GIT Essentials

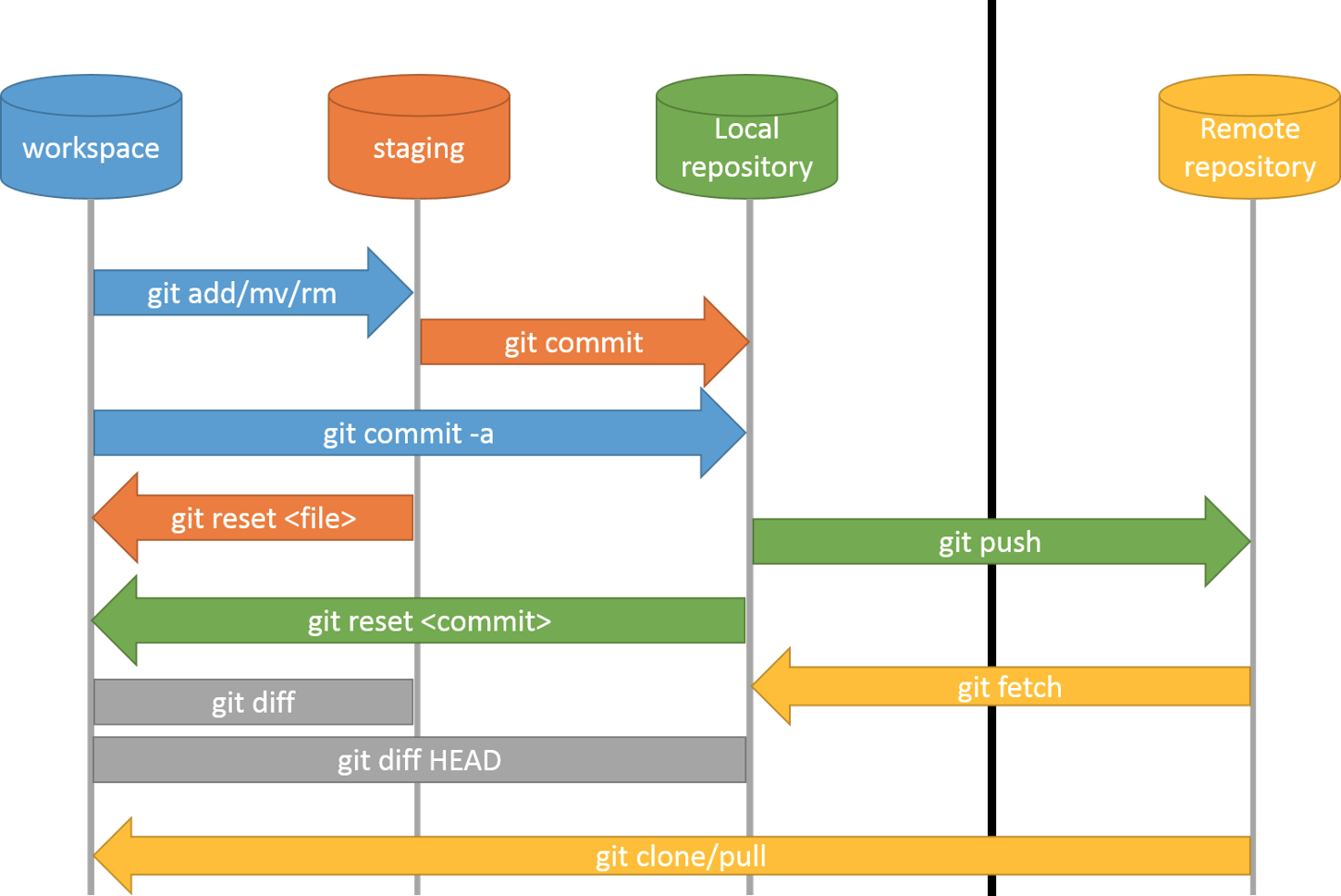
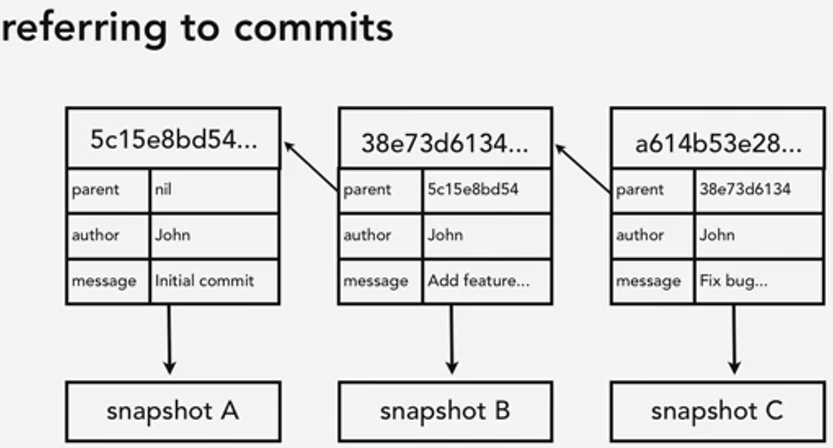
What is Git?  
Git is software that keeps track of changes that you make to files and directories, and it is especially good in keeping track of text changes that you make. Git keeps track of files or directory changes in different versions for you, and it allows you to move back and forth between different versions, to compare the different versions and to see what changed between each one.

All Version Control Systems (VCS) that have ever been created had one single primary purpose in mind when they were created, and that was for managing source code. So we call them Source Code Management Tools or SCM for short.  
  
**SCM History**  
1. Source Code Control System (SCCS) – 1972, closed source, free with Unix (only available in Unix). SCCS stored the original file and then kept track of all the changes that went after it. With SCCS if we wanted the current document, and there were 20 sets of changes, we had to pull up the original and then apply 20 sets of changes to get there.  
  
2. Revision Control System (RCS) – 1982, open source, cross platform. RCS flipped that around so that it kept the most recent file in its whole form, and if you wanted previous versions, then you applied the change *snapshots* to go backwards in time.  
  
SCCS and RCS only allowed you to work with an individual file one at a time. So you could track changes in a single file, but not in sets of files, or in a whole project.  
  
3. Concurrent Versions System (CVS) – 1986-1990, open source, cross platform. CVS allows you to track changes in a whole project. Now the real innovation with CVS is the concurrent part, the fact that we can have a place where we store our code, called a *Code Repository*, you can put that on a *remote server*, and more than one user can work on the same file at the same time, they can work concurrently.  
  
SVN can track the history of directories. For examples, CVS had a hard time if you renamed a file. SVN though would track that change with no problem. If you add a file directory, remove a file, rename it, it's watching the directory as a whole to see what happens and taking a snapshot of that, whereas CVS was just looking at a collection of individually named files.  
  
4. Apache Subversion (SVN) – 2000, open source, cross platform. SVN allows saving of non-text files, like images, whereas CVS couldn't do that. Most importantly, the big innovation for SVN was that it was tracking not just changes to files, or to groups of files, but actually watching what happened in a directory as a whole, watching the files in the directory and actually taking a snapshot of the directory not just the files.  
  
5. BitKeeper SCM – It is a closed source proprietary Source Control Management (SCM) tool; one important feature is its Distributed Version Control. The community version of BitKeeper SCM was free and was used for SCM of the Linux kernel from 2002 to 2005. In April 2005 the community version of BitKeeper SCM stopped being free; so they needed an alternative for managing source code. Therefore, Git was born April 2005, created by Linus Torvalds. Its open source and cross platform. Git is also a Distributed Version Control.  
  
**Distributed Version Control** – different users --or teams of users--each maintain their own repositories instead of working from a central repository. And the changes are stored as change sets or patches, and *we're focused on tracking changes not the versions of the document*. Git really focuses on these change sets in encapsulating a change set as a discrete unit and then those change sets can be exchanged between repositories. We're not trying to keep up-to-date with the latest version of something instead the question is, do we have a change set applied or not? So there is no single master repository, there is just many working copies each with their own combination of change sets.  
  
Now *by convention*, we often do designate a repository as being *a master repository*, but that's not built-in to Git, it's not part of the Git architecture, that's just convention, that we say, okay, this is the master one and everyone is going to submit their changes to the master one, we're going to trying all stay in sync from that one or we don't have to  
  
Configuring Git  
There’s three places that Git stores configuration information.

1. The first largest is System level configuration, that is configurations that ought to apply to every user of this computer. Now each user, of course, can overwrite it with their own, but these are going to be default configurations. On Unix the System level configuration file is going to be in the etc directory, in a file called ‘gitconfig’. It's also going to be in the same file, inside the same folder on Windows, but it will be stored in a different place. Most likely it will be inside your program files, inside the application for Git.  
Unix: /etc/gitconfig  
Windows: C:\Program Files\Git\etc\gitconfig

2. The second is the *User level configuration*, the most useful place to store configurations; these configurations are going to apply to a single user. On Unix, that's going to be in your home directory, inside a file called ‘.gitconfig’.  
Unix: ~/.gitconfig  
Windows: $HOME\.gitconfig

3. And then the third place that we can store configurations is on a project-by-project basis. So in a single project we can have configurations that apply only to that project. Now most configurations, you are probably don't want to use from project-to-project, and you want to put them in the User configuration. But if there is something specific to a single project you can put it inside of the project, look for a folder inside there called .git, and then inside there we have file called config.  
my\_project/.git/config  
Example 🡪 C:\<path>\<app-project>\.git\config  
  
Git gives us some commands that we can use to make editing these configurations easy. For all three of them, it's going to be git config, followed by a modifier that tells at what level we want to do the configuration, and then followed by the configuration itself that we want to do. So if we want to do a system-wide configuration, then it's --system at the end, if it's User level then that's --global, don't let that throw you, global doesn't mean system, it means global to the user, and then if we don't have any modifier then it's just on a single project basis.  
\* System  
 git config –system  
\* User  
 git config --global  
\*Project  
 git config  
  
‘git help <command>’ will give you more information on a specific git command, how to use it and options. ‘man git-log’ will give you the same thing.  
  
To initialize a git project, create a directory for your git project and using your command line or terminal go to this directory and run this git command ‘git init’. The hidden directory .git was added, that’s where git is actually going to do all of its storing and tracking; this directory is a directory where Git stores all of its tracking info. 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. If we remove the .git directory, then suddenly Git is no longer tracking our project. Let Git manage the files inside the .git director and let git put things where they need to be. The only exception to that would be the config file, that's the only thing that you would really ever come in here and either take a look at or even possibly edit.   
  
Commit message best practices  
\* short single-line summary (less than 50 characters)  
\* optionally followed by a blank line and a more complete description  
\* keep each line to less than 72 characters  
\* write commit messages in present tense, not past tense  
 - “fix bug” or “fixes bug,” not “fixed bug”  
\* bullet points are usually asterisks or hyphens  
\* can add “ticket tracking numbers” from bugs or support requests  
\* can develop shorthand for your organization  
 - “[css,js] ”  
 - “Bugfix: “  
 - “#38405 – “  
\* be clear and descriptive  
 - Bad: “Fix typo”  
 - Good: “Add missing > in project section of HTML”  
 - Bad: “Update login code”  
 - Good: “Change user authentication to sue Blowfish”  
  
Viewing the commit log by executing the ‘git log’ command will show us the log of commits that have taken place  
The commit ID is the unique identifier for the commit. The Author information was pulled from the global [User] configuration.  
  
Git Concepts and Architecture  
Three-tree architecture  
Tree 1 – the working copy (workspace).  
Tree 2 – the staging index.  
Tree 3 – the [local] repository.  
  
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.  
  
The other five files are still saved in my working tree, but they never got added to the staging index or to the repository. They are sitting there waiting for me to make another commit, to stage those changes and then commit them to the repository. And of course we can pull things out of the repository in the same way. It's possible to pull them from the repository to the staging index, from the staging index to the working directory, usually that's not what we do. Usually we go ahead and pull them straight from the repository down to the working directory. And in the process the staging index will be updated too. We have our working copy, where we have our changes that we've made, and we've saved, and saved to our hard drive, but we have not yet committed them to the repository, we haven't told Git to make this a changed set and to track it.

Then we have the staging index, which is where we prepare things, we stage them for the commit, and then after they've been staged, we commit them to the repository so that they are permanently tracked and they now have a commit message attached to them.   
  
