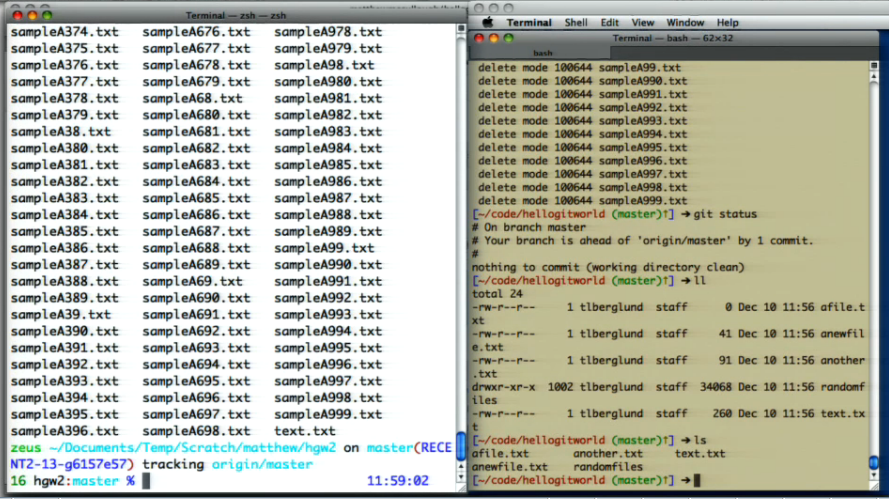
**Notes on Git Version Control**

By: Gabriel Moreira

**The Idea**

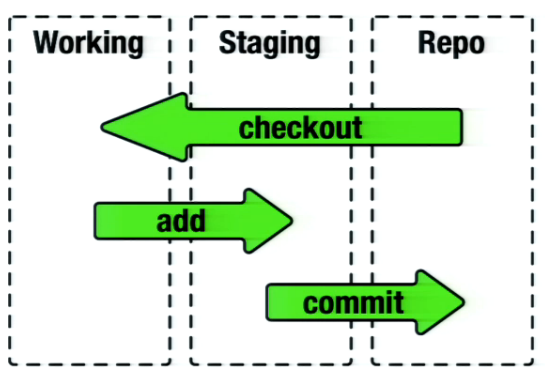
* git tracks the content of files, the source code. Git is a content tracking system
  + that’s why git ignore blank folders as if they didn’t exist
* git makes it so your files are on your disk, not on your network. You barely ever talk to the network, therefore the speed of commands is REALLY fast
  + generate, add, commit, and push 2000 random files will take ~2 seconds on git, or up to 5 hours on other VCSs
  + when you push the code, it compressed all of them together. So there is only 1 send and one open
* Because all of the files of your project are in your computer, giving that you already did *git clone,* you are not talking to the server anymore. Therefore if two people clone the same repo, the can work on the same thing at the same time, this could cause conflicts because lets say code X is being updated by person A, but person B is also working on code X. Once A pushes the updated code, B has to update his local directory. Otherwise A and B’s changes might interfere each other
  + E.g: Two people working on the same exact repository on two different machines. Note that person B deleted all of the files on his personal Repo, but person A still has all of them on his personal Repo. Person A can say git pull, and the server would say that all of your files are up to date, since person B did not push his directory yet. Remember every change you do is done local on disk, until the point you talk w the network to push/pull the code

