NOTES: GIT ESSENTIAL TRAINING

GETTING STARTED WITH GIT

INITIALIZING A REPOSITORY

Once you have Git installed and configured, the next step is to initialize Git in a project, essentially to tell Git to start tracking things in this Git project. And the way we are going to do that is with another Git command, which is git space I-N-I-T, init, short for initialize. So we are going to tell Git to initialize the project to get everything ready to start doing its tracking. Now the first thing we have to decide is where we want to put this project.

`git init`

Git initialize, so git init, and this will tell Git, set up this as your home-base, make this a Git repository, and track all the files that come and go, the changes that are made inside this directory, things outside this directory, it's not concerned about, things in this directory, Git will be aware of, and it doesn't matter how deeply nested they are, Git is going to watch for them.

UNDERSTANDING WHERE GIT FILES ARE STORED

To show the hidden dot files (.files) use the -la attribute on the ls command

`ls -la`

So `.git` is a directory that is created by that initialize command. And this directory is a directory where Git stores all of its tracking information. Now that's all of its tracking information, it doesn't matter how deep down in other folders that we've got files going on, they are always going to be stored at the top level of our project inside this `.git` directory. You can think of it as Git's workspace where Git does everything that it's going to do, and if we wanted to remove Git and remove version control from our project, well, then it would just be a simple matter of removing this .git directory.

This is for our project level configuration. Now we have an interface through Git that we saw in the Configuration chapter where we don't have to come in here and edit it directly, you can just use git commands to set the different values that you need. But if you did ever need to, that's where this file is located. Everything else, you want to leave alone.

MAKING A COMMIT

Once you've made changes to your project, like adding, editing or deleting a file, you can switch over to the command line and tell Git to add all changes that have been made to your project. You do this with the `add` command.

```
git add .
```

In git, the dot, dot is short for, this directory. Once the changes have been added, you instruct git to then make these changes to its permanent repository.

```
git commit -m "message_text"
```

Now your changes are recorded in Git. This is the basic process for working with Git. It's really quite simple. You just make your changes. Then you add those changes. Then you commit them to the repository with a message, and that's it. That's the basic cycle that you're going to be following, ***make changes, add the changes, commit the changes***.

WRITING COMMIT MESSAGES

What you really want to do is have a commit message that describes the changes that you're making in that commit set, so added file to project, that would be more descriptive saying that we added this first file. That you added the JavaScript to something, or that you were fixing a bug.

We are labeling what we were doing in this change set so when we come back and look at it later, we can just look at the commit message and know what's inside, what's contained in that set. So we want to have good descriptive commit messages. There are also some other best practices that we should follow. We want to start with a short single-line summary, less than 50 characters, keep it short. Optionally, we can follow that by a blank line and then a more

complete description. Now if you are just making a tiny little change, a single-line summary will do it, but if you're committing a change that has lots of changes, that take place in lots of files, it might be worthwhile to have a more complete description.

Now the way we were doing it before from the single command line, that's a little tricky to do inside those double quotes, but we can also use a text editor to make these commit messages well, and that makes it easier to do multiline commit messages. Even then you want to keep those additional lines to less than 72 characters, and that's because different people may be looking at your commit log from different types of tools, they may be using Mac or Windows, viewing it on the web, graphical user interfaces, they may be receiving this commit information via email, so we want to limit it to 72 characters. And you want to write commit messages in the present tense, not in the past tense, you are labeling what this commit does, not what you--as the creator--were doing.

Label what it does, e.g., This fixes a bug, *not* I fixed a bug. It's not about you, it's about what this commit is. If you need to have bullet points that describe what happens you usually use asterisk or hyphens, and you can add ticket tracking numbers from bugs or support requests or develop a shorthand for your organization. So maybe you put it, for example, in square brackets at the beginning of your message that you are messing with the CSS or the JavaScript or maybe you label all bug fixes with bugfix:, or put a tracking number in the front letting it know what sort of support request ticket it goes with.

VIEWING THE COMMIT LOG

Now that we've made our first commit, and we've talked about how to write good commit messages, let's take a look at where those commit messages show up by viewing the commit log. The way that we do that is with git log, very common sense, and this will show us the log of commits that have taken place so far, right now there is only one. If there have been more than one then we would see them listed one after another.