  
When we submit [git commit] the changes from staging to [local] repo, at that point Git generates a checksum, using hash values (SHA-1), for each changed set. A checksum is a number that’s generated by taking data and feeding it into an algorithm, so checksum algorithm converts data into a simple number, and we call that simple number a checksum (The commit ID is the unique identifier for the commit). The same data put into the algorithm always equals the same checksum coming out that's important because if we change the data going in we get a different checksum out. So one of the most common uses for checksums in computers is to make sure that the data didn't change, if the data changed well then the checksum will be different. And this data integrity is fundamentally built into Git that's very different from other version control systems, they don't use checksums to validate that the data hasn't change. Git does it makes sure that you can't change what's in a commit or else you'll change the checksum that comes out of it. Changing the data changes the checksum.  
  
The number that the algorithm generates is always going to be a 40 character hexadecimal string. Hexadecimal means they can have the numbers 0 through 9 and the letters a through f. So an example might look something like this 5c15e8bd540 and so on, 40 characters long made up of those characters.  
  
It is a number that will be unique to the changes that are in this commit. So the way they get actually attaches that information is that if we have those three snapshot (A, B and C) those sets of changes it feeds them into its algorithm to come up with the S-H-A value, and then it attaches a bit of meta information to each one of those snapshots, it has that commit number at the top, it has the parent commit the commit that comes before it, the author of the commit, and then the commit message.  
  
So here 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.   
  
The HEAD pointer in GIT is a reference variable that references or points to a specific commit in the [local] repository; as we make new commits the pointer is going to change or move to point to a new commit. Therefore, HEAD always points to the “tip” of the current branch in our [local] repo; it’s where the next commit is going to take place. It’s the place where we left off in our [local] repo for the things that we’ve committed.  
Now in Git we have the ability to create new branches that is to create a new set of code that we're working, and it's separate from our master branch, this new branch is going to start having its own commits that are separate from the master branch commits. When we make the first commit HEAD moves to that commit, and then we make another commit HEAD moves along that and along that. Now we can switch between these branches we can check out one or the other, so if we've have our new branch checked out well then HEAD is pointing at the last commit of the current branch, new branch. If we check out our master branch and switch back there. Well guess what? HEAD moves to point to the tip of the current branch that we have checked out their, master. And if we were to check out new branch again it would switch back and forth. So HEAD always points to the tip of the currently checked out branch from the repository.  
  
Git uses the HEAD file to know where the HEAD pointer is pointing to.  
my\_project/.git/HEAD  
Example 🡪 C:\<path>\<app-project>\.git\HEAD

ref: refs/heads/feature/svn\_bb  
You’ll notice that inside the HEAD file it says it refers to refs/heads/feature/svn\_bb, it doesn’t point to a current commit, it points to a current branch, but that current branch has a tip it has a latest one, and that is maintained in this refs folder. So it’s pointing into this folder, inside the svn\_bb file you’ll see that it points to the current commit ID 🡪 91c7dcaa9d9ce536fa0ff607e863fcaa12cceaf8.  
my\_project/.git/refs/heads/feature/svn\_bb  
Example 🡪 C:\<path>\<app-project>\.git\refs\heads\feature\svn\_bb  
  
*Viewing changes or differences between versions of the same file with the ‘git diff’ command*. It compares what’s in the [local] repo, the version that HEAD is pointing at, versus what’s in our working directory [workspace]. So the version in the [local] repository is the one with the minuses, the one that has the pluses is going to be what’s in the new version.  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git diff

diff --git a/pom.xml b/pom.xml

index 0502317..59057d5 100644

**--- a/pom.xml**

**+++ b/pom.xml**

@@ -1,7 +1,8 @@

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

-

+ ^M

***+ <!-- Comments -->^***M

<parent>

<groupId>com.<company>.ccc</groupId>

<artifactId>ccc-parent</artifactId>

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

It is actually comparing the [local] repo against the staging index to tell you or just report the changes that are in our working directory [workspace], which have are not track because they have not been added to the staging index. So it's things that are unique, things that are different about the working directory [workspace] only.  
  
To compare the [local] repo against the staging index to tell you or just report the changes that are in our staging index, which is staged, run the ‘get diff –staged’ command.  
  
How to delete tracked files in the [local] repo?  
1. Manually delete a file from your git project directory. When you run ‘git status’, Git will let you know that the file was deleted from the working directory [workspace] because Git was tracking that file and it was in the [local] repo. To tell Git to remove the deleted file from the [local] repo, you first have to stage the remove intent in the staging index by running the ‘git rm <filename>’ command. Then run the ‘git commit –m “Deleted first file”’ command.

2. An easier way is to remove the file using Git by executing ‘git rm <filename>’ command. It completely deletes the file (it does not go to your trash bin). Then run the ‘git commit –m “Deleted first file”’ command.

How to rename and move tracked files in the [local] repo?  
1. Manually rename a file from your git project directory. When you run ‘git status’, Git tells us that the old filename was deleted and the new filename is a new file in the working directory [workspace]. To help Git figure out that the filename was rename, you stage the rename filename with ‘git rm <old-filename>’ and the new filename with ‘git add <new-filename>’.

2. An easier way is to rename the file using Git by executing ‘git mv <old-filename> <new-filename>’. This command also adds the change to the staging area.  
  
  
Undoing Git Changes  
<> How we can undo changes done in the working directory (workspace); for instance, making changes to a file, saving the changes, closing the file and then when testing the changes not getting the results you expected. That changed was a mistake. So how do we undo the changes? If we go to Git, and we say ‘git status’, it shows us that the file has been modified. If we do “git diff”, it comes up and it tells us what the change was.  
So what we want is we want the [local] repository version back. We want the version that Git has saved for us to be restored. Git is using the version of the file we want in the [local] repo to compare it with the version of the file in the working directory (workspace) to compare when it does the diff. For more details reference page 6 above. Therefore, we are going to ask Git to get the version we want back from the [local] repo, so to check it out for me and replace what I have in my working directory with it. And to do that, we use the ‘git checkout’ command. You could say ‘git checkout <filename>’. However, the thing about checkout is that it’s used for more than one purpose. It’s also used for working with branches. Because what checkout does is go to the repository, get the named thing that I've gave you, 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 and space, followed by filename that says 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.  
  
<> How we can undo changes done in the staging index or in other words to un-stage things that we’ve staged there. What we want to do is un-stage changes without losing the changes in our working directory (workspace). To un-stage use ‘git reset HEAD <filename>’ command; what we’re telling it is go look at the HEAD pointer. The HEAD pointer points to the last commit of the tip of the current branch. Go look at that last commit and reset yourself to be the same as what that has. Here we’re resetting the index to be the same as that.  
  
<> How we can undo changes done in the [local] repo, undoing commits that we’ve made? Due to the built-in security and data integrity of Git with the commit ID (SHA-1, see page 5 above), it’s only possible to change the last commit, because nothing (previous commits) depends on it yet. So the most recent commit, the commit that HEAD points to we do have the ability to edit. 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 ‘git commit –amend’ command. Every time you execute ‘git commit –amend’ command a new commit ID is generated. Furthermore, you can use the “git commit –amend –m ‘Change commit message’” command to change the commit message.  
  
<> To retrieve older version of files committed into the [local] repo using Git you use the commit ID that has the version previous to the commit changes to retrieve them. You do that by executing ‘git checkout <commit0-ID> -- <filename>’. This moves the old version of the file from the [local] repo into the staging area. Now you can commit this revision file back into the [local] repo or you could bring it down to the working directory (workspace) to continue your work from this version of the file. But Git also gives us a helpful way when we really want to just undo the changes of a commit completely and totally, we can use the ‘git revert’ command.  
  
What the ‘git revert’ command will do is it will take all of the changes that were there, and it will flip them around. It will do the exact opposite of those changes. So, anything that was added will be deleted, anything that was deleted will be added back in again, and anything that was modified will be changed back to its previous state. It's going to be a complete mirror image of this commit. We have to provide the commit ID we want to revert to in the ‘git revert <commit-id>’ command. Remember, we don't need the whole thing. Now, this is going to do it all in one step for us. It's not going to assemble the commit and then wait for us, it's going to go ahead and make that commit. It's going to pop up with the chance for me to edit the commit message, and it's going to do that in a text editor. So it pops up, and it says, all right, if you want to edit this message before I commit it, go ahead and do it now. It went ahead and made the complete commit. So, if I do git status, you can see my working directory is still clean, but git log, and you can see that it did make this new commit here which reverts that old commit. Now, you can pass in the -in option with revert, and then it won't actually do the commit, it will just stage it and then wait for you to actually do the commit yourself. If you want to revert changes that are really complicated, then use merging.

<> Using reset to undo multiple commits. The ‘git reset’ command allows us to specify where the HEAD pointer should point to. Normally, we just let Git manage the HEAD pointer for us. We make a commit and Git moves the HEAD pointer to point to that commit, we make another commit, and it moves the HEAD pointer to point to that commit. Well, here, 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." The ‘git reset’ command says let's rewind back to our previous commit, and that's where we're going to start recording from now on, and we're going to just overwrite whatever came after that. Git reset always moves the HEAD pointer. That's one thing that it does in every case. But there are three different options that the ‘git reset’ command can use to control some of the other behaviors that it has, and those options are going to be soft, mixed, and hard.  
  
*Soft* is going to move the HEAD pointer to the specified commit, and it's not going to change the staging index, or the working directory at the same time. It's just going to move the pointer. It's the safest of all these options, that's why it's called soft. Just a soft reset, move the pointer and do nothing else. 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 [local] 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. git reset --soft <commit-id>  
  
*Mixed* reset is in between soft and hard, 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 [local] repository. It does not change your working directory though. So, at this point, the staging index and the [local] 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 [local] repository are still in our working directory (workspace). We haven't lost any work. It's just waiting for us to stage it and then commit it. git reset --mixed <commit-id>  
  
A *hard* reset will not only move the pointer of the [local] repository, but it will make your staging index and your working directory (workspace) match that as well. So that means any changes that came after that commit are completely obliterated. They don't exist in the [local] 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. git reset --hard <commit-id>  
  
I said that it throws everything away, that's not entirely true, it doesn't throw everything away immediately, it's just not sitting here waiting for us to make those commits, it's not at hand. However, those old commits are still there until they get garbage collected, we can still move our HEAD back to this later point in time. So let's do that, git reset --hard, and we'll put in the later commit ID again, it now moved it up there, that git object was still there, it's still sitting there in the git folders, it had just moved the HEAD pointer away from it.

At some point if we hadn't done anything with it, it would have gotten garbage collected and thrown away, that would have been a long ways down the line. It would have hung on to it for a while just to make sure that we didn't want to go back to it. But it wouldn't have been easy to go back to it if we didn't record or saved the git log or commit history with the commit IDs. We wouldn't have those changes in our working directory, and we wouldn't know the name of the commit that would take us back there to get to it. So unlike the other two examples that we did with reset, let's go ahead and take the additional step of making a new commit here.

<> Removing untracked (un-stage) files from our working directory (workspace). The idea is that if we 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, is there an easy way that we can do that? We certainly could go through and just delete them one by one, but Git gives us a quick and easy way to just tell all those files that should be thrown away. So specially if you have things like log files, zipped directories, compiled code, and things like that that you don't want in your Repo, and you'd like to get them out of your working directory too.

We can get rid all of those by using the ‘git clean’ command. What we want to do is get rid of these files we don’t need. And the way that we just say throw them away is with the ‘git clean’ command, and git clean on its own won't actually do anything. They will come and say you know what I need either a -n or a -f option. -n is a test run, so let's try that first, ‘git clean –n’, and it comes up and it says would not remove the folder explorers. Don't worry about that, we'll talk more about directories in the next section below (ignoring files). It says that it would remove junk1.txt, junk2, and junk3.txt, so it tells us the files that it would remove. That's what the -n option does.  
  
The ‘git clean –f’ command forces it to run, and essentially 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 [local] repository. However, if we have something in our staging directory, junk1.txt like that, now if we run git clean -f, it did remove junk2 and junk3, but it did not remove junk1.txt, because junk1.txt is in our staging index, not in our working directory, so it didn't get thrown out. git reset HEAD on junk1.txt, now if we do git clean with the -f option, it throws it away as well, git status is empty. And we take a look, and we can see that all three of those files have been thrown away. So git clean is pretty simple, especially compared with all the work that we were doing with different kinds of resets. It has one simple purpose, which is just to throw away the untracked files.

Now be careful when using it, because of course it does destructively throw away these files. They're not stored in our [local] repo, they're not stored in our staging index. If we throw them out of the working directory, they're not stored there either, and they're not stored in your trashcan, they're just going to be permanently deleted.  
  
