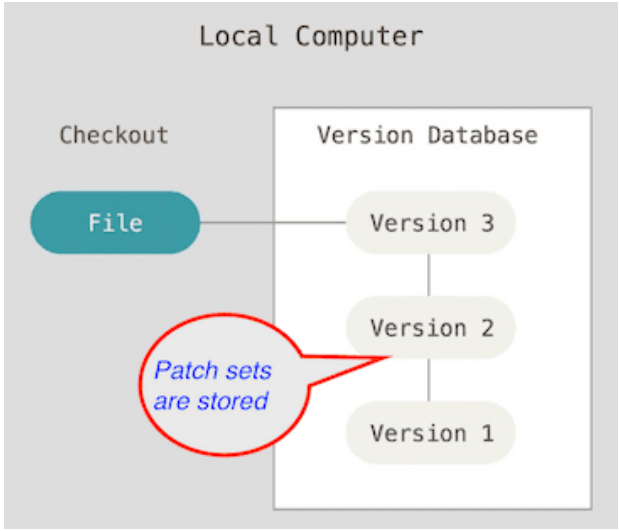
Git Internals

Understanding *How Git Works* is Essential for Every Software Engineer

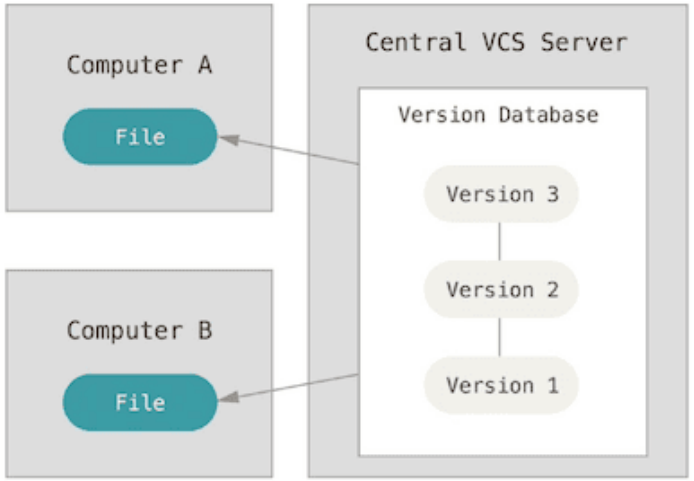
VCS

- To track the changes, many tools have been invented. We call them Version Control Systems (VCS).
- The most famous one is the RCS command in the UNIX OS.
- RCS works by keeping patch sets (differences between files) in a special format.
- It can recreate any file by



