GIT & GITHUB

Source Control Management

Prashanth Puranik
Web Developer and Trainer

Version Control System (VCS)

- Version Control System (VCS) – A Source Control Management (SCM) tool
- Maintain versions of your project code and collaborate on code
  - Create snapshots (**commits**), and **branches** of your codebase
- Track and navigate history
  - You can revert the codebase to a previous commit
- VCSes can be
  - ***Centralized*** *– The codebase versions are maintained in a central server*
  - ***Distributed*** *– The codebase versions are maintained in each node (locally). To share code across the entire team, there usually is a designated "remote" (central) repository.*

About Git

Git

- A **distributed VCS** created by Linux Torvalds in 2005 with following goals

- Used by many organizations (including major ones) and most open-source projects

- **References**
  - Linux Torvalds' talk on Git
  - Introduction to Git by Scott Chacon

Alternatives

- **Centralized**
  - Apache Subversion
  - Perforce HelixCore
  - IBM Rational ClearCase
  - CVS, RCS etc.

- **Distributed**
  - Mercurial
  - Microsoft TFS (more features than just version control)

About Git

Advantages

- Stores all the history, branches, and commits locally
  - Operations are thus faster – example, branching is fast, and allows to sandbox your features till they are ready for the mainstream. In centralized VCSes, branches are created centrally as sub-directories – this makes branching a costly operation.
  - No network connection needed
  - Team members not constrained by work done by others – create branches locally and continue your work, merging it with work of others only when satisfied
- Supports multiple workflows (including Subversion type workflow)
- Very hard to corrupt data
- Selectively stage files and commit for fine-grained control

About Git

Disadvantages

- Steep learning curve when compared to other SCM tools
  - Need to know internals to some extent to work with it
- Does not handle large files and frequently updated binary files, as well as it does text files

Setup & Configuration

Download and Install

- Download the CLI from https://git-scm.com/download/ and install
- Many GUIs are also available - https://git-scm.com/downloads/guis
  - The GUI is an optional download
- IDEs have good integration with Git
  - Usually built-in and hence no plugin/extension required
  - GitHub Desktop is a popular choice
  - We shall use VSCode in the training

Configuration

- Once installed, open Git Bash on Windows, or Terminal in Mac/Linux and set these global configs

```
$> git config --global user.name "Your Name"
$> git config --global user.email youremail@yourorganization.com
```

- View a global config setting this way

```
$> git config --global user.email
```

Configuring the Editor for Git

Git opens an editor to enter commit message (if omitted in the command line) etc. To set VS Code as the editor…

1. Add VS Code to your path

   For Mac OS X, add /Applications/Visual Studio Code.app/Contents/Resources/app/bin to /etc/paths

2. Set EDITOR / GIT_EDITOR environment variable to code

   For Mac OS X,

   ```
   $> EDITOR=code
   $> export EDITOR
   ```

3. `git config --global core.editor "code -w"`

Basic Concepts & Commands

Working with Git Locally

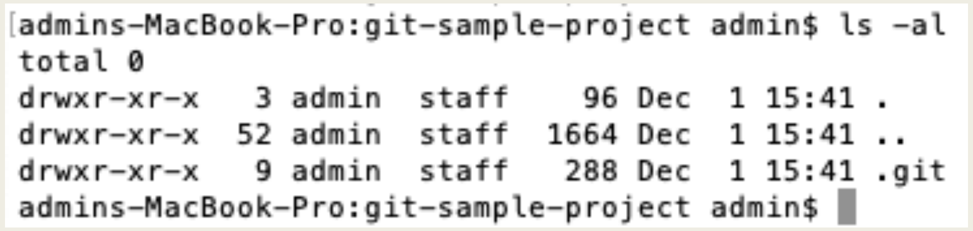
Git Repository (repo)

- A Local collection of all the files related to a project
- Contains a **.git subdirectory** in the project root
- Git tracks state of the files under this folder
- The state of the repo (the files and folders in current state) is referred to as the **working tree**

Creating a Repository

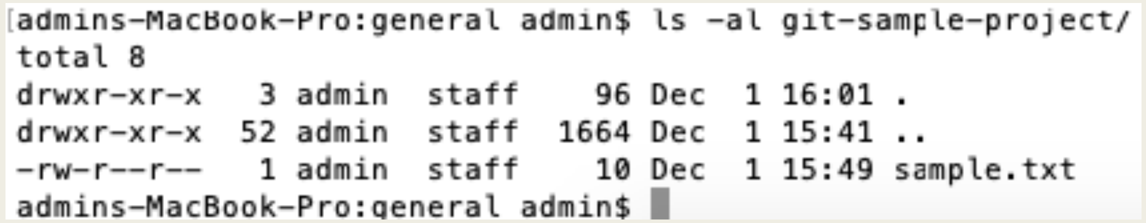
Creating a repo when folder is created

```
$> mkdir git-sample-project

$> cd git-sample-project

$> git init
```



