Git Internal

Plumbing commands

git commit --amend

Changing the Last Commit

- The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit.
- It can also be used to simply edit the previous commit message without changing its snapshot. But, amending does not just alter the most recent commit, it replaces it entirely, meaning the amended commit will be a new entity with its own ref

Change most recent Git commit message

- git commit –amend
- git commit --amend -m "an updated commit message"

Changing committed files

- Let's say we've edited a few files that we would like to commit in a single snapshot, but then we forget to add one of the files the first time around. Fixing the error is simply a matter of staging the other file and committing with the --amendflag:

- git commit --amend --no-edit

- The --no-edit flag will allow you to make the amendment to your commit without changing its commit message. The resulting commit will replace the incomplete one, and it will look like we committed the changes
to hello.py and main.py in a single snapshot.

Don’t amend public commits

- Amended commits are actually entirely new commits and the previous commit will no longer be on your current branch. This has the same consequences as resetting a public snapshot. Avoid amending a commit that other developers have based their work on. This is a confusing situation for developers to be in and it’s complicated to recover from.

Refernece

- https://www.atlassian.com/git/tutorials/rewriting-history

Git bisect

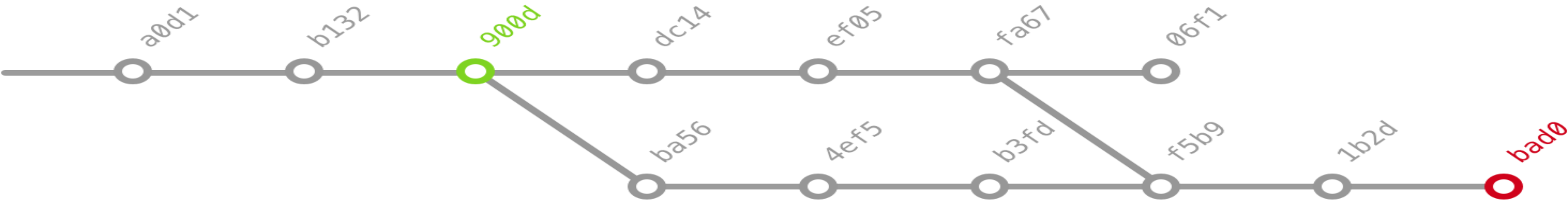
Find the change that introduced a bug in your code, **quickly**.

Git bisect

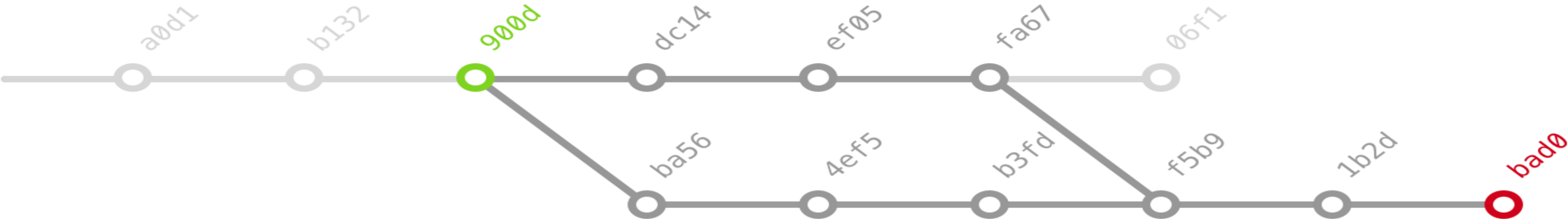
- The idea behind git bisect is to perform a binary search in the history to find a particular regression
- You have a git commit in your history that is causing a bug but you do not know which commit it is.
  Here's how to use git bisect to find the commit that causes the bug.

- Git uses the bisection algorithm to help you search the offending commit. To start, you need to mark a bad commit and
  a good commit, git will checkout a commit in the middle for you to test. Then you mark it either as good or bad, and then the process starts again.

