Next-Level Git

Revision 1.1 - 08/13/17

Brent Laster

Setup - Installing Git

If not already installed, please go ahead and install Git on your laptop. Install packages for Windows and Mac are available via the internet.

Windows: (Bash shell that runs on Windows)

http://git-scm.com/download/win

Mac/OS X:

http://git-scm.com/download/mac

Linux:

http://git-scm.com/download/linux

After installing, confirm that git is installed by opening up the Git Bash shell or a terminal session and running:

```
git --version
```

You'll also need a free GitHub account from http://github.com.

Please ensure these prerequisites are done before the labs.

==

END OF SETUP

==

Lab 1 - Using rerere

Purpose: In this first lab, we'll learn how to use the Git Rerere functionality to teach Git how to resolve merges automatically.

1. Go to http://www.github.com

2. Log in to your own github account.

3. Browse to the greetings project at https://github.com/brentlaster/greetings

4. Click on the **Fork** button at the top and wait while the repository is forked to your userid.

   (Note: The **Fork** button is on the right in the same row as the brentlaster/greetings title.)

5. Open up a git session on your local machine and start out in your home directory.

6. Configure your basic Git identifier information.

```
$ git config --global user.name "<first> <last>"
$ git config --global user.email <email address>
```

7. Clone your new repository down to your local system.

```
$ git clone https://github.com/<your github userid>/greetings.git
```

8. Change into the greetings directory.

```
$ cd greetings
```

Look around either locally or on github. Note that we have two branches, **master** and **topic1** - each with a different version on the tip of the branches. Each of these versions is also different from their common ancestor.

9. Now, enable the rerere functionality.

```
$ git config --global rerere.enabled 1
```

10. While the clone brought down all of the origin branches, we need to create a local branch to track **origin/topic1**. Execute the command below:

```
$ git branch topic1 --track origin/topic1
```

11. Now, let's merge **topic1** into **master**

$ git checkout master (if not already on master)

```
$ git merge topic1
```

Note that there were merge conflicts, but you can also see the new message about "**Recorded preimage…**" Git is remembering the conflicts and resolution.

12. We can use options to git rerere to see what git rerere knows at this point.

$ git rerere status (to see what files rerere is tracking)

$ git rerere diff (to see resolution so far)

13. Edit the file and resolve the conflicts as you want. Then, stage and commit the changes.

<edit file and resolve conflicts>

```
$ git commit -am "<comment>"
```

Note the line in the output that mentions "**Recorded resolution…**" Git has now learned how to resolved this conflict in the future.

14. Since this was only a test merge, undo it now.

```
$ git reset --hard HEAD~1
```

15. Now, assume that it's later and we want to rebase **topic1** onto **master**. Issue the following commands.

```
$ git checkout topic1
$ git rebase master
```

16. Note that even though there were conflicts, git states that it "**Resolved… using previous resolution**". (You'll have to scan through the other output to find this.) This is because we taught it how earlier.

17. Since the conflict is already resolved, stage the changes and let the rebase continue.

```
$ git add  .
$ git rebase --continue
```

==

END OF LAB

==

Lab 2 - Using the Git bisect command

Purpose: In this lab, we'll learn how to quickly identify the commit that introduced a problem using Git Bisect.

1. Go back to your home directory (or choose a different directory from lab 1) on your machine and clone the bisect project from GitHub.

```
git clone https://github.com/brentlaster/bisect
```

2. Change directory into the new bisect directory and run the sum.sh program to see what it does and see if you get a correct answer.

```
cd bisect
```

```
./sum.sh
```

3. We need to figure out where the problem was introduced.   Start the git bisect operation.

```
git bisect start
```

4. Tell git that the current version is incorrect.

```
git bisect bad
```

5. Get a list of the SHA1 commits in the branch.

```
git log --oneline
```

6. Tag the original commit to make it easier to work with. (Note version 1 not version 12.)

```
git tag first <SHA1 of version 1 commit>
```

7. Take a look at the history now “decorated” with the tags and references.

```
git log --oneline --decorate
```

8. Get a copy of the first version.

```
git checkout first
```

   (Why didn’t we have to specify a file here?)

9. Run the first version of sum.sh (1.01) and verify that it works as expected.

```
./sum.sh
```

10. Mark that revision as good.

git bisect good

11. Git has now checked out a version of the file from the “middle” of the set of revisions in the branch. Which one did it checkout and is that version good?