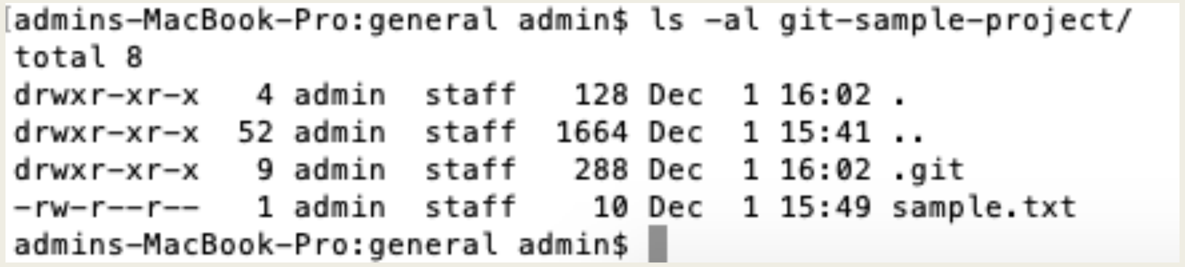
```
admins-MacBook-Pro:git-sample-project admin$ ls -al
total 0
drwxr-xr-x   3 admin  staff    96 Dec  1 15:41 .
drwxr-xr-x  52 admin  staff  1664 Dec  1 15:41 ..
drwxr-xr-x   9 admin  staff   288 Dec  1 15:41 .git
admins-MacBook-Pro:git-sample-project admin$
```

Creating a repo out of an existing folder



```
admins-MacBook-Pro:general admin$ ls -al git-sample-project/
total 8
drwxr-xr-x   3 admin  staff    96 Dec  1 16:01 .
drwxr-xr-x  52 admin  staff  1664 Dec  1 15:41 ..
-rw-r--r--   1 admin  staff    10 Dec  1 15:49 sample.txt
admins-MacBook-Pro:general admin$
```

```
$> git init git-sample-project
```



```
admins-MacBook-Pro:general admin$ ls -al git-sample-project/
total 8
drwxr-xr-x   4 admin  staff   128 Dec  1 16:02 .
drwxr-xr-x  52 admin  staff  1664 Dec  1 15:41 ..
drwxr-xr-x   9 admin  staff   288 Dec  1 16:02 .git
-rw-r--r--   1 admin  staff    10 Dec  1 15:49 sample.txt
admins-MacBook-Pro:general admin$
```

Getting help on a command

Syntax

```
$> git <command> --help
```

Help on initializing a Git repo

```
$> git init --help
```

Listing .git folder in VSCode

- The .git folder is hidden by default in VSCode file explorer
- Make the following changes to .vscode/settings.json (Workspace settings)

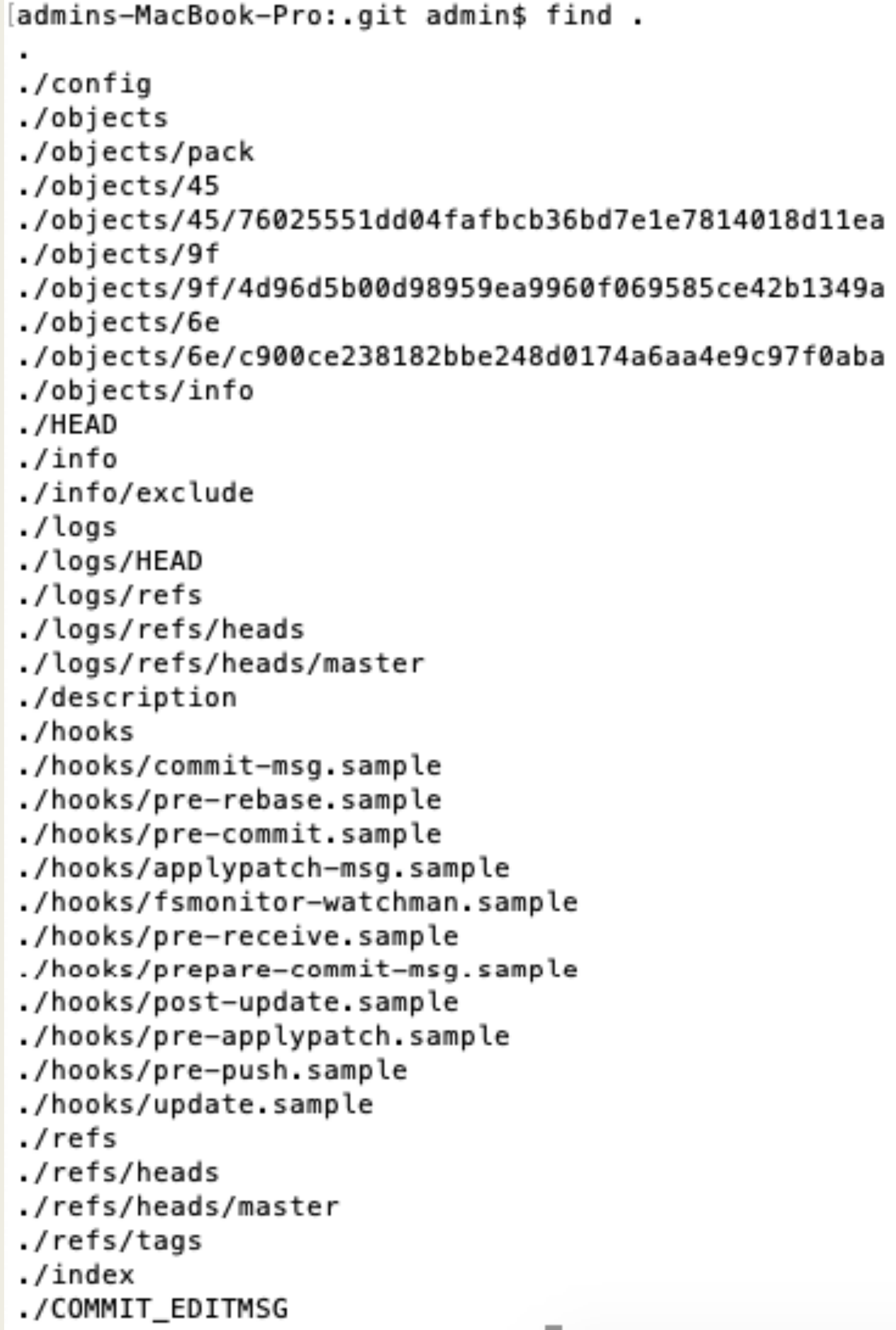
```
{
	"files.exclude": {
		"**/.git": false
	}
}
```

