Using GIT for software version management

2. Advanced Git Commands and Exercises

Author: Taco Walstra

University of Amsterdam

Faculty of Science

February 10, 2018

1

Graphical tools for Git

I recommend to use command line tools, but some people prefer to use GUI tools. Some possible tools are the following:

See http://git-scm.com/download/gui/

TortoiseGit is a counter part of TortoiseSVN (for Subversion) on Windows. It becomes part of your Explorer application and add some menus and changed icons on files in git directories. Be careful that you still add comments when checking in.

Git extensions is a graphical tool for Windows, Linux and Mac (not tested).

For a complete git environment you will also need a git server (github and bitbucket are not safe for academic research results). A standard solution for the faculty would be a number of Git server instances located on virtual servers. It is possible to host a webserver on such systems too which allows the installation of git-lab software. This will result in a complete copy of a bitbucket-like configuration. Costs are approximately 500 euro / year / installation.

2

A few important new commands

2.1 Moving and deleting files

Moving and deleting files from the repository is achieved in an easy way:

```
$git rm myfile
git commit -m "deleting file myfile"
```

Please note that if you delete a file using the standard 'rm' command a graphics application like explorer, git will notice that the file is deleted. You can check this with **git status**

2.2 Merging aspects

A **git pull** command basically consists of 2 components: a **git fetch** and a **git merge**. When the pull results in a clean merge, which is often the case if you work alone, this is fine of course. The difficult case when it results in conflicts. The problem with git pull is that it will change files when your perform the merge and it will not let you see what will change. It's therefor recommended to use a git fetch, so you can see exactly what files will be merged and how.

```
$git fetch
$git diff HEAD FETCH_HEAD
(this will display all changes which are to be merged)
diff --git a/another_version b/another_version
index be4fdef.c3b2f6 100644
--- a/another_version
+++ b/another_version
@@ -3,4 +3,4 @@ Tiny change
Small change 2
ABC DEF GHI JKL MNO PQR
ABC
-DEF
+ABC
```

What you can do in this case is change you local HEAD version to look like the FETCH_HEAD. So you edit first your local file and then perform git merge. This will not result in some unexpected changes you perhaps do not want. After your edit you peform

```
$git merge FETCH_HEAD
```

3

Forking your project

Often you want to work on a new part of an application. You could just continue with your **master branch** but it's often convenient to start your work on a new branch. The procedure is as follows:

```
$git branch
* master
$git branch new_dev
$git branch
* master
  new_dev
```

As you see the new branch has been created but we are still working in the active master branch. The new branch will contain the same content initially as the master branch. It looks as if new_dev is a copy but internally it's handled in a different way.

You move to your new branch by the following command:

```
$git checkout new_dev
$git branch
  master
* new_dev
```

You can of course also switch back to master and extend your files there.

4

Stashing your work

Suppose you are working on your 'new_dev" branch and you in the middle of some fancy coding, but not tested and certainly not finished. Now you get the question from somebody that your master branch has some serious flaw which needs fixing and of course not tomorrow. You could commit your changes into the new_branch ofcourse, but it will be a commit which is not working. Not something you should do normally. Git helps with this with the stash command: you put your work aside temporarily. The following example should give you an idea how stash works. Again, the $xxx are commands. Comments are between parentheses.

```
$git status
# On branch master
nothing to commit, working directory clean
$git checkout our_new_branch
(now we starting working in this new branch, let's say on file mars.txt.
In the middle of the work we need to do someting on the
master branch. We try to move directly to the master branch:)
$git checkout master
error: Your local changes to the following files would be overwritten
 by checkout:
jupiter
Please, commit your changes or stash them before you can switch branches.
Aborting
(this explains all. We could commit the changes using git commit -a -m"my intermedia
commit" but we can also stash it temporarily:)
$git stash
Saved working directory and index state WIP on
 another_fix_branch: 29c7e58 Renaming c and d.
HEAD is now at 29c7e58 Renaming c and d.
(Git stores the work in a commit, but it's reachable ONLY by the git stash command.
$git status
#On branch our_new_branch
nothing to commit, working directory clean
(As you see we moved to a clean situation of our branch, so we can also move to a di
$git checkout master
$git status
#On branch master
nothing to commit, working directory clean
(now to the changes to the master branch and move afterwards back to our_new_branch)
$git checkout our_new_branch
(You could have multiple stashes which can be listed:)
$git stash list
stash@{0}: WIP on new_dev: bcdd64f change to jupiter
(this shows that I worked on the item jupiter. So now we reapply our previous change
our_new_branch:
$git stash pop
# On branch new_dev
# 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:   saturn.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (47b320e2460550ac88f869623c17c5bd31aa8475)
(our stash list will be empty now)
```