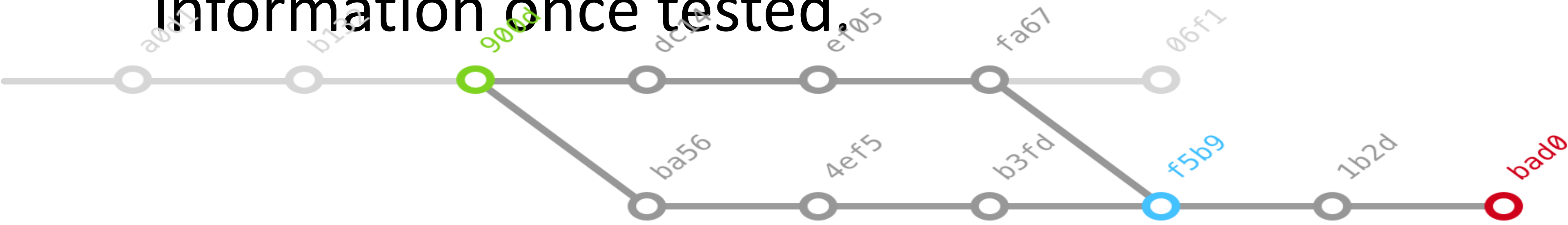
We start by identifying a good (900d) and a bad (bad0) commit



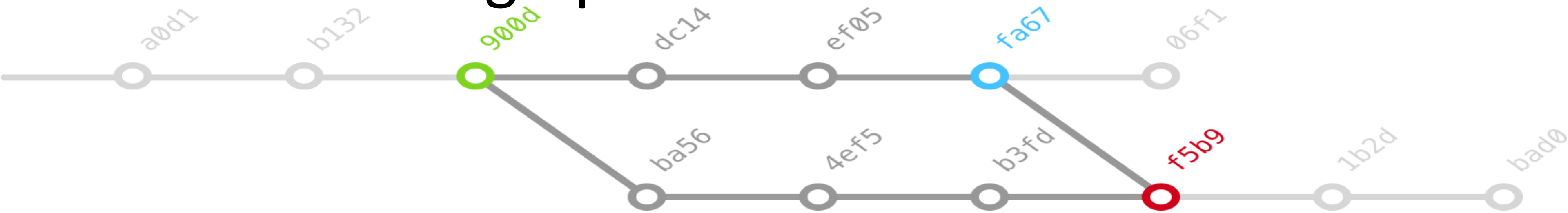
- Git will keep the commits that are both descendants of the good commit and ancestors of the bad commit, which will leave us with a smaller graph to work with.



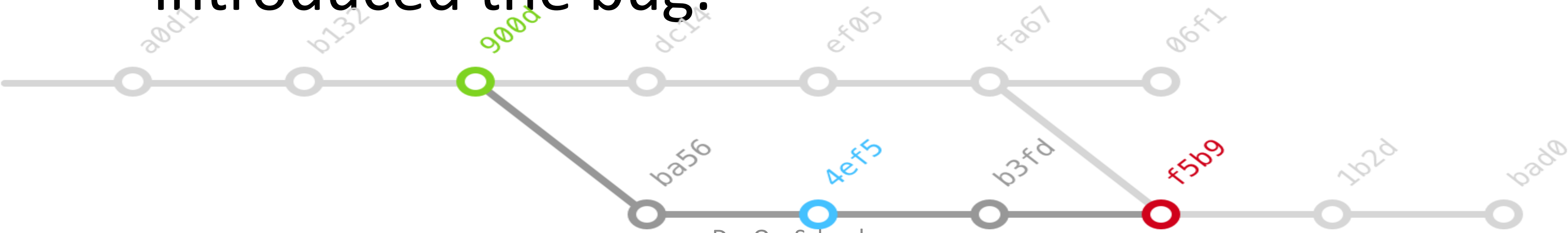
- Git chooses a commit between the good and bad commits. As this is a directed acyclic graph, there is no commit “in the middle”, git chooses the commit that will provide more information once tested.



