Distributed Version Control System(s): Git

Preface

- The internals of Git center around graph theory, directed acyclic graphs, hash tables, tries, blobs, trees, snapshots, checksumming via SHA-1 hashing, and garbage collection, among other elements. The ecosystem is quite vast.

- In addition, there are over 100 declarative git commands with optional arguments that you can run in a terminal that impact different stages of reachable files. Some commands are more frequently used than others and it is perfectly reasonable to not know each and every command as we can lean on the documentation if necessary.

- **Slide Deck Focus**: We will primarily concentrate on commonly used commands and scenarios to begin to develop and hone an independent, functional git workflow while highlighting and visualizing core aspects of Git infrastructure. This slide deck will **not** cover team-based git workflow, branching, and merging in-depth.

Slide Deck Outline

1) Overview
2) Setup
3) Big Picture I
4) Stash
5) Workspace
6) Index
7) Local Repository
8) Upstream Repository
9) Big Picture II
10) Core Commands and Additional Commands
    a) Git Basics, Undoing Changes, Rewriting Git History, Git Branches, Remote Repositories
11) Summary

1

Overview

A bird’s eye view of version control systems and git

Overview I

- Version control systems are **a category of software tools** that help a software team manage changes to source code over time.
- Version control software **keeps track of every modification** to the code in a special kind of database.
- If a mistake is made, developers can turn back the clock and **compare earlier versions of the code** to help fix the mistake while minimizing disruption to all team members who are working together concurrently.

Overview II

- Version control helps solve these kinds of problems and provides:
  - **A complete history of every file**, which enables you to go back to previous versions to analyze the source of bugs and fix problems in older versions.
  - **The ability to work on independent streams of changes**, which allows you to merge that work back together and verify that your changes conflict.
  - **The ability to trace each change with a message** describing the purpose and intent of the change and connect it to project management and bug tracking software.

Overview III

- Linus Torvalds, the principal developer of Linux, created the **distributed version control system** (DVCS) known as “Git” in 2005
  - Git (created in 2005 by Linus Torvalds) is **not** GitHub (created in 2008 by Chris Wanstrath, Tom Preston-Werner, and PJ Hyett)!
    - Git and Mercurial are two of the most popular DVCS’s
- Git was initially designed with some of these things in mind:
  - Speed
  - Simple design
  - Strong support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects efficiently (speed and data size)

Overview IV

- Two main types of version control systems exist: centralized and distributed (decentralized)

Overview V

- Centralized version control systems
  - In centralized version control systems like **Subversion** or **Perforce**, there is a single main copy of the project stored on a server. You commit your changes to this central copy.
  -
  - Unlike distributed systems like Git, you typically only have a partial copy of the project locally in a centralized system. You pull and update specific files as needed, but you don't have a complete local copy of the entire project.

- Distributed version control systems (DVCS)
  - In distributed version control systems like **Git** and **Mercurial**, you don't rely on a central server to store all versions of a project's files. Instead, you clone a repository locally, giving you the complete history of the project.
  - While a central repository is not required, services like **GitHub** provide a "central" place to store your code, allowing you to share and collaborate with others. By keeping a copy of your code in a GitHub repository, you can use a distributed version control system locally and push/pull code to collaborate with your team.
  -

Overview VI

- Git is used within numerous (5000+) companies such as: Netflix, Shopify, Lyft, Facebook, Apple, Amazon, Atlassian, Microsoft

Long story short...

- Git is a popular DVCS that helps streamline the non-linear, concurrent development of a project's patches, features, and versions on cross-functional teams

2

Setup

Downloading, installing, and configuring Git

Setup I

- This section of the slide deck is written with the assumption that the prework slide deck was read and completed (that slide deck can help with setup)

- If you currently do not have Git on your machine, that is okay --- here are official resources to help with that (ask for assistance, too!)
  - https://www.atlassian.com/git/tutorials/install-git
  - https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
  - https://help.github.com/en/github/using-git/getting-started-with-git-and-github

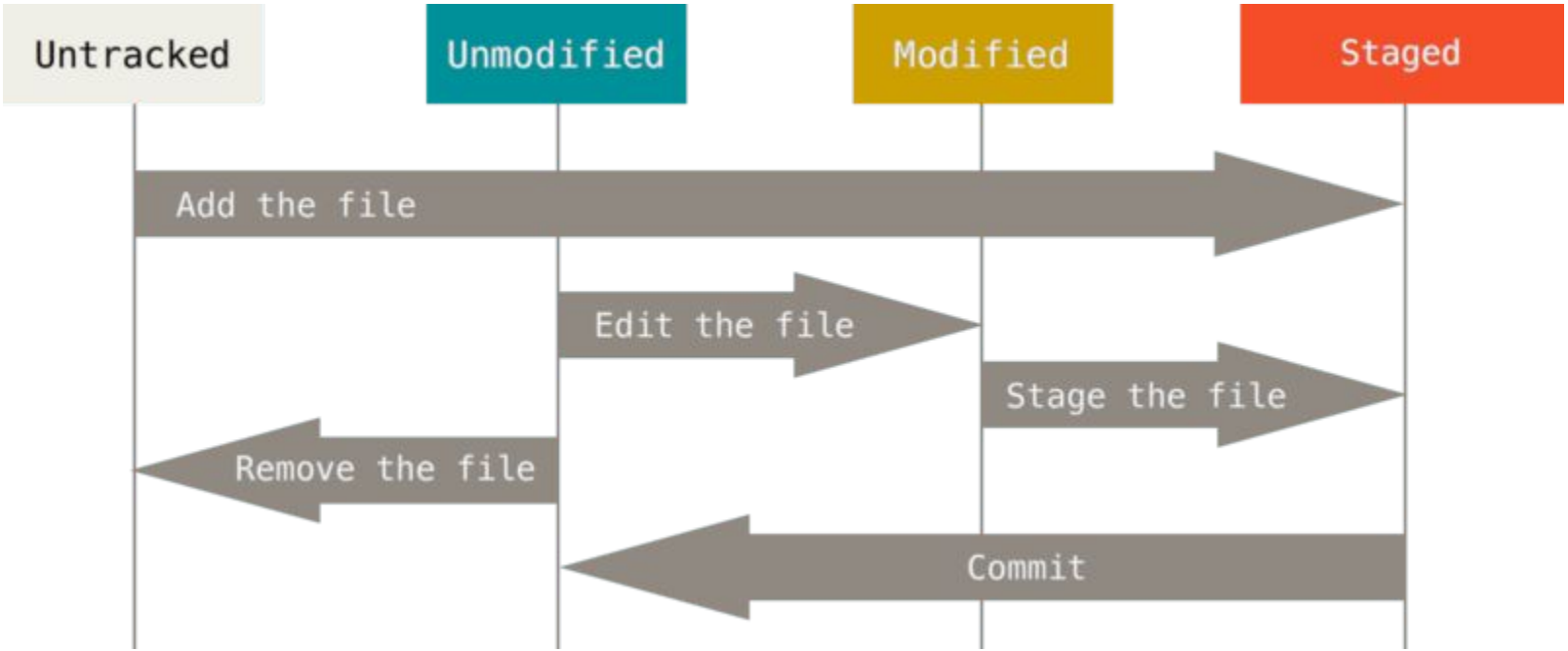
Long story short...

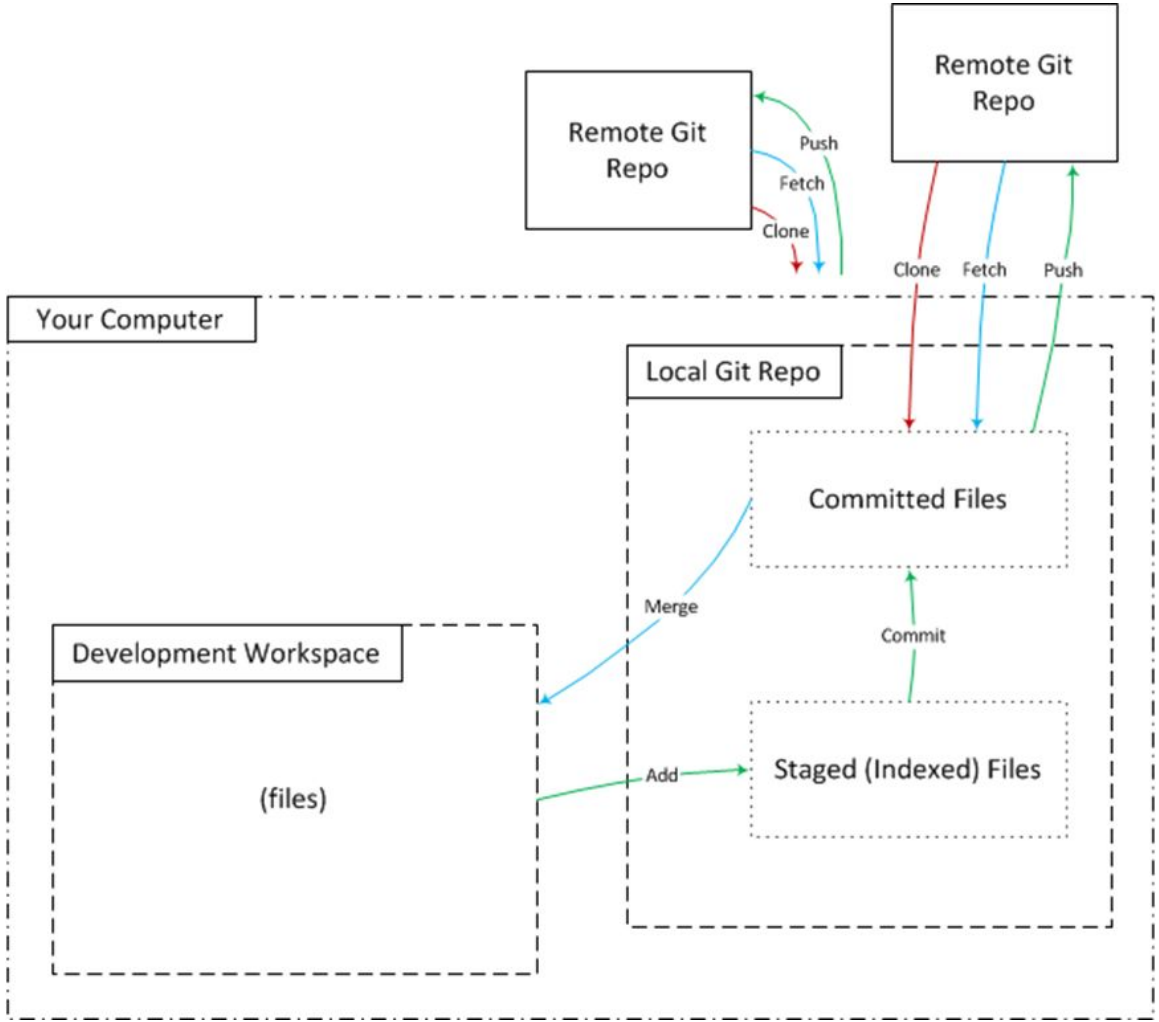
- Downloading, installing, and configuring Git is necessary for any individual and/or team to concurrently collaborate on projects with organization, efficacy, and efficiency

