git Primer

`git` is a wicked-powerful distributed revision control system. It is confusing to many, so there are myriad tutorials and explanations online to help people understand it. This one will focus on the fundamental concepts and tasks rather than trying to compete with the documentation.

> "I'm an egotistical bastard, and I name all my projects after myself. First Linux, now git." ~ Linus Torvalds

DEFINITIONS

Working Directory
: – the *working directory* is the directory where you have content that you want to manage with `git`.

Commit
: – a *commit* is a full snapshot of the contents of your working directory (everything being tracked by `git`, anyway), and it's kept track of using a unique 40 character SHA1 hash. This way, the exact state of your project can be referred to, copied, or restored at any time.

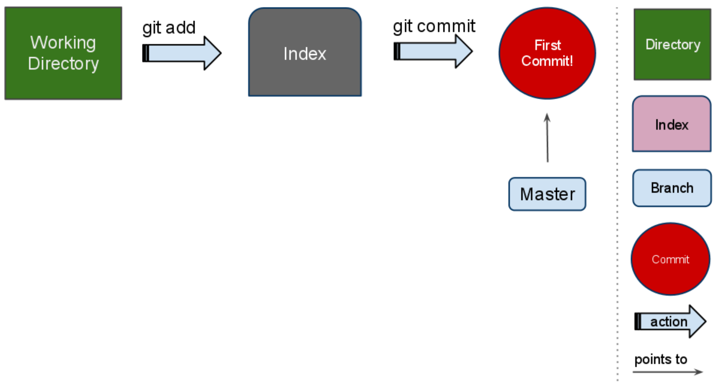
Index
: – the *index* can be considered a staging area. It's where snapshots of changes are placed (using `git add`) before they get committed. The index is crucial to `git` because it sits between the working directory and your various commits.

Branch

– a *branch* is similar in concept to other versioning systems, but in `git` it's simply a pointer to a particular commit. As you make additional commits within a branch, the branch pointer moves to point to your latest one. To revert to a previous version of your code, the branch pointer is simply moved to point to another commit within that branch.

Understanding how these components work together is the key to understanding `git`.

GIT'S INDEX



First and foremost, it's important to understand that `git` has something called an *index* that sits between your working directory and your commits. It's basically a staging area, so when you `git add` you copy a snapshot of your working directory to the index, and when you `git commit` you copy that same thing from the index to create a new commit.

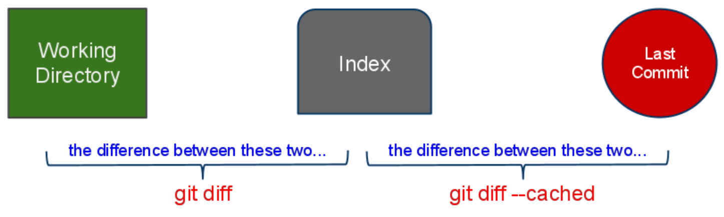
It is crucial to understand the intermediate (staging area) nature of the `git` index in order to grasp the relationship and differences between adding and committing content.

`git status` is very helpful in understanding git because it shows you the differences between the working directory and the index and previous commits. The "status" refers to the status of the working directory, so if you make a change in your content — say to *index.php* — and you run `git status`, it'll show you what's changed that isn't yet staged for a commit (in your index):

