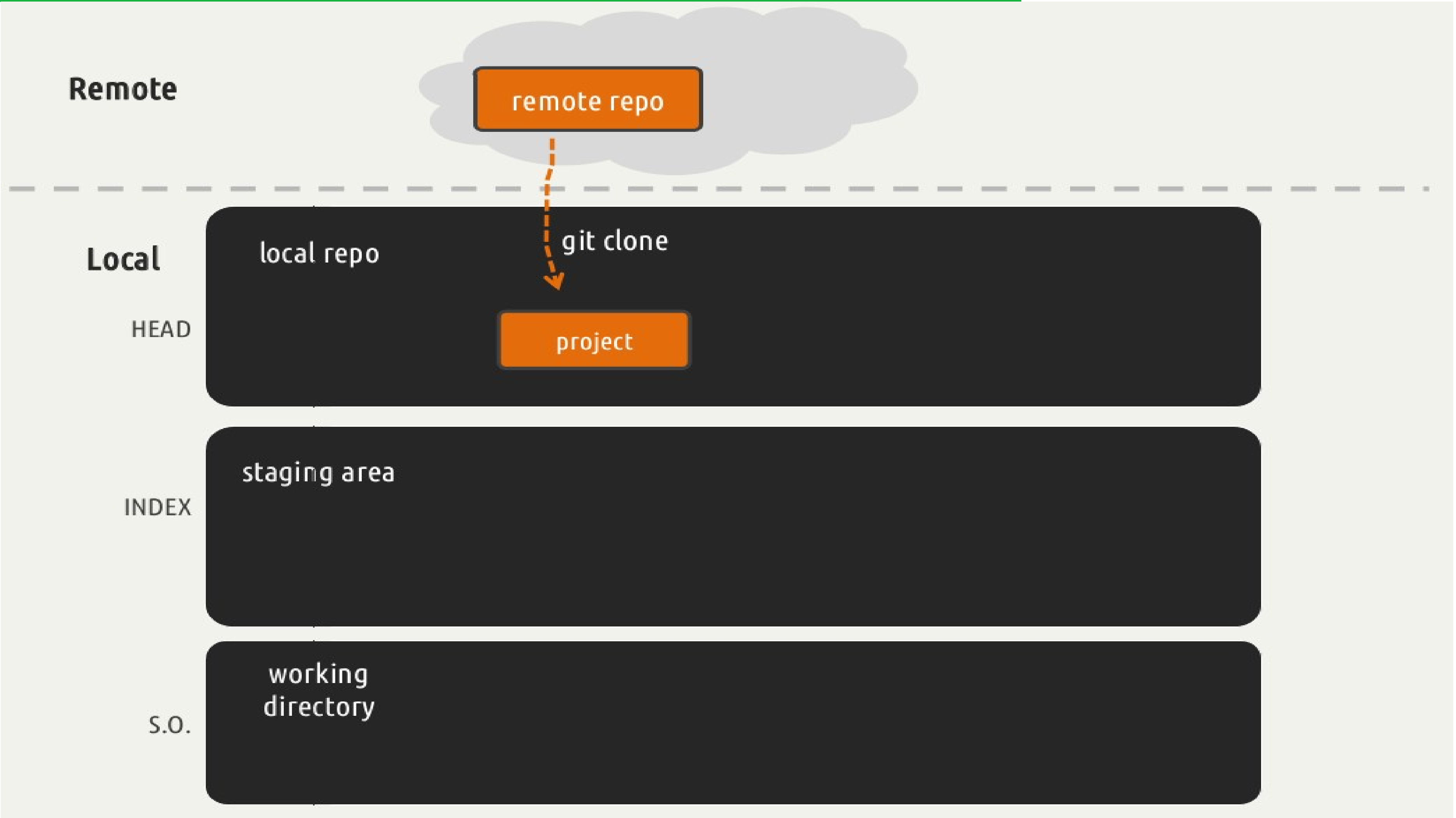
Managing conflicts

- When you open the modified file, you can see markers that have been added by Git to indicate conflicts in that section of the file.

```
Upstream puppet modules (unmodified)

Puppet modules for central platform

To add a new module, use the add_module.sh script.

<<<<<<<<<<< HEAD

A puppet module for managing elasticsearch nodes

=========

Scripts to initialize webservices

>>>>>>>>>>>> 17c860612953c0f9d88f313c8dfbf7d858e02e91
```

