Play this first

<http://pcottle.github.io/learnGitBranching/>

**Overview**

The play ground project is a good start for your Git development. You may learn Git command here, and experience the whole check in process including Git, Gerrit, Jenkins, Sonar and Nexus. You can also try working with your team member through user sandbox provided by Git. It is better to go through below content by yourself and understand the functionality of the commands before real development in formal projects.

Since Git command line is more clear then EGit Eclipse plugin, and EGit Eclipse plugin has some bugs and limitations, all Git related content is demostrated with Git command line, while you may still use Eclipse for file editing. After falimiar with Git command line, it is easy to leverage [EGit](https://wiki.wdf.sap.corp/wiki/display/LeanDI/Working+with+Git+and+Gerrit" \o "EGit) and switch to command line if necessary in real work.

**Clone the project**

1. Open Git bash shell, go to a local folder you would like use as Git source root (for example /c/git);
2. Run "**git clone ssh://<YOUR\_I\_NUMBER>@git.wdf.sap.corp:29418/com.sap.sbo.anw.playground.git playground**";
3. A folder name as "playground" <PG\_HOME> will be created in /c/git folder, and this is your local Git repository for play ground project;
4. "**cd playground**", "**git branch**". You will see a local branch called "master" is there and treated as default branch;
5. Run "**scp -p -P 29418 <YOUR\_I\_NUMBER>@git.wdf.sap.corp:hooks/commit-msg .git/hooks/**", this command will add the automatic change-id generation hook for you. Without this hook your change contains no necessary change-id, and your push action will be denied by Gerrit.
6. Open Eclipse, "File"->"Import"->"Maven"->"Existing Maven Projects", choose <PG\_HOME>, and you will be prompted to finish the import task; Git share is automatically done, which means you can already interact with Git for this project;
7. Right click the project, "Configure"->"Associate with Sonar" and click "Finish";

**Make a change**

As a good practise we do not use the default "master" branch for development, instead each time when we need to make changes for some purpose, we create a local branch directly from remote master branch. in this way we can prevent local branch dependency which is complex for remote merge.

1. "**git fetch origin**" to get latest changes from remote repository to local repository;
2. "**git checkout -b AddNewClass origin/master**", create a local branch called "AddNewClass" (can also name this branch with JIRA issue ID) from local copy of remote branch master, and switch to the branch;
3. "**git branch**", see current branch status that indicate we are in "AddNewClass" branch;
4. "**git branch -d master**", delete the local master branch as we do not use it;
5. Go to Eclipse to add some content, for example add a new class like com.sap.sbo.anw.playground.user.i053543.MyTestClass, also add test class in test source folder, and run test to ensure it is passed;
6. Run the test with test coverage (right click the test case, select "Coverage As"->"JUnit Test", and check out the coverage report to ensure your newly added content has been covered by the test case; This is important as sooner your check in will be denied if it leads to test coverage not meeting quality metrics; ***Notice: if you are using SAP JVM you may see some error logs when running coverage test, you may just delete them. And our projects are also configured to ignore the log files. If you do not like the generation of log files, install a Sun JVM (you may copy from <\\10.58.6.49\B1\_OnDemand\W3\Dev\_Softwares\jdk>) and set it as default JRE used by Eclipse projects***.
7. Right click the project, "SonarQube"->"Analyze". Follow the indication of "SonarQube Issues" view to fix all issues; This is important as sooner your check in will be denied if it leads to static code check result not meeting quality metrics. If you run into errors during local analysis, please open "Windows"->"Preferences"->"SonarQube"->"Preview Analysis Properties" and add a new property: "sonar.buildbreaker.skip"="true".
8. Go to Git bash shell and run "**git status**", you will see the files you add are marked as untracked files;
9. "**git add -A**" to add the new files to stage area, and "**git commit -m "My new class"**" to commit the changes to local repository;
10. Now run "**git status**" you will see no untracked files and your local branch is 1 commit ahead of remote master branch;
11. Run "**git push origin head:refs/for/master**" to push your change to remote branch. Caution here you cannot push to head:master as Gerrit will reject the direct push to remote master branch, instead you need to push to refs/for/master so that the code review and build verification can be made before Git accepting your change.
12. Go to [Gerrit](https://git.wdf.sap.corp:8080/" \o "Gerrit), sign in with your domain account, and go to "My"->"Changes"->"Outgoing reviews", you will see your change there. Click the change and add reviewers (your team member or anyone you think is related), the reviewers will receive mail for your review request. They can logon to Gerrit, view the change you made and add comments on the code snippet, and finally give a vote about the change.
13. In the meantime you will see a reviewer called "b1od-ci" adds comments to your change, and a build link is provided for you to track build status in our Jenkins server.
14. Jenkins server will try to build the project, run unit test and run static code check for the whole project and compare the result with quality metrics. If any step fails, Jenkins marks the change as "verification failed", and the change cannot be accepted without modification.
15. If any of a reviewer votes a "-2" review comment, or no reviewer votes a "+2" review comment, the change cannot be accepted without modification.
16. If build verification passes and reviewer gives a "+2" review, the change is accepted and merged to remote master branch automatically.
17. A new build will be triggerred in Jenkins to deploy the accepted build to Nexus for others usage.
18. In Git Bash shell run "**git fetch**", you will see the accepted change being synchronized to local repository.
19. Delete your local branch at anytime by "**git branch -d AddNewClass**" and create new branches for other development request.