The .git folder

- **config** – repo configuration
- **description** – file that describes the repo
- **HEAD** – has HEAD pointer
- **hooks/** - event hooks
- **info/exclude** – excluded files
- **objects/info/** – object information
- **objects/pack/** – pack files
- **refs/heads/** – branch pointers
- **refs/tags/** – tag pointers

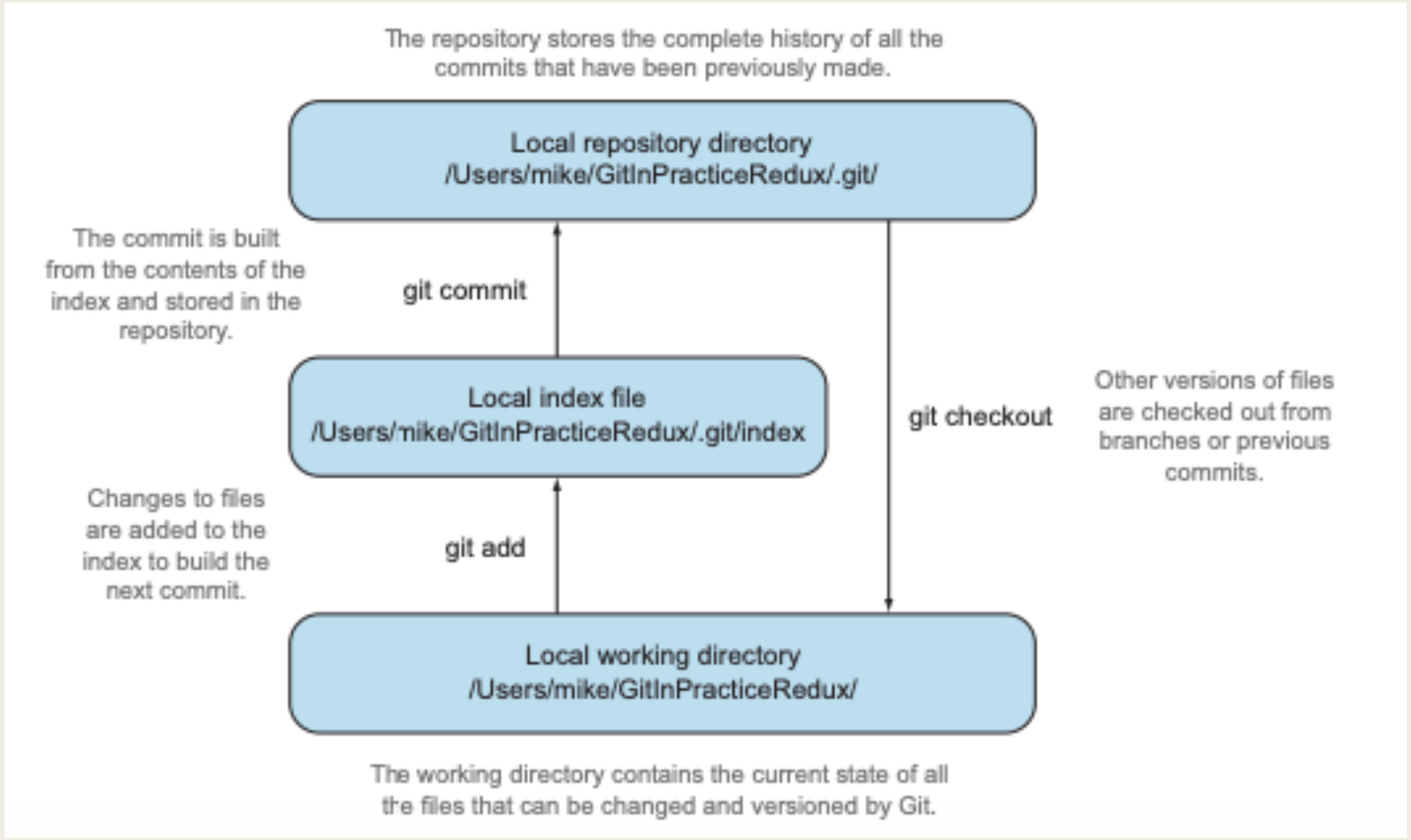
Note: Never modify these files yourself

Reference: https://www.linkedin.com/pulse/git-internals-how-works-kaushik-rangadurai/



```
admins-MacBook-Pro:.git admin$ find .
.
./config
./objects
./objects/pack
./objects/45
./objects/45/76025551dd04fafbcb36bd7e1e7814018d11ea
./objects/9f
./objects/9f/4d96d5b00d98959ea9960f069585ce42b1349a
./objects/6e
./objects/6e/c900ce238182bbe248d0174a6aa4e9c97f0aba
./objects/info
./HEAD
./info
./info/exclude
./logs
./logs/HEAD
./logs/refs
./logs/refs/heads
./logs/refs/heads/master
./description
./hooks
./hooks/commit-msg.sample
./hooks/pre-rebase.sample
./hooks/pre-commit.sample
./hooks/applypatch-msg.sample
./hooks/fsmonitor-watchman.sample
./hooks/pre-receive.sample
./hooks/prepare-commit-msg.sample
./hooks/post-update.sample
./hooks/pre-applypatch.sample
./hooks/pre-push.sample
./hooks/update.sample
./refs
./refs/heads
./refs/heads/master
./refs/tags
./index
./COMMIT_EDITMSG
```