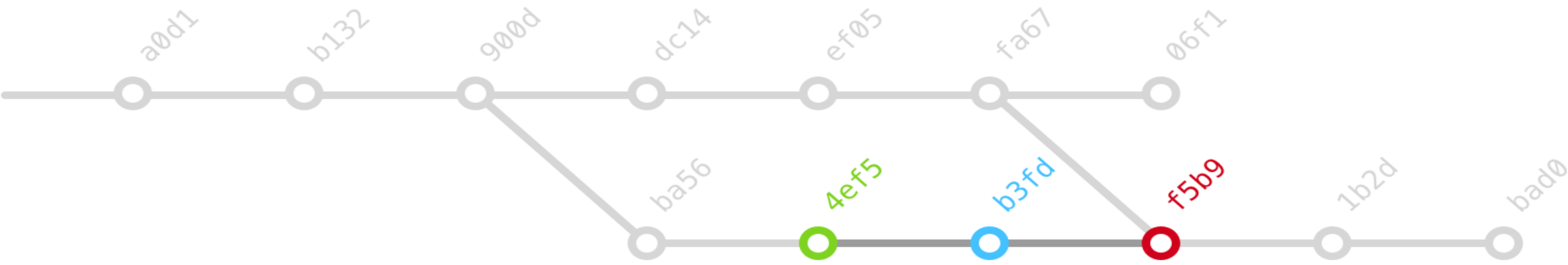
- Now we need to test this commit as being good or bad. Let’s say it was bad, we mark it as such and git proposes a new commit to be tested. The graph now is smaller.



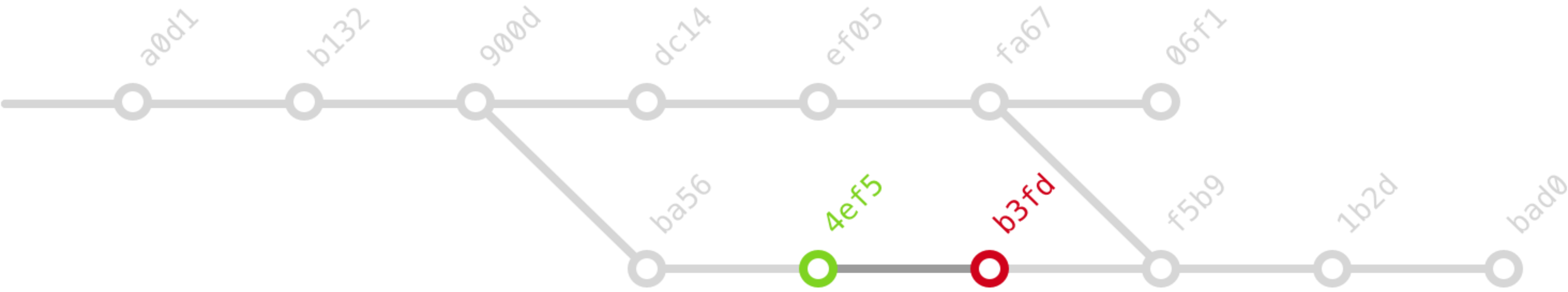
- We test this commit as well (let’s say it was good). In this case, the upper branch is removed from the commits to test, as we are under the assumption that only **one** commit introduced the bug.



- We test the next commit (it was good) and git proposes the last comit to be tested.