**Deal with build failures and review failures**

Sometimes your change cannot pass build verification due to different reasons, or reviewer does not accept your changes. Here is how to procceed. Please try together with another colleague.

1. "**git fetch && git checkout -b mytest origin/master**" to get latest change and create a new branch from it;
2. Make a change to break the build (for example write a test case which will fail always);
3. "**git push origin head:refs/for/master**", push to Gerrit for review; invite your colleague to review your change and ask him to deny the change;
4. Make changes in the original branch to resolve the build failure;
5. "**git add -A**" to put all your fixes to stage area;
6. "**git commit --amend**" to modify your last commit and include the fixes;
7. "**git push origin head:refs/for/master**" to redo the changes; Gerrit will generate a new patchset within the original change, and trigger build verification with your latest changes. You may ask the colleague to go on review process with latest changes.
8. If conditions are all met the latest change will be accepted by Git.

Here the point is everytime a commit is created, the hook script we setup for the repository will generate a unique change-id and put the change id to commit message automatically. Later on when we run command "git commit --amend", the new changes will be applied to the last commit with exact same change-id, and Gerrit treats changes with same change-id as different patchsets for same change request.

That is why you should create different branch for different topics: when you need to apply new patchset to a commit, you can easily restore the commit history.

**Make parallel changes**

Sometimes you will work on different topics in parallel. With Perforce you need to carefully manage your changelist to prevent unwanted check-in. But things are much simpler with Git branch. You can easily create different local branch for different topics, and switch between them frequently.

Assume you are working on a local branch "Issue1" currently and need to work on another urgent hotfix immediately. But you do not want to include your current changes to the hot fix.

**Current change can be committed**

If your current work is done somehow, you can commit the changes. You can either push the changes or not at this stage. The important thing is the working directory has been cleaned with the commit command.

1. [On Issue1 branch] "**git commit -m "Fix issue1**"; commit the changes to local repository and clean up the working directory;
2. [On Issue1 branch] "**git fetch origin && git checkout -b Hotfix1 origin/master**"; it will create a new branch Hotfix1 from latest origin/master branch and switch to it;
3. [On Hotfix1 branch] Make some changes, run unit test to ensure the issue has been fixed, and run code analysis to fix all violations introduced;
4. [On Hotfix1 branch] "**git commit -m "Resolve the hotfix**"; commit the changes to local repository;
5. [On Hotfix1 branch] "**git push origin head:refs/for/master**"; push the hotfix for review;
6. After review and verification the Hotfix1 change has been accepted;
7. [On Hotfix1 branch] "**git fetch**"; get accepted change to local repository;
8. [On Hotfix1 branch] "**git status**"; ensure the hotfix1 has been finished without any issues;
9. [On Hotfix1 branch] "**git checkout Issue1**" and "**git branch -d Hotfix1**; switch to original Issue1 branch and delete Hotfix1 branch;
10. [On Issue1 branch] "**git pull**" to get latest changes from remote repository and merge to current local branch; Git automatically create a new commit for the merge, if conflicts do not exist you need to do nothing; if conflicts exist you need to manually resolve the conflicts.
11. [On Issue1 branch] "**git status**" to list files with conflicts; Resolve the conflicts in Eclipse;
12. [On Issue1 branch] "**git add -A**" to add resolved files to stage area;
13. [On Issue1 branch] "**git commit -m "merge changes"** to commit the merge result;
14. [On Issue1 branch] "**git push origin head:refs/for/master**" to push the change for Issue 1 for review;
15. Follow normal process to handle the change.

