%%%
Contents
%%%

%%%
Part #1 - Take Your First Steps With Git <- ABOUT GIT
%%%

Part #1 - Take Your First Steps With Git <- ABOUT GIT
1. Get the Most Out of This Course
2. Discover the Magic of Version Control
3. Understand the Use of Remote Repositories in GitHub
4. Start Your Project With GitHub
5. Install Git on Your Computer
 Quiz: Take Your First Steps With Git

1. Get the Most Out of This Course
- -> file history <- tracking changes
- course outline
  - -> installing / configuring git
  - -> git commands
  - -> correcting mistakes with git

2. Discover the Magic of Version Control
- -> managing development projects
- <u>github vs git</u>
  - -> github <- stores code
  - -> git <- does version control / stores different changes
  - -> <u>git hub is a hub of code and git is for branches etc</u>
  - -> <u>git is the version of the code on the person's computer, and github is the remote version</u>

    - -> github is the shared one and git is the local one
- why version control
    - -> to restore previous versions of the code
        - in case something goes wrong when clients want changes made
    - -> if there are multiple people working on the project
    - -> to record the different changes (about them / who did them etc)
    - -> to revert to original versions
- as a version control system
    - -> versioning is managing the different versions of files, logging the changes to them etc
    - in teams of developers
        - -> you have the same starting code, everyone makes their own changes to it
        - -> then they push their versions to the original document
        - -> the other people on the team approve those changes
- git is decentralised
    - -> new repos are created every time changes are made

3. Understand the Use of Remote Repositories in GitHub

- what repos are
    - -> repos are file archives
    - -> so you can compare changes made or revert to previous versions
    - -> e.g when the computer stops working -> you can back the code up on the cloud / host the project on git
    - -> on git, repos can be public or private <- some developers allow other people to change their code
    - -> git is the organisation system for version control
- local repos
    - -> the files on your computer
    - -> the entire idea that github is remote and git is local
    - -> versions are stored in the local repo
- remote repos
    - -> read only / read / write permissions e.g
    - -> you can build up a version history
    - -> private repos are only yours
    - -> local repos <- making changes to the code
    - -> there are open source projects <- anyone in the public can change it
- different platforms
    - -> github <- to make it easier to interact with other people
        - people use this for their portfolios
        - free and subscription versions
    - -> gitlab
        - -> anti-microsoft
        - -> git was overtaken by microsoft
    - -> bitbucket
        - -> Atlassian's version control system
        - -> Atlassian is for managing projects

4. Start Your Project With GitHub

- outline
    - -> creating an account and repo on git
    - -> how to use github
    - -> creating an open source project <- which anyone can access / change the code
- create a git account

    - -> there are different subscription options
    - -> the username is public on git
- the git UI
    - -> pull requests
    - -> repos
    - -> team pages
    - -> activity or organisations / repos you follow
    - -> start a project
    - -> you can edit your profile and see contributions
    - -> pull requests <- you can tell the team the changes you want to make
    - -> explore <- this is to find open source projects
    - -> you can create your own repos
        - public and private
        - copyrights
        - readme.md
    - -> gitignore <- the files you want to be ignored in the project

5. Install Git on Your Computer

- introduction / about
    - -> you can work from a local repo and then push it to git
    - -> alt., you can only push the changes you make to git
    - -> git it a distributed system, made from many computers
        - -> using git on your computer is similar to working with github
        - -> IDEs offer git integration
- install git
    - -> it's installing git on the computer
    - -> you install git on the computer <- this is different to the github webpage, it's using git on a local repo
    - -> you have to select git bash within the installation
    - -> you need the version of it for your operating system
    - -> for mac you can install it in the terminal
        - literally paste in 'brew install git'
        - then wait
        - then paste in 'sudo port install git'
- initialise git
    - -> he's in the terminal
    - -> git is for file management, so it makes sense that for the local repos its interacted with in linux
    - -> git back is for running git from the command line, in the local repo
    - -> cd <- changing directory

TO USE GIT IN THE TERMINAL WITH THE PROJECT

1. **open the terminal**
2. **type 'cd ', then drag in the folder with the project in it**
3. **hit enter in the terminal**
4. **then type 'git init' and hit enter**
5. **then you have it open in git in the terminal**

```
Open Classrooms — firstProject $ git config --global user.name "John Doe"
Open Classrooms — firstProject $ git config --global user.email johndoe@example.com
Open Classrooms — firstProject $ git config --global color.diff auto
Open Classrooms — firstProject $ git config --global color.status auto
Open Classrooms — firstProject $ git config --global color.branch auto
Open Classrooms — firstProject $ git config --global core.editor vim
Open Classrooms — firstProject $ git config --global merge.tool vimdiff
Open Classrooms — firstProject $ git init
Initialized empty Git repository in /Users/lurnid/Documents/firstProject/.git/
Open Classrooms — firstProject $
```

- <- these are the lines which he uses to configure git (the local version in the terminal when working with the file)

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

```
$ git config --global color.diff auto
$ git config --global color.status auto
$ git config --global color.branch auto
```

```
$ git config --global core.editor notepad++
$ git config --global merge.tool vimdiff
```

    - make the local repo (just mkdir)
    - then cd into it into the terminal
    - then enter all of those config lines
    - then git init -> that's it

- create your local repo
    - -> two methods
    - -> you can make a repo, or clone one from a remote repo
    - -> he's not using the file explorer at all, he's just doing it in linux
    - to make a repo
        - -> make a folder
        - -> ls into the file in the terminal
        - -> then in gitbash

```
johndoe ~ $ cd Documents/FirstProject
johndoe ~/Documents/FirstProject $ git init
Initialized empty Git repository in c:/users/JohnDoe/Documents/FirstProject/
```

