The Road to Git

In order to master Git, 6 tasks have been prepared for demonstrating some features and problems that may occur when using version control and the features in Git. To make life a bit easier we will pretend that we are monitoring two different developers, named Petter and Sandy, who are working for the same company. This company has recently started a project called *HelloWorld*. However, the product manager (Webmäster) of the company would like to make some changes to *HelloWorld* and has put both Petter and Sandy to work on the project. Please work in groups with two computers, one representing Petter and the other on Sandy.

Try solving problems by following the instructions carefully and/or consult the following two documents for a list of common git commands and some useful tips and explanations.
https://github.com/I-sektionen/Workshops/blob/master/Git/Gitkommandon.pdf
https://github.com/I-sektionen/Workshops/blob/master/Git/Gittips.pdf

Task 1. Setting up a Git repository and importing a project

In this first task you will create a Git repository and add a few files to the *HelloWorld* project to the repository.

If you haven't generated a SSH-key from before, follow this guide:
https://github.com/I-sektionen/Workshops/blob/master/Git/GuideSSH-nycklar.pdf

If you haven't runned the following commands before open a terminal and run the following commands:
git config --global user.name "<first name> <surname>"
Change <first name> to your name and <surname> to your surname

git config --global user.email <liu-id>@student.liu.se
Change <liu-id> to your liu-id

Sandy can now go to https://gitlab.ida.liu.se/ and create the project "HelloWorld". Once the project is created, invite Petter as a fellow programmer. Make sure to assign Petter as a Master.
Before moving on to the next task, go to the filepage of the project on gitlab. Click on the "+"-sign → new file. Name the file ".gitignore" and write ".idea/" on the first text row. save the file.

Task 2. Checking out working copies to two different workspaces

In this task we will set up two workspaces, one for Petter and one for Sandy on their respective computers, and check out one working copy from the Git repository to each of the workspaces.

Open a terminal on the Petter's computer and run the following commands:

mkdir -p ~/webgroup_workshop/git_repo/Petter
This creates a folder for Petter.

cd ~/webgroup_workshop/git_repo/Petter
Navigate to Petter's folder

git clone git@gitlab.ida.liu.se:<liu-id>/HelloWorld.git
Change <liu-id> to liu-id of the person who created the project. Clone a working copy of the repository to Petter's folder.

Now open a terminal on the Sandy's computer, and follow the same procedure. However, now you will create a workspace for Sandy:

mkdir -p ~/webgroup_workshop/git_repo/Sandy
cd ~/webgroup_workshop/git_repo/Sandy
git clone git@gitlab.ida.liu.se:<liu-id>/HelloWorld.git

Open PyCharm on Petter's computer.
Go to File→ Open...
Choose /webgroup_workshop/git_repo/Petter/HelloWorld as your workspace. HelloWorld is now your current project.

Do the same for Sandy.

Task 3. Create the file HelloWorldFrame

Now that the environment has been set up, it is time to create a file to work with.

1. Petter creates the file HelloWorldFrame.py in PyCharm. (Right click the map and select new → Python file).
2. Petter writes a python program which prints "Hello World"
    a. Copy paste the following code:

```
# Hello world program
print("Hello world")
```

3. Petter now wants to add this file to git and push the file to master. Do this by entering the following commands in the terminal:
    a. **git status** - Checks for unstaged changes
    b. **git add HelloWorldFrame.py** - Puts the file "HelloWorldFrame.py" in the staging area.
    c. **git status**

    d. **git commit -m “enter commit message here”** - Commits the changes.
    e. **git pull** - Pulls down updates from the server.
    f. **git push** - Updates the server with your changes.
4. Now Sandy wants to bring her project up to date.
    a. **git status**
    b. **git pull**
    c. **git status**
5. Both Petter and Sandy can now use the command “**git log**” to see what has happened in the project.

Petter and Sandy should now both have the HelloWorldFrame.py in their project. As seen above checking “git status” often helps to see the status and how you should proceed. In the example above Petter has followed what could be said to be the standard work-flow when adding his changes to the project with the exception that he pushes straight to master without first creating a branch, more on branches in Task 6.

Task 4. Modify-Commit-Conflict that can be automatically resolved

Both Petter and Sandy will now begin their modifications of *HelloWorld*. Their next assignment is to add some comments in the file *HelloWorldFrame.py*. Although, it is the only file in the project, the strict Webmäster still wants to clarify that it is the main file.