The add/commit/checkout workflow


The repository stores the complete history of all the commits that have been previously made.
Local repository directory /Users/mike/GitInPracticeRedux/.git/
The commit is built from the contents of the index and stored in the repository.
git commit
Local index file /Users/mike/GitInPracticeRedux/.git/index
git checkout
Other versions of files are checked out from branches or previous commits.
Changes to files are added to the index to build the next commit.
git add
Local working directory /Users/mike/GitInPracticeRedux/
The working directory contains the current state of all the files that can be changed and versioned by Git.

Adding to staging area

- Staging area allows you to selectively add files/folders, in order to commit them later

```
$> git add file_or_folder
```

- Staged files are tracked and indexed
- Staged files are re-indexed by calling `git add` on them again
- The index is what gets committed. So it is important to call `git add` each time you make changes (if you need changes to be committed)

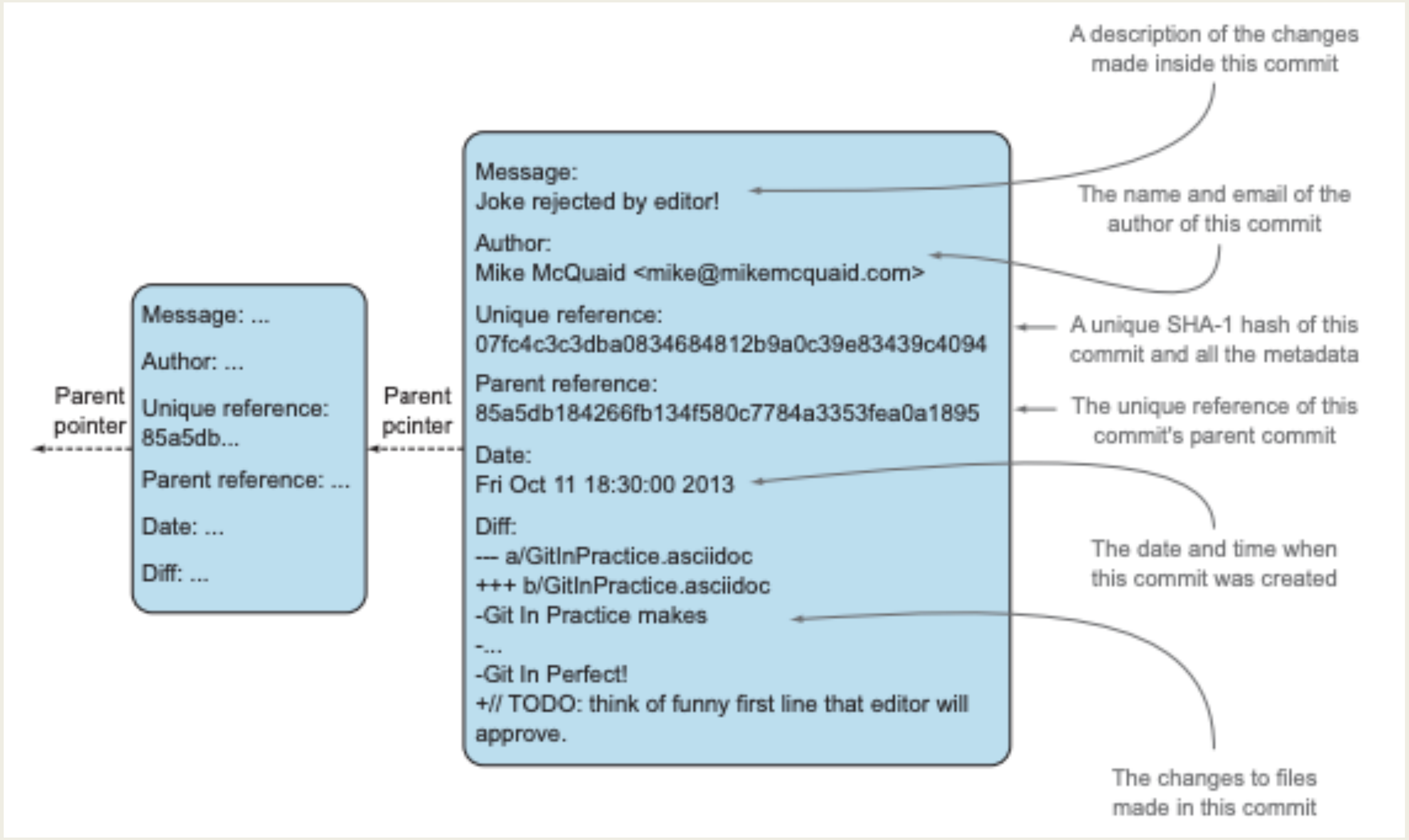
View Status of Repo

- Status of a repo includes
  - *Current branch*
  - *History with respect to upstream branch*
  - *List of untracked, tracked files*
  - *Modifications made to tracked files*
  - *Indexed files*

- To view status type

```
$> git status
```

Committing


A description of the changes made inside this commit
The name and email of the author of this commit
A unique SHA-1 hash of this commit and all the metadata
The unique reference of this commit's parent commit
The date and time when this commit was created
The changes to files made in this commit
Parent pointer
Parent pcinter
Message: ...
Author: ...
Unique reference: 85a5db...
Parent reference: ...
Date: ...
Diff: ...