        - -> so you are cd'ing into a folder in the terminal, and then git init'ing it
            - -> which creates a hidden git file
                - git init initialises the file as a repo
            - -> this is for local version control of that file
            - -> it's like time machine on mac, but for one file
            - -> in finder on mac, you can open the hidden git file in the folder (which is now a repo on the machine) by cmd + shift + [.]
            - -> there are other ways to do this in windows
            - -> this hidden file contains information about the file -> the difference between a folder and a git repo is that the repo has had 'git init' entered in the terminal when that folder has been cd'd into
            - -> this contains the config, logs, branches et al
- recap
    - -> you need to download git on the machine
    - -> for a git repo, either
        - make a folder, cd into it on the terminal and then git init <- which creates a hidden file, turning the folder from a directory into a git repo (a local one)
        - or clone one from git

Quiz: Take Your First Steps With Git

- version control systems
    - -> for keeping a history
    - -> backtracking is a problem arises
    - -> working in a team
    - -> keeping track of changes
    - -> everything except putting a website online <- no, you can use git when putting websites

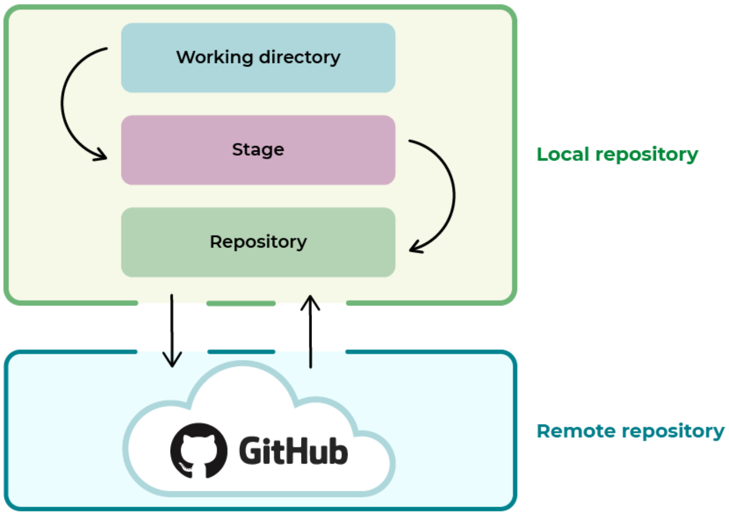
online - e.g when deploying from netlify - sites you use to deploy websites can integrate with git

- git vs github
  - -> github isn't installed on the computer
  - -> git is
  - -> would say right answer is "Git is a version control system, and GitHub is an online service that hosts Git repositories."
- is git a decentralised version control system
  - -> yes because it operates locally
- a local repo
  - -> The place on your computer where a copy of a project is stored
- git remote repos allow you to
  - -> work in a team
  - -> contribute to open-source projects
- to launch a github project, you need to
  - -> create a github account and repo
  - -> you can't install github on your computer, you can install git on it
- to install git, you need to
  - -> download and configure git
  - -> you don't actually need a github account for it
- <u>in gitbash, aka in the terminal <- you use git init to start the project</u>

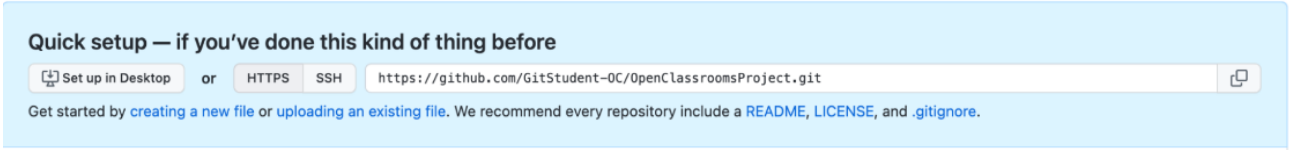
%%
Part #2 - Use Basic Git Commands <- USING GIT FOR BRANCHES
%%

1. Work in Your Local Git Repository

- working on a repo using git commands
- -> initialising repos (in the working directory)
- so
  - -> say you're editing the file in the working directory, you've made changes
  - -> then you git add -> which is sending them to the stage area
    - the changes are approved
  - -> then that's committed to the main repo (on the local branch)
  - -> then when someone on the local branch has a version, they push it to github - to the version on the cloud, which other people can see

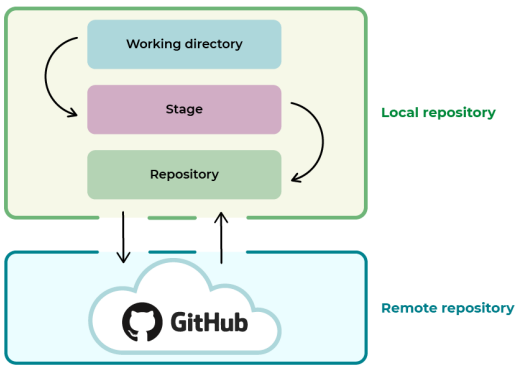


- with versioning on the local repo
  - the working directory vs the stage vs the repository
    - the working directory is like the pwd on the local version
    - the stage is like the code in the IDE
    - and then when you save it, it's in the local repo
  - so for example
    - -> you have a project
    - -> one of the files in the project changes
    - -> you send the changed file to the stage
    - -> and then a new version of the project is created
- task
  - -> make a repo
  - -> pwd in gitbash <- the current working directory
  - -> then initialise a repo <- git init (when in a directory)
  - -> main is the name of the main branch
  - -> this was changed from master after 2020 to make git more inclusive
- solution
  - -> he's in git bash
  - -> in other words, he's cd'd into the local repo on his machine
    - (the repo which was initialised)
  - -> then in the terminal
    - touch index.html <- this is how you create files in linux
    - ls <- this shows it to you
    - git status <- this shows you the status of the file in git
    - git add index.html <- for example this on when you run git status again will be green on the screen
    - git commit <- this moves the files in the stage to the repo
    - git commit -m <- m is the message with the description of the change which is being made to the repo
- the process which he goes through in the text below the video
  - -> created an index.html file and styles.css file in the local repo
  - -> so he's changed a repo in the terminal by adding files to it
    - you make the changes
  - -> then $ git add index.html styles.css <- to add the changes to the stage
    - you submit the changes for approval
  - -> then to create a new version of the project with those changes - you commit them
    - you add the changes to a new version of github
    - this is to create a new version oft he repo -> with the new changes on them - git commit
    - git commit -m “Addition of basic html and css files” <- m is the message you use to log which change was made
      - -> m is an argument
      - -> without an m, it will make you type text with the message for it
      - -> commit them or they won't be saved
      - -> these help you keep track of which changes were made when you are reverting the project back to a previous version
        - this is as opposed to naming the versions with numbers - they are named with descriptions, listed according to what the exact change was
  - -> then you push the commit to the remote repo on github
    - -> you need to link the repo on git to the local one which you've just made changes to - and are going to push the changes to it
      - i.e the local terminal and the one on the cloud those need linking
      - you go to the git repo and get the ssh or https link for it