1. Sandy starts by adding a comment at the **TOP** of the file *HelloWorldFrame.py*, clarifying that this is the main file;
2. *# This is the main file.*
3. Be sure that you are working with Sandy’s computer, add this comment and save the file!
4. Sandy then checks if there has been a new revision to the repository and performs an update. To do this run *git pull* in Sandy’s terminal. Has anything happened since the latest revision?
5. Switch to Petter’s computer.
6. Petter, who is lagging behind somewhat, adds a comment at the **BOTTOM** in HelloWorldFrame.py;
7. *# This file is surely the main file.*
8. Add this to his working copy of the file and save it. Petter also checks for new revisions. Do this! Has anything happened? Why/why not?
9. Switch to Sandy’s workspace.
10. Sandy now feels pretty satisfied and decides to commit her working copy to the repository.

Do this for her using the following commands:

git status
To check for unstaged changes, HelloWorldFrame.py Should be marked in red.

git add <file path>
This command puts the file in the staging area. Change <file path> to the unstaged file ex. HelloWorldFrame.py

git commit -m “Enter commit message here”
This commits the changes.

git pull

git push
This updates the server with your changes.

git log -2
This shows the last 2 commits.

11. Turn to Petter’s computer.
12. Petter also feels it is time to make a commit. Therefore commit the file to the repository using the same procedure as above. Remember to enter a commit comment.
13. When running **git pull** read the message and think about what happened.

Task 5. Modify-Commit-Conflict that must be manually resolved

Webmäster has asked Petter to change the “Hello World”-message to “Arbete stärker karaktären. Du är svag, jobba mer. -Webgroup”. However, due to a misunderstanding Sandy thought she was assigned to do this, but she thought the message should be changed to “Skriv mer kod, I-sektionen ser dig. -Webgroup”.

1. Switch to Sandy’s computer.
2. Make a **git pull** for Sandy so that she has the latest revision. Locate the section with the “Hello World”-message, and then change it. Save the file and run the application to make sure the text is changed. Then make a commit with a suitable commit message and push it to the server.
3. Switch to Petter’s computer.
4. **Without** first making an update with pull, help Petter change the text. Save the file and test the application! When you are done commit the file and then do a **git pull**.
5. You should now have a conflict that must be solved must solve the conflict in it and commit the merge!

Tip. on Petter’s computer open a text editor and look at the file (do not open it in PyCharm) and look for rows looking like these:
<<<<<<< HEAD
// This is your local version
=======
// This is the conflicting change from the server
>>>>>>> 9c421ebb4def402d2204d05301aec9b1b07e148a

There are now two ways of solving this:

1. You could do it from the text editor by replacing the <<<... ...===... …>>> with what it should be. Save the file and run **git add** on the file to mark the conflict as solved and **git commit** to mark that you are done merging. Lastly push the changes.

2. You could do it through PyCharm by right clicking on the file and click Git→Resolve Conflicts… and then merge. Select by clicking the arrows or X to solve the conflict. then commit and push the changes

Think about why you got the conflict and why this problem couldn't be solved like before.

Task 6. Branching

In order to avoid the earlier conflicts we will now introduce branching. Sandy is going to create a branch to add a new feature, *header,* while Petter is going to create another branch to add a another feature, *footer*. Different changes made in separate branches will not affect each other until you want to merge them to a master.
Now to break down the branching business. Do this:

git status
You should be up-to-date and on the master branch.

git branch [branch-name]
this creates a branch Sandy changes [branch-name] to header and Petter to footer.

git checkout [branch-name]
now your HEAD points to it and all changes made will only concern this branch

Sally can now add a comment "Header" on top of the file *HelloWorldFrame.py*
Petter can now add a comment "Bottom" on the bottom of the file *HelloWorldFrame.py*

git status
git add *HelloWorldFrame.py*
git commit -m "relevant message"
-m stands for message
git push

Now you want to update your branch with the newest version of master (in case someone has pushed new changes to master).

git checkout master
git pull
git checkout [branch-name]
git merge master

Resolve merge conflicts if there are any.
Now with your branch working with the newest version of master you will want to merge this new awesome feature to master.

git checkout master
git pull
git merge [branch-name]
git push
When the feature is done, end with deleting the feature branch:
git branch -d [branch-name]

For Petter's footer feature, you basically do the same thing but instead with another branch name. And if the changes in Petter's branch does not collide with that of Sandy's there should be no problem. **But** in this case Petter will have added the "footer" comment at the same place as Sandy's "header", totally unreasonable but true.

So try to recreate the scenario of branching, now for the footer feature, and add the comment "footer" at the top of the file. *Commit* and *push* Petter's branch.
Now Petter wants to update his feature with the newest version of master.

git checkout master
git pull
git checkout [branch-name]
git merge master

Finally Petter corrects the mistake and merges the corrected feature to master.

Now you are a master of Git. Git out of here.