Message:
Joke rejected by editor!

Author:
Mike McQuaid <mike@mikemcquaid.com>

Unique reference:
07fc4c3c3dba0834684812b9a0c39e83439c4094

Parent reference:
85a5db184266fb134f580c7784a3353fea0a1895

Date:
Fri Oct 11 18:30:00 2013

Diff:

```
--- a/GitInPractice.asciidoc
+++ b/GitInPractice.asciidoc
-Git In Practice makes
-...
-Git In Perfect!
+// TODO: think of funny first line that editor will
approve.
```

Committing

```
$> git commit --message "commit message"
$> git commit -m "commit message"
```

Add known files (those listed in the index) and commit

```
$> git commit –a -m "commit message"
Or
$> git commit –am "commit message”
```

Note: If commit message is omitted, Git does not commit the changes.

Committing - Best Practices

- Keep commits byte-sized
  - *Every small logical piece should have its own commit*
  - *Other logic should not be part of commit*
  - *Keeps history readable*
  - *Easy to pick only undesirable changes and revert changes if required*

- Maintain well-formatted commit messages
  - *In present tense*
  - *Describe changes well*
  - *Short first line for subject (<= 50 characters recommended)*
  - *Separate rest of lines from subject by an empty new line*
  - *Rest of lines describe in more detail*

Why staging (`git add`) exists?

- You can selectively choose files and “hunks” to be added
  - *Pick and choose from individual files and pieces of changes made to tracked files*
  - *Helps make smaller commits*
  - *Indexing happens as and when you add, and can be done multiple times before a commit - may save some time when you have changes to a lot of files.*
- It allows you to keep only desired changes and blow away changes that are not required (by committing desired changes and then doing a hard reset)

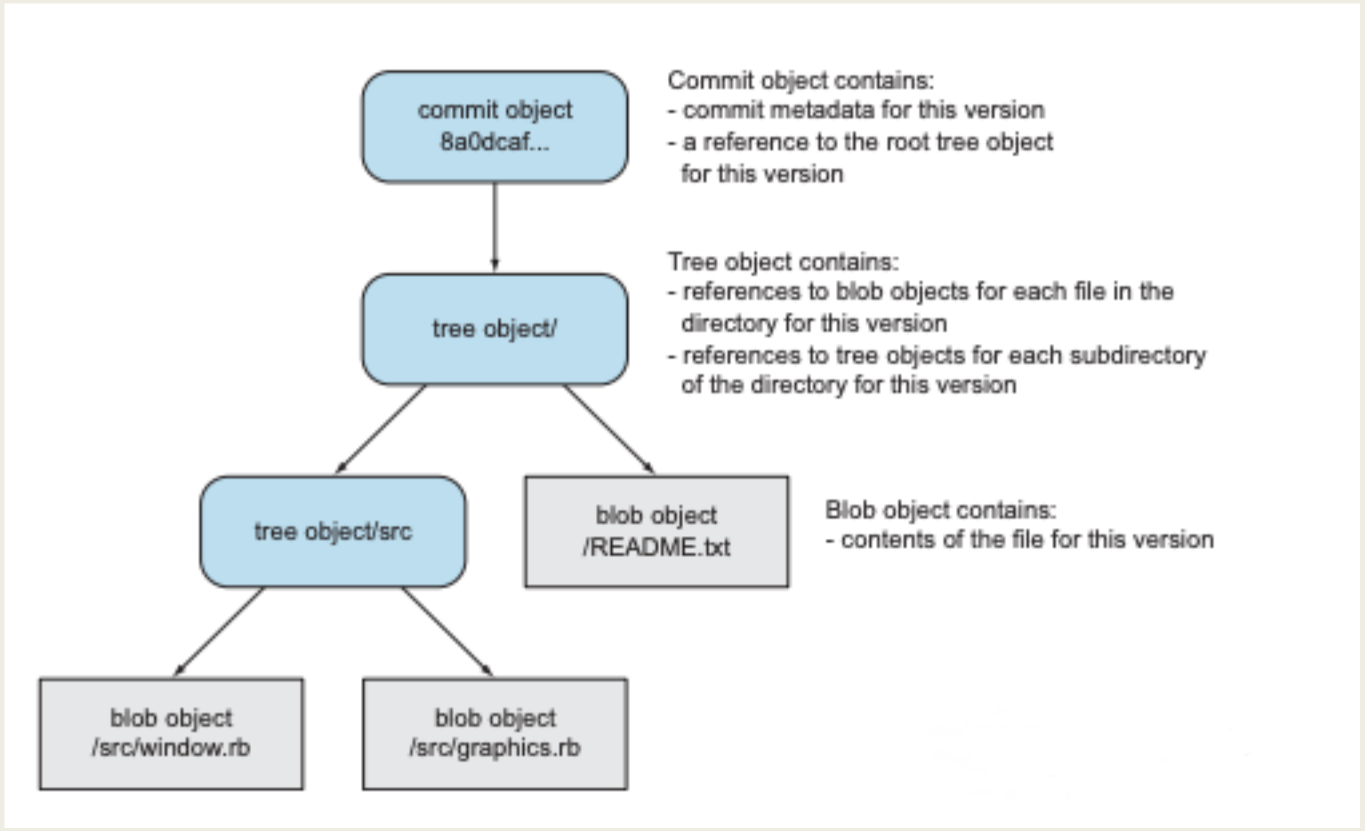
Reference: https://stackoverflow.com/questions/4878358/why-would-i-want-stage-before-committing-in-git