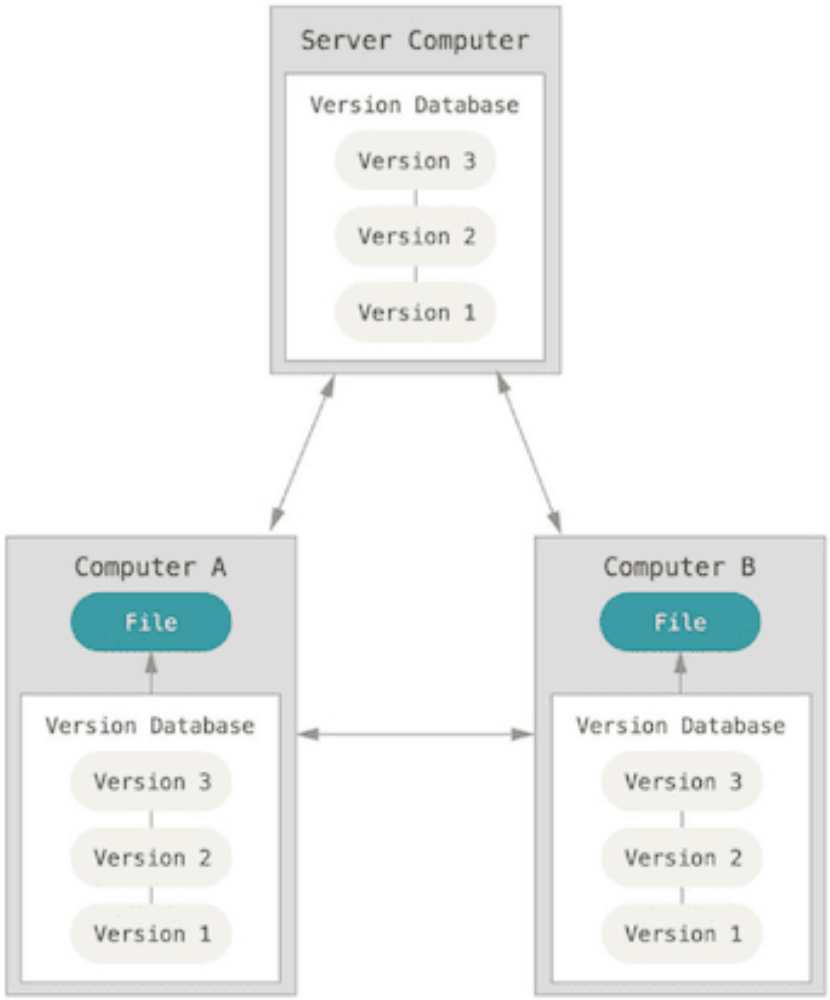
CVCS

- With the invention of the Internet, we can have a server that manages the VCS system.
- Any clients (computers) can access the central CVS server to create, read, update, and delete versions.
- However, the CVSC system cannot be used when the computers cannot access the network.



DVCS

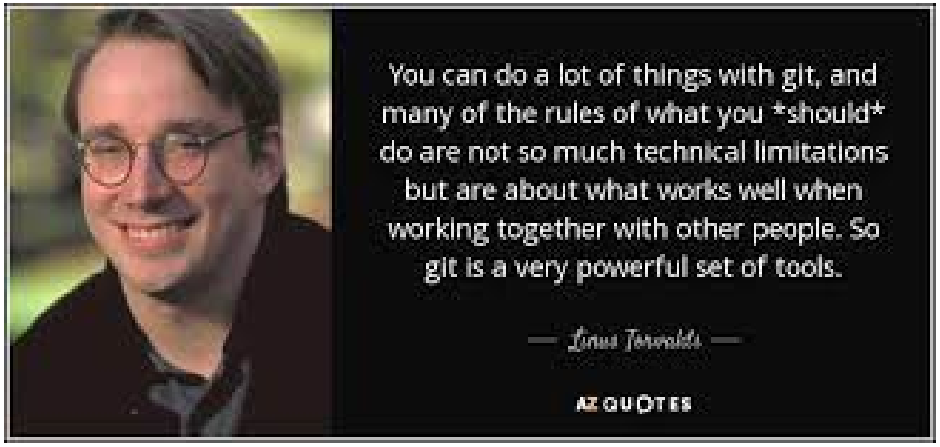
- So, Distributed Version Control System (DVCS) was invented to address this issue.
- In a DVCS, clients don't check out the latest snapshot (version) only to the server, but to a local computer, or to even any computer nearby



DVCS Systems

- There are many DVCS available.
- Darc, Bazaar, and Mercurial are some of the most popular DVCS systems.
- In general, Git is the most widely used DVCS with many web-based Git servers available, including Github, Gitlab, or Bitbucket.

Why Git was Invented?



- Git was invented by Linus Torvalds (the inventor of Linux) to manage the Linux source code.

The Requirements of Git

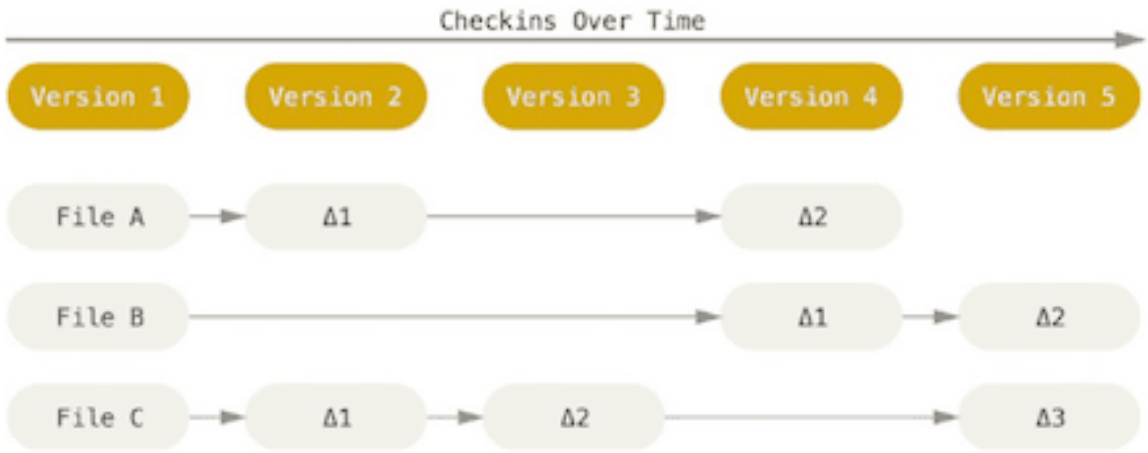
- **DVCS:** Fully distributed version control system
- **Scalable:** Used by millions worldwide
- **Fast:** Handles many files quickly
- **Efficient:** Manages large file sizes effectively