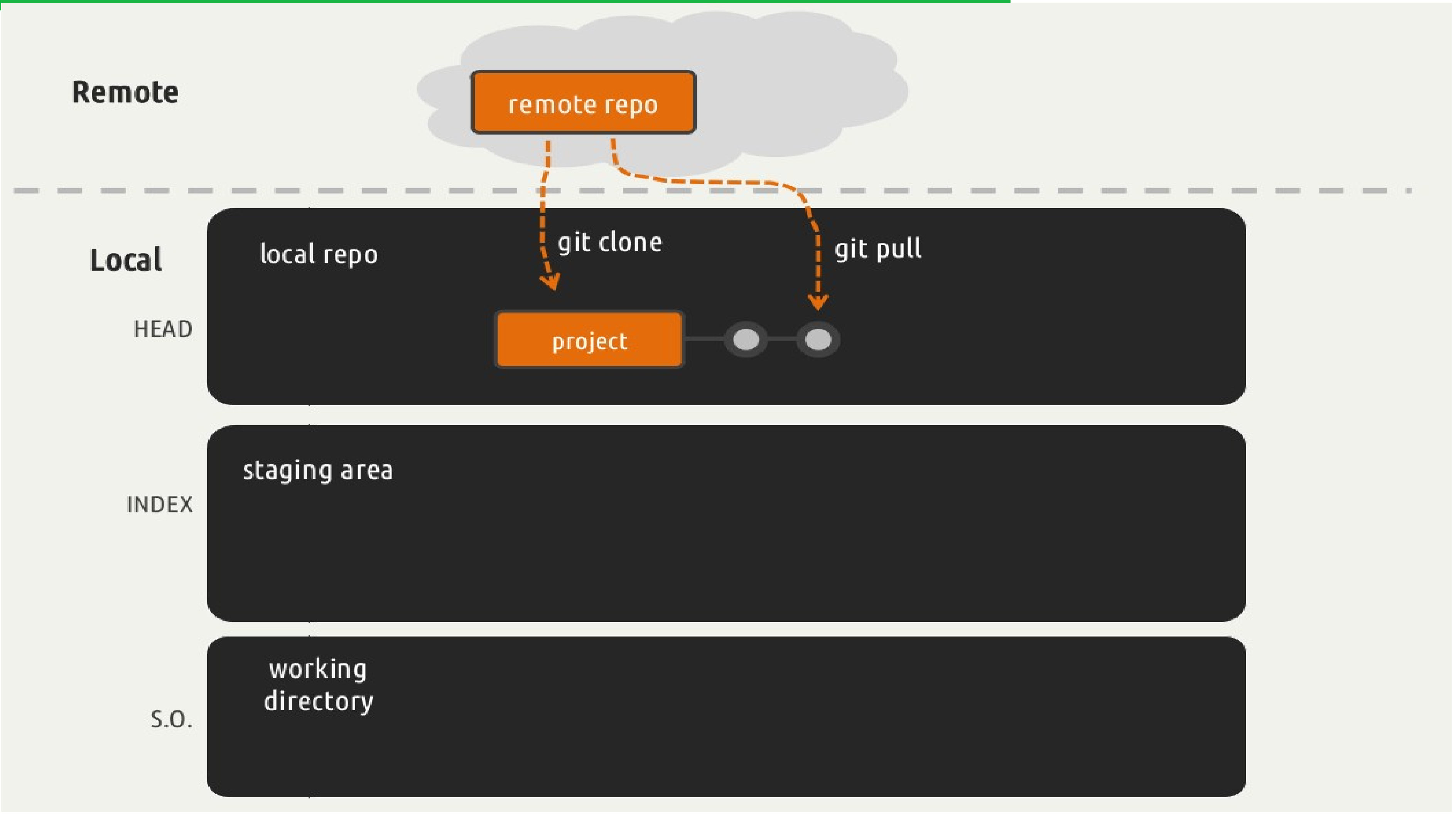
Git Basics

The Three States



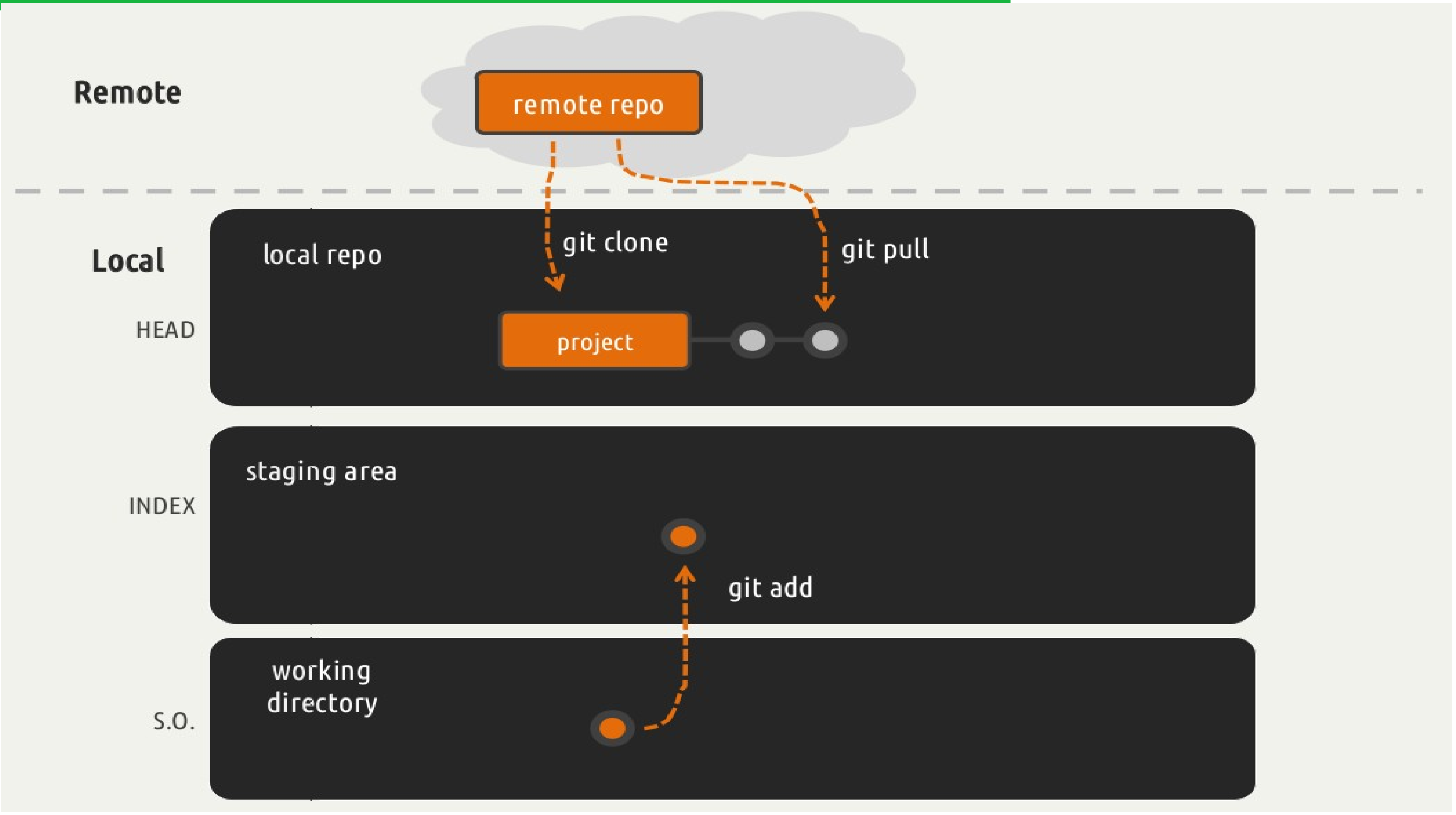
Git Basics