- then you paste it into the computer terminal -> in git bash
  - git remote add origin https://github.com/GitStudent-OC/Openclassroomsproject.git
  - git branch -M main
- -> that links the git repo to the local version
- -> then to push the changes on the local repo to the one on github
  - git push -u origin main

- -> for instance, to then create changes to the index.html file on the git repo by doing it from the local one
  - -> git add index.html <- this stages the modified html file
  - -> git commit -m “H1 title change” <- creates a new version
  - -> git push origin main <- sends the version to the remote repo
  - -> so it's
    - git add
      - to add files to the stage
    - git commit
      - to create a new version from the added git files
      - creating messages to explain these changes
    - git push
      - sending those changes to github



2. Understand the Branch System

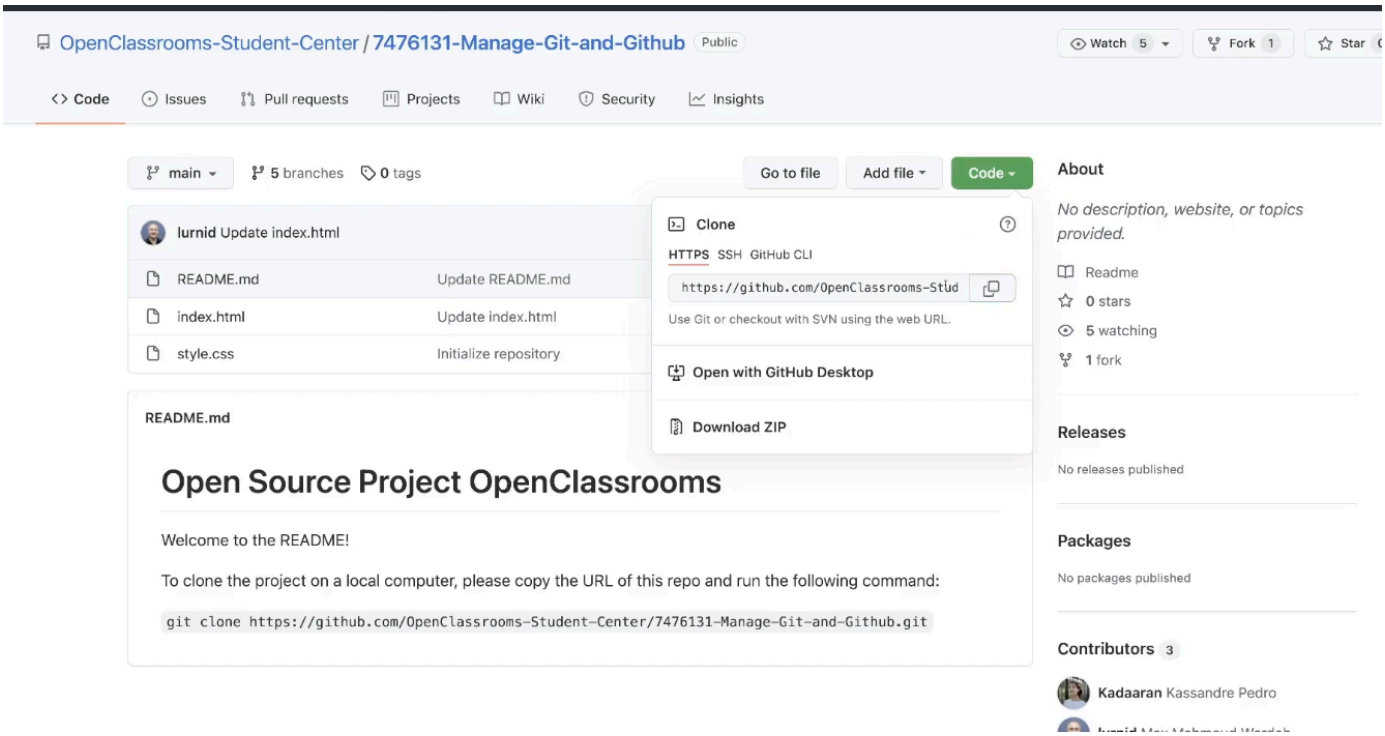
- branches
  - -> once you have a local repo <- you have the main branch -> and all of the changes branch off of the main one
  - -> new versions branching off of the main branch
  - -> you can create new files in branches
  - -> a branch is a version of the code you can play around with
    - -> you make changes on branches, which you test - before committing them to the main one
    - -> you can make git branches without affecting the main code
  - -> you can create virtual branches
- -> version control phases
  - untracked
    - -> the local repo isn't being updated with the changes made to the file
    - -> you can make changes / delete parts of the code
  - staging
    - -> getting ready to share it
    - -> you can choose what to get ready, before moving it in
  - committing
    - -> think carefully before committing a change
    - -> it can check that there are no conflicts before adding the changes into the main repo
- -> to run git branches in the terminal