3

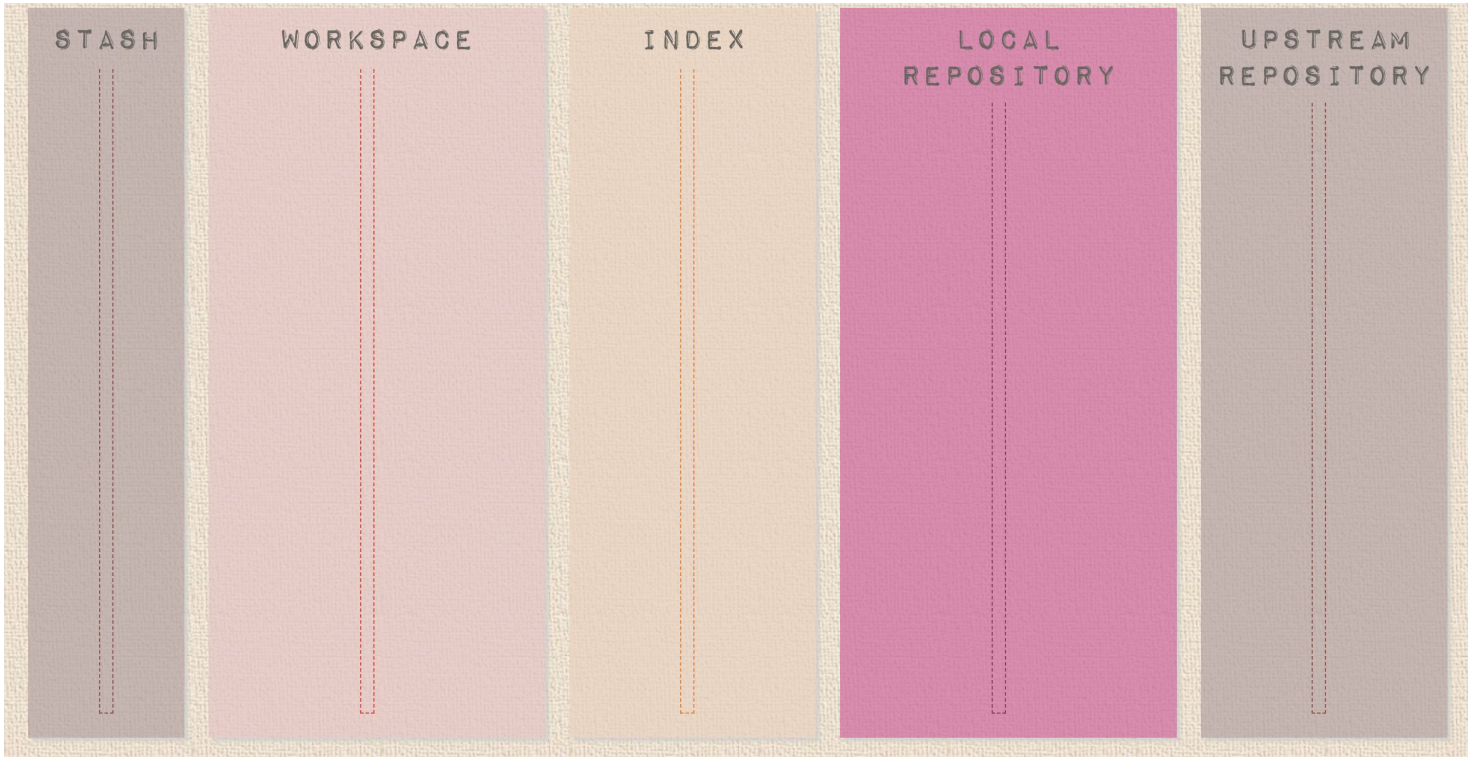
Big Picture I

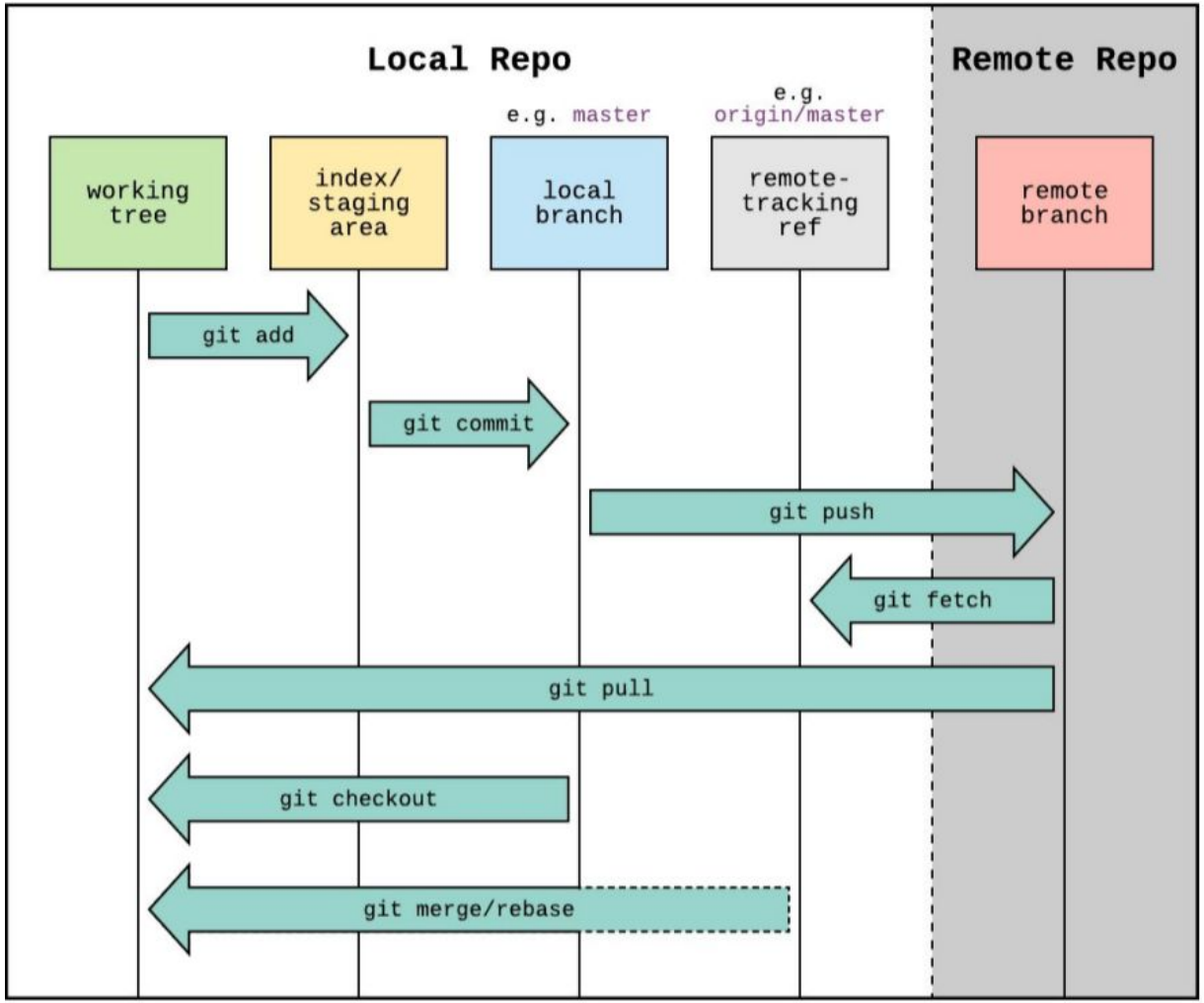
Understanding Git through diagrams and images

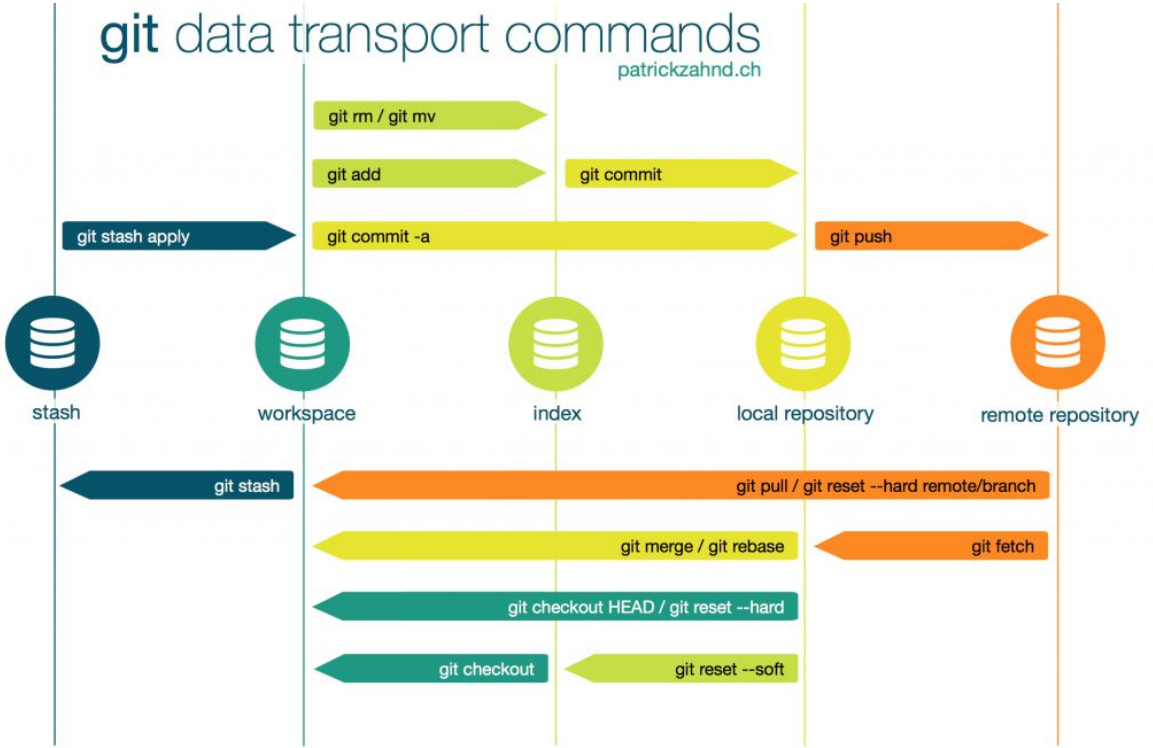

Untracked
Unmodified
Modified
Staged
Add the file
Edit the file
Stage the file
Remove the file
Commit


Remote Git Repo
Remote Git Repo
Push
Fetch
Clone
Clone
Fetch
Push
Your Computer
Local Git Repo
Committed Files
Merge
Commit
Development Workspace
(files)
Add
Staged (Indexed) Files

https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo




Local Repo
Remote Repo
e.g. master
e.g. origin/master
working tree
index/ staging area
local branch
remote-tracking ref
remote branch
git add
git commit
git push
git fetch
git pull
git checkout
git merge/rebase


git data transport commands
patrickzahnd.ch
git rm / git mv
git add
git commit
git stash apply
git commit -a
git push
stash
workspace
index
local repository
remote repository
git stash
git pull / git reset --hard remote/branch
git merge / git rebase
git fetch
git checkout HEAD / git reset --hard
git checkout
git reset --soft

Long story short...

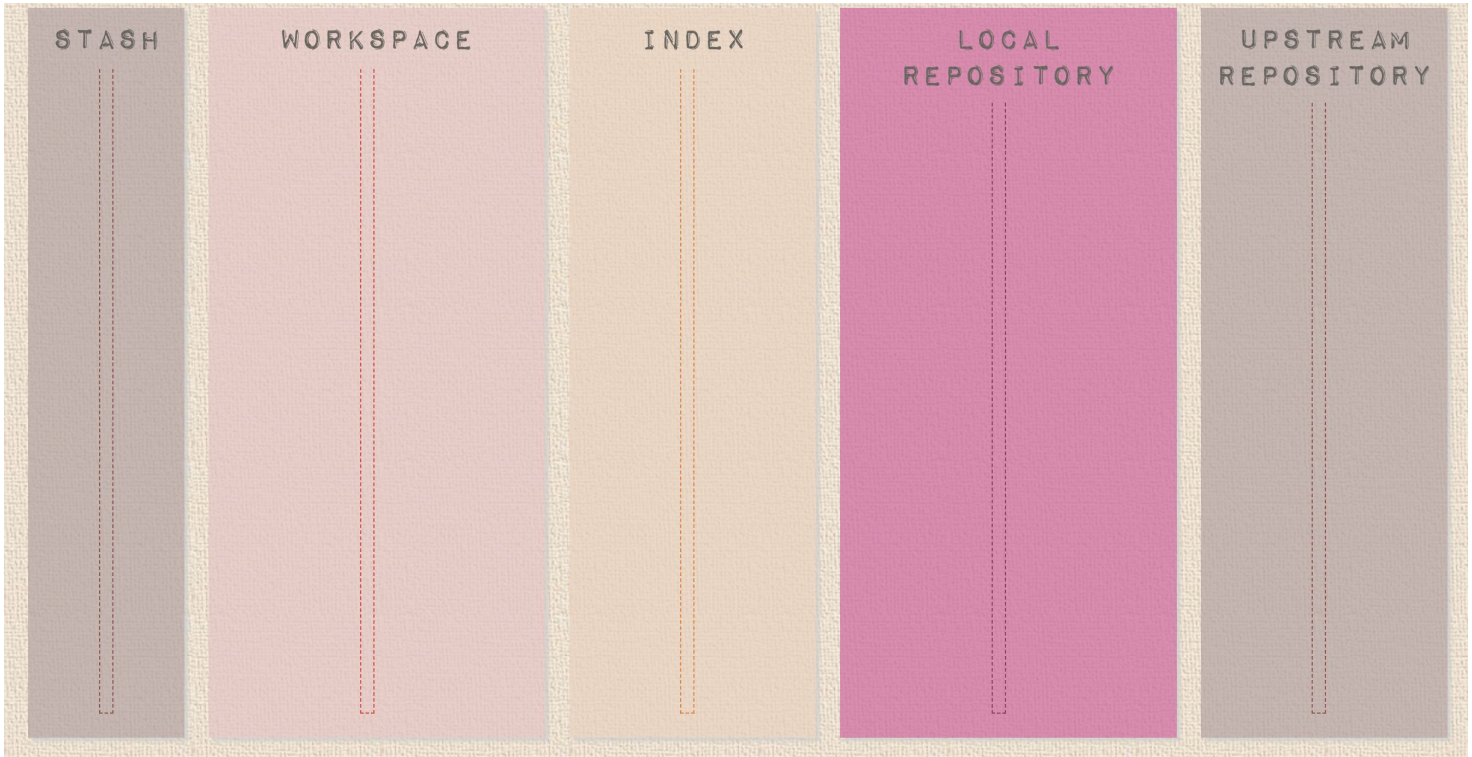
- In version control systems like Git, changes made to a file in your workspace (text editor) can be added to the index (also called the staging area), committed within the local repository, and then pushed upstream to the remote repository.
- Depending on the situation, different commands may or may not be applicable to track, review, or apply the changes to your files in the version control system.

4

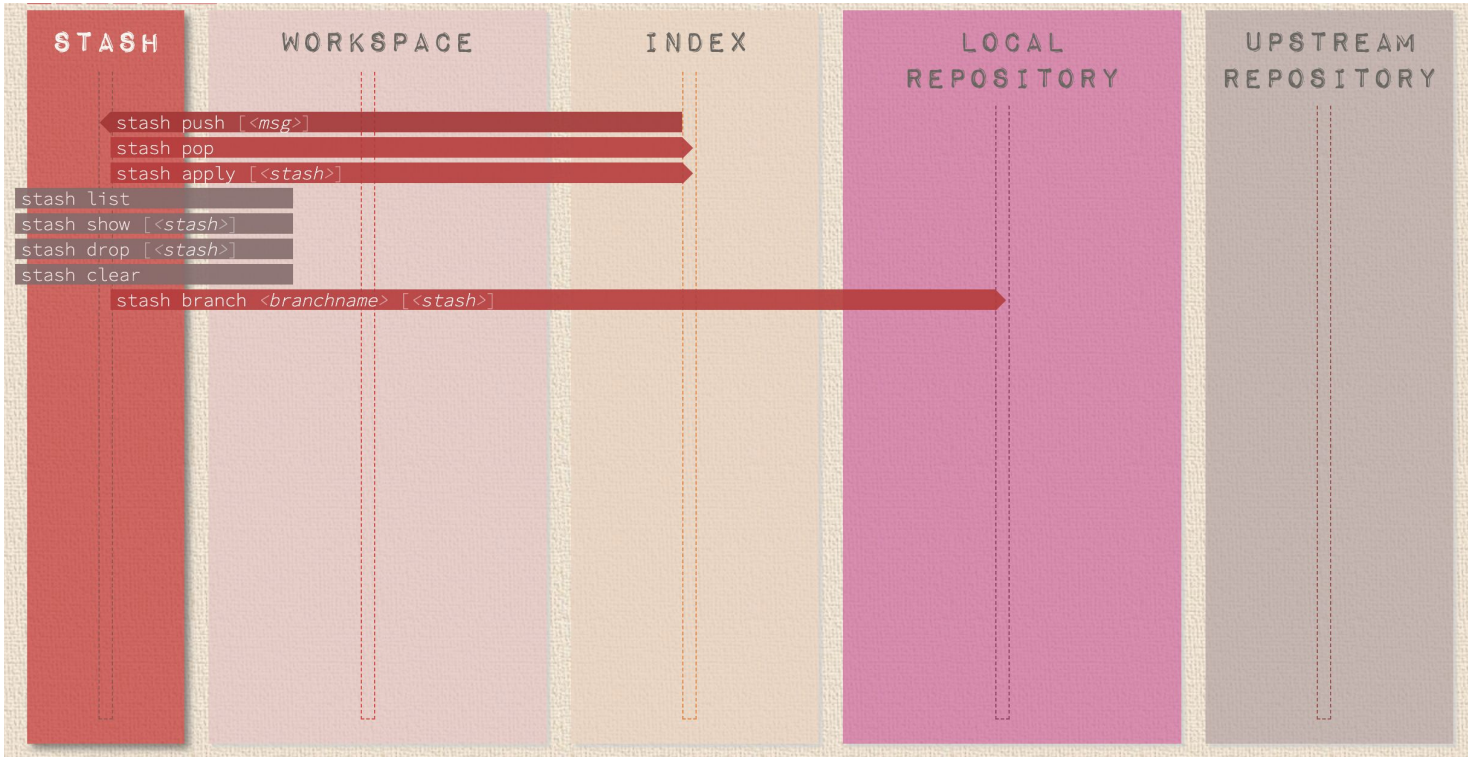
Stash

Understanding the section of data flow concerning the stack of stash[ed] objects

https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo



https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo



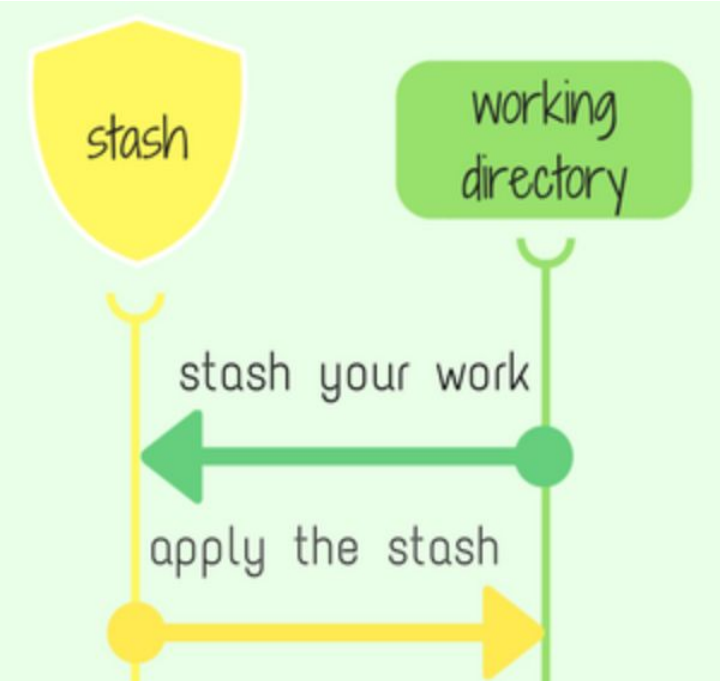
Stash I

- Consider the stash to be: **a shelf that allows you to temporarily store a snapshot of your changes without committing them to the local repository.**

- **NOTE**: Stashed changes are available from any branch (**we will cover branches later**) within that repository.
- **NOTE**: By default, `git stash` will stash only staged and modified tracked files.
  - Staging a file using git add apple.js marks it for commit.
  - Modified but not staged files are changes made to tracked files that have not been staged.
  - Untracked files include those specified in the .gitignore file and brand new files that have not been staged yet. Untracked files have not been included in any commits.
- **NOTE**: Using the stash can be useful but is not always essential for developers. You can develop an entire full-stack web application without needing to interact with the stash.