SHA-1 Hashes and Git Objects

- Git creates SHA-1 hashes for every
  - *Tracked file (contents)*
  - *Tracked folder (tree structure)*
  - *Commit*

- The SHA-1 hashes acts like keys that index into various objects
  - *file contents (blob)*
  - *folder structure (tree object)*
  - *commit details (commit object)*

- Commit objects include
  - *author details*
  - *date of commit*
  - *description*
  - *parent commit pointer (SHA-1 of previous commit in branch) etc.*


commit object
8a0dcaf...
Commit object contains:
- commit metadata for this version
- a reference to the root tree object for this version
tree object/
Tree object contains:
- references to blob objects for each file in the directory for this version
- references to tree objects for each subdirectory of the directory for this version
tree object/src
blob object
/README.txt
Blob object contains:
- contents of the file for this version
blob object
/src/window.rb
blob object
/src/graphics.rb

Refs

- Refs are pointers to various objects. Some popular ones used in Git commands are
  - *HEAD*: A pointer to the latest commit on the current branch
  - *branch_name*: A pointer to the latest commit on the branch with given name
    - **Example**: ***master*** is a pointer to the latest commit on master branch
  - ***HEAD~1, HEAD~2, master~1*** etc.: n commits before latest one on HEAD, master etc.
  - **origin/master, origin/test, origin/test~2**: Similar pointers but on remote branches (remote named **origin**)

Viewing history

- Git maintains commit history, information on branches, tags and merges.
- To view history type

```
$> git log
```

- Visual tools like gitk (and other platform-specific ones like gitx) help view history graphically

```
$> gitk
```

Viewing diffs

- A diff (aka change / delta) is the difference between 2 commits

You can diff commits, branches and tags

- Difference between previous commit and latest one on current branch

```
$> git diff HEAD~1 HEAD
```

- Difference between last 2 commits on master branch

```
$> git diff master~1 master
```

- Difference between the current working directory and the index staging area.

```
$> git diff
```

- Difference between the current working directory and the last commit on master branch

```
$> git diff master
```

Viewing diffs

- Difference between any 2 commits

```
$> git diff commit-x-sha1 commit-y-sha1
```

- Summary of diffs rather than actual changes

```
$> git diff --stat commit-x-sha1 commit-y-sha1
```

- Difference between a given file on 2 commits

```
$> git diff commit-x-sha1 commit-y-sha1 -- sample.txt
```

Note: The diffs are easily views using gitk/gitx

Exercise: Find out the option that given word-by-word difference instead of line-by-line

Squashing commits

- Multiple related consecutive commits can be squashed into a single commit
- **Use case:** You may add a new feature in one commit, and fix bugs in some following commits – Squash these to a single one
- This is a best practice
- `$>`

Remote Repository & Git

Git-based source code hosting sites

- Many services let you host your Git managed codebase in the cloud. Some are
  - GitHub
  - GitLab
  - BitBucket
- GitHub is very popular among these
- We shall use GitHub repos as a remotes for our local Git repos
- You can have **public** (free) as well as **private** (paid) repos in GitHub.

- The central GitHub repo usually hosts the shared codebase for the team
  - Every team member pull changes from it, and updates changes to it
  - This codebase is the one that is deployed on staging/production servers etc.