- We test this last commit and we are done, the first bad commit was b3fd.



```
# in the git root, start the git bisect
> git bisect start

# mark current commit as bad
> git bisect bad

# tell git which is the commit is good (it can be any good commit in the
    far history)
> git bisect good <commit-hash>
```

# git will tell you roughly how many revisions you need to find the bad commit

# eg. Bisecting: 24 revisions left to test after this (roughly 5 steps)

# You are now at a commit that is chosen by git bisect

# Run your test to determine if the commit is okay

# if commit is okay, then

> git bisect good

# if commit is bad, then

> git bisect bad

# Repeat the above until git tells you which is the bad commit
# eg. f5836a61c5e989b972499e5265760519580d3cfe is the first bad commit
# Note down your bad commit hash

# Now that we know which commit is the bad one, we can look at the diff to see what changed:

```
>  git show 5e3c5e
```

# Upon finishing, then

```
> git bisect reset
```

# Work on your bad commit!!!

Reference

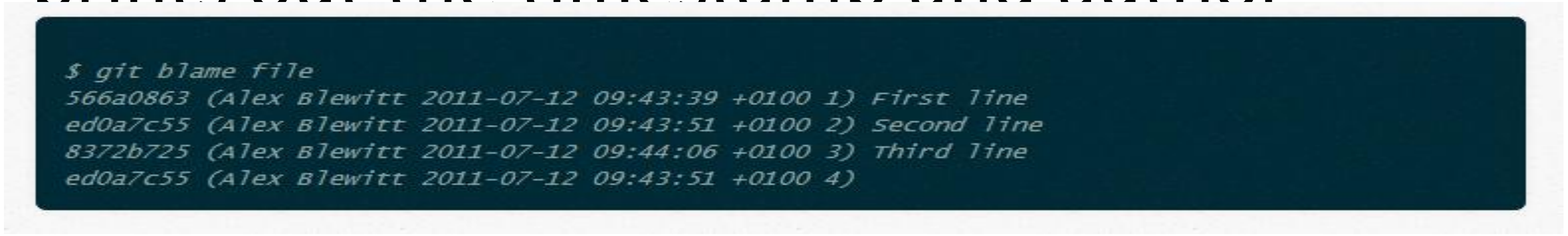
- https://github.com/scmgalaxy/howto-git-bisect
- https://git-scm.com/docs/git-bisect

Git blame

Show what revision and author last modified each line of a file

- Annotates each line in the **given file** with information from the revision which last modified the line. Optionally, start annotating from the **given revision**.
- When specified one or more times, -L restricts annotation to the requested lines.

- To find out who changed a file, you can run git blame against a single file, and you get a breakdown of the file, line-by-line, with the change that last affected that line. It also prints out the timestamp and author



```
$ git blame file
566a0863 (Alex Blewitt 2011-07-12 09:43:39 +0100 1) First line
ed0a7c55 (Alex Blewitt 2011-07-12 09:43:51 +0100 2) Second line
8372b725 (Alex Blewitt 2011-07-12 09:44:06 +0100 3) Third line
ed0a7c55 (Alex Blewitt 2011-07-12 09:43:51 +0100 4)
```

In Simple

- The command explains itself quite well, it's to figure out which co-worker wrote the specific line or ruined the project so you can **blame** them

- Please note that git blame **does not** show the per-line modifications history in the chronological sense. It only show who was the last person to have changed a line in a document up to the last commit in HEAD.
- That is to say that in order to see the full history/log of a document line, you would need to run a git blame path/to/file for each commit in your git log.

Example

- git blame filename
- git blame -l filename

What's Wrong with git blame?

- git blame only shows the **last person to modify a line.** This is rarely what you want. More often what you want is the **original author of a line.**

Solution

- git log -p -M --follow --stat -- path/to/your/file

Reference

- https://blog.andrewray.me/a-better-git-blame/

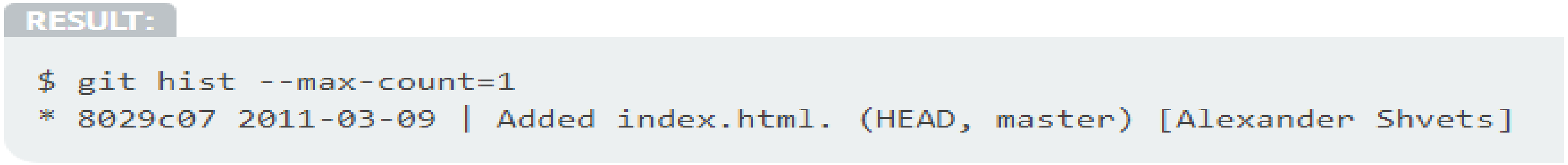
git cat-file

Provide content or type and size information for repository objects

- Explore the structure of the database objects
- Using SHA1 hashes for searching the content in repository