The Three States



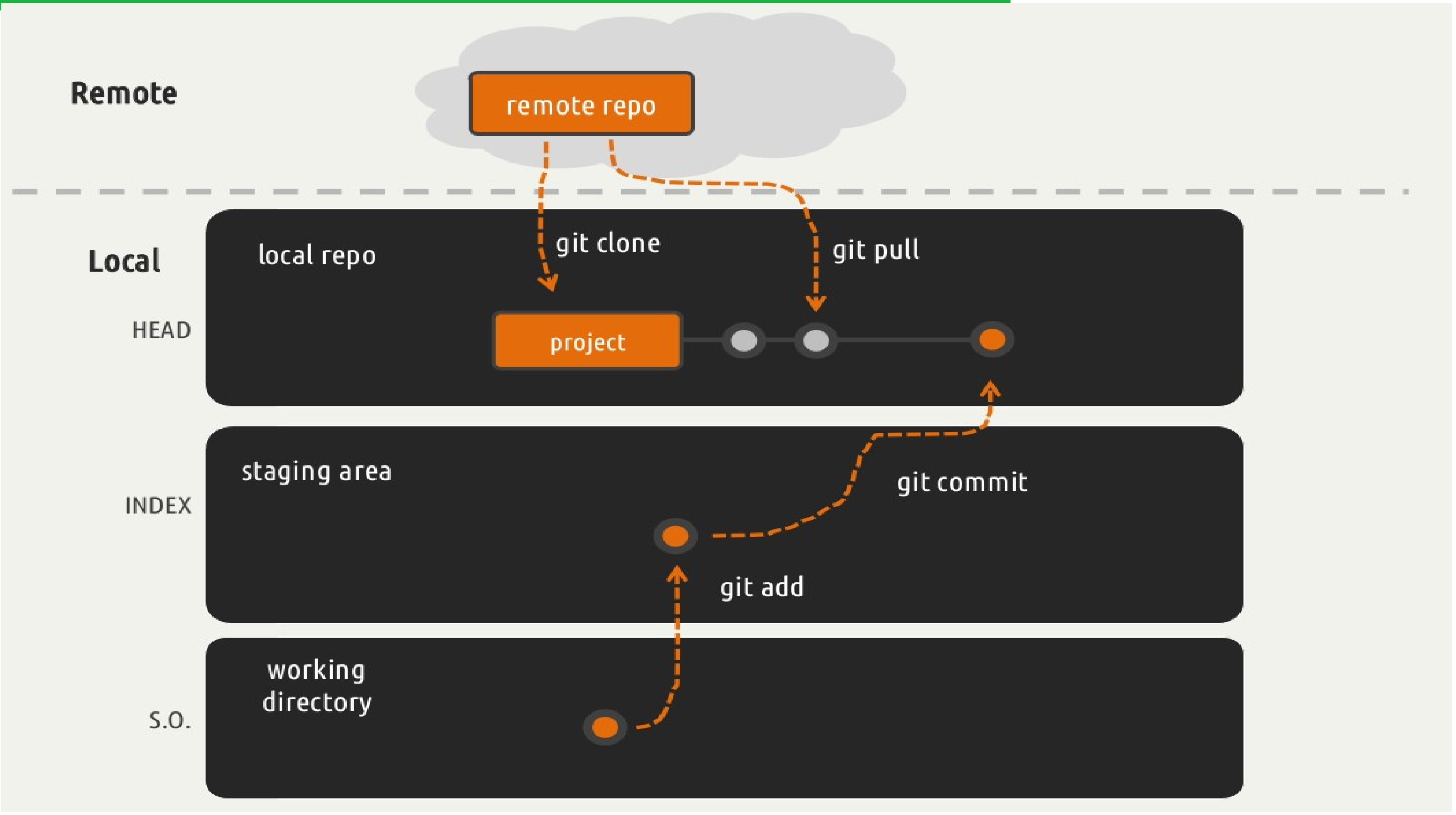
Git Basics

The Three States



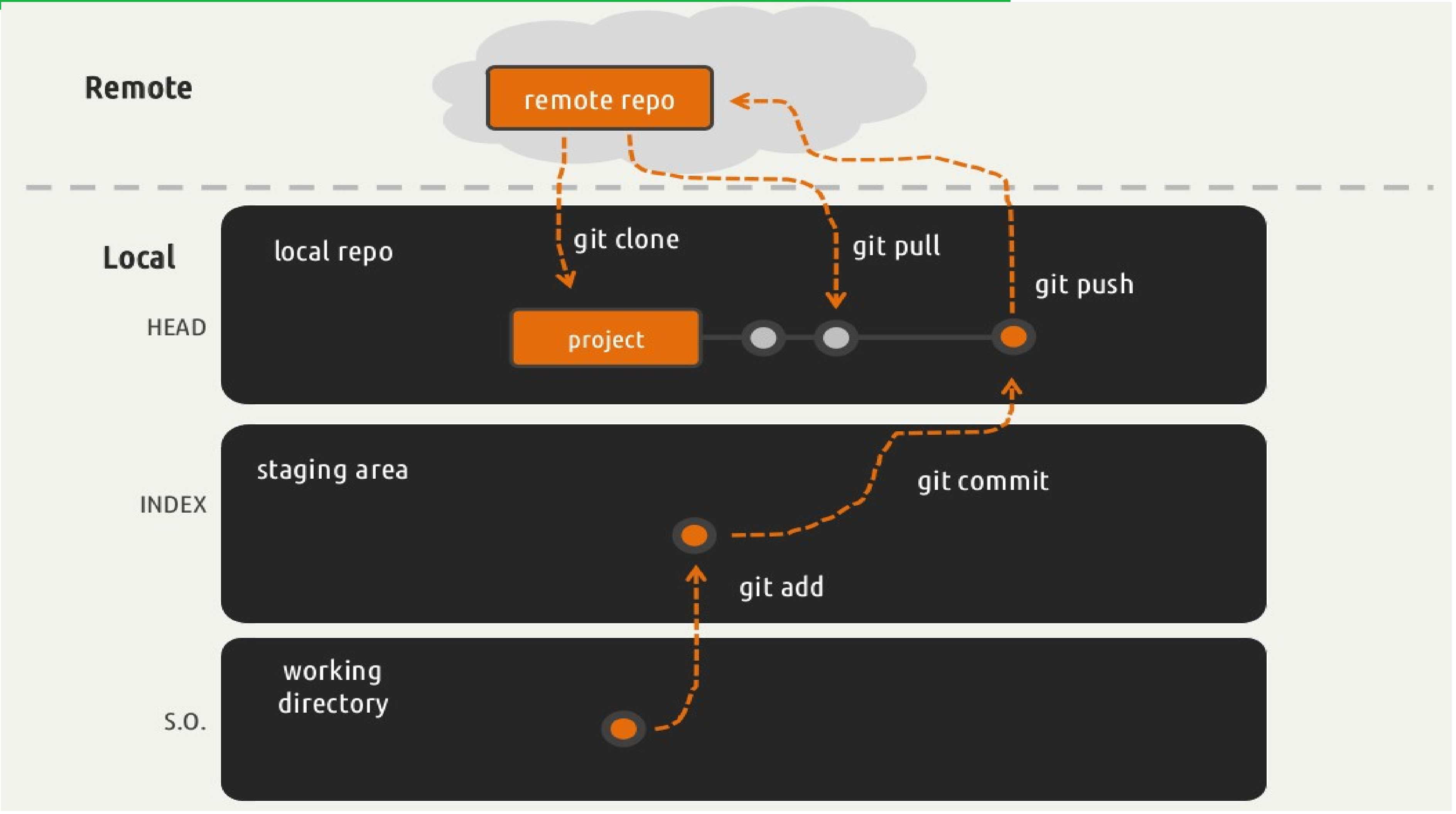
Git Basics