The push–pull/fetch workflow

Send new commits made by you from local -> remote repo

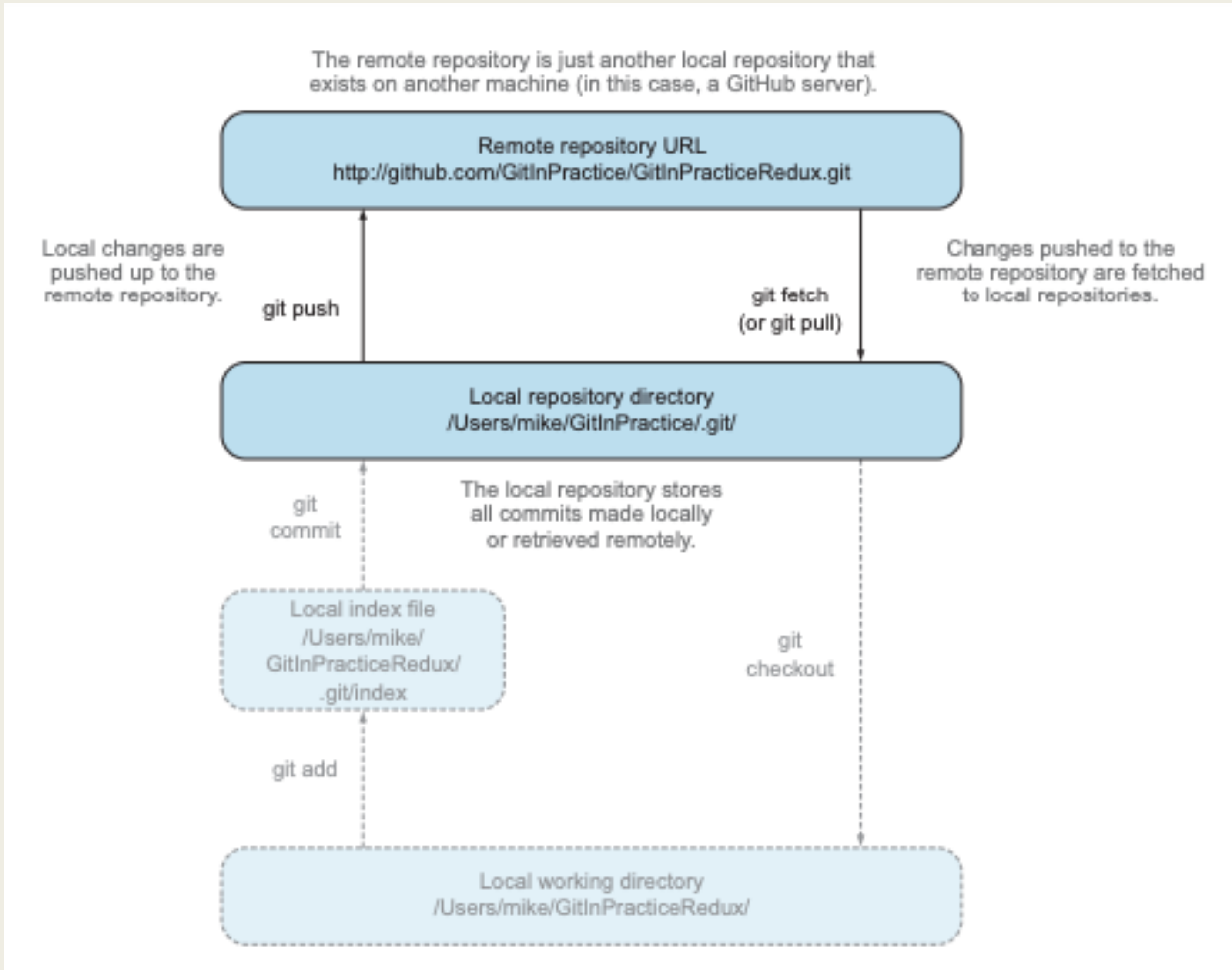
```
$> git push
```

Retrieve new commits made by others from remote -> local repo

```
$> git fetch
```

or

```
$> git pull
```


The remote repository is just another local repository that exists on another machine (in this case, a GitHub server).
Remote repository URL
http://github.com/GitInPractice/GitInPracticeRedux.git
Local changes are pushed up to the remote repository.
git push
git fetch
(or git pull)
Changes pushed to the remote repository are fetched to local repositories.
Local repository directory
/Users/mike/GitInPractice/.git/
git commit
The local repository stores all commits made locally or retrieved remotely.
Local index file
/Users/mike/GitInPracticeRedux/.git/index
git checkout
git add
Local working directory
/Users/mike/GitInPracticeRedux/

Creating a remote and linking to it

- You can add multiple remote repos, linked to a single local repo
- Usually however, only one remote exists, and called **origin** by default

- Login to your GitHub account and create a repo there. Note its URL.
  https://github.com/<username>/git-sample-project

- Link your local Git project to this remote repo

```
$> git remote add origin https://github.com/<username>/git-sample-project
```

- Verify all remotes for a local repo this way

```
$> git remote –verbose
```

Note: Normally the remote repo to push to and fetch from are the same

Pushing changes to remote

- Git can make network operations using **https**, **ssh** and its custom protocol **git**
- Git sends only diffs over the network

- To push changes from current local branch to the master branch of remote (origin)

```
$> git push origin master
```

Notes:

1. A local branch may have its commits pushed to a remote branch with different name, but due to the confusion this causes, it is never done.
2. A username and password may be asked for. This may be for first push only, or even subsequent pushes depending on your OS. For fetch/pull, the credentials are required only for private repos.

Pushing changes to remote

- The **`--set-upstream`** option (**alias `-u`**) sets the local master to track origin/master. Tracking branches are local branches that have a direct relationship to a remote branch.

```
$> git push --set-upstream  origin master
```

- This setting means further push requires no arguments. If you are on local master, the following pushes further changes to origin/master henceforth.

```
$> git push
```

- To push all branches to remote (not recommended as there may be work-in-progress branches)

```
$> git push --all
```

Cloning a remote to local

- **Cloning:** The process of creating a new local repo from an existing remote repo
- Repos may be cloned from GitHub etc. (although local -> local clones are also allowed)
- Cloning downloads all commits, branches, and tags – in short, the entire history to local
- It also sets up remote automatically (say, to the GitHub repo)

```
$> git clone https://github.com/<username>/<reponame>.git
```

- By default a folder with same name as remote repo is created. You can change the folder name for local this way

```
$> git clone https://github.com/<username>/<reponame>.git <LocalRepoName>
```

- On local, only master is created by default. When you checkout another branch locally, it is created on local and tracks the corresponding remote branch.

```
$> git branch <branch-name> origin/<branch-name>
```

Exercise: Assuming you have pushed the entire local repo to GitHub (using `git push --all`), delete the local and recreate from the remote repo on GitHub.

Pulling changes from remote

- Create another clone of the GitHub repo, make a new commit, and git push it.
- Now switch to previous clone and pull in changes using

```
$> git pull
```

- All objects and refs are fetched. However, only the current branch is updated. The other branches need to be checked out and changes pulled in similarly.
- The above is a **fast-forward merge**, since the branch in remote was simply ahead of local. There is **no merge commit created** in this case.
- In cases where the local and remote copy have diverged, a pull would result in a merge commit.

git pull = git fetch + git merge

- To fetch objects and refs from remote without updating local branches,

```
$> git fetch
```

Exercise: Repeat the previous exercise but this time use fetch, instead of pull. Notice that changes are fetched but not merged.