    ./sum.sh (Note the version number that is output.)

    (Note that if you wanted to get a clearer idea of where this one was in the chain, you could always do “git log master --oneline” and find the sha1 in that list.)

12. Mark this version appropriately - depending on whether it works or not.

    git bisect good (if it returns the right answer)

    OR

    git bisect bad (if it doesn’t return the right answer)

13. Repeat step 12 (running the shell script and doing git bisect bad or git bisect good) until you see a message that says:

    “<SHA1> is the first bad commit” (Message will be displayed immediately under bisect command.)

14. Once you see the message about the “…first bad commit”, reset to the commit that is prior to the bad one. To do this, use the following command:

    git bisect reset refs/bisect/bad^

    (Here, the refs/bisect/bad is the first bad revision and the “^” on the end means “the one before”.)

15. Create a new branch from this version and switch to it. Examine the history to make sure it looks correct.

    git checkout -b <branch-name>

    git log --oneline

==

END OF LAB

==

Lab 3 - Working with Worktrees

Purpose: In this lab, we'll get some experience with worktrees.

1. In my GitHub space, I have a project called calc2 which is a simple javascript calculator program. It has multiple branches for features, documentation, etc. For this lab, I have split it up into three separate projects: *super_calc*, a version of the calc2 project with only the master and feature branches; *sub_ui*, a separate repository consisting of only the content of the ui branch split out from the calc2 project; and *sub_docs*, a separate repository consisting of only the content of the docs branch split out from the calc2 project.

Log in to your GitHub account and fork the three projects from the following listed locations. (As a reminder, the fork button is in the upper-right corner of the pages.) This will prepare your area on GitHub for doing this lab, as well as the labs on subtrees and submodules.

https://github.com/brentlaster/super_calc.git

https://github.com/brentlaster/sub_ui.git

https://github.com/brentlaster/sub_docs.git

2. In a new directory, clone down the super_calc project that you forked in step 1, using the following command:

```
$ git clone https://github.com/<your github userid>/super_calc.git
```

3. Now, change into the cloned directory - super_calc.

```
$ cd super_calc
```

4. In this case, you want to work on both the master branch and the features branch at the same time. You can work on the master branch in this directory, but you need to create a separate working tree (worktree) for working on the features branch. You can do that with the worktree add command, passing the -b to create a new local branch from the remote tracking branch.

```
$ git worktree add -b features ../super_calc_features origin/features
```

i5. Change into the new subdirectory with the new worktree. Note that you are on the features branch. Edit the calc.html file and updated the line in the file surrounded by <title> and </title>. The process is described below.

```
$ cd ../super_calc_features
```

Edit calc.html and change

i

<title>Calc</title>

to

<title> github_user_id's Calc</title>

substituting in your GitHub user ID for "github_user_id".

6. Save your changes and commit them back into the repository.

```
$ git commit -am "Updating title"
```

7. Switch over to your original worktree.

```
$ cd ../super_calc
```

8. Look at what branches you have there.

```
$ git branch
```

9. Note that you have the features branch you created for the other worktree. Do a log on that branch; you can see your new commit just as if you had done it in this worktree.

```
$ git log --oneline features
```

10. You no longer need your separate worktree. However, before you remove it, take a look at what worktrees are currently there.

```
$ git worktree list
```

11. You can now remove the worktree. First, remove the actual directory; then use the prune option to get rid of the worktree reference.

```
$ rm -rf ../super_calc_features
$ git worktree prune
```

==

END OF LAB

==

Lab 4 - Working with Submodules

Purpose: In this lab, we'll get some practice working with Submodules in Git

1. Start out in the super_calc directory for the super_calc repository that you cloned from your GitHub fork in Lab 3. You're going to add another repository as a submodule to super_calc.

2. Add the sub_ui repository as a submodule to the super_calc project by running this command:

```
$ git submodule add https://github.com/<your github userid>/sub_ui sub_ui
```

3. This adds the sub_ui as a submodule to your super_calc repository. In the process, Git clones down the repository into the subdirectory and also creates and stages a .gitmodules file to map the connection with the super_calc project. Look at the directory listing to see the new subdirectory. Then look at the status to see the staged .gitmodules file. Finally, display the contents of the .gitmodules file to see what's in there; run the following commands from the super_calc subdirectory:

```
$ ls
$ git status
$ git show :.gitmodules
```

