Understanding Git: Repositories

In This Section

Repository Contents

The purpose of Git is to manage a project, or a set of files, as they change over time. Git stores this information in a data structure called a repository.

A git **repository** contains, among other things, the following:

- A set of **commit objects**.
- A set of references to commit objects, called **heads**.

The Git repository is stored in the same directory as the project itself, in a subdirectory called `.git`. Note differences from central-repository systems like CVS or Subversion:

- There is only one `.git` directory, in the root directory of the project.
- The repository is stored in files alongside the project. There is no central server repository.

Commit Objects

A **commit object** contains three things:

- A set of **files**, reflecting the state of a project at a given point in time.
- References to **parent commit objects**.
- An **SHA1 name**, a 40-character string that uniquely identifies the commit object. The name is composed of a hash of relevant aspects of the commit, so identical commits will always have the same name.

The parent commit objects are those commits that were edited to produce the subsequent state of the project. Generally a commit object will have one parent commit, because one generally takes a project in a given state, makes a few changes, and saves the new state of the project. The section below on merges explains how a commit object could have two or more parents.

A project always has one commit object with no parents. This is the first commit made to the project repository.

Based on the above, you can visualize a repository as a directed acyclic graph of commit objects, with pointers to parent commits always pointing backwards in time, ultimately to the first commit. Starting from any commit, you can walk along the tree by parent commits to see the history of changes that led to that commit.

The idea behind Git is that version control is all about manipulating this graph of commits. Whenever you want to perform some operation to query or manipulate the repository, you should be thinking, “how do I want to query or manipulate the graph of commits?”

Heads

A **head** is simply a reference to a commit object. Each head has a name. By default, there is a head in every repository called *master*. A repository can contain any number of heads. At any given time, one head is selected as the “current head.” This head is aliased to *HEAD*, always in capitals.

Note this difference: a “head” (lowercase) refers to any one of the named heads in the repository; “*HEAD*” (uppercase) refers exclusively to the currently active head. This distinction is used frequently in Git documentation. I also use the convention that names of heads, including *HEAD*, are set in italics.

A Simple Repository

To create a repository, create a directory for the project if it doesn’t exist, enter it, and run the command `git init`. The directory does not need to be empty.

```
mkdir [project]
cd [project]
git init
```

This will create a `.git` directory in the `[project]` directory.

To create a commit, you need to do two things:

1. Tell Git which files to include in the commit, with `git add`. If a file has not changed since the previous commit (the “parent” commit), Git will automatically include it in the commit you are about to perform. Thus, you only need to add files that you have added or modified. Note that it adds directories recursively, so `git add .` will add everything that has changed.
2. Call `git commit` to create the commit object. The new commit object will have the current *HEAD* as its parent (and then, after the commit is complete, *HEAD* will point to the new commit object).

As a shortcut, `git commit -a` will automatically add all modified files (but not new ones).

Note that if you modify a file but do not add it, then Git will include the previous version (before modifications) to the commit. The modified file will remain in place.

Say you create three commits this way. Your repository will look like this:

```
---->  time  ----->

(A) <-- (B) <-- (C)
                 ^
                 |
               master
                 ^
                 |
                HEAD
```

where (A), (B), and (C) are the first, second, and third commits, respectively.

Other commands that are useful at this point:

- `git log` shows a log of all commits starting from *HEAD* back to the initial commit. (It can do more than that, of course.)
- `git status` shows which files have changed between the current project state and *HEAD*. Files are put in one of three categories: new files that haven't been added (with `git add`), modified files that haven't been added, and files that have been added.
- `git diff` shows the diff between *HEAD* and the current project state. With the `--cached` option it compares added files against *HEAD*; otherwise it compares files not yet added.*
- `git mv` and `git rm` mark files to be moved (rename) and removed, respectively, much like `git add`.

*More precisely, `git diff` compares the staging area, that is, *HEAD* as modified by all added files, against the current state of all added files. Thus, if you add some but not all changed files and then run `git diff`, you will see only files that are changed but not added.

My personal workflow usually looks like:

1. Do some programming.
2. `git status` to see what files I changed.
3. `git diff [file]` to see exactly what I modified.
4. `git commit -a -m [message]` to commit.

Referring to a Commit

Now that you've created commits, how do you refer to a specific commit? Git provides many ways to do so. Here are a few:

- By its SHA1 name, which you can get from `git log`.
- By the first few characters of its SHA1 name.
- By a head. For example, `HEAD` refers to the commit object referenced by *HEAD*. You can also use

the name, such as `master`.

- Relative to a commit. Putting a caret (^) after a commit name retrieves the parent of that commit. For example, *HEAD^* is the parent of the current head commit.

Go on to the next page: Branching