CONFIGURATION

git config –global user.name “Jose Iriarte”

git config –global user.email “josemariairiarte@gmail.com”
these are important to set so that we can tell who committed what

git config –list
see the changes you've made to configuration

git config user.name
see what the user name is for that project

git config –global core.editor “notepad.exe”
sets notepad to be the core editor for the global user

git config –global color.ui true
sets the user interface to use colors in git

git help
brings the man pages for git

git help [command]
launches the man pages for that command

INITIALIZING GIT ON A PROJECT

git init
to be used in your projects root folder, creates the git directory where git will track all the changes for that project

git add .
Used in the root project directory, adds everything in the current directory to git memory

git commit -m “initial commit”
Commits changes

git log
shows a log of all past commits for the projects

git log -n 1
shows a log of only the last change, changing the number, changes the number of logged commits it can show

git log –-since=2012-05-14
shows a list of commits since that date

git log –—until=2012-05-14

git log –—author=”Jose”
shows a list of commits done by author with name of Jose

git log –—grep=”Init”
shows a list of commits matching the regex “Init”, more complex regular expressions can be used to search through different commits.

HEAD
head is a pointer that points to the last commit of the current branch. To know where head points to at any given moment we go into .git folder in our projects base directory and do
cat HEAD
we follow the directory it points to and we cat the file to see what the hash value (the identifier) for the commit is. If we do

git log
we can see which of the commits we've made has that hash value, that's where our head is pointing to. Because head points to the last commit of the current branch, when we do git log, the first commit we'll see (the last we've made) is where head points to.

git add filename.extension

will add a file to the staging area, ready to be committed into the repository
with

git commit -m “message”

you can undo that is, take out of the staging area back to the working directory
with

git reset filename.extension

GIT DIFF

git diff
shows all changes additions and subtractions between files in the repository vs working directory, it lists files one at time comparing repository and working directory versions. The output shows changes by line numbers between @@ signs and uses minuses to show what's been deleted and + signs to show what's been added.

git diff filename.txt
shows changes additions and subtractions in the filename.txt file between the repository version and the working directory version.

git diff -–staged
compares files in the staging area to files in the repository

git diff –-color-words file.txt
color the changes

DELETING FILES

1. delete the file manually from your project folder
2. then do
git rm filename.txt
git commit -m “file removed”

this deletes file in repository if previously committed

OR

1. do
git rm filename.txt

this deletes file completely from your working directory and puts in into staging

git commit -m “removed 2nd file”

this deletes file in repository if previously commited

MOVE AND RENAME FILES

TWO WAYS

1. DO IT OURSELVES AND TELL GIT ABOUT IT

you can rename a file from windows
from first_file.txt to primary_file.txt

git status
tells us that a file was deleted and a new file is untracked

git add primary_file.txt
git rm first_file.txt
git status
git notices that files have been renamed

OR

2. DO IT FROM GIT

git mv second_file.txt secondary_file.txt
just like in unix a move and a rename is the same thing

git status
shows you the rename

TO MOVE A FILE

git mv third_file.txt directory/third_file.txt

OTHER

git commit -am “changed 24 hour support number”
the “a” adds and commits, it's a short cut
it grabs everything in your working directory
for new files and deleted files, it doesn't work well

git add folder/
adds everything inside the folder directory

always make commits of related changes, that is commits should have some sort of unity logic, for example, commit changes relating to style in one commit, and fix a bug on a different commit

UNDOING CHANGES

To working directory:
we want the repository version back and replace what i have in my working dir.

git checkout -– index.html

the double dash indicates to git we don't want to get the whole branch, use it to avoid confusion between files and folders AND branches that are named the same

Unstaging documents – taking them out of the staging area (which we stage with git add). You will use this most when you are trying to assemble a coherent commit.

git reset HEAD file.txt

HEAD indicates git to go to the pointer and reset that file

Ammendings Commits – Undoing commits that we've made. We can change the last commit, the one the head points to.

First we stage whatever changes we want to ammend to the last commit (with git add). The we do

git commit –-amend -m “Repeat the message from the last commit here”

if you choose a different message the previous message in the commit in the repository will be overwritten.

An alternative to ammending older commits

first, get from the repository the file you want to get

git checkout 83iik393k3 –- filetogetfromthecommit.txt

where the numbers and letters are a copy paste of the first 10 or so digits from the commit which has the file you want to get (you can see she code or sha from the git log command). Now it puts it into the stage it area. You can then commit it to the last commit.

Undo changes from commit definitevely

Anything added will be deleted and so forth, undo what was done, will be a mirror image of what we have in the commited

git revert 3i3l3l3k3k3

will give you the option to edit the commit message. It will automatically make a new commit that reverts the old commited

this is good to make simple changes, but if files have moved or if they have been renamed. In this case we will use the merge command

Undoing mulitiple commits – dangerous commits

it will allow us to specify where the head pointer should point to and that's where you are going to start recording from now on. It's like rewinding to a certain point in time – a commit. And overwrite whatever comes afterwards.

git reset –-soft (move the pointer and do nothing else)
git reset –-mixed (moves the head pointer and changes the staging index to match the repository, the working directory is not touched)
git reset –-hard (move the pointer of the repository and deletes everything after that commit in the staging area AND your working directory too!) this is very dangerous

examples

git reset –-soft 3k3jh3h3h3h3
in working directory and staging area we still have the file from the last commit before the reset.

We can commit these changes, or if we made note (in notepad) of the sha of the last commit before the reset and do git reset –-soft 3j3lin3ij3 and undo the first reset

git reset –-mixed 3k3k3k3k3k3
almost as safe as the soft reset, the changes from later commits still exist but only in the working directory, not in the staging area

We can always go back to the last commit before the reset if we made the note of the sha on notepad

git reset –-hard 39309393jjj
the most destructive: moves the head to the commit we indicate, throws out everything that comes after that commit, deleting everything in the repo, the staging area and the working directory.

Those old commits are still there however, we can put the shas of later commits with git reset –-hard 3839393jjjj and undo the first git reset hard if it was a mistake.

Removing Untracked files from working directory

git clean -n
is a test of what it would remove and reports

git clean -f
forces it to run and throw away anything that is not in our staging directory or repository. Use git reset head file.txt to unstage a file and then use git clean -f to permanently delete them (they wont go to the trash)

GIT IGNORE

We need to create a special file in the root of the working directory .gitignore, this file will tell git which files use for commit and which to ignore.

For example:

tempfile.txt

will ignore tempfile.txt

In this file we can use basic regexs:
* ? [aeiou] [0-9] !

Ignore all files in a directory with a trailing slash

assets/videos/

comment lines begin with #, blank lines are skipped

1. create the file using nano or using notepad and save the file as .gitignore, no extension

2. we want to commit the .gitignore file to the repository.

3. when we do git status we will see how the files we included in .gitignore in our working directory are being ignored by git

Ignoring files globally – we want to ignore files for many projects, we need to configure git to ignore files in all repositories. These ignorations will be user specific not repository or project specific.

1. we need to create the file with nano or notepad and we create a file called .gitignore_global (we can call it whatever we want as long as it is in the User root folder)

2. git config –-global core.excludesfile ~/.gitignore_global

Ignore tracked filed (files in your staging area)

remember that even though you can do this, you can ignore only files you stage or track after you told it to ignore it

1. nano .gitignore
2. add the file or file type you want to ignore

(
to have a file stop being tracked we do
git rm –-cached tempfile.txt
)

Git does not track empty directories
To keep track on empty directories, add a tiny file to it called .gitkeep, you don't even have to put any content on .gitkeep (it can be called anything but convention says that we call it .gitkeep)

touch assets/pdfs/.gitkeep

TREE-ISH and Navigating the Git Tree

a reference to a commit (e.g., full sha-1 hash, short sha-1 hash (at least 4 characters, head pointer, branch reference, and the ancestry)

reference to find parent

HEAD~1 (tilde is going back (1) generation)
acf87504~3 (going back three generations from that sha)

to list out a tree
git ls-tree HEAD

this is current set of files

git ls-tree master
git ls-tree master assets/
shows the files that are in that directory

a blob is a file
a tree is a directory

GIT LOG

git log –-oneline
gives us a one line list of whats in our log file

git -log –-oneline -3
shows just three commits back

git log –-since=”2012-06-12”
git log –-until=”2 weeks ago”

git log --author=”Jose”

git log –-grep=”temp”
can search regexes on commit messages and file names

git log 38409..ac4444 –-oneline
shows the range of commits from the first to the second

git log 38409.. index.html
shows the commits of index from the commit with hash 38049

git log –-stat -–summary
gives you statistics of commits and summaries

git log –-format=full
git log –-format=email

git log –-graph
shows a a graph showing branches and merges

LOOKING AT A SPECIFIC COMMIT

git show 38803kkh

shows author, date, commmit message, diff and other information about the commit
shows different things for trees and blobs and other refrences

git show –-format=oneline HEAD~2

COMPARING COMMITS

comparing the state of all files in a repository

git diff cdae0ed
returns all differences between directory at that point in time and the current directory