`git log`

Each commit has a unique ID. It also lists the Author of the commit, pulled from my global configuration. So, it's very important to set the user name when you first set up your Git Repo, put in your information so that it can tell with each commit who is making the commit.

`git log -n number`

Limits the number of commits returned by the `git log` command.

`git log --since=2015-06-01`

Lists all the commits since 6/1/2015

`git log --until=2015-06-01`

Lists all the commits until 6/1/2015

`git log --author="sea"`

Lists all the commits by authors whose name begins with "sea"

`git log --grep="init"`

Lists all the commits that have strings in the commit message field that match the regular expression , in this case, "init"

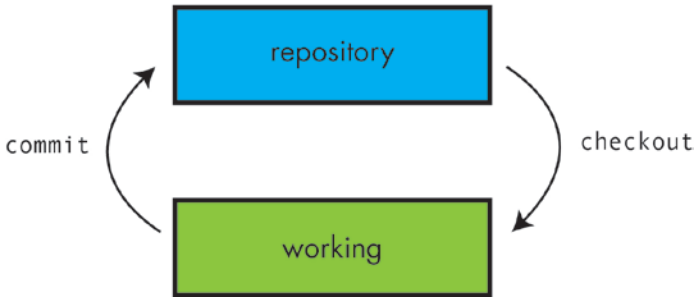
GIT CONCEPTS AND ARCHITECTURE

THREE-TREES ARCHITECTURE

A lot of other version control systems use two trees. We call them trees because they represent a file structure. All right, our working copy begins with the top of our project directory and below of that might be four or five different folders that have a few files in them, maybe a few more folders, each of those folders has a few more folders in it, and you can imagine that if you map that out, that each of those folders would then branch out like the branches of a tree.

It's really a directory tree whose trunk begins with the root of our project. Now the repository also has a set of files in it. And when we want to move files between the repository or the working copy, we check out copies--that's the term that we use--we check it out from the repository into our working directory, and when we finish making our changes we commit those changes back to the repository. Now the reason why there are two distinct trees is that these files don't have to be the same between them. If I check out copy from the repository, I make some changes into it. I save those changes on my hard drive.

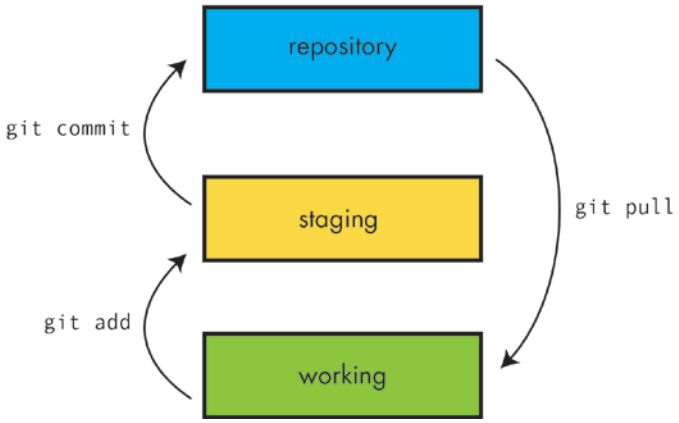
two-tree architecture



Now those changes are saved. They are permanent, but they are saved in my working copy only. They're not yet committed to the repository. So my working copy looks different from the repository. Both are saved, it's not like I haven't saved the files, I've done that. They just aren't saved and tracked in the version control repository. Now if the repository is a shared repository, and there are many people working from it, they may commit their changes to the repository. And if I haven't checked out a copy recently to get those changes, then my working copy doesn't have their changes.

So once again the repository and the working trees will not have the same information in them. That's a typical two-tree architecture. Git however uses a three-tree architecture.

three-tree architecture



It still has the repository and the working copies, but in between is another tree which is the staging index. Remember when we did our first commit in the last chapter, we didn't just do a commit, we did an add first. We added, then we committed, it was a two-step process. That add, added our files to the staging index, and then from there we committed to the repository.

It's important that you understand that this is part of the architecture of Git, and it's a really nice feature. Because then what it means is that we can make changes to ten different files in our working copy. And then we can say, all right, I am ready to make a commit, but I don't want to commit all ten of those, I just want to commit five of these as one changed set. So what I am going to do is I am going to put those on the staging index, add them to the staging index, get those five files ready to go, and as soon as I am satisfied that they are ready, now I will commit those five files in one changed set to the repository.

