Dr Kirill Sidorov

Cardiff University
School of Computer Science and Informatics

LAB EXERCISE FIVE

Version control using Git

Introduction

Soon you will begin working on your project *in teams*. When several people work on the same project concurrently, they need to somehow coordinate their joint efforts, as well as organise and control revisions they make to the shared code in a methodical and logical way. This is what *revision control* (or *version control*) systems are for[1].

There are several conceptual models for organising concurrent development of code. Here, we will consider the simplest one, the *centralised* model:

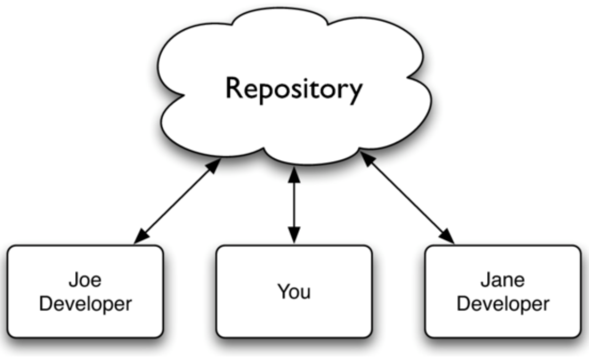

Figure 1: Centralised model of sharing code.

Your team will be storing your shared code in a central *repository* on a server. Each team member will also have their local copy of the code (a *working copy*) to which they will be making changes.

When a developer makes changes to their working copy, they *commit* them to the central repository[2] thus making the changes available to the other developers. Similarly, to retrieve the changes made by others, one *updates* their working copy from the central repository.[3]

The job of the revision control system is, in essence, to keep track of changes made by the developers and merge the changes together thus allowing for concurrent contributions to a common project.

[1]Revision control systems are useful even when only one developer is working on a project: when, for example, she uses multiple computers to contribute to the project, or when she wants to be able to roll back changes made to the project *etc*.

[2]In `Git`, as opposed to, say, SVN, each developer also has a full local copy of the repository. Performing a *commit* in `Git` means committing changes to the local repository; to make the changes available to other developers, they need to be *pushed* to a remote (central) repository.

[3]Similarly to the previous footnote, in `Git` the changes need to be first *fetched* from a remote repository into the local repository, before the working copy can be updated.

Exercises

This set of exercises will guide you through the basic usage of a version control system called `Git` for managing your code. This will help you coordinate your efforts as you work on your project in teams. Remember to ask our tutors for help if you get stuck!

Good luck!

1 `Git` is already installed on the lab machines. If you want to install `Git` on your computer, follow these instructions[4]:

- Get the installer from `http://git-scm.com/downloads` and run it.
- Choose a reasonable destination location.
- When asked *“How would you like to use Git from the command line?”*, choose *“Use Git from the Windows Command Prompt”*.
- When asked *“How should Git treat line endings in text files?”*, choose the default *“Checkout Windows-style, commit Unix-style line endings”*.

2 Make sure that `Git` works. Open the command prompt (terminal) and type[5]:

```
» git
```

You should see `Git`’s help message with a list of available commands.

3 You all have accounts on `https://git.cardiff.ac.uk/`, a `Git` server hosted in our school[6]. Please log in to your account.

4 Create a new repository, using the `https://git.cardiff.ac.uk/` web interface (*“New project”* in the top right corner). Choose `CM1101` for the project (repository) name. Leave all other settings as is. You have now created a blank remote repository.

5 Introduce yourself to `Git` using the following commands:

```
» git config --global user.name "YOUR NAME"
» git config --global user.email "YOUR EMAIL ADDRESS"
```

This way your commits will be properly labelled. The email you specify should be the same one you use to log in to GitLab.

6 Open the command prompt (terminal) and navigate to some folder in which you want to store the files for this exercise. Create a folder named `repos` in which you will be storing you local repositories. Go into this folder.

```
» mkdir repos
» cd repos
```

[4] These instructions are for Windows. On GNU/Linux the installation is usually trivial.

[5] Hereafter, the symbol » denotes the prompt. Do not actually type it.

[6] Many developers host their code on the popular public servers such as `http://github.com`. But for this exercise please use our in-house `https://git.cardiff.ac.uk/` since you can get technical support more easily.

7 Clone your GitLab repository to your local computer:

```
» git clone https://git.cardiff.ac.uk/username/CM1101.git .
```

(Replace `username` with your GitLab username.) A folder `CM1101` should appear, containing the (still blank) clone of your remote repository. Go into this folder:

```
» cd CM1101
```

8 Use the `git status` command to examine the situation (from inside the `CM1101` folder):

```
» git status
```

You should see something like:

```
On branch main

Initial commit

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

9 Create a file called `hello.py` in your `CM1101` folder with the following contents:

```
print("Hello")
```

Now run `git status` command again. You should see something like:

```
On branch main

Initial commit

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

        hello.py

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

Observe how `Git` now points out the file `hello.py` is not yet under version control.

10 Use the `git add` command to tell `Git` that it should now track this file:

```
» git add hello.py
```

You have now added `hello.py` to the *staging area*. Staging area stores information about what will go into your next commit (think of it as a loading dock where you get to determine what changes get shipped away). Issue the `git status` command again. What is the output? Note how `Git` gives you a hint as to what to do if you staged a file by mistake.

