Maintaining an audit and reversion trail without thousands of refs

Currently, *each* time a new version of an app is deployed, we create a ref:

```
refs/release/stg.www.starterview-2019-09-06T114410.954-0700-treesha-cb14dbb3528
7d81ac5ee19f06c872f5fd6b1aa56
```

As these refs proliferate, many operations in the Git repo become slower and slower.

We can record this information in a way that is at least as useful and which doesn't clutter up the ref list.

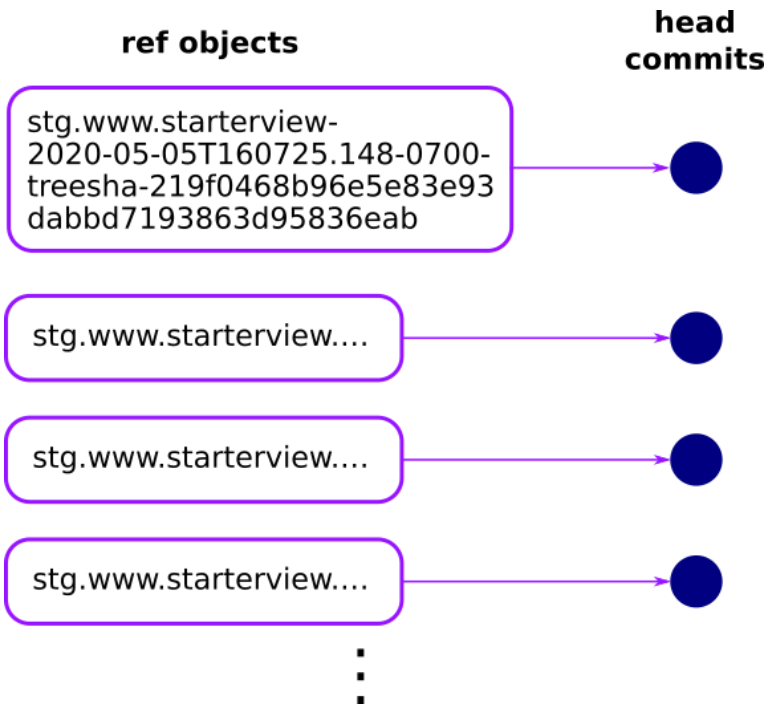
Current practice

At present, the release procedure creates a ref something like this:

- New release is created at commit *H*
- New tag object is created, pointing to *H*, something like

```
git tag -a refs/release/stg.www.starterview-2019-09-....  H
```

Every new tag object has a name in the ref database. This adds a name for every release. The structure is something like this:

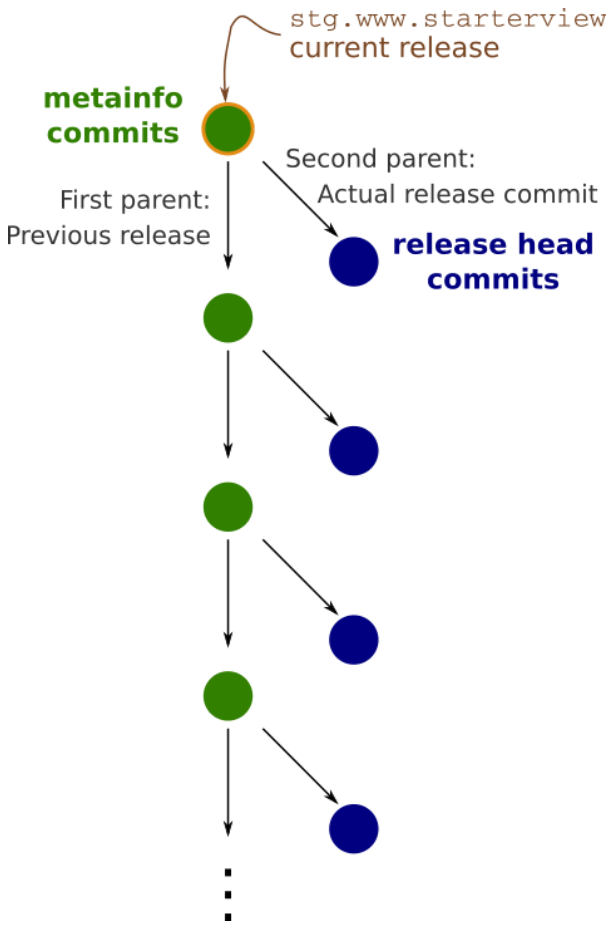


Alternative proposal

- Each (tier, app) pair is represented by a linked list of release commits
- The head of the list is the most recent release
- **One ref** points to the head of the list
- Older releases can be found by following the links back in time

Main point of this proposal: There is only **one ref** for each (tier, app) pair, no matter how many releases have been done.

