7.11 Git Tools - Submodules

**Submodules** https://devtopia.esri.com/nobi4775/LearnGit.git

It often happens that while working on one project, you need to use another project from within it. Perhaps it’s a library that a third party developed or that you’re developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other.

Git addresses this issue using **submodules**. Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

**Starting with Submodules**

We’ll walk through developing a simple project that has been split up into a main project and a few sub-projects.

Let’s start by adding an existing Git repository as a submodule of the repository that we’re working on. To add a **new** submodule you use the git submodule add command with the absolute or relative URL of the project you would like to start tracking. In this example, we’ll add a library called “DbConnector”.

**$ git submodule add https://github.com/chaconinc/DbConnector**

Cloning into 'DbConnector'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

By default, submodules will add the subproject into a directory named the same as the repository, in this case “DbConnector”. You can add a different path at the end of the command if you want it to go elsewhere.

If you run git status at this point, you’ll notice a few things.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: .gitmodules

new file: DbConnector

First you should notice the new .gitmodules file. This is a configuration file that stores the mapping between the project’s URL and the local subdirectory you’ve pulled it into:

[submodule "DbConnector"]

path = DbConnector

url = https://github.com/chaconinc/DbConnector

If you have multiple submodules, you’ll have multiple entries in this file. It’s important to note that this file is version-controlled with your other files, like your .gitignore file. It’s pushed and pulled with the rest of your project. This is how other people who clone this project know where to get the submodule projects from.

|  |  |
| --- | --- |
| Note | Since the URL in the .gitmodules file is what other people will first try to clone/fetch from, make sure to use a URL that they can access if possible. For example, if you use a different URL to push to than others would to pull from, use the one that others have access to. You can overwrite this value locally with git config submodule.DbConnector.url PRIVATE\_URL for your own use. When applicable, a relative URL can be helpful. |

The other listing in the git status output is the project folder entry. If you run git diff on that, you see something interesting:

$ git diff --cached DbConnector

diff --git a/DbConnector b/DbConnector

new file mode 160000

index 0000000..c3f01dc

--- /dev/null

+++ b/DbConnector

@@ -0,0 +1 @@

+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc

Although DbConnector is a subdirectory in your working directory, Git sees it as a submodule and doesn’t track its contents when you’re not in that directory. Instead, Git sees it as a particular commit from that repository. If you want a little nicer diff output, you can pass the --submodule option to git diff.

$ git diff --cached --submodule

diff --git a/.gitmodules b/.gitmodules

new file mode 100644

index 0000000..71fc376

--- /dev/null

+++ b/.gitmodules

@@ -0,0 +1,3 @@

+[submodule "DbConnector"]

+ path = DbConnector

+ url = https://github.com/chaconinc/DbConnector

Submodule DbConnector 0000000...c3f01dc (new submodule)

When you commit, you see something like this:

$ git commit -am 'added DbConnector module'

[master fb9093c] added DbConnector module

2 files changed, 4 insertions(+)

create mode 100644 .gitmodules

create mode 160000 DbConnector

Notice the 160000 mode for the DbConnector entry. That is a special mode in Git that basically means you’re recording a commit as a directory entry rather than a subdirectory or a file.

Lastly, push these changes:

$ git push origin master

**Cloning a Project with Submodules**

Here we’ll clone a project with a submodule in it. When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet:

$ git clone https://github.com/chaconinc/MainProject

Cloning into 'MainProject'...

remote: Counting objects: 14, done.

remote: Compressing objects: 100% (13/13), done.

remote: Total 14 (delta 1), reused 13 (delta 0)

Unpacking objects: 100% (14/14), done.

Checking connectivity... done.

$ cd MainProject

$ ls -la

total 16

drwxr-xr-x 9 schacon staff 306 Sep 17 15:21 .

drwxr-xr-x 7 schacon staff 238 Sep 17 15:21 ..

drwxr-xr-x 13 schacon staff 442 Sep 17 15:21 .git

-rw-r--r-- 1 schacon staff 92 Sep 17 15:21 .gitmodules

drwxr-xr-x 2 schacon staff 68 Sep 17 15:21 DbConnector

-rw-r--r-- 1 schacon staff 756 Sep 17 15:21 Makefile

drwxr-xr-x 3 schacon staff 102 Sep 17 15:21 includes

drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 scripts

drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 src

$ cd DbConnector/

$ ls

$

The DbConnector directory is there, but empty. You must run two commands: git submodule init to initialize your local configuration file, and git submodule update to fetch all the data from that project and check out the appropriate commit listed in your superproject:

$ git submodule init

Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'

$ git submodule update

Cloning into 'DbConnector'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'

Now your DbConnector subdirectory is at the exact state it was in when you committed earlier.

