Under the Hood

- Useful videos
    - https://www.youtube.com/watch?v=ig5E8CcdM9g (this guy is still heading training at GitHub)
    - https://www.youtube.com/watch?v=MYP56QJpDr4

Intro (SLIDES)

- **[move into sides]**
- Start by talking about how Git stores data under the hood; build on concepts later
    - We have quite a bit of material to get over, so may move relatively quickly in some places; don't worry, the point is mostly to expose you to new commands and ideas so you can explore them in more detail yourself. There are reference materials available that covers a lot of the more detail-oriented stuff.
- "What" - Git is like DB for your project — not only stores contents of files/folders, but also stores entire history of all files/folders. Think of Git **commands** as the SQL to query/manipulate DB. Git has two types of commands: the porcelain, and the plumbing:
    - Porcelain are the nice outside-facing commands you interact with
    - Plumbing are lower-level commands that tie things together
- "How" - Important thing to keep in mind: Git doesn't store changes between files, unlike Subversion, Mercurial, and others.
    - Git stores entire files, but does so in a way that minimizes wasted space; more later
- If simplified a lot, Git is made up of four major object types: blobs, trees, commits, refs. Almost every command affects one of these object types.
    - **Blobs** store contents of files
    - **Trees** store contents of directories
    - **Commits** store metadata about a snapshot as well as a pointer to a tree
    - **Refs** point, or refer to, other objects via more friendly names
- Let's build some objects to get a better understanding. Feel free to experiment with me in your shells, though some things will only work on OS X or Linux.

Looking at .git folder

- Start w/ simple set of steps: creating a file, adding it to git, and making a commit. First, **let's create a new repo**:
    - $ mkdir testing1
    - $ cd testing1
    - $ git init
- The only folder we have now is the \`.git\` folder. This folder *is* the Git repository. It contains everything Git knows about related to the project.
- **Let's look inside**:
    - $ find .git
- There's not really many files! I'm going to use a special program on my computer to viz better:
    - Split window
    - $ watch -n 1 'tree .git'
- There's this \`hooks\` directory that contains sample files for hooks, we may talk about later but can ignore for now. Leaves us with very few things:

    - In root: HEAD, config, description
    - `info` folder with `exclude` in it
    - Empty folders: `objects/info`, `objects/pack`, `refs/heads`, `refs/tags`
- That's it! Not much to a new repo at all. **Let's create a file, add it, commit it, and see how it changes**.
    - $ echo "Hello world" > hello.txt
    - $ git add hello.txt
- Woah! Notice that we have a new file in our repository. **Let's look**:
    - $ cat .git/objects/…
- This isn't readable. I'll give you a hint and say that it's compressed with Gzip. I'll create a little alias that uses Perl to decompress
    - $ alias deflate="perl -MCompress::Zlib -e 'undef $/; print uncompress(<>)'"
    - $ deflate .git/objects/sha
- Okay, so we have the word blob, the number 12, and the text "Hello world". It turns out that this is how Git stores objects with info. The SHA is comprised of:
    - Object type
    - Space
    - Number of bytes the content takes up
    - A null byte we can't see
    - The bytes that make up the content
- After creating the SHA, git creates a file and places it in the the first two-character's folder; the file is titled as the remaining 38 characters file that make up SHA
    - But where did Git come up with the SHA?
- **Let's try something**... only works if you have a `shasum` cmd
    - $ printf "blob 12\000Hello world\n" | shasum
- So we can see that Git gets the SHA of an object by literally taking the SHA1 digest of the contents of the object
- That means objects with the same contents have the same SHA - important later!