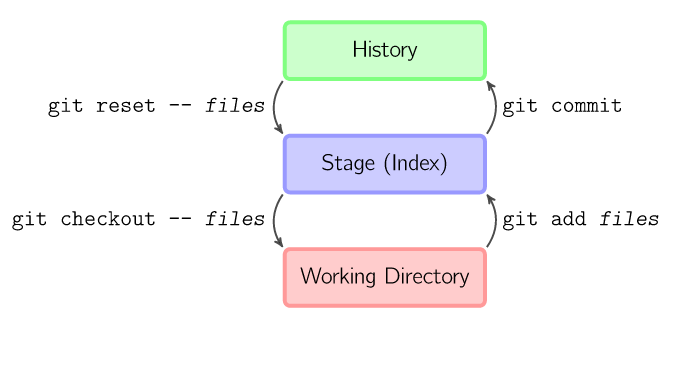


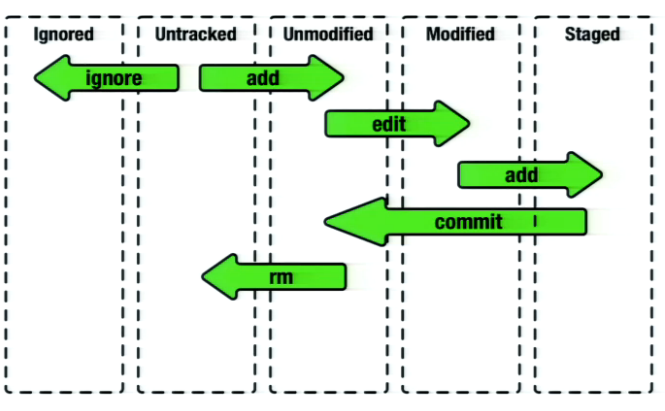
* Git uses a DAG, Directed Acyclic Graph, like a single directional linked list to store its data. Every commit you do becomes one more node to this graph. Therefore the more commits you do, the slower git commands become
  + There is a copy of the entire tree per checking
  + This creates a time vs space trade off. Its faster, but it takes more space
* Git-flow is one of the many branching models **INFO FOR BLACKBAUD GIT FLOW:**
  + The master branch reflects the released versions of the code that are in production.
  + The develop branch contains what is going to go out into production with the next release. This means you should not check your code into the develop branch, but rather into a bug/feature branch where QA can test and sign off and code review can occur before the branch is merged into develop.
  + Always create a new feature branch for each feature or bug-fix you are working on. This ensures that the branches are created in a consistent way and will be made available to load on our BVT test sites through Alfred.
  + Continually merge from develop into your active feature branches as often as possible. This will help you keep your branch up to date with the work other developers have done and will make the merge back into the develop branch easier.
  + Try to merge feature branches back into the develop branch as often as possible. If possible break up long-running feature into smaller parts that can be merged into develop as 'dark code' that isn't really called if needed.
  + You should always update develop before creating a new feature branch.
    - Git fetch –p
      * Gets latets changes from origin
  + You should **never** do merge commits in develop when updating from origin.
  + **More IMPORTANT info here:** http://twiki.corp.convio.com/bin/view/Engineering/LuminateOnlineGitDocs#Best\_Practices