There is another way to do this which is a little simpler, however. If you pass --recurse-submodules to the git clone command, it will automatically initialize and update each submodule in the repository.

$ git clone --recurse-submodules https://github.com/chaconinc/MainProject

Cloning into 'MainProject'...

remote: Counting objects: 14, done.

remote: Compressing objects: 100% (13/13), done.

remote: Total 14 (delta 1), reused 13 (delta 0)

Unpacking objects: 100% (14/14), done.

Checking connectivity... done.

Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'

Cloning into 'DbConnector'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'

**Working on a Project with Submodules**

Now we have a copy of a project with submodules in it and will collaborate with our teammates on both the main project and the submodule project.

**Pulling in Upstream Changes**

The simplest model of using submodules in a project would be if you were simply consuming a subproject and wanted to get updates from it from time to time but were not actually modifying anything in your checkout. Let’s walk through a simple example there.

If you want to check for new work in a submodule, you can go into the directory and run git fetch and git merge the upstream branch to update the local code.

$ git fetch

From https://github.com/chaconinc/DbConnector

c3f01dc..d0354fc master -> origin/master

$ git merge origin/master

Updating c3f01dc..d0354fc

Fast-forward

scripts/connect.sh | 1 +

src/db.c | 1 +

2 files changed, 2 insertions(+)

Now if you go back into the main project and run git diff --submodule you can see that the submodule was updated and get a list of commits that were added to it. If you don’t want to type --submodule every time you run git diff, you can set it as the default format by setting the diff.submodule config value to “log”.

$ git config --global diff.submodule log

$ git diff

Submodule DbConnector c3f01dc..d0354fc:

> more efficient db routine

> better connection routine

If you commit at this point then you will lock the submodule into having the new code when other people update.

There is an easier way to do this as well, if you prefer to not manually fetch and merge in the subdirectory. If you run git submodule update --remote, Git will go into your submodules and fetch and update for you.

$ git submodule update --remote DbConnector

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 4 (delta 2), reused 4 (delta 2)

Unpacking objects: 100% (4/4), done.

From https://github.com/chaconinc/DbConnector

3f19983..d0354fc master -> origin/master

Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'

This command will by default assume that you want to update the checkout to the master branch of the submodule repository. You can, however, set this to something different if you want. For example, if you want to have the DbConnector submodule track that repository’s “stable” branch, you can set it in either your .gitmodules file (so everyone else also tracks it), or just in your local .git/config file. Let’s set it in the .gitmodules file:

$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 4 (delta 2), reused 4 (delta 2)

Unpacking objects: 100% (4/4), done.

From https://github.com/chaconinc/DbConnector

27cf5d3..c87d55d stable -> origin/stable

Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'

If you leave off the -f .gitmodules it will only make the change for you, but it probably makes more sense to track that information with the repository so everyone else does as well.

When we run git status at this point, Git will show us that we have “new commits” on the submodule.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .gitmodules

modified: DbConnector (new commits)

no changes added to commit (use "git add" and/or "git commit -a")

If you set the configuration setting status.submodulesummary, Git will also show you a short summary of changes to your submodules:

$ git config status.submodulesummary 1

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .gitmodules

modified: DbConnector (new commits)

Submodules changed but not updated:

\* DbConnector c3f01dc...c87d55d (4):

> catch non-null terminated lines

At this point if you run git diff we can see both that we have modified our .gitmodules file and also that there are a number of commits that we’ve pulled down and are ready to commit to our submodule project.

$ git diff

diff --git a/.gitmodules b/.gitmodules

index 6fc0b3d..fd1cc29 100644

--- a/.gitmodules

+++ b/.gitmodules

@@ -1,3 +1,4 @@

[submodule "DbConnector"]

path = DbConnector

url = https://github.com/chaconinc/DbConnector

+ branch = stable

Submodule DbConnector c3f01dc..c87d55d:

> catch non-null terminated lines

> more robust error handling

> more efficient db routine

> better connection routine

This is pretty cool as we can actually see the log of commits that we’re about to commit to in our submodule. Once committed, you can see this information after the fact as well when you run

git log -p.