**Current change cannot be committed**

If your current work is not done yet, and you do not want to commit the changes, use "stash" command to temporarily store your current working directory and stage area, and makes the working directory cleaning for usage.

The only difference with previous section is the first command.

1. [On Issue1 branch] "**git stash**"; this command put all untracked files in working directory and stage area status to temporary storage and clean the working folder;
2. [On Issue1 branch] "**git status**"; currently the working directory is clean;
3. Now you can safely do steps 2 ~ 8 to fix the hotfix issue;
4. [On Hotfix1 branch] "**git checkout Issue1**", now you want to go on the change with Issue 1;
5. [On Issue1 branch] "**git stash list**" to see stash history;
6. [On Issue1 branch] "**git stash apply stash@{0}**" to restore the working status;
7. [On Issue1 branch] "**git stash drop stash@{0}**" to drop the temprary storage (optional);
8. Go on development for issue1

**Deal with conflicts due to others' changes**

Sometimes during your push and review, someone else's change has been accepted by Gerrit. In this case your change has to be rebased (no conflict) or merged and repushed (conflicts occurs) to make your push being procceeded. Please try together with another colleague.

1. Developer A creates a local branch from origin/master and made some changes;
2. Developer B creates a local branch from origin/master and made some changes as well;
3. Developer A submits his change and pushes to Gerrit for review;
4. Developer B submits and pushes to Gerrit for review as well;
5. Developer B's change is reviewed without problem, and verification passes as well, hence is merged to master branch;
6. Developer A's change is being reviewed:
   1. If there is no conflicts between developer B's change and developer A's change, reviewer can just click "Rebase" button to rebase developer A's change, and go on reviewing the change;
   2. If the rebase fails with verification error, or there are conflicts between developer A's change and developer B's change, developer A has to fetch latest change from master branch, merge with local changes, and repush the change to Gerrit again:
      1. [Developer A on his working branch] "**git fetch**", developer A get latest change from remote repository;
      2. [Developer A on his working branch] "**git rebase origin/master**", developer A rebases his changes to latest status of master branch;
      3. [Developer A on his wokring branch] "**git status**", developer A checks which files have conflicts;
      4. [Developer A on his working branch] developer A fix the conflicts by editing the conflict files;
      5. [Developer A on his working branch] "**git add -A**", developer A adds all resolved files to stage area;
      6. [Developer A on his working branch] "**git rebase --continue**", developer A goes on the rebase operation;
      7. [Developer A on his working branch] "**git push origin head:refs/for/master**", developer A pushes the new patchset to Gerrit for review;
      8. Developer B will be notified a new patch set is available for review
7. After review developer B thinks it is ok, and gives the change pass and submit the change to Git.
8. Developer A and developer B fetch latest change from remote repository, and delete local branch after confirming the changes have been accepted by Git.

**Cooperate with others**

Sometimes you would like to exchange code with teammate without really check in the changes to master branch, maybe because it is an experimental change, maybe because the change should be committed by the colleague together with his chanages. In this case the Gerrit sandbox branch is the right choice. Please try together with another colleague.

**Provide changes to others**

1. [On working branch] "**git fetch origin && git checkout -b MyTwoCents origin/master**", update local repository with latest remote status, and create a branch MyTwoCents for the change; Since you may not push the changes to master branch, it is better to have a separate branch for the purpose and you may delete the branch at convenience
2. [On MyTwoCents branch] make necessary changes and commit;
3. [On MyTwoCents branch] "**git push origin HEAD:refs/sandbox/<YOUR\_I\_NUMBER>/<ANY\_NAME\_YOU\_WANT>**". This command will push your local commit to your remote sandbox branch without triggering Gerrit review process; for example "git push origin HEAD:refs/sandbox/i053543/ForCid/Issue1";

