GitLab Walkthrough 2
Initializing Your New Remote Repository

Mark Eramian and Anthony Kusalik

October 22, 2020

Initializing Your New Remote Repository

In this walkthrough, we will play the role of a Team Lead who has previously created a new empty remote repository on GitLab and is now ready to populate it with the first new files for the project. The new files we add to the project in our current role as Team Lead, will later be used by programmers on our team.

Step 0: Identify Ourselves as the Team Lead

For the purposes this walkthrough we want to take on a the identity of a team lead so that we can distinguish changes we make to the repository. To do this, type the following commands:

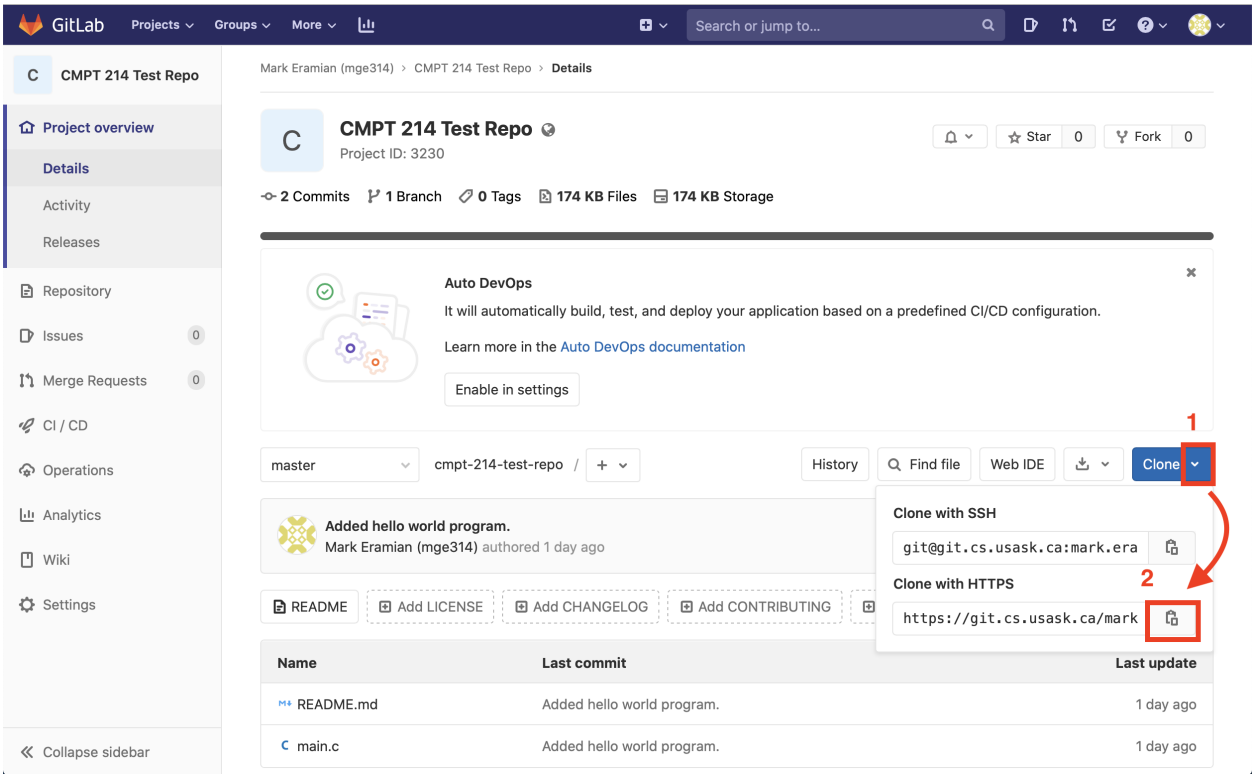
```
git config --global user.name "team-lead"
git config --global user.email "TeamLead@cs.usask.ca"
```

This will set the name and email address that are placed in the git logs when we make changes to the repository to be those of a hypothetical Team Lead. Under normal use conditions, you don't have to do this; the name and email will be set to something reasonable by default. We will change our identity again in the next walkthrough to simulate being someone else.

Step 1: Cloning your New Repository

If you just finished GitLab Walkthrough 1, you will have a newly created remote repository on GitLab which contains no files. Before you continue, you'll need to get the URL for your project. If you need to retrieve it, click on the GitLab logo on the top-right of any page on `git.cs.usask.ca`, and then click on your newly created repository. On the right there will be a blue "Clone" button. Click the little arrow and it will reveal the

URLs you need to clone the repository. Click on the clipboard icon beside "Clone with SSH", and this will copy the URL to your clipboard.