The Three States



Git Basics

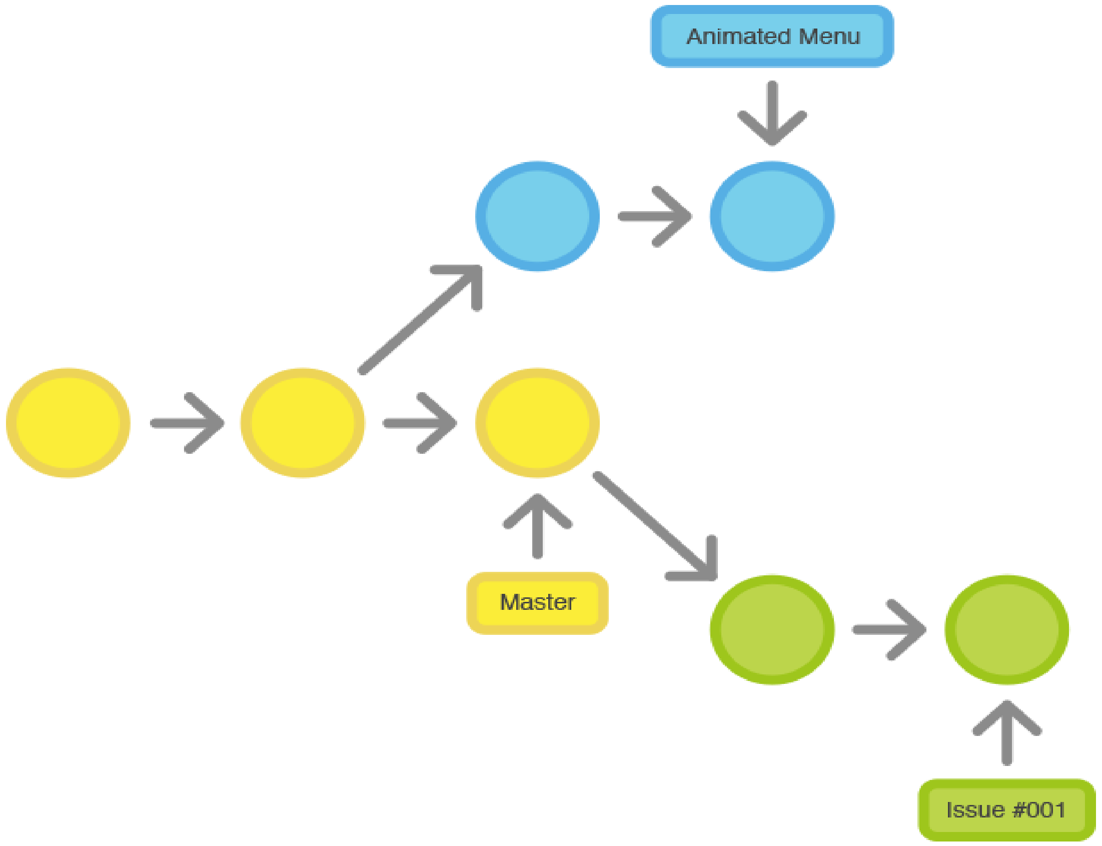
The Three States



Git Branching

What Branching is?