$ git log -p --submodule

commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae

Author: Scott Chacon <schacon@gmail.com>

Date: Wed Sep 17 16:37:02 2014 +0200

updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules

index 6fc0b3d..fd1cc29 100644

--- a/.gitmodules

+++ b/.gitmodules

@@ -1,3 +1,4 @@

[submodule "DbConnector"]

path = DbConnector

url = https://github.com/chaconinc/DbConnector

+ branch = stable

Submodule DbConnector c3f01dc..c87d55d:

> catch non-null terminated lines

> more robust error handling

> more efficient db routine

> better connection routine

Git will by default try to update **all** of your submodules when you run git submodule update --remoteso if you have a lot of them, you may want to pass the name of just the submodule you want to try to update.

**Working on a Submodule**

It’s quite likely that if you’re using submodules, you’re doing so because you really want to work on the code in the submodule at the same time as you’re working on the code in the main project (or across several submodules). Otherwise you would probably instead be using a simpler dependency management system (such as Maven or Rubygems).

So now let’s go through an example of making changes to the submodule at the same time as the main project and committing and publishing those changes at the same time.

So far, when we’ve run the git submodule update command to fetch changes from the submodule repositories, Git would get the changes and update the files in the subdirectory but will leave the sub-repository in what’s called a “detached HEAD” state. This means that there is no local working branch (like “master”, for example) tracking changes. With no working branch tracking changes, that means even if you commit changes to the submodule, those changes will quite possibly be lost the next time you run git submodule update. You have to do some extra steps if you want changes in a submodule to be tracked.

In order to set up your submodule to be easier to go in and hack on, you need to do two things. You need to go into each submodule and check out a branch to work on. Then you need to tell Git what to do if you have made changes and then git submodule update --remote pulls in new work from upstream. The options are that you can merge them into your local work, or you can try to rebase your local work on top of the new changes.

First of all, let’s go into our submodule directory and check out a branch.

$ git checkout stable

Switched to branch 'stable'

Let’s try it with the “merge” option. To specify it manually, we can just add the --merge option to our update call. Here we’ll see that there was a change on the server for this submodule and it gets merged in.

$ git submodule update --remote --merge

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 4 (delta 2), reused 4 (delta 2)

Unpacking objects: 100% (4/4), done.

From https://github.com/chaconinc/DbConnector

c87d55d..92c7337 stable -> origin/stable

Updating c87d55d..92c7337

Fast-forward

src/main.c | 1 +

1 file changed, 1 insertion(+)

Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'

If we go into the DbConnector directory, we have the new changes already merged into our local stablebranch. Now let’s see what happens when we make our own local change to the library and someone else pushes another change upstream at the same time.

$ cd DbConnector/

$ vim src/db.c

$ git commit -am 'unicode support'

[stable f906e16] unicode support

1 file changed, 1 insertion(+)

Now if we update our submodule we can see what happens when we have made a local change and upstream also has a change we need to incorporate.

$ git submodule update --remote --rebase

First, rewinding head to replay your work on top of it...

Applying: unicode support

Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'

If you forget the --rebase or --merge, Git will just update the submodule to whatever is on the server and reset your project to a detached HEAD state.

$ git submodule update --remote

Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'

If this happens, don’t worry, you can simply go back into the directory and check out your branch again (which will still contain your work) and merge or rebase origin/stable (or whatever remote branch you want) manually.

If you haven’t committed your changes in your submodule and you run a submodule update that would cause issues, Git will fetch the changes but not overwrite unsaved work in your submodule directory.

$ git submodule update --remote

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 4 (delta 0), reused 4 (delta 0)

Unpacking objects: 100% (4/4), done.

From https://github.com/chaconinc/DbConnector

5d60ef9..c75e92a stable -> origin/stable

error: Your local changes to the following files would be overwritten by checkout:

scripts/setup.sh

Please, commit your changes or stash them before you can switch branches.

Aborting

Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'

If you made changes that conflict with something changed upstream, Git will let you know when you run the update.