USING HASH VALUES (SHA-1)

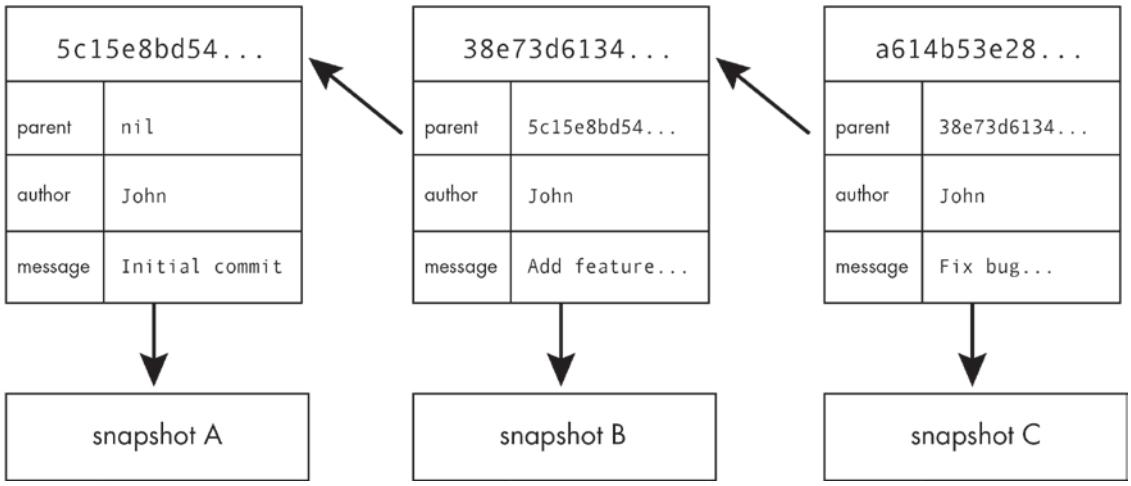
- Git generates a checksum for each change set
  - checksum algorithms convert data into a simpler number
  - same data always equals same checksum
- data integrity is fundamental
  - changing data would change checksum
- Git uses SHA-1 hash algorithm to create checksums
  - 40-character hexadecimal string(0-9, a-f)
  - Example: 5c15e8bd540c113cd2d9eac6f64cacbc5ff6fe9c

So what Git does is it takes the entire set of changes, runs them through in algorithm, and in the end comes out with this one 40 digit number, you can see this number by using the `git log` command. The commit value or commit ID, you can call it whatever you want, is the SHA value. But it is a number that will be unique to the changes that are in this commit. So the way that Git actually attaches that information is that it feeds the snapshot of the file-set changes into its algorithm and comes up with an S-H-A value. Then it attaches a bit of meta information to each snapshot, combining that commit number at the top, along with the parent commit or the commit that comes before it, the author of the commit, and then the commit message.

So below you can see how the series of those commits are linked together, you can see that the parent for each one refers to the SHA-1 value of the other one before the identifier that come before, and that's how it knows the sequence of those commits. And then each one of those, each bit of meta information, points at a snapshot a set of changes or a Git object.

Understanding how Git generates these hash values is important, because it helps us understand how Git summarizes these snapshots. It illustrates the data integrity that's built into Git, and most importantly we're going to be using these SHA-1 hash values to refer to the commits.

referring to commits



WORKING WITH THE **HEAD** POINTER

HEAD

- Pointer to “tip” of current branch in repository
- Last state of repository, what was last checked out
- Points to parent of next commit
    - where writing commits takes place

Now you can move the *HEAD* around like the recording head on a cassette recorder. You can move the head to different places, but wherever the head is positioned when you hit Record again that's where it's going to start recording. The HEAD pointer in Git is very similar. It points at the place where we're going to start recording next. It's the place where we left off in our repository for the things that we've committed.

So HEAD always points to the tip of the currently checked out branch from the repository.

MANAGING CHANGES TO FILES

ADDING FILES

`git status`

Git status is going to report back to us the difference between our working directory, the staging index, and the repository. It's going to let you know, what is the status between those three different trees?