git diff cdae0ed tours.html
makes comparison for that file

git diff cf899d..cd99d0
compares the two commits

git diff -b 393939..HEAD
ignores changes to white space changes in the comparison

git diff -w 393939..HEAD
ignores all spaces

BRANCHES AND BRANCHING

create a branch and if it works out you can merge with the master branch, if changes dont work you can throw it out

create a branch to create a different feature, once everybody's happy with it, merge it

everytime we switch branches, git will swap working directory files to matching

HEAD will always point to the tip (last commit) of whatever branch we are in

CREATING BRANCHES

git branch
shows the branches we have in our machine
the asterisk shows the current branch
when we add new branches it adds references in the HEAD folder

git branch new_feature

creates branch titled new feature, but HEAD still points to tip of current branch not the one just created

cat .git/HEAD
see where the head points, what branch

git checkout new_feature
switches to branch new_feature, but to the last commit you made

git commit -am “Modified Title of Index.html”
adds a commit to the new branch

We could create a branch and checkit out (switch to it) at the same time:

git checkout -b shorten_title
creates new branch called shorten_title and switches to it

branches will be created off of the branch at which we are when we create a new branch

if your working directory is not clean (have unstaged changes), git won't let you switch branches, why, because if git switched branches with unsataged changes they would all be lost in that working directory.

COMPARING BRANCHES

git diff master..new_feature
compares master to new feature branches

git diff master..shorten_title –-color--words
compares master to shorten_title branches with colored differences

To find whether one branche completely contains another branches:
git branch –merged
shows which branches are completely contained in the current branch

RENAME BRANCHES

git branch –-move new_feature seo_title
git branch -m new_feature seo_title
either command will rename new_feature branch to seo_title

DELETE BRANCHES

git branch -d seo_title
git branch –-delete seo_title
deltes seo_title branch

you cannot delete the branch you are currently on, git won't let you, it's a security measure

git will ask you to use -D instead of -d to delete all changes you've made to a branch which has not been fully merged.

Git branch -D branch_to_delete

always use the -d option to make sure git warns you properly.

CONFIGURE COMMAND PROMPT TO TELL US WHICH BRANCH WE'RE ON

install the git completion script in the configuration

Linux:

within a git directory we type

__git_ps1

echo $PS1
shows us the unix prompt

export PS1='\W $(__git_ps1 “(%s)”) > '
it gives us a new command prompt showing the branch

in order to keep it beyond the current session:

nano .bash_profile

#comment what we dont want to execute
paste the code

close terminal, open it again, it should be available

Windows

__git_ps1

open notepad

and write the command on a blank file

export PS1='\W $(__git_ps1 “(%s)”) > '

save it in the user directory and call it .bash_profile
do not put an extension in the end

close the window, open a new one

MERGING BRANCHES

git diff master...seo_title
to see the changes between branches

git checkout master
move to the branch that will receive the merge

git merge seo_title
merge seo_title into master branche

now we can completely delete the merge branch

you'll want to run merge on a clean working directory so you can potentially sort out problems with merges

fast foward merge: means that it inserts the commit of the alternative branchf forward ahead of the last commit in the master branch. It's like moving the commits from one branch to another.

git merge –-no-ff branch
dont do a fast forward

git merge -ff--only branch
dont do a merge unless you can do a fast forward merge

git merge branch_name
merge made by recursive strategy
creates new commit in master branch
recognizes changes in any given file and applies changes from both branches

GIT CONFLICTS

when there are two changes in two different commits in the same place
when we go to merge, git doesn't know which one you prefers
git will mark the conflict and hope you can tell it how to solve the problems

for example if you change html tags to a same line in two different branches in two commits one for each branch, and then you want to merge one branch with the other

git status
“both modified”
shows you there is a conflict to be resolved
you need to edit the document (git marks the merge conflicts
<<<<<<
=====
>>>>>>

)

RESOLVING CONFLICTS

a) abort the merge

git merge --abort
aborts the merge

b) resolve the conflicts manually

(open filewithconflict.html)
make the changes you want to make
removing from the document the repeated lines as well as git markings
(save it, close it)
git add filewithconflict.html
git commit
(if you are in the middle of a merge you don't need to provide a message to the commit)

c) use a merge tool

git mergetool
shows you tool candidates

git mergetool = tool

TIPS TO REDUCE CONFLICT PROBLEMS

keep lines short in your documents – easier to see the conflicts
keep commits small and focused
beware stray edits to whitespace (spaces, tabs, line returns)
merge often to your master branch – makes merge conflicts less and less likely

STASHING CHANGES (an area separate from working directory, staging area and repo)

when you need to switch branches but you haven't commited changes:

git stash save “changed mission page title”
puts things in the stash

LOOKING AT THINGS IN THE STASH

git stash list
shows a list of the things that are in the stash

stash@{0} is the way in which we refer to the item in the stash

git stash show stash@{0}
shows as stat about what changed in this file

git stash show -p stash@{0}
shows us the typical diff

PULL ITEMS OUT OF THE STASH

git stash pop stash@{0}
git stash apply stash@{0}
will bring the stash into whatever working directory we are on

git stash pop also removes it from the stash
git stash apply leaves a copy in the stash

if we dont specify what element we want to pop or apply it will bring the first one on the list

DELETE ITEMS THAT ARE IN THE STASH

git stash drop stash@{0}
deletes the stash@{0} item from the stash

git stash clear
deletes everything on the stash

REMOTES

git remote
gives us a list of the remote git knows about

git remote <alias> <url>
for, example give it the name of origin (this is a default convention)

git remote add origin https://github.com/Iriarte81/explore-california.git

git remote -v
shows us the urls that git will use for fetching, and for pushing

git remote rm origin
will remove the origin remote we added to our computer

PUTTING COMMITS IN REMOTE REPOSITORIES

what we are pushing are branches

git push -u origin master
where origin is the alias for the remote repository and master is the branch we have in our computer
it will ask for username and password for github.com

ls -la .git/refs/remotes
shows you your remote

git branch -r
shows you remote branches

gir branch -a
shows you local and remote branches

CLONING A REMOTE REPOSITORY

how to get a local copy of a remote repository if we don't have a local copy of it

git clone https://github.com/Iriarte81/explore-california.git name_of_new_repo

TRACKING BRANCHES

git push -u
the -u option asks git to make note of the push, to track it
this means the -u option allows us to communicate with remote repositories

git branch –-set-upstream non_tracking origin/non_tracking
to make a branch be able to be tracked.

git push
if we are on a tracking branch, git knows where to push it

we can go to a different project and fetch the changes made to another project
fetch synchronizes origin master with what's in the remote repository in github.

git fetch origin
will fetch from the repository in github.com
now origin/master is in synch with the remote

with git branch -r
we can see the new non tracking branch we downloaded

it didn't bring it into our master branch, it brought it to origin/master, which is the synch of the remote repository.

a fetch is not harmful at all

always fetch before you start working to get all the work from your collaborators

fetch before you push

MERGING FETCHED RESPOS

fetching brings branches from master in remote into origin/master in our local repo, but not into our master branch in our local repo. To do this we must merge

git merge origin/master

merges into origin/master the branch we are in (the master branch)

GIT PULL

git pull = git fetch + git merge

git pull is very convenient, but it obscures that we are doing a two step process

CHECKINGOUT REMOTE BRANCHES

remote branches can't be checkedout but we can create a branch from it

git branch non_tracking origin/nontracking
creates a non tracking branching tracking origin/nontracking

git checkout -b non_tracking origin/non_tracking
we move to a non_tracking branch and once we make changes that will go to the remote repository

PUSH TO AN UPDATED REMOTE SERVER

if there are more new commits done to the remote server and we want to push a branch to the remote server, we must first do a fetch, then we merge our changes
with origin/master, and then we can push to the remote server

DELETE A REMOTE BRANCH
tell github to delete a branch

first method:
git push origin :non_tracking
delete nontracking to the server
this doesn't affect the local branch
how to read the command. “push to origin nothing up to the branch non_tracking”

second method:
git push origin –-delete non_tracking

COLLABORATING IN GITHUB

we have to add it from our github page, you gotta know the persons username and addthem to the project. Github will provide them the url for the project to read and write for the project.

On an open source project: everyone has read access, but to make changes you need to make a fork. Forking will make your own version of the project in your own repository. Once you got it all done you go to the original project and you do a pull request explaining what changes you made. If they like your changes they'll accept your changes and merge them with the original projects

ADVANCED TECHNIQUES

CREATING ALIASES TO SPEED WORKFLOW

git config –-global alias.st “status”

git st
will work just like git status

some common aliases (the double quotes are necessary when there are spaces in the original command)

git config –-global alias.co checkout
git config –-global alias.ci commit
git config –-global alias.br branch
git config –-global alias.df diff
git config –-global alias.dfs “diff --staged”
git config –-global alias.logg “log –-graph –-decorate –-oneline –-abbrev-commit -–all”

PASSWORD CACHING

might want to search in github how to use a keychain so you don't have to type username and password everytime you want to access the remote from the terminal

otherwise we can use a SSH key, github has a help article on this alternative technique