Searching for the last commit

- This command should find the last commit in the repository. SHA1 hash is probably different on our systems; however you should see

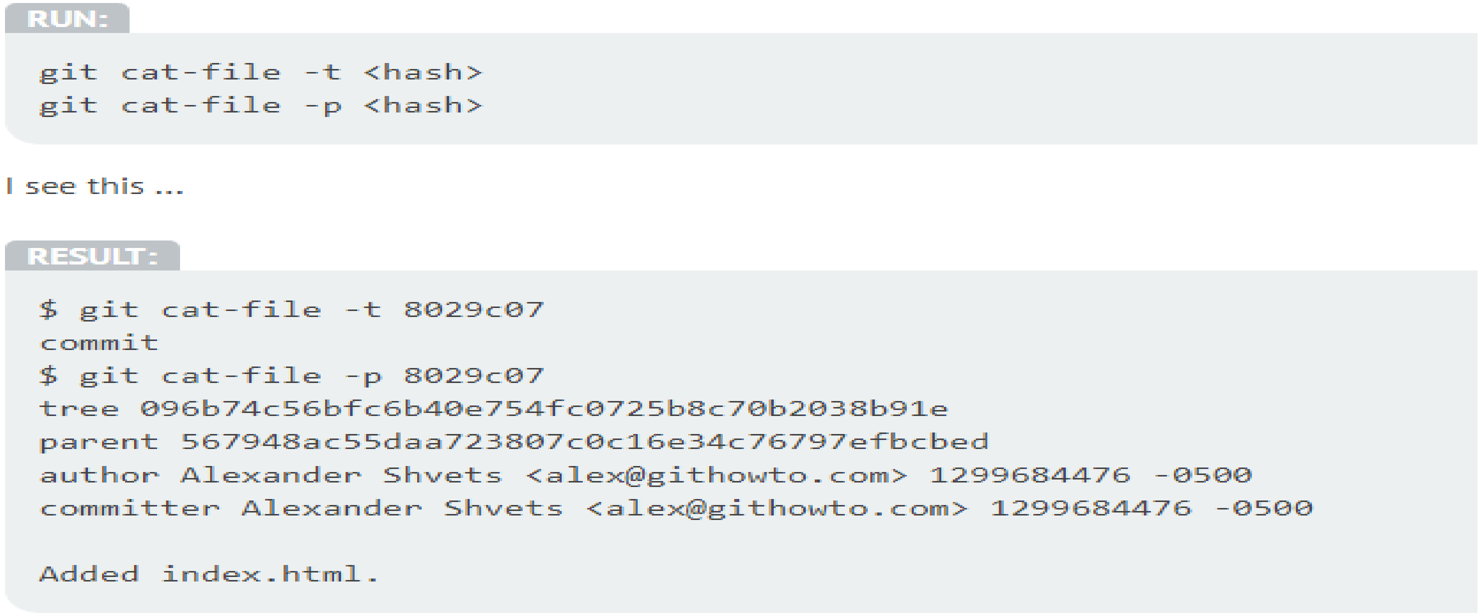


RESULT:

```
$ git hist --max-count=1
* 8029c07 2011-03-09 | Added index.html. (HEAD, master) [Alexander Shvets]
```

Display of the last commit

- With SHA1 hash of a commit, as above...