How to configure Git to ignore certain files  
<> The .gitignore file tells Git which files it ought to ignore at the Git project level (in the root of the working directory); therefore, to do that in your Git project folder create a .gitignore file (my\_project/.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 use some very basic regular expressions. We can use the Asterisk, the Question Mark, and 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.

Very basic regular expressions: \* ? [aeiou] [0-9]

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. Don't let it confuse you this sort of a double-negative, we're talking about not ignoring things, that means these would be tracked. So files ending in .php don't get tracked, but index.php does get tracked.

Negate expressions with !  
 \*.php  
 !index.php

And you can tell to ignore all files in a directory by just having a trailing slash at the end, and that will tell that all files in this directory should be ignored. Example: assets/videos/. This will ignore all files in the videos directory.

If you want to add comments to the file, you can start those with the Pound or Hash sign at the beginning, and blank lines will just be basically skipped. You can simply open up your Text Editor and create the .gitignore file that way. But I think because it has a dot in front of it which makes it hard to see anywhere except from command line, it's best to create the file from the command line. So I'm going to use a Unix command vi or vim which will lets you create a file. Tracked/commit the .gitignore file to your [local] repo to keep as part of the project.  
<> Understanding what to ignore  
\* compiled source code  
\* packages and compressed files  
\* logs and databases  
\* operating system generated files  
\* user-uploaded assets (images, PDFs, videos, etc.)  
\* Check out the following [GitHub article for ignoring files](https://help.github.com/articles/ignoring-files/).  
\* Check out the following [GitHub gitignore repo](https://github.com/github/gitignore).   
  
<> Ignoring files globally  
To configure Git to globally ignore some files, that means for Git to ignore those files in all [local] repos, not just in any one specific [local] repo you do 'git config’ with the global option and then core.excludesfile, and we tell it where the file is. So the filename can be named anything we want, we can locate it wherever we want, we just have to tell Git in its config file, hey, this is the file you should use for globally ignoring things.

The .gitignore\_global file should be listed in the .gitconfig file  
Unix: ~/.gitconfig  
Windows: $HOME\.gitconfig

git config --global core.excludesfile <full-path>/.gitignore\_global  
  
<> Ignoring tracked files, that is files that Git has already started tracking, we want to ignore them after the fact. When we were looking at .gitignore previously, we were looking mainly at ignoring new files. So a new file is out of the [local] repo, we don't want to track it, so we tell Git that it ought to ignore it rather than continue to show it to us. But you should note that Git will not ignore a file that was already tracked before a rule was added to the .gitignore file, telling it that it ought to ignore it. If that’s the case Git will still try to keep track of the file, because it’s a tracked file, so Git still sees it, because Git still keeps track of tracked files. Therefore, to tell Git to stop tracking the after the fact ignore files, the file must first be untracked.  
  
One way we could do that is just to remove it, right? Git remove that would do it, git remove app.log. It would remove it from the [local] repository, and it would remove our version, it would do both. What if we didn't want to remove our version? What if this was a log file? Or what if it's a set of images or something or Photoshop files that we want, and we want to keep on ours but we want to tell it, you know what, stop tracking it? We may even want to leave it in the [local] repository so that it stays in the [local] repository for other people to download. We just want to ignore changes that happened to it after that, right? That certainly might be the case with like a log file or something. We want to have a placeholder for the log file that everyone can have, but we don't necessarily want to have changes to that log file be tracked. Well, what we want to do here instead of just remove is a remove with the cached option.

git rm --cached <filename>

So --cached, that's going to tell it to remove this file from the staging index, not from the [local] repository, just from the staging index. That will cause the file to stop being tracked. It will still leave the copy in the [local] repo, it will still leave the copy in my working directory. It's just going to take it out of the index. If you do ‘git status’, Git will show the <filename> as deleted. Next add or tracked the changes made to .gitignore and then commit to the [local] repo .gitignore and <filename> by executing git commit –m “Remove <filename> from staging index”. Now if you do git status, we will see that the working directory (workspace) is clean, but the <filename> is still listed in the working directory. Git did not get rid of the <filename>, but Git is now ignoring the file for any changes made in the working directory.  
  
So if you want Git to ignore files that are already being tracked, but you don't want to remove them completely, then what you need to do is not only ignore it, but tell Git that it ought to also remove them from the staging index or the cache.  
<> Tracking empty directories. By default, Git does not track empty directories, because *Git is designed to be a file-tracking system*. Its purpose is to track files and the content in those files. Therefore, Git tracks files, and Git tracks the directories it takes to get to those files, but Git ignores directories that have not files at all. So the trick that we use in order to keep track of empty directories is to put a file in them and by convention people either name the file .gitignore to match the .gitignore file or more often now they use **.gitkeep** (an empty file or with a comment), the opposite, basically telling it, it should keep this directory. One way to quickly create an empty file is by using the touch Unix command. So touch is a way to just create a file that doesn’t exist. Track and commit the .gitkeep file so the directory is always tracked by Git.

touch assets/pdfs/.gitkeep  
git add assets/pdfs/  
git commit –m “Add ‘empty’ directory with .gitkeep file in it”  
  
  
Navigating the Commit Tree (local repo)  
<> Referencing commits in Git. Tree-ish is the structure of files in the Git [local] repository. It’s similar to a directory in your file system. In Git, tree-ish means something that references part of the tree. It’s ish because what that something is can vary widely. Now in its simplest terms, a tree-ish is a reference to a commit because that commit then in turn references the tree, the Git [local] repository and all the files that are in there at that point. So if you have a hard time thinking about all the things that a tree-ish can be, the simplest version is that it's just something that points out a commit.  
  
The following are ways you can tree-ish (reference a commit) in Git:  
1. Full SHA-1 hash: The easiest way to do it is to use the entire 40 character string commit ID or checksum.  
2. Short SHA-1 hash: At least 4 characters, for large projects or unambiguous use 8-10 characters of the commit ID.  
3. HEAD pointer: The HEAD pointer remember always points to the commit that’s at the tip of the currently checked out branch (the branch in use by Git). So that's going to be a tree-ish that points to part of the tree.  
4. Branch reference: If we use a branch reference, then we are referring to the tip of the branch. Now it doesn't have to be the currently checked out branch, it can be a branch that's not checked out.  
5. Tag reference.  
6. Ancestry: We can refer to commits is by using any one of these methods and then referring to that object’s ancestry, that is, refer to the parent commit or great-great grandparent commit of something.  
  
So if we want to refer to the parent commit of something, we first provide the reference for what we want to focus on, and then we say, find its parent by using the caret. So that caret character ^ comes right after it, you can think of it is pointing up, pointing up to the parent on the tree. If you think of like a genealogy chart where the grandparents at the top and the children come down, we are moving up. So we are moving up to the parent, and that's what the up arrow nature of the caret suggest to us. We can also use a tilde notation ~ that is to have a reference to the commit and then a tilde followed by the number of generations we want to go up. So HEAD going back one, we can also leave off the one, and it's just assumed. Most people tend to use the caret in this case instead of the tilde, but they do the same thing.  
  
So HEAD going back one, we can also leave off the one, and it's just assumed. Most people tend to use the caret in this case instead of the tilde, but they do the same thing. If our HEAD commit is acf87504 then all five of these, all refer to the same thing. They all refer to the parent commit of acf87504. Now in addition to the parent commit, we can refer to the grandparent commit. Of course, if we want the parent of the parent, well then we just provide two up arrows. We say start at the HEAD and go up twice, that will give us the grandparent. This is where the tilde notation becomes a lot more useful because now we can say HEAD~2, and it's especially apparent when we have the great grandparent commit because then we start having a whole lot of caret characters coming after our commits, whereas the tilde notation is much more compact and clear as to exactly what we were doing. We are going from the HEAD, we are moving back three commits to the great grandparent. So these are some useful ways that we can refer to commits that are in the commit tree.

* Parent commit
  + Caret notation: HEAD^, <commit-id>^, <branch-name>^
  + Tilde notation: HEAD~1 or HEAD~, <commit-id>~1 or <commit-id>~, <branch >~1 or <branch-name>~
* Grandparent commit
  + Caret notation: HEAD^^, <commit-id>^^, <branch-name>^^
  + Tilde notation: HEAD~2, <commit-id>~2, <branch-name>~2
* Great-grandparent commit
  + Caret notation: HEAD^^^, <commit-id>^^^, <branch-name>^^^
  + Tilde notation: HEAD~3, <commit-id>~3, <branch-name>~3

<> Exploring tree listings using tree-ish. To list out the contents of a tree object, we use the ‘git ls-tree <tree-ish>’ command. The following are ways you can tree-ish (reference a commit) in Git:  
1. Full SHA-1 hash  
2. Short SHA-1 hash  
  
3. HEAD pointer  
git ls-tree HEAD  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git ls-tree HEAD

100644 blob 591d37f695cd6a02a1d028b4d37ce51e8fede394 Jenkinsfile

100644 blob b64050d97807163d24a670518d6a428d57e3410c README.md

100644 blob 0502317c6082916eb4e59a7b8eb74a4ee5659530 pom.xml

100644 blob 6dc164a7dddeda827dd3b12e22cecaf9e13969e3 sonar-project.properties

040000 tree 293b24bb1d3d0c1a8df8775674e31c2866149f32 src

4. Branch reference  
git ls-tree <branch-name>  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git ls-tree feature/svn\_bb

100644 blob 591d37f695cd6a02a1d028b4d37ce51e8fede394 Jenkinsfile

100644 blob b64050d97807163d24a670518d6a428d57e3410c README.md

100644 blob 0502317c6082916eb4e59a7b8eb74a4ee5659530 pom.xml

100644 blob 6dc164a7dddeda827dd3b12e22cecaf9e13969e3 sonar-project.properties

040000 tree 293b24bb1d3d0c1a8df8775674e31c2866149f32 src

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git ls-tree feature/svn\_bb src/

040000 tree 3402117e55731acbe047fc433992048fd48a897a src/main

040000 tree 6fde15dcf3a744406e0c4a5e0e374dd6e2bc6fff src/test

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$  
  
5. Tag reference  
6. Ancestry  
git ls-tree <branch-name>~12

git ls-tree <branch-name>~11 <directory>  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git ls-tree feature/svn\_bb~10

100644 blob 90fab61f8a6057b6bee158cd4e07060fd411a001 Jenkinsfile

100644 blob b64050d97807163d24a670518d6a428d57e3410c README.md

100644 blob b7610767d09f08b6b933925121c8fea3a9ab62f1 pom.xml

100644 blob a7e18929c21fc9e97874060808d148734d40e894 sonar-project.properties

040000 tree e1560623298db9ea793833f2a12527b71f7fa5f9 src

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git ls-tree feature/svn\_bb~11

100644 blob 90fab61f8a6057b6bee158cd4e07060fd411a001 Jenkinsfile

100644 blob b64050d97807163d24a670518d6a428d57e3410c README.md

100644 blob b7610767d09f08b6b933925121c8fea3a9ab62f1 pom.xml

040000 tree e1560623298db9ea793833f2a12527b71f7fa5f9 src

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git ls-tree feature/svn\_bb~11 src/

040000 tree 0eeab16a6d55c9bcf9484c7eeaa2ef3093f277f3 src/main

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git ls-tree feature/svn\_bb~12

100644 blob 60a49d317920e6a19ea829cff001d71c75156480 Jenkinsfile

100644 blob b64050d97807163d24a670518d6a428d57e3410c README.md

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$

<> Getting more from the commit log to really find-tune exactly what you are looking for and tease out information about the commits using all of the available git log configuration options.  
  
git log --oneline  
It's probably the single most useful option, because it just gives us a oneline list of what's in our log file, instead of having that long scrolling list, it just compresses it for us. It still gives us part of the commit ID (SHA-1 or checksum) here that we can use to reference each one those commits if we want to work with it.  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git log --oneline

91c7dca (HEAD -> feature/svn\_bb, origin/feature/svn\_bb) Saturday 9th December release.