    - git branch <- this lists the name of the current branch
        - -> make one branch per feature you want to add to the code
    - git branch name-of-branch <- this creates another branch for a new feature
    - -> then git branch shows you all of the branches and which you are on
    - -> then git checkout name-of-branch <- moves you back to the other branch
    - he's making changes to the branch
        - -> touch name-of-file <- he's made a new file
        - -> then git add name-of-file <- moved it to the staging area (the change has been made ot the file)
        - -> then git commit -m "message text" <- this is changing the local version of the file on the branch
        - -> then git push -u origin name-of-change <- this pushes the change ot the git repo
        - -> then he's merging the change with the main branch
- -> then he merges the changed branch with the main branch
    - git checkout main <- this switches back to the main branch
    - git branch <- to check he's on the main branch
    - then to merge the two
        - git merge name-of-branch <- he moved to the main branch in order to merge the new one with it -> so it moved back to the main one after the new one whose changes he wanted to merge was complete
- notes on the text below the video
    - -> git branch <- to see the branches which we have
        - if there are no others, then you will have 'main' (this is the main branch)
        - the branch you are currently on has a *
        - git branch, in other words - show me the branch I'm on, * this is it
        - -> make a branch to the project if you are creating new changes which could affect the main one
        - **-> when you're making edits to the project, don't make those edits to the main branch - make a new branch for those changes first**
    - -> git branch name-of-branch <- this creates a branch
        - -> locally, it creates it locally
        - -> not on github
    - -> then git branch <- this shows the branch which you've now created
        - -> for the changes
        - -> you want to move onto it -> the current branch we're on it shown with a *
        - -> in this case it's the main one
    - -> git checkout name-of-branch <- to move to the new branch
    - -> then you can check you're on the new branch if you git branch again <- and this now shows the main and the new branch, the * is by the new branch in the terminal
    - -> you can't just create a new branch, you have to move onto it -> git checkout
    - -> so you change that branch, and you have the main branch
    - -> you want to merge the changes with the main branch
    - -> git commit -m “Message description of the changes”
    - -> then you push the changes to the main branch
        - previous chapter
        - -> you have to be in the main branch (the branch you want the changes to manifest)
        - -> you only do this after you are happy with the changes you've done
        - -> git checkout main <- you're switching back to the mainbranch (checking out of the supermarket)
        - -> git merge name-of-change <- then this is merging the changes back into that branch
    - -> then you push the main branch back to git
- recap

- -> a branch is a copy of the project to make changes
- -> the default is the master branch
- -> you can switch between them with git checkout
- -> you can join them with git merge

3. Work With a Remote Repository

- Access a Remote Repository
  - -> cloning projects from git
  - -> he's in a project in github
  - -> you copy the link for the repo



    - -> he's done it on https
    - -> then in the terminal <- git clone link-pasted-here
  - -> he's done it while in git bash <- in other words, you need to git init etc
    - and be cd'd into the right directory in the terminal
  - -> you need to do this e.g if you are reviewing the changes that other people have made to the project
  - -> (below, optional) <- this is how you give the link / repo a short name
    - git remote add OC https://github.com/OpenClassrooms-Student-Center/7162856-G-rez-Git-and-GitHub.git
- Update Your Local Repository
  - -> git pull OC main <- if someone has pushed changes to the main repo on git, then this git pull downloads those changes from git and merges them with your local copy of the file
- Work on a Team With GitHub
  - -> github and teamwork
  - -> the simple UI of github
  - -> accessing the changes without the command line
  - -> git branch -a <- this launches all of the changes to the main branch (including from other people)
  - -> you can't merge the changes yourself, you need to do a pull request
    - other people in the team approve of the changes to the main branch of code, and vice versa
  - -> if you're working individually, then you can put push the changes / merge it to the main branch

        - but with teamwork the main branch is generally locked
- Make a Pull Request
    - -> pull requests are requests for you to change the code on the main branch, when it's owned by someone else
        - -> this is done by merging your changes with theirs
        - -> your main branch has to be merged with the changes which you've made in another branch so -
            - you're cloning the repo
            - making a new branch
            - making edits to that branch locally
            - then merging them with the main branch which you have saved locally
            - then making a request to pull the changes to the repo where the code originally came from
            - -> you can see this on git under the pull requests tab for a repo
    - -> you can add a comment to explain the reasons for the changes so the owner of the code will approve them
    - -> the changes are colour coded on git
        - -> red deleted elements of the code
        - -> blue added code
        - -> it looks a bit like the w3c html / css validators, except we're looking at the changes to the code
- Request a Code Review
    - -> this is other people reviewing your code
        - before changes are merged with the main branch
    - -> this can allow you to ask questions
    - -> there are alerts wherever changes have ben pushed to git
    - -> pull requests
    - -> there are forms where there are names and descriptions for it
    - -> essentially so whenever anyone wants to make changes to the code, they are explained and reviewed by other people - who can see that the code which is being added - what exactly has been deleted and what has been changed
    - -> the collaborators are validating the code
- Let's Recap!
    - -> you can access the url of the repo
    - -> git clone <- to make a local copy of it
    - -> git add <- to give it a shorter name
    - -> git pull <- to make a copy of the remote repo, locally

Quiz: Use Basic Git Commands

- git remote add REP https://github.com/xxxxxx/myRemoteRepository <- this is the line of code you use to give a repo a shortened name it is referred to by
- checkout is for switching branches (like being in a supermarket and checking out of the current isle, you're moving onto a new branch)
- -> add, then commit, then push the changes
- -> it's git checkout main, git merge changes
    - if you want to merge changes with the main branch, you have to be on that branch first