Git Structure

1. Snapshots, Not Patches
2. Blobs and Trees
3. Commit, Tag, and Branch
4. Hash
5. Local Repository in .git
6. The .gitignore File
7. The git status and git log
8. Diff, patch to understand CherryPick and Rebase
9. Merge

1. Snapshots, Not Patches

- All Version Control System systems stores only patches (changes) when uploading versions to the repo (history).
- This makes the repo small, but it takes time and complex to make any version of the file from applying patches from an original file.



Git's Snapshot Approach

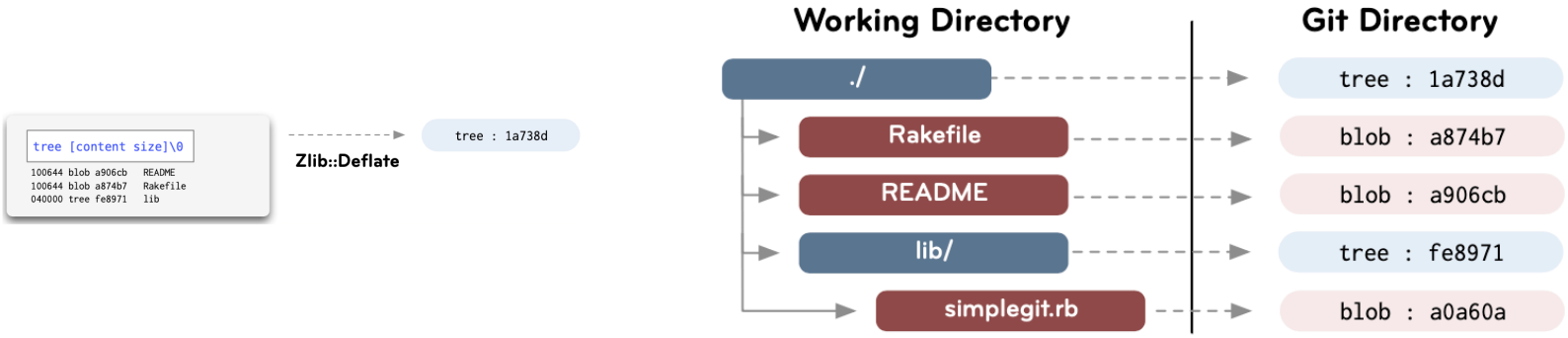
- Git does not store patches (differences), instead it stores the whole files in a snapshot.
- Git stores only the modified files in a snapshot.


center

2. Blobs and Trees

1. When we add or commit files, Git does not store files as they are; Instead, Git reads the content of a file and zips it and generate a hash.
2. Git does not store the directory; Instead, Git uses a tree to store the directory information.

Blobs and Trees Structure



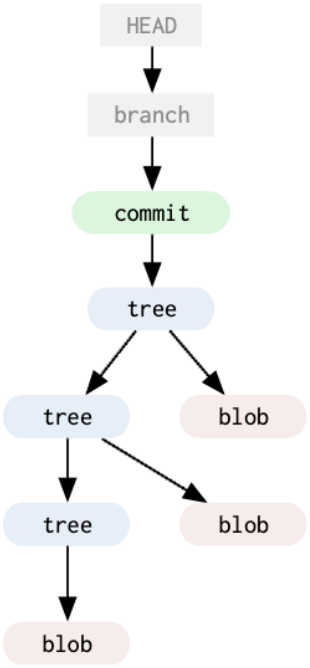
- The left-side shows how Git manages a root directory; it has blobs for files and trees for a directory.
- The right-side shows the tree information. The directory has two files (blobs) and one directory (tree) that has one file (blob).

