Git Under the Hood

Step by step lab guide for exploring how Git works internally

This handout is designed as a practical companion to the talk. It is not a list of commands to memorize. It is a sequence of small experiments that reveal what Git stores, where it stores it, and why Git behavior becomes predictable once the object model clicks.

All examples are safe in a throwaway folder. Use a fresh demo repository and do not run destructive commands inside real work projects unless you know exactly what they do. Git is friendly, but it is also a tiny database with knives.

Note: Shell differences matter. macOS and Linux examples use Bash or zsh. Windows examples use PowerShell when needed.

---

0. Setup: create a clean demo repo

Core idea: Start with a tiny repository so every object Git creates is easy to inspect.

```
mkdir git-under-hood-lab
cd git-under-hood-lab
git init

git config user.name "Git Demo"
git config user.email "git-demo@example.com"
```

If your default branch is called **master**, keep using master. If it is called **main**, use main. The ideas are identical.

Summary: At this point Git has created the hidden **.git** directory. That directory is the repository. Your files are only the working directory.

---

1. Hashing: Git gives content an identity

Core idea: Git does not store file content by filename first. It stores content by hash. Same content means same identity.

```
echo "hello git" > hello.txt

git hash-object hello.txt
git hash-object -w hello.txt
```

The first command calculates the object ID Git would use. The second command stores the object in Git's object database.

Find where Git stored it

Suppose the hash starts with **ce013625...**. Git splits the hash into two parts. The first two characters become the folder name. The rest becomes the file name.

```
# macOS/Linux
find .git/objects -type f

# PowerShell
Get-ChildItem .git/objects -Recurse -File
```

The object file contains the real file content, but compressed. The hash is not the content. The hash is the address.

```
git cat-file -t <hash>
git cat-file -p <hash>
```

Summary: Git stores the actual compressed content under a hash-based address. You cannot reverse a hash to get the file. Git keeps the content too.

---

2. Same content, same hash

Core idea: Git cares about content, not filenames.

```
cp hello.txt another-name.txt

git hash-object hello.txt
git hash-object another-name.txt
```

Both files produce the same hash because their contents are identical.

```
echo "hello git!" > hello.txt
git hash-object hello.txt
```

One extra character creates a new hash. That is why Git can detect exact content changes so reliably.

Summary: The content determines the identity. File names are labels added later by tree objects.

---

3. Objects: where the files actually live

Core idea: The object database lives in **.git/objects**.

Loose objects are stored as individual compressed files:

```
.git/objects/ab/cdef123...
.git/objects/ce/013625...
```

Git object types include:

Object type	What it means
blob	File content
tree	Directory structure
commit	Snapshot plus parent and metadata
tag	Annotated tag object

Summary: The hash is the lookup key. The object file is the stored data.

---

4. Trees: Git learns filenames and folders

Core idea: A tree object represents a directory. It maps names to objects.

```
mkdir src
echo "console.log('hello')" > src/app.js
echo "readme" > README.md

git add .
git commit -m "add structure"
```

Inspect the commit and tree

```
git cat-file -p HEAD
```

You will see a **tree** line. Copy that tree hash and inspect it:

```
git cat-file -p <tree-hash>
```

You might see entries like this:

```
100644 blob <hash>    README.md
040000 tree <hash>    src
```

100644 means a normal file. **040000** means a directory. The tree gives names to blobs and points to subtrees.

Summary: Blobs store content. Trees give names and structure to that content.

---

5. Commits: Git learns history

Core idea: A commit is a snapshot with memory. It points to a tree and usually to a parent commit.

```
echo "more content" >> README.md
git add .
git commit -m "update readme"

git cat-file -p HEAD
```

Look for these lines:

```
tree <tree-hash>
parent <parent-commit-hash>
author ...
committer ...
```

Git history is not magic. A commit points to the project snapshot, then points back to what came before.

Summary: Commits store snapshots logically, not diffs. The parent chain creates history.

---

6. Packfiles: Git learns efficiency

Core idea: Snapshots sound expensive, but Git stores them efficiently.

Create a bunch of small commits so there is something to pack.

```
# macOS/Linux
for i in {1..20}; do
  echo $i >> demo.txt
  git add .
  git commit -m "commit $i"
done

# PowerShell
1..20 | ForEach-Object {
  Add-Content demo.txt $_
  git add .
  git commit -m "commit $_"
}
```

Count loose objects

```
# macOS/Linux
find .git/objects -type f | wc -l

# PowerShell
(Get-ChildItem .git/objects -Recurse -File).Count
```

Pack the objects

```
git gc

# macOS/Linux
ls .git/objects/pack
```

```
# PowerShell
Get-ChildItem .git/objects/pack
```

Git moves many loose objects into packfiles. The **.pack** file stores objects. The **.idx** file is a lookup index.