Stash II

- Use `git stash` when you want to save and clear your changes, returning to a clean working directory
- The `git stash` command saves your local modifications and reverts the working directory to match the latest commit. You can restore the stashed changes using `git stash` apply, potentially on top of a different commit.
    - Using `git stash` without arguments is the same as `git stash push`.
- By default, a stash is listed as "WIP on branchname..." to indicate work in progress. However, you can provide a more descriptive message on the command line when creating a stash.



Stash III

- **General Scenario:** Leveraging the stash can be helpful in the event that you have changes in your workspace that you do not want to presently commit, but also do not want to permanently discard either.
  - And, for this moment in time, you require a clean workspace (how the workspace existed prior to your stashable changes) to operate with.

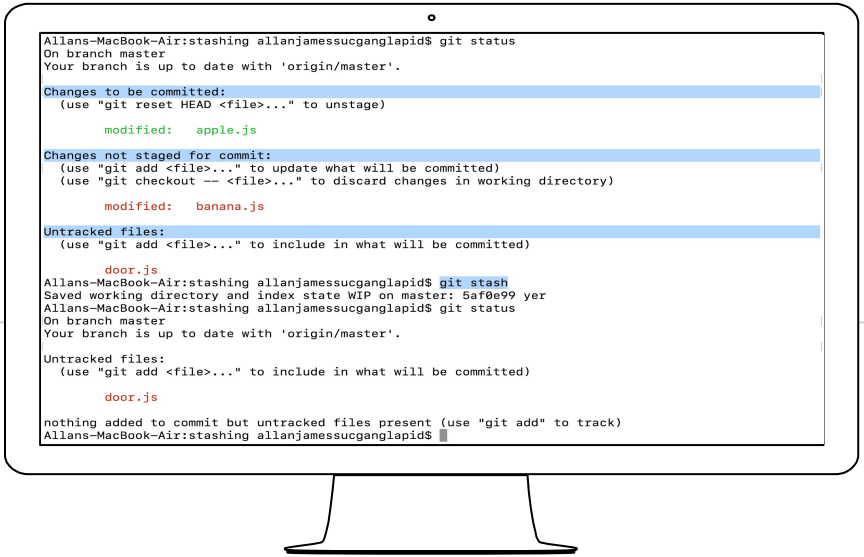
- **Scenario A:** After suddenly realizing the most recent work you are doing is a bit off-topic for the branch (**we will cover branches later**), you can stash the changes and then apply them in a more fitting commit later and elsewhere.

- **Scenario B:** After suddenly realizing the most recent work you are doing has a more plausible alternative approach, you can stash the changes and then try to experiment with the alternative idea knowing you stashed away your initial idea

- **Scenario C:** An urgent matter in another [feature] branch arises, and you need to stash what you're working on because it is incomplete (commits are serious business --- we only commit working, thorough, finished code!)

Stash IV

```
git stash list [<options>]
git stash show [<options>] [<stash>]
git stash drop [-q|--quiet] [<stash>]
git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]
git stash branch <branchname> [<stash>]
git stash [push [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet]
             [-u|--include-untracked] [-a|--all] [-m|--message <message>]
             [--pathspec-from-file=<file> [--pathspec-file-nul]]
             [--] [<pathspec>…]]
git stash clear
git stash create [<message>]
git stash store [-m|--message <message>] [-q|--quiet] <commit>
```



Long story short...

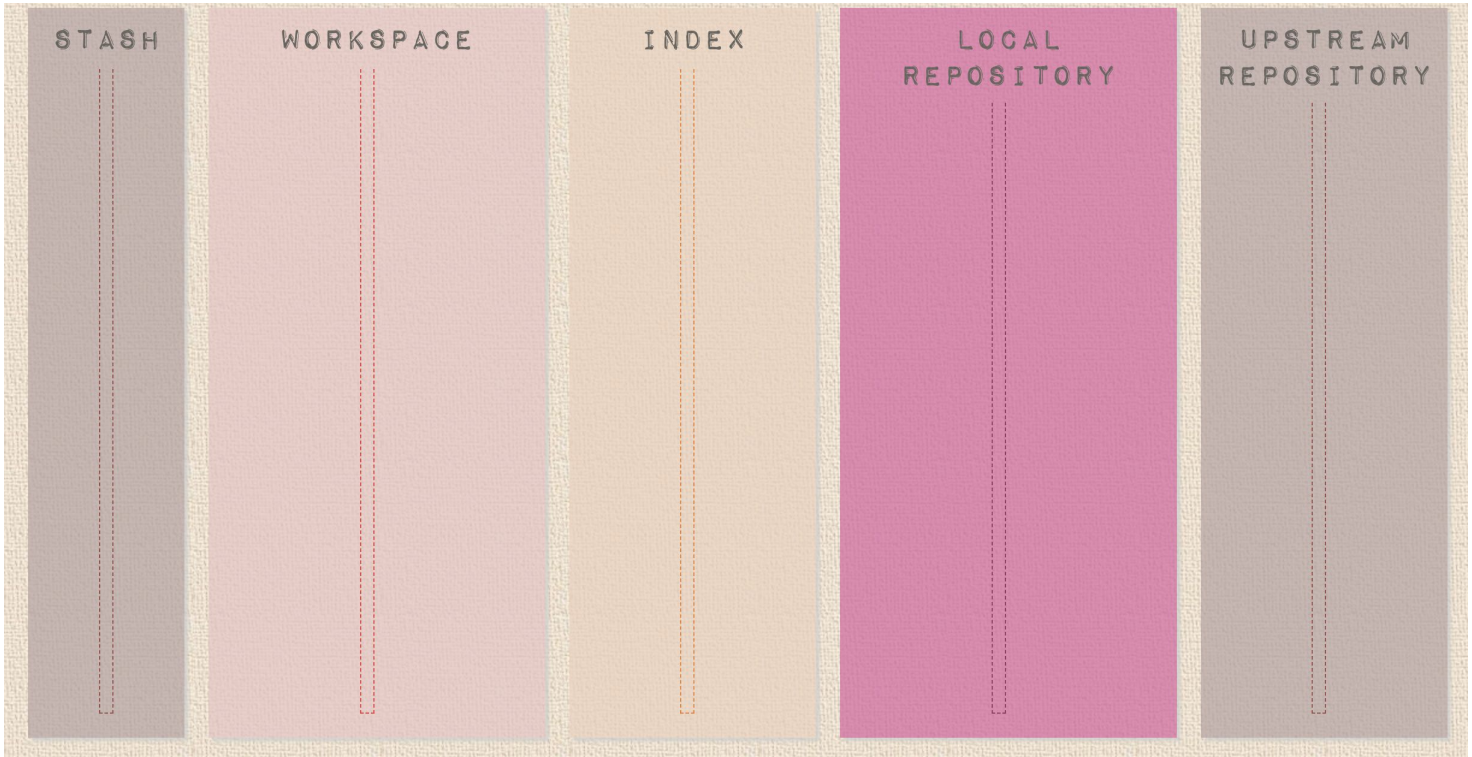
- The stash is a convenient mechanism, located in .git/refs/stash, that allows us to temporarily or permanently set aside changes.
- `git stash` saves both "changes to be committed" and "changes not staged for commit," bringing your workspace back to the state of the latest commit (HEAD).
- `git stash apply` will apply the stashed changes
  - If a file, let's say File X, has both incoming stashed changes and new changes made since the initial git stash call, conflicts or errors may arise. The computer cannot make assumptions about which change(s) to include, so these conflicts in File X need to be manually resolved by the developer. More details on resolving Git conflicts will be covered later in the course.

5

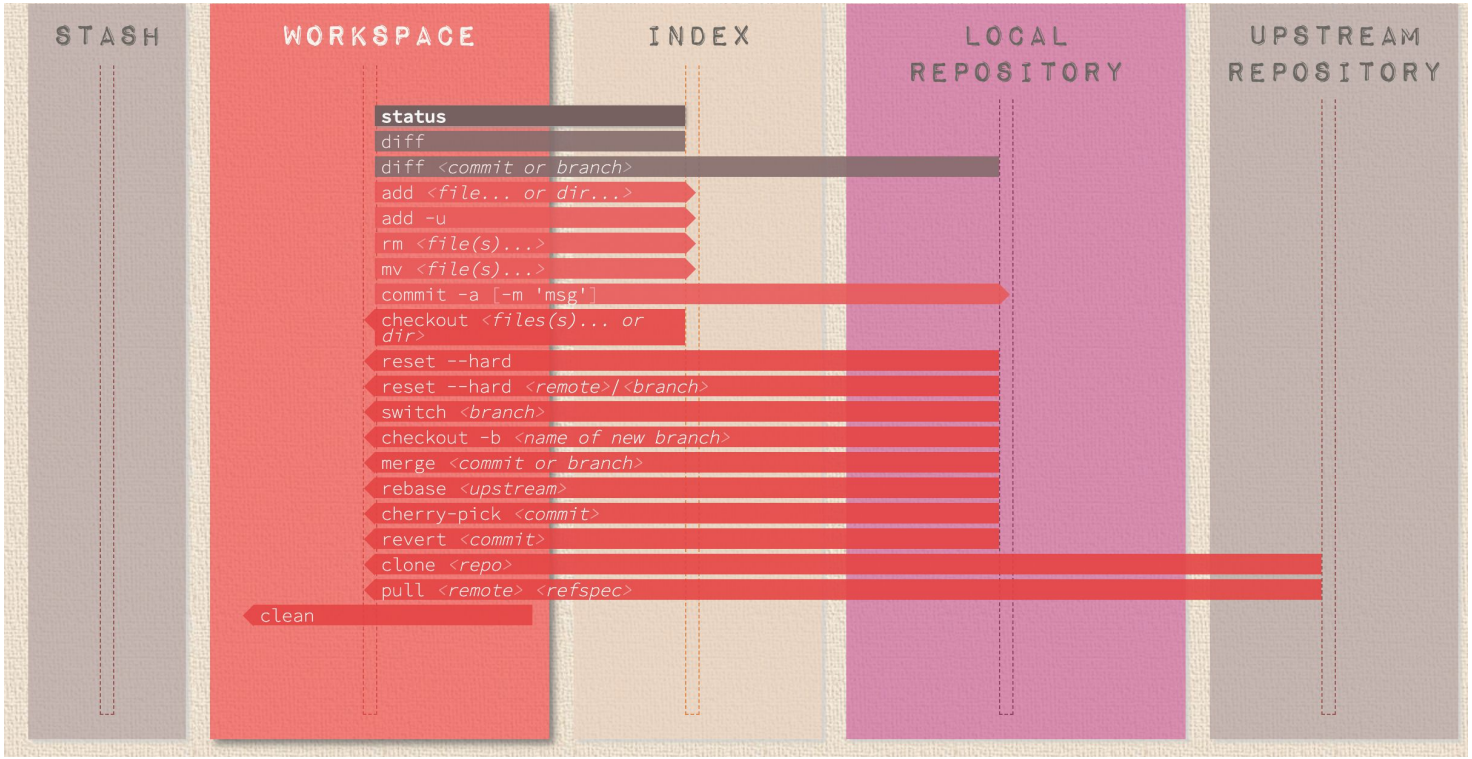
Workspace

Understanding the section of data flow concerning the entity of a workspace (also known as the working directory/working tree/working area)

https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo



https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo



Workspace I

- Consider the workspace to be: **a sandbox where you can make modifications to folders and files* before committing them to the index/staging area**
  - ***If working on a cloned repository: These folders and files are pulled out of the compressed database in the hidden Git directory and placed on disk for you to use or modify
  - ***If working on a locally generated repository (via `git init`): These folders and files are all of the folders and files in the directory in which you executed the `git init` command

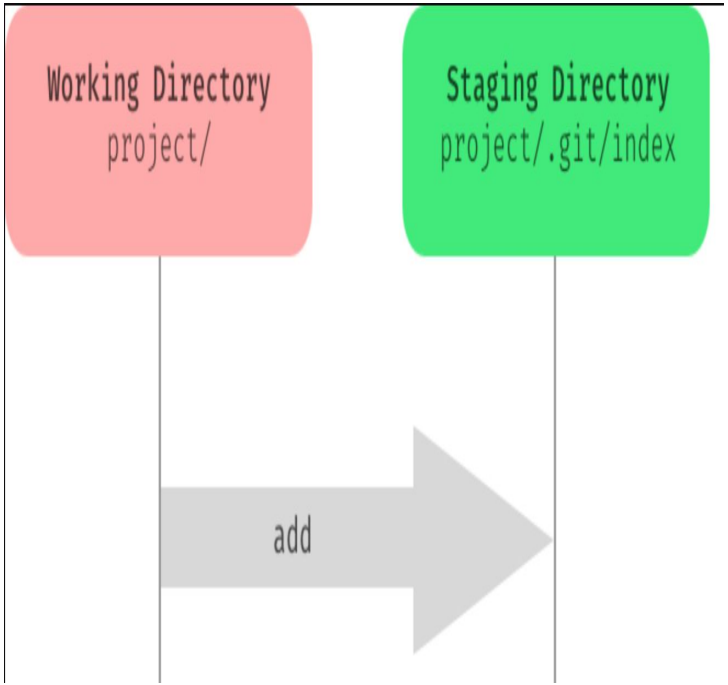
- **NOTE**: Any change --- adding a line, deleting a line, adding a file, deleting a file --- that is not yet included in a `git add` is something that is performed in the workspace

- **NOTE**: All "untracked files" (untracked ≠ ignored) and "changes not staged for commit" form the bulk of the workspace --- technically speaking, "it is a local checkout of a version of your project"

- **NOTE**: Practically speaking, post-git-init, the area in that you work with on your machine, specifically your text editor, is commonly referred to as the workspace/working area/working tree --- aka all files that are currently not in the index/staging area --- the boundary ends after `git add`

Workspace II

- Aside from being the primary location to handle operations involved with branching (making branches, deleting branches, switching branches, merging branches, etc), the workspace is solely responsible for getting changes to travel from the working area to the staging area via a `git add` command.
- Here, you will **modify** content and then **add** it to the index/staging area, thereby staging the changes you want to be included in the next commit.