5589206 Added install parameters

a694c1b Changed to 4.3 web-domain version

f85aea2 Removed snapshot for web-domain dependency

7228e6d Sonar changes

ff5f1a5 group ID change

e0f4cc4 Sonar changes

1290466 Changed groupId for web\_domain dependency

ae17470 Added Unit test

341418f Updated to 4.2 SNAPSHOT

9dc246d Added sonar changes

60aa0ab Initial checkin

2e6c817 (origin/master, origin/HEAD) initial commit

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$

We can also use a number of commits to limit the number of commits that it goes backwards, so, for example, -3 will show us just three commits, -5 will show us five commits.  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git log --oneline -3

91c7dca (HEAD -> feature/svn\_bb, origin/feature/svn\_bb) Saturday 9th December release.

5589206 Added install parameters

a694c1b Changed to 4.3 web-domain version

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git log --oneline -5

91c7dca (HEAD -> feature/svn\_bb, origin/feature/svn\_bb) Saturday 9th December release.

5589206 Added install parameters

a694c1b Changed to 4.3 web-domain version

f85aea2 Removed snapshot for web-domain dependency

7228e6d Sonar changes

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$

git log --since=”CCYY-MM-DD”  
We can also filter the log by time period. So for example, git log, and then we can use the since option, since equals and then in quotes, and we can put a lot of things in here for the time. For example, 2017-11-20, and it understands that as being 2017 November 20th. Hit Return, and that's what it gives us, just the entries that are since then.  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git log --since="2017-11-20"

commit 91c7dcaa9d9ce536fa0ff607e863fcaa12cceaf8 (HEAD -> feature/svn\_bb, origin/feature/svn\_bb)

Author: <Lastname>, <Firstname> <uid@<company>.com>

Date: Mon Dec 18 15:49:53 2017 -0600

Saturday 9th December release.

commit 55892068cb3867b0cdf3d4430f0b539b983d8747

Author: <Lastname>, <Firstname> <uid@<company>.com>

Date: Fri Dec 8 14:14:40 2017 -0500

Added install parameters

commit a694c1b2467eef5130db0b256a8b949eb8770942

Author: <Lastname>, <Firstname> <uid@<company>.com>

Date: Tue Dec 5 15:47:18 2017 -0500

Changed to 4.3 web-domain version

commit f85aea24cb11814e1de465d60b2a1825dfe6b034

Author: <Lastname>, <Firstname> <uid@<company>.com>

Date: Tue Nov 28 14:16:07 2017 -0500

Removed snapshot for web-domain dependency

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$

git log --until=”CCYY-MM-DD”  
We can always use after instead of since, those are synonymous. We can do the same thing, but use until so that's all the commits that are until that date, and we can use before as well.  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git log --until="2017-11-10"

commit 9dc246d60cd09ba372352b050b4e2b7cb2955cb0

Author: <Lastname>, <Firstname> <uid@<company>.com>

Date: Thu Nov 9 16:17:35 2017 -0500

Added sonar changes

commit 60aa0ab42a5ca023c7274c3a004f672a51ae4d49

Author: <Lastname>, <Firstname> <uid@<company>.com>

Date: Thu Nov 9 15:25:19 2017 -0500

Initial checkin

commit 2e6c817930398664e306cb41386030792d2ee9a7 (origin/master, origin/HEAD)

Author: nbk1qeu <nbk1qeu@F6C3BE51214AD>

Date: Tue Nov 7 16:04:28 2017 -0500

initial commit

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$

git log --since=2.weeks --until=3.days  
git log --since=”2 weeks ago” --until=”3 days ago”  
We can pass in the time period in other formats as well and Git is able to figure this out, like 2 weeks ago and until 3 days ago. Git gives you a lot of flexibility in the way that you specify the time period that you want to use.  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git log --since="4 weeks ago" --until="3 days ago"

commit 91c7dcaa9d9ce536fa0ff607e863fcaa12cceaf8 (HEAD -> feature/svn\_bb, origin/feature/svn\_bb)

Author: <Lastname>, <Firstname> <uid@<company>.com>

Date: Mon Dec 18 15:49:53 2017 -0600

Saturday 9th December release.

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$  
  
git log --author=”Gibran”  
We can also search by author, so author is the person that made the commits. I can say I’m looking for all the commits made by Gibran. If there are multiple Gibrans, then we’ll see all the commits done by all the different developers that have the first name Gibran. Instead if I wanted just Gibran Castillo’s commits, then I would write in the first name and the last name ‘Gibran Castillo’.  
  
git log --grep=”<message-snipe\_or\_regular-expression>”  
We can also GREP the commit messages. GREP is a global regular expression search, so we can do git log --grep equals and then whatever regular expression we want to search for we can put here in quotes. This is really nice if you want to see everything that was committed about a certain topic.  
  
git log <older-short-commit-id>..<newer-short-commit-id> --oneline  
We can also specify a range of time periods using shortened commit ids, the “..” indicate a range between commit ids. We can also ask Git for information about what happens to a particular file. For example, we can ask Git to tell us everything that has happened since initial commit with ‘git log <short-first-commit-id>.. <filename>’. We can find out more details about the commits by using ‘git log -p’, this is the patch option, and it shows us a diff of what actually changed in each one of these. So here’s the additions and here’s the subtractions. So we can really see what’s different about each one. You can use it with ‘git log –p <short-first-commit-id>.. <filename>’.  
  
git log --stat --summary  
We can do something similar also with git log --stat and --summary, and you can use those separately or together. What they'll do is they'll tell you statistics about what changed in each one, so you see here this, the git ignore file added whole bunch of things. Here in resources, something was added and something was taken away, and it goes ahead and gives you a little summary here as well, that's what that summary is. That's nice if you're not as concerned with the actual details of what happened, you are just wanting to get an idea of the quantity of things that changed and where.  
  
git log --format=oneline  
This returns the entire commit id (SHA-1) instead of just a partial commit id (checksum).   
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git log --format=oneline

91c7dcaa9d9ce536fa0ff607e863fcaa12cceaf8 (HEAD -> feature/svn\_bb, origin/feature/svn\_bb) Saturday 9th December release.

55892068cb3867b0cdf3d4430f0b539b983d8747 Added install parameters

a694c1b2467eef5130db0b256a8b949eb8770942 Changed to 4.3 web-domain version

f85aea24cb11814e1de465d60b2a1825dfe6b034 Removed snapshot for web-domain dependency

7228e6d5f2feb992f8eeba28e804337d14c32fc0 Sonar changes

ff5f1a56b6bf6834b1e3c4b86b5468f007c9e6e5 group ID change

e0f4cc4e84bb0e84a11dc74f803e25ff32534f4f Sonar changes

12904661137a337036aeaa8c47eaeb351f900716 Changed groupId for web\_domain dependency

ae17470c46c6f8f25f497d4d936d3b11b9b4011b Added Unit test

341418f90441f6e11ad09d365e41665220f03f9b Updated to 4.2 SNAPSHOT

9dc246d60cd09ba372352b050b4e2b7cb2955cb0 Added sonar changes

60aa0ab42a5ca023c7274c3a004f672a51ae4d49 Initial checkin

2e6c817930398664e306cb41386030792d2ee9a7 (origin/master, origin/HEAD) initial commit

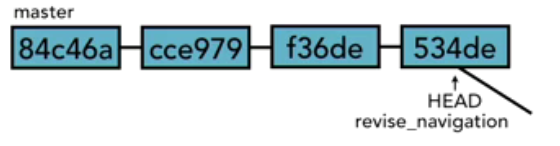
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

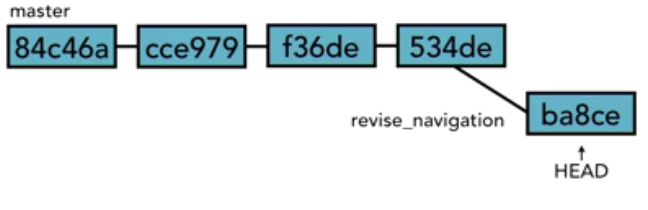
$

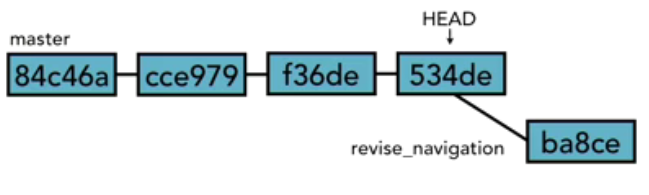
There is other formats like  
git log --format=short  
git log --format=full   
git log --format=fuller   
git log --format=email   
git log --format=raw   
  
git log --graph  
This shows us a graph of each one of our commits. Now right now our graph is pretty straightforward and linear, but if we start having branches, and we start branching thing off and then merging things back in, then this really shows us those branches are merged.  
  
git log --oneline --graph --all --decorate  
This combination gives us a nice compact list. It'll show all the branches that take place and even tells us right now that this is where the HEAD is pointing, and this is the tip of the master branch. So this is a nice combination of options that's worth remembering.   
  
<> Viewing commits with the ‘git show <commit-id>’ command allows you to examine a specific commit and see what was committed. It shows the author, the date, the commit message and then gives you the diff of the commit (the diff between what was there before and what was there after). You can also pass formats like  
git show --format=oneline  
git show --format=oneline HEAD  
git show --format=oneline HEAD~3  
git show --format=oneline <commit-id>  
git show --format=short  
git show --format=full   
git show --format=fuller   
git show --format=email   
git show --format=raw  
  
<> Comparing commits; we’re not actually comparing the commit snapshot, just those changes that were made that are stored with that commit. We're comparing the directory that that commit references; the actual state of all the files in the repository at that point in time. And that makes sense because a commit is not just a snapshot of those changes, but it includes all of the ancestors all the way back to the beginning of the repository as well.  And the sum total of all those ancestors is the directory, or tree, at that point in time where the commit is.  
  
So when we compare commits what we're actually doing is comparing two directories and seeing what has changed between those two directories. Now it might be comparing what's changed over time. For example, I might be comparing a commit that I made on Monday morning with a commit that I made on Friday afternoon, and comparing those two directories will show me all of the changes that have been made during that one week. On branches, we can compare two different branches to see how they differ. What's changed in each of these branches? In order to make these comparisons we're going to use a tool that's already familiar to us, and that is *diff*.

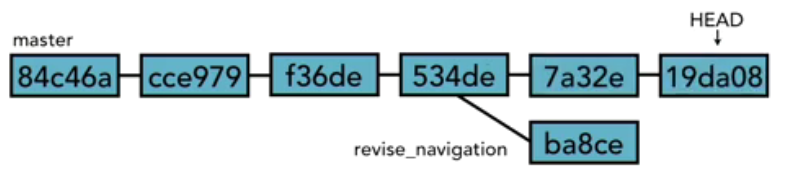
When we used it the first time, we just used it simply by saying git diff. And what that did was it returned the changes that were made between our working directory and the staging index. So it's all the things that could be put into our staging area, changes that we had made recently that were not yet staged. Once we put them into the staging area, they didn't show up in git diff anymore, you'll remember we had to add another option to it, which was staged or cached, and those are synonymous. And what those do is they show us the difference between the staging index and the repository or HEAD, where the HEAD pointer is pointing.

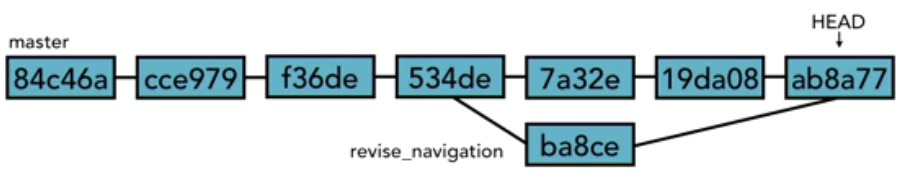
Well, the diff tool is very flexible and allows us to pass in other things as well so that we can compare more than just our working directory, staging index, and [local] repository. If we pass in a commit id (SHA-1) that references a commit, it will show us the difference between our working directory (workspace) and the directory at the point in time that that commit was made. So ‘git diff <short\_commit-id>’ will show you the differences between the directory at that point in time and your current working directory (workspace) and all the different things that have changed, including renaming files and things like that. If you wanted to be more specific, you could pass a filename like this ‘git diff <short\_commit-id> <filename>’. You can also use a range ‘git diff <older-short-commit-id>..<newer-short-commit-id>’ and you pass a filename ‘git diff <older-short-commit-id>..<newer-short-commit-id> <filename>’. You can also do ‘git diff --stat --summary <short-commit-id>..HEAD’. The -b option is the same thing as the longer ignore-space-change. ‘git diff –ignore-space-change <commit-id>..HEAD.  So it will ignore whether or not someone changed one space to two spaces, four spaces to five spaces, it ignores that, that's the b option. The other one that you can use is the w option, which is ignore-all-space. It ignores every single change that could be made to space, it just says forget about it.  
  