Now we can clone the empty repository form GitLab. Create an empty directory, and change to that directory in your terminal. Then type:

```
git clone <URL>
```

where `<URL>` is the URL you copied from gitlab above. It should be in the form `git@git.cs.usask.ca:<NSID>/<repository-name>.git`. The result should look something like this:

```
bash-3.2$ git clone git@git.cs.usask.ca:ajk449/git-ex.git
Cloning into 'git-ex'...
warning: You appear to have cloned an empty repository.
bash-3.2$
```

We can ignore the warning about cloning an empty repository. You should now have a new subdirectory that has the same name as your repository. It contains the local copy of the remote repository, as well as a working copy of the local repository. Now, change working directory to the directory git just created:

```
cd <repository-name>
```

where `<repository-name>` is the project's name, and the name of the folder that was created by the cloning of the repository.

Now let's check the status of our local project by typing `git status`:

```
bash-3.2$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Yep, that checks out. Our working copy should be empty since we cloned an empty repository. Now type: `ls -a` and you should get:

```
bash-3.2$ ls -a
.     ..    .git
```

Again, this checks out. There's the hidden `.git` folder that contains the repository (visible now since we used the `-a` option of the `ls` command). There are no other files because the repository is empty, so there are no files in the working copy to see.

Step 2: Add some Files ot the Repository

Now it's time to add some files to the git repository. First we need to create some files t add. Let's start with a `README.md` file. A `.md` file is a "markdown file". There is a brief explanation of markdown files under the heading "Markdown" at https://gist.github.com/m-kyle/fb0f3e9edc369adfcac7. Enter the following command:

```
echo "# Example for the GIT portion of CMPT 214 #" > README.md
```

This will create a file `README.md`. Then type the following commands to create some more (fake) C source files:

```
echo "this is file foo.c" > foo.c
echo "this is file bar.c" > bar.c
echo "this is the makefile" > Makefile
```

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

```
bash-3.2$ git status
On branch master

No commits yet

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

     Makefile
     README.md
     bar.c
```

```
    foo.c

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

The status shows us that while we have created new files, they are currently "untracked", that is, they are not yet actually part of our working copy. The next step is to add the files to the working copy. Type the following commands:

```
git add README.md
git add foo.c
git add bar.c
git add Makefile
```

If you do this correctly you shouldn't see any output from any of these commands. We could also have used the command

```
git add .
```

which adds all untracked files in the current directory and all subdirectories to the working copy which would have saved us some keystrokes! What is the status of our local project now? Type `git status`:

```
git status

bash-3.2$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   Makefile
    new file:   README.md
    new file:   bar.c
    new file:   foo.c

bash-3.2$
```

This shows that all of the files we created are now recognized as new files in the working copy and are now being "tracked". To actually add them to the repository and create the first version of our project, we need to perform a *commit* of these changes to our local copy of the repository. Type type command:

```
git commit -m "Initial commit by TeamLead"
```

When you do this, you should see the following:

```
bash-3.2$ git commit -a -m "Initial commit by TeamLead"
[master (root-commit) d68c1c7] Initial commit by TeamLead
 4 files changed, 4 insertions(+)
 create mode 100644 Makefile
 create mode 100644 README.md
 create mode 100644 bar.c
 create mode 100644 foo.c
bash-3.2$
```

This indicates that these new files have been created within the local copy of the remote repository.

Step 3: Push Updates to the Remote Repository

The local copy of our remote repository now has four new files it. But the remote repository is still empty. We must upload our updates to the remote repository. To do this, type the command:

```
git push -u origin master
```

which instructs git to push our changes to the `master` branch of the remote repository `origin` (which is a default alias for the URL that we cloned our local repository from so we don't have to remember the URL!). You should see the following results:

```
bash-3.2$ git push -u origin master
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 440 bytes | 440.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To git.cs.usask.ca:ajk449/git-ex.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
bash-3.2$
```

The last line is interesting, it tells us that the master branch of our local repository is now "linked" with the master branch of the remote repository. In the future we can push from and pull to the local master branch using simply `git push` or `git pull` without having to specify which branch on the remote repository we want push to or pull from. You can see details of this "link" by typing `git remote show origin`.

Note that you may be prompted for a user name (NSID) and password if you did not set up SSH keys for your login at `git.cs.usask.ca` (See GitLab Walkthrough 0).

If you get an email message saying that the "Auto DevOps" pipeline ran and failed (as a result of your push to the repository), you can ignore it. If you don't like getting these emails you can follow the steps to turn off "Auto DevOops" at the end of GitLab Walkthrough 1.

We can now further investigate what has transpired by typeing `git log`. Try it! What you'll see is a log entry for the first version of your project. If you performed Step 0 correctly, you'll notice that the commit in the log entry is attributed to the Team Lead.

Conclusion

That's it for this walkthrough. In the next walkthrough we will take on the role of a developer who will do some work on the remote repository we just set up.