Workspace III

- **General Scenario:** The workspace is where you will be spending a majority of your time when developing for a project. You can use the workspace to instantiate and prepare any changes that need to be included in the next commit. Some commonly associated non-branch commands with this area of development with Git are: `git status`, `git log`, and `git pull`
- **Scenario A:** I want to become more acquainted of my progress so far and I want to be sure I have an understanding of the different states of different files, so I will use `git status` in order to see “untracked files”, “changes not staged for commit”, and “changes to be committed” as well as how far behind or how far ahead my local branch is with *my local copy of the remote counterpart* (if on the master branch, this is origin/master, which differs from origin master)
- **Scenario B:** I will use `git log` to list out the commit history of this project to see a description of the last change
- **Scenario C:** While I was working on a feature, someone else was working on a different feature and uploaded their local changes to the remote repository, and I want to get the most recent updates from a remote repository in order to be up to date before I upload my own local changes to the remote repository --- so I will use `git pull` to update my local copy

Workspace IV

Scenario D

Preface: **Most commands rely on some sort of initial commit to exist at all,** so even though that process is related to the index, we will include that step (1-4) here (first 5 commands in image of terminal)

Situation: There are changes we made since the last commit, but we did not yet add these changes to the index. We have only been working for a little while, and we are totally okay with discarding these changes (NOTE: these will undo any changes in the workspace).

5. Added content to `food.js`
6. Checked the status
7. Used suggested code from the prompt: `git checkout -- food.js`
8. Checked the status to verify removal of workspace changes that were originally under “changes not staged for commit”

Workspace IV

	New Files	Modified Files	Deleted Files	
git add –A	✓	✓	✓	Stage All (new, modified, deleted) files
git add .	✓	✓	✓	Stage All (new, modified, deleted) files
git add --ignore-removal .	✓	✓	✘	Stage New and Modified files only
git add –u	✘	✓	✓	Stage Modified and Deleted files only

Workspace V

```
~/p/p/slides (:gh-pages) git status
# On branch gh-pages
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   css/theme/jr0cket.css
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   getting-started-with-git.html
#       modified:   getting-started-with-git.org
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       images/
~/p/p/slides (:gh-pages) _
```

Image showing the Working Tree (red) and Staging Area (green)

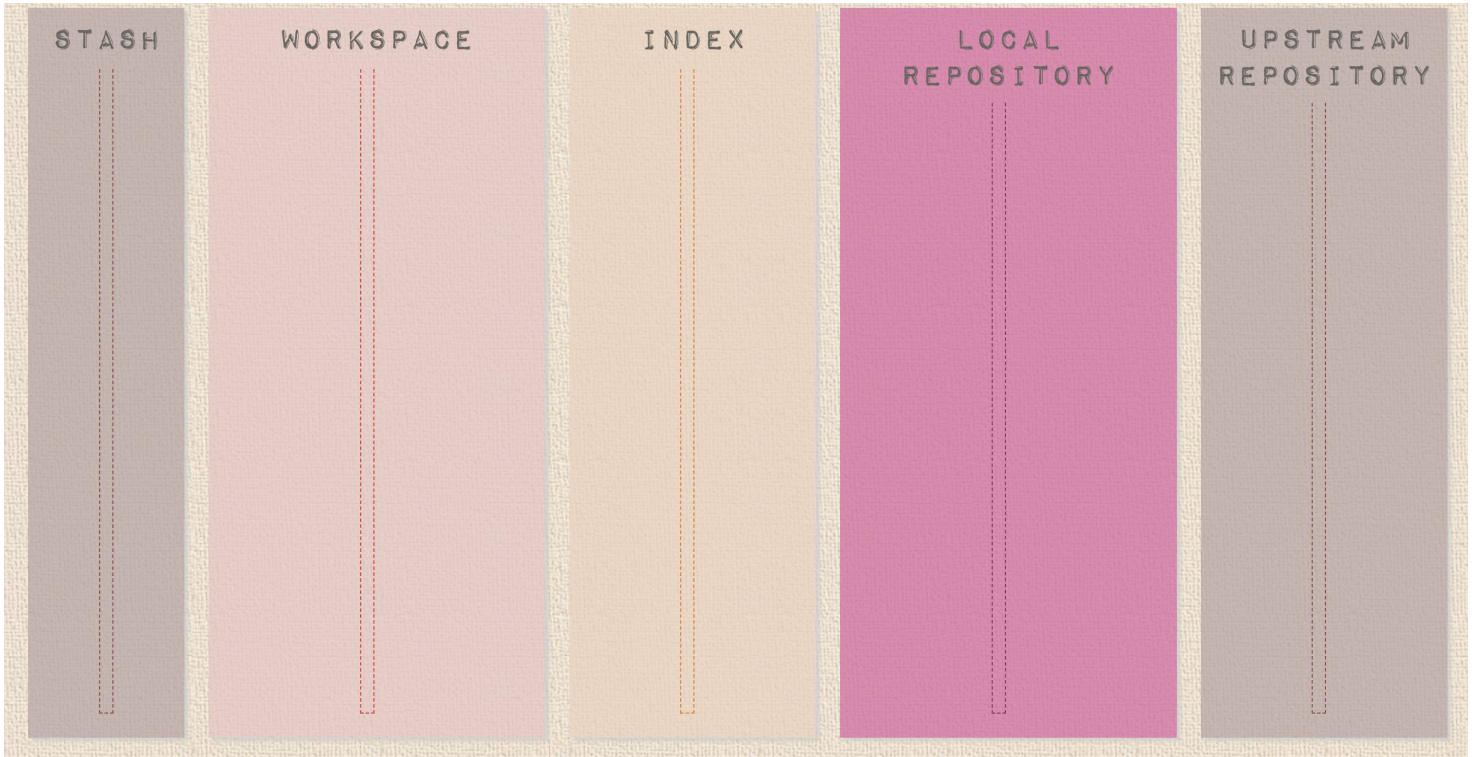
Long story short...

- The workspace consists of files that you are currently working on; you can think of a workspace as a file system where you can view and modify files.

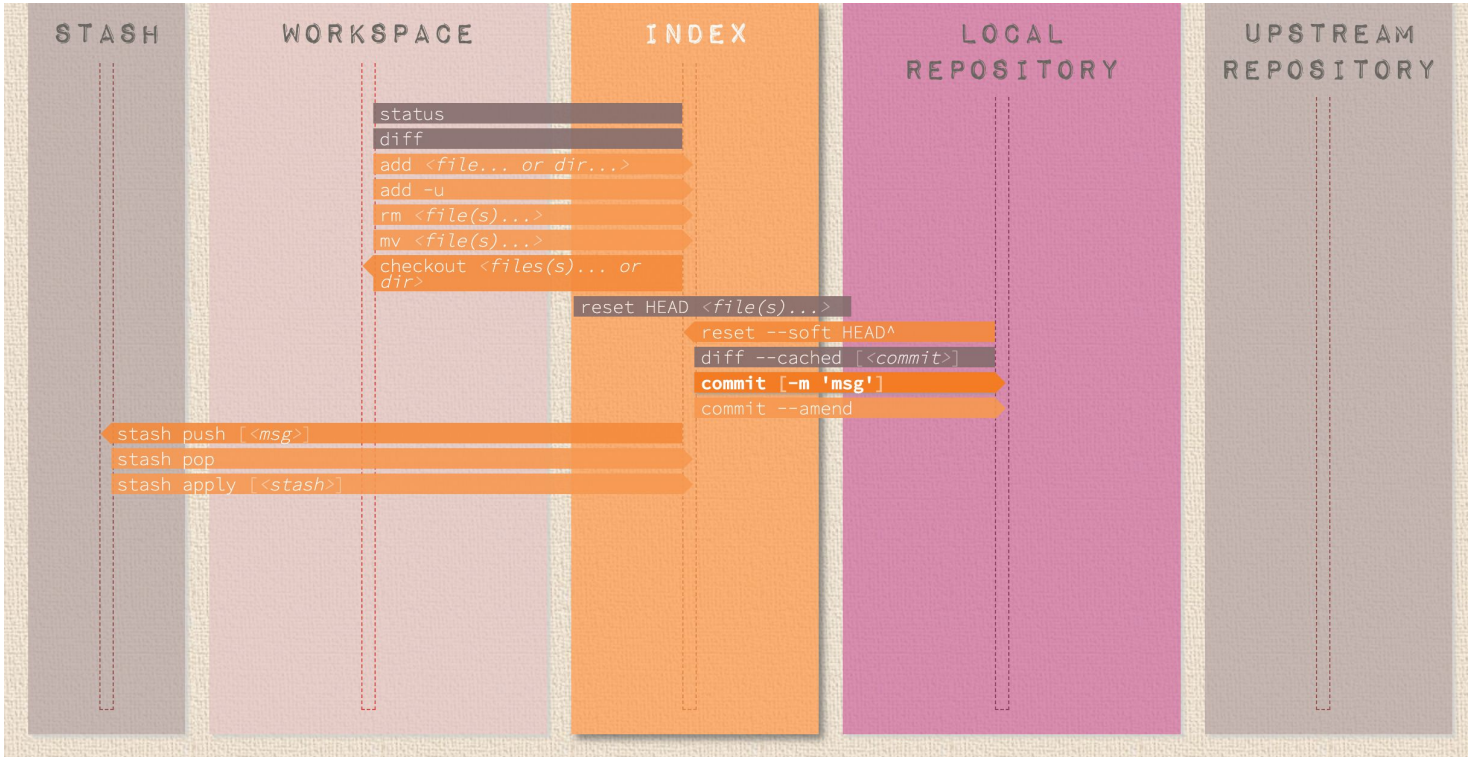
6 Index

Understanding the section of data flow concerning the entity of the index (also known as the staging area)

https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo

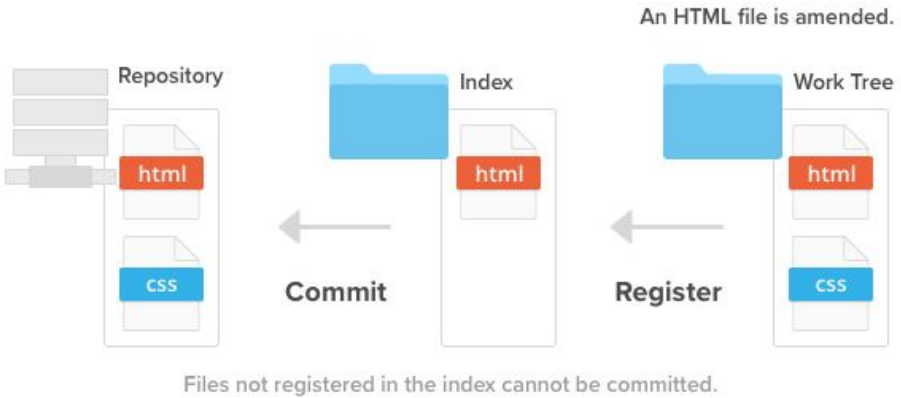


https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo



Index I

- Consider the index to be: **a staging area where commits are prepared**
  - The index compares the files in the workspace to the files in the repository.
  - When you make a change in the workspace, the index marks the file as modified before it is committed.




```
~/p/p/slides (:gh-pages) git status
# On branch gh-pages
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   css/theme/jr0cket.css
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   getting-started-with-git.html
#       modified:   getting-started-with-git.org
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       images/
~/p/p/slides (:gh-pages) _
```

Image showing the Working Tree (red) and Staging Area (green)

Index II

Basic Git Workflow

1) Modify files in the workspace

2) Stage the changes you want to be included in the next commit

3) Commit changes so that the modified files are now safely stored in the repository; committing will take the files from the index and store them as a snapshot in the repository (side note: this is located at: /.git/objects)

Index III

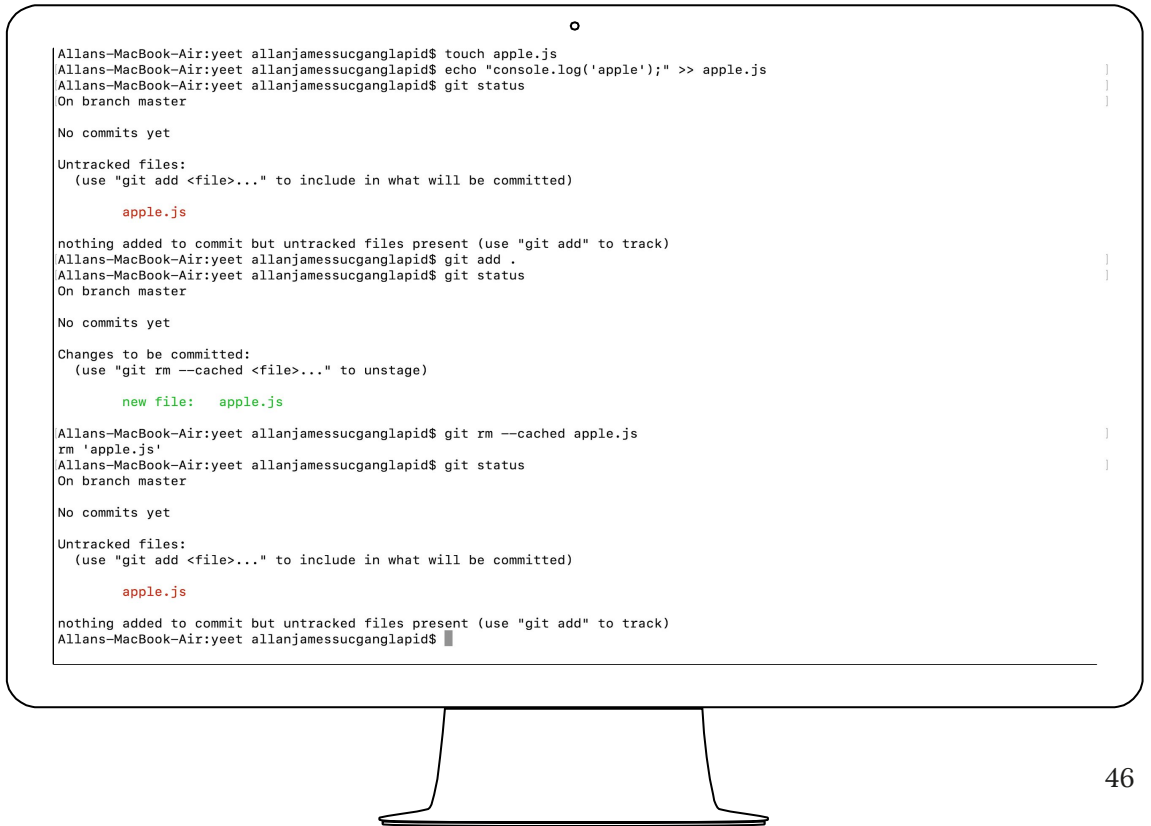
- **General Scenario:** After staging all of my changes, I have committed some changes (which are now stored in the local repository) that are ready to be uploaded to the remote repository associated with this project.
  a. `git add .`
  b. `git commit -m “great commit message”