- **Branch** is essentially an independent line of development.



Git Branching

Why should I care about it?

- You are working in a software project and create a copy of your development code on your local machine.
  - You are working on a new feature
- You receive a call that another issue is critical and you need a hotfix.
  - What do you do?

Git Branching

Why should I care about it?

- You are working in a software project and create a copy of your development code on your local machine.
  - You are working on a new feature
- You receive a call that another issue is critical and you need a hotfix.
  - What do you do?

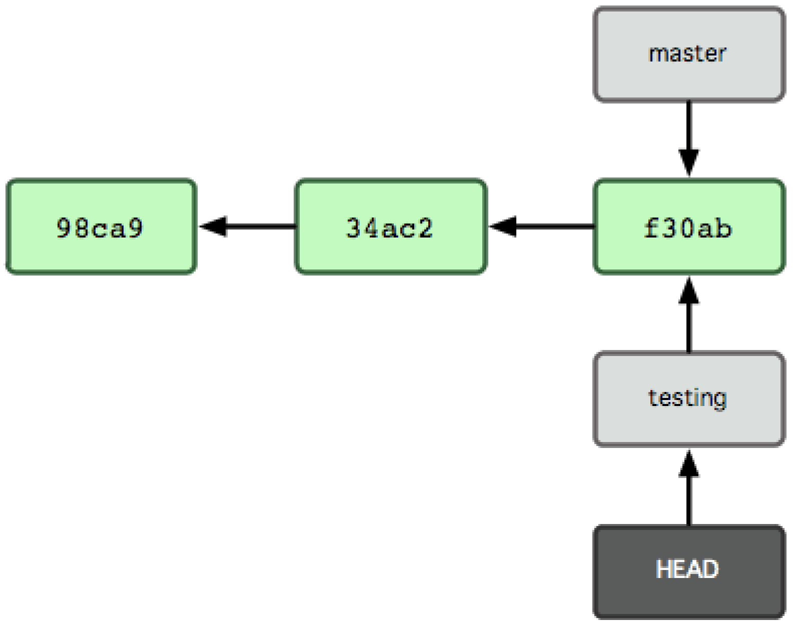
    1. Revert back to your production branch.
    2. Create a branch to add the hotfix.
    3. After it's tested, merge and push to remote repository.
    4. Switch back to your original story and continue working.