  
Branching  
<> Branching overview: Branches are one of the most powerful features in Git, in large part because of how easy they are to use. Getting the most out of Git will mean using branches often and effectively. In Git, branches are cheap: they don't take a lot of processor power, they don't take up a lot of storage space, they are easy to create and delete. Branches also allow you to isolate features or sections of work. This can be especially useful when you're collaborating with others. When we create branches we're still going to have just one working directory (workspace). All the files that we're working with will still be in that same project folder that we've been in before. But when we switch branches, Git is going to do fast context switching. It's going to take all of the files and folders that are in our working directory (workspace) and make them match what's in the branch.  It will swap out the two sets of changes.  
  
So we have our master branch, and we know that the HEAD pointer always points to the last commit in the master branch, the tip of the current branch.  


Once we create a new branch though, that changes. We create the new branch, at that point HEAD still points to commit 534de, both the master and revise\_navigation at this point are exactly the same. There have been no additional commits made, so HEAD points to the same commit.  


It's only once we make new commits to the revise\_navigation branch that then HEAD moves to a new commit, a commit that's not on the master branch anymore.  


Now we can switch back and forth. If we switch back to master branch, well then, the HEAD points to the tip of the current branch, that's the master branch.  


If we switch back to revise\_navigation, it will switch back to point to the tip of the revise\_navigation branch. Again, it's like that play head that's always saying where we're going to start making new commits. So let's say I've got my master branch that I'm working on, that's where HEAD is pointing. I make more commits there.  


Then when I'm finally ready to merge back in to revise\_navigation branch, it creates that merge commit and the HEAD moves to that merge commit.  


<> Viewing and creating branches. The way you can see all the branches in your local repository is by executing the ‘git branch’ command; this will show you a list of all the branches you have on your local repo. To create a new branch you execute ‘git branch <branch-name>’, but the HEAD pointer is still pointing to the current branch you were on before you created the new branch.  
  
<> Switching branches. If you have more than one branch in your local repo and you want to switch from the current branch you have checkout (you are using), that’s what’s in our working directory (workspace), to a different branch then you would execute ‘git checkout <branch-name>’.  
  
<> Creating and switching branches. To create and switch to the new created branch at the same time execute ‘git checkout –b <branch-name>’. The –b option means both created and switch at the same time.  
  
<> Switching branches with uncommitted changes. Git will not allow you to switch branches if the current branch does not have a clean working directory (workspace). Actually, mostly clean. In other words, if your working directory (workspace) of the current branch have modified files that are not staged (untracked), then Git will not allow you to switch branch. Git is telling you, sorry, I can't do this, because if I do, if I make your working directory look exactly like master or whatever branch you’re trying to switch to, you're going to lose the changes that you just made. So rather than destructively just blow out your changes, which you might want to keep, you might not have realized that this was going to be a problem, Git stops and does the safe thing, and it says you need to deal with this problem first. You have three options at this point:  
One, you can scrap the changes by checking out the file again. That's git status git checkout -- and the file name.

The second thing you can do is you can commit the changes to the current branch’s [local] repo. Once they're committed, they won't get lost anymore. They're stored in Git, we can just switch back and forth between the branches, and it will just switch back and forth between the HEAD of each one of those branches.

The third possibility is that we can ***stash*** the changes, and we'll talk about the stash a little later on. Basically it's a little pocket that we can put things in and save them until later. We can put them away to a little area that we can pull them back when we're ready.

Now I said it has to be mostly clean, because it doesn't have to be completely clean. It just has to be clean enough that there are no conflicts.  So that's what I mean when I say mostly clean, we can't have anything that would cause us to lose data if we made the switch.  
  
<> Comparing branches using ‘git diff <branch-name>..<other-branch-name>’; the other in which you put the branch names does not matter, it will give you the same results. Another nice feature that we can use with diff is colorwords option, like this ‘git diff --color-words <other-branch-name>..<branch-name>’.

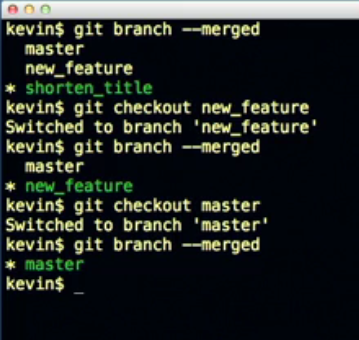
Now of course, what we are passing in here is not just a branch, we are passing in a tree-ish. We talked about tree-ishs.  A tree-ish can be a lot of things, it can even include the ancestors. So for example, I can compare <branch-name> with the previous commit to the HEAD of <other-branch-name>. So go to <other-branch-name>, and it's not the last the commit, it's the commit right before that, that I want to compare it to.

git diff --color-words <branch-name>..<other-branch-name>^  
git diff --color-words <branch-name>..<other-branch-name>~3

There is one more important way that we can compare branches, and that's that we can find out, whether one branch completely contains another branch or not. That is whether or not everything in it has been merged into the current branch. We do that by not using the diff tool but using git branch with the dash dash merged option after it. That will show us all branches that are completely included in this branch.

git branch --merged

So in this case we're on the shorten\_title branch, all of the commits that are in new\_feature are also in shorten\_title, all of the commits that are in master are also in shorten\_title.

What that lets us know is that we can actually delete new feature if we wanted to and shorten\_title wouldn't be affected. Shorten\_title has all of those commits in it. Let's try switching to different branch, git checkout new\_feature, surround it again, git branch merged. Now it says, all right, new feature doesn't have all of the commits that are in shorten\_title. Shorten\_title has some things that are not merged into here yet, we would need to do a merge to get those changes into new feature first. What it actually does is it goes back up the ancestor chain of the new\_feature branch to see does it have the tip of master in it.  


If it has that final commit of master, then it has all of the ancestors as well. It does not have the final commit of shorten\_title so it doesn't list it here. And just for good measure, let's do checkout master, and we will do the same thing there, and you see that master doesn't contain those changes that are in new\_feature and shorten\_title. So well, it's not strictly speaking comparing branches, the same way that diff does, I think it's important way to get some information about the comparison of the content of what is in each of the branches. Those come in very handy soon, when we start trying to delete branches.

<> Renaming branches using the ‘git branch -m <branch-name> <new-branch-name>‘ command, the –m option means move. Or you can use ‘git branch --move <branch-name> <new-branch-name>‘.  
  
<> Deleting branches using the ‘git branch –d <branch-name>’ command, the –d option means delete. Or you can use ‘git branch --delete <branch-name>’. To avoid disasters, deleting a branch did not meant to delete or were not ready to delete yet, Git has a few checks in place to make sure that you don’t do something stupid.

Git check one, *you cannot delete the current branch or the branch you are currently on*, you have to checkout another branch and then delete the branch you were previously on.

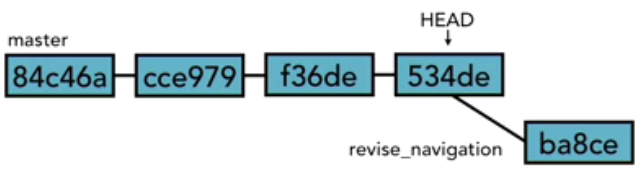
Git check two, if your current branch is the master branch and you are trying to delete a branch that has a changed in the local repo that it is not in the master branch’s local repo, then Git will notify or give you a warning that those changes have not been merged to the master branch. So I'm not comfortable just throwing away those commits. That might not be a smart thing to do. So Git is asking you tell it if you really meant to delete that branch without merging first. If yes, then use capital D. Capital D will tell Git that it's okay to throw away all those changes that are in that branch, but with lowercase d, it's sort of like we have a safety on it.  It's going to check and make sure that we're not losing something. So the -d option, the branch must be fully merged into this branch. With the -D option, then it'll delete the branch irrespective of its merged status.

So now let's use the -D option to tell it we really mean it. Now it's deleted, git branch, and you can see that it's gone. So it's pretty easy to delete branches but luckily git does give you a couple of checks to make sure that you don't do something dumb. So always use the lowercase -d option first and then let git tell you, hey, I want to warn you about something before I actually do this. And then you can change it for the capital D option, if that's what you really need.  
  
<> Configuring the command prompt or terminal to show the current branch you are on.

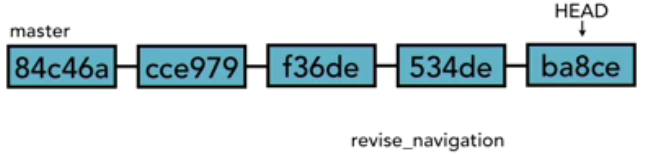
For Unix, Linux and macOS operating systems the first thing you want to do is to make sure that you install the Git Completion Script, which you can put in the .git-completion.bash file in your home directory and then make an entry in your .bash\_profile file to load it because .git-completion.bash declares the \_\_git\_ps1 function that returns the name of the branch you are on. Next in your .bash\_profile file add the Git Command Line prompt you want to have in the PS1 variable (Prompt String 1) so it displays the current branch you are on; export PS1=’\W$(\_\_git\_ps1 “(%s)”) > ‘  
The .bash\_profile file could look as follows:  
alia ll=’ls –lahG’  
  
if [ -f ~/.git-completion.bash ]; then  
 source ~/.git-completion.bash  
 export PS1=’\W$(\_\_git\_ps1 “(%s)”) > ‘  
fi

For Windows, Git usually installs the Git Completion Script, the Git Command Line prompt and it went ahead and set my prompt for me so that it shows it. Now if we want to update the existing Git Command Line prompt or do it ourselves permanently if is not set then you would either update the .bash\_profile in your home directory or create one. So export PS1= and then inside single quotes, we are going to type \W$ and then inside parenthesis I will put our Git command, \_\_git\_ps1, space, quotes, parenthesis, and make sure I got that right, and then I also want to have a caret at the end.

export PS1=’\W$(\_\_git\_ps1 “(%s)”) > ‘

So there is the same command that we had before, and then when you’re done with that, you need to save it as, and you want to save it in your home directory, that's your user directory, and you want to call it .bash\_profile, and you want to make sure that Save as type is set to be All Files, so it won't put any kind of extra file extension at the end. It will just be called .bash\_profile.  Now in order to run that bash\_profile script, just type ‘source ~/.bash\_profile’, and now it runs that command and sets your prompt to be what you want it to be  
  
  
Merging Branches<> Merging Code. Merging changes from one branch into another branch; for instance, a feature branch to the master branch. The first step is that you want to make sure that we checkout the branch that things are being merge into, the receiver branch, then execute ‘git merge <merging-from-branch-name>’. You’ll see a summary of the changes that were merge or made:  
Updating <from-commit-id>..<to-commit-id>  
Fast-forward  
 index.html | 2+-  
 1 file changed, 1 insertion(+), 1 deletion(-)  
  
This was a fast-forward merge, meaning you did not make any changes or commits on the local repo of the branch (say master) from which you created your new branch (say revise\_navigation), you only made changes to the new branch (say revise\_navigation). Fast-forward merge looks like this:  
  
Now if you were to do ‘git diff <receiver-branch-name>..<merging-from-branch-name>’, Git comes back and tells us that there is no difference between them. If we were to use ‘git branch --merged’, Git will tell us that the <merging-from-branch-name> is fully incorporated into the <receiver-branch-name>.

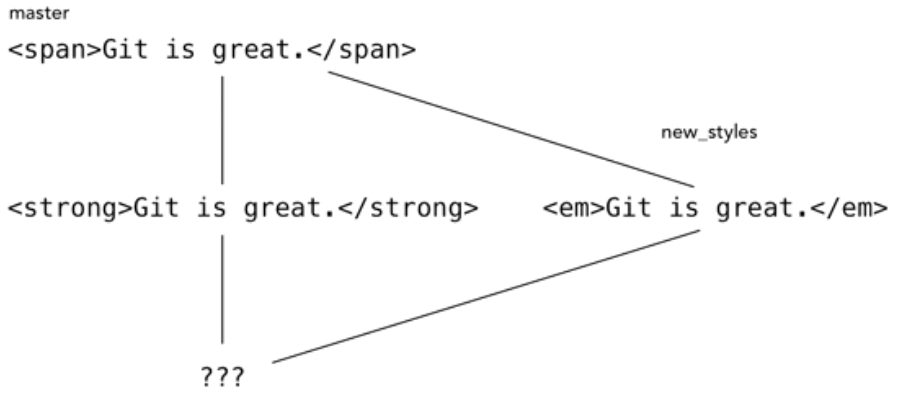
Note that you always want to run git merge or merge branches with a clean working directory (workspace).  
  