%%

Part #3 - Make and Fix Common Mistakes <- USING GIT FOR TROUBLESHOOTING

%%

1. Make and Fix Mistakes in a Local Repository

- You've Accidentally Created a Branch You Didn't Want

- -> if you make commits on the wrong branches etc, or want to change the messages which describe them
- -> to revert the code to an old version / branch
- -> git has commands which can fix common mistakes
- -> other examples include - forgetting files in commits, creating branches you don't want, changing the wrong branch
- -> he's created a repo sandbox
  - -> in the terminal, he's just created a folder and cd'd into it
  - -> then git init <- so now it's a local repo
  - -> sandbox <- an area of code for messing around in, like a codepen not quite like a vm
  - -la <- to show hidden folders
  - command shift + <- in the finder on mac to see the hidden files, the .git file in a directory which makes something a local git repo
- -> he's made a file in the repo
  - this is done in the terminal using the touch file-name <- this creates the file
  - -> git add FirstFile.txt git commit
  - -> then enter and validate the commit message
    - in other words, he's created a file in the terminal, then edited it using nano in the terminal (or vim)
    - -> so he's just made changes in the local git repo
- -> then he's created another branch
  - git branch name-of-branch
  - git branch <- this shows the names of the branches
- -> git branch -d name-of-branch <- this is to delete the branch called name-of-branch
- so it's
  - -> git branch name-of-branch <- when it comes to creating a new branch
  - -> then git branch <- to see the different branches
  - -> then to delete the branch it's the line underlined above
  - -> then git branch to chek it's no longer there
  - -> git branch -D name-of-branch <- to delete the changes which were made to the branch

- You've Changed the Main Branch
  - -> if you change the main branch by accident <- rather than putting the changes into a new branch
  - -> use the stash commands
  - -> the changes are put to the side until a new branch is made and then the changes can be added ('stashed') there
  - -> as long as the commit hasn't been made
  - -> in the terminal
    - -> he's cd'd into a local repo
    - -> then he's made a change to the main branch -> touch file-name.txt
      - this has made a file
    - -> then git add file-name.txt
      - this adds the file to the branch
    - -> then git status
      - this shows you the state of the files (they've been changed but not committed)
      - if the changes have been made or not
      - -> the stash only works if the changes haven't been committed to the main branch
      - -> he uses this constantly to see if the changes have been made or not
    - -> then he's doing a stash
      - git stash <- this cleans the main branch and puts the changes which were made to it aside - when the changes were made to the main branch

        - git status
            - -> then your changes have been saved
    - -> git new-branch
        - -> git checkout new-branch <- he's made a new branch and moved to it, this is the branch which we want to commit the stash to (checking out, like in a supermarket, checkout of the main branch and move to it)
        - -> git stash list <- then he's added the changes which were put aside from the main branch - he's added those changes to this branch
            - this line shows the changes which we're stashing
        - -> git stash apply <- this is to apply the stash
            - this change applies then
            - if you have multiple stashes, then it's git stash list <- to see the full list of them
            - -> every time you change the main branch and then make a stash, you are separating the branch before the changes which were made from the changes which were made
                - each of the stages has a different id
                - -> then to apply the changes from one of the stashes, it's git stash apply stash@{0}, e.g
- You've Changed the Branch and Made a Commit
    - context
        - -> if you have made changes to the main branch instead of another branch - and then you've committed them to the main branch
        - -> stashing (above) - this was for making changes to the main branch which you shouldn't have, but not having had committed those changes
    - find the id for the commit
        - -> the git log analyses the most recent commits
        - -> the 'hash' is the ID of the commit you want to get rid of
        - -> git log <- in the terminal
            - this lists the most recent commits first
            - -> this returns the commit id of the most recent one (the one we want to get rid of)
    - then you delete the last commit
        - -> git reset --hard HEAD^
            - head - the last commit is the one you want to delete
        - -> while you are on the main branch, enter the above
        - -> but record the commit id first
        - -> this deletes the last commit from the branch
    - then we create a new branch and move the commit onto it
        - -> so we are deleting the commit from the main branch, then creating a new branch and moving the commit onto it
        - -> we've deleted the commit from the main branch
        - -> so we first create a new branch -> git branch name-of-branch
        - -> then we checkout of the main branch and move to the new branch
        - -> this is done by git checkout name-of-new-branch
        - -> then it's git reset to apply the change to the new branch <- git reset --hard ca83a6df
            - and that's the id of the commit -> which you're moving onto the new branch, after having deleted it from the main branch
- You've Made a Mistake in Your Commit Message
    - -> when you make changes to code and commit them, you write a message, aka a description of the change
    - -> git commit -m "changes made"
        - so he's made a change and committed it
    - -> before the change is pushed - you can't change it after it's been pushed

    - -> git commit --amend -m "new message" <- this is to change the message which is on the last commit made - before it's been pushed, and it has to be the last one made
    - -> basically when you are making changes, make sure you spell the message right before submitting it
- You Forgot a File in Your Last Commit
    - -> if you commit the changes but didn't add a file which you should have
    - -> the git amend command
        - git amend is for modifying the last commit
    - -> git add file-name.txt
        - this adds the file
    - -> git commit --amend --no-edit
        - this adds the file to the commit
        - this allows you to amend your most recent commit
    - -> after you committed the changes
    - -> what you do is, in the terminal, cd'd into the local repo you want
        - then -> git --amend
            - this is to amend the repo, to make changes to the commit which you've just created
            - i.e, we're going to add a file to it -> which is done via git add name-of-file.txt
        - after you've made a commit and left out files you wanted, then
            - -> git add file-name.txt
            - -> then it's git commit --amend --no-edit
            - -> the no edit <- add the file without changing the message which is attached to the commit
- Let's Recap!
    - -> commit --amend <- this is to add changes to the last commit, in case you left something out or want to delete / change something
        - -> -m <- you can use this option to change the log message
    - -> git branch -d <- this deletes a branch
    - -> git status <- to check the status of files
    - -> git stash <- if you make changes to the main branch and then want to separate out those changes from the main branch you initially started with, you can then move those stashed changes onto a new branch and be left with the main one in tact
        - this is for unstaged changes
    - -> git log <- to show the commit history on the current branch
    - -> git reset with --hard HEAD^ <- to reset the staging area and working directory to how they were at the last commit
    - -> commit --amend <- to select the last commit and make changes to it
        - -> amend is all for amending the last commit (e.g adding a file you missed or changing the message which describes the changes which were made)