4. Now you need to commit and push the staged submodule mapping and data to your local and remote repositories. Run the following commands:

```
$ git commit -m "Add submodule sub_ui"
$ git push
```

5. Now you can clone a new copy of this project with the submodule. Change to a higher-level directory and clone a copy of the project down as super_calc2.

```
$ cd ..
$ git clone https://github.com/<your github userid>/super_calc super_calc2
```

6. Change into the super_calc2 directory and look at what's in the sub_ui subdirectory. Run the submodule status command to see what the status of the submodule is.

```
$ cd super_calc2
$ ls sub_ui
$ git submodule status
```

7. Notice the hyphen (-) in front of the SHA1 value. This indicates that the submodule has not been initialized yet relative to the super project. You could have done this at clone time using the --recursive option. However, because you didn't, you need to use the update --init subcommand for the submodule operation, as follows:

```
$ git submodule update --init
```

8. Git clones the sub_ui code into the submodule. Look at the sub_ui subdirectory to see the contents, and then run the submodule status command again.

```
$ ls sub_ui
$ git submodule status
```

This time, you see a space at the beginning (instead of the minus sign) to indicate the submodule has been initialized.

9. Now, you need to make a simple update to the code in the submodule.

Change into the sub_ui subdirectory and edit the calc.html file there as follows: change the line

<title>Advanced Calculator</title>

in the file to

<title> your_name_here Advanced Calculator</title>

substituting your actual name for "your_name_here".

Save your changes.

10. Commit your changes into the submodule.

```
$ git commit -am "Update title"
```

11. Do a quick log command and note the SHA1 value associated with the commit you just made.

```
$ git log --oneline
```

12. Change back to the super_calc2 project (up one level) and run a submodule status command.

```
$ cd ..
$ git submodule status
```

13. Note that the submodule SHA1 reference now points to your latest commit in the submodule, but there is a plus sign (+) at the front of the reference. This indicates that there are changes in the submodule that have not yet been committed back into the super project. Run a git status command to get another view of what's changed for the super project.

```
$ git status
```

14. The status command gives you much more information about what's changed. It tells you that the sub_ui module has been changed and is not updated or staged for commit. To complete the update, you need to stage and commit the sub_ui data. You can then push it out to your GitHub remote repository. To complete the process, execute the commands below.

```
$ git commit -am "Update for submodule sub_ui"
$ git push
```

15. Now that you've updated the submodule and the supermodule, everything is in sync. Run a git status and a git submodule status command to verify this.

```
$ git status
$ git submodule status
```

==

END OF LAB

==

Lab 5 - Working with Subtrees

Purpose: In this lab, we'll see how to work with subtrees in Git

1. Start out in the super_calc directory for the super_calc repository (not super_calc2) that you cloned from your GitHub fork in lab 4. You're going to add another repository as a subtree to super_calc.

2. To add the repository, use the following command.

```
$ git subtree add -P sub_docs --squash https://github.com/<your github user id>/sub_docs master
```

Even though you don't have much history in this repository, you used the --squash command to compress it. Note that the -P stands for prefix, which is the name your subdirectory gets.

3. Look at the directory structure; note that the sub_docs subdirectory is there under your super_calc project. Also, if you do a git log, you can see where the subproject was added and the history squashed.

```
$ ls sub_docs
$ git log --oneline
```

Note that there is only one set of history here because there is only one project effectively - even though we have added a repository as a subproject.

4. Now, you will see how to update a subproject that is included as a subtree when the remote repository is updated. First, clone the sub_docs project down into a different area.

```
$ cd ..
$ git clone https://github.com/<your github user id>/sub_docs sub_docs_temp
```

5. Change into the sub_docs_temp project, and create a simple file. Then stage it, commit it, and push it.

```
$ cd sub_docs_temp
$ echo “readme content” > readme.txt
$ git add .
$ git commit -m “Adding readme file”
$ git push
```

6. Go back to the super_calc project where you have sub_docs as a subtree.

```
$ cd ../super_calc
```

7. To simplify future updating of your subproject, add a remote reference for the subtree’s remote repository.

```
$ git remote add sub_docs_remote https://github.com/<your github user id>/sub_docs
```

8. You want to update your subtree project from the remote. To do this, you can use the following subtree pull command. Note that it’s similar to your add command, but with a few differences:

- You use the long version of the prefix option.
- You are using the remote reference you set up in the previous step.
- You don’t have to use the squash option, but you add it as a good general practice.

```
$ git subtree pull --prefix sub_docs sub_docs_remote master --squash
```

Because this creates a merge commit, you will get prompted to enter a commit message in an editor session. You can add your own message or just exit the editor.

9. After the command from step 8 completes, you can see the new README file that you created in your subproject sub_docs. If you look at the log, you can also see another record for the squash and merge_commit that occurred.

```
$ ls sub_docs
$ git log --oneline
```

10. Changes you make locally in the subproject can be promoted the same way using the subtree push command. Change into the subproject, make a simple change to your new README file, and then stage and commit it.

```
$ cd sub_docs
$ echo “update” >> readme.txt
$ git commit -am “update readme”
```

11. Change back to the directory of the super_project. Then use the subtree push command below to push back to the super project’s remote repository.

```
$ cd ..
$ git subtree push --prefix sub_docs sub_docs_remote master
```

Note the similarity between the form of the subtree push command and the other subtree commands you've used.

12. The next few steps show you how to take a subproject, put it onto a different branch, and then bring that content into a separate repository.
First, use the subtree split command to take the content from the sub_docs subproject and put it into a branch named *docs_branch* in the super_calc area.

```
$ git subtree split --prefix=sub_docs --branch=docs_branch
```

13. Look at the history for the new docs_branch. You can see all of the content that you have in the sub_docs project.

```
$ git log --oneline docs_branch
```

14. Create a new project into which you can transfer the docs_branch content.

```
$ cd ..
$ mkdir docs_proj
$ cd docs_proj
$ git init
```

15. Finally, use the git pull command in a slightly different context to pull over that content into the master branch of your new project.

```
$ git pull ../super_calc docs_branch:master
```

16. Do a git log of the master branch in the new project to see the copied content.

```
$ git log --oneline master
```

===

END OF LAB

===

Lab 6 - Working with Filter-branch

Purpose: In this lab, we'll see how to work with the filter branch functionality in Git

1. For this lab, we'll need a more substantial project to work with. We'll use a demo project with several parts that I use in other classes. In your home directory (or a directory without any existing clones of git projects), clone down the roarv2 project from http://github.com/brentlaster/roarv2.

```
$ git clone https://github.com/brentlaster/roarv2
```

2. We are interested in splitting the web subdirectory tree out into its own repository. First, you note what the structure looks like under web, as well as the log, for a reference point.

```
$ cd roarv2
$ ls -la web
$ git log --oneline
$ git log --oneline web
```

3. Now we can run the command to split the web piece out into its own repository. (Note the syntax at the end. It's the word "web" followed by a space, then a double hyphen, then another space, then "—all".)

```
$ git filter-branch -f --subdirectory-filter web -- --all
```

4. After the preceding step runs, you should see messages about "refs/heads/<branch name> being rewritten". This is the indication that the process worked. Do an ls and look at the history to verify that your repository now contains only the web pieces.

```
$ ls -la
$ git log --oneline
```

5. Now, let's restore the original larger repository back for the other parts of this lab. To do this, we'll find the original SHA1 value for HEAD from before we did the operation and reset back to it. Take the first 7 characters of the value you find in the first step below and plug them in to the second step.

```
$ cat .git/refs/original/refs/heads/master
$ git reset --hard <first 7 characters from SHA1 output of step above>
```

6. Let's look at another application of filter-branch - removing files from the history. In this case, we'll use filter-branch to remove the file build.gradle from the master branch just as a prevalent example. You should be on the master branch. First, let's see everywhere this file has been involved in a change. Run the command below to do this: (Note that there are spaces on each side of the last --).

```
$ git log --oneline --name-only --  build.gradle
```

7. Now, let's run the filter-branch command to remove the build.gradle file from all of the commits in this branch. (Note that these are single quote characters around the git rm command, not backticks.)

```
$ git filter-branch -f --index-filter 'git rm --cached --ignore-unmatch build.gradle'
```

8. Take a look now and notice that the file is no longer in the commits for this branch.

```
$ git log --oneline --name-only --  build.gradle
```

9. Let's do one more use case for the filter-branch command. We'll use the env-filter to change the email address for a few of the commits in the master branch. First, take a look at the commits and email addresses before the change with the command below:

```
$ git log --format="%h %ae"
```

10. To do this, as part of the command, set the desired email address to your email address, and then we'll export it for the filtering. Here's the command to run:

```
$ git filter-branch -f --env-filter 'GIT_AUTHOR_EMAIL=<your email address>; export
GIT_AUTHOR_EMAIL'
```

11. After this change, you can run the same command as above to see the updates:

```
$ git log --format="%h %ae"
```

==

END OF LAB

==

Lab 7: Working with Interactive Rebase

Purpose: In this lab, we'll see how to use interactive rebase to squash multiple commits into one.

