GitLab Walkthrough 3a
Developing with the Remote Repository

Mark Eramian and Anthony Kusalik

October 22, 2020

Developing with the Remote Repository

In this walkthrough, we will play the role of a developer who will be working on a software project stored in a remote repository that was set up by our team lead (see GitLab Walkthrough 2).

Step 0: Fake a New Identity

Since we want to pretend we are now a developer on the team, rather than the team lead, we will change our Git identity. Execute the following commands:

```
git config --global user.name "programmer-a"
git config --global user.email "ProgA@cs.usask.ca"
```

Step 1: Clone the Remote Repository

If you still have the repository you created in Gitlab Walkthrough 2 and have not modified it since completing that walkthrough, you can just open a terminal and change to the directory containing the existing working copy. Otherwise, clone the remote repository that you set up in Gitlab Walkthrough 2 (follow the procedure in Gitlab Walkthrough 2, Step 1).

The working copy should now contain four files: `bar.c`, `foo.c`, `Makefile`, and `README.md`.

Step 2: Modify the Working Copy

Simulate modifying the code by executing these two commands:

```
echo "Give foo.c another line"  >> foo.c
echo "this is the Makefile" > Makefile
```

The first command adds a line to the end of `foo.c`. The second command replaces the makefile with a new makefile. We will learn more formally what the > and » terminal command syntax does in an upcoming lecture topic.

Now check the status of the local repository with `git status`:

```
bash-5.0$ 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:   Makefile
        modified:   foo.c

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

This shows that Git has noticed that `Makefile` and `foo.c` have changed, and that these changes have not yet been committed to the local repository as a new version. Also note the first line which indicates that the local repository (not the working copy) is up to date with respect to the remote repository, that is, there are no new changes in the remote repository.

We can even ask Git what exactly about these two files changed. Execute the command `git diff`:

```
bash-5.0$ git diff
diff --git a/Makefile b/Makefile
index 4b5806c..536274c 100644
--- a/Makefile
+++ b/Makefile
@@ -1 +1 @@
-this is the makefile
+this is the Makefile
diff --git a/foo.c b/foo.c
index d68c11e..d3a2226 100644
--- a/foo.c
+++ b/foo.c
@@ -1 +1,2 @@
 this is file foo.c
+Give foo.c another line
```

The blue lines (colours added by the author) indicate a change record. The lines beginning with a - (coloured red) are lines that were deleted, lines beginning with a + are lines that were added. So this shows that in `Makefile` the line `"this is the makefile"` was deleted and replaced by the line `"this is the Makefile"`. In the second instance, we see that `foo.c` had the line `Give foo.c another line` added after the line `this is file foo.c` (which is a line that begins with a space indicating that it did not change because it is not preceded by + or -). We can also see that Git is generating these change records by running the terminal command `diff`. `diff` is a command line tool that determines the differences between two text files. You can even run these commands yourself. Try the command `diff -git a/Makefile b/Makefile` and compare the reuslt to the output above.

Step 3: Commit The Modified Code as a New Version

Now it's time to commit our changes and create a new version in the repository for our updated code. Use the command

```
git commit -a -m "appended line to foo.c and replaced content of Makefile"
```

The result should look like this:

```
bash-5.0$ git commit -m "appended line to foo.c and replaced content of Makefile"
[master ae6dc03] appended line to foo.c and replaced content of Makefile
 2 files changed, 2 insertions(+), 1 deletion(-)
```

Now let's look a the change logs. Type `git log`:

```
bash-5.0$ git log
commit ae6dc03471876f8bc68055f0b05891b1b6af7ec8 (HEAD -> master)
Author: programmer-a <ProgA@cs.usask.ca>
Date:   Mon Nov 25 09:26:57 2019 -0600

    appended line to foo.c and replaced content of Makefile

commit d68c1c7a7db4fc65944be623d5cc832d626c8aea (origin/master, origin/HEAD)
Author: team-lead <TeamLead@cs.usask.ca>
Date:   Mon Nov 25 08:52:14 2019 -0600

    Initial commit by TeamLead
```

The log shows the most recent change first, which was the change by "programmer-a". The earlier change was the initial commit done by the team lead, and we can clearly see this from the logs.

Step 4: Push The New Local Version to the Remote Repository

If we quickly do another `git status` we get the following:

```
bash-5.0$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Notice the line that says that our local branch is ahead of the `origin/master` branch. If we remember that `origin` is a default alias for the URL of our remote repository, we will realize that the status report is telling us that we have changes in our local repository that have not yet been pushed to the remote repository. So let's do that now.