RUN:

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

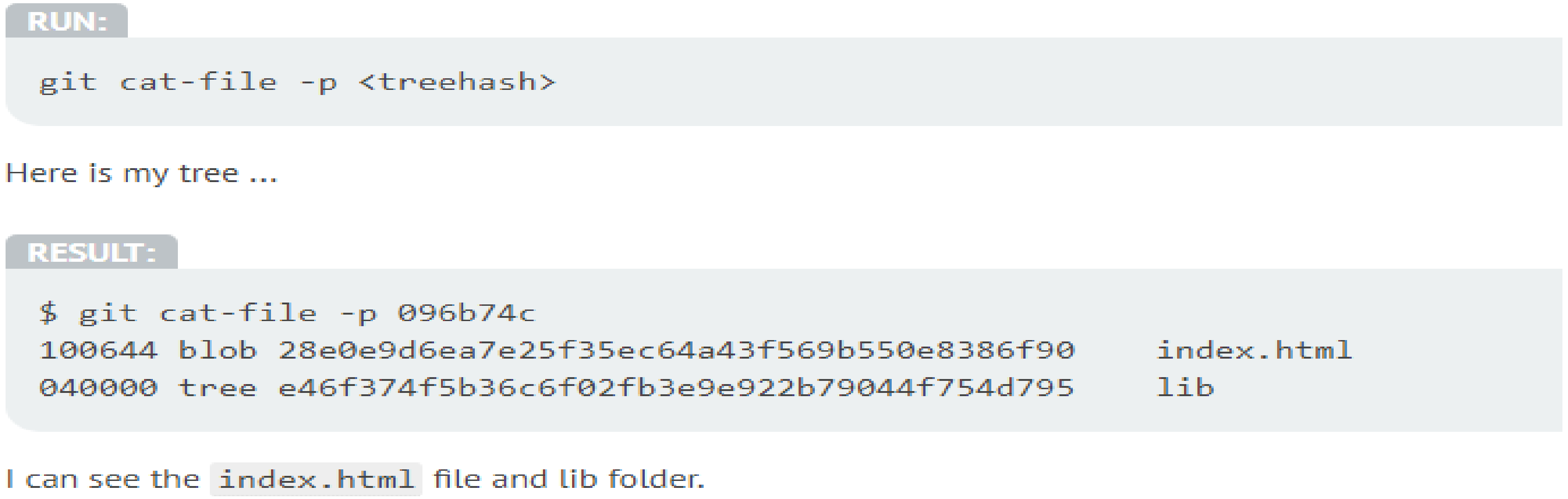
I see this ...

RESULT:

```
$ git cat-file -t 8029c07
commit
$ git cat-file -p 8029c07
tree 096b74c56bfc6b40e754fc0725b8c70b2038b91e
parent 567948ac55daa723807c0c16e34c76797efbcbed
author Alexander Shvets <alex@githowto.com> 1299684476 -0500
committer Alexander Shvets <alex@githowto.com> 1299684476 -0500

Added index.html.
```

Tree search



RUN:

```
git cat-file -p <treehash>
```

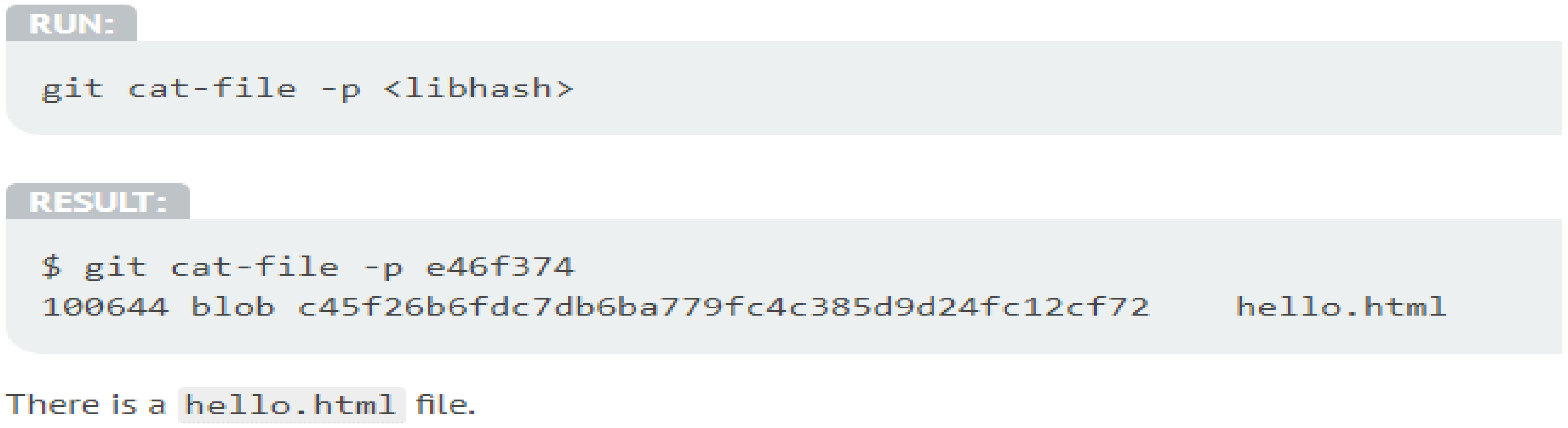
Here is my tree ...