Index IV

Scenario A

I took these steps to dismount `apple.js` from index, returning to the state of the workspace:

1. Created a file called `apple.js`
2. Wrote a string to the file
3. Check the status for any changes
4. Did **not** commit (at all)
5. *Added file to index*
6. Checked the status to verify step 5
7. *Used suggested code from prompt to unstage file from index (if there was an initial commit, this suggestion would state to use `git reset HEAD apple.js`)*
8. Checked the status to verify step 7
9. Note: `apple.js is back in the workspace, just as it existed prior to step 5

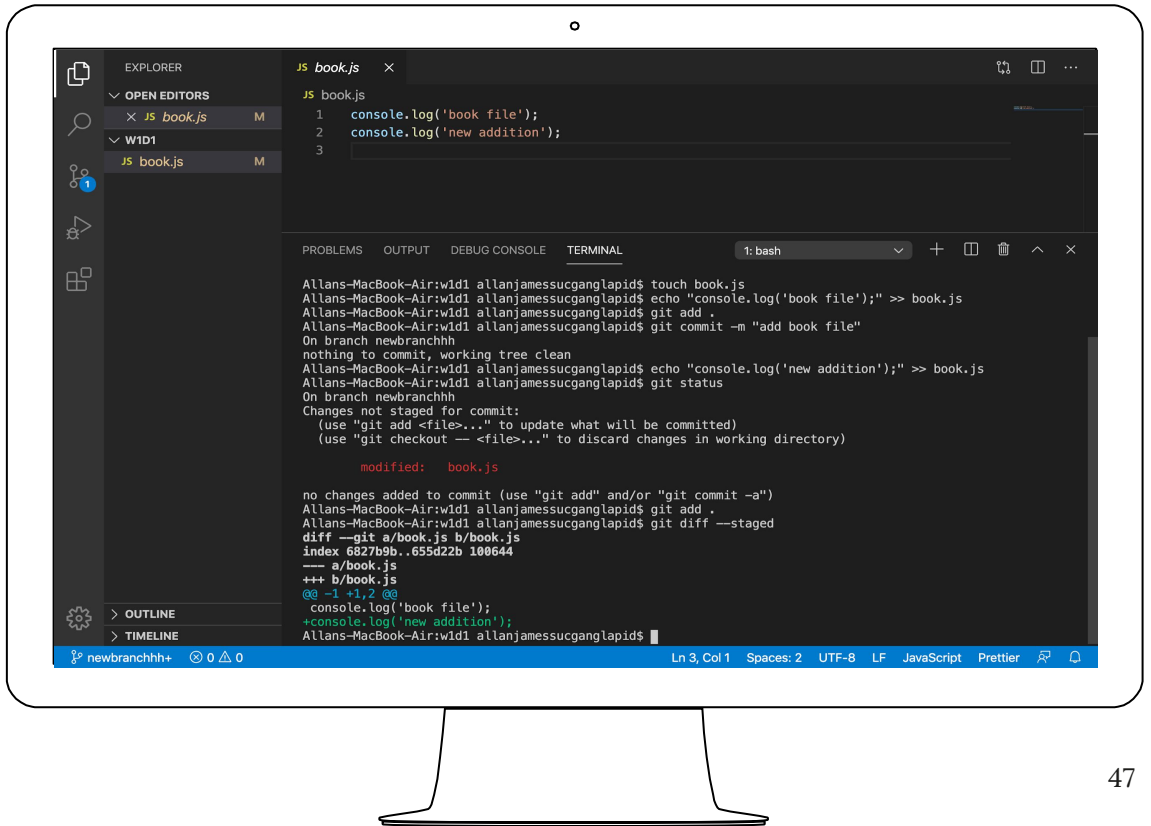


Index V

Scenario B

Some steps to take to see the **diff**erence between the index and the most recent commit, to preview things and in case we need to dismount anything from the index:

1. Created a file, `book.js`
2. Wrote a string to the file
3. Staged a change
4. *Committed the change*
5. Made a **new** change (added a string)
6. Checked the status
7. *Staged this new change*
8. Executed `git diff --staged` to compare the index (Step 7) with the most recent commit (Step 4)

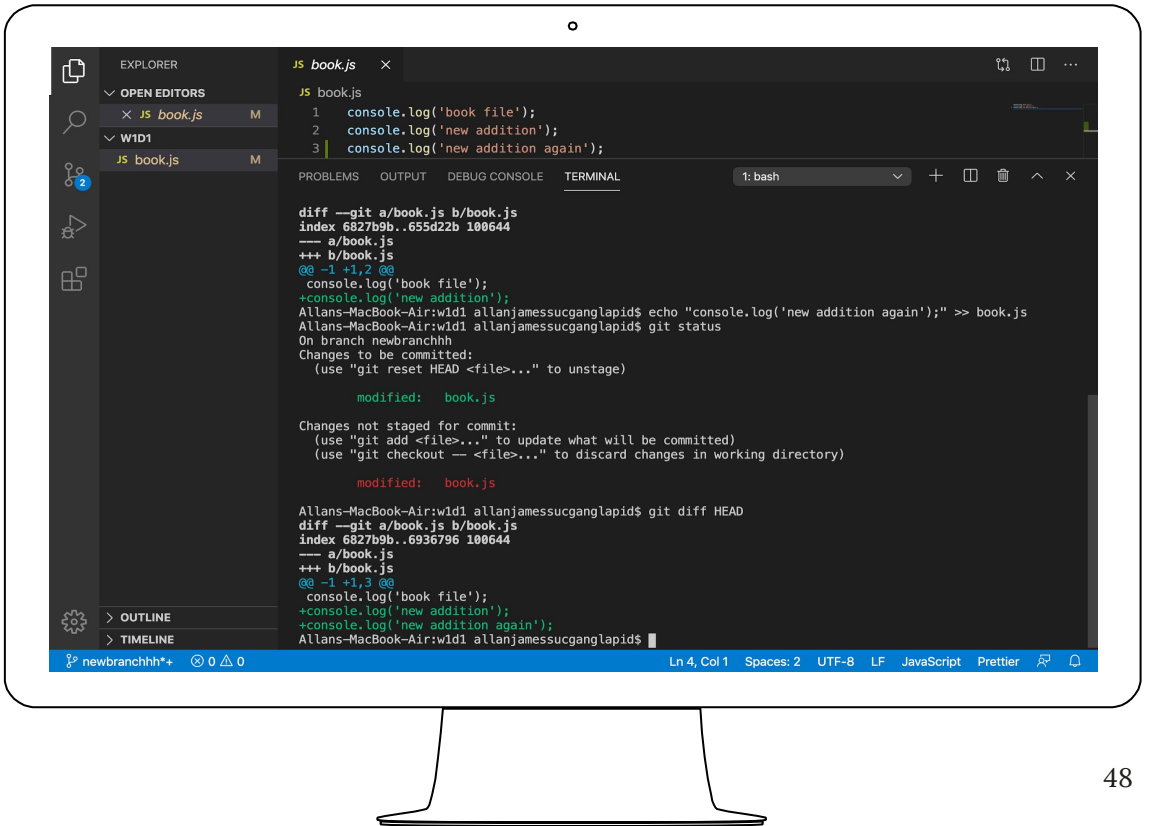


Index VI

Scenario C (continuing from Scenario B)

Here are additional steps in order to see **diff**erence between the index + modified, tracked files (unstaged/in the workspace) and typically the most recent commit (conventionally known as HEAD, but more on this in a later slide deck)

1. Wrote a new string to `book.js`
2. Checked the status
3. Neither added it to the staging area nor made any commit with this change
4. Executed `git diff HEAD` in order to accomplish our goal



Long story short...

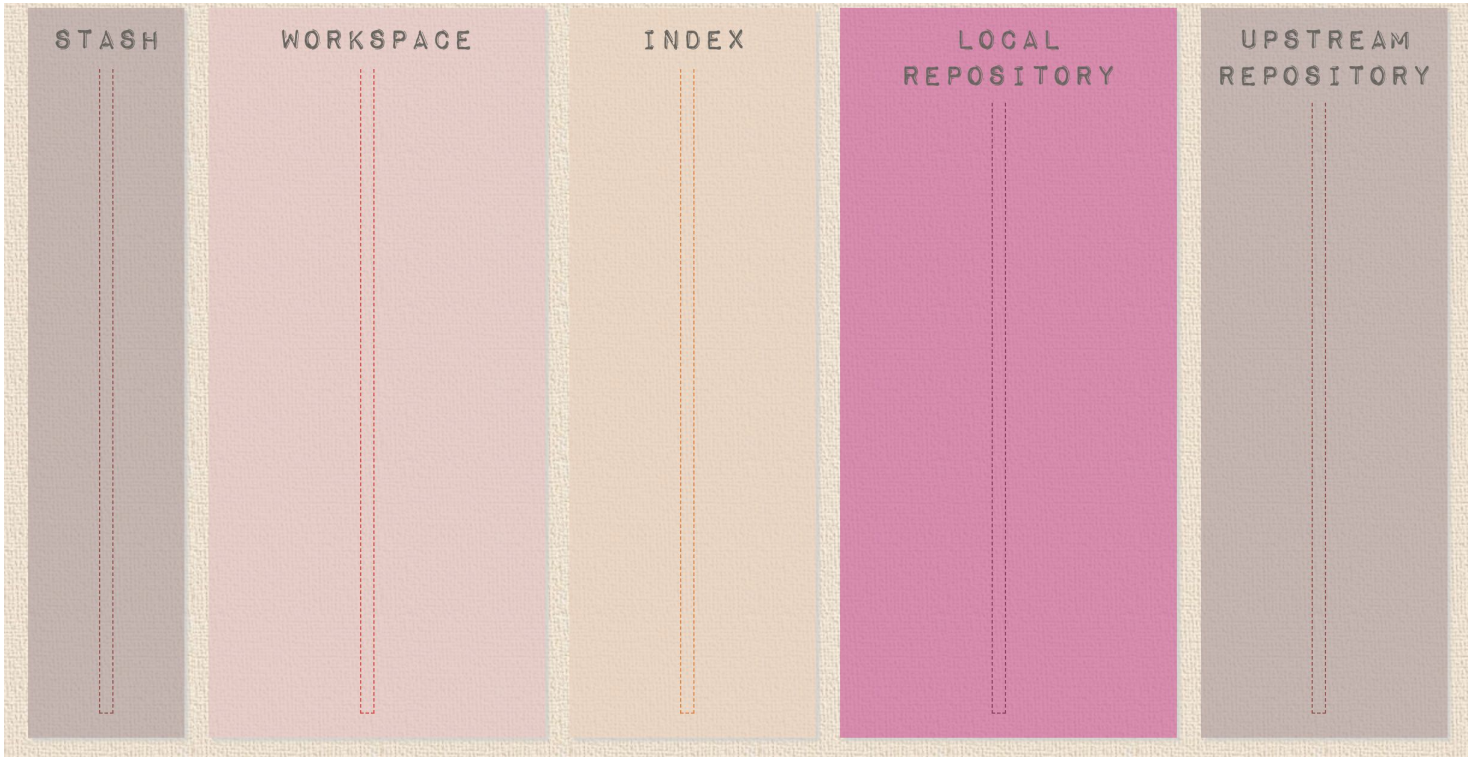
- The index is largely an intermediate step between work that was done in the workspace and information about changes that are saved and then inevitably uploaded via a `git push`
- Files in the git index are files that git would commit to the local repository if you used the `git commit` command

7

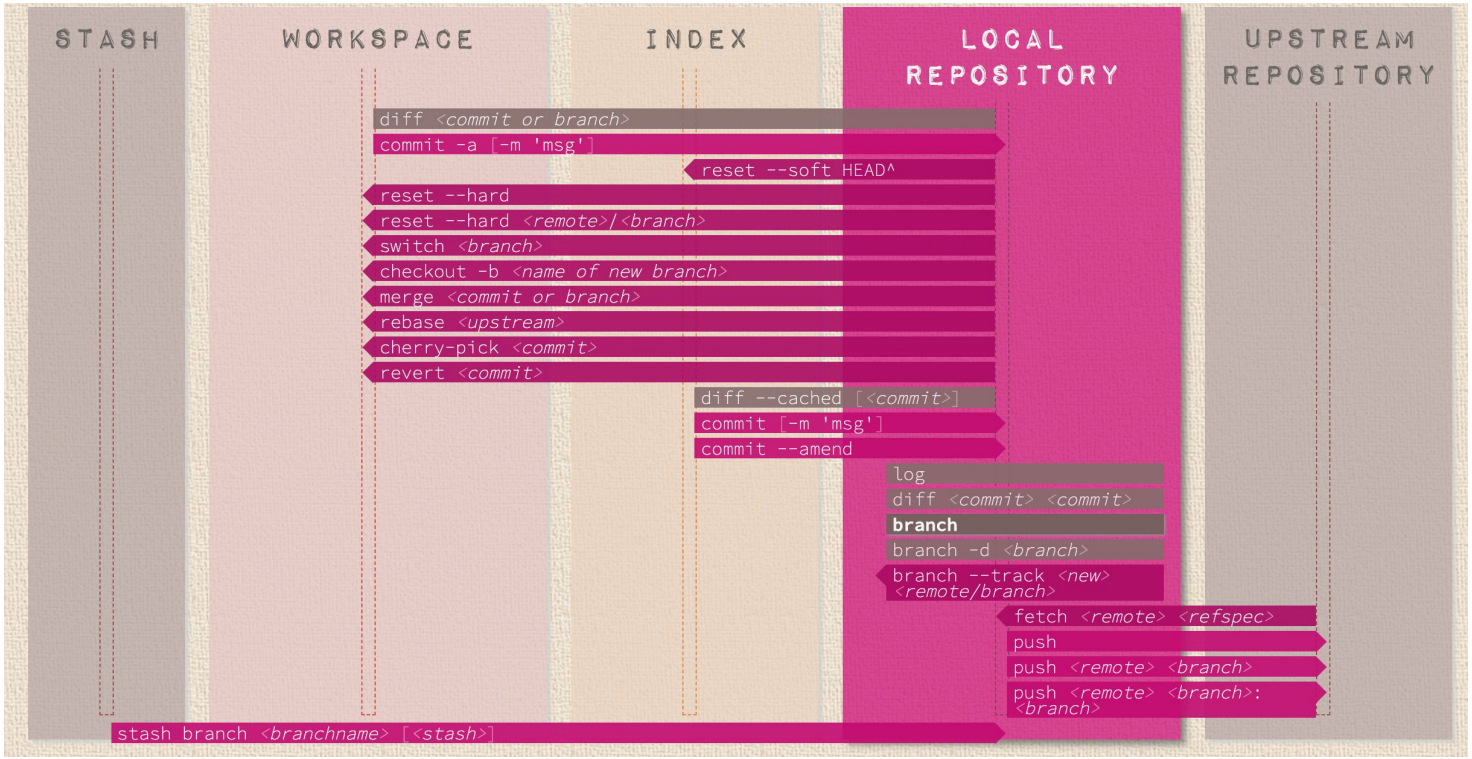
Local Repository

Understanding the section of data flow concerning the entity of the local repository

