Understanding and Working with Submodules in Git

Submodules in Git are really just standard Git repositories. No fancy innovation, just the same Git repositories that we all know so well by now. This is also part of the power of submodules: they're so robust and straightforward because they are so "boring" (from a technological point of view) and field-tested.

The only thing that makes a Git repository a submodule is that it's placed *inside* another, *parent* Git repository.

Other than that, a Git submodule remains a fully functional repository: you can perform all the actions that you already know from your "normal" Git work --- from modifying files, all the way to committing, pulling and pushing. Everything's possible in a submodule.

Adding a Submodule

Let's take the classic example and say we'd like to add a third-party library to our project. Before we go get any code, it makes sense to create a separate folder where things like these can have a home:

```
$ cd ~/work
$ mkdir subproject_demo
$ cd subproject_demo
$ git init
$ mkdir lib
$ cd lib
```

Now we're ready to pump some third-party code into our project --- but in an orderly fashion, using submodules. Let's say we need a little "timezone converter" JavaScript library:

```
$ git submodule add https://github.com/spencermountain/spacetime.git
```

When we run this command, Git starts cloning the repository into our project, as a submodule:

```
Cloning into 'subproject_demo/lib/spacetime'...
remote: Enumerating objects: 7768, done.
remote: Counting objects: 100% (1066/1066), done.
remote: Compressing objects: 100% (445/445), done.
remote: Total 7768 (delta 615), reused 975 (delta 588), pack-reused 6702
Receiving objects: 100% (7768/7768), 4.02 MiB | 7.78 MiB/s, done.
Resolving deltas: 100% (5159/5159), done.
```

Let's see what else has happened: a new `.gitmodules` file has been created in the root folder of our main project. Here's what it contains:

```
[submodule "lib/spacetime"]
  path = lib/spacetime
  url = https://github.com/spencermountain/spacetime.git
```

This `.gitmodules` file is one of multiple places where Git keeps track of the submodules in our project. Another one is `.git/config`, which now ends like this:

```
[submodule "lib/spacetime"]
  url = https://github.com/spencermountain/spacetime.git
  active = true
```

And finally, Git also keeps a copy of each submodule's `.git` repository in an internal `.git/modules` folder.

All of these are technical details you don't have to remember. However, it probably helps you to understand that the internal maintenance of Git submodules is quite complex. That's why it's important to take one thing away: **don't mess with Git submodule configuration by hand!** If you want to move, delete, or otherwise manipulate a submodule, please do yourself a favor and do *not* try this manually.

Let's have a look at the status of our main project, now that we've added the submodule:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  new file:   .gitmodules
  new file:   lib/spacetime
```

As you can see, Git regards adding a submodule as a change like any other. Accordingly, we have to commit this change like any other:

```
$ git commit -m "Add timezone converter library as a submodule"
```

Checking Out Revisions

In a "normal" Git repository, we usually check out branches. By using `git checkout <branchname>` or the newer `git switch <branchname>`, we're telling Git what our currently active branch should be. When new commits are made on this branch, the HEAD pointer is *automatically moved* to the very latest commit. This is important to understand --- because Git submodules work differently!

In a submodule, we're always checking out a specific revision --- not a branch! Even when you're executing a command like `git checkout main` in a submodule, in the background, the currently latest *commit* on that branch is noted --- not the branch itself.

If you want to find out what revision your submodules are using, you can request this information in your main project:

```
$ git submodule status
   ea703a7d557efd90ccae894db96368d750be93b6 lib/spacetime (6.16.3)
```

This returns the currently checked out revision of our `lib/spacetime` submodule. And it also lets us know that this revision is a tag, named "6.16.3". It's pretty common to use tags heavily when working with submodules in Git.

Let's say you wanted your submodule to use an *older* version, which was tagged "6.14.0". First, we have to change directories so that our Git command will be executed in the context of the submodule, not our main project. Then, we can simply run `git checkout` with the tag name:

```
$ cd lib/spacetime/
$ git checkout 6.14.0
Previous HEAD position was ea703a7 Merge pull request #301 from spencermountain/dev
HEAD is now at 7f78d50 Merge pull request #268 from spencermountain/dev
```

If we now go back into our main project and execute `git submodule status` again, we'll see our checkout reflected:

```
$ cd ../..
$ git submodule status
+7f78d50156ae1205aa50675ddede81a61a45fade lib/spacetime (6.14.0)
```

Take a close look at the output, though: the little `+` symbol in front of that SHA-1 hash tells us that the submodule is at a different revision than is currently stored in the parent repository. As we just changed the checked out revision, this looks correct.

Calling `git status` in our main project now informs us about this fact, too:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
  modified:   lib/spacetime (new commits)
```

You can see that Git considers moving a submodule's pointer as a change like any other: we have to commit it to the repository if we want it to be stored:

```
$ git commit -m "Changed checked out revision in submodule"
$ git status
```

Working with Submodules in Git

We've covered the basic building blocks of working with Git submodules. Other workflows are really quite standard!

Checking for new changes in a submodule, for example, works like in any other Git repository: you run a `git fetch` command inside the submodule repository, possibly followed by something like `git pull origin main` if you want to indeed make use of the updates.

Making changes in a submodule might also be a use case for you, especially if you manage the library code yourself (because it's an internal library, not from a third party). You can work with the submodule like with any other Git repository: you can make changes, commit them, push them, and so on.