Git Branching

Basic Branching

- Create a new branch and switch to it at the same time

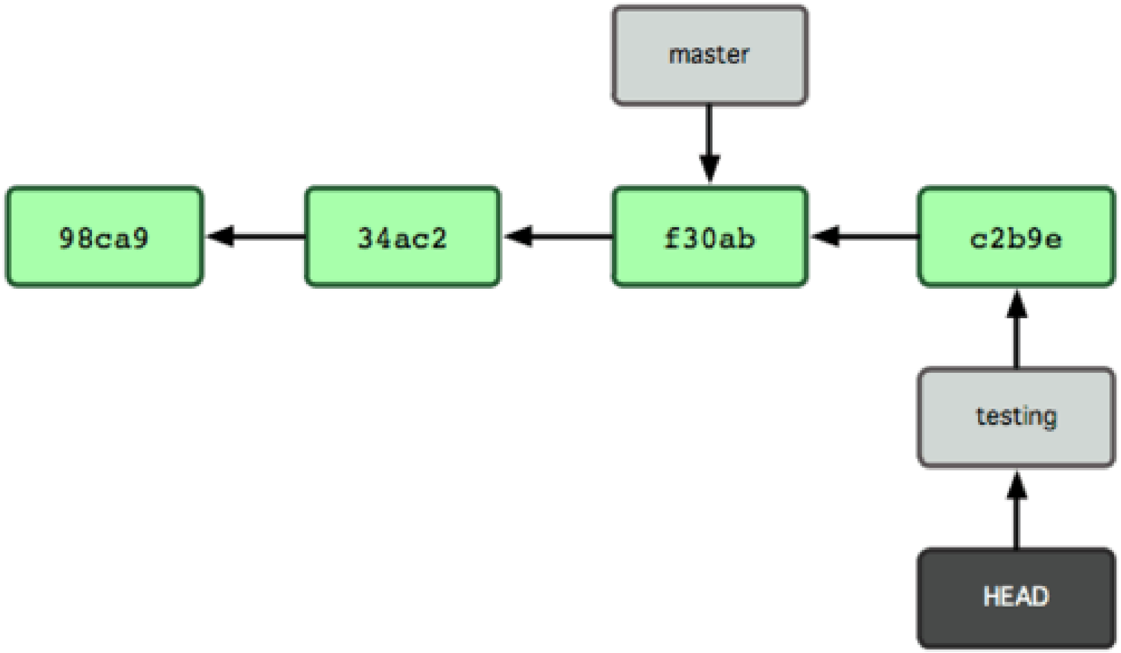
```
$ git checkout -b new-branch
```



Git Branching

Basic Branching

- Let's do another commit

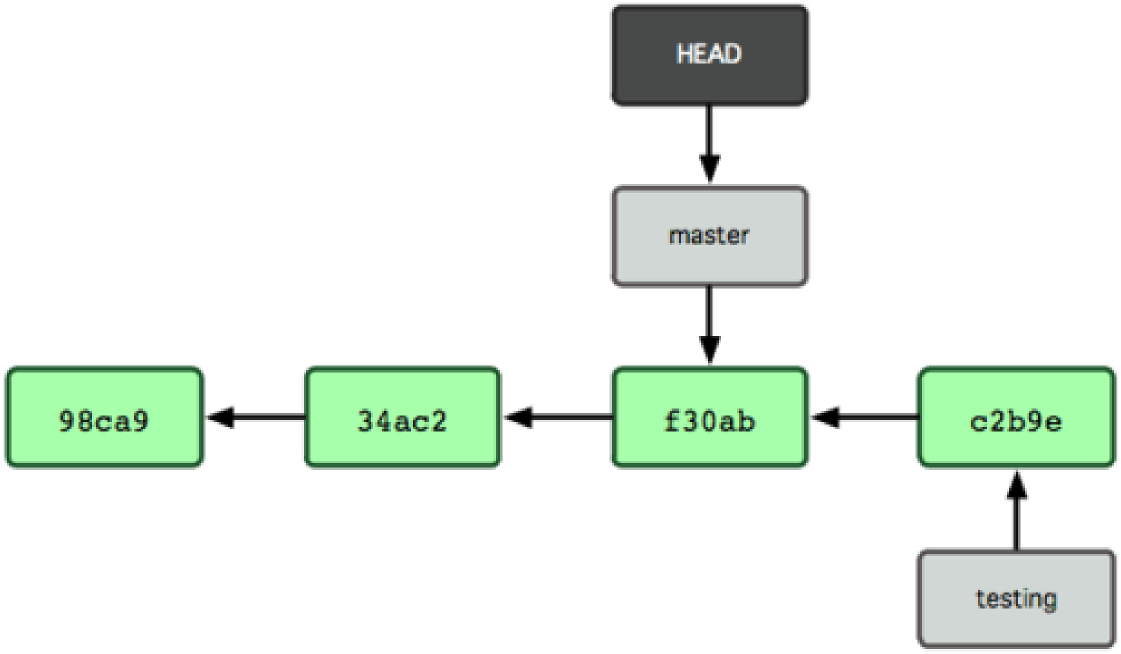


Git Branching

Basic Branching

- Let's switch back to the master branch

```
$ git checkout master
```



Git Branching

Basic Branching

- Merge into your master branch to deploy on production

```
$ git checkout master

$ git merge wip-hotfix
```

Git Branching

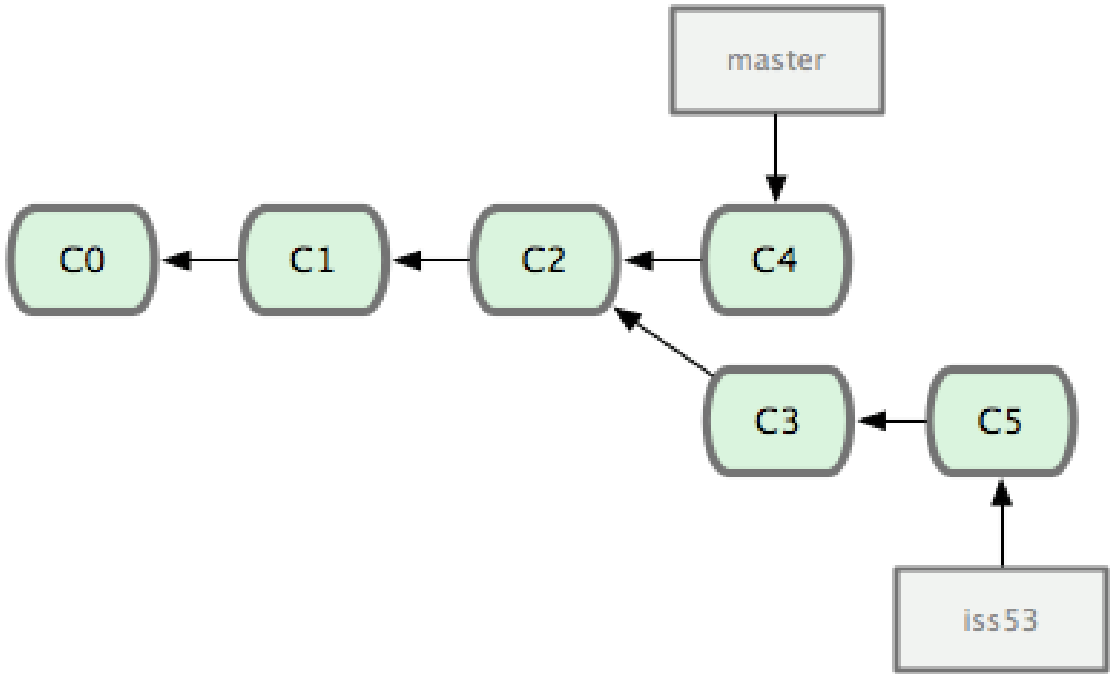
Basic Branching

- Merge into your master branch to deploy on production

```
$ git checkout master
$ git merge wip-hotfix

Updating f42c576..3a0874c
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
```