$ git submodule update --remote --merge

Auto-merging scripts/setup.sh

CONFLICT (content): Merge conflict in scripts/setup.sh

Recorded preimage for 'scripts/setup.sh'

Automatic merge failed; fix conflicts and then commit the result.

Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'

You can go into the submodule directory and fix the conflict just as you normally would.

**Publishing Submodule Changes**

Now we have some changes in our submodule directory. Some of these were brought in from upstream by our updates and others were made locally and aren’t available to anyone else yet as we haven’t pushed them yet.

$ git diff

Submodule DbConnector c87d55d..82d2ad3:

> Merge from origin/stable

> updated setup script

> unicode support

> remove unnecessary method

> add new option for conn pooling

If we commit in the main project and push it up without pushing the submodule changes up as well, other people who try to check out our changes are going to be in trouble since they will have no way to get the submodule changes that are depended on. Those changes will only exist on our local copy.

In order to make sure this doesn’t happen, you can ask Git to check that all your submodules have been pushed properly before pushing the main project. The git push command takes the --recurse-submodules argument which can be set to either “check” or “on-demand”. The “check” option will make push simply fail if any of the committed submodule changes haven’t been pushed.

$ git push --recurse-submodules=check

The following submodule paths contain changes that can

not be found on any remote:

DbConnector

Please try

git push --recurse-submodules=on-demand

or cd to the path and use

git push

to push them to a remote.

As you can see, it also gives us some helpful advice on what we might want to do next. The simple option is to go into each submodule and manually push to the remotes to make sure they’re externally available and then try this push again. If you want the check behavior to happen for all pushes, you can make this behavior the default by doing git config push.recurseSubmodules check.

The other option is to use the “on-demand” value, which will try to do this for you.

$ git push --recurse-submodules=on-demand

Pushing submodule 'DbConnector'

Counting objects: 9, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (8/8), done.

Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.

Total 9 (delta 3), reused 0 (delta 0)

To https://github.com/chaconinc/DbConnector

c75e92a..82d2ad3 stable -> stable

Counting objects: 2, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.

Total 2 (delta 1), reused 0 (delta 0)

To https://github.com/chaconinc/MainProject

3d6d338..9a377d1 master -> master

As you can see there, Git went into the DbConnector module and pushed it before pushing the main project. If that submodule push fails for some reason, the main project push will also fail. You can make this behavior the default by doing git config push.recurseSubmodules on-demand.

**Merging Submodule Changes**

If you change a submodule reference at the same time as someone else, you may run into some problems. That is, if the submodule histories have diverged and are committed to diverging branches in a superproject, it may take a bit of work for you to fix.

If one of the commits is a direct ancestor of the other (a fast-forward merge), then Git will simply choose the latter for the merge, so that works fine.

Git will not attempt even a trivial merge for you, however. If the submodule commits diverge and need to be merged, you will get something that looks like this:

$ git pull

remote: Counting objects: 2, done.

remote: Compressing objects: 100% (1/1), done.

remote: Total 2 (delta 1), reused 2 (delta 1)

Unpacking objects: 100% (2/2), done.

From https://github.com/chaconinc/MainProject

9a377d1..eb974f8 master -> origin/master

Fetching submodule DbConnector

warning: Failed to merge submodule DbConnector (merge following commits not found)

Auto-merging DbConnector

CONFLICT (submodule): Merge conflict in DbConnector

Automatic merge failed; fix conflicts and then commit the result.

So basically what has happened here is that Git has figured out that the two branches record points in the submodule’s history that are divergent and need to be merged. It explains it as “merge following commits not found”, which is confusing but we’ll explain why that is in a bit.

To solve the problem, you need to figure out what state the submodule should be in. Strangely, Git doesn’t really give you much information to help out here, not even the SHA-1s of the commits of both sides of the history. Fortunately, it’s simple to figure out. If you run git diff you can get the SHA-1s of the commits recorded in both branches you were trying to merge.

$ git diff

diff --cc DbConnector

index eb41d76,c771610..0000000

--- a/DbConnector

+++ b/DbConnector

So, in this case, eb41d76 is the commit in our submodule that **we** had and c771610 is the commit that upstream had. If we go into our submodule directory, it should already be on eb41d76 as the merge would not have touched it. If for whatever reason it’s not, you can simply create and checkout a branch pointing to it.

