

NISSAN DevOps Training-Git

Pre-requisites:

- 1.) User should have GitSCM installed
- 2.) User should have GitHub Account
- 3.) GitHub account and a github repository for cloning.

Git Global Configuration

LAB - 1

Configure user information for all local repositories

The version of the Git can be checked by using the below command –

```
$ git --version
```

Add Git username and email address to identify the author while committing the information. Set the username by using the command as

```
$ git config --global user.name "USERNAME"
```

After entering username, verify the entered user name with the below command

```
$ git config --global user.name
```

Next, set the email address with the below command

```
$ git config --global user.email "email_address@example.com"
```

You can verify the entered email address as

```
$ git config --global user.email
```

Use the below command to check the entered information

```
$ git config --global --list
```

Cloning GIT Repositories

We have a bare repository on the Git Hub and User1 also pushed his first version. Now, User2 can view his changes. The Clone operation creates an instance of the remote repository. User2 creates a new directory in his home directory and performs the clone operation.

LAB - 2

```
$ mkdir <repo_name>
```

```
$ cd <repo_name>
```

```
$ git clone https://github.com/xxxxxx/xxxxxxxxxx.git
```

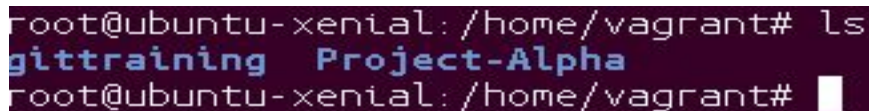
The above command will produce the following result.

Initialized empty Git repository in /home/user2/user2_repo/project/.git/

remote: Counting objects: 3, done.

Receiving objects: 100% (3/3), 241 bytes, done.

remote: Total 3 (delta 0), reused 0 (delta 0)

A terminal window screenshot with a dark purple background. The prompt is 'root@ubuntu-xenial: /home/vagrant#'. The command 'ls' has been executed, and the output is 'gittraining Project-Alpha'.

```
root@ubuntu-xenial: /home/vagrant# ls
gittraining Project-Alpha
root@ubuntu-xenial: /home/vagrant#
```

User2 changes the directory to new local repository and lists its directory contents.

```
$ cd <project- directory>
```

```
$ ls
```

```
README.md test.txt
```

GIT CHECKOUT

LAB -3

git-checkout - Switch branches or restore working tree files

To switch directories do

<i>\$ git branch test</i>	<i>(creating a branch test)</i>
<i>\$ git checkout test</i>	<i>(switching to the test branch)</i>
<i>\$ echo "hello world" > test.txt</i>	<i>(this will create a test.txt file</i>
<i>with a line "hello world")</i>	
<i>\$ git add test.txt</i>	<i>(this will add test.txt to git)</i>
<i>\$ git commit -m "Frist Commit"</i>	<i>(once changes are confirmed</i>
	<i>you can commit with</i>
<i>comments)</i>	
<i>\$ git push origin test</i>	

```
root@ubuntu-xenial: /home/vagrant/gittraining# git push origin test2
Username for 'https://github.com':
```

```
root@ubuntu-xenial:/home/vagrant/gittraining# git push origin test2
Username for 'https://github.com': arshad75
Password for 'https://arshad75@github.com':
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 271 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'test2' on GitHub by visiting:
remote:   https://github.com/arshad75/gittraining/pull/new/test2
remote:
To https://github.com/arshad75/gittraining.git
 * [new branch]      test2 -> test2
root@ubuntu-xenial:/home/vagrant/gittraining#
```

GIT PUSH

LAB - 4

User2 modified his last commit by using the amend operation and he is ready to push the changes. The Push operation stores data permanently to the Git repository. After a successful push operation, other developers can see user2's changes. He executes the git log command to view the commit details.

\$ git log

Once the user2 is happy with his changes and he is ready to push his changes.

\$ git push origin master

user2's changes have been successfully pushed to the repository; now other developers can view his changes by performing clone or update operation.

GIT STASHING

LAB - 5

Suppose you are implementing a new feature for your product. Your code is in progress and suddenly a customer escalation comes. Because of this, you have to keep aside your new feature work for a few hours. You cannot commit your partial code and also

cannot throw away your changes. So you need some temporary space, where you can store your partial changes and later on commit it.

In Git, the stash operation takes your modified tracked files, stages changes, and saves them on a stack of unfinished changes that you can re-apply at any time.

```
$ git branch                                ( make sure you are on the test branch)
  master
* test
$ echo "some changes" >> test.txt           ( make some changes on the test file)
$ git status -s
M test.txt                                ( M represents modified file)
```

Now, you want to switch branches for customer escalation, but you don't want to commit what you've been working on yet; so you'll stash the changes. To push a new stash onto your stack, run the **git stash** command.

```
$ git stash
Saved working directory and index state WIP on test: 87142de First Commit
HEAD is now at 87142de First Commit
```

Now, your working directory is clean and all the changes are saved on a stack. Let us verify it with the **git status** command.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Now you can safely switch the branch and work elsewhere. We can view a list of stashed changes by using the **git stash list** command.

```
$ git stash list
stash@{0}: WIP on test: 87142de First Commit
```

Suppose you have resolved the customer escalation and you are back on your new feature looking for your half-done code, just execute the **git stash pop command**, to remove the changes from the stack and place them in the current working directory.

```
$ git stash pop
```

```
On branch test
```

```
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: test.txt
```

Revert Uncommitted Changes

Let us suppose a user accidentally modifies a file from his local repository. But he wants to undo his modification. To handle this situation, we can use the **git checkout** command. We can use this command to revert the contents of a file.