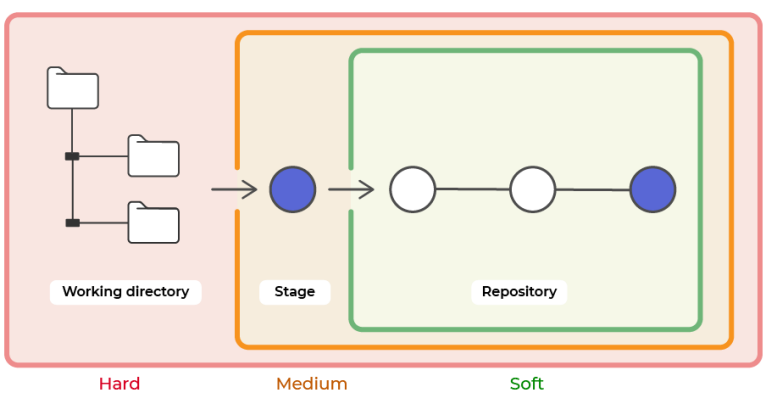
2. Correct Your Mistakes in Your Remote Repository

- Correct Your Mistakes Locally and Remotely
    - -> if you push code which has mistakes in it
    - -> you're changing a public project -> other people also use that code
        - you need to tell other people in case they are using it
    - -> then git revert <- this is how you revert the changes
        - in other words how to undo a comit -> using another commit
        - -> it just commits the previous version of the code, rather than the ones you created
        - -> git reset is for local directories and git revert isn't
    - -> in case you've pushed the wrong files
- Remote Access Isn't Working
    - -> authentication problems with the network <- in case the terminal isn't connecting to the

repo
        - the connection between your computer and a remote repo
    - -> so you don't have to re-sign into git
        - you can use these for accessing the repo from other machines
    - -> they suggest an ssh key
    - he generates an ssh key for the repo in the terminal
        - -> he's in the terminal <- ssh-keygen -t rsa -b 4096 -C "email@gmail.com"
            - $ ssh-keygen -t rsa -b 4096 -C "johndoe@example.com"
            - then it's generating a key to the repo
            - -> this generates ssh keys for that computer
            - -> then he goes to the directory in the terminal for the keys
            - -> one of them has a private key, and the other has a public key
            - -> one of the elements has a cat in it, and the other doesn't
            - -> he copies the key, in Windows it's stored at C:\Users\YourUserName\
                - -> in mac it's a hidden file
                - -> command, shift + to show the hidden files
                - -> there is a file for the hidden and another file for the public keys
                    - id_rsa.txt <- this is the private key
    - then he links that ssh key to git
        - -> in the webpage for git
        - -> he pastes the key
        - -> settings > ssh and gpg keys > new ssh key > name it and paste it in
        - -> he's only using the public key for git
- Change Your Login Information and Delete the Key
    - -> settings in the git account <- it's the same stages above
    - -> he's named the ssh key
- Let's Recap!
    - -> git revert HEAD^ <- to undo a commit by creating a new one
    - -> ssh-keygen <- to generate a new ssh key pair, one of which can be entered into git
    - -> the next chapter is git reset

3. Use Git Reset

- -> when the client changes what they want on the webpage
- -> they want to keep the original version of the webpage
- -> you have to git reset <- soft /mixed / hard, to undo the changes
    - soft
        - -> --soft <- in the terminal
        - -> doesn't delete files
        - -> back to the stage files before the commit
        - -> this is telling it to go back to an old commit, which can be used to make branches
        - -> you go back to an old commit, and then can make branches off of it

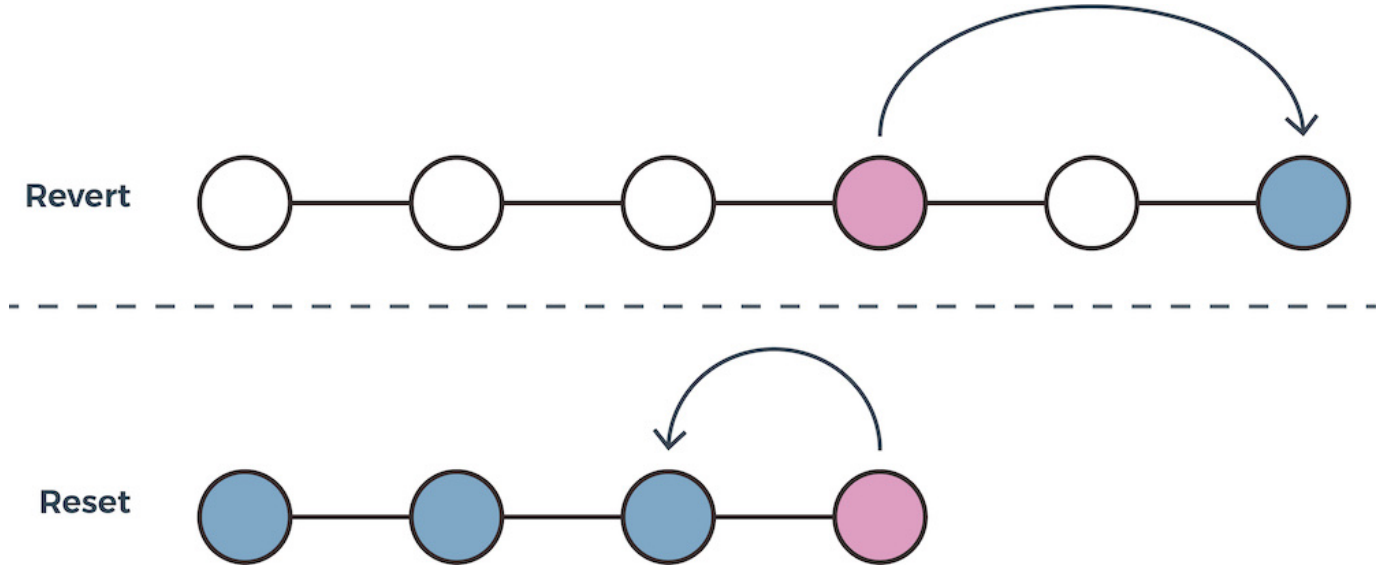


    - mixed reset <- in the terminal
        - -> --mixed
        - -> back to the last commit, so nothing is deleted
        - -> git reset --mixed <- to go back to where you were before the last commit
        - without deleting the changes you're working on now