https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo



https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo



Local Repository I

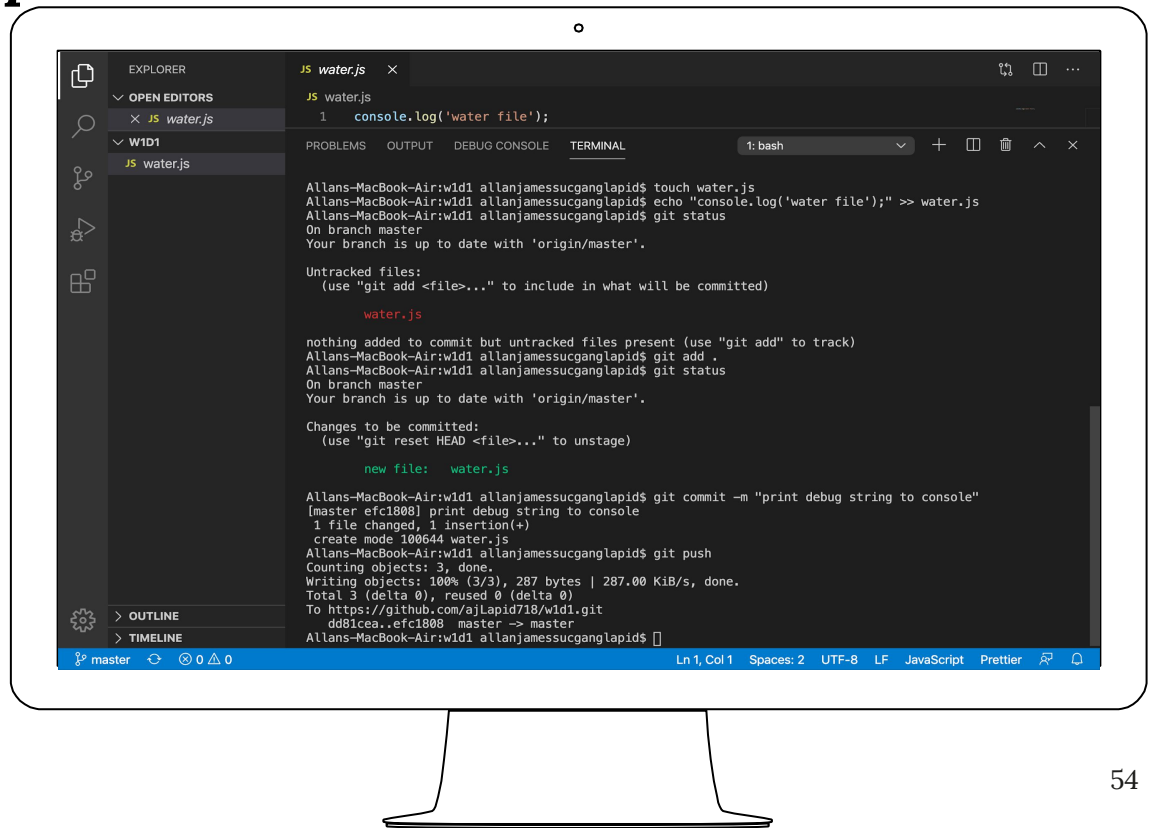
- The local repository encapsulates the stash, the workspace, the index, the committed changes --- anything Git-related, essentially.
- The local repository is on your computer and has all the files and their commit history, enabling full diffs, history review, and committing when offline.
  - This is one of the key features of a “distributed” version control system (DVCS), locally having the full repository history.
- A primary distinction here is what is stored in the local repository after traveling this path: workspace ---> index ---> local repository
  - Committed changes are organized and positioned **here** in the local repository!

Local Repository II

Scenario A

Assuming our project has a pointer to a remote destination, the goal is to make some changes, stage those changes, commit those changes, and then upload those changes to the remote counterpart.

1. Create file, `water.js`
2. Write a string to the file
3. Check status
4. Stage file and changes
5. Check status
6. Commit via `git commit`
7. Push via `git push`
    a. This copies your files from the local repository to the remote repository (**only the changes the remote repository does not have**)

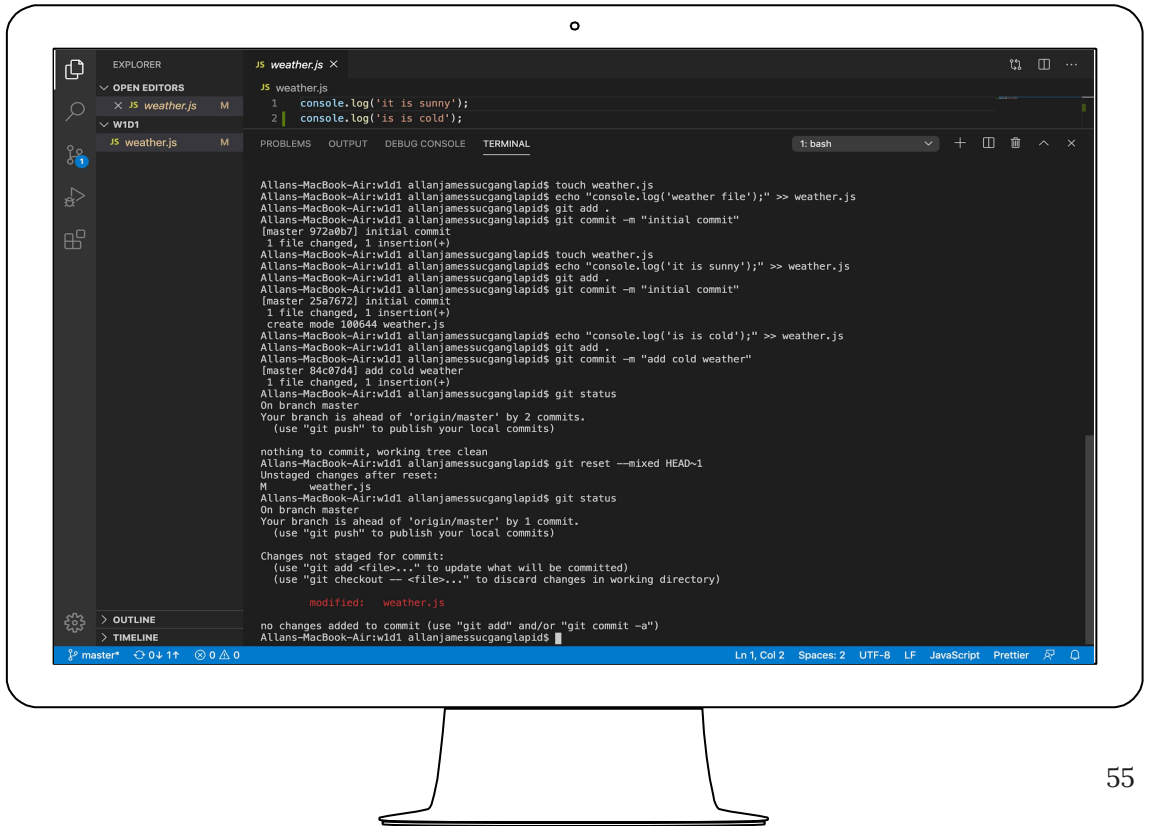


Local Repository III

Scenario B

We accidentally staged a recent change and then committed it. We realized that this commit does not belong as is, for whatever reason. Much like unstaging something from the index, we want to instead, in this case, “travel back” from the local repository to the most recent commit (aka, before we ran git add and git commit).

1. Create file, `weather.js`
2. Write a string to the file
3. Stage file and commit it (1st commit)
4. Write new string to file
5. Stage file and commit it (2nd commit)
6. **Uh oh: `git reset --mixed HEAD~1`**
    a. --mixed will undo staging, but keep workspace in tact (must add and must commit again)
    b. HEAD (effectively refers to the most recent commit, in this case)
    c. tilde (~) indicates relativity, and ~1 says “let’s go back one generation of commits (the parent of the most recent commit aka 1 step back)
        i. Please be careful in cases when commits have more than 1 parent

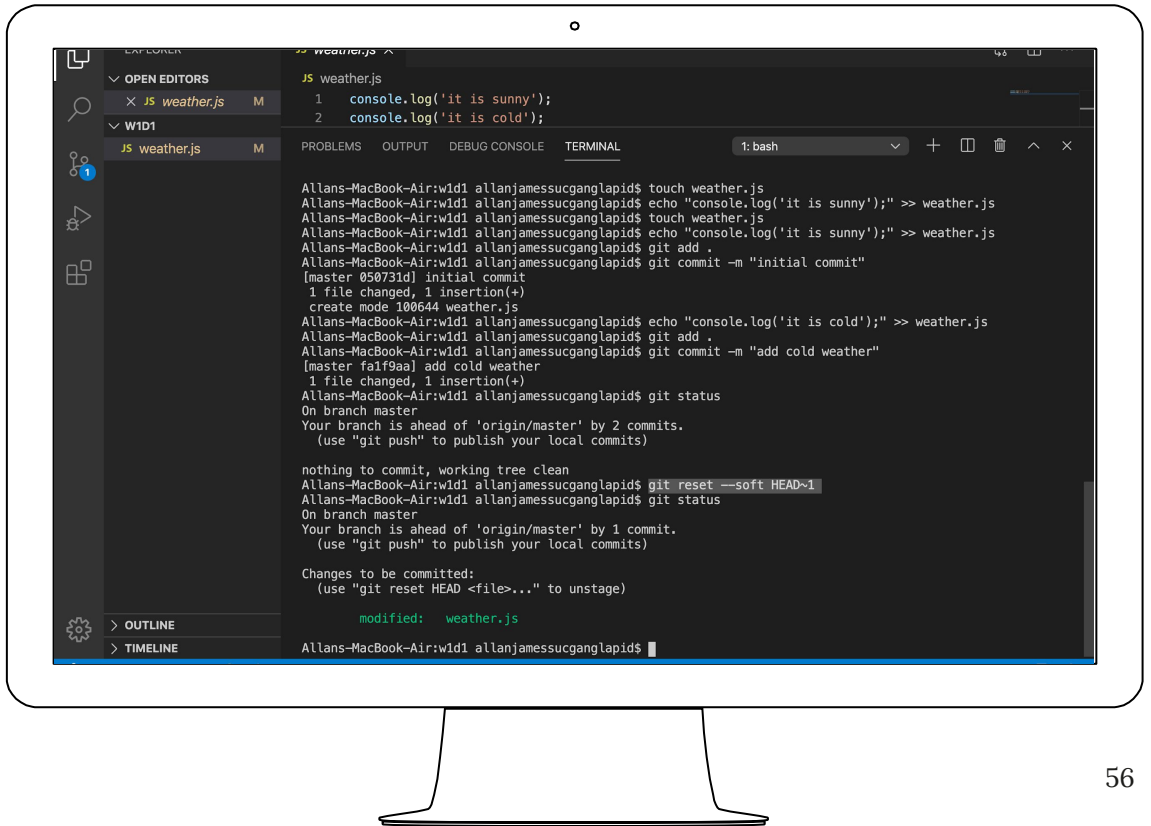


Local Repository IV

Scenario C (similar to Scenario B)

We accidentally staged a recent change and then committed it. We realized that this commit does not belong as is, for whatever reason. Much like unstaging something from the index, we want to instead, in this case, “travel back” from the local repository to the most recent commit (aka, before we ran git commit).

1. Create file, `weather.js`
2. Write a string to the file
3. Stage file and commit it (1st commit)
4. Write new string to file
5. Stage file and commit it (2nd commit)
6. **Uh oh: `git reset --soft HEAD~1`**
    a. --soft will undo commit, but keep index and workspace in tact (you can just git commit again, then)
    b. HEAD (effectively refers to the most recent commit, in this case)
    c. tilde (~) indicates relativity, and ~1 says “let’s go back one generation of commits (the parent of the most recent commit aka 1 step back)
        i. Please be careful in cases when commits have more than 1 parent

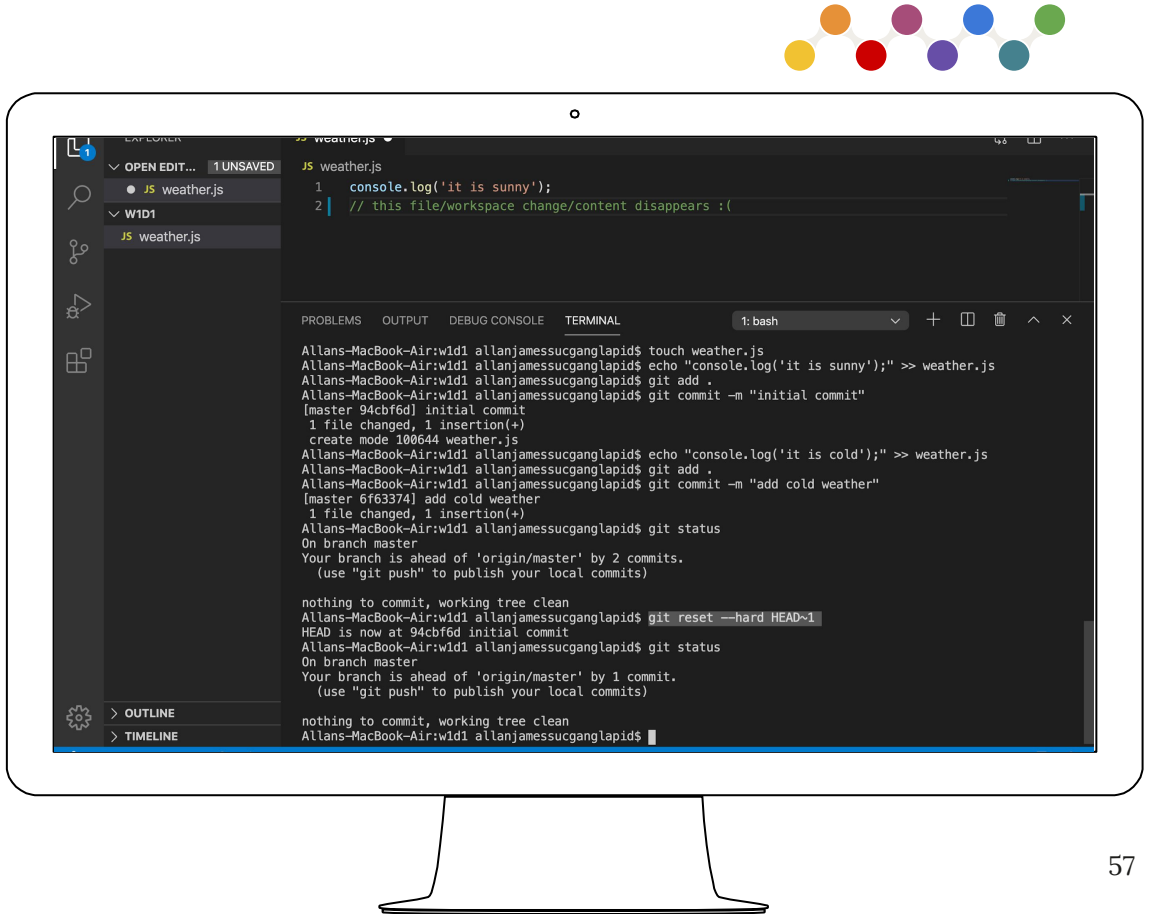


Local Repository V

Scenario D (also similar to Scenario B)

We accidentally staged a recent change and then committed it. We realized that this commit does not belong as is, for whatever reason. Much like unstaging something from the index, we want to instead, in this case, “travel back” from the local repository to the most recent commit (aka, before we ran git add and git commit and saved the file).