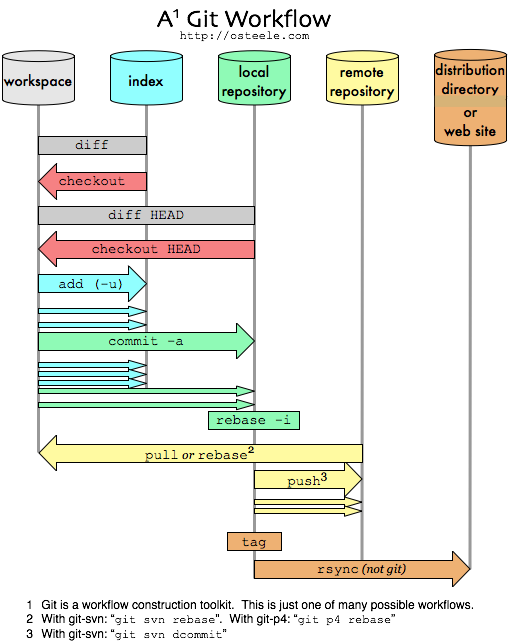
**Terminology:**

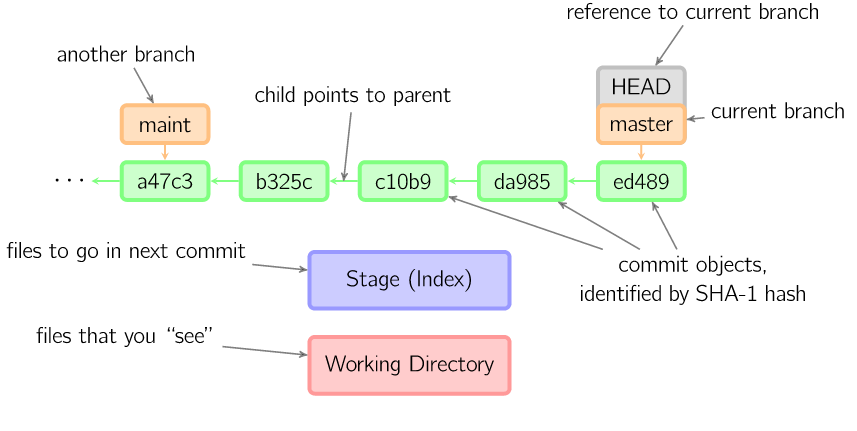
* **Version Control**: Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.
* There are three stages to version control
  + Working, Staging, Repository
    - **Working:** That’s where you are pulling files from
      * Your files system, your directory where you are currently in
    - **Staging:** That’s where you are temporally keeping you files
      * Semitransparent layer where you put code to, but not final/checked in the version control. A Purgatory location where you can chose to put back in the working directory or push it forward and actually commit it
    - **Repository (AKA Repo):** That’s where you will move you files once you are done
  + Imagine it like a super market
    - Working: is the shop, holding all the different files
    - Staging: is your shopping cart. You pick what items you want to add there, you can check at any time what is in you cart and see if anyone else put any items there while you were not looking. You can them make them important decision of what will I actually buy? (what will I actually push to the Version Control system?)
    - Repository: is once you paid for those items, and there is nothing left to do
* **Ignored**: files that I want git to keep their hands of
  + Generally important things, like a file containing passwords
* **Untracked files**: files not in the staging area. Git has not yet been told to pay attention to this file
  + To make git pay attention to this file, you would use the git add command
* **Modified:** when you change the content of a file and do git status, git will display what files had their contents changed
  + If shown in red, the file has not been added, shown in green otherwise
* **Branch:** Branches are used to develop features isolated from each other. The master branch is the "default" branch when you create a repository. Use other branches for development and merge them back to the master branch upon completion.
* **Treeish:** shorthand for hash codes. This hash codes are generated through every commit. Each change to your file, once you commit, will result in a new generated hash code
* **HEAD:** symbolic word that will expand out the hash code

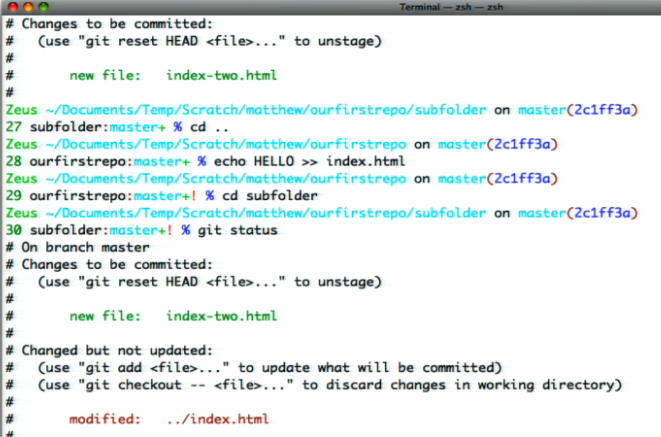




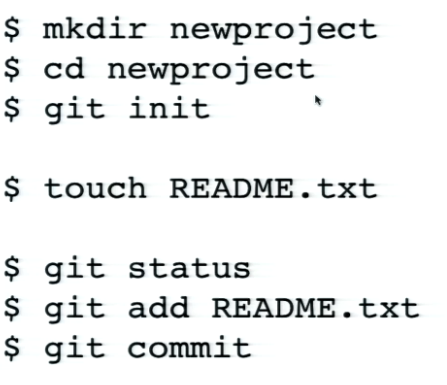
* + More in depth:



* 

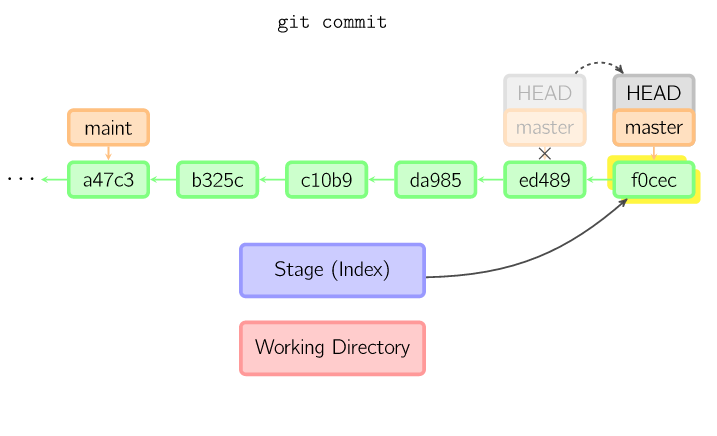


The flow of commands, example:

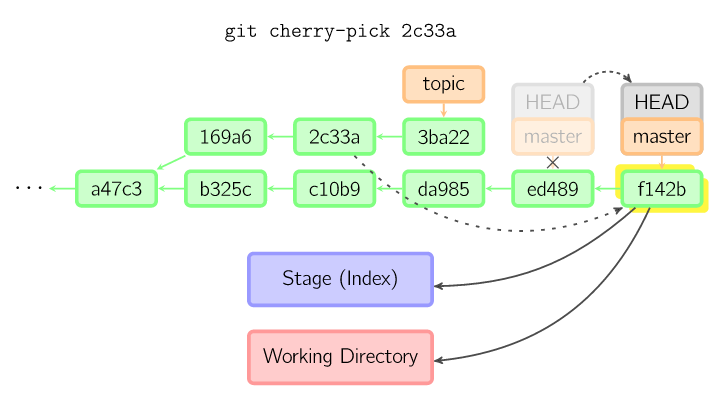


**Commands:**

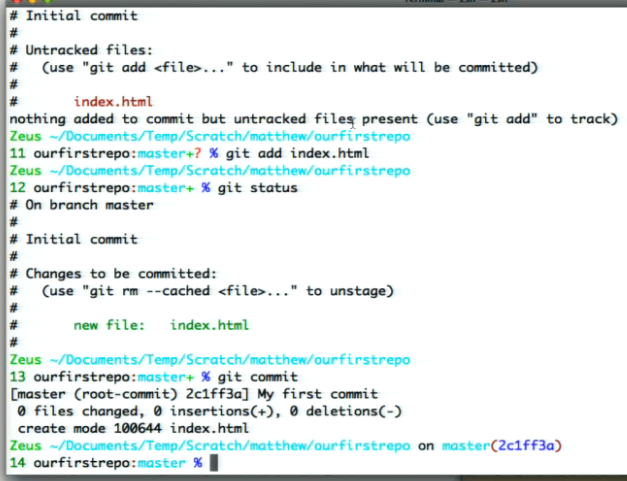
* ***git init*** whateverName
  + Creates a new directory, a staging area, of the name you chose (whateverName) in this case. This is “your shopping cart”; this is where you are temporally holding your files.
  + This staging area is a folder on your computer, this folder contains three hidden files (you can see then by typing *ls –al* in your terminal), you are now inside /whateverName/.git/ folder
    - /.git/ holder all the mechanics, configurations, objects, binary storage, compressed archives, resource of where to reach push/pull
  + When you enter this directory, this is the Version Control
  + This creates a new branch, master.
* ***Touch***filename.datatYouChose
  + Creates an empty file in your staging area (folder)
* ***Git status***
  + Gives you the state of your file, answering all the below questions.
    - What has been added?
    - What has been removed?
    - What branch am I on?
  + From where you are it gives you all the info of your tree, all the way to the leaf nodes. In layman’s term; in the current folder (directory) you are at, it shows you all the files there, and its sub-directories (sub folders), and the content of those sub-directories (sub folders).
* ***Git Add***filename.dataType
  + Copies files (at their current state) from the working directory (untracked files) to the staging area
    - Git is now paying attention to that file
    - Pick the files you want, say add, now they are staged
  + A file will move from being show in red to being shown in green in your terminal
  + Git add .
    - ^^ note the dot ‘.’ After add, that adds all the files to the Version Control System
      * From here down, adds all the files recursively to the VCS
      * If there is nothing new, nothing will be added
    - Git add. –A
      * Add all files recursively, even deleted, removed, or renamed files
      * -A means ALL. Not including –A means only modified files will be added
  + Git add folder1/folder2/filename.dataType
    - Provides a path
* ***Git commit***
  + Saves your file from the staging area to your Version Control system, your repository
    - Permanently burns it into the history of the source tree
    - Makes your changes official
  + A commit message pops up
    - Describe what changes you have done on the file there
    - You can also do git commit –m‘your message here’
      * This automatically commits and creates the message
  + After commiting and commenting, your terminal will display how many files were changed, how many things were inserted (+) or deleted (-), it also shows you were did the file go to
  + Commiting with no message means “I don’t want to continue on this”
  + git commit -a is equivalent to running git add on all filenames that existed in the latest commit, and then running git commit.
  + Only commits to your local repository. Use Push to share on Gitlab
  + When you commit, git creates a new commit object using the files from the stage and sets the parent to the current commit. It then points the current branch to this new commit. In the image below, the current branch is master. Before the command was run, master pointed to ed489. Afterward, a new commit, f0cec, was created, with parent ed489, and then master was moved to the new commit.