$ git status

```
On branch master
Changed but not updated:
   (use "git add ..." to update what will be committed)
   (use "git checkout -- ..." to discard changes in working directory)



modified:   index.php



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

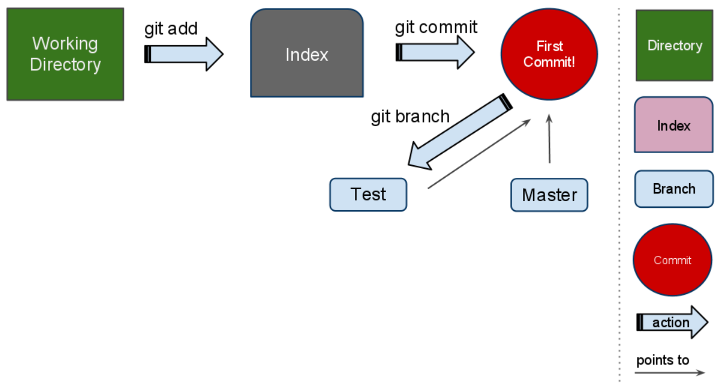
`git diff` is similar to `git status`, but it shows the differences between various commits, and between the working directory and the index. `git diff --cached`, on the other hand, shows you the difference between what's in the index vs. in your last commit.


Working Directory
Index
Last Commit
the difference between these two...
git diff
the difference between these two...
git diff --cached

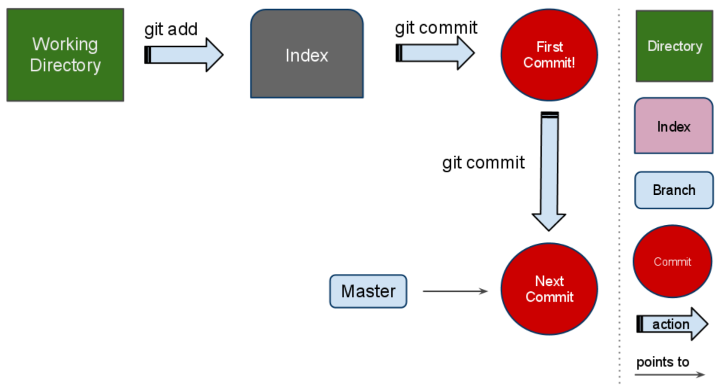
BRANCHES

Branches are just pointers to various commits. Every project has a default pointer called *Master*. As you continue to create commits within the default branch, the Master pointer follows you along so that it always points to the latest commit in the branch.

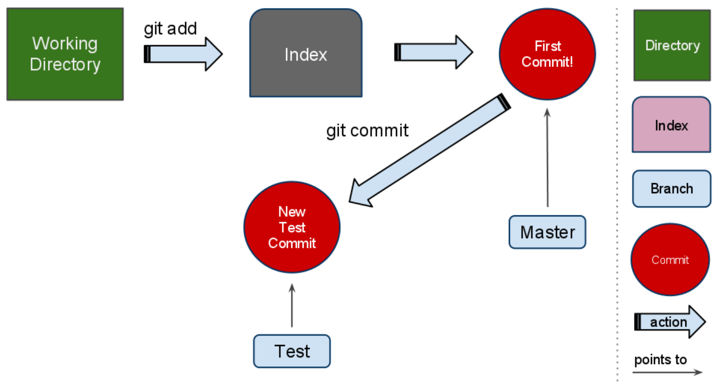
Branches are created using the `git branch` command, which creates a new branch label which points to the current commit. So, until a new commit is made, both the previous and new branch labels will point to the current commit. One way to think of them is "File->Save As Copy" for your codebase.



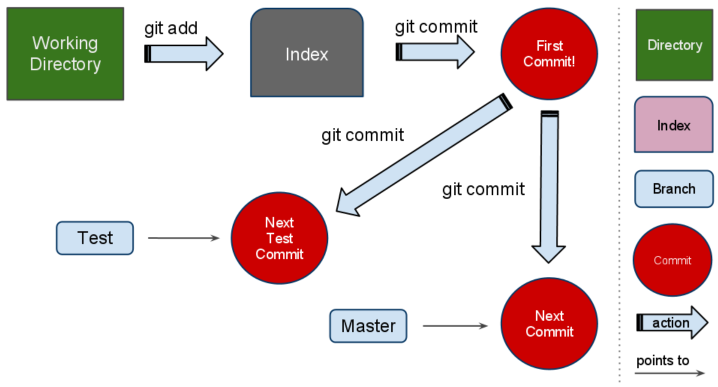
Using this model, someone can then checkout the Master branch, make changes, and commit them. The result would look like this.



Keep in mind that the "Next Commit" commit would only happen for someone who had the Master branch checked out. If the same person who performed the `git branch` command immediately made changes, added them, and then committed them, they'd have made another commit along the Test branch instead, like so:



If both branches had been checked out, worked on, and had a commit made, the diagram would look like so:



To change to and start working on a given branch, you have to check it out. This is done with the `git checkout` command:

```
$ git checkout
```

Keep in mind that when you do this you will pull the copy of your project that exists at that branch's latest commit point, and it will overwrite your current working directory with that version of your project.

REMOTE REPOSITORIES

Creating code branches on a single system is enjoyable enough, but the real purpose of `git` is allowing people in disparate locations to contribute to a project. You add remote repositories in `git` by using the `git remote` command, like so:

```
$ git remote add
remotesitessh://yourdomain.tld/path/to/remote/gitdirectory.gi
```

[This is using SSH as the protocol, but `git` supports many others as well.]

Then you push your content from your local repository to the remote one, using the name you've set up:

```
$ git push remotesite+master:refs/heads/master
```

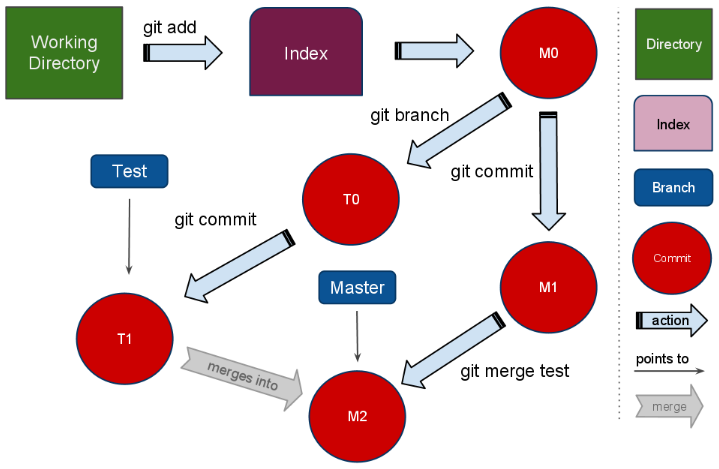
Then, to perform updates, you simply run `git push` with your target defined:

```
$ git push remotesite
```

MERGES

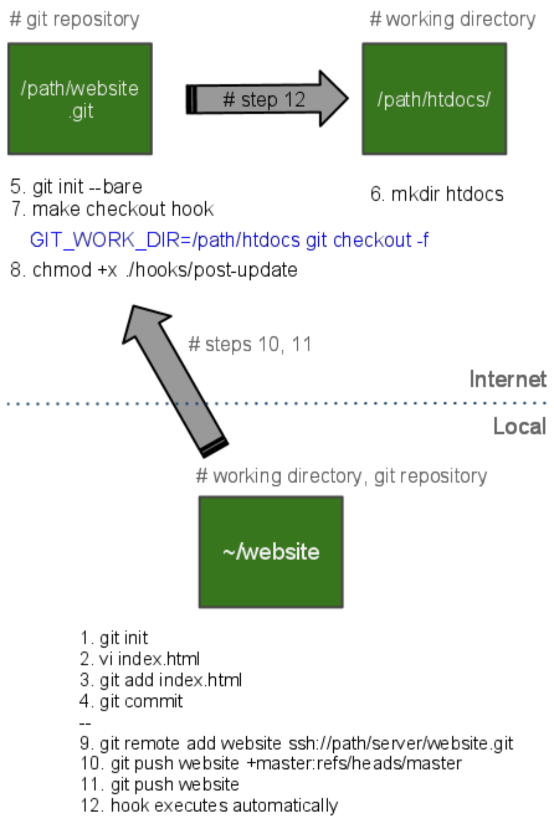
If you have more than one person working on the project — each doing his own thing — eventually you're going to have to synchronize. `git` handles this quite elegantly by simply moving the necessary branch pointers upon completion.

In the diagram below we merge Test into Master, and we simply end up with a later version of Master, as you might expect — only this version includes the changes that existed in Test as well.



NOTE: You want to make sure you're fully committed before you attempt a merge. Not doing so invites wrath.

MANAGING A WEBSITE USING `GIT`



So now that we've seen the various components of `git` in play, let's see how it all works together by creating a setup that allows us to manage a website remotely.

ON YOUR LOCAL SYSTEM

Either have, or create, a web directory on your local system. This is the main place you'll make changes from. Start by changing directory into it.

```
git init
```

Create some content.

```
vi index.html
```

Add the content to `git`'s index.

```
git add index.html
```

Commit the changes.

```
git commit
```

ON YOUR SERVER

Initiate a new `git` repository on the server.

```
mkdir /path/website.git && cd /path/website.git && git init
–bare
```

Create your hook that will checkout the code to your actual web directory.

```
vi /path/website.git/hooks/post-update
```

```
GIT_WORK_TREE=/path/htdocs git checkout -f
```

```
chmod +x /path/website.git/hooks/post-update
```

BACK ON THE LOCAL SYSTEM

Add the remote directory to the local config.

```
git remote add websitessh://path/server/website.git
```

Push the contents of the local repository to the remote one.

```
git push website +master:refs/heads/master
```

Then, change things locally and to upload changes, simply do a:

```
git push website
```

The changes will upload to the remote `git` repository and trigger the post-update hook, which copies the contents of the repository to the working directory — your live website directory (htdocs).

[To update from the server side you can execute `git`commands as usual, but you must provide environment context with each command, like so:]

```
GIT_DIR=/path/website.git GIT_WORK_DIR=/path/htdocs git
$some_git_command
```

[On the server you won't have to push after you add and commit, as using the environment variables above will mean the committed changes will already be present in the repository.]

TAGS

`git` has a powerful feature called *tags*, which allow you to define an intuitive name to a given commit — like "Gold Version", or "Version 3", or "Before Production".

These are done using `git tag` (easy enough). You can then check out that version just like a branch, only this will point to that specific point in time.

```
git tag Gold
```

…and later on…

```
git checkout Gold
```

ESSENTIAL COMMANDS

There are a good number of `git` commands, and the documentation is excellent, so I won't cover many here. Here are some I think are worth mentioning.

- `git bisect` – allows you to isolate the exact code push that caused a problem by executing `git bisect bad` to mark the current (broken location), then restore to a known-good configuration, and then git will step you through each commit you've made in between the two. For each one you then either git bisect good or bad until you've found your error-causing commit.
- `git stash` – sets changes you've made off to the side in a way that lets you bring them back later. Usage: you are about to checkout another branch which would crush your current changes, so you `git stash` them before doing your checkout.
- `git rm / mv` – deletes / moves items in the working directory with visibility to `git` so that you can commit afterwards.
- `git reset` – resets to a specified state (commit).
- `git status` – show the status of the working tree — including differences between the index and the current commit, differences between the working tree and the index, and items within the working tree that are not tracked by `git`.
- `git log` – show a log of commits
- `git clone` – clone a repository into a new directory.

- `git clean` – remove untracked files from the working directory.