What is important is the SHA-1 of the commit from the other side. This is what you’ll have to merge in and resolve. You can either just try the merge with the SHA-1 directly, or you can create a branch for it and then try to merge that in. We would suggest the latter, even if only to make a nicer merge commit message.

So, we will go into our submodule directory, create a branch based on that second SHA-1 from git diffand manually merge.

$ cd DbConnector

$ git rev-parse HEAD

eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610

(DbConnector) $ git merge try-merge

Auto-merging src/main.c

CONFLICT (content): Merge conflict in src/main.c

Recorded preimage for 'src/main.c'

Automatic merge failed; fix conflicts and then commit the result.

We got an actual merge conflict here, so if we resolve that and commit it, then we can simply update the main project with the result.

$ vim src/main.c **(1)**

$ git add src/main.c

$ git commit -am 'merged our changes'

Recorded resolution for 'src/main.c'.

[master 9fd905e] merged our changes

$ cd .. **(2)**

$ git diff **(3)**

diff --cc DbConnector

index eb41d76,c771610..0000000

--- a/DbConnector

+++ b/DbConnector

@@@ -1,1 -1,1 +1,1 @@@

- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135

-Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d

++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a

$ git add DbConnector **(4)**

$ git commit -m "Merge Tom's Changes" **(5)**

[master 10d2c60] Merge Tom's Changes

1. First we resolve the conflict
2. Then we go back to the main project directory
3. We can check the SHA-1s again
4. Resolve the conflicted submodule entry
5. Commit our merge

It can be a bit confusing, but it’s really not very hard. Interestingly, there is another case that Git handles. If a merge commit exists in the submodule directory that contains **both** commits in its history, Git will suggest it to you as a possible solution. It sees that at some point in the submodule project, someone merged branches containing these two commits, so maybe you’ll want that one.

This is why the error message from before was “merge following commits not found”, because it could not do **this**. It’s confusing because who would expect it to **try** to do this?

If it does find a single acceptable merge commit, you’ll see something like this:

$ git merge origin/master

warning: Failed to merge submodule DbConnector (not fast-forward)

Found a possible merge resolution for the submodule:

9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes

If this is correct simply add it to the index for example

by using:

git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a "DbConnector"

which will accept this suggestion.

Auto-merging DbConnector

CONFLICT (submodule): Merge conflict in DbConnector

Automatic merge failed; fix conflicts and then commit the result.

The suggested command Git is providing will update the index as though you had run git add (which clears the conflict), then commit. You probably shouldn’t do this though. You can just as easily go into the submodule directory, see what the difference is, fast-forward to this commit, test it properly, and then commit it.

$ cd DbConnector/

$ git merge 9fd905e

Updating eb41d76..9fd905e

Fast-forward

$ cd ..

$ git add DbConnector

$ git commit -am 'Fast forwarded to a common submodule child'

This accomplishes the same thing, but at least this way you can verify that it works and you have the code in your submodule directory when you’re done.

**Submodule Tips**

There are a few things you can do to make working with submodules a little easier.

**Submodule Foreach**

There is a foreach submodule command to run some arbitrary command in each submodule. This can be really helpful if you have a number of submodules in the same project.

For example, let’s say we want to start a new feature or do a bugfix and we have work going on in several submodules. We can easily stash all the work in all our submodules.

$ git submodule foreach 'git stash'

Entering 'CryptoLibrary'

No local changes to save

Entering 'DbConnector'

Saved working directory and index state WIP on stable: 82d2ad3 Merge from origin/stable

HEAD is now at 82d2ad3 Merge from origin/stable

Then we can create a new branch and switch to it in all our submodules.

$ git submodule foreach 'git checkout -b featureA'

Entering 'CryptoLibrary'

Switched to a new branch 'featureA'

Entering 'DbConnector'

Switched to a new branch 'featureA'

You get the idea. One really useful thing you can do is produce a nice unified diff of what is changed in your main project and all your subprojects as well.