* ***Git push*** nameOfAlias nameOfBranch
  + Update remote refs along with associated objects
  + To share the changes you’ve done with others, you need to push your changes to the remote repository.
  + Will attempt to make your [branch] the new [branch] on the [alias] remote. Let's try it by initially pushing our 'master' branch to the new 'github' remote we created earlier.
  + Your changes are now in the HEAD of your local working copy.
  + To send those changes to your remote repository, execute:
    - git push origin master
* ***mkdir*** folderNameHere
  + creates a sub-directory with the name of your choice (folderNameHere in this example)
  + there is no /.git/ folder in any of the sub-directories, only on the root
* **git rm** filename.dataType
  + Removes a file from Version Control. Moved it back to the track stage
  + it will delete the file, unless you unclude –cashed
    - git rm myfile.rb
    - git commit -m "why I deleted the file"
    - git push origin master
* ***git clone***serverHere
  + e.g: git clone [git@github.com:blackbaud/lcrm.git](mailto:git@github.com:blackbaud/lcrm.git)
    - This goes to the repository on GitHub, and copies all of the files there to your computer in a folder. The folder is named by the word between the / and .git, in this case the folder would be called lcrm
  + git cloneserverHere customFolderNameHere
    - e.g: git clone [git@github.com:blackbaud/lcrm.git](mailto:git@github.com:blackbaud/lcrm.git) theLCRMfiles
      * + The folder is named by the word following “.git “, in this case the folder would be called “theLCRMfiles”
* ***Git reset –*** filesName
  + unstages files; that is, it copies files from the latest commit to the stage. Use this command to "undo" a git add files. You can also ***git reset*** to unstage everything.
    - Moves the current branch to another position, and optionally updates the stage and the working directory.
    - It also is used to copy files from the history to the stage without touching the working directory.
    - If a commit is given with no filenames, the current branch is moved to that commit, and then the stage is updated to match this commit. If --hard is given, the working directory is also updated. If --soft is given, neither is updated.
  + Reverting all local changes that have not been committed (oops)
    - Git reset –hard HEAD
* ***git checkout –*** filesName
  + The checkout command is used to copy files from the history (or stage) to the working directory, and to optionally switch branches.
  + When a filename (and/or -p) is given, git copies those files from the given commit to the stage and the working directory.
    - For example, git checkout HEAD~ foo.c copies the file foo.c from the commit called HEAD~ (the parent of the current commit) to the working directory, and also stages it. (If no commit name is given, files are copied from the stage.) Note that the current branch is not changed.
  + Reverting a single local change that has not been committed
    - git checkout HEAD path/to/file
* ***Git diff*** fileOne.dataType fileTwo.DataType
  + Sees the difference between commits. What inside of the file changed from one commit to another?
* ***Git merge*** branchA branch
  + A merge creates a new commit that incorporates changes from other commits.
  + Before merging, the stage must match the current commit.
    - The trivial case is if the other commit is an ancestor of the current commit, in which case nothing is done.
    - The next most simple is if the current commit is an ancestor of the other commit. This results in a fast-forward merge. The reference is simply moved, and then the new commit is checked out.
    - Otherwise, a "real" merge must occur.
* ***Git cherry-pic*** commitName
  + The cherry-pick command "copies" a commit, creating a new commit on the current branch with the same message and patch as another commit.