Git plumbing command cat-file (**SLIDES**)

Commits -> Trees -> Blobs

- I mentioned that we git added our file to the index. We have a plumbing command to look at the index, too! It's git ls-files -s -- Let's take a look at the index:
    - $ git ls-files -s
- And here we can see the SHA for the blob Git created.
    - This says that, when we make our commit, we want this version of hello.txt to be included.
- **Let's see what that looks like**
    - $ git commit -m "First commit"
- Oo look, we got a bunch of new files! **Let's look at them**
    - $ tree .git
        - There's two new objects…
        - ...refs/heads/master…
        - ...and logs/refs/heads/master
- So what are the new files? **Let's take a look at the types…**
    - $ git cat-file -t SHA1
    - $ git cat-file -t SHA2

- Okay, here we have a tree, and a commit. Let's look more closely at the commit to see what it looks like.
    - $ git cat-file -p COMMIT_SHA
- Okay, we see that I authored and committed this
    - It also has the date and time I made the commit, and the commit message
    - And notice it points to a SHA at the top here
    - It's the same as the tree we just looked up
    - The SHA of a tree, like any other object, is made up of the sha 1 digest of its contents; so let's look at the content
- **Let's take a look at the tree**
    - This is the root directory this commit.
    - It shows all the files and folders that existed when this commit was made (currently just hello.txt)
    - Notice that the SHA of the hello.txt file is the one we saw earlier on the index

Review what just happened (**SLIDES**)

Changing an existing committed file's content

- **Let's change the text in our hello world file and commit it**
    - $ echo "Hello Git" > hello.txt
- **[PAUSE and ask how many objects will be created when we add, answer = 1]**
    - $ git add hello.txt
- **[PAUSE and ask how many objects will be created when we commit, answer = 2]**
    - $ git commit -m "Change text"
- We can see in the commit output what our new commit SHA is. **Let's look at it**
    - $ git cat-file -p NEWSHA
- So we have another tree, author, committer, message.
    - Every commit other than the first one has at least one parent
    - You can trace commits, generally, from newer > older by following
- **Let's look at the tree**
    - We still have hello.txt, but now it's next to a different blob
- **Let's look at the blob**
- It's important to note that this file has no information about what the file *used* to be, or any other information about what *changed*, only what *is*
    - That's what I mean when I say Git doesn't store deltas
    - Every commit contains an *entire copy* of the repo contents at that point
    - The diff w/ +/- in `git log` ($ git log -p) is being computed on the fly

Committing a second file

- If every commit has a tree that has a full copy of repo, why doesn't size grow more quickly?
- **Let's create a second file, stage it, and commit**
    - $ echo "More text" > second.txt
- **[PAUSE and ask, answer = 1]**
    - $ git add second.txt
- **[PAUSE and ask, answer = 2]**
    - $ git commit -m "Add second.txt"
- Now **let's compare the tree we just created**.... With the tree we had in the prior commit

    - $ git cat-file -p head^{tree}
    - $ git cat-file -p head^^{tree}
- So notice in the new tree that we have our new file with a new blob
    - But the new tree has the *same* blob for hello.txt!
- **[ASK WHY]**
- Git bases the SHA on the contents, and we didn't change the contents of hello.txt
    - Git can fully reuse that object for ANY file that contains the same data
    - Turns out most commits don't touch that many files; usually only changing a few
    - That means that Git can reuse most of the blobs from one commit to the next
        - Make a hundred commits and don't change a file, Git will never create a new blob for it

Committing a file in a subfolder with the same content

- **Let's take a look at subfolders**
    - $ mkdir subfolder
    - $ echo "Hello Git" > subfolder/subfile.txt
- **[PAUSE and ask, answer = 0]**
    - $ git add subfolder
- **[PAUSE and ask, answer = 3]**
    - $ git commit -m "Add subfolder"
- The two blobs are the same (same content). Now there's a new tree. **Let's look at it**
    - $ git cat-file -p TREE_SHA
- This subfolder has a file in it, but notice that the SHA is the same sha for hello.txt in the parent folder — again, since the files have the same *content*, they get the same object

Review Refs, Branches, and Tags (**SLIDES**)

Refs, Branches, and Tags

Branches

- Let's remove the objects from our output
    - $ watch -n 1 'tree .git -I "hooks|objects"'
- Branches live in refs/heads
    - $ cat .git/refs/heads/master
- Simply a plain text file that has a SHA in it. If we look at the Git log
    - alias glol='git log --graph --pretty=format:'\''%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset'\'' --abbrev-commit'
    - $ glol
- We can see that that's the SHAs match.
- That makes branches straightforward
    - They're stored as refs, which are text files pointing to other objects
- Let's create a branch by hand that points to the last commit
- First, let's use another plumbing command to turn head~, which is the first ancestor of head, into a SHA:
    - $ git rev-parse head~1
- Now let's put that text into .git/refs/heads/newbranch
    - $ git rev-parse head~1 > .git/refs/heads/newbranch