<> Using fast-forward merge vs. true or real merge.

Fast-forward merge was explained above in the previous section. When Git goes to do a merge, it takes the thing that you’re merging in (<merging-from-branch-name>), and it starts at the end of it, and it looks back at all the ancestors, all the way back up to the beginning. In our fast-forward merge illustrated above, Git starts with ba8ce and then goes to 534de, all the back to get to 84c46a, all the way down the commit chain. And along the way, Git looks to see to which commit id is the HEAD pointer pointing to in the current branch. In our case, the HEAD pointer was pointing at the commit id 534de where the branch was made, no movement had been made. If it does that, if in the chain of ancestors it sees the HEAD, it says oh, I'm safe to do a fast-forward merge. I don't need to do that fancy merging and make a new commit, instead what I can actually do is I can just move that commit up into my timeline and move the HEAD along to it.  Now revise\_navigation also points to the same thing, they both point to ba8ce. There was no need to make a new commit. You could just fast-forward along the chain and merge that way.  
  
Now notice that the top commit here ba8ce, that's the thing that had changed before in <merging-from-branch-name>, that's what we merged in to <receiver-branch-name>. If you take a look now at <receiver-branch-name> you’ll see the same commits here, it's got the same commit id (SHA-1 or checksum) ba8c2, it's the exact same object stored in Git. It did not make a new commit in order to merge these two together, it did a fast-forward.

Now there are a couple of options with merge that are related to the fast-forward. The first is the *no-ff option* so that would be ‘git merge --no-ff <merging-from-branch-name>’ and then whatever branch you were trying to merge. The no-ff option forces Git to create a merge commit anyway. It says, don't do a fast-forward, make a new commit with the commit message anyway. And the main reason you'd want to do this is if you wanted some kind of documentation of the fact that you did do this merge.

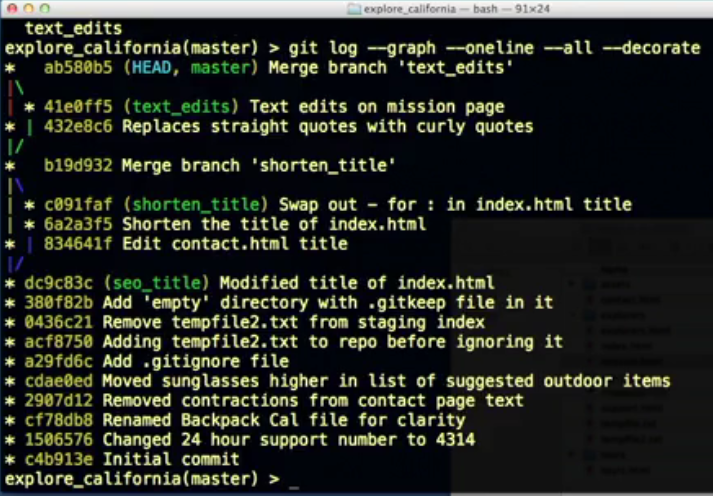
You didn't want it to just quietly do it, you wanted it to sort of make some noise in your git log. The other option you should know about is the *ff-only option*. Don't get those confused, no-ff says don't do a fast-forward, ff-only says do the merge only if you can do a fast-forward. If you can't do a fast- forward merge, then just abort. Don't try and do it at all just exit.  
  
True or real merge is a non-fast-forward merge and the way to do a real merge is by making another commit to the local repo of the <receiver-branch-name> (say master) after you created the <merging-from-branch-name> (say revise\_navigation). If there's another commit on <receiver-branch-name> (say master), then Git'll no longer be able to just fast-forward, because now HEAD in the <receiver-branch-name> (say master) will have moved to a new commit that is not in the <merging-from-branch-name> (say revise\_navigation). Now you are ready to do a real merge, first make sure that you’re in the branch that things are being merge into, the receiver branch, then execute ‘git merge <merging-from-branch-name>’. If you configure Git to ask you to enter a commit message in this scenario then Git popped up, and it asks you to provide a commit message here.

Git merge has different strategies for merging, recursive is typically the one that you're going to see in use by Git.  
  
<> Merging conflicts. Git merge is able to merge changes to the same file without any issues from two different branches into one branch when the changes in one file are in different lines, say one branch has a change at the top of the file and the other branch has a change at the bottom of the file. Git sees those, it recognizes the line numbers and realizes that those are not next to each other, and it takes both of those changes and incorporates them into one composite document. However, a conflict occurs when there're two changes to the same line or set of lines in two different commits, because then Git can't decide which one to use or how to merge them together and so we get a merge conflict. And this is the one headache about working with branches is resolving merge conflicts.

Let's say that we have our master branch, and we have a line somewhere in one of our HTML files this says Git is great, and it has <span> tags around it. From my master branch we create a new branch, we'll call it the new\_styles branch. And we start making edits over there. We got all sorts of commits going on, at the same time we're switching back to master, and we're making edits back there as well. One of the edits that we commit to master is that we change the <span> tags into <strong> tags, and then we keep making more commits, we switch back to new\_styles, and we’re making more commits over there. At some point, we make an edit in the new\_styles branch so that we change the <span> tags into <em> tags. So now we've two different versions of this line. In the master branch it has <strong> tags around it, in the new\_styles branch it's got <em> or emphasis tags around it.  


So when we go to merge new\_styles back into master, Git doesn't know which one of these we prefer. Git says I can't choose between these. I don't have any idea what's your intention was, so I need you to tell me. So git will mark the conflict and wait for you to fix the problem.  
  
Auto-merging <filename>  
CONFLICT (content): Merge conflict in <filename>  
Automatic merge failed; fix conflicts and then commit the result.  
  
Now notice that your branch now says master|MERGING. You’re not fully on your master branch, you’re in the middle of a merge, and Git is letting you know that. If you do ‘git status’, Git comes up and tells you, unmerged paths, Git tells you to use the ‘git add/rm <filename>…’ as appropriate to mark the resolution, Git tells you that both version of the <filename> have been modified, and so nothing is ready to commit yet.  You need to fix these problems. Open up the <filename>, and let's scroll down to see how Git marked those problems inside the <filename>.  So now it's up to you to go through and decide which one of these you want. But there are three lines that were different in each one, and each one had some conflicts in it. Now that we see how merge conflicts occur, and we see that Git marks them using symbols like “<<<<< HEAD”, “=====”, “>>>>>> new\_styles”, we're ready to talk about how to resolve these merge conflicts, and we'll do that in the next section below.  
  
<> Resolving merge conflicts. You've three choices as to what to do to resolve these conflicts.

First, you can simply abort the merge. You can say oops! That was not what I wanted to do, I was not anticipating all of these problems, get me out of here. To abort the merge, while we're in this state, all we have to do is execute ‘git merge –abort’ while in the master branch. That's it. Now notice I'm no longer in my master|MERGING state, git status, everything is clean, and if I do git log --oneline on my master branch, you can see that it did not bring anything else. This is the state of things right before the merge. So nothing got merged, I simply aborted it. Now let's do our merge again, git merge new\_styles, once again it did the same thing, created the same conflicts for us.

The second is that you can resolve the conflicts manually. Most of the time this is what you're going to do. What you need to do is you need to resolve the conflicts by hand in <filename>, and then add and commit the result. So that's what you're going to do. Open up <filename>. Look for the markers made by Git inside the file, this arrows pointing to the left, going down to the equal sign all way down here to the arrows pointing to the right, is how we know where the conflicts are. Now you can go through these line by line and try and sort it all out. Next, you’ll want to inspect it and make sure that it all looks good, make sure that you’re completely happy with the results, that the merge conflicts has been resolved to your satisfaction. Now in the master branch execute ‘git add <filename>’ and then ‘git status’ shows that <filename> is ready to be committed to the master branch’s local repo. Do ‘git commit –m <message>’, note that you don’t have to provide the commit message when you’re in the middle of a merge, Git has a standard default message that it would use. If you use ‘git log --graph --oneline --all --decorate’ you get a nice showing or graphical representation of the different branches and what happened.  


And then the third one is that you can use a merge tool. There're number of tools out there that will help you to resolve these kinds of conflicts.  
  
<> Exploring strategies to reduce merge conflicts –and to make them easier to deal with.  
1. Keep lines short.  
2. Keep your commits small and focused.  
3. Beware stray edits to whitespace  
 - spaces, tabs, line returns  
4. Merge often when feasible  
 - merge conflicts with 3 files is easier to deal with instead of 50 files.  
 - Lets just take a quick look to make sure that that's clear. If you have your master branch, let's say, and we've  
 our text\_edits branch. So we're making commits to both of those, we're merging back in. Now we don't have  
 to throw away our text\_edits branch at this point. We can still continue to make new commits into the  
 text\_edits branch, and new commits to the master branch and then merge those commits back in. We can  
 make more edits in to our text\_edits branch, more commits in the master branch and then merge those back  
 in, so this is what I mean by merging often.  
  
  
5. Track changes to master as you go.  
 -  As changes continue to happen in master, keep bringing those changes into your branch so that your branch  
 stays mostly in sync with master, it doesn't get far out of sync.  
  
 - And then that way when you finally decide that you want to merge text\_edits back into master, it's not that  
 far away anymore. Branch text\_edits has most of the changes incorporated in master already, and we'll  
 reduce the number of conflicts that you get when you merge back in. We call this process tracking, and it is  
 an important strategy to use.  
  
  
Stashing Changes<> Saving changes in the stash. The stash is a place where we can store changes *temporarily* without having to commit them to the [local] repository.  The stash is not part of the [local] repository, the staging index or the working directory (workspace), it's a special fourth area (working tree) in Git, separate from the others. And the things that we put into it aren't commits, but they're a lot like commits, they work in a very similar way. They're still a snapshot of the changes that we were in the process of making, just like a commit is. But they don’t have a SHA-1 or checksum associated with them. To stash your changes in your working directory (workspace) execute ‘git stash save <stash-message>’, this saves the working directory and index stat on the current branch and it tells you in which commit id (tree-ish) were your changes stash.

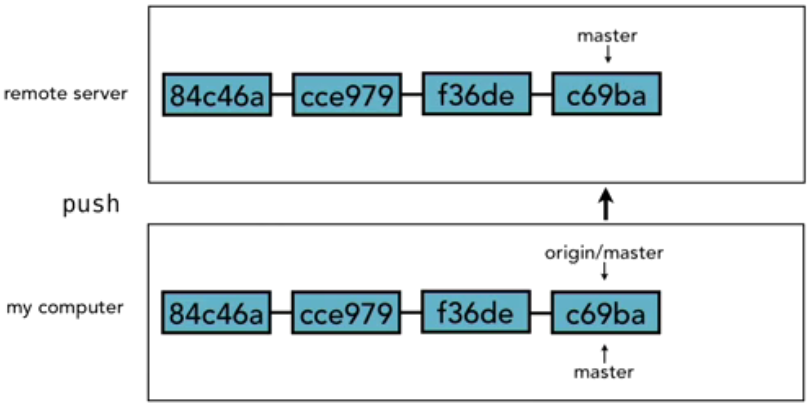
After you put those changes into the stash, Git ran ‘git reset hard HEAD’, that takes whatever is in the local repo, and puts that into both our index and our working directory (workspace), so they're exactly the same as the commit where HEAD is pointing right now.

\* Note that changes saved in the stash are put on the top of the Stack (LIFO – Last In First Out).  
  
<> Viewing stashed changes by executing the ‘git stash list’ command. Things save into the stash are saved and reference as ‘stash@{0}: On <branch-name>: <stash-message>’; the number here is 0, next stash save will have a 1, then 2, then 3, then 4 and so on. Next is tells you the <branch-name>, that’s the branch that you were on, when you put your changes in the stash because the stash is going to be accessible even when I switch branches. So if you switch to another branch, say your master branch, git checkout master, and then you do ‘git stash list’. The stash@{0} still there. The stash is available all the time. You can always pull from it. That makes it really handy, especially if you start to make a change on one branch, and then you realize this isn't the branch that you wanted to commit those changes to, you can put them in the stash, change your branches and then pull them back out of the stash again.