Optional deep inspection

```
# macOS/Linux
git verify-pack -v .git/objects/pack/*.idx

# PowerShell
$idx = Get-ChildItem .git/objects/pack/*.idx | Select-Object -First 1
git verify-pack -v $idx.FullName
```

Summary: Git gives you the simple mental model of snapshots, but internally stores objects like an optimized compressed database.

---

7. Branches: Git creates movable pointers

Core idea: A branch is just a name pointing to a commit.

```
git branch feature-demo

# macOS/Linux
cat .git/refs/heads/feature-demo

# PowerShell
Get-Content .git/refs/heads/feature-demo
```

That file contains a commit hash. That is the branch.

```
git checkout feature-demo
echo "feature work" >> feature.txt
git add .
git commit -m "feature work"

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

Summary: Branches are cheap because they do not copy files. They are movable labels pointing at commits.

---

8. HEAD: where am I right now

Core idea: HEAD tells Git what your current location is.

```
# macOS/Linux
cat .git/HEAD

# PowerShell
Get-Content .git/HEAD
```

Normally HEAD points to a branch:

```
ref: refs/heads/feature-demo
```

Then the branch points to a commit. The chain is:

```
HEAD -> branch -> commit
```

Summary: HEAD is your current position. Usually it points to a branch, not directly to a commit.

---

9. Merging: combining histories

Core idea: A merge commit is a commit with two parents.

```
git checkout master
# or: git checkout main

echo "main work" >> main.txt
git add .
git commit -m "main work"

git checkout feature-demo
echo "feature more work" >> feature.txt
git add .
git commit -m "feature more work"

git checkout master
# or: git checkout main

git merge feature-demo
```

If Git creates a merge commit, inspect it:

```
git cat-file -p HEAD
```

A merge commit has two **parent** lines. That is how Git records that two histories were connected.

Summary: Merge does not rewrite history. It connects histories with a new commit.

---

10. Rebase: replaying commits

Core idea: Rebase copies changes and creates new commits with new parents.

```
git checkout -b rebase-demo

echo "change A" >> rebase.txt
git add .
git commit -m "change A"

echo "change B" >> rebase.txt
git add .
git commit -m "change B"

git checkout master
# or: git checkout main

echo "main moved" >> main.txt
git add .
git commit -m "main moved"

git checkout rebase-demo
git rebase master
# or: git rebase main

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

The old commits are not edited. Git replays their changes on top of the new base and creates new commits.

Summary: Rebase is not moving commits. It is making new commits that contain the same changes in a new place.

---

11. Cherry-pick: copying one change

Core idea: Cherry-pick is like rebase for one commit. It copies the patch, not the commit object.

```
git checkout -b feature-mini-act
```

```
echo "important fix" >> demo.txt
git add .
git commit -m "important fix"

echo "unfinished work" >> demo.txt
git add .
git commit -m "unfinished work"

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

Copy the hash of **important fix**. Then return to your main branch and cherry-pick only that commit.

```
git checkout master
# or: git checkout main

git cherry-pick <hash-of-important-fix>

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

You now have two commits with similar messages and the same code change, but different hashes.

```
git cat-file -p <original-important-fix>
git cat-file -p HEAD
```

The parent is different, so the commit identity is different.

Summary: Cherry-pick is useful for hotfixes, backports, and commits made on the wrong branch.

---

12. Squash merge: keeping the result, dropping the process

Core idea: A squash merge takes the final content from a branch and creates one new commit.

```
git checkout -b feature-squash

echo "step 1" >> squash.txt
git add .
git commit -m "wip 1"

echo "step 2" >> squash.txt
git add .
git commit -m "wip 2"

echo "step 3" >> squash.txt
git add .
git commit -m "fix typo"

git checkout master
# or: git checkout main

git merge --squash feature-squash
git status
git commit -m "add feature as one clean commit"

git cat-file -p HEAD
```

There is only one parent. A squash merge is not a real merge commit.

Summary: Squash keeps the final result but removes the internal feature branch story from the target branch.

---

13. The staging area: the next snapshot

Core idea: Git commit does not save your working directory. It saves the staging area.

```
echo "staged line" >> staged.txt
git add staged.txt
```

```
echo "unstaged line" >> staged.txt

git status
git diff
git diff --staged
```

git diff shows unstaged changes. **git diff --staged** shows what will be committed.

```
git commit -m "commit staged snapshot"
```

Summary: The staging area is the exact snapshot Git will use for the next commit.

---

14. What git commit does behind the scenes

Core idea: git commit is a friendly wrapper around several lower-level steps.

Conceptually, Git does this:

```
1. Read the staging area
2. Write a tree object
3. Create a commit object pointing to that tree
4. Add the parent commit hash
5. Move the current branch pointer forward
```

Explore manually

```
git write-tree
git commit-tree <tree-hash> -p HEAD -m "manual commit"
```

The **git commit-tree** command creates a commit object but does not move your branch by itself. That is plumbing territory. Look, but maybe do not live there.

Summary: Porcelain commands are human-friendly. Plumbing commands expose the machinery.

---

15. Reset and reflog: work is often not gone

Core idea: Reset moves pointers and optionally changes the staging area and working files.

```
echo "temporary work" >> reset.txt
git add .
git commit -m "temporary work"

git log --oneline
git reset --hard HEAD~1
git log --oneline
```

The commit appears to be gone. Now check reflog:

```
git reflog
git checkout <lost-commit-hash>
```

You are now on the commit directly. To rescue it:

```
git branch rescued-work
git checkout rescued-work
```

Summary: Most lost commits are really lost pointers. Reflog is the map of where HEAD and branches used to point.

---

16. Detached HEAD: scary name, simple idea

Core idea: Detached HEAD means HEAD points directly to a commit instead of a branch.

```
git checkout HEAD~1

# macOS/Linux
cat .git/HEAD

# PowerShell
Get-Content .git/HEAD
```

Now HEAD contains a commit hash directly. You can still make commits here, but no branch will move with you.

```
echo "detached work" >> detached.txt
git add .
git commit -m "detached work"

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

To keep that work, create a branch before leaving:

```
git branch rescue-detached
git checkout rescue-detached
```

Summary: Detached HEAD is not broken Git. It is Git saying: you are standing on a commit, not on a branch.

---

17. Worktrees: multiple branches checked out at once

Core idea: A worktree is another working directory attached to the same Git object database.

```
git worktree add ../worktree-feature worktree-feature
git worktree list
```

Open the new folder's .git file:

```
# macOS/Linux
cat ../worktree-feature/.git

# PowerShell
Get-Content ../worktree-feature/.git
```

It is not a full .git directory. It points back to metadata inside the original repo.

```
git rev-parse --git-common-dir
```

Worktrees share objects, commits, trees, and packfiles. Each worktree has its own HEAD, index, and working files.

Summary: This is useful for parallel work, hotfixes, experiments, and AI coding agents working on multiple branches at the same time.

---

18. Plumbing commands: should you use them

Core idea: Plumbing commands are low-level Git building blocks. Porcelain commands are the friendly interface.

Porcelain	Possible plumbing underneath
git add	git update-index
git commit	git write-tree, git commit-tree
git show	git cat-file
git log	git rev-list
git status	git diff-index

You usually should not use plumbing commands for everyday work. They are excellent for learning, debugging weird repository states, and writing advanced automation.

Summary: Porcelain helps humans use Git. Plumbing shows how Git actually works.

---

19. Decision guide: merge, rebase, squash, cherry-pick

Situation	Good choice	Why
Need to preserve the full branch story	merge	Keeps both histories and creates a connecting commit
Need a clean linear history before sharing	rebase	Replays commits on top of a newer base
Feature branch contains noisy WIP commits	squash	Keeps final result as one clean commit
Need one fix without the whole branch	cherry-pick	Copies one change to another branch
Need to recover something that vanished	reflog	Finds where HEAD or refs used to point

Summary: Git becomes less scary when you stop thinking in commands and start thinking in objects, snapshots, and pointers.

---

20. Final mental model

Git is not mainly a folder backup tool. It is a content-addressable object database with a version control interface.

```
blob    = file content
tree    = folder structure
commit  = snapshot plus parent and metadata
branch  = movable pointer to a commit
HEAD    = current location
index   = next snapshot
reflog  = recent pointer history
pack    = optimized object storage
```

The whole talk can be condensed into this:

Objects do not change. Pointers move.

Once that clicks, Git commands stop feeling like random spells and start looking like predictable database operations.