`git add file_name`

Adds a specific file to the staging directory

`git add .` *or* **`git add --all`**

Adds all new and changed files to the staging directory.

VIEWING CHANGES WITH *DIFF*

Often you will want to see what changes have been made before making a decision about whether to commit those changed files to a repository.

In the UNIX world it's very common to use a program called diff, D-I-F-F, in order to compare two files. And so Git uses that as the term that it uses to show us a diff between the old version and the new version. And we do that just with `git diff`.

`git diff`

So `git diff` will compare the two, that's comparing what's in the repository, the version that HEAD is pointing at, versus what's in our working directory. So the version in the repository is the one with the minuses, the one that has the pluses is going to be what's in the new version.

VIEWING ONLY STAGED CHANGES

The command, `git diff`, reports the changes that are in the working directory, comparing those files against the staging index and the repository. So what if you want to compare the differences between the staging directory and the repository? Well, we do the same thing using git diff, but now we pass in another option, which is staged, `git diff --staged`.

`git diff --staged`

It's the option to diff that tells it to look at what's in the staging index and compare that against the repository. It reports on only the changes pending in the staging directory, where as git diff by itself will return the changes that are in the working directory.

DELETING FILES

When we talk about deleting files, we're talking about deleting them from the repo. We're talking about the things that are tracked files. So, the first thing we need to do is add these to our repo, and then we can go about deleting them.

Now, there are two different ways of doing deletions. The first way is to simply delete a tracked file from the working directory. If you delete a tracked file from your working directory and then run `git status`, git will report just that – a tracked file has been deleted.

To remove the file from the staging directory, you will need to use the UNIX command, `rm`.

`git rm file_to_delete`

That's the first way to remove files. The second and easier way is to just do the whole remove process through git, without first deleting the file. Just run `git rm file_to_delete`. It does the exact same thing with one slight difference. In this case the file is completely erased it because it uses the UNIX remove, not the "let's put things in the trash" remove.

MOVING AND RENAMING FILES

As with deleting files, there's two ways that you can move or rename a file. The first way is through the regular operating system – basically doing all of your moving and renaming there, then coming back to git and having it stage those changes.

The second way is to do it all from Git, and let Git handle working in the operating system for us, just like it did with delete. So let's look at both of these. First, let's look at renaming. So, let's say we've got *first_file.txt* and you want to change it to *primary_file.txt*.

So, if we rename it using the file system, `git status` reports that *first_file.txt* was deleted, and that there is a new file, *primary_file.txt,* that's untracked. That's how Git sees what happened. It sees the fact that the file that it was expecting to find is not there, but now there is a new file there instead.

If you add those changes to the staging index, Git will now recognize that the file has been renamed. So, once it gets to the staging index, it actually compares the data between the two and says oh, these are pretty close. The files don't have to be exactly the same. Some changes could have taken place. I think the threshold is about 50%. As long as the data is about 50% the same between the two of them, it says oh, okay, this is the same file.

The second method of renaming a file is to let Git managing the file system for you, using the UNIX mv command.

`git mv` *`old_file_name.txt  new_file_name.txt`*

So, we're moving old_file_name.txt to new_file_name.txt, which is essentially the same thing as a rename. We tell Git to do it though. If no do `git status`, you'll see that it went ahead and added it to the staging index for us just like delete did. It said, all right, I'm going to change it in the file system, and add it to the staging index all in one step. So, I think the lesson from both deleting and the moving and renaming is that it's easier to have Git handle it because Git will take care of adding it to the staging index for you right away.

UNDOING WORKING DIRECTORY CHANGES

It's very common that you say, "Oh you know what? I want to undo what I just did," and Git can help you do that. One way of restoring files to an earlier version is to go back to the repository, get that version of the file, check it out, and replace what I have in my working directory with it.

`git checkout file_name|branch_name`

The checkout command is used for more than one purpose. In addition to retrieving the current version of a file from the repository, it's also used for working with branches. What checkout does is go to the repository, get the named thing, and make my working directory look like that. That's what it does. So if that named thing is a branch, it brings the branch down. If that named thing is the file, it brings the file down. Imagine for a moment that we wanted to bring down this resources folder.

