CSCC01 - Tutorial 1, using Git & GitHub

In this lab, we will go through a short session of using Git. We will talk about the internals of Git, and its terminology as we go along.

Task 1: Basics

Please sign in (if you do not already have an account, you might sign up at this moment). Create a new repository, check the README option, so you have a repository ready to clone. If you are using a laptop, make sure you have git installed on your system.

Make sure you have already created a work folder in your system. In what follows, we assume you are using `git` from the command line.

Clone your repository:

```
git clone REPO-URL
```

Check the status of your repo

```
git status
```

At this point, we have a clone (i.e. files and full history) of the remote repository on our local machine. Let's create a new file (in the working directory)

```
echo "Hi" > a.txt
```

Add the file (to the index/staging-area)

```
git add a.txt
```

Check the status of your repo, and see what's different

```
git status
```

Now that Git knows that we are interested in including `a.txt` in our next commit, let's commit the file (to the history)

```
git commit -m "Adding a.txt"
```

Check the status of your repo again. Let's stop for a second, and see what happened here:

- A repo is a graph of commit objects.
- A commit object is a snapshot of the codebase, with some extra metadata.
- When we run git commit, a new commit object gets created and added to the graph. An edge is created from this new commit to its parent (i.e. The most recent commit before we ran `git commit`).

At this point, `a.txt` is committed to your local repo (i.e. "the clone"), let's push our changes to the remote repository (i.e. "the origin"):

```
git push origin master
```

You're probably wondering what the origin master arguments are ...

- origin - The name of the repo we're pushing our changes to.

  - Git allows you to define remotes, a remote is a name that is associated with a URL of a remote repository.
  - When you clone a repo, Git automatically creates the origin remote, and associates it with the URL of the repository that you cloned.
- master - Which branch (of the remote repository) we're pushing the changes to. By default, a Git repo starts with a single branch, called master. We'll talk more about branches soon.

Notice that you can push changes to other repos, not just the one you cloned. Let's try that as follows:

- Fork (i.e. clone on GitHub) the remote repo, using a different GitHub account.
- Add a remote called other and set its URL to the URL of our forked repo.
- Try to push some changes to other.
- Notice: We will need to grant our user push permissions.

Summary

- We can clone a repo.
- We can work with a local clone (i.e. commit changes).
- When we are ready, we can push the changes to a remote repo.
  - The remote repo doesn't have to be the one we originally cloned - As long as you have push permissions, and the changes that you are pushing "make sense to Git", you're good.
  - If Git cannot safely push the changes, it will ask you to pull the changes from the remote repo, resolve any conflicts, commit your conflict resolution(s), and only then it will allow you to push the changes back.

Branching

So far, we've pushed changes between different repositories. That is, each repo was a single sequence of commit objects. Branching is used to work on different sequences in parallel (in a single same repo).

Let's create (and switch to) a new branch

```
git checkout -b feature1
```

We use the `checkout` command to switch between branches

```
git checkout master
git checkout feature1
```

Let's make some changes in our branch

```
echo "Bye" > b.txt
git add b.txt
git commit -m "Adding b.txt to the feature1 branch"
```

Notice that the changes exist only in the `feature1` branch

```
git checkout master
ls
git checkout feature1
ls
```

Modifying files

Let's modify `a.txt` in the `feature1` branch

```
echo "Modification 1" >> a.txt
git commit -m "Modifying a.txt in feature1 branch"
```

Oops, Git complains that there is nothing to commit (and give us a hint). Git doesn't assume you want to commit modified files by default. For those of you who are used to SVN, this behaviour may seem a little odd. We have two options now, we can either run `git add a.txt`, or we can run `git commit` with the `-a` option, which tells Git to add all (tracked) modified files to this commit.

```
git commit -a -m "Modifying a.txt in feature1 branch"
```

At this point `feature1` branch doesn't exist on the remote repo (origin), let's push it there

```
git push origin feature1
```

Merging & Resolving Conflicts

Next, let's switch back to the `master` branch, and make other modifications to `a.txt`

```
git checkout master
echo "Modification 2" >> a.txt
git commit -m "Modifying a.txt in master"
```

Now, let's try to merge `feature1` into `master` (we need to be on the master branch in order to do that, which we are).

```
git merge feature1
```

Since there are conflicts, Git "plays it safe" - You will see a message asking you to resolve conflicts, and commit the changes.

The working directory, index and history

This is a good time to dive a little deeper into how a Git repository works (and its terminology).
Each Git repo keeps tracks of three things:

- Working directory (aka working tree) - The files on your local system, at their current state.
- Index (aka Staging Area) - All the changes that will be committed with the next commit.
- History - The graph of commit objects that were committed.

Just to be clear - These 3 components are a part of a single repo, they have nothing to do with a remote repo. A remote repo has its own working directory, index and history. When you push changes to a remote repo, you're updating that repo's history.
Let's revisit the Git commands we've already used:

- We add changes from the working directory to the index.
- We commit changes from the index to the history.
- A branch is simply a pointer to a commit object in the history (i.e. a pointer to a node in the graph).
- `HEAD` is a pointer to (the most recent commit of) the current branch.

Undoing Things

There are multiple ways to undo things in Git - revert and reset are two different commands, the main difference between them is:

- `revert` creates a new commit object in the graph.
- `reset` doesn't create new commit objects, it changes the HEAD pointer (and optionally changes the working directory and/or index).

For this tutorial, we will only use `reset`.

- Use `git reset --soft COMMIT-ID` to move `HEAD` to the specified commit, without changing the index or the working directory.
- Use `git reset COMMIT-ID` to move `HEAD` to the specified commit, update the index, but not the working directory.
- Use `git reset --hard COMMIT-ID` to move `HEAD` to the specified commit, and update both the index and working directory.

Let's see an example ...

```
git log
```

With `git log` we can see the id's of all commit objects in the history. We will use the id of the most recent commit (i.e. `HEAD`) when we run `git reset`.

Let's commit some changes

```
touch c.txt
git add c.txt
git commit c.txt
git status
```

Let's start with

```
git reset --soft COMMIT-ID
git status
```

At this point, `HEAD` points to the most recent commit before `c.txt` was added. Notice that the index/staging-area did not change. That is, `c.txt` is staged, and ready to be committed.

```
git commit -m "Committing changes after a soft reset"
git status
```

Now, let's try the default `reset` behaviour.

```
git reset COMMIT-ID
git status
```

This time, the index changed as well, and `c.txt` is no longer staged. If we run `git commit` at this point, Git will complain that there is nothing to be committed. Let's add `c.txt` to the index again, and commit it.

```
git add c.txt
git commit -m "Adding c.txt again"
```

Now, let's try a hard reset.

```
git reset --hard COMMIT-ID
git status
ls
```

This time, `HEAD`, the index and the working directory were all updated. When we run `ls`, we see that `c.txt` is gone.

Summary

- Git's model is a little more complex than SVN. A single repository keeps track of 3 things
  - Working directory
  - Index
  - History
- `git add` submits changes from the working directory to the index.
- `git commit` submits changes from the index to the history.
- `git commit -a` add alls modified files from the working directory to the index, and commits them to the history.
- `git reset --soft` is used to undo changes in the history (but not the index or the working directory).
- `git reset` is used to undo changes in the history and the index (but not the working directory).
- `git reset --hard` is used to undo changes in the history, index and working directory.