Now we want to push our changes to the local repository. Either of these commands should work. If the first one doesn't the second one should.

```
git push

# or

git push -u origin master
```

Regardless of which command you successfully used, the result should be as follows;

```
bash-5.0$ git push -u origin master
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 48 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 429 bytes | 85.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To git.cs.usask.ca:ajk449/git-ex.git
   d68c1c7..ae6dc03  master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

If we now execute another `git status` command, we'll see that the local and remote repositories are once again in sync:

```
bash-5.0$ git status
On branch master
Your branch is up to date with 'origin/master'.
```

Step 5: Take a Break

In our role as Programmer A, we are now tired and will take a break. In the meantime, Programmer B will independently make changes to our repository.

After you take a break, rest your eyes and your brain, you should go and complete GitLab Walkthrough 3b. When that is complete, return to this point and continue with this walkthrough.

Step 6: Interlude

Before continuing with this step, you should have already completed GitLab Walkthrough 3b. That walkthrough simulated a different programmer, Programmer B, making changes and pushing them to the remote repository. We now return to the role of Programmer A who will go on to make additional changes and when they go to push those changes, discover that someone else (Programmer B!) has already made changes to the remote repository which necessitates some extra steps before we, Programmer A, can push new changes.

Before proceeding, make sure you return to the directory with your local repository for programmer A which you started in Step 0 of this walkthrough.

Step 7: Fake Programmer A's Identity again

Since we are once again pretending to be programmer A, change our identity:

```
git config --global user.name "programmer-a"
git config --global user.email "ProgA@cs.usask.ca"
```

Step 8: Follow the Typical Multi-Programmer Workflow

Return to the original clone of the remote repository that you were using when you first began this walkthrough. Keep in mind that this local repository does not yet have the changes that Programmer B pushed to the shared remote repository in Gitlab Walkthrough 3b.

We will now simulate the multi-programmer workflow that was described in Section 6.5.4 of the textbook. Here it is to remind you:

Typical Multi-programmer Workflow

1. Decide to implement new feature — check out the `develop` branch.
2. Make changes to local copy of `develop` branch
3. Commit changes to `develop` branch.
4. If feature is done, goto step 5, otherwise return to step 2.
5. Check out the `master` branch.
6. Pull remote repository updates to `master` using `git pull`.
7. Check out the `develop` branch.
8. Rebase the `develop` branch on the current version of the `masgter` branch — use the command `git rebase master`.
9. Check out the `master` branch.
10. Merge `develop` branch into `master` branch.
11. Push `master` branch updates (which now includes of your changes integrated with all updates from the remote repository since your last pull) to the remote repository with `git push`.
12. Goto step 1.

First, we create a new branch called develop and change our working copy to that branch using the following two commands (step 1 in the workflow):

```
git branch develop
git checkout develop
```

Typeing `git branch` shows that we are now on the `develop` branch:

```
bash-5.0$ git branch
* develop
  master
```

Now we will simulate Programmer A making some changes to the `develop` branch using the following commands (step 2 in the workflow):

```
echo "new first line for bar.c" > newbar.c
cat bar.c >> newbar.c
mv newbar.c bar.c
```

The result of all of this is that `"new first line for bar.c"` becomes the new first line of `bar.c` and its existing contents follow. If you do a `git status` you'll see that `bar.c` now has changes waiting to be committed.

Now commit the new changesgit with the command (step 3 in the worfklow):

```
git commit -a -m "added new first line to bar.c"
```

The result is:

```
bash-5.0$ git commit -m "added new first line to bar.c"
```

```
[master 830be3a] added new first line to bar.c
 1 file changed, 1 insertion(+)
```

Now let's assume that Programmer B has finished the new feature (step 4 of workflow), in which case it is time to get our changes to the remote server. But before we do so, we have to make sure that there are no changes on the server we don't yet have (and there are, from Programmer B!).

To get any remote changes that have to be merged with ours before we can push our changes, we return to the master branch (step 5 of the workflow) using

```
git checkout master
```

If you like, you can type `git branch` to verify that you are now on the master branch:

```
bash-5.0$ git branch
  develop
* master
```

Now it's time to pull any new updates from the master branch of the remote repository to our local master branch (step 6 of the workflow). To do this, we use `git pull`:

```
bash-5.0$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From gitusask:mark.eramian/cmpt-214-test-repo
   c16fc13..294d0f6  master     -> origin/master
Updating c16fc13..294d0f6
Fast-forward
 bar.c | 1 +
 1 file changed, 1 insertion(+)
```

Now we have the changes from the remote repository on our local master branch, and the changes Programmer A just made to the `develop` branch, and we need to merge them.

To merge the two sets of changes we start by checking out the `develop` branch again (step 7 of the workflow):

```
git checkout develop
```

Now we *rebase* the `develop` branch using the current `master` branch as a base. This replays the changes we made to the `develop` branch using the new `master` branch as a starting point. This is the best way to safely merge your changes with remote changes in a way that doesn't risk putting the master branch in a state of disrepair. To do this, run the command `git rebase master` (step 8 of the workflow):

```
bash-5.0$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added new first line to bar.c
```

The changes in the commit with the log entry `added new first line to bar.c` was applied to the `current` master branch! Now the remote repository changes are incorporated with the changes we made to the `develop` branch.

The next step is to get the combined updates in the `develop` branch back into the `master` local master branch so we can push them to the remote repository. So first we check out the master branch (step 9 of the workflow) using

```
git checkout master
```

Now we need to merge the `develop` branch with the `master` branch (step 10 of the workflow) using `git merge develop`:

```
bash-5.0$ git merge develop
Updating 294d0f6..85d707c
Fast-forward
 bar.c | 1 +
 1 file changed, 1 insertion(+)
```

Everything went smoothly and now all the changes are in the local `master` branch. `git status` will show us that there are commits in the local master branch that are not in the remote repository:

```
bash-5.0$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

All we need to do now is to push the updates in our local `master` branch to the remote repository (step 11 of the workflow) using `git push`:

```
bash-5.0$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 326 bytes | 326.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To gitusask:mark.eramian/cmpt-214-test-repo
   294d0f6..85d707c  master -> master
```

Now `git status` will tell us that both the local and remote `master` branches are in sync:

```
bash-5.0$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

The log will now show the changes made by different programmers:

```
bash-5.0 $ git log
commit 85d707c89b0a81d6bcc28998b52c9d842093921a (HEAD -> master, origin/master,
    origin/HEAD, develop)
Author: programmer-a <ProgA@cs.usask.ca>
Date:   Thu Jun 25 10:35:51 2020 -0600

    added new first line to bar.c

commit 294d0f6b0b37e04ddaf7866f3dba60ffc832440a
Author: programmer-b <ProgB@cs.usask.caend{lstlisting}>
Date:   Thu Jun 25 10:15:30 2020 -0600

    appended a line to bar.c

commit c16fc13dd5b1f0631bf1d4721aec653eb1cf4085
Author: programmer-a <ProgA@cs.usask.caend{lstlisting}>
Date:   Thu Jun 25 09:52:07 2020 -0600

    appended line to foo.c and replaced content of Makefile

commit 412971010eb800e279bdfe5561f98ad61805cf89
Author: team-lead <TeamLead@cs.usask.ca>
Date:   Thu Jun 25 09:45:16 2020 -0600
```

Notice the tags on the first commit that indicates that all of `(HEAD -> master, origin/master, origin/HEAD, develop)` are on the same version.

Fix some Mistakes

In this section we can demonstrate how to undo unwanted changes to the repository. Stay with the repository for Programmer A as we will not be involving any simulated Programmer B activities in this discussion.

First, let's simulate adding an erroneous line to the Makefile using this command:

```
echo "erroneous stuff" >> Makefile
```

Suppose that on this occasion we are lucky, and we realize that we have made the unwanted change to the Makefile before we commit anything. We can revert uncommitted changes to a local file using the command:

```
git checkout HEAD <filename>
```

This tells Git to check out the version of the file **<filename>** from the version `HEAD` (which is a synonym for“most recent commit”, so you don’t have to look for the hash of the most recent commit). So let’s now run the following command to undo our changes to `Makefile`:

```
git checkout HEAD Makefile
```

The result should be something like:

```
bash-5.0$ git checkout HEAD Makefile
Updated 1 path from c95fdda
```

The Makefile has now been restored to the version in the most recent commit. We can see this, because `git status` says our working copy is clean:

```
bash-5.0$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

But what if we made an unwanted change and didn’t realize it until **after** we committed it? Let’s commit an unwanted change and see how to undo it. First, execute these commands to commit an unwanted change:

```
echo "erroneous stuff" >> Makefile
git commit -a -m "added stuff to Makefile"
```

Now note the two most recent[1] entries in the log:

```
bash-5.0$ git log -n2
commit 12448895ef691246e574cc8e884694584b6fd324 (HEAD -> master)
Author: programmer-a <ProgA@cs.usask.ca>
Date:   Thu Jun 25 11:16:33 2020 -0600

    added stuff to Makefile

commit 85d707c89b0a81d6bcc28998b52c9d842093921a (origin/master, origin/HEAD, develop)
Author: programmer-a <ProgA@cs.usask.ca>
Date:   Thu Jun 25 10:35:51 2020 -0600

    added new first line to bar.c
```

We can permanently erase the last commit using the command:

```
git reset --hard HEAD^
```

[1]The command `git log -n<K>` where `<K>` is an integer number, displays the `K` most recent log entries.

which should produce the following result:

```
bash-5.0$ git reset --hard HEAD^
HEAD is now at 85d707c added new first line to bar.c
```

Remember that `HEAD` is a synonym for the hash of the most recent commit on the current branch. Our `Makefile` is once again as it was and the log has no record that our erroneous commit ever happened:

```
bash-5.0 $git log -n2
commit 85d707c89b0a81d6bcc28998b52c9d842093921a (HEAD -> master, origin/master,
    origin/HEAD, develop)
Author: programmer-a <ProgA@cs.usask.ca>
Date:   Thu Jun 25 10:35:51 2020 -0600

    added new first line to bar.c

commit 294d0f6b0b37e04ddaf7866f3dba60ffc832440a
Author: programmer-a <ProgA@cs.usask.caend{lstlisting}>
Date:   Thu Jun 25 10:15:30 2020 -0600

    appended a line to bar.c
```

Conclusion

Congratulations! You're done. You've accomplished a lot of practice with using local and remote Git repositories.