So we say git checkout resources. Well, that's fine, but what if we also have a branch named resources? Then it's hard for Git to tell which one we need and in that case it actually would give us the branch resources instead of the folder resources. So as a result, it's a good practice when we're not trying to checkout a branch to put dash--dash, followed by a file or directory name. That instructs Git to stay on the current branch. That bare double dash is just there to indicate that we're not checking out a new branch, we're just talking about a file in the current branch.

UNDOING STAGING DIRECTORY CHANGES

Let's say you need to unstage a file, leaving everything else alone all the other ones in there -- a common real world application of how Git is used. How do you unstage a file?

`git reset HEAD file_name`

What we're telling Git to do is to go look at the HEAD pointer. The HEAD pointer points to the last commit of the tip of the current branch, which is master. That's our current branch. Go look at that last commit and reset yourself to be the same as what that has. Very similar to

what checkout did. Checkout went and checked out that file from the Repo. Here we're resetting the index to be the same as that.

AMMENDING COMMITS

Well, what about undoing changes that we've made to the repository itself, undoing commits that we've made? Well, that becomes a lot trickier than reverting to older file versions in the working directory, or unstaging files in the staging index. If you remember, Git refers to commits by a hash value that's generated from all of the data that's in the snapshot. If we change any of that information, then the hash changes, which completely breaks the data integrity of the data that's in Git. So Git doesn't want us to do that.

However, it is possible for us just to change the last commit, because nothing depends on it yet. The most recent commit that HEAD points to can be edited. Once we've tacked another on to the end of that we can't edit in anymore, but the one at the end is still editable, and we can do that using the amend option.

```
git commit --ammend  -m "commit message"
```

RETRIEVING OLD VERSIONS

We've seen how difficult it is to amend older commits, because it violates the data integrity, which is an important feature of Git. Instead, if we need to make changes to those older commits, the best advice is to make new commits, commits that undo what was done in those older commits. Not only does it maintain the data integrity of Git, but then the log file also accurately reflects the changes that were made over time, and it shows that a change was made and then several commits later another commit was made.

That information might actually have some importance, especially if the commits in between played off of those changes in any way. Now, what it doesn't do is allow us to hide our mistakes. Instead, you'll just have to record the mistake you made and then record the fix that goes with it. In order to make new commits that undo changes, there's a number of different ways that we could do it, one would be we could simply manually make those changes just like we did the first time and then commit the result. But there is another way that we can do it that's a little bit faster too.

To revert to an older version of a particular file, you will need the first 10 digits or so of the commit id. You can get this by running `git log`. When you have the id, you can pull down the older version with

`git checkout XXXXXXXXXX -- file_name`

The double-dash tell git that you want to pull the file from the current branch. With this command, it puts the file in the staging index, unlike the previous example of the checkout command, where it brought the file into the working directory. To see the differences between the version new in the staging index and the copy in the working directory use

`git dif --staged`

When reverting to a previous commit, it's a good idea to refrence the SHA value of that commit.

REVERTING A COMMIT

Git also gives us a helpful way when we really want to just undo all the changes for a commit completely and totally, the Revert command. To revert a commit you need to the id number of the commit to be undone. You can get the id by running the `git log` command.

`git revert XXXXXXXXXXXXXXXXXXXX`

This command reverts the version and commits it to the repository all in one step. To revert and stage without making the commit, use the `-n` flag

`git revert XXXXXXXXXXXXXXXXXXXX -n`

USING RESET TO UNDO MULTIPLE COMMITS

Git provides a very powerful tool that allows us to undo multiple commits. And because it's powerful, it's also very dangerous, so ***you need to use this with extreme caution***. The command that we're going to be using is *git reset*. What git reset does is it allows us to specify where the HEAD pointer should point to. Normally, we just let Git manage the HEAD pointer for us.

With Reset, we're telling Git, "I want to be in control, I want to move the HEAD pointer over here, and that's where you're going to start recording from now on, that's where you're going to start making your commits."

Git reset always moves the HEAD pointer. That's one thing that it does in every case. But there are three different options that I want us to look at that we can use to control some of the other behaviors that it has, and those options are going to be *soft*, *mixed*, and *hard*.