1. Pick one of your directories that already has a git project in it. (You can use the roarv2 directory from the last lab for simplicity if you want.) In that working directory, make three new changes and commit each one separately. (Once you type the first one, you can just the recall function (up arrow) to bring up the line again and edit the last number if you want.)

```
$ echo data >> lab.txt; git add lab.txt; git commit -m "Update to lab.txt, version 1"

$ echo data >> lab.txt; git add lab.txt; git commit -m "Update to lab.txt, version 2"

$ echo data >> lab.txt; git add lab.txt; git commit -m "Update to lab.txt, version 3"
```

2. Do a git log to see the 3 separate commits showing up in the history.

```
$ git  log  (or git log --oneline)
```

3. After doing the 3 commits, we'll initiate an interactive rebase for the last 3 commits. As stated, we specify the starting point as the one before the first one we want to change.

$ git rebase -i HEAD~3

4. Now, in the editor window that comes up with the list of commits (titled "git-rebase-todo"), modify the action on the left to be "squash" for the bottom two entries.   The format of your file should look similar to below (note that this is demonstrating the contents of the upper part of the file NOT commands you type in and execute):

```
pick    <sha 1>  "Update to lab txt, version 1"
squash <sha 1> "Update to lab.txt, version 2"
squash <sha 1> "Update to lab txt, version 3"
```

5. Save your file and exit the editor. Git will now start the interactive rebase running. You can watch it go through the steps.   **After it first runs, it will stop and bring up the editor again.** This is so you can enter what you want the commit message to be for the squashed commit (since you won't have the 3 separate ones anymore).   You can just put whatever you want in this field (or leave it as-is). Save any changes and exit the editor. The rebase should then continue and complete.

6.   Finally, do another git log and notice that there is now a single (squashed) commit where there were previously 3.

$ git log (or git log --oneline).

===

END OF LAB

===

Lab 8: Using a Git Attributes File to Create a Custom Filter

Purpose: In this lab, we'll see how to work with two kinds of custom filters to modify file contents automatically on checkout and commit.

Setup:

Suppose that you have a couple of HTML files that you use as a header and footer across multiple divisions in your company. They contain a placeholder in the form of the text string %div to indicate where the proper division name should be inserted.

You want to use the smudge and clean filters to automatically replace the placeholder with your division name (ABC) when you check the file out of Git, and set it back to the generic version if you make any other changes and commit them.

1. Pick one of your directories that already has a git project in it. (You can use the roarv2 directory from the recent labs for simplicity if you want.) Create two sample files to work with. Commands to create them are below (you may use an editor if you prefer).

```
$ echo "<H1>Running tests for division:%div</H1>" > div_test_header.html

$ echo "<H1>Division:%div testing summary</H1>" > div_test_footer.html
```

2. Go ahead and stage and commit these new files into your local Git repository.

```
$ git add div*.html

$ git commit -m "adding div html files"
```

3. Now, we'll create a custom filter named "insertDivisionABC" that will have the associated "clean" and "smudge" actions. This is done by using git config to define this in our Git configuration. Execute the following two config commands:

```
$ git config filter.insertDivisionABC.smudge "sed 's/%div/ABC/'"

$ git config filter.insertDivisionABC.clean "sed 's/ision:ABC/ision:%div/'"
```

4. To see how this is setup in the config file, take a look at your .git/config file and find the section for the insertDivisionABC filter:

```
$ cat  .git/config
```

Look for the section starting with "[filter "insertDivisionABC"]

5. Now, we just need to create a Git attributes file with a line that tells Git to run your filter for files matching the desired pattern. We will create a very simple one-line file for this.

```
$  echo "div*.html   filter=insertDivisionABC"  >  .gitattributes
```

6. Now, we'll remove our local copies of the files and check them out again so they will get the filter applied.

```
$ rm div*.html
$ git checkout div*.html
```

7. Take a look at the contents of the two files after the checkout has applied the smudge filter.

```
$ cat div*.html
```

Notice that we have the division name (ABC) inserted into the files now.