It will look like this:



The **one ref** (which might or might not be an actual ref object) is named `stg.www.starterview`. That points not to the actual release commit, but to a commit that contains metainformation, such as the treesha. The metainformation could be stored in any of several ways. (See below.) The exact time of the release doesn't need to be stored explicitly because it is exactly the time at which the metainfo commit was created.

The metainfo commit has two parent commits. One is the actual release commit, the one that, in the current system, is the target of the named ref object. The other parent is the metainfo commit from the *previous* release.

The metainfo commits are chained together exactly the same way Git normally chains commit history, so the usual Git commands will work on the release history in reasonable and useful ways. In particular, the command `git log stg.www.starterview` will produce a listing of the release history.

Technical details

Creating this sort of two-parent commit is easy. Suppose the ref `stg.www.starterview` points to the current metainfo commit. And suppose we want to release new commit *H*. We do:

```
(Write some useful commit message into /tmp/message.$$.)
git commit-tree   -p refs/release/stg.www.starterview -p H  \
  H^{tree} < /tmp/message.$$
```

This creates a new metainfo commit with two parents, given by the arguments of the `-p` options. The first parent is the metainfo commit for the current release and the second is new commit *H*.

The `git commit-tree` command creates the new metainfo commit and writes its SHA on standard output. Say this SHA is *N*.

```
git push N:refs/release/stg.www.starterview
```

The release ref now points to the metainfo commit for the new release.

Usage

This will be easier to use than what we have now, because Git's command set was designed to manage a list of commits linked in this way; this is exactly how Git normally expects to represent history. This natural fit with Git's command set shows up in many places.

You can fetch a complete release history for any project with a single fetch:

```
git fetch origin refs/release/stg.www.starterview:HISTORY
```

After this, `git log HISTORY` will list the entire release history in reverse chronological order. All the usual git-log options make sense to filter or search this list. For example, if we want to find the last release before a certain date, we can use:

```
git log -1 --until=date HISTORY
```

If we wanted a forward-chronological list of release commit SHAs with their release dates, we could use something like `git log -r --format="%P %cd"`.

At all times:

- The current release commit is accessible at `HISTORY^2`.
- The metainfo commit from *n* releases ago is at `HISTORY~n`.
- The release commit from *n* releases ago is at `HISTORY~n^2`.

These are easy to find. We can locate a previous release for rollback. When we roll back, we can roll back to the exact same commit as before: the new metainfo commit and the old one can share the exact same release commit object.

We can see the differences between the last two releases, without knowing their exact dates, with something like `git diff HISTORY~1^2 HISTORY^2`.

The metainfo commit is a full commit with a full set of Git commit metadata, such as a creation date. It has a commit message where we can record any unstructured text that we want. We could also require the commit message to be in TOML format, or we could use the tooling provided by Git to store semi-structured information in the commit message; see `git-interpret-trailers`.

If the built-in commit structure is not big enough for everything we want to record, the metainfo commit also has an associated tree where we could record any amount of structured information. In the example above I had its tree be identical to the release commit's tree, which might be convenient. (That way, if you wanted to see the files that were released, you could look in either the release commit or the metainfo commit; this would make some of the commands simpler.) But the tree doesn't have to be the same. We could also add metainformation files to this tree, or we could commit an entirely unrelated tree with nothing in it other than structured metainformation.

Performance?

Jeremy Donahue asks about the performance issues of tracing out a linked list of commands. But this is exactly the same mechanism that Git normally uses to record history. When we do `git log`, Git is tracing a linked list of commits in exactly the way suggested here. If any operation in Git is going to be fast, it will be this one.

Git currently takes a few seconds to trace the 275,000-commit history of our monorepo. Tracing the similarly-structured history of a few hundred release commits is not going to be a significant issue.