$ git diff; git submodule foreach 'git diff'

Submodule DbConnector contains modified content

diff --git a/src/main.c b/src/main.c

index 210f1ae..1f0acdc 100644

--- a/src/main.c

+++ b/src/main.c

@@ -245,6 +245,8 @@ static int handle\_alias(int \*argcp, const char \*\*\*argv)

commit\_pager\_choice();

+ url = url\_decode(url\_orig);

/\* build alias\_argv \*/

alias\_argv = xmalloc(sizeof(\*alias\_argv) \* (argc + 1));

alias\_argv[0] = alias\_string + 1;

Entering 'DbConnector'

diff --git a/src/db.c b/src/db.c

index 1aaefb6..5297645 100644

--- a/src/db.c

+++ b/src/db.c

@@ -93,6 +93,11 @@ char \*url\_decode\_mem(const char \*url, int len)

return url\_decode\_internal(&url, len, NULL, &out, 0);

}

+char \*url\_decode(const char \*url)

+{

+ return url\_decode\_mem(url, strlen(url));

+}

char \*url\_decode\_parameter\_name(const char \*\*query) {

struct strbuf out = STRBUF\_INIT;

Here we can see that we’re defining a function in a submodule and calling it in the main project. This is obviously a simplified example, but hopefully it gives you an idea of how this may be useful.

**Useful Aliases**

You may want to set up some aliases for some of these commands as they can be quite long and you can’t set configuration options for most of them to make them defaults. We covered setting up Git aliases in [Git Aliases](https://git-scm.com/book/en/v2/ch00/_git_aliases), but here is an example of what you may want to set up if you plan on working with submodules in Git a lot.

$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"

$ git config alias.spush 'push --recurse-submodules=on-demand'

$ git config alias.supdate 'submodule update --remote --merge'

This way you can simply run git supdate when you want to update your submodules, or git spush to push with submodule dependency checking.

**Issues with Submodules**

Using submodules isn’t without hiccups, however. For instance switching branches with submodules in them can also be tricky. If you create a new branch, add a submodule there, and then switch back to a branch without that submodule, you still have the submodule directory as an untracked directory:

$ git checkout -b add-crypto

Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary

Cloning into 'CryptoLibrary'...

...

$ git commit -am 'adding crypto library'

[add-crypto 4445836] adding crypto library

2 files changed, 4 insertions(+)

create mode 160000 CryptoLibrary

$ git checkout master

warning: unable to rmdir CryptoLibrary: Directory not empty

Switched to branch 'master'

Your branch is up-to-date with 'origin/master'.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

CryptoLibrary/

nothing added to commit but untracked files present (use "git add" to track)

Removing the directory isn’t difficult, but it can be a bit confusing to have that in there. If you do remove it and then switch back to the branch that has that submodule, you will need to run submodule update --init to repopulate it.

$ git clean -ffdx

Removing CryptoLibrary/

$ git checkout add-crypto

Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init

Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/

Makefile includes scripts src

Again, not really very difficult, but it can be a little confusing. The other main caveat that many people run into involves switching from subdirectories to submodules. If you’ve been tracking files in your project and you want to move them out into a submodule, you must be careful or Git will get angry at you. Assume that you have files in a subdirectory of your project, and you want to switch it to a submodule. If you delete the subdirectory and then run submodule add, Git yells at you:

$ rm -Rf CryptoLibrary/

$ git submodule add https://github.com/chaconinc/CryptoLibrary

'CryptoLibrary' already exists in the index

You have to unstage the CryptoLibrary directory first. Then you can add the submodule:

$ git rm -r CryptoLibrary

$ git submodule add https://github.com/chaconinc/CryptoLibrary

Cloning into 'CryptoLibrary'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

Now suppose you did that in a branch. If you try to switch back to a branch where those files are still in the actual tree rather than a submodule – you get this error:

$ git checkout master

error: The following untracked working tree files would be overwritten by checkout:

CryptoLibrary/Makefile

CryptoLibrary/includes/crypto.h

...

Please move or remove them before you can switch branches.

Aborting

You can force it to switch with checkout -f, but be careful that you don’t have unsaved changes in there as they could be overwritten with that command.

$ git checkout -f master

warning: unable to rmdir CryptoLibrary: Directory not empty

Switched to branch 'master'

Then, when you switch back, you get an empty CryptoLibrary directory for some reason and git submodule update may not fix it either. You may need to go into your submodule directory and run a git checkout . to get all your files back. You could run this in a submodule foreach script to run it for multiple submodules.

It’s important to note that submodules these days keep all their Git data in the top project’s .git directory, so unlike much older versions of Git, destroying a submodule directory won’t lose any commits or branches that you had.

With these tools, submodules can be a fairly simple and effective method for developing on several related but still separate projects simultaneously.