Git Branching

A branch can move forward independently



Git Branching

A branch can move forward indenpendently

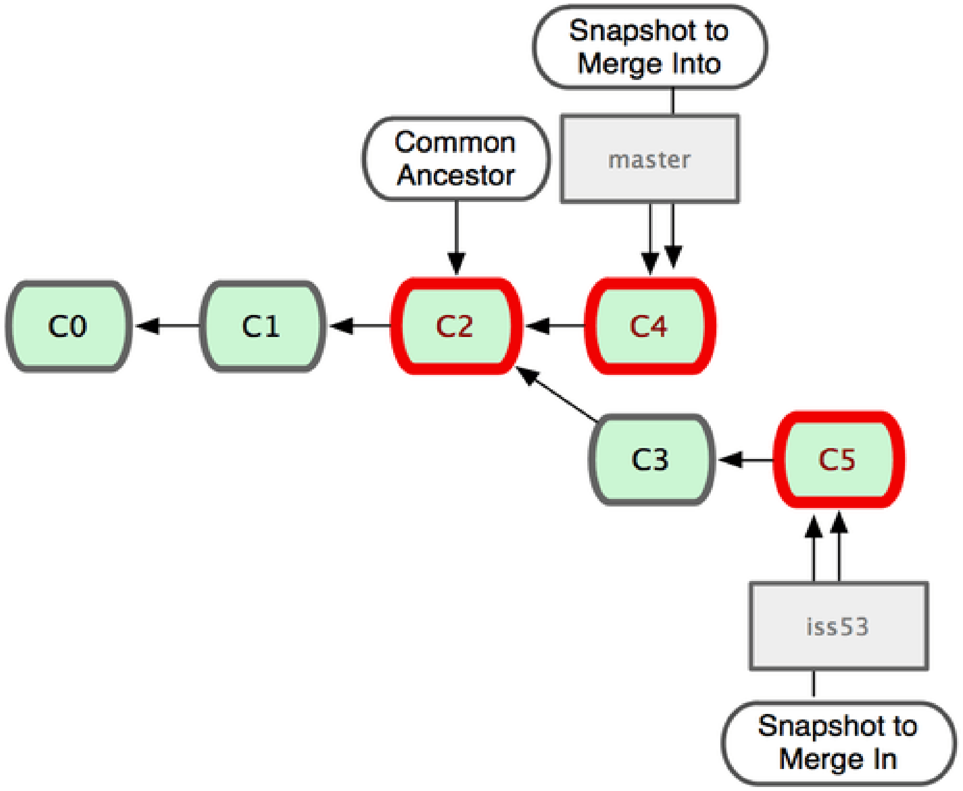
- Merge into your master branch to deploy on production

```
$ git checkout master

$ git merge iss53


Auto-merging README

Merge made by the 'recursive' strategy.

 README | 1 +

 1 file changed, 1 insertion(+)
```

Git Branching

A branch can move forward indenpendently

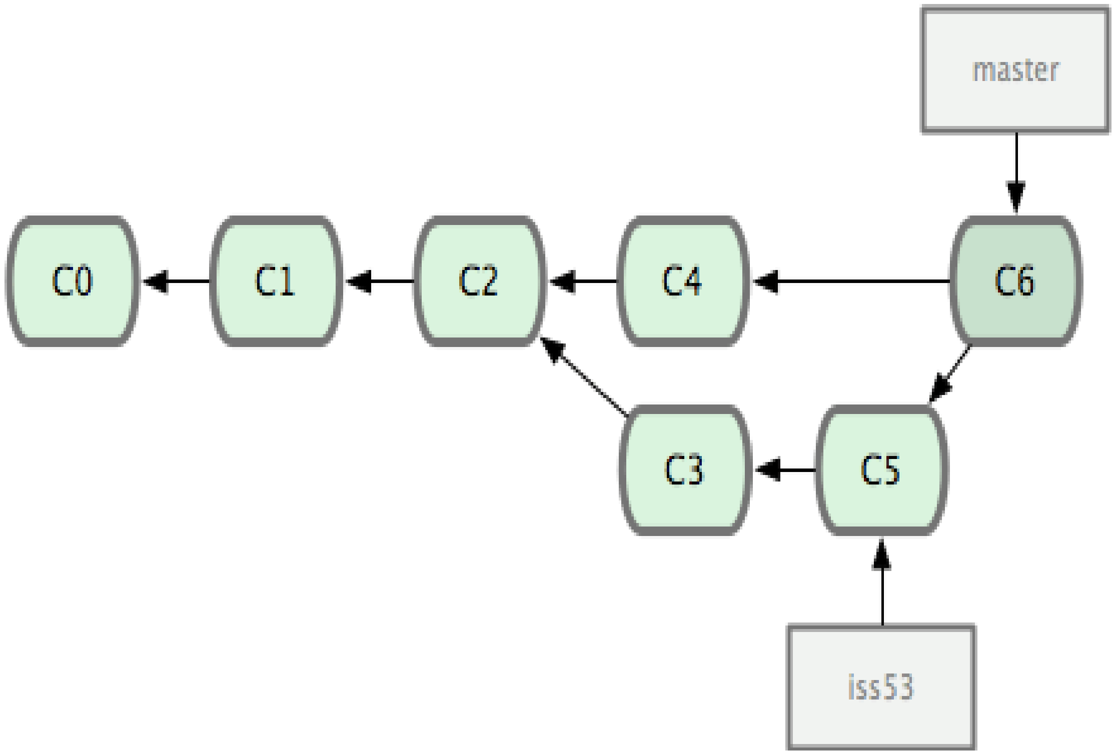
- Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.



Git Branching

A branch can move forward independently

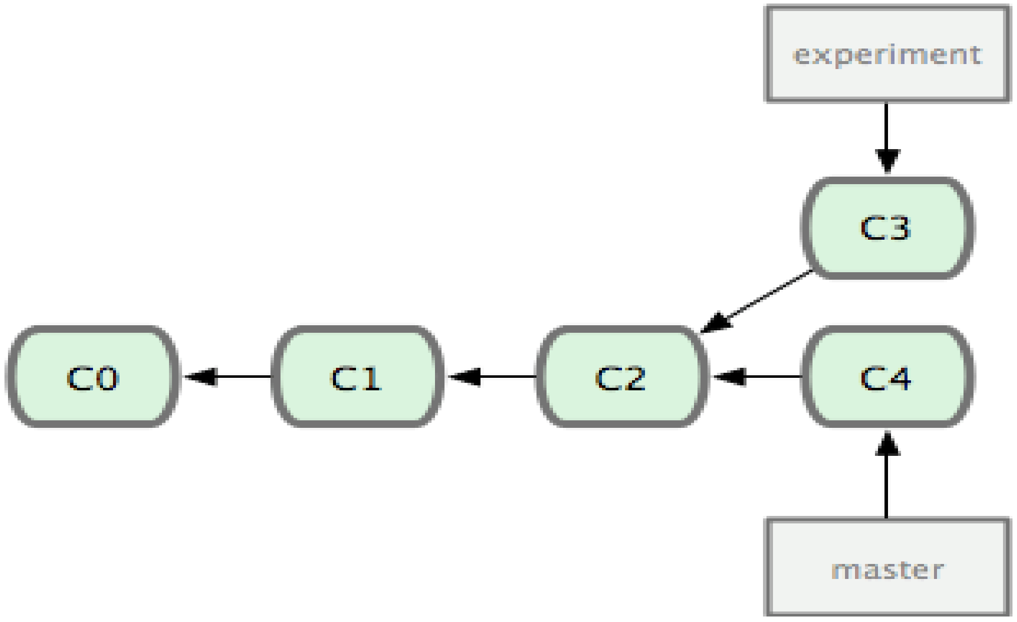
- Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it.