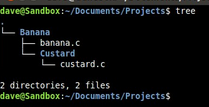
Example of the display after adding, doing status and commit a file



**Other interesting but less used git info**

* **export EDITOR=**fileNameHere.exe
  + Chose which ever text editor you prefer (e.g. notepad++, jEdit), now whenever you commit something, the message box will pop up in the editor of your choice. It gives you an alternative editor instead of the default VI.
* If you create **‘.gitignore’** (touch .gitignore) it creates this file in your current directory. Open it up on a notepad. In there type a patterns of things you want git to ignore
  + \*.dataType
    - Ignore all files of this dataType on your current directory
  + \*\*/\*.dataType
    - Ignore all files of this dataType on your current directory AND all subfolders
  + folderName
    - ignore all files inside this folder (e.g. folderName)
  + You can create other .gitignore files on your subfolders so you can be more precise
  + ‘.gitignore’ is something you track just like any other Version Control file, therefore you have to say ‘git add .gitignore’ and ‘git commit –m’your message’
* Built-in git GUI
  + **gitk**
* Use colorful git output
  + **git config color.ui true**
* Show log on just one line per commit
  + **git config format.pretty oneline**
* Use interactive adding
  + **git add -i**

**TERMINAL basic commands Linux Ubuntu**

* **cd** folderNameHere
  + changes directory
  + cd ..
    - goes up one directory
  + cd ~
    - goes to home directory
  + cd /folder1/folder2/folder3/…/folder/
    - goes to whatever folder you want
* **ls**
  + list the contents inside a folder
  + ls –al
    - list all of the contents (even the hidden protected files)
* **tree**
  + list the contents inside a folder in a tree style
  + 
* **cat** filename.dataType
  + Shows the content of a file
* **mv** filename.dataType1 filename.dataType2
  + Renames the filename.dataType1 to filename.dataType2

**Resources**

Lecture videos: <http://techbus.safaribooksonline.com/video/software-engineering-and-development/version-control/9781449304737>

Info with pictures: <http://marklodato.github.io/visual-git-guide/index-en.html>

More info: <http://blog.osteele.com/posts/2008/05/my-git-workflow>

Pro Git Book: <http://git-scm.com/book>

Blackbaud’s cheat sheet: <https://github.com/blackbaud-qa/qa-readme/blob/master/documentation/git_knowledge/git-cheatsheet.md>

Another cheat sheet: http://ndpsoftware.com/git-cheatsheet.html#loc=workspace;

Git work flow: <http://nvie.com/posts/a-successful-git-branching-model/>

A simple guide to Git: <http://rogerdudler.github.io/git-guide/>

Issues w GitWork Flow: <http://scottchacon.com/2011/08/31/github-flow.html>

Luminate Online Git Documentation: <http://twiki.corp.convio.com/bin/view/Engineering/LuminateOnlineGitDocs>

Interactive git branching exercises: <http://pcottle.github.io/learnGitBranching/>

Git crash course: [http://www.tangowithdjango.com/book/chapters/git.html#git-crash-course](http://www.tangowithdjango.com/book/chapters/git.html" \l "git-crash-course)

More info: <http://gitref.org/remotes/>

More info: <http://think-like-a-git.net/>