3. Commit, Tag, and Branch

- Git manages the commit, branch, and tag using the same hash that has the tree information.
- They are all regarded as a tree.

Git Object Model

- Any commit, tag, branch, remote, or HEAD is just a reference of a tree.

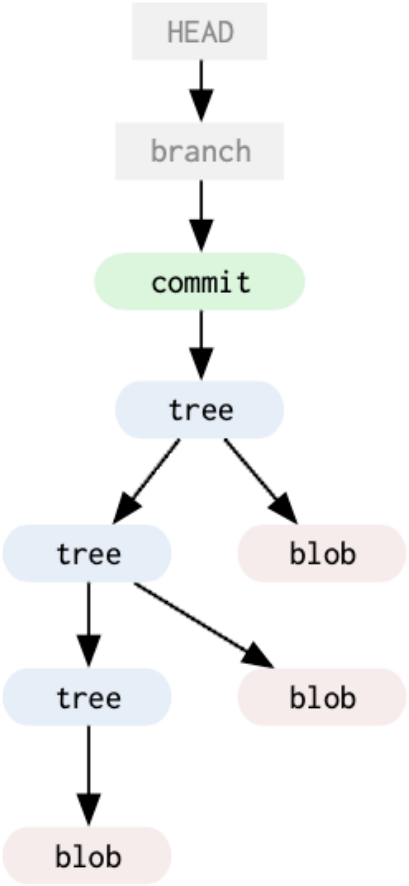


Hash-Based System

```
|-- F1
`-- lib
      |-- base
      |   `-- F3
      `-- F2
```

- We have a directory with one file (F1) and a directory (lib).
- The lib directory has a sub-directory (base) and a file (F2).
- the base directory has a file (F3).

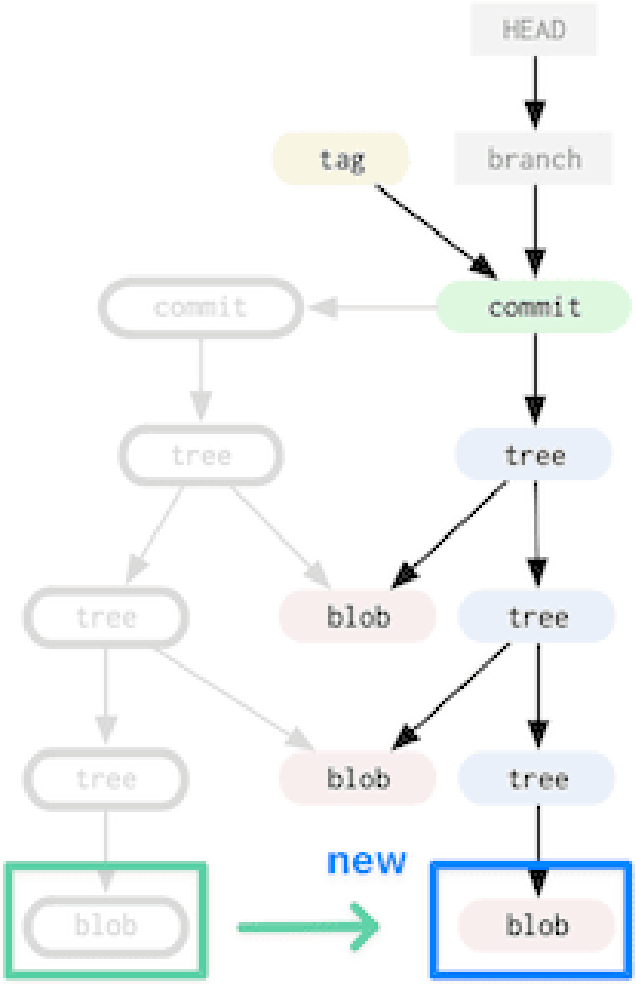
- This is generated tree with a commit.
- Any branch or HEAD is a reference to the commit.



```
|-- F1
`-- lib*
      |-- base*
      |   `-- F3*
      `-- F2
```

- When we modify F3 (*), Git traces the impact of the changes.
- F1 and F2 are not changed, but base and lib trees should be updated (*).

- Adding a tag is making a reference to point to the commit.