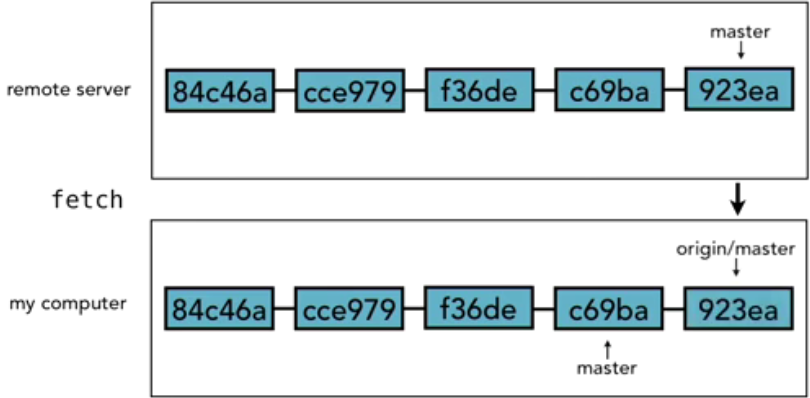
To see more information about each stash execute ‘git stash show stash@{<number>}’ and by default Git shows you what’s call diff stat. If you want to see more information use the –p option, which tells Git to you the stash as a patch. A patch is a section of code that you can apply to different things to modify them and change them. So this is basically saying, show us the edits. Execute ‘git stash show –p stash@{<number>}’ and this will show you the typical diff.  
  
<> Retrieving stashed changes by executing ‘git ‘, which will bring those changes back into the working directory (workspace) of the current branch you’re on. Like with merges, there's the possibility that there may be merge conflicts that those changes don't apply cleanly, and in that case it works like merge does. It does its best to try and figure out how to merge the changes, but if not, then there will be conflicts, and it'll be up to you to resolve those conflicts.

Now there are two git commands that we can use to pull items out of the stash. The first is going to be ‘git stash pop’ and the other one is ‘git stash apply’, both of them will pull what's in the stash out, and put it in the working directory. The difference is that stash pop also removes the changes from the stash as well. Git stash apply leaves a copy of the changes in the stash, so pop is going to pop it out so that it's no longer in the stash at all. The idea here is that it might be some change that you want to apply to where we are now, then you might want to switch to a different branch and apply it there, switch to another branch and apply it there as well. So you want to keep it in the stash as we move between each of those branches. Or it might be something that you want to apply then make a few more commits and then apply it again, and so on.

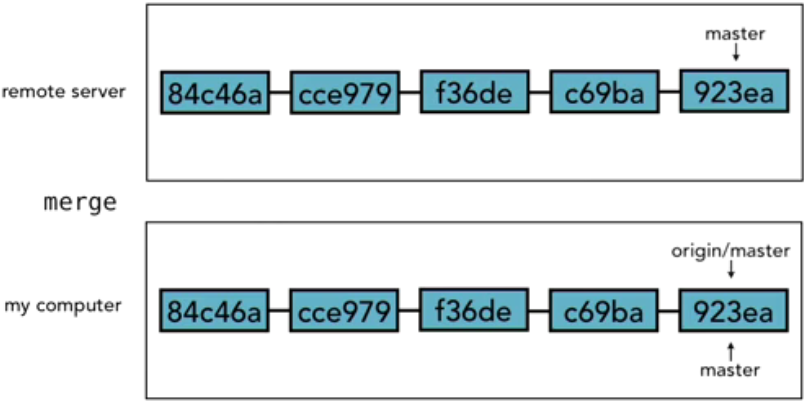
After pop, you need to specify which stash item you want Git to pull out, ‘git stash pop stash@{<number>}’. If you don't say then by default, Git's going to pull the first one.  If you had three stashes, and you wanted to pop up the third one, well then you'd do ‘git stash pop stash@{2}’, 2 because the third one is actually number 2 because they start numbering at 0; 0, 1 and 2. Apply works the same way, you need to specify which stash item you want Git to apply, ‘git stash apply stash@{<number>}’. If you don't say then by default, Git's going to pull the first one in apply mode.  
  
<> Deleting stashed changes by executing ‘git stash drop stash@{<number>}’ will drop (delete) that one reference stash item. To delete everything that’s in the stash all at once execute ‘git stash clear’.  
  
  
Remote Repositories or Remote Repos or Remotes<> Using local and remote repositories. Remote Repos allows us to collaborate with others. The concept is that there is a remote server, and we'll take our changes that we've made from our local repo(s) and put them on that remote server so that other people can then see them. They can then download the changes that we've made to their local repositories, they can make changes of their own, upload those back to the remote server (remote repo[s]), and we can pull those back down into our local repository to get their changes. The remote server is just simply a Git repository.



-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

  
  
origin/master is just a branch like any other branch with its own pointer (just like HEAD) on our local repo (machine) that references the remote server branch, and it always tries to stay and sync with that.   
  
<> Setting up a GitHub account. GitHub is the most popular Git host because they offer features like charts and graphs that show the state of all the different branches and the commits that have been made, or graphs showing the activity on a project over time.

The following is a list of other Git Hosting companies or options: Beanstalk, BitBucket, Gitbox, Git Extensions, GitHub, GitLab, Gitorious, GitSwarm, GitX, RhodeCode, SmartGit, SourceTree, [Microsoft] Team Services, TortoiseGit, etc. See Git Wiki for all options <https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools>.  
  
<> Adding a remote repository. In your laptop’s git bash navigate to your git project and execute ‘git remote’, this will give you a list of all the remotes (remote repos) that we know about, like ‘origin’ (default name); if you execute ‘git remote –v’ will give you more info (-v for verbose).  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git remote -v

origin ssh://git@<hostname>:<port>/web\_app.git (fetch)

origin ssh://git@<hostname>:<port>/web\_app.git (push)

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$

If you have a git project (local repo) in your machine that is not connected to a remote repo (say Bitbucket or GitHub) and you wanted to push your changes from your local repo to the remote repo, first you have to connect your local repo to the remote repo, execute ‘git remote add <remote-alias-name> <remote-server-url-or-ssh>’ inside your git project.

If you want to disconnect your local repo from the remote repo, then inside your git project execute ‘git remote rm <remote-alias-name>’.  
  
<> Creating a remote branch. To move your code from your local repo to the remote repo you push your current branch in your local repo to the remote repo by executing ‘git push –u <remote-alias-name> <branch-name>’, for example ‘git push –u origin master’. The –u option sets up your branch in your local repo to track branch in the remote repo; in other words, -u allows you track your remote branch. You can see the branches in your local repo by executing ‘git branch’. To see the branches in your remote repo connected to your local repo execute ‘git branch –r’. To see both execute ‘git branch –a’.  
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git branch

\* feature/svn\_bb

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git branch -r

origin/HEAD -> origin/master

origin/feature/svn\_bb

origin/master

<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$ git branch -a

\* feature/svn\_bb

remotes/origin/HEAD -> origin/master

remotes/origin/feature/svn\_bb

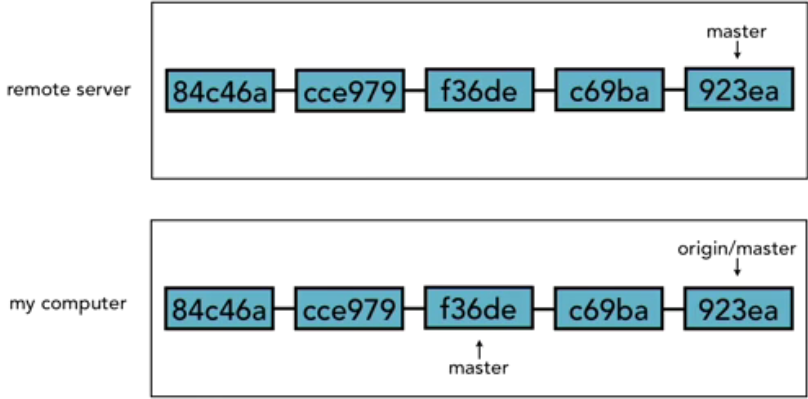
remotes/origin/master

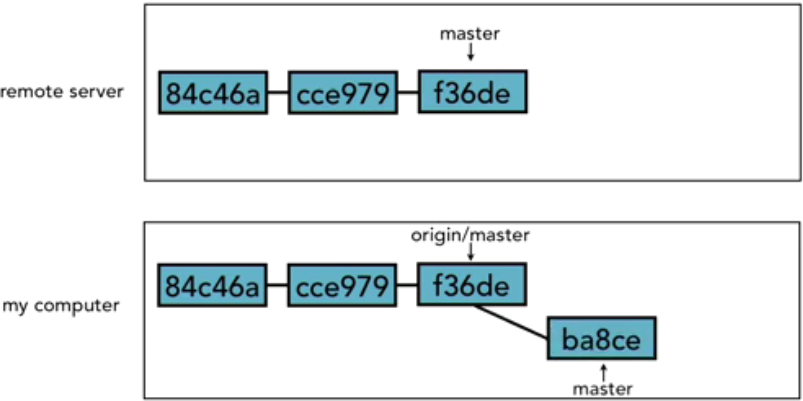
<UID>@<HOSTNAME> MINGW64 /c/<path>/web\_app (feature/svn\_bb)

$

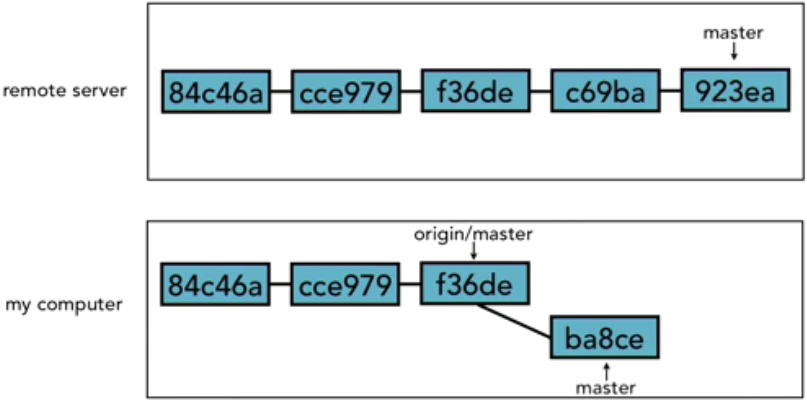
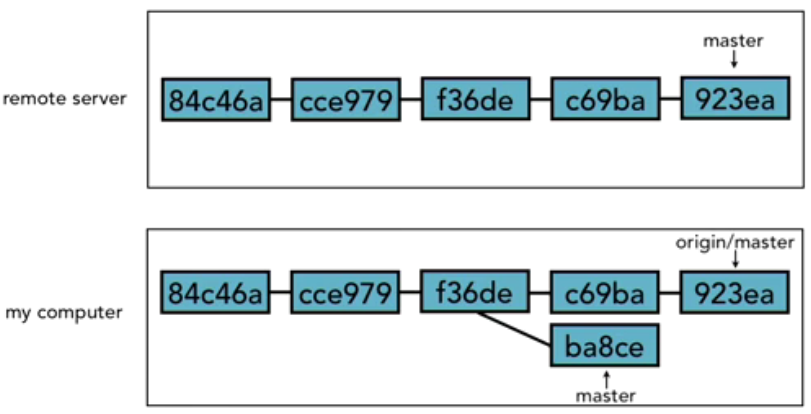
<> Cloning a remote repository  
‘git clone <remote-server-url-or-ssh>” gets copy of the master branch from the remote repo into your computer.  
‘git clone <remote-server-url-or-ssh> <git-project-folder-name>” gets copy of the master branch from the remote repo into your computer with git project folder name you specify in the git clone command.  
  
<> Tracking remote branches with the –u option in the ‘git push –u <remote-alias-name> <branch-name>’ command. The -u option says push it up there, and also make a note of the fact because we're going to be coming back and working with this branch frequently. When you do a git clone, it does track the remote branch.

<> Pushing changes to a remote repository  
git log --oneline <remote-branch-name>, example ‘git log --oneline origin/master’ gives you a list of your commit ids from your copy of the remote branch.  
git diff <local-branch-name>..<remote-branch-name>, example ‘git diff master..origin/master’ or ‘git diff origin/master..master’.  
If you are pushing your changes to tracking [remote] branch you just execute ‘git push’ and Git knows where to push because Git is tracking that [remote] branch.  
  
<> Fetching changes from a remote repository. Origin/master in your local repo does not automatically reflect what’s on the remote repo, you have to tell Git to do a sync between the two. ‘git fetch <remote-alias-name>’, for example ‘git fetch origin’. Git fetch synchronizes the local repo with the remote repo, and that means it pulls down any Git objects we don’t have, and it pulls down bookmarks that reference the tips of each of the branches that are on the remote repo.