1. Create file, `weather.js`
2. Write a string to the file
3. Stage file and commit it (1st commit)
4. Write new string to file
5. Stage file and commit it (2nd commit)
6. **Uh oh: `git reset --hard HEAD~1`**
    a. --hard will undo commit, add, and changes (these will disappear from index and workspace)
    b. HEAD (effectively refers to the most recent commit, in this case)
    c. tilde (~) indicates relativity, and ~1 says “let’s go back one generation of commits (the parent of the most recent commit aka 1 step back)
        i. Please be careful in cases when commits have more than 1 parent or when resetting too far back



Long story short...

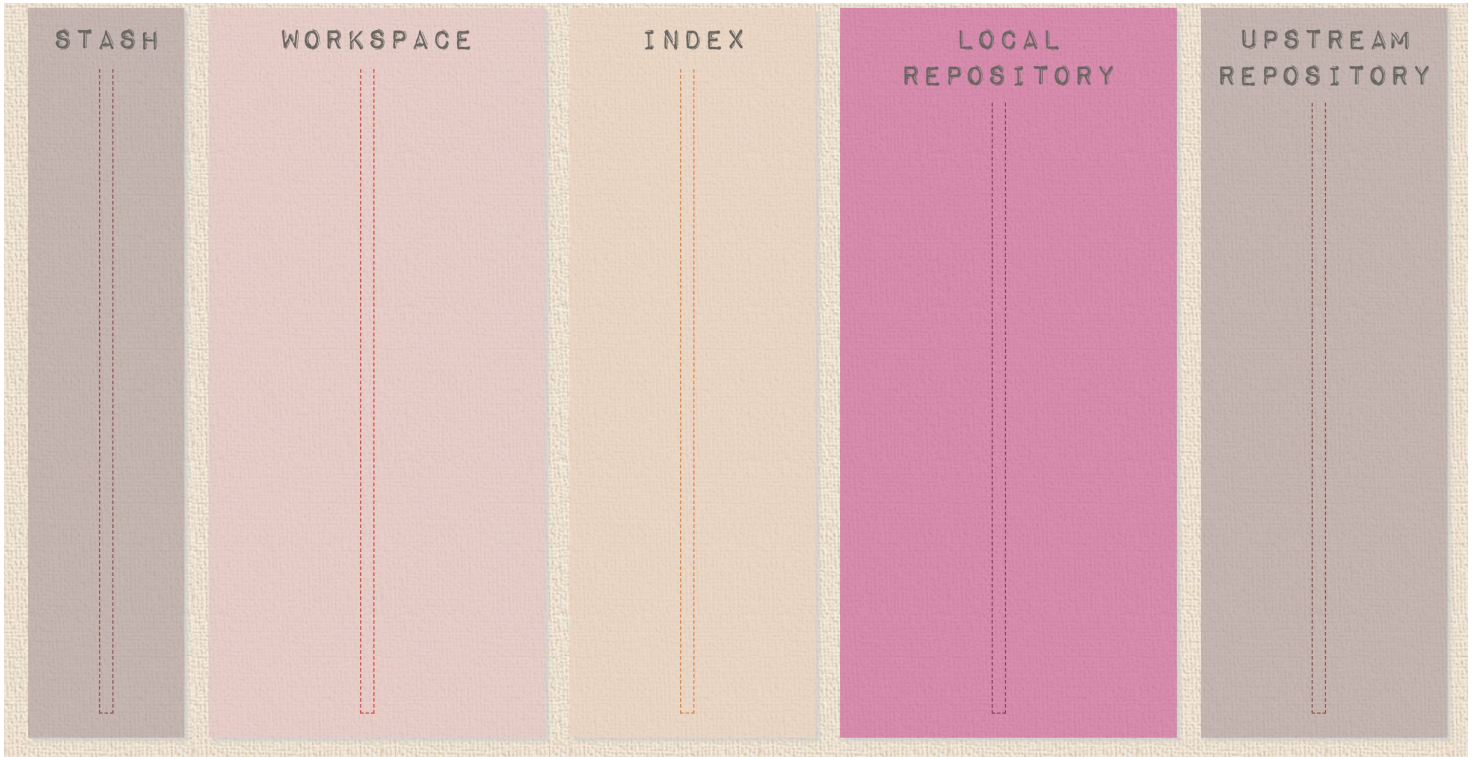
- Practically speaking, the local repository encapsulates the stash, workspace, index, and committed changes --- the entirety of the `.git` directory. You can have the full git experience, with commits, branches, merges, etc, with only a local repository. In a proof-of-concept kind of way, you actually don't need to have a remote repository at all, despite being a common next step,

8

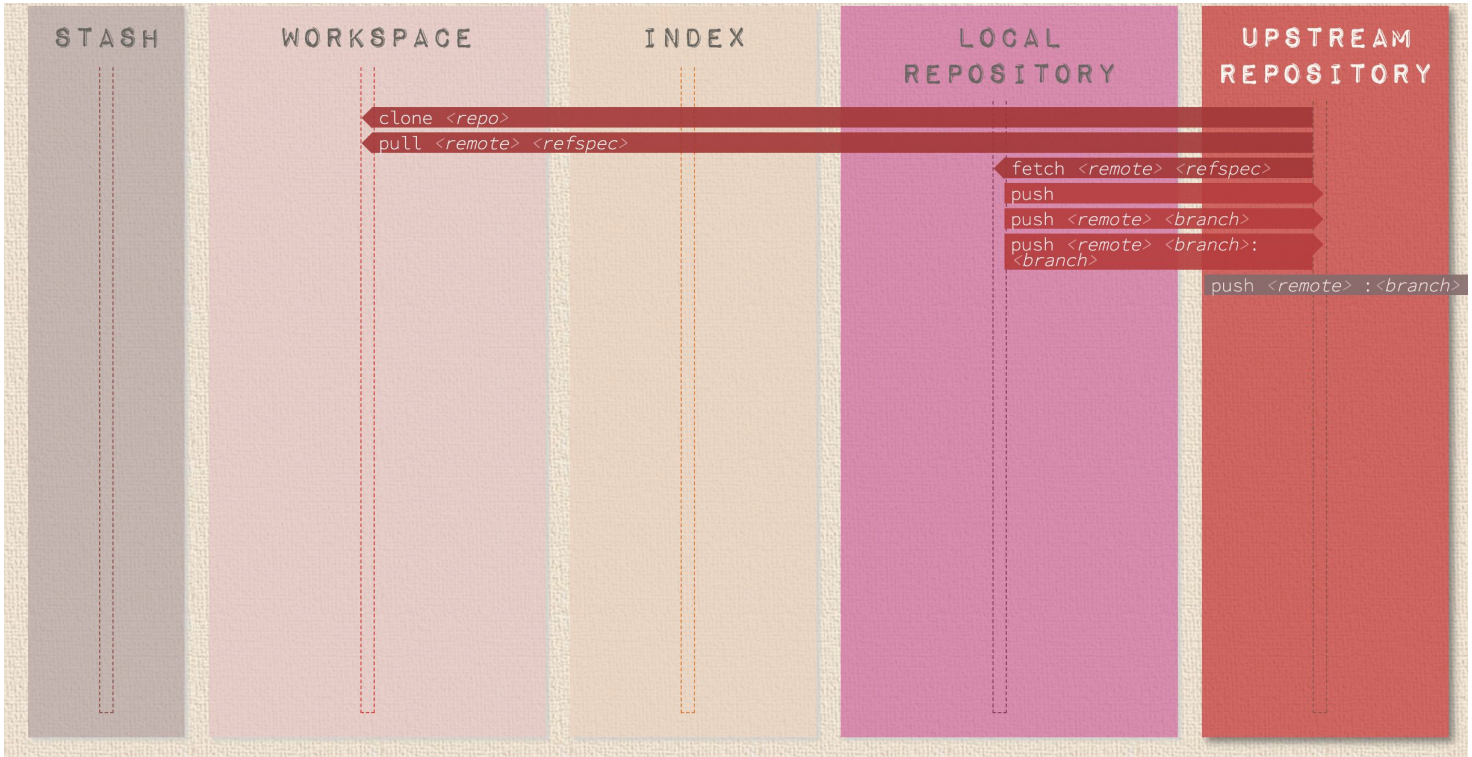
Upstream Repository

Understanding the section of data flow concerning the entity of the upstream repository (also colloquially known as the remote repository)

https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo

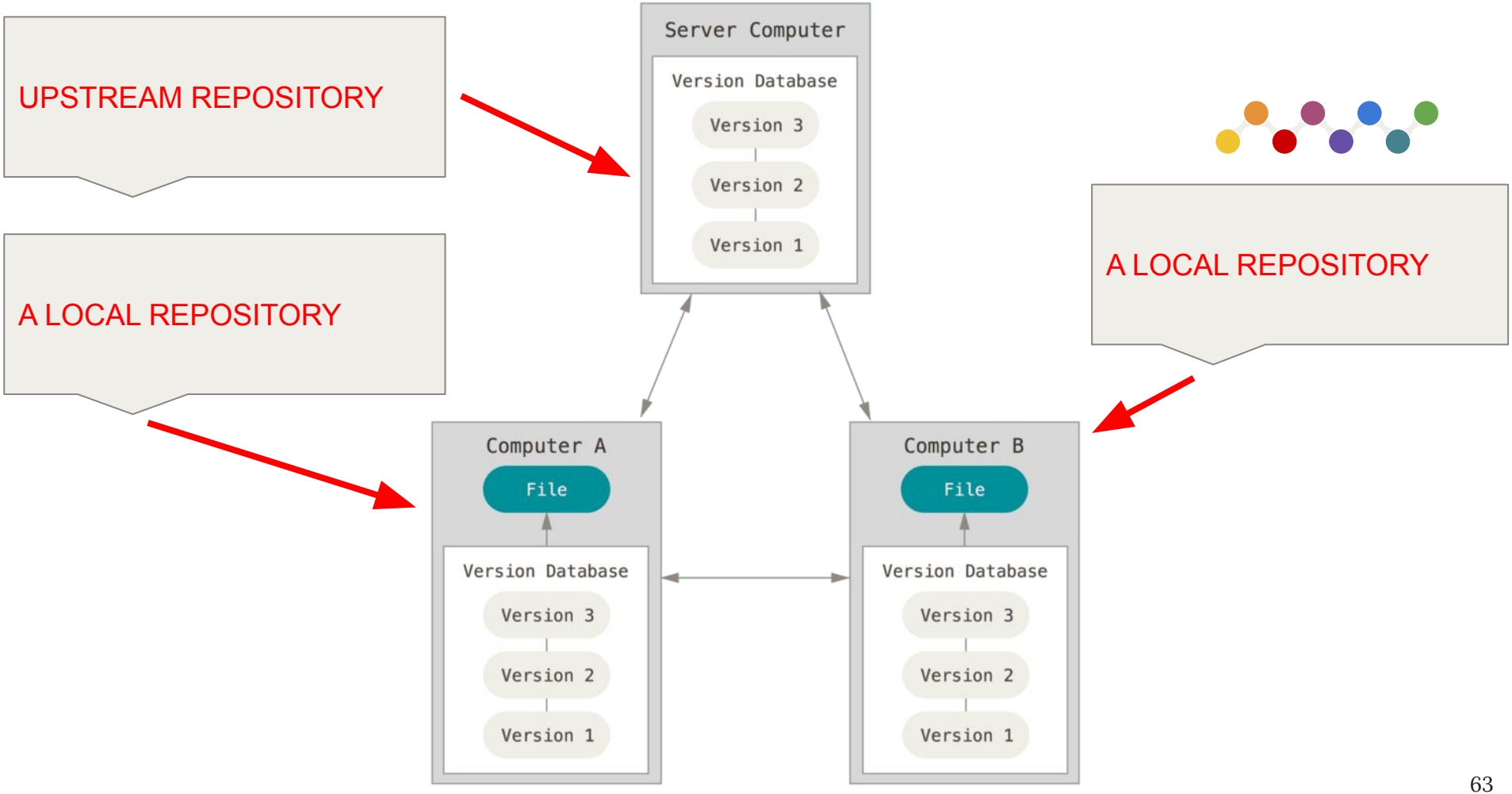


https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo



Upstream Repository I

- DVCS-supporting platforms (such as GitHub) have a collaborative approach to development depends on publishing commits from your local repository for other people to view, fetch, and update
- The upstream repository houses versions of your project that are hosted on the Internet or network (in our case, GitHub), ensuring all your changes are available for developers with the appropriate permissions and privileges
- The default name, or way to reference the upstream repository is conventionally: origin


UPSTREAM REPOSITORY
A LOCAL REPOSITORY
A LOCAL REPOSITORY
Server Computer
Version Database
Version 3
Version 2
Version 1
Computer A
File
Version Database
Version 3
Version 2
Version 1
Computer B
File
Version Database
Version 3
Version 2
Version 1

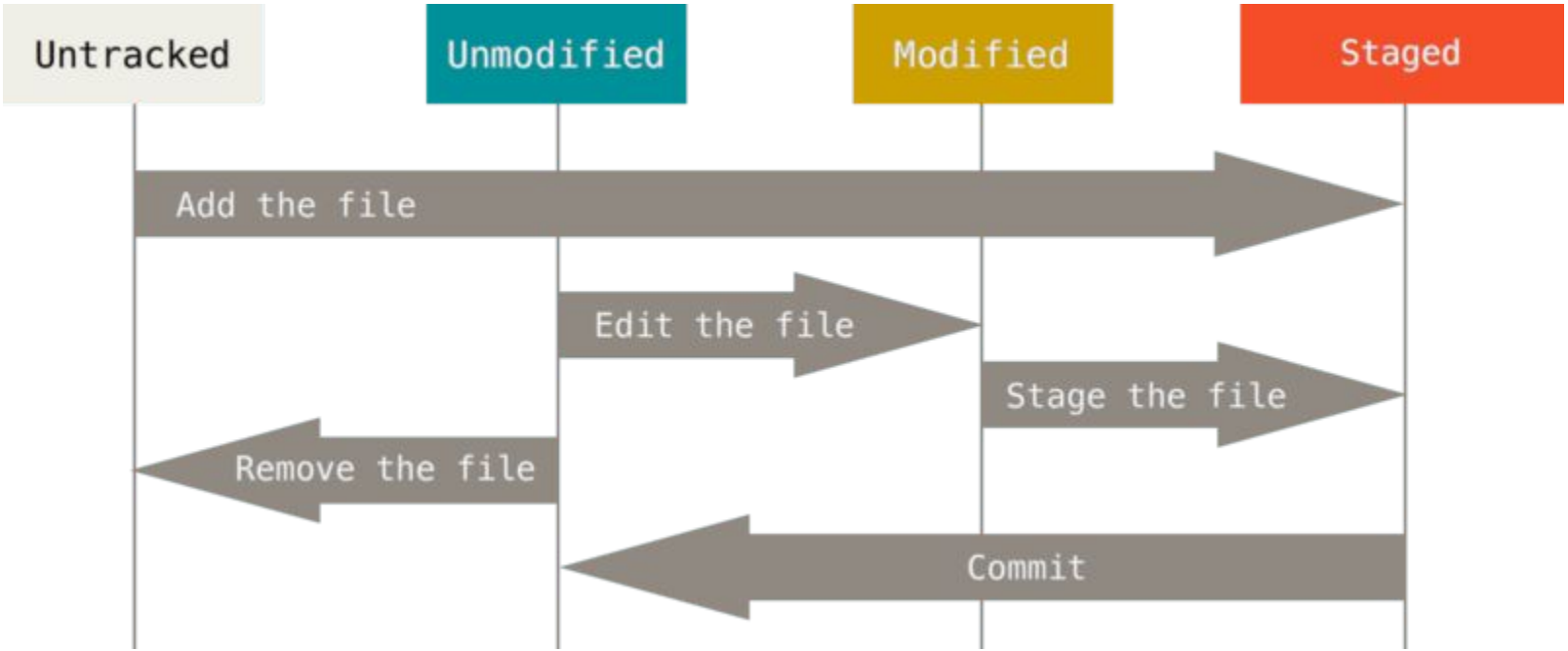
Long story short...