- SOFT -- moves the HEAD pointer to the specified commit, but it's not going to change the staging index. It's the safest of all the options. The result of that, if we've rewound backwards is that our staging index and our working directory are going to contain the files in their later revised state. The repository is going to be set back to an earlier version. So, if we do a `diff` between the two, it's going to tell us about all those changes that have happened between the point where the HEAD is pointing, and all the files that are sitting in our staging index and working directory.

- MIXED -- This is in between soft and hard which is why it's called mixed, and it is the default. What it does is it moves the HEAD pointer to the specified commit, and it also changes the staging index to match the repository. It does not change your working directory though. So, at this point, the staging index and the repository will be set in one place, our working directory, though, has all those changes that we've made. All the things that were in later versions of the repository are still in our working directory. We haven't lost any work. It's just waiting for us to stage it and then commit it.

- HARD -- A hard reset will not only move the pointer of the repository, but it will make your staging index and your working directory match that as well. So that means any changes that came after that commit are completely obliterated. They don't exist in the repository, the staging index, or the working directory, they're completely gone. So use this one with caution. It really is for when things have gone completely wrong, and you really just want to reset everything back to a specific point in time, and you don't mind losing whatever came after it.

THE SOFT RESET -- MAKE SURE TO COPY THE LAST FEW COMMITS FROM THE LOG BEFORE REWINDING. ONCE HEAD HAS BEEN MOVED, THE LATER COMMITS WON'T SHOW UP ON THE LOG. TO CHECK WHERE THE HEAD CURRENTLY POINTS

```
cat .git/HEAD
```

To get the id for the current version

```
cat .git/refs/heads/master
```

This will return the SHA for the current commit. To perform a soft reset use this:

```
git reset --soft XXXXXXXXXXXXXXXXXXXX
```

THE MIXED RESET -- CHANGES THE STAGING INDEX TO MATCH THE REPOSITORY

```
git reset --mixed XXXXXXXXXXXXXXXXXXXX
```

So if you find that you've made two or three commits, and then you think, you know what? I wish I could just go back and redo those commits again, this is what does it for you. It allows you to rewind back, let's say, three commits, all of those changes are still in your working directory, now you can go and make those commits over again.

THE HARD RESET -- CHANGES THE STAGING INDEX AND THE WORKING DIRECTORY TO MATCH THE REPOSITORY

Of the three types of reset, the *hard reset* is the most destructive. All three types of reset rewind the HEAD pointer to point to another commit, but the other two leave the files either in our staging index or in our working directory so that we then have those changes still at hand ready to remake those commits. The hard reset doesn't do that.

Hard reset makes our staging index and our working directory exactly match the repo. It throws out everything that happened after that. Those commits are not just sitting there waiting for us to recommit, we're now essentially rewound back to that previous commit.

```
git reset --hard XXXXXXXXXXXXXXXXXXXX
```

REMOVING UNTRACKED FILES

If you have a lot of files that have been added to our working directory that are not tracked that we don't want, we just want to get rid of them, Git gives us a quick and easy way to just tell all those files that should be thrown away.

The way that we just say throw them away is with `git clean`, and git clean on its own won't actually do anything. You need either a -n or a -f option. -n is a test run. -n generates a report as to which changes would be thrown out.

`git clean -n`

`git clean -f` forces it to run. We have to add this extra flag to it because it is going to be destructive. It is going to throw away anything that is not in our repository. However, if we have something in our staging directory it will not delete the file in the working directory until it's unstaged.

`git clean -f`

IGORING FILES

USING .GITIGNORE FILES

To tell Git which files it ought to ignore, we're going to create a special file in the root of our project, in the root of the working directory. And that file is going to be called .gitignore, so it's all run together, no spaces or punctuation except for the period at the beginning, .gitignore this file is going to provide Git with a set of rules that it can use to know which files to use for commits and which ones should be ignored. Those rules can be very simple, just a list of files one for each line or we can get little fancier, and you some very basic regular expressions.

We can use the Asterisk, the Question Mark, a bracket of characters, a character set, or a range like 0-9. So it's really pretty limited, we just have some basic wildcards that we can use. We can also negate expressions by putting an exclamation point in them. So, for example, we could say ignore any file that ends in .php. We're using the asterisk wildcard for one or more characters, so one or more characters ending in .php will get ignored but don't ignore index.php.