- Now merge changes manually into the local copy of master using

```
$> git merge origin/master
```

- These 2 steps are essentially what happen when you `git pull`

- Prefer a fetch over a pull
  - Fetch objects without worrying about merge conflicts. Resolve later even without internet connection.
  - Other preferable ways of combining changes (eg. rebase) may be used instead of merge

Need for Branches

- Linear approach to committing is not sufficient
  - You may need to make new commits that are not ready for public consumption
  - A new feature that is being developed parallelly with other work
  - You want to have commits independent of work done by team on some branch

- Branching allows multiple independent tracks to be created and committed
  - Do the feature development on a separate branch and then merge to master
  - Make your own commits on another branch, branched off from feature branch!

Creating a branch

- Create a branch (say, bug-fix) out of the current branch

```
$> git branch bug-fix
```

- Create a branch out of a given branch (say, test)

```
$> git branch bug-fix test
```

- Create a branch out of a given commit(say, 09301d)

```
$> git branch bug-fix 09301d
```

- To start tracking a remote branch when a local one is created

```
$> git branch <new_local_branch> ---track origin/<remote_branch>
```

Notes:

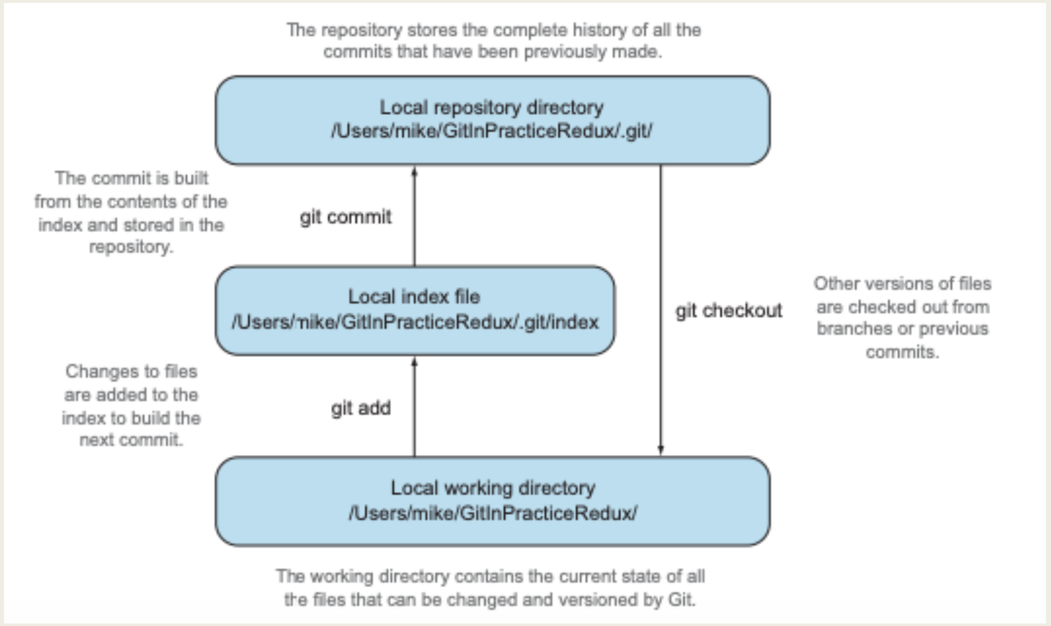
1. After creating a new branch, you still remain on the old one
2. Give meaningful branch names (hyphens are allowed)

Checking out a branch

- Checking out a branch sets the working directory state to that of the branch

```
$> git checkout <branch_name>
```

- You need to **commit current branch** (it is okay to have untracked files though), or **stash changes** before switching branch.
- You can use the `--force` flag instead but this is not recommended


The repository stores the complete history of all the commits that have been previously made.
Local repository directory /Users/mike/GitInPracticeRedux/.git/
The commit is built from the contents of the index and stored in the repository.
git commit
Local index file /Users/mike/GitInPracticeRedux/.git/index
git checkout
Other versions of files are checked out from branches or previous commits.
Changes to files are added to the index to build the next commit.
git add
Local working directory /Users/mike/GitInPracticeRedux/
The working directory contains the current state of all the files that can be changed and versioned by Git.

Pushing a local branch to remote

- To push a local branch (say bug-fix) to remote, first checkout the local. Then,

```
$> git push --set-upstream origin bug-fix
```

- This creates a bug-fix branch on origin (if it does not exist), and pushes changes
- It also sets local bug-fix branch to track remote/bug-fix

Merging a branch into another

- You can “merge” contents of another branch into the current one.
- This is done to integrate, say
  - a feature branch with master
  - Your local branch (say bug-fix) into the local branch tracking a remote one, etc.
- A “merge commit” is created on current branch (unless the branches have not diverged)
- The merge commit has 2 parent commits (instead of the regular one)
- The feature branch may be removed or preserved (as per your requirements)

Merging a branch into another

- To merge contents of another branch (say bug-fix) into the current one (say master)

```
$> git merge bug-fix
```

- If the current branch commits are all on the one being merged (say, it was created from current branch, and the current branch had no new commits after the branching), then no merge commit is created
- Usually however, a merge commit with 2 parent commits (from each branch) is created
- Git tries its best to resolve conflicts automatically, however, merge conflicts may crop up (eg. a piece of code in the same file has been modified by both branches)
- In case of conflicts, “resolve” these conflicts manually, and create the the commit (a merge commit).

References

Official documentation and downloads

https://git-scm.com/

A popular and free book on Git – Pro Git 2nd Edition, by Scott Chacon and Ben Straub, published by Apress

https://git-scm.com/book/en/v2

Thank you