**Get changes from others' sandbox branch**

1. [On working branch] "**git ls-remote | grep <KEYWORD>**" to query branch name; for example query by submitter id: "git ls-remote | grep i053543";
2. [On working branch] "**git fetch ssh://<YOUR\_I\_NUMBER>@git.wdf.sap.corp:29418/com.sap.sbo.anw.playground refs/sandbox/<PROVIDER\_I\_NUMBER>/<PATH\_TO\_THE\_CHANGE>"**. For example, "git fetch ssh://i053543@git.wdf.sap.corp:29418/com.sap.sbo.anw.playground refs/sandbox/i053543/ForCid/Issue1";
3. If you have local changes not committed, you may commit your local changes or stash them if you do not want wo commit yet.
4. [On working branch] "**git merge FETCH\_HEAD**" to merge the fetched sandbox changes to local branch;
5. [On working branch] "**git stash apply stash@{0}**" to restore stashed changes before merge
6. Continue your work. You can also commit the merged changes and push back to the sandbox branch for the provider's reference.

**Discard changes provided to others**

1. [On working branch] "**git push origin :refs/sandbox/<THE\_SANDBOX\_BRANCH\_TO\_DELETE**"; for example "git push origin :refs/sandbox/i053543/ForCid/Issue1"

**(Important) Squash changes before pushing to remote repository**

Our quality check is based on changes. Everytime you commit a change to local repository, it will also become a change in Gerrit when you push local repository to remote repository. This brings many overhead to central server, because for each change you push, you need to find someone to review it, and Jenkins server has to make a build for it. Also these changes are chained and have relationships in between, hence if part of them is accepted the Git history becomes chaos and you need to manully fix the problem. You would better do something locally to prevent such case.

You have two options to prevent too much changes:

**Use "stash" instead of "commit" to store local changes**

You can use stash command to store temprary status of your local working directory, and only commit once when everything is ready. For details please refer to [Git Stashing](http://git-scm.com/book/en/Git-Tools-Stashing" \o "Git Stashing) or "Make parallel changes" section above. But if you want to share codes with other you need to commit your change, in this case please use next option.

**Squash local changes before push**

Another option is squashing multiple local changes to one change, then pushing to remote repository. In this way your local commit history is modified, and you may decide which commit should be squashed or not. Here is an example.

1. [On your local branch] "**git status**", and it shows your local repository is ahead of origin/master for several commits, hence a squash is needed before push.
2. [On your local branch] "**git log origin/master -1 --pretty=oneline**", this is to get the latest commit id in remote repository, assume it is 16f3278a7d1d4c5cfd8231a200cb8ff4e471294a
3. [On your local branch] "**git rebase -i 16f3278a7d1d4c5cfd8231a200cb8ff4e471294a**", using interactive rebase to squash changes made from latest one in origin/master;
4. An editor will be opened to ask for your further action:
5. Change the operation command from "pick" to "squash" following below instructions, then save and exit the eidtor;
6. Then another editor will be displayed to ask you edit commit message, you may reuse original messages of the commits, or create a new messsage.
7. [On local branch] "**git log origin/master..HEAD**", After save and exit, Git will rebase and squash the selected changes into one change. You will see only one change difference between current branch and remote repository;
8. Push your change to Gerrit for upcoming review and verification stuff.

***Be carefule, never squash/rebase changes which has been available in remote repository, you should only do so for you local changes.***

**(Important) Best practises**

The flexibility of Git also brings complexity. To keep things simple, you are encouraged to follow below best practises:

1. Always create dedicated local branch from remote branch for each development topic: "git checkout -b NewTopic origin/master"
2. Always fetch latest remote repository status before create local branch, hence no merge is needed before pushing: "git fetch orgin && git checkout -b NewBranch origin/master";
3. Always squash local changes before pushing to remote;
4. Always fetch latest changes from remote repository and do necessary merge/rebase before pusing to remote: see section "deal with conflicts due to others' change";
5. Do not execute command If you are not sure what will happen, do look for helps from network/colleagues.

If you would like to use EGit instead of Git command line please have a look at [Working with Git and Gerrit](https://wiki.wdf.sap.corp/wiki/display/LeanDI/Working+with+Git+and+Gerrit" \o "Working with Git and Gerrit) and [EGit User Guide](http://wiki.eclipse.org/EGit/User_Guide" \o "EGit User Guide).