- And that's it! Let's check it out with
    - $ git branch
- And
    - $ git checkout newbranch
- If you use a git log format that shows branches, you can see it there too
    - $ git checkout master
    - $ glol

HEAD

- There are a select few refs that get special treatment; one is known as HEAD, and it refers to the commit you're on at any moment. We can see what it is by **looking at the HEAD file**:
    - $ cat .git/HEAD
- This is a bit different; saying that HEAD is attached to `master`, and that Git should to look at `master` to figure out exactly which commit we're on.
- **If we checkout a specific commit:**
    - $ git checkout head~1
- We're in what's known as "detached head state"
- You can do whatever you want but unless you create a tag to get back to it, you can't get back to it easily…. The only way to get back to it easily as you create more commits is by checking out the sha
- Means HEAD doesn't point to a branch, but directly to a commit. **Let's see**
    - $ cat .git/HEAD
- In reverse, you can also change the branch you are looking at by modifying the .git/HEAD file
    - $ echo 'ref: refs/heads/master' > .git/HEAD
- Creating a new commit automatically moves HEAD to that commit
    - $ echo "Keep adding text" > fourth.txt
    - $ git add fourth.txt
    - $ git commit -m "Add fourth.txt"
    - $ cat .git/HEAD
- If HEAD is pointing to a branch it also updates that branch to also point at the new commit

Tags

- Lastly we'll talk about another kind of ref: tags
- Tags are mostly like branches, except they don't move
    - You can think of branches as alive and growing; you use branches as you're actively developing (like a WIP)
    - Whereas tags are static and just stick to it; you use tags when you want to label a commit as meaningful (releases, milestones) (commemorating a past)
- You can create a basic tag with `git tag`
    - $ git checkout master
    - $ git tag mytag
- We can take a look by looking in .git/refs/tags
    - $ cat .git/refs/tags/mytag
- It's pointing to a sha, let's see what that object's type is
    - $ git cat-file -t THESHA
- So it's pointing to our commit. So this ref works exactly the same way as the branch
- You can also create something called an *annotated tag*
    - They're much like regular tag, but include an author and a message

    - Useful for things like deploy versions or other more "permanent" pointers
    - Useful when you want to add other metadata (add people to release in notes) (google tags v annotated tags)
- Let's create one
    - $ git tag -a v1.0.0-alpha
    - $ cat .git/refs/tags/v1.0.0-alpha
    - But that's a different SHA
- Let's look at the object type
    - $ git cat-file -t SHA_TAG
- We can see that it's a *tag* object. THIS IS THE 4th and LAST kind of git object! If we look at the file
    - $ git cat-file -p SHA_TAG
- We can see that it's this special tag object that has my name and the message in it

Reflog

- now let’s take a look at the last unexplored bit of our .git folder - you might be wondering what this logs folder is for
- any guesses? understandable guess is `git log`
    - `git log` is actually from walking up the parent line
    - git cat-file -p HEAD, git cat-file -p parents …
- rather, they are a history of everywhere the ref has been . a ref is just a pointer - we have
    - cat .git/logs/HEAD
- git reflog is a handy command which stands for “reference log”. Basically for any action that you perform inside of Git where data is stored, you can find it inside of the reflog. It’s your safety net that you can go to after you reset something but needed code in a commit that you lost. You can find that commit sha, and show/commit/cherry-pick or whatever you need to save your life.
    - git reflog
- What’s in the last folder, .git/logs/refs/heads/master?
    - cat .git/logs/refs/heads/master
        - git reflog master
- we’ll see how this comes in super handy later on in Katrina’s presentation

Conclusion

And that's it! You now know almost every important piece of how Git stores data under the hood. It’s lunch time!!