RESULT:

```
$ git cat-file -p 096b74c
100644 blob 28e0e9d6ea7e25f35ec64a43f569b550e8386f90    index.html
040000 tree e46f374f5b36c6f02fb3e9e922b79044f754d795    lib
```

I can see the `index.html` file and lib folder.

Display lib directory



RUN:

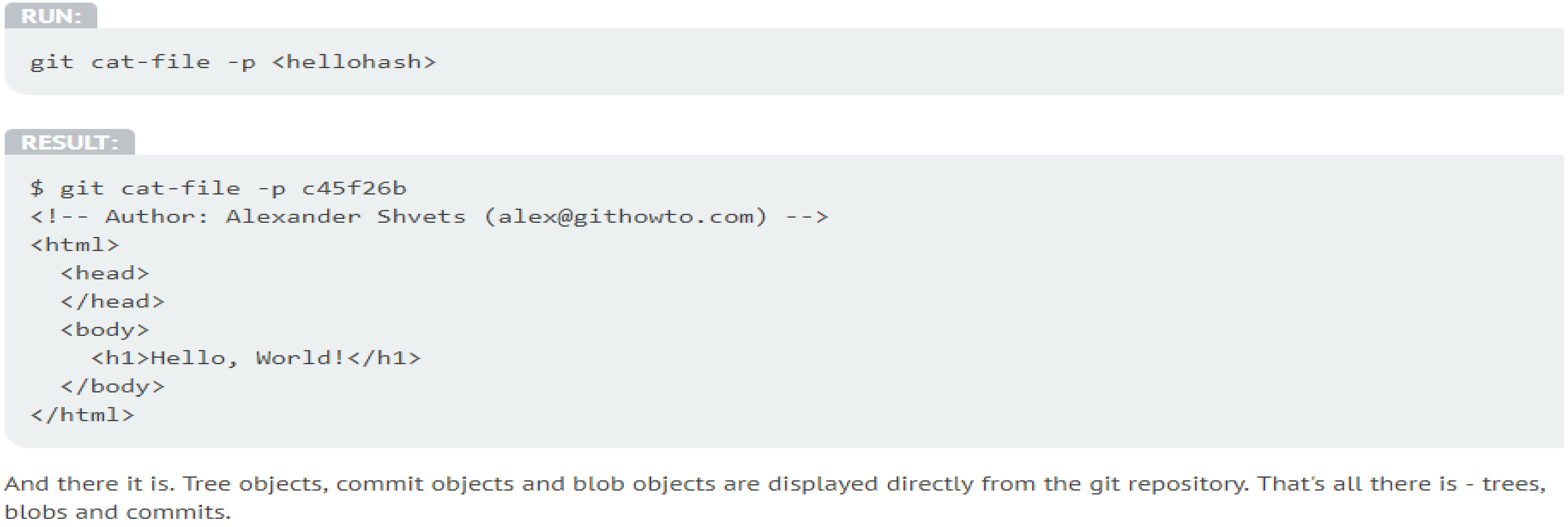
```
git cat-file -p <libhash>
```

RESULT:

```
$ git cat-file -p e46f374
100644 blob c45f26b6fdc7db6ba779fc4c385d9d24fc12cf72    hello.html
```

There is a `hello.html` file.

