Kevin Chen

12/29/16

Git Tutorial Notes

What is Git?

* Understanding Version Control
  + Keeps track of changes, esp. text changes.
  + Version Control System (VCS): manage versions, esp. of source code
  + Source Code Management (SCM): VCS for managing source code. VCS and SCM are almost identical since almost all VCS is SCM.
* The History of Git
  + Source Code Control System (SCCS), 1972. Closed source, free with Unix. Keeps the original document, then keeps a snapshot of all the changes to it between versions.
  + Revision Control System (RCS), 1982. Open source. Improvement of SCCS. Faster because it kept the most recent file and kept changes that allow you access previous versions.
  + Concurrent Versions System (CVS), 1986-1990, open source. Unlike SCCS and RCS, allow you to work on multiple files instead of just a single one. Additionally, more than one user can work on a single file at a single time.
  + Apache Subversion (SVN), 2000, open source. Tracking changes not just to files but also watching to what happen to the files in a directory (e.g. rename, delete)
  + BitKeeper SCM, 2000, closed source, proprietary. Distributed version control.
  + Git: April 2005. Created by Linus Torvalds. Was a replacement for Bitkeeper to manage Linux kernel source code. Distributed version control.
* About distributed version control
  + Different users maintain their own repositories instead of working from a central repository.
  + Changes are stored as “change sets” or “patches”. Tracks changes, not versions. Change sets can be exchanged between repositories.
  + No single master repository; just many working copies, each with their own combination. All repos are considered equal: what matters is what changes they have. By convention, we may have a “master repository”, but that is not intrinsic in Git.
  + No need to communicate with a central server (faster, no network access required, no single failure point). Encourages participation and “forking” of projects (developers can work independently. Submit change sets)
* Who should use Git?
  + Anyone who wants to track edits.
  + Anyone needing to share changes with collaborators
  + Anyone not afraid of command-line tools
  + Programmers and developers
  + Not as useful for tracking non-text files

Installing Git

* Install Git by downloading it online.
  + The command “which git” will show if it is installed on your computer.
* Configuring Git
  + 3 levels of configuration:
  + System: configurations that apply to every user of the computer. Not used very often. Location: Program Files\Git\etc\gitconfig
  + User: apply to a single user. $HOME\.gitconfig. ($HOME is C:\Users\Kevin)
  + Project: my\_project/.git/config
  + Using Git commands to modify configuration information: git config –system (for system), git config –global (for user), git config (for project).
  + git config –global user.name “*firstname lastname*”
  + git config –global user.email “email”
  + git config –list (lists all the configuration settings). List a particular setting: git config *setting.name*
  + ls –la lists all files in the current directory. ls or dir only lists non-hidden files. cat *filename* for Unix displays the content of the file.
  + git config --global core.editor “*name.of.editor* –wl1” tells Git the text editor you are using. –wl1 which tells Git to wait until the editor is done before you keep going, and put the cursor at line one.
  + git config –global color.ui true gitells git to use colors when outputting things to the command line
* Exploring Git auto-completion
  + Windows has auto-completion included already
* Using Git help
  + git help lists some commonly used git commands.
  + git help <*command*> provides information about a specific command
  + When navigating multiple pages for a result, press f and b to go forwards and backwards, and q to quit
  + git help and man git- are the same thing

Getting Started

* Initializing a repository
  + git init makes the current directory a git repository. Adds a .git directory
  + When changing the current directory of git to a folder with a space, you need to use the escape sequence of “\ ” in order to enter a space.
* Understanding where Git files are stored
  + The .git folder, located at the root of the directory being tracked, contains all the tracking information of the directory.
  + Removing this folder gets rid of Git tracking for that directory
  + Don’t modify files inside .git except for the config file, which is the project-level configuration file.
* Performing your first commit
  + For this entire project, add all changes from the working directory to the staging index: git add . (the dot indicates this directory)
  + Tell Git to commit all changes in the staging index (tell Git to put it in permanent memory: git commit –m “*message*”
  + Make changes -> add the changes -> commit changes with a message
* Writing commit messages
  + Short single-line summary. Optionally followed by a blank line and a more complete description (keep each line to less than 72 characters). Present tense, not past tense.
  + Bullet points are usually asterisk/hyphens, can use ticket numbers, can develop shorthand.
  + Clear and descriptive
* Viewing the commit log
  + git log shows the log of commits that have taken place.
  + Contains a unique ID, the author, the date of the commit, and the message.
  + git log –n *number* limits the number of commits returned to *number*
  + git log –since=*date1* –until *date2* lists commits since *date1* and until *date2*
  + git log –author=“*person\_name*” lists commits with a given author.
  + git log –grep=“*expression*” (global regular expression search) does a Ctrl + F of the expression in all the commit messages of all commits