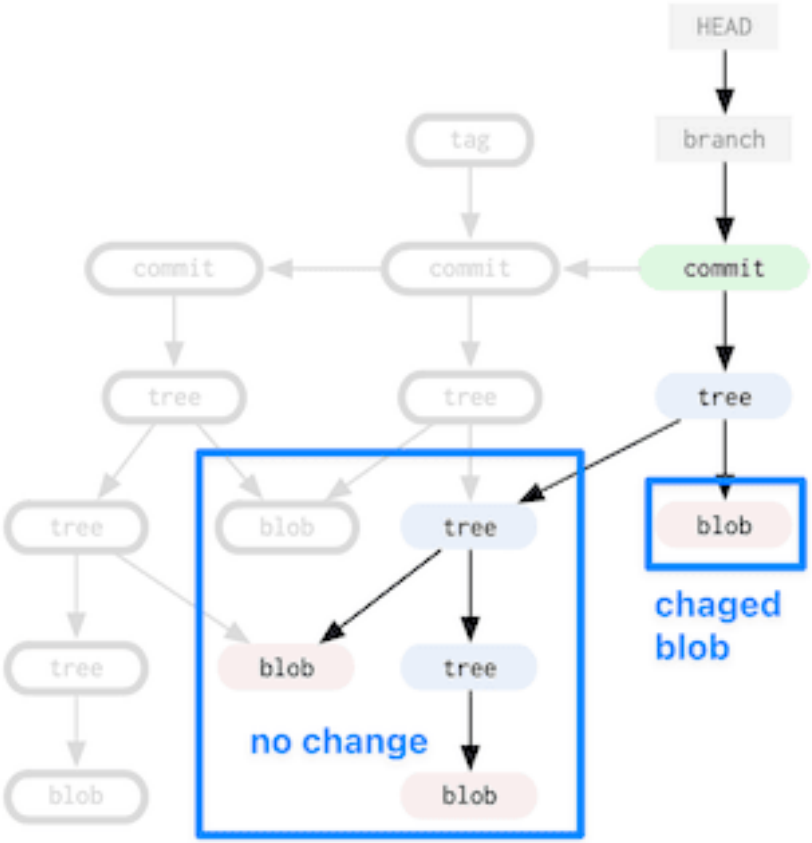


Tree Reuse

```
|-- F1**
`-- lib*
      |-- base*
      |`-- F3*
      `-- F2
```

- When F1(**) is modified, the whole lib directory (tree) is not upated, and can be reused.

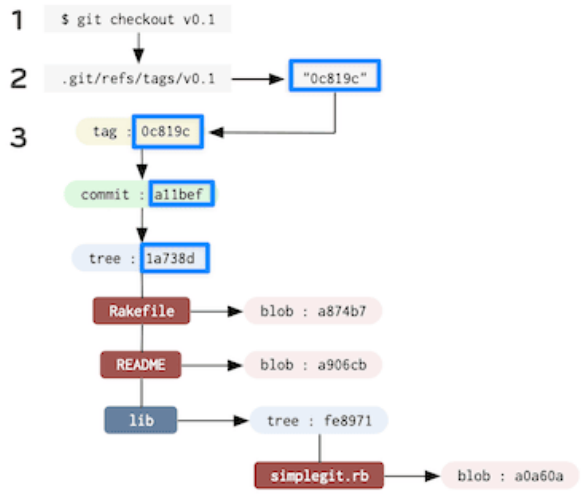
- Git keep tracks of the changes to manage commits effectively.



3. Hashes

- Git can detect changes efficiently because it **uses hashes** to identify and track objects.
- A **hash** is the result of passing data through a **one-way formula**; only the **same data** produces the same hash.
- If **anything changes**, a **new hash** is generated, and this means a change.
- In Git, **everything is identified by its hash** — this enables extremely fast operations to detect a change.

Checkout with Hashes



When checking out tag `v0.1`, Git:

1. Git looks up the tag name v0.1 in its reference database.
2. Git finds the hash for the tag.
3. Git follows the chain of hashe

Fast Operations with Hash

In Git, everything is identified by its hash.

> So when Git “finds a commit,” it’s really just resolving a hash — following a chain of references that ultimately lead to the commit object.

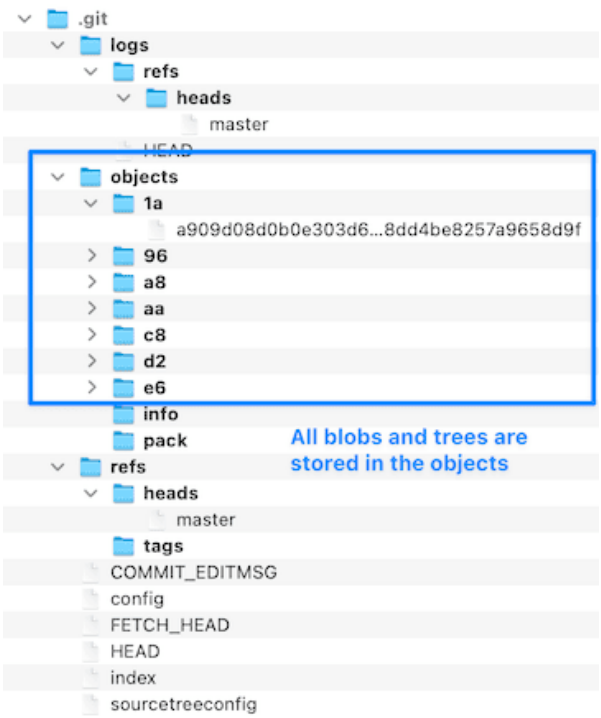
5. Local Repository in .git

To start tracking a project, run `git init` to create a `.git` directory.

- Use `git status` (or `git status -s` for short) to check the repository status.

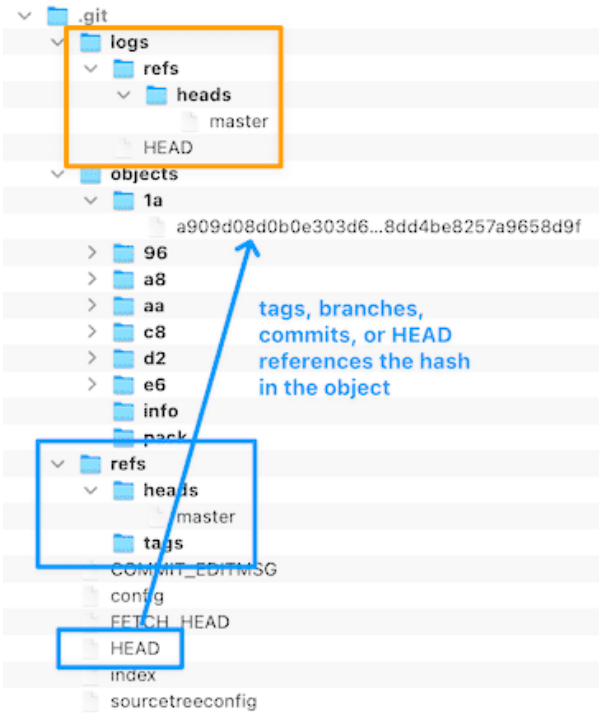
```
git init
git status
git status -s
```

The .git Directory Structure



- The git commands manage objects in the .git directory.
- The `git status` command reads information in the .git directory.
- Git objects (blobs and trees) are stored in the object directory using the hash values.

The .git Directory Content



- The log directory contains all the commit information; We can access this directory using `git log` command.
- The refs directory contains all the references including HEAD, branches, and tags.

The .gitignore File

The `.gitignore` file tells Git which files or folders to skip — "don't add these to staging."

- See git-scm.com/docs/gitignore for syntax and examples.
- Add .gitignore before adding files to Git
- Check what's ignored: `git status --ignored`

Project-Level `.gitignore`

- Put it in the **root directory** of your Git project.
- Applies to all files and folders in that repository.
- Example:

```
myproject/
├── .git/
├── .gitignore   ← here
├── src/
└── README.md
```

Subdirectory `.gitignore`

- You can also place `.gitignore` in **any subfolder**.
- It only affects files **inside that folder**.

```
src/
├── .gitignore   ← ignores only files in src/
└── main.c
```

Global `.gitignore`

- Used for all repositories on your computer.
- Create it once and set it globally:

```
git config --global core.excludesfile ~/.gitignore_global
```

Include: OS or editor files (e.g., .DS_Store, .vscode/, .idea/).

Rules

`/` at the end means a directory.