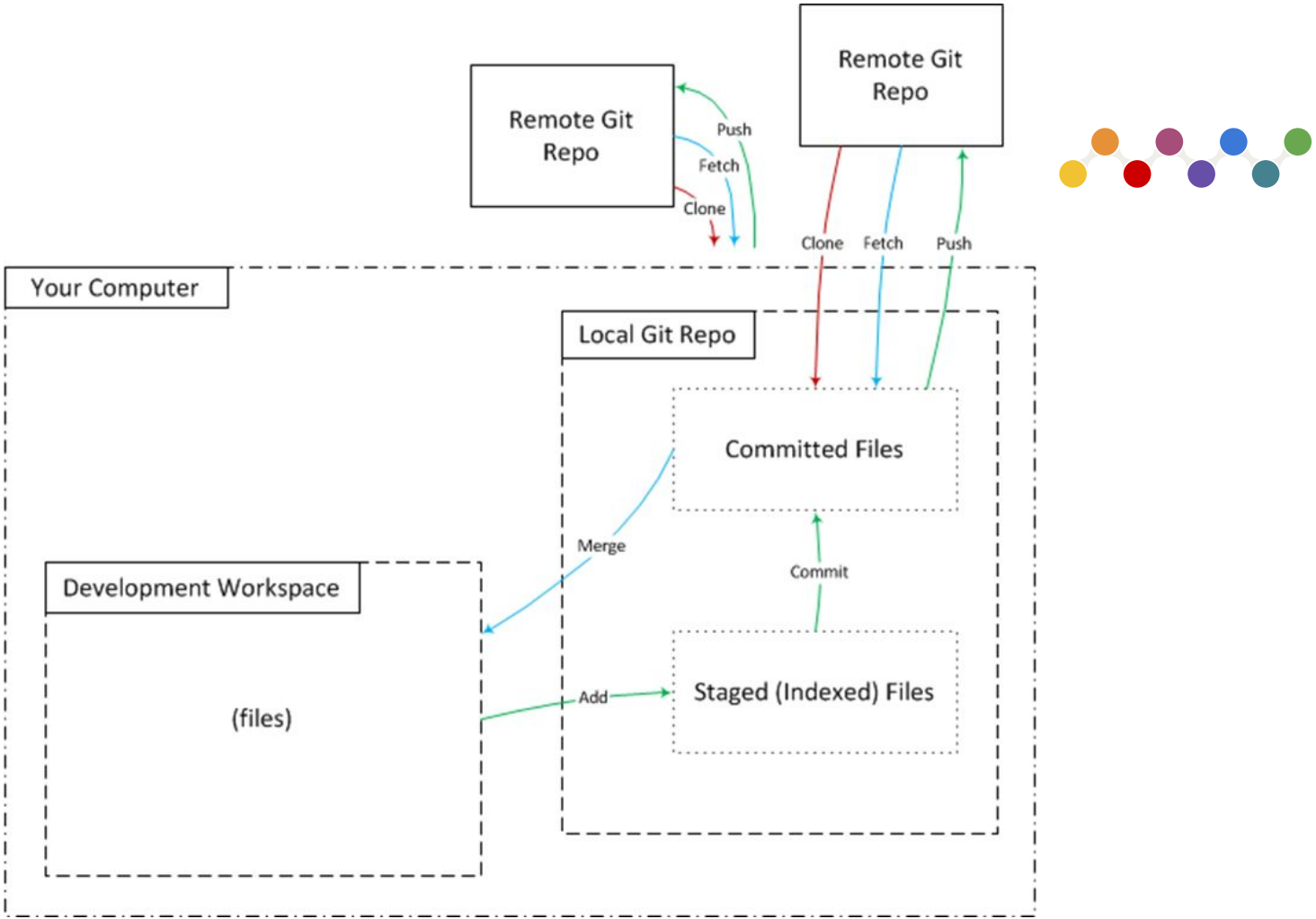
- The upstream repository should be effectively treated as the single source of truth for projects, and thanks to the nature of DVCS, it is a common base of information that multiple developers can pull information from and push information to after making modifications on their local repositories

9

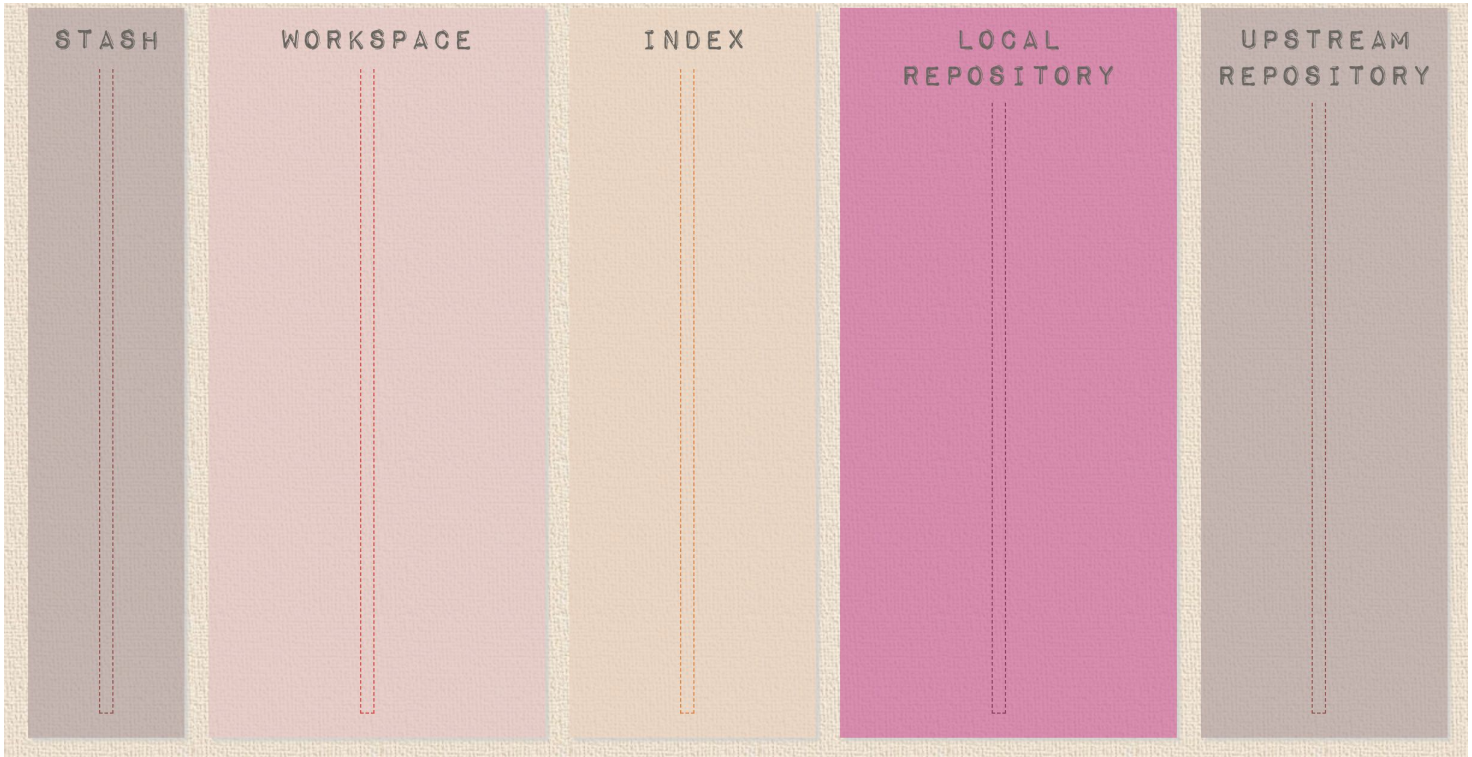
Big Picture II

Review: Revisiting Git through diagrams and images


Untracked
Unmodified
Modified
Staged
Add the file
Edit the file
Stage the file
Remove the file
Commit


Remote Git Repo
Remote Git Repo
Push
Fetch
Clone
Clone
Fetch
Push
Your Computer
Local Git Repo
Committed Files
Merge
Commit
Development Workspace
(files)
Add
Staged (Indexed) Files

https://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo



Long story short...

- Within your computer, as it pertains to Git, you have a: a stash, a workspace, an index, a local repository, and a means to interact with an upstream repository. During each area, stage, or phase, certain commands exist to: bring in changes, track changes, push changes, or rollback changes, etc.

10 Core and Additional Commands

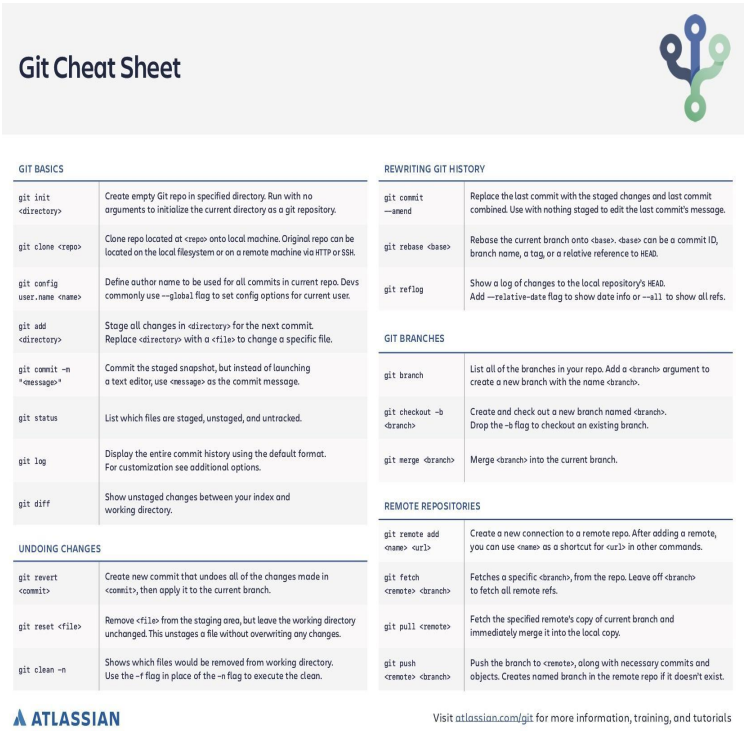
Some nice-to-know core commands and additional commands with Git

Core Commands and Additional Commands I

- Here are links (and a preview) that provide a downloadable file (.pdf) containing core commands and additional commands with git

- https://education.github.com/git-cheat-sheet-education.pdf

- https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet



Git Cheat Sheet

GIT BASICS

`git init <directory>`	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
`git clone <repo>`	Clone repo located at <repo> onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
`git config user.name <name>`	Define author name to be used for all commits in current repo. Devs commonly use `--global` flag to set config options for current user.
`git add <directory>`	Stage all changes in <directory> for the next commit. Replace <directory> with a <file> to change a specific file.
`git commit -m "<message>"`	Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.
`git status`	List which files are staged, unstaged, and untracked.
`git log`	Display the entire commit history using the default format. For customization see additional options.
`git diff`	Show unstaged changes between your index and working directory.

UNDOING CHANGES

`git revert <commit>`	Create new commit that undoes all of the changes made in <commit>, then apply it to the current branch.
`git reset <file>`	Remove <file> from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
`git clean -n`	Shows which files would be removed from working directory. Use the -f flag in place of the -n flag to execute the clean.

REWRITING GIT HISTORY

`git commit --amend`	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
`git rebase <base>`	Rebase the current branch onto <base>. <base> can be a commit ID, branch name, a tag, or a relative reference to HEAD.
`git reflog`	Show a log of changes to the local repository's HEAD. Add `--relative-date` flag to show date info or `--all` to show all refs.

GIT BRANCHES

`git branch`	List all of the branches in your repo. Add a <branch> argument to create a new branch with the name <branch>.
`git checkout -b <branch>`	Create and check out a new branch named <branch>. Drop the -b flag to checkout an existing branch.
`git merge <branch>`	Merge <branch> into the current branch.

REMOTE REPOSITORIES

`git remote add <name> <url>`	Create a new connection to a remote repo. After adding a remote, you can use <name> as a shortcut for <url> in other commands.
`git fetch <remote> <branch>`	Fetches a specific <branch>, from the repo. Leave off <branch> to fetch all remote refs.
`git pull <remote>`	Fetch the specified remote's copy of current branch and immediately merge it into the local copy.
`git push <remote> <branch>`	Push the branch to <remote>, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

ATLASSIAN

Visit atlassian.com/git for more information, training, and tutorials

Long story short...

- There are a lot of commands, but only a few will be used as part of your rotation throughout this course (and the others will be on a case-by-case basis as determined by your situation)!

11

Summary

A bird’s eye view of version control systems and git

Summary I

- In Git, there is the local repository, consisting of the stash, workspace, and index, and the upstream repository, typically hosted on GitHub for the purpose of this course.

- Git provides commands for transferring information (changes) between the local and upstream repositories, enabling seamless synchronization and collaboration.

Summary II

- Git is NOT Github
  - Git is a DVCS, and GitHub is a platform that hosts Git repositories
  - With the way Git is designed, if the GitHub server went down for a few hours, you would still have a full copy of the project on your local machine
    - It is important to distinguish these two as separate technologies --- thanks to Git, you can have a local repository and thanks to GitHub, you can have an upstream repository
  - Another way to remember this is: Microsoft took control of GitHub in an acquisition, and that has nothing at all to do with Git

Summary III

- Here are some (not all) questions and ideas we recommend that you ask yourself and step through while navigating Git:
  - Which part of the funnel am I in and working with? The stash, the workspace, the index, the local repository, the upstream repository? This will help with eligibility of commands and what to search on the internet for further assistance (a key part of learning and software development).
  - Am I able to use `command abc` at this moment in time given the commands I have previously ran AND the area I am in?
  - What is my current goal and what is the difference between `command x` and `command y` in general and how do they pertain to my goal?
  - Are there any prompts (suggestions) in the terminal output that can help guide me toward or away from certain commands? How can I leverage `git <command> --help` to clarify some of the documentation and features?

Conclusion