11 Commit your file to the (local) repository:

```
» git commit -m "Yay, I committed my first file!"
```

This command takes an argument `-m 'Some comment'` which allows you to document your changes. It is always a good idea to use descriptive comments, to indicate what changes you are committing, *e.g.* “Fixed a major bug in game logic”. You should see something like this:

```
[main (root-commit) 8014e92] Yay, I committed my first file!
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
```

Again, examine the situation with the `git status` command.

12 Create another file called `another.py` with the following code in it:

```
print("Another")
```

Modify your `hello.py` by adding a line to it:

```
print("Hello")
print("World")
```

Now you have a new (untracked) file `another.py` and a modified `hello.py`. Examine the situation with `git status`. Add the files in question to the staging area and commit all changes using the `git add` and `git commit` commands as before. Make sure that after you do this, `git status` reports that everything is fine and there are no changes to commit.
You can examine the history of your changes using the `git log` command; it would output something like this:

```
» git log

commit 745689304615bca2a57802eb0df0893319667d56
Author: Kirill Sidorov <k.sidorov@cs.cf.ac.uk>
Date:   Wed Oct 6 14:08:32 2014 +0100

    Added another and made changes to hello.

commit 8014e92aa580f715d296d7df48c7bc47da517574
Author: Kirill Sidorov <k.sidorov@cs.cf.ac.uk>
Date:   Wed Oct 6 13:57:43 2014 +0100

    Yay, I committed my first file!
```

13 Modify your `hello.py` to include a stupid error:

```
print("Hello")
print("World")
Stupid error
```

Check what `git status` reports. Suppose you are unhappy with your latest changes and want to revert them. Use the following command to update (revert) your working copy to the good version of the code which you now have in your local repository:

```
» git checkout -- hello.py
```

This should revert your `hello.py` to the state before the stupid error was introduced.

14 Now push your changes to the remote repository (which you have created on GitLab):

```
» git push origin main
```

`Git` will ask you for the password (the one you used for the GitLab account). Here, `origin` is the destination (remote name) and `main` is the branch name (you do not yet have to understand what this means). In subsequent pushes, you can omit the `origin main` arguments. The output should be something like this:

```
Password for 'https://username@git.cardiff.ac.uk': <enter your password>

Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (9/9), 824 bytes | 0 bytes/s, done.
Total 9 (delta 0), reused 0 (delta 0)
To https://username@git.cardiff.ac.uk/CM1101.git
 * [new branch]      main -> main
```

15 In your browser, navigate to the repository view on GitLab (again, replacing `username` accordingly):

```
https://git.cardiff.ac.uk/username/CM1101
```

Your files which you have pushed in the previous exercise should now be available through the web interface.

16 Let us now examine how multiple users can use the same remote repository. Create a folder called `another` somewhere outside the `CM1101` repository folder:

```
» cd ..
» mkdir another
» cd another
```

Now `git clone` the repository into this folder:

```
» git clone https://git.cardiff.ac.uk/username/CM1101.git .
```

Examine the contents of the folder. It should contain the exact copy of you original `CM1101` repository. This is how you would, for example, clone the repository created in the lab onto your home computer, or how your teammates may each clone your shared repository.

17 In the `another` copy of your repository, make some changes to `hello.py`:

```
print("Hello")
print("World")
print("More changes")
```

Add and commit your changes:

```
» git add hello.py
» git commit -m "Some more changes."
```

Use the `git push` command to push the changes to the central repository. Using GitLab's web interface, ensure that your changes have been successfully pushed (view the files on GitLab).

18 Navigate back to your *first* copy of the repository, `CM1101`. Use the following command to fetch the changes from the central repository and merge them with your working copy:

```
» git pull
```

What is the contents of `hello.py` now? It should now be synchronised with the changes you pushed from the `another` repository (via the central repository on GitLab).

19 In the `CM1101` folder make further changes to `hello.py` to look like this:

```
print("Hello")
print("World")
print("More changes")
print("Still more changes")
```

In the `another` folder make further changes to `hello.py` to look like this:

```
print("Hello")
print("Changes in another")
print("World")
print("More changes")
```

This simulates the situation when two developers have simultaneously made changes to their respective local repositories. Add, commit, and push the changes from `CM1101`. Then add, commit, and *try to* push changes from `another`. You should see an error like this:

```
» git push
Password for 'https://username@git.cardiff.ac.uk': <enter your password>

To https://username@git.cardiff.ac.uk/CM1101.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to
'https://username@git.cardiff.ac.uk/CM1101.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Aha! You need to pull the changes first, before pushing. The command (in `another` folder)

```
» git pull
```

should fetch the changes and merge them with the working copy. Examine the contents of `hello.py` now. Then execute

```
» git push
```

to finally push the changes you made in `another`. (You may verify the result of pushing by viewing your files on GitLab as before.)
Finally, in `CM1101` do

```
» git pull
```

to fetch and merge the changes previously pushed from `another`. Examine the contents of `hello.py`.

20 Repeat the previous exercise with a friend. First, add your friend to the list of collaborators on GitLab. To do so, in the menu on the left select select *Project* → *Settings* → *Members*, or simply go to

```
https://git.cardiff.ac.uk/username/CM1101/-/project_members
```

Add your friend as a collaborator. This will give them permissions to push changes to your repository. Now try making changes to your files concurrently with your friend, adding, committing, pushing, and pulling as appropriate.