Ignores the logs/ folder and everything inside it.

```
build/     # ignore files in the build directory
```

`*` means wildcards

- matches any number of characters
- Useful for patterns (not just exact names)

```
*.log      # all .log files
temp*      # files starting with 'temp'
*cache*    # any file containing 'cache'
```

Pattern Matching

- /storage/ Only ignores folder at repo root (/storage/)
    - We can specify specific directory: /storage/abc/def/xyz
- storage/ Ignores all folders named storage, anywhere

Example:

```
/storage/app.log    ← ignored
src/storage/app.log ← ignored only if pattern is `storage/`
```

Basic Examples

```
# OS files
.DS_Store
Thumbs.db

# Logs
*.log

# Temporary files
*.tmp
*.swp

---

# Python
__pycache__/
*.pyc

# Node.js
node_modules/
dist/
.env

# PHP / Laravel
/vendor/
.env
/storage/
```

7. The git status and git log

A, M, and ?? mark

The command `git status -s` displays a short format of file status.

Mark	Meaning	Details
A	Added	File staged for the first time
M	Modified	Changed file (`M` = staged, `M` = unstaged, `MM` = staged + modified again)
??	Untracked	New file not added yet
U	Unmerged	Merge conflict — file modified in both branches
D	Deleted	File removed (`D` = staged for deletion, `D` = deleted but not staged)
R	Renamed	File renamed (`R` = rename staged)
C	Copied	File copied (rare, needs special git config)
!!	Ignored	File is ignored by .gitignore (only shows with `--ignored`)

- **??** (Untracked): Files that Git doesn't track yet. You need to either:
  - Add them with `git add`
  - Ignore them with `.gitignore`

Example output:

```
$ git status -s
A  newfile.txt     # Added to staging
M  modified.txt    # Modified and staged
 M changed.txt     # Modified but not staged
?? ignored.log     # Not tracked by Git
```

Git log basics

- `git log` shows the commit history with full information
- `git log --oneline` shows compact view with one commit per line
- `git log --graph` shows a visual representation of branches
- `git log --all` shows all branches, not just current one

Example:

```
git log --oneline --graph --all
```

8. Diff, patch to understand CherryPick and Rebase

Git Diff

- `git diff` shows changes between commits, commit and working tree, etc.
- It helps understand what changed before committing

Common diff commands:

```
git diff              # Changes in working directory vs index
git diff --staged     # Changes in index vs last commit
git diff HEAD         # Changes in working directory vs last commit
git diff <commit1> <commit2>  # Between two commits
```

Patch

A **patch** is a text file containing the differences between files. It shows exactly what changed, making it easy to share or apply changes without sharing entire files.

Generating patches

```
# Create patch from unstaged changes
git diff > changes.patch
```

Reading diff output (patch)

**Original file (a/hello.py)

```
def greet():
    print("Hello")
    return True
```

**Modified file (b/hello.py)

```
def greet():
    print("Hello World")
    print("Welcome!")
    return True
```

Patch file structure

```
> diff --git a/hello.py b/hello.py
--- a/hello.py
+++ b/hello.py
@@ -1,3 +1,4 @@
 def greet():
-    print("Hello")
+    print("Hello World")
+    print("Welcome!")
     return True
```

- **diff --git**: Shows which files are being compared
- **index**: Git's internal object hashes
- **@@**: Line number info (-1,3 = old file starting line 1, showing 3 lines)
- **-**: Lines removed and **+**: Lines added

Applying patches

```
# Apply a patch
git apply changes.patch

# Check if patch can be applied (dry run)
git apply --check changes.patch

# See what patch will do without applying
git apply --stat changes.patch

# Apply patch and add to staging
git apply --index changes.patch
```

On macOS, if git apply doesn't work, you can use the traditional patch command.

Real-world example

1. Developer A creates changes:

```
echo "def calculate():" > math.py
echo "    return 2 + 2" >> math.py
git add math.py
git commit -m "Add math function"

# Make improvements
echo "    # This adds numbers" >> math.py
git diff > improvements.patch
```

2. Developer B applies the patch:

```
# Check what the patch contains
cat improvements.patch

# Test if it applies cleanly
git apply --check improvements.patch

# Apply it
git apply improvements.patch
```

Cherry-pick and Rebase: Automated Patching

Both `git cherry-pick` and `git rebase` essentially **create and apply patches automatically**:

Cherry-pick = Extract patch + Apply

```
# This command:
git cherry-pick abc123

# Is conceptually similar to:
git format-patch -1 abc123           # Extract commit as patch
git apply --3way abc123.patch       # Apply to current branch
```

```
# Cherry-pick
Original: A---B---C---D (main)
               \
                E---F (feature)

After cherry-pick C to feature:
          A---B---C---D (main)
               \
                E---F---C' (feature)
                        ↑
                   C' is a patch of C applied on F
```