        - you can use this to upstage files which haven't been committed
        - -> so, you're working on files and want to go back to the old version without deleting the changes you're currently working on - this is where you use a mixed git reset
    - git hard reset <- in the terminal
        - -> --hard
        - -> these changes can't be undone
        - -> permanently deletes the commit - use wisely (it's like a sudo)
            - wisely because if you use it - then it deletes all of the commits which were made after it
        - -> git reset OurTargetCommit --hard
    - default
        - -> if you just type git reset into the terminal, the default one it uses is the HEAD^
        - -> this refers to the last change / commit which was made
        - -> HEAD refers to the current repo, it's a pointer and often used as default when no options are presented to the commands in the terminal
    - -> the changes can't be shared yet
- conflicts when merging branches
    - -> when you are merging two branches and conflicts arise
    - -> if one element has been changed on both branches, and then you're trying to merge those two branches together
        - we end up with a conflict when there were two changes to the same element
    - -> so you're checking out of the branch which you're currently on, and back to the main branch
    - -> you try and merge the two on the terminal and then you get a merge conflict

```
git checkout main
git merge fundingPotImprovement
Auto-merging FundingPot.php CONFLICT (content): merge conflict in FundingPot.php
Automatic merge failed; fix conflicts and then commit the result
```

    - -> to resolve the conflict, you go into the place where it is emerging in a file editor in the terminal
    - -> then you add the merge by using git add name-of-change.txt git commit in this example
- adding the wrong file to a commit
    - -> git has a time travel feature
    - -> it's like time machine on the mac -> the older versions of the file - this is a version control system for the file management
    - -> git reset -> this one goes back to the old state without saving the current one
    - -> git revert <- this one goes back to the old state, but it saves the current one
        - so it saves the current one, then makes a new one which is the same as the old one
        - -> it's a reset but - the current state is saved
        - -> a new state is created which is the same as the old one
        - -> a revert is a reset which saves the current version which you don't want
        - -> but both are reverting the file back to it's initial state - just one is remembering the file you made but didn't want, and the other isn't - which means revert is probably better - or not, depending on the context
- an example for this is git revert HEAD



- recap
  - -> git reset
    - --soft, --mixed or --hard
    - -> a soft git reset can be used to go back to the previous state without making changes - just seeing what it was e.g
  - -> git merge
    - -> if the same file has been changed several times then we have a conflict
    - -> then you resolve the conflicts by telling it which changes you want to keep
  - -> git revert
    - -> to go back to the previous state by creating a new commit
    - -> this is in comparison to git reset, which just 'visits' the old state before the changes were made, rather than making an entirely new commit out of those old changes
    - -> thinking about moving between the different branches / commits before / after the changes were made to them

4. Correct Commit Mistakes

- correct commit mistakes
  - -> if multiple developers are making changes to the code
  - -> this is how to find where the error is, who made it and at what phase of the project it is at
  - -> the techniques are
    - git log
      - -> the tree structure of the branches <- a log is where the files are saved
        - -> you are looking at the log of the different files in the git repo (it's remote in this case, e.g it can be shared with other developers)
      - -> to see each commit and the files they contain
    - gitref log
      - -> looking for specific changes
    - git blame
      - -> this shows who made which commit and where
    - cherry pick
      - -> this is when you're looking for a specific commit
      - -> each commit has a specific ID
      - -> you use that ID to find where the change was made
  - -> if you want to go back to a previous version of the project, or know who made which commit
  - -> git has history tools which can be used for this

    - -> a history is the even of saving a file and the log is the place where it's saved
    - -> version control systems
        - to save the changes made to the code
        - -> you are using a system which tracks the changes, every time someone makes and change and then commits it to the branch it gets logged
        - -> you are looking over those logs to find where the issues are / who wrote the code in the places where we want more changes made
            - who wrote what
            - where the bugs are
            - undoing changes which cause problems
    - -> to use the log of changes to track them -> git log in the terminal
        - -> this lists commits in reverse chronological order
            - they also have IDs
            - -> SHA ID <- secure hash algorithm - this is a type of ID, a secure one
        - -> so, you make a change to the code and it gets logged
        - -> then the most recent ones are the ones which show first
- Forgotten Something? Git Reflog!
    - -> git reflog
        - -> this commits the changes to the messages / merges / resets
        - -> this also shows the IDs for the different logged actions
        - -> git checkout e789e7c <- to get back to a change
            - those eight characters
                - -> you git reflog to find the ID of the change which you want to go back to
                - -> then you copy that
                - -> it's the first eight characters of that key
                - -> then you're checking out to it -> to see the version of the file for it
    - -> to go back to a previous action / commit
    - -> it lists them from the most recent to the oldest
    - -> git reflog -> then he's moved back to when the change was made (using the example code above)
- Who’s Been Messing Around in Your Repository? Git Blame
    - -> logging
    - -> finding the author of the last commit
    - -> looking at who made the bug -> looking at who made what changes
    - -> git blame file-name.txt <- this returns a log of who made the changes, when, the line number and content
    - -> where the error int he code came from and who wrote it
        - you can find out - they are logs of when the change was made - by someone who made a mistake - it's literally logged
        - -> for each line of code changed, git blame shows
            - the ID
            - the author
            - timestamp
            - line number
            - line content
- You Want to Cherry-Pick Specific Commits
    - -> cherry picking
        - working with developers on a larger project
        - -> when you just want to merge one of the changes you've made with the main branch, rather than merge one entire branch with the main one
            - -> you're cherry picking the branch you want to add
        - -> this isn't a method they recommend they because it duplicates existing commits