Now if we look at the log for just master, not origin/master, but just our master branch, you can see that that commit is still not there. I didn't bring it all the way into our master branch, that's ours to manage. It just brought it into the origin/master, which is going to try and always be a perfect sync with what's on the remote repository. The origin/master branch is just our local cached version of the remote repo. So that's an important point, when we synchronize with remote repository using fetch, we just update origin/master. Master doesn't change at all. If we're in the middle of working on something, and we've got some edits in our working directory in the master branch, those aren't affected. If we've made some commits over the last hour, those aren't affected in master branch.  
  
Three basic guidelines for fetching:  
1. Always fetch before you start coding.  
2. Fetch before you push.  
3. Fetch often.  
  
<> Merging in fetched changes. origin/master is just a branch like any other branch in the local repo (machine) that references the remote server branch, and it always tries to stay and sync with that. Note that you cannot checkout the origin/master branch because Git really needs to be able to keep it in synch with what’s on the remote server, and it does not want developers getting in the way of that.  
  
In the illustration above, you can see that the remote server has five commits in it, and a fetch has taken place, because the origin/master branch in the local repo is perfectly in sync with master branch on the remote repo. Our master branch in the local repo however is two commits behind, so it needs to have those two commits added to it, and the process that we do that is by merging. But in this particular illustration above, if we were to do a merge right now, the master in the local repo would do a fast-forward merge. Now if we made additional commits on the master branch in the local repo, in the meantime, then those would need to be merged in, and we would have a merge commit that would join those together. And of course, we have got to resolve any merge conflict that comes up.

To merge the origin/master branch with the master branch in your local repo or any other branch in your git project, checkout the receiving branch and then execute ‘git merge origin/master’. Another way to fetch and merge in one Git command is with git pull which is equal to git fetch + git merge. Git pull finds the latest in the remote repo master branch, brings changes down into original/master branch in your local repo and then merges the original/master branch into your current branch, all in one step.  
  
<> Checking out remote branches  
‘git clone <remote-server-url-or-ssh> -b <branch-name>” gets copy of a specific branch from the remote repo into your computer, but if your local repo is tracking a branches in a remote repo then you can just get a copy of the remote branch in your local repo into a new branch, because you can’t checkout remote branches in your local repo. You do this by executing the ‘git branch <new-branch-name> <remote-alias-name>/<remote-branch-name>‘ for example ‘git branch non\_tracking original/non\_tracking’ or by executing ‘git checkout –b <new-branch-name> <remote-alias-name>/<remote-branch-name>‘ for example ‘git checkout –b non\_tracking original/non\_tracking’, git checkout –b means create a new branch and checkout the new branch, all in one step.  
  
<> Pushing to an updated remote branch. Imagine that we have our remote server and our local repository and both of them have three commits in it. We do a fetch, so we know that we are completely in sync, we just have these three git commits. So once I know that I am in sync, I start my work for the day. And over the course of the next hour I put together a commit. I then commit it to my local repository.  


At the same time though, my coworkers and collaborators have also been making commits and they've been pushing their commits up to the remote server and to the same master branch in the remote repo.

So now when I try to push my commits to the remote server, the remote server says, well, I don't know what to do with your commits, there has been some new stuff that's come in in the meantime. Git never tries to do a merge during a push. Instead what Git says is, some new stuff has come in. I'm not sure what to do about all this.  
  
You need to fetch the changes that are on the remote server, then sort it out on your end (local repo) and come back and try again. So we do a fetch. We've got the new information about what's on the remote server, we now see why there's a problem, oh, there were some other commits there.  
  
But that doesn't solve our problem. We've still got to merge our changes in with those. This is not just because there's a conflict, it's not because I changed Line 37 of index.html and someone else changed Line 37 of index.html, it's just simply because there were new commits. So what we need to do in this case is git merge origin/master, that will then merge my changes in with origin/master and create a new merge commit, and then I can push again and the remote server will accept it.

<> Deleting a remote branch. Essentially, telling the Git host (like GitHub or Bitbucket) to erase one of the branches in its [remote] repo. There are two ways you can do this:  
1. The ‘git push <remote-alias-name> :<remote-branch-name>‘ command will have the effect of deleting the remote branch in your local repo and the branch on the remote server but you still have the regular [local] branch in your local repo; for example, ‘*git push origin :non\_tracking*’ deletes the non\_tracking remote branch in your local/remote repo but it keeps the non\_tracking local branch in your local repo. ‘git push origin non\_tracking’ is shorthand for ‘git push origin non\_tracking:non\_tracking’ which means push to origin the non\_tracking branch in my local repo to the non\_traacking branch in my remote repo. So when we are doing a delete, what you are actually doing is saying, push to origin nothing up to the branch non\_tracking. So that's why that colon is there.  
2. The ‘git push --delete <remote-branch-name>’ command will delete the remote branch in your local repo and the branch on the remote server but you still have the regular [local] branch in your local repo; for example, ‘*git push --delete non\_tracking*’ deletes the non\_tracking remote branch in your local/remote repo but it keeps the non\_tracking local branch in your local repo.  
  
<> Enabling collaboration; in other words, allowing other Git host (remote servers like Github, Bitbucket, etc.) developers using Git to have access to your projects in the remote server. For example, in an open-source project the way that you make changes is that you need to make a *fork*. This will make your own version of the project on your own Git host (remotes like GitHub) repository. It's no longer part of the main one, and this one you will have write access to. So you go ahead and clone the repository, work with it locally just like you normally would, commit those changes up to your version of the project, and then once you got it all done, it's ready to go, you go back to the Git host (remotes like GitHub) page for the main project, and you issue a *Pull Request*.

Essentially a *Pull Request* is like raising your hand and saying I have something here that I want to show you. You submit a message with your request so you identify what the problem was that you saw, or what feature you decided you wanted to add, talk about how you want to do it, and why you think it's good for the project, and if you make the case effectively and your code looks good then they will accept your changes and incorporate them into the main project. They'll grab your branch and merge it in. And then at that point, everyone will have access to your new feature. So that's how you can enable collaboration using Git.  
  
<> A collaboration workflow  
Developer 1’s Work  
> git checkout feature/<branch\_name>  
> git fetch  
> git merge origin/master  
> git checkout –b referral\_form  
> git add referral.jsp  
> git commit –m “Add customer referral form”  
> git fetch  
> git push –u origin referral\_form  
> notify developer 2  
 > git fetch  
 > git log –p referral\_form..origin/referral\_form  
 > git merge origin/referral\_form  
 > git checkout master  
 > git fetch  
 > git merge origin/master  
 > git merge referral\_form  
 > git push  
  
Developer 2’s work  
> git checkout feature/<branch\_name>  
> git fetch  
> git merge origin/master  
> git checkout –b referral\_form origin/referral\_form  
> git log  
> git show <commit-id>  
> git commit –am “Add alert selector to referral form”  
> git fetch  
> git push  
> notify developer 1  
  
  
Tools and Next Steps<> Setting up aliases for common commands in .gitconfig with ‘git config --global alias.<alias> <git-command>’.  
st for status  
co for checkout  
ci for commit  
br for branch  
df for diff  
dfc for “diff --cached”  
dfs for “diff --staged”  
logg for “log --graph --decorate --oneline --abbrev-commit --all”  
  
<> Using SSH keys for remote login  
  
<> Exploring integrated development environments. Most editors and IDEs have some kind of Git integration available; for example, macOS editor TextMate has bundles, and under Bundles you’ll see there is a listing for Git. So TextMate > Bundles > Git. And there’s lots of different features that you can do right from TextMate in Git. The following are some of the most popular editors and IDEs that you might use: Vim, Emacs, TextMate, Eclipse, Netbeans, Visual Studio, IntelliJ, RubyMine, PHPStorm, etc.  
  
<> Exploring graphical user interfaces. There are several Git GUIs like  
- [GitWeb](https://git-scm.com/docs/gitweb), which is included with Git cross-platform.  You need to actually set up a web server to be able to do this, because you do need to be able to set up the web server to host GitWeb, but what it does is it lets you see your git repository as a little mini web site on the web where it's available for everyone to be able to see.  
  
Git GUIs for macOS/Windows  
- [Gitbox](http://www.gitboxapp.com) automatically fetches new commits from the remote server helping you avoid merge conflicts/commits.  
- [GitHub Desktop](https://desktop.github.com), second best Git GUI, this is GitHub’s GUI.  
- [Git Extensions](http://gitextensions.github.io) allows you to browse the full commit history.  
- [GitX](http://gitx.org), the most popular for macOS and probably one of the oldest ones.  
- [SmartGit](http://www.syntevo.com/smartgit/) works with GitHub, BitBucket and Atlassian Stash to create/resolve Pull Request and Review comments  
- [SourceTree](https://www.sourcetreeapp.com) helps you visualize and manage your repos.  
- [TortoiseGit](https://tortoisegit.org) provides overlay icons showing the file status, a powerful context menu for Git.  
- [Tower](https://www.git-tower.com), Gibran’s favorite, the best, seamless integration with Beanstalk, GitLab, GitHub, Bitbucket, etc.  
  
There are many, many of these and new ones are appearing all the time. There is actually a pretty good list maintained on the [Git Wiki – Interfaces Frontends and Tools](https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools), that'll give you a list of all the different interfaces, front-ends, and tools that are available to you.

|  |  |
| --- | --- |
| **Command** | **Description** |
| **git status** | Tells you if you have any unstaged/uncommitted changes. Can also tell you if you're ahead/behind your branch (but don't trust it until you know how it works) **USE THIS COMMAND ALL THE TIME!!!** |
| **git log** | [Show commit log. There are a lot of options depending on what you're trying to find https://git-scm.com/docs/git-log](https://git-scm.com/docs/git-log) |
| **git add <. || path/to/filename (wildcards like \* accepted)>** | This stages modified/new files for commit. If you use ".", it will stage all modified/new files recursively. This is the most common usage. If you want to add one file or one directory, you can use path/to/filename or path/to/\* |
| **git commit [-a, -m, -am]** | Commit staged files to the local repository. **This does not put them on the server**. |
| -a will perform *git add .* as well as *git commit* |
| -m allows you to specify a commit message inline. |
| Most common usage is *git commit -am "commit message"* - this allows us to add and commit all modified files in one command |
| **git pull origin** | Pulls all changes from origin remote, including branches **Note: this does not give you local copies of all branches** |
| **git checkout --track origin/<branchname>** | Checkout a remote branch, set the local branch to track the remote branch **If you use 'git pull origin' first, you can just git checkout <branchname> and it will do the same thing.** |
| **git pull origin <branchname>** | Pull latest changes from a specific branch on the origin remote. You need to be in the same branch on your local repo. |
| **git push origin <branchname>** | Push **all local commits** that are not on the remote repo already. If you've made 3 commits since your last push, it will push all 3. |
| **git push origin :<branchname>** | Delete a branch on the server. You should only do this after you've deleted the branch locally with git branch -d <branchname> |
| **git merge <branchname>** | Merge a local branch into your current local branch |
| **git checkout <branchname>** | Switch to the specified branch. **If you have any uncommitted changes, they will switch branches with you.** |
| **git branch [-d] <branchname>** | Delete a branch locally. Make sure you're in some other branch when you perform this command |
| **git reset --hard HEAD** | undo any non-committed changes in your local repository so you have a clean working copy |
| **git stash <save|apply|list|show|pop|drop|clear>** | [https://git-scm.com/docs/git-stash This is worth knowing about. When you have changes locally that you don't want to commit, but you need to switch branches, you can use git stash. This is just one of many, many uses for this very handy command.](https://git-scm.com/docs/git-stash) |
| **git bisect** | [https://git-scm.com/docs/git-bisect This is also worth knowing about. When you have an issue and you can't figure out what it is or what commit introduced it, git bisect is your best friend](https://git-scm.com/docs/git-bisect) |
| **git blame** | [Show what revision and author last modified each line of a file. This is handy if you need to know who did what to a given file. https://git-scm.com/docs/git-blame](https://git-scm.com/docs/git-blame) |
| **git config --global alias.<short> "long form"** | Set alias so you don't have to type it all. Example: *git config --global alias.cam "commit -am"* Now, you can simply type *git cam "my commit message"* and it will add any unstaged changes and commit them with the message specified, just like *git commit -am* would. |