Rebase = Series of patches

```
# This command:
git checkout feature
git rebase main

# Git internally does this:
git format-patch main..feature      # Extract YOUR commits as patches
git reset --hard main               # Move feature branch to main's tip
git am *.patch                      # Apply YOUR patches on top of main
```

```
Before rebase:
          A---B---C (main)
           \
            D---E---F (feature) ← you are here

After git rebase main:
          A---B---C (main)
                   \
                    D'---E'---F' (feature) ← your commits replayed
```

Why this matters

Understanding the patch concept helps explain:

1. **Why conflicts happen** - When the "patch" can't apply cleanly
2. **Why commit hashes change** - New patch applied to different base = new commit
3. **How Git moves code** - It's not moving commits, it's creating new ones with same changes

Interactive rebase = Edit patches before applying

```
git rebase -i HEAD~3
```

This lets you:

- **pick** = Apply patch as-is
- **edit** = Apply patch, then modify
- **squash** = Combine patches
- **drop** = Don't apply patch

It's like having access to the patch files before applying them!

Real-world analogy:

- Cherry-pick = Photocopy one page and paste it elsewhere
- Rebase = Photocopy several pages and paste them in a new location

9. Git Merge

Merge combines changes from different branches:

```
git checkout main
git merge feature-branch
```

Merge conflicts occur when:

- Same line is modified in both branches
- File deleted in one branch but modified in another

Resolving conflicts:

```
<<<<<<< HEAD
print("Main branch version")
=======
print("Feature branch version")
>>>>>>> feature-branch
```

1. Edit the file to keep desired changes
2. Remove conflict markers (<<<<, ====, >>>>)
3. `git add <file>`
4. `git commit`

Fast Forward vs Three-way merge

Fast Forward Merge

When: Target branch has no new commits since the feature branch was created

```
# Before merge
Main:    A---B
                \
Feature:         C---D

# After fast-forward merge
Main:    A---B---C---D
```

Command:

```
git merge feature-branch  # Git does fast-forward automatically

# Default (same as --ff) - Fast-forward when possible
git merge feature-branch
git merge --ff feature-branch        # Explicitly saying "allow fast-forward"

# Force merge commit (no fast-forward)
git merge --no-ff feature-branch     # Creates merge commit even if FF possible

# Fast-forward ONLY
git merge --ff-only feature-branch   # Fails if fast-forward not possible
```

Characteristics:

- No merge commit created
- Linear history
- Simple and clean

Three-way Merge

When: Both branches have new commits

```
# Before merge
Main:    A---B---E---F
               \
Feature:        C---D

# After three-way merge
Main:    A---B---E---F---M
               \         /
Feature:        C-------D
```

Command:

```
git merge feature-branch  # Creates merge commit M
```

Characteristics:

- Creates a merge commit (M)
- Preserves complete history
- Shows branch structure

Why "three-way" merge enables automation

Case 1: Only one side changes → Git auto-merges! ✓

```
Ancestor: total += item.price
File A:   total += item.price          (no change)
File B:   total += item.price * 0.9    (added discount)

# Git thinks: "Only B changed, so use B's version"
# Result: total += item.price * 0.9    ← Automated!
```

Case 2: Both sides change differently → Conflict ✕

```
Ancestor: total += item.price
File A:   total += item.price * 1.08   (added tax)
File B:   total += item.price * 0.9    (added discount)

# Git thinks: "Both changed the same line differently"
# Result: CONFLICT — human must decide
```

The magic of three-way merge

Without ancestor (two-way): Can't auto-merge anything!

```
File A: total += item.price
File B: total += item.price * 0.9

# Which is the "new" version? No way to know!
```

With ancestor (three-way): Can auto-merge most changes!

```
# Git automatically handles:
✓ Changes in different files
✓ Changes in different parts of same file
✓ Changes where only one side modified
✓ Additions that don't overlap
✓ Deletions that don't conflict
```

```
# Git only asks humans when:
✗ Both sides change the same lines
✗ One side edits, other side deletes
✗ Complex restructuring conflicts
```

That's why Git successfully auto-merges ~95% of changes!

The three-way comparison makes this possible.