Display hello.html file



RUN:

```
git cat-file -p <hellohash>
```

RESULT:

```
$ git cat-file -p c45f26b
<!-- Author: Alexander Shvets (alex@githowto.com) -->
<html>
  <head>
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

And there it is. Tree objects, commit objects and blob objects are displayed directly from the git repository. That's all there is - trees, blobs and commits.

Reference

- https://githowto.com/git_internals_working_directly_with_git_objects
- https://git-scm.com/docs/git-cat-file

git-commit-tree

Create a new commit object

- Creates a new **commit object** based on the provided **tree object** and emits the new commit object id on stdout. The log message is read from the standard inpu

Recommended

- Part of Lab Already

- git-commit - Record changes to the repository Stores the current contents of the index in a new commit along with a log message from the user describing the changes.

- http://www.devopsschool.com/tutorial/git/Creating-git-Blobs-Trees-and-Commits.html

git filter branch

Rewrite branches and Remove a file from all git history

Why

- Password
- Wrong Copyright
- Etc

- Lets you rewrite Git revision history by rewriting the branches mentioned in the <rev-list options>, applying custom filters on each revision. Those filters can modify each tree (e.g. removing a file or running a perl rewrite on all files) or information about each commit. Otherwise, all information (including original commit times or merge information) will be preserved

Rewritten all the commits

- Once you remove a **file** from all the branch, all the commits will re-wrriten from the point where the **file** was introduced

Implication

- NOT INTO PUBLIC REPO

Referece

- https://stackoverflow.com/questions/3142419/how-can-i-move-a-directory-in-a-git-repo-for-all-commits

git hash-object

- http://www.devopsschool.com/tutorial/git/Creating-git-Blobs-Trees-and-Commits.html

Git ls-tree

- http://www.devopsschool.com/tutorial/git/Creating-git-Blobs-Trees-and-Commits.html

Git read-tree

- http://www.devopsschool.com/tutorial/git/Creating-git-Blobs-Trees-and-Commits.html

git reflog

Git keeps track of updates to the tip of branches using a mechanism called reference logs, or "reflogs." Many Git commands accept a parameter for specifying a reference or "ref", which is a pointer to a commit. Common examples include:

- git checkout
- git reset
- git merge

- Reflogs track when Git refs were updated in the local repository.
- By default, git reflog will output the reflog of the HEAD ref. HEADis a symbolic reference to the currently active branch.
- In addition to branch tip reflogs, a special reflog is maintained for the Git stash.
- Reflogs are stored in directories under the local repository's .git directory.
- git reflog directories can be found at .git/logs/refs/heads/., .git/logs/HEAD, and also .git/logs/refs/stash if the git stash has been used on the repo.

The most basic Reflog use case is invoking:

- git reflog

This is essentially a short cut that's equivalent to:

- git reflog show HEAD

Reflog references

- By default, git reflog will output the reflog of the HEAD ref. HEADis a symbolic reference to the currently active branch.
- Reflogs are available for other refs as well.
- The syntax to access a git ref is name@{qualifier}.
- In addition to HEAD refs, other branches, tags, remotes, and the Git stash can be referenced as well.

You can get a complete reflog of all refs by executing:

- git reflog show --all

To see the reflog for a specific branch pass that branch name to git reflog show

- git reflog show otherbranch

Reference

- https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog

git update-ref

- http://www.devopsschool.com/tutorial/git/Creating-git-Blobs-Trees-and-Commits.html

git write-tree

- http://www.devopsschool.com/tutorial/git/Creating-git-Blobs-Trees-and-Commits.html