Git Rebase

The basic Rebase

- You can see that you diverged your work and made commits on two different branches.



Git Rebase

The basic Rebase

- Take all the changes that were committed on one branch and replay them on another one.

```
$ git checkout experiment

$ git rebase master
```

Git Rebase

The basic Rebase

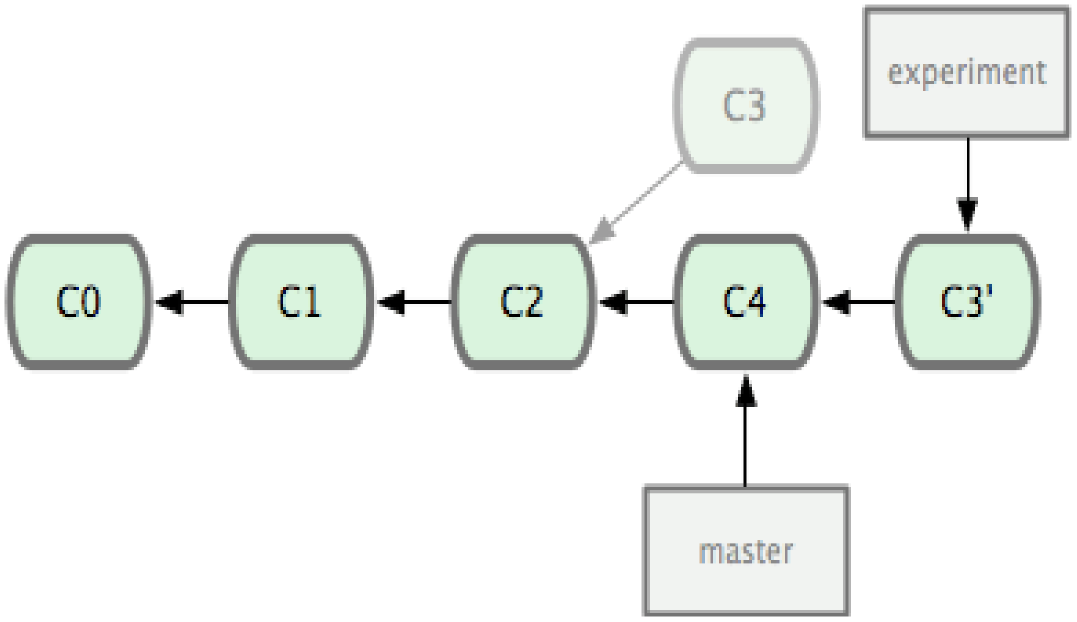
- Take all the changes that were committed on one branch and replay them on another one.

```
$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: added staged command
```

Git Rebase

The basic Rebase

- Rebasing the changes introduced in C3 and C4

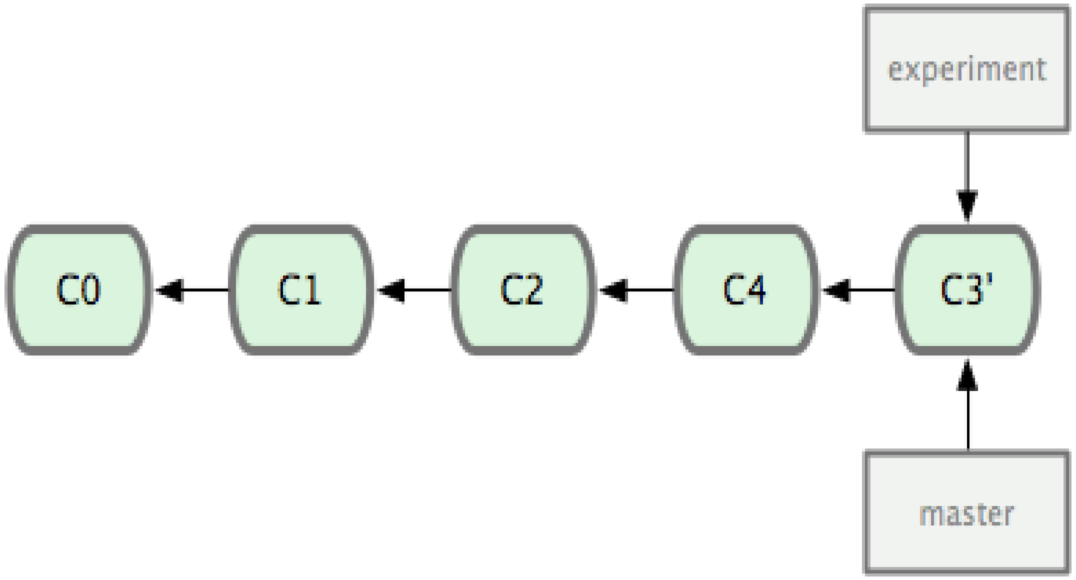


Git Rebase

The basic Rebase

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

```
$ git checkout master

$ git merge experiment
```



References

Proposed Scenario

- Chacon, S. “Pro Git”. Apress
  - Available on http://git-scm.com/book
- http://eagain.net/articles/git-for-computer-scientists/
- http://try.github.io/levels/1/challenges/1