8. Edit the html files and make a simple modification. For example, you might change "testing summary" to "full testing summary" and/or "tests for division" to "all tests for division". Save the changes.

9. Now do a git diff on the files.   Notice that with the clean filter in place, the diff takes into account the filter and just shows you the differences without the substitution being applied.

```
$ git diff
```

10. Add the files back into Git and run a status command to see the state.

```
$ git add div*.html
$ git status
```

11. Run a git diff command and note that it shows no differences. Then cat the local versions of the files. Notice that they still show the substitution from the smudge.

```
$ git diff
$ cat div*.html
```

12. We know that the clean filter should have removed the substitution when the files were staged. To verify that, we'll use a special form of the show command to see the versions of the files in the staging area.

```
$ git  show  :0:div_test_header.html
$ git  show  :0:div_test_footer.html
```

Notice that the change to insert the division has been "cleaned" per the clean filter.

==

END OF LAB

==

Lab 9: Creating a post-commit hook

Purpose: In this lab, we'll see how to create a post-commit hook and put it in place to be active in our local Git environment. We could create the hook in many different programming languages, but we'll use shell scripting here because it's portable among most unix implementations (including the Git bash shell for Windows).

Setup:

Suppose that you want to mirror out copies of files when you commit them into your local repository - but only for branches that start with "web". Further you must have the config value of hooks.webdir set to a valid directory on your system.

NOTE: Up through step 7, the parts in bold represent code that you are typing into the editor, not direct commands to run.

1. Open an editor session.

2. Enter the identifier for the bash shell file and a simple echo that will be shown when this is running. Type or paste these lines into the editor (adjusting the bash location if needed.)

```
#!/usr/bin/env bash
echo Running post-commit hook
```

3. Now, add a line to get the value of the hooks.webdir custom configuration setting.

```
web_dir=$(git config hooks.webdir)
```

4. Next, we'll figure out the current HEAD.

```
new_head_ref=$(git rev-parse --abbrev-ref HEAD)
```

5. We unset this environment variable since it is not needed and will cause problems if set for our "checkout" to a non-Git area.

```
unset GIT_INDEX_FILE
```

6. Now we will create the conditional execution block to mirror the code if the conditions are met. The first part is the if statement that checks for the configuration value being set and another condition that checks that the branch starts with "web".

```
if [[ -n "$web_dir" ]] && [[ $new_head_ref =~ ^web.*$ ]]; then
```

7. For the body of the conditional, we will add a line that calls "git" and checks out the current code into the configured location. Notice that this statement uses the git-dir setting to redirect the checkout to the desired mirror directory. We will also add the closing "fi" to finish our if block.

```
    results=$(git --work-tree="$web_dir" --git-dir=$GIT_DIR checkout -f)
fi
```

8. Save the file, making absolutely sure that it is **saved in the .git/hooks directory as post-commit** (with no file extension). (For example, if you were putting this into the roarv2 area, you would save the file as <path to roarv2>/roarv2/.git/post-commit) From your working directory, verify that you see the file in the directory.

```
$ ls .git/hooks/post-commit
```

9. Create a directory (somewhere outside of your local Git working directory for the hook to eventually mirror your code into. Also configure the hooks.webdir value to point to that location.

$ mkdir (some-directory-name) (such as mkdir ../mirror)

(in your working directory with the git project)

```
$ git config hooks.webdir  (some-directory-name)
```

(make sure to use the correct absolute or relative path to get to the directory you created above)

10. Next, we'll test out our hook. In your original working directory, modify any of your files and stage and commit it.

```
$ echo more >   (some-file-name)
$ git commit -am "update file"
```

Notice that after you do the commit, you should see the "Running post-commit hook" message. However, the hook won't do anything else because the branch is master and not web*. To verify that's the case, you can inspect the directory you created for the mirror.

```
$ ls (some-directory-name)
```

There should be nothing there.

11. Now, let's set things up so our hook will mirror out the contents of our repository. Create a new branch and switch to it. You can name it anything you want as long as it starts with "web". An example is shown below.

```
$ git checkout -b  webtest
```

12. Make a change to a file and stage and commit it into your repository.

```
$ echo  more  >>  (some-file-name)
$ git commit -am "update"
```

This time, the hook should fire since we have the config value defined and are in a branch that starts with "web". You should see the "Running" message from the hook and then be able to see the contents of your repository in the directory that you configured.

```
$ ls  (some-directory-name)
```

END OF LAB

==