Git Concepts and Architecture

* Exploring the three-trees architecture
  + Two-tree architecture: repository and working copy. Checkout copies from the repository to the working directory. When we are done making changes, we commit those to the repository. Your working copy is out-of-date if someone else commits changes before you finish making your changes.
  + Three-tree architecture: repository, staging index, and working copy. This allows us to make changes to many files but add and commit only a subset of those changed files (in a single commit). Usually checkout copies from the repository directly to the working directory and skip the staging index.
* Git workflow
  + New file: Add file.txt to the working directory. Then use git add file.txt to move it to the staging index. Then use git commit to move file.txt to the repository.
  + Edit file: Make changes to file.txt in the working directory. Then use git add file.txt to move it to the staging index. Then use git commit to edit the file in the repository.
* Using hash values (SHA-1)
  + Git generates a checksum for each change set. A checksum (basically a hash value) is a number generated using a checksum algorithm (basically a hashing algorithm). Same data always equals same checksum.
  + Data integrity is fundamental since chaging data would change checksum.
  + Git uses SHA-1 hash algorithm to create checksums. The algorithm generates a 40-character hexadecimal string.
  + For the snapshot of each commit, Git stores the SHA-1 value of the commit, the SHA-1 value of the previous commit, etc. See the SHA-1 value of a particular commit in the log under the commit entry.
* Working with the HEAD pointer
  + HEAD is a pointer to the “tip” of the current branch in repository. It is the last state of the repository, or what was last checked out. It points to the parent of next commit, or where writing commits takes place.
  + By default we work on the master branch. We can work on new branches (a new set of code). When this happens, HEAD moves to the new branch. We can switch branches by checking out a particular branch, causing HEAD to move to the “tip” of the current branch.
  + You can find a file, HEAD, inside .git. When you open this file (using the command cat), you see it is referenced inside refs/heads/master. If you take a look at master, you can see the current commit HEAD is pointing to.
  + git log HEAD (list commits starting at the head) is the same as git log.

Making Changes to Files

* Adding files
  + git status reports the difference among the working directory, the staging index, and the repository. It also displays what branch you are on. If you see “nothing to commit, working tree clean”, that means there is nothing on the staging index, and there is no difference between the working directory and where the head pointer points to in the repository.
  + After creating new files, typing git status has an untracked files section. This is saying that if you make changes to these, Git won’t know because it is not tracking these. After adding these files to the staging index (using git add *file\_name*.txt), these files will appear under the section “changes to be commited”.
  + Typing “git commit” commits all files from the staging index to the repository. In the output, you get part of the SHA-1 value and some information about what changed.
* Editing files
  + If you enter git status after editing a file, the file is listed as modified under the section “changes not staged for commit”
  + Adding the file is the exact same process (git add <file> and git commit).
* Viewing changes with diff
  + git diff: looks at the difference between the repository (where HEAD is located) and the working directory. (It provides the difference in a line-by-line basis.)
  + git diff <file> looks at the difference of just one file.
* Viewing only staged changes
  + git diff –staged: looks at the difference between the staging index and the repository (where head is located). (git cached does the same thing.)
* Deleting files
  + Method 1: Delete the file in the working directory using Windows Explorer. (git status would list the file as deleted in the section “changes not staged for commit”.) Then type git rm <file> to move the deletion into the staging index. Then type git commit. (Notice the output shows “delete mode”.)
  + Method 2: git rm <file> deletes the file. (Difference from method 1 is that is used a Unix-remove. That is, it permanently removed it, and you can’t find it in the recycle bin.) And then git commit.
* Moving and renaming files
  + Method 1, renaming files: Rename the file from Windows Explorer. (Note that git status shows that the original file has been deleted and a new file has been created with the new name.) Perform git add <file> on the new file and git rm <file> on the deleted file. (Note that git status now realizes that the file has been renamed, as long as the file hasn’t been changed too much.) Then git commit
  + Method 2, renaming files: git mv <original\_filename> <new\_filename> (mv stands for move.) Then git commit.
  + Moving files (method 2): git mv *file*.txt path/to/new/location/*newname*.txt. (*file* and *newname* are the same if you are just moving it.) (Note git status show this as rename instead of moving.) Then git commit.

Using Git with a Real Project

* Introducing the Explore California web site: we are going to use Git in a web site project
* Initializing Git
  + git status on a folder that doesn’t have git results in an error message: Not a git repository (or any of the parent directories): .git
  + After initializing git, git log results in an error: “bad default revision ‘HEAD’”, which basically means HEAD isn’t pointing anywhere special yet.
  + git status shows a list of untracked files, so git add . and git commit.
* Editing the support phone number
  + Sometimes the results of a Git command needs multiple pages to display, so Git uses a Pager to allow you to scroll between the pages. Press f and b to go forward and backwards (or space to go forward). While inside of the pager, you can toggle fold long lines by pressing minus sign + shift + s and then return
  + git diff –color-words <file> puts the difference side by side instead of putting the different lines above each other.
  + git commit –am “*message*” essentially performs git add . and then git commit. Note that this doesn’t work with adding or removing files; only works well with modifying files
* Editing the backpack file name and links
  + (The browser has nothing to do with what is in the repository. It shows what is in the working directory.)
  + (Best to group commits by changes conceptually. Like if you are ready to make a commit with a certain set of changes, but then you noticed another problem and made another set of changes, separate the two sets of changes into two separate commits.)

Undoing Changes

* Undoing working directory changes
  + Useful if you made changes to a set of files in the working directory but want to undo those changes by resetting them to the ones in the repository.
  + git checkout -- <file>: this finds the file in the repository and makes the working directory look like that. (Note that git checkout <item> could work, but the command is also used for branches, so if <item> is the name of a branch as well, it will bring down the branch instead of the file. That’s why “--” is needed, which says to stay on the current branch.)
* Unstaging files
  + git reset HEAD <file> to unstage a file: moves a file from the staging index to the working directory. We’re telling it to look at the HEAD pointer (which points to the last commit) and reset itself to be the same as what that has.
* Amending commits
  + Not easy to edit commits because we’re messing with the data integrity of the Git repository. That causes the SHA-1 value of the changed commit to be different, causing the parent pointer of the next commit to point to an invalid commit. And if we change the parent pointer, that causes the SHA-1 value of that commit to be different, causing the next parent pointer to be messed up. And so on.
  + But we can change the last commit (the one HEAD points to) because nothing points to it yet.
  + git commit --amend -m “*message*” replaces the last commit with a new commit (committing changes in the working directory)
* Retrieving old versions
  + To make changes to older commits, the best advice is to make new commits (commits that undo older commits).
  + To do this, we can manually make those changes and then commit the result.
  + But this is faster: Find the SHA-1 value of the commit that contains the version of the file you want. git checkout *SHA-1\_value* -- *<filename>* (you only need about 10 characters of the SHA-1 value. The double dash means from the current branch). This puts the file in the staging index (and working directory) with the change needed to revert the file. Then do git commit.
* Reverting a commit
  + git revert *SHA-1\_value* will revert the changes in a new commit by doing the exact opposite of all changes made in the commit with the specified SHA value.
  + After typing this command, this will open a text editor, allowing you to edit the message of the change before reverting the commit.
  + Passing in an additional parameter of --n would cause git to stage the commit instead of committing it automatically.
  + But if changes are complicated (e.g. in the meantime, other things have changed, such as files moving), then you need to do merging (we will talk about this later)
* Using reset to undo commits
  + git reset undoes multiple commits. Use with caution!
  + We move the HEAD pointer back to a previous commit, and rewrite what came after it. There are three options for resetting:
  + Soft reset does not change staging index or working directory; it just moves the HEAD pointer, so the repository is set to an earlier version but the working directory and staging index are at the latest version.
  + Mixed reset (default) moves the HEAD pointer and changes the staging index to match the repository, but it does not change the working directory. So both the staging index and the repository are at the earlier version.
  + Hard reset changes the staging index and the working directory to match the repository, which points to the earlier version. Any changes after are completely obliterated.
* Demonstrating a soft reset
  + When doing resets, always a good idea to copy and paste the most recent few entries of the log beforehand.
  + git reset --soft *SHA\_value* does a soft reset.
  + Soft reset does not have to go back in time; it can also go forward because the later commits are still there (we only moved the HEAD pointer)
* Demonstrating a mixed reset
  + git reset --mixed *SHA\_value* does a mixed reset.
  + Git shows that it unstaged changes.
  + Note that git reset is also used for unstaging files. (git reset HEAD <file>) Unstaging files is the same as resetting: it uses the HEAD instead of the SHA directly, and then tells which file to pull down from HEAD to make the staging index match it. Similarly, in a mixed reset, we move the HEAD pointer and make the staging index match it.
  + git reset is the same as git reset –mixed
  + Similar to soft reset, we can go forward using a mixed reset because the later commits are still there
* Demonstrating a hard reset
  + Can’t remake commits. Only use when things in your working directory have really gotten out of hand.
  + git reset --hard *SHA\_value* performs a hard reset
  + Even though changes are no longer in the working directory and staging index, we can go forward using a hard reset because the later commits are still there.
* Removing untracked files
  + We can delete them one by one, but there is an easier way.
  + git clean –n is a test run. It will tell us the files it would remove.
  + git clean –f actually removes the files (permanent delete…not in recycle bin). (Note it only removes files in the working directory, not in the staging index.)

Ignoring Files

* Using .gitignore files
  + We want to tell git to ignore certain files to ignore.
  + Add a special file in the root of our directory named .gitignore to provide a git with a certain set of rules of which files it ought to ignore.
  + Can specify individual files, use basic regular expressions, negate expressions with !, ignore all files in a directory with a trailing slash (e.g. assets/videos ignores all files in the videos folder). List each rule on its own separate line. Comments begin with #, blank lines are skipped.
  + (On Unix, you can create a new file by typing in: nano *nameoffile*)
  + Basic regular expressions: asterisk (\*) means any set of characters for files. (So \*.php would ignore all files ending in .php. Note that this only applies to files, so log/\*.log would not cause git to ignore log/archive/access.log). [0-9] means any number.
  + Add .gitignore file to the repository (git add and git commit).
* Understanding what to ignore
  + Compiled source code (but don’t ignore the uncompiled code)
  + Packages and compressed files. You’re not using these files in the project itself.
  + Logs and databases (files that change often)
  + Operating system generated files
  + User-uploaded assets (images, PDFs, videos)
  + <https://help.github.com/articles/ignoring-files> is a help article containing a general list of what to ignore. <https://github.com/github/gitignore> is a GitHub repository which has some great ideas of what to ignore for specific types of projects.
* Globally ignoring files
  + You might be finding yourself ignoring the same type of files in all of your projects in each of your different Git repositories
  + Configure Git to globally ignore files (ignore files in all repositories). This is user-specific instead of repository-specific.
  + To configure git, first create the global ignore file (good idea to put it in C:/Users/*YourName*, and call the file .gitignore\_global). Then type in git config --global core.excludesfile /path/to/file/*filename*. (/path/to/file/*filename* is ~/.gitignore\_global if you used above location and file name)
* Ignoring tracked files
  + Git will not ignore a file that was already tracked before a rule was added to the .gitignore file that tells it that it ought to ignore the file.
  + To get Git to untrack a file, you can remove it from the repository (git rm <file>). But if you don’t want to remove this version, then remove it with the cached version: git rm --cached <file>. This causes git to remove it just from the staging index, not the repository.
* Tracking empty directories
  + Git does not track empty directories because it is a file-tracking system. (If you do ls –la on an empty directory, you get . and .., which are not files. In Unix, this is a reference ot the current directory and a reference to the parent directory.)
  + The trick is track empty directories to put an empty file in them. By convention, it is called .gitignore or .gitkeep (second used more often). You can do this manually or using the command touch path/to/empty/folder/.gitkeep

Navigating the Commit Tree

* Referencing commits
  + Tree-ish means something that references part of the tree. (-ish indicates the “something” can vary widely). Most simply, a tree-ish can be a commit. Can be referenced using a full SHA-1 hash, a short SHA-1 hash (at least 4 chars and so it is unambiguous. 8-10 characters is usually safe), HEAD pointer, branch reference, tag reference, ancestry (e.g. parent commit).
  + Ancestry: parent commit: put a caret after the reference (e.g. HEAD^ is the parent of HEAD) OR put a tilde and a number indicating how many times you want to go up (e.g. HEAD~1 references the parent of HEAD. By default, it is one, so you can leave it off in this case: HEAD~). Grandparent commit: put two carets after the reference or replace the 1 after the tilde with a 2. Add more carets or increase the number after the tilde to go up even more levels.
* Explore tree listings
  + git ls-tree <tree-ish> lists out a tree starting at the tree-ish. Example: pass in HEAD. If HEAD is currently in master, then passing in master does the same thing.
  + You can also pass in a file path after that to list the tree within the directory specified by the file path. Note that the file path is included inside each entry in the tree listing.
  + You can look at the tree in its previous state by passing in HEAD^ or master^.
  + Each entry is a tree or a blob. A blob is a file. If it’s not a blob, then it’s a directory (a directory)
  + A tree has an object number (a SHA). You can explore that tree by passing its object number where the <tree-ish> is needed. Note that the file path to get to each entry in the tree listing is not listed.
* Getting more from the commit log
  + git log --oneline gives a one-line list of what is in the log file.
  + git log –*number* limits the number of items displayed in the log to *number*
  + git log --since=*date1* –until *date2* lists commits since *date1* and until *date2*. Can use after instead of since and before instead of until. Can specify time period instead of date (e.g. “2 weeks ago” or “3 days ago” or “2.weeks” or “3.days”)
  + Look above for git log --author=“*name*” and git log –grep=“*sometext*”
  + git log *SHA1*..*SHA2* gets the range of logs from the commit with *SHA1*till the commit with *SHA2*. Leaving *SHA2* blank will list it till the end.
  + git log <file> will list all entries concerning a particular file.
  + git lop –p gets us more information about each log entry. (It gets us the diff between the logs.)
  + git log --stat --summary gives us statistics and a summary about each change.
  + git log --format=oneline is the same as git log oneline except it returns the full SHA for each entry instead of just a partial SHA. Instead of oneline, can pass in short, medium (this is the default we are already seeing), full, fuller (gives more and more info for each entry). Can also pass in email (good format for emailing the log) or raw (shows raw data of the log).
  + git log --graph shows a graph of each one of our commits.
  + A good combination is git log --oneline --graph --all --decorate
* Viewing commits
  + git show *SHA* provides details about the commit specified by the *SHA*.
  + git show --oneline SHA gives us our oneline format for the top part of the output
  + Instead of passing in commits, we can pass in blobs, trees, or tags (do this by passing in the object number as shown when from the output of git ls-tree <treeish>). You must pass in a tree-ish; can’t pass in a file directly, for example.
* Comparing commits
  + We are comparing the directories that the commits reference, not the commit snapshot or the changes that were made with that commit.
  + git diff <*treeish*> returns the difference between the directory at that point in time and the current working directory.
  + git diff <*treeish*> <file> returns the difference concerning only the specified file.
  + git diff <*treeish1*>..<*treeish2*> returns the difference between the two directories referenced by the two treeishes. Can add a file to return the difference concerning only the specified file.
  + Passing in --stat and --summary displays stats and a summary of what was changed.
  + Passing in --b or --ignore-space-change ignores changes to whitespace.
  + Passing in --w or --ignore-all-space ignores every single change that could be made to space.

Branching

* Branching overview
  + Branches are cheap (easy to try new ideas, can isolate features or sections of work).
  + We have one working directory. But when we switch branches, Git is going to make all the files in the working directory match the branch we are switching to.
  + Merge branches to incorporate a set of commits on one branch to another branch. Use a commit to merge branches.
  + Remember HEAD points to the last commit of the current branch. Immediately after you make a new branch, HEAD still points to the last commit of the original branch because there are no commits in the new branch. Only once there are commits in the new branch can HEAD point to the last commit in the new branch.
  + Switching between branches causes HEAD to switch between the last commits in the two branches.
* Viewing and Creating Branch
  + git branch returns all the branches in our local repository. The asterisk before a branch name indicates that that is the current branch or the currently checked out branch.
  + git branch *new\_branch* creates a new branch by the name *new\_branch*. There can’t be any spaces, and stay away from punctuation.
  + Remember .git/HEAD stores where the HEAD is currently pointing to. .git/refs/heads contains references of where HEAD would point for each branch checked out.
* Switching branches
  + git checkout *nameofbranch* checks out the specified branch.
  + After switching, cat .git/HEAD confirms that HEAD points to the new branch instead.
* Creating and switching branches
  + git checkout –b *nameofbranch* creates a branch with the specified name and switches to it at the same time.
* Switching branches with uncommitted changes
  + The working directory must be (mostly) clean in order to switch (cannot have uncommitted changes) because you would lose the changes you just made.
  + Three choices: scrap the changes by checking out the file, commit the changes to the current branch, and stash the changes (we’ll talk about this later)
  + It must just be clean enough so that there are no conflicts. (Example: adding a new file is okay because the new file would still be there even after switching branches.)
* Comparing branches
  + git diff *branch1*..*branch2* compares the two branches. (*branch1* should be the old state and *branch2* should be the newer state.)
  + git diff --color-words *branch1*..*branch2* shows each diff next to each other all on one line instead of having the two lines above each other.
  + git branch --merged returns the branches that are contained by the current branch (or the branches which everything in it has been merged into the current branch). We can delete branches returned by this command and the current branch wouldn’t be affected by it. What this command does is it looks at each branch and sees if the current branch has the final commit of that branch.
* Renaming branches
  + git branch –m *oldname* *newname*. (Just like renaming files, we actually move the branch into the same place instead of renaming it directly.) Can replace –m with --move
* Deleting branches
  + git branch -d *branchtodelete* deletes the specified branch
  + Git has a few checks to make sure that you don’t do something stupid with this. First one is that you can’t delete a branch that is currently checked out. Second one is that if there are changes in the branch you want to delete from the branch you are currently on, git will refuse to delete the branch unless you pass in –D instead of –d.
* Configuring the command prompt to show the branch
  + On Windows you should already be seeing the current branch by default.
  + But if you aren’t: echo $PS1 to show what it’s using currently. (\_\_git\_ps1 shows the current branch.) Create a new text file and type inside it: export PS1=‘\w$(\_\_git\_ps1 “(%s)”) > ’. Save it as .bash\_profile inside C:/Users/*YourName*. Then enter the command: source ~/.bash\_profile.

Merging Branches

* Merging code
  + Switch to the receiver branch.
  + Then type in the command git merge *mergedbranch*.
  + You only want to merge code with a clean working directory (don’t want uncommitted changes).
* Using fast-forward merge vs. true merge
  + If you created a new branch, made changes to the new branch, and merged it back to the original branch without any changes to the original branch, Git will do a fast-forward merge. Git tells you it did a fast-forward merge after you entered the command git merge *mergedbranch*.
  + What Git did was it looked up the chain of commits from the head of the new branch all the way to the first commit. If, along the way, the commit Git is currently looking at is the head of the original branch, then Git knows it can do a fast-forward merge. So it sets HEAD of the original branch to be the tip of the new branch. There was no need to make a new commit.
  + git merge --no-off *branch* forces Git to make a true merge (a merge commit).
  + git merge --ff-only *branch* makes Git perform a merge only if it can do a fast-forward merge.
  + The process to perform a true merge is the same. (Git has multiple strategies to do this. Recursive strategy is typically the one you will see.)
* Merging conflicts
  + Merge conflicts occur if there are two changes to the same line or set of lines in two different commits between the two branches.
  + After trying to merge branches and getting a conflict, you notice you are on the branch: *originalbranch*|MERGING. git status shows that you are on unmerged paths and to use git add/rm <file> as appropriate to mark resolution. Open the file with conflicting merges. Git marked those problems with <<<<<<<HEAD followed by several equals signs (the text in between these two marks is the text in the original branch). Next is some text followed by the >>>>>>>*newbranch*; the text here is in the *newbranch*. If there are multiple merge conflicts, Git will separate them into sections and mark each of the sections as described above.
* Resolving merge conflicts
  + Three choices: abort merge, resolve conflicts manually (this is done most of time), use a merge tool.
  + git merge --abort aborts the merge.
  + Resolving the conflicts manually: Edit the version you want to keep. Once you are done editing, delete the other version and all the markers. Then do git add and git commit with the conflicting file. (You can use the default commit message by not specifying –m.)
  + Using a merge tool: git mergetool --tool=“*nameoftool*”. You can see the list of tools by just typing in git mergetool.
* Exploring strategies to reduce merge conflicts
  + Keep lines short makes it easier to spot where the conflicts actually occur.
  + Keep commits small and focused.
  + Beware stray edits to whitespace.
  + Try to merge often.
  + Track changes to master (keeping bringing in changes from master so that your branch doesn’t get too far out of sync).

Stashing Changes

* Saving changes in the stash
  + The stash is where you can store changes temporarily without having to commit them to the repository.
  + Remember that we can’t switch branches if you have uncommitted changes. Stashing is useful to deal with this this.
  + git stash save “*message*” saves the changes made in the current branch into the stash.
  + What it actually did was it ran git reset hard HEAD.
  + If you have untracked files, you can include these as well by using the include, untracked option, but normally this is used for (tracked) files in the working directory.
* Viewing stashed changes
  + git stash list shows us a list of things in the stash.
  + You can put your changes in the stash and then switch changes. This allows you apply a set of changes to a different branch.
  + git stash show *changeinstash* gives us more info (the diff stat) about the files changed in the *changeinstash*. *changeinstash* refers to the “commit” inside the stash, which is referenced by the value obtained at the beginning of the output generated from git stash list. (Example: stash@{0})
  + To see even more information, type git stash show -p *changeinstash*.
* Retrieving stashed changes
  + Can result in conflicts just like merge, and it resolves them just like it does with merge conflicts.
  + “git stash pop” and “git stash apply” pull what’s in the stash out and put it in the working directory. The difference is that stash pop also removes it from the stash as well. So git stash apply is useful for applying the changes in the stash multiple times (and possibly in multiple branches).
  + You also need to specify which item to pop out using the reference (e.g. git stash pop stash@{0}). By default, it will pull the first item.
* Deleting stashed changes
  + git stash drop *changeinstash* deletes the “commit” in the stash.
  + git stash clear deletes all items in the stash.

Remotes

* Using local and remote repositories
  + We collaborate with others by uploading our changes to and downloading changes from a remote server, which is a Git repository.
  + We push our changes to the remote server, causing the remote server to have the same branch. Others can see our changes now. Additionally, Git makes another branch on our local computer typically called origin/*branchname*, which references the remote server branch and always tries to stay in sync with that.
  + When others make a contribution to the remote server, we need to pull those changes by doing what’s called a fetch. This causes those changes to come into our origin/*branchname* branch, but it doesn’t bring it into the *branchname* branch until we do a merge.
  + The Origin/*branchname* branch doesn’t create duplicate objects of the *branchname* branch. It creates a separate reference for it and moves it appropriately.
* Setting up a GitHub account
  + GitHub is the most popular Git host. Many great features.
  + Have the option to initialize the repository with README file (which is often done as part of the first step of creating a repository), and adding a .gitignore file.
  + Then they give instructions on creating a new repository or pushing an existing repository. The steps to push an existing repository are: git remote add *remotealias* *RepoURL* and then git push –u *remotealias* master. (For *RepoURL*, use HTTP instead of SSH for now. SSH is only if you are logged in via SSH instead of via HTTP.)
* Adding a remote repository
  + git remote shows us all the remotes that we know about.
  + git remote add <alias> <url>. By convention, “origin” is used for the alias. URL is the URL of the remote repository.
  + git remote –v shows the URL’s we use to fetch from and push to each remote.
  + cat .git/config will show you the remote repositories.
  + git remote rm *remoterepo* will remove the specified remote.
* Creating a remote branch
  + git push –u <alias> <branch> (e.g. git push –u origin master).
  + cat .git/config will show you that there is a branch definition for the branch you created a remote for. The definition contains the remote (origin by default) and a reference for what Git will use when doing merges.
  + It stores the above information in .git/refs/remotes. Inside this folder, you would see a folder by the name of the remote (origin by default). Inside that folder, you will see a file by the name of the branch. If you open that, it is a reference to a SHA.
  + git branch –r shows you all remote branches (or remote-tracking branches because all remote branches are being tracked).
  + git branch –a will show you both local and remote branches.
* Cloning a remote repository
  + We want to be able to do the opposite of what we have just done, which was taking a local repository and pushing it to a remote. Now we want to go with a remote repository and pull it down to have a local copy to work with.
  + Get the URL of the remote repository by selecting “Clone or download”
  + git clone *RepoURL* clones the remote repository.
  + git clone *RepoURL name* allows you to name the directory of the remote repo.
* Tracking remote branches
  + The process of tracking is where you have one branch that stays rather closely in sync with the progress of another branch. Example: we regularly pull updates from master into a separate branch.
  + The –u option in creating a remote branch (git push –u <alias> <branch>) caused the branch to be a tracking branch. This branch tracks what is happening in GitHub.
  + If you have a nontracking branch that you want to be tracking, then you have three options. First is the edit the .git/config file directly. Second is use the command git config branch.*nontrackingbranch*.remote *remotealias* (e.g. git config brach.non\_tracking.remote origin) and then git config branch.*nontrackingbranch*.merge refs/heads/master. Third is to enter the command git branch --set-upstream *nontrackingbranch* *remotealias* /*nontrackingbranch*.
* Pushing changes to a remote repository
  + git push *remotealias* *localbranch*. But if you are on *localbranch* and *localbranch* is tracking *remotealias*, then you can just do git push.
  + You can see that commit on GitHub.
* Fetching changes from a remote repository
  + *remotealias*/*localbranch* (e.g. origin/master) does not automatically update itself when there is a new change to the remote repository. It only does that when we tell Git that we want to do a sync between the two.
  + git fetch *remotealias* does this sync between the remote repository and *remotealias/localbranch* (e.g. origin/master). (If you have only one remote repository, *remotealias* can be omitted.) Note that *localbranch* is still not synced up yet.
  + Always fetch before you work. Fetch before you push. Fetch often.
* Merging in fetched changes
  + Note that *remotealias*/*localbranch* is almost just like a normal branch (only difference is that we can’t check them out because Git is control of them). So merging fetched changes is just like merging *remotebranch*/*localbranch* with *localbranch*.
  + So we do git merge *remotealias/localbranch*.
  + git pull = git fetch + git merge. More convenient.
* Checking out remote branches
  + git branch *newbranch* *remotealias*/*originalbranch* creates a duplicate of a branch (whose name is specified using *newbranch*) from the remote alias. (Example: git branch my\_feature origin/new\_feature creates a duplicate branch of origin/new\_feature whose name is my\_feature.) Good idea to have *newbranch* the same name as *originalbranch*.
  + Git will set it up so that *newbranch* is tracking *remotealias*/*originalbranch*.
* Pushing to an updated remote branch
  + Git cannot push your changes to the remote repository if there have been changes to the remote repository since you last fetched from it.
  + To resolve this, you have to fetch and merge the changes from the remote repository before you can push your changes.
* Deleting a remote branch
  + Two ways to delete a remote branch:
  + If you want to delete the remote branch, remotes/*remotealias*/*branchname*, then enter git push *remotealias* :*branchname*. This is the logic behind the colon: when you did git push *remotealias* *localbranch*, it was actually a shortcut for git push *remotealias* *localbranch*:*localbranch*, which says push to *remotealias* my *localbranch* to the remote branch called *localbranch*. But when you do git push *remotealias* :*branchname*, you are pushing to *remotealias* nothing up to the branch *branchname*.
  + git push origin --delete *localbranch* deletes the remote branch as well.
* Enabling collaboration
  + To allow other GitHub users to work on your project, go to the project on github.com. Then go to settings, then collaborators, and then add usernames of the users you want to allow to collaborate.
  + For open source projects, not everyone can make commits to it (or else it would be hectic, insecure, etc.) In order to make edits to these projects, you first need to fork it. This will make your own version of the project on your own GitHub repository, and you have write access to this one. Make edits as you would normally and commit them. Then go back to the GitHub page for the main project, and issue a Pull Request. Then the people who have write access to the main project would review your branch and see if they will incorporate those changes.
* A collaboration workflow
  + Do git fetch at the beginning. Do merges as necessary if new changes pop up.
  + Then make your changes and add and commit those changes. Then do git fetch again in case new changes have been made to the remote repository. Do merges as necessary if new changes pop up. Then push your changes to the remote repo.
  + When you are done with a branch (the feature that the new branch adds has been finished), first do git fetch and merge origin/master with your local master branch if needed. Then merge the other branch with the master branch, and do git push.

Tools and Next Steps

* Setting up aliases for common commands
  + Put the aliases in the user-specific configuration.
  + git config --global alias.*alias* “*originalcommand*”. (Example: git config –global alias.st “status”.) Double quote are optional if *originalcommand* does not have any spaces in it.
  + You can see alias entries inside ~/.gitconfig inside the [alias] section.
  + Standard aliases (you don’t have to use these, but these are used quite commonly among developers): st for status, co for checkout, ci for commit, br for branch, df for diff, dfs for diff --staged, dfc for diff --cached.
  + We used git log --graph --decorate --oneline --abrev-commit --all quite often. Can replace that with git logg.
* Using SSH keys for remote login
  + Every time we wanted to communicate with the remote repository, we had to enter our GitHub credentials. Painful.
  + Two ways to solve it: keychain program to store your username and password. Look at <https://help.github.com/articles/set-up-git#password-caching> to see how to set it up (only for Mac)
  + Other way: we have a little bit of code on a computer and put a complementary part of the code on the GitHub server. When I make a request, Git automatically sends that bit of code along with the request, which authenticates me. Go to <https://help.github.com/articles/generating-ssh-keys> to see how to set it up.
  + Instead of using the http URL format, you must use the SSH format instead.
* Exploring integrated development environments (IDEs)
  + It’s very possible that the IDE tool you are using has some kind of Git configuration already inside it.
  + If your IDE does not have it, there may be a plug-in that adds that functionality.
  + These programs have Git integration (uncomprehensive list): Vim, Emacs, TextMate, Eclipse, Visual Studio, IntelliJ, RubyMine, PHPStorm.
* Exploring graphical user interfaces
  + GitWeb (you need to set up a web server to host it)
  + For Mac: GitX, GitHub, SourceTree, Tower, SmartGit, Gitbox
  + For Windows: TortoiseGit, GitHub, Git Extensions, SmartGit.
  + A good list is maintained here: <https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools>
* Understanding Git hosting
  + We are talking about hosting remote repositories that are hosted somewhere where multiple users can all have access to that one remote.
  + We need a Git server to set up and handle requests. We can have a hosting company do this (we explored GitHub. There is also Bitbucket and Gitorious. They do everything for you for free or a small price.) or we can self-host.
  + Git Self-Hosting: often done for people who wants to keep their code inside a firewall (e.g. gov. agencies). Main way use to be using a software called Gitosis, but development recently stopped on it. Gitolite is in active development.