            - -> they are favouring merges - because these don't duplicate existing commits
    - -> you can use git cherrypick to take a specific change e.g to send to clients
        - -> and you're picking those particular changes, not the rest of the commits
        - -> you use the ID of the commit to select it, another word for the ID of the commit is it's SHA key -> it's secure hash algorithm
        - -> git cherrypick ... . <- the ...'s are the SHAs, aka the IDs of the changes you want
            - -> this is creating duplicates
                - we are creating duplicates from the current branch
                - taking the two commits with the, or three -> taking the number of commits which are being cherrypicked, and then adding them to the main branch for cherrypicking
                    - this creates two of them (duplicates)
- Let's Recap!
    - -> git log <- to see the commit history of the current branch
    - -> this is the same as git reflog
        - except gitreflog shows the actions which were carried out
            - the local actions -> i.e the ones which happened on git on the machine, rather than on github where the shared repo for the project is
    - -> git checkout <- then you can give it the SHA-1 ID for a specific change which you want to go back to
        - each of the changes has an ID, you find this using git log -> then you can checkout to that change
    - -> git blame <- who made which changes to a file, when and which changes were made
    - -> git cherry-pick <- this can be used to commit one specific change which was made on a branch and send it back to the main one, but it duplicates it when doing this (so this isn't a preferred method among developers)
        - you do git cherry-pick and then the SHA (aka the ID of the commit which you want to send back to the main branch)
        - -> you get the SHA from git log

5. Recap What You've Learned

- -> install and configure git
- -> git commands
- -> branches and pull requests
- -> common git mistakes
- -> the next is a quiz
- -> the key is practice

Quiz: Make and Fix Common Mistakes

- there are 8 questions in the quiz
- A SHA is an ID for commits and other actions kept in Git's memory
- git blame to know who has changed a particular line in the file test.html
- when there are conflicts - e.g you are trying to merge two branches and changes have been made to the same line twice, then you need to resolve the conflict git revert creates a new commit, but git reset doesn't <- to reset the branch which you're working on to the one it was before you made commits to it
- Git reset --soft / Git reset --mixed / Git reset --hard <- the different ways that git reset can be used (a soft / mixed / hard reset)
- to change the message of the previous commit
    - -> git commit --amend -m "Test"
    - -> amend (add it onto the previous version)
    - -> -m <- you're telling it to edit the message which logs the changes

- -> and you're telling it what that new change to the message is

GitHub
GIT CHEAT SHEET

Git is the free and open source distributed version control system that's responsible for everything GitHub related that happens locally on your computer. This cheat sheet features the most important and commonly used Git commands for easy reference.

INSTALLATION & GUIS

With platform specific installers for Git, GitHub also provides the ease of staying up-to-date with the latest releases of the command line tool while providing a graphical user interface for day-to-day interaction, review, and repository synchronization.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

For Linux and Solaris platforms, the latest release is available on the official Git web site.

Git for All Platforms
http://git-scm.com

SETUP

Configuring user information used across all local repositories

`git config --global user.name "[firstname lastname]"`
set a name that is identifiable for credit when review version history
`git config --global user.email "[valid-email]"`
set an email address that will be associated with each history marker
`git config --global color.ui auto`
set automatic command line coloring for Git for easy reviewing

SETUP & INIT

Configuring user information, initializing and cloning repositories

`git init`
initialize an existing directory as a Git repository
`git clone [url]`
retrieve an entire repository from a hosted location via URL

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status`
show modified files in working directory, staged for your next commit
`git add [file]`
add a file as it looks now to your next commit (stage)
`git reset [file]`
unstage a file while retaining the changes in working directory
`git diff`
diff of what is changed but not staged
`git diff --staged`
diff of what is staged but not yet committed
`git commit -m "[descriptive message]"`
commit your staged content as a new commit snapshot

BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

`git branch`
list your branches. a * will appear next to the currently active branch
`git branch [branch-name]`
create a new branch at the current commit
`git checkout`
switch to another branch and check it out into your working directory
`git merge [branch]`
merge the specified branch's history into the current one
`git log`
show all commits in the current branch's history

INSPECT & COMPARE

Examining logs, diffs and object information

`git log`
show the commit history for the currently active branch
`git log branchB..branchA`
show the commits on branchA that are not on branchB
`git log --follow [file]`
show the commits that changed file, even across renames
`git diff branchB...branchA`
show the diff of what is in branchA that is not in branchB
`git show [SHA]`
show any object in Git in human-readable format

SHARE & UPDATE

Retrieving updates from another repository and updating local repos

`git remote add [alias] [url]`
add a git URL as an alias
`git fetch [alias]`
fetch down all the branches from that Git remote
`git merge [alias]/[branch]`
merge a remote branch into your current branch to bring it up to date
`git push [alias] [branch]`
Transmit local branch commits to the remote repository branch
`git pull`
fetch and merge any commits from the tracking remote branch

TRACKING PATH CHANGES

Versioning file removes and path changes

`git rm [file]`
delete the file from project and stage the removal for commit
`git mv [existing-path] [new-path]`
change an existing file path and stage the move
`git log --stat -M`
show all commit logs with indication of any paths that moved

REWRITE HISTORY

Rewriting branches, updating commits and clearing history

`git rebase [branch]`
apply any commits of current branch ahead of specified one
`git reset --hard [commit]`
clear staging area, rewrite working tree from specified commit

IGNORING PATTERNS

Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

`git config --global core.excludesfile [file]`
system wide ignore pattern for all local repositories

TEMPORARY COMMITS

Temporarily store modified, tracked files in order to change branches

`git stash`
Save modified and staged changes
`git stash list`
list stack-order of stashed file changes
`git stash pop`
write working from top of stash stack
`git stash drop`
discard the changes from top of stash stack