LAB - 6

```
$ git status -s
```

```
M test.txt
```

```
$ git checkout test.txt
```

```
$ git status -s
```

Further, we can use the **git checkout** command to obtain a deleted file from the local repository. Let us suppose user deletes a file from the local repository and we want this file back. We can achieve this by using the same command.

```
$ ls -l
```

```
README.md
```

```
test.txt
```

```
$ rm test.txt
$ ls -1
README.md
$ git status -s
D test.txt
```

Git is showing the letter **D** before the filename. This indicates that the file has been deleted from the local repository.

```
$ git checkout test.txt
$ ls -1
Test.txt
```

GIT RESET

For traversing the commit history and rolling back to a previous state, you'll need the git reset hard command.

Git reset hard command example

To demonstrate how the `git reset hard` command works refer to Lab -7

LAB - 7

we need to create a bit of a local commit history in order to see the full power of the `git reset hard` command. To create that local commit history, simply create five HTML files using the `touch` command, and after each file is created, add the file to the Git index and issue a commit. The `git bash` commands for this are as follows:

```
$ git branch
$ git checkout test
$ touch a.html
```

```
$ git add . & git commit -m "Commit #1 - 1 file"
$ touch b.html
$ git add . & git commit -m "Commit #2 - 2 files"
$ touch c.html
$ git add . & git commit -m "Commit #3 - 3 files"
$ touch d.html
$ git add . & git commit -m "Commit #4 - 4 files"
$ touch e.html
$ git add . & git commit -m "Commit #5 - 5 files"
```

After you've issued the series of `touch` and `git commit` commands, a `git reflog` command will display the rich local commit history we have created.

```
$ git reflog
2e1dd0a (HEAD -> master) HEAD@{0}: Commit #5 - 5 files
868ca7e HEAD@{1}: commit: Commit #4 - 4 files
ebbbca3 HEAD@{2}: commit: Commit #3 - 3 files
882bf98 HEAD@{3}: commit: Commit #2 - 2 files
2f24f15 HEAD@{4}: commit (initial): Commit #1 - 1 file
```

The `git reset hard` command

Now let's imagine that everything after the third commit was bad, and we want to roll back the HEAD reference to the commit with an ID of `ebbbca3`. To do so, we would issue a `git reset --hard` command using that ID. By using the `--hard` switch, the `git reset` will not only change the HEAD reference, but it will also update all of the files in the index and the working directory as well. The full `git reset hard` command is as follows:


```
$ git reset --hard ebbbca3  
HEAD is now at ebbbca3 Commit #3 - 3 files
```

When we inspect the working tree, we will see that there are now only three files, as files *d.html* and *e.html* have been removed.

```
$ ls  
a.html  b.html  c.html
```

DAY 2

BRANCHES

LAB - 8

Create Branch

```
$ git branch <your branchname>
```

Note:- The branch name in this example is **New_branch**, you can create branch with any name.

```
$ git branch  
master  
new_branch  
* test
```

Switch between Branches

```
$ git checkout new_branch
Switched to branch 'new_branch'
```

```
$ git branch
  master
* new_branch
  Test
```

Shortcut to Create and Switch Branch

```
$ git checkout -b test_branch
Switched to a new branch 'test_branch'
```

```
$ git branch
  master
  new_branch
  test
* test_branch
```

Delete a Branch

```
$ git branch
  master
  new_branch
  test
* test_branch
```

```
$ git checkout master
Switched to branch 'master'
$ git branch -D test_branch
Deleted branch test_branch (was 5776472).
```

```
$ git branch
* master
  new_branch
  test
```

Rename a Branch

```
$ git branch
* master
  new_branch
  test
```

```
$ git branch -m new_branch new_branch1
```

```
$ git branch
* master
  new_branch1
  test
```

Merge Two Branches

User1 implements a change in his code and after testing, he commits and pushes his changes to the new branch.

Note that user1 is pushing these changes to the new branch, which is why he used the branch name **test** instead of **master** branch.

Now, if user2 wants the same functionality in the master branch instead of re-implementing, one can achieve this by merging user1's branch with the master branch.

LAB - 9

```
$ git checkout test
```

Add some test to test.txt as shown below

```
$ echo "New line" >> test.txt
```

```
$ git add *
```

```
$ git commit -m "Merge Commit"
```

```
$ git push origin test
```

```
$ git checkout master
```

```
$ ls
```

Here you will not see the same test.txt file. Now to merge the test_branch with the master branch do.

```
$ git merge origin/test
```

```
$ git push origin master
```

 (to push the changes to the master branch)

To verify the changes do

```
$ git log -1
```

Merge Conflicts - Handling Conflicts

User1 is working on the **test_branch** branch. He changes some text of a file and after testing, he commits his changes.

LAB - 10

```
# git branch
```

```
* master
```

```
Test_branch
```

```
$ git checkout test_branch
```

Switched to branch 'test_branch'

\$ ls

README.md test.txt

\$ echo "some changes" >> test.txt

\$ cat test.txt

hello world1

some changes

\$ git diff

diff --git a/test.txt b/test.txt

index 3ad7759..536bc2a 100644

--- a/test.txt

+++ b/test.txt

@@ -1 +1,2 @@

hello world1

+some changes

\$ git status -s

M test.txt

\$ git add test.txt

\$ git commit -m "added new line"

[test_branch 7f584b2] added new line

1 file changed, 1 insertion(+)

\$ git push origin test_branch

Username for 'https://github.com': xxxxxxxx

Password for 'https://xxxxxxx@github.com':

Counting objects: 3, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 301 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To <https://github.com/xxxxxxxx/xxxxxxxx.git>
ea6c2e9..7f584b2 test_branch -> test_branch

```
$ git diff
diff --git a/test.txt b/test.txt
index 3ad7759..bd27b46 100644
--- a/test.txt
+++ b/test.txt
@@ -1,2 @@
 hello world1
+adding new line
```

```
$ git add test.txt
$ git commit -m "adding new line"
$ git push origin master
$ git checkout test_branch
$ git pull origin test_branch
From https://github.com/xxxxxxxx/gittraining
* branch      test_branch -> FETCH_HEAD
Already up-to-date.
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

```
$ git pull origin test_branch
From https://github.com/xxxxxxxx/xxxxxxxx
* branch      test_branch -> FETCH_HEAD
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.
```

```

$ git diff
diff --cc test.txt
index bd27b46,536bc2a..0000000
--- a/test.txt
+++ b/test.txt
@@@ -1,2 -1,2 +1,6 @@@
  hello world1
++<<<<<< HEAD
  +adding new line
++=====
+ some changes
++>>>>>> 7f584b200317b4b2899c219816467f30cb4a6a65

```

```

$ git commit -a -m 'Resolved conflict'
$ git pull origin test_branch\

```

GitScm - SSH Key Setup

The SSH stands for *Secure Shell* or *Secure Socket Shell* used for managing the networks, operating systems and configurations and also authenticates to the GitHub account without using username and password each time. You can set the SSH keys to provide a reliable connection between the computer and GitHub.

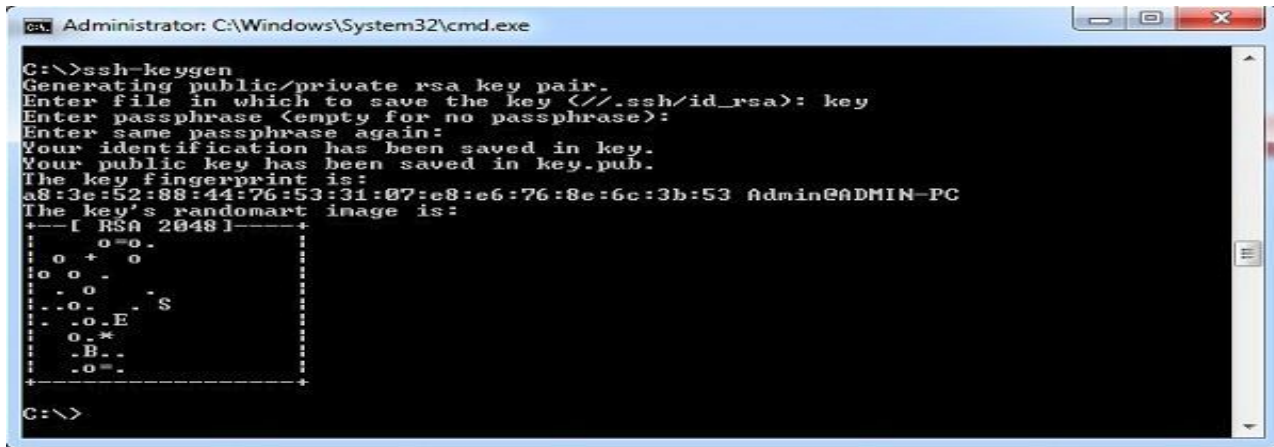
LAB - 11

Creating SSH Key

Step 1 – To create SSH key, open the command prompt and enter the command as shown below –

```
C:\-ssh-keygen
```

It will prompt for 'Enter file in which to save the key (//.ssh/id_rsa):', just type file name and press enter. Next a prompt to enter password shows 'Enter passphrase (empty for no passphrase):'. Enter some password and press enter. You will see the generated SSH key as shown in the below image



```
C:\>ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (//.ssh/id_rsa): key
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in key.
Your public key has been saved in key.pub.
The key fingerprint is:
a8:3e:52:88:44:76:53:31:07:e8:e6:76:8e:6c:3b:53 Admin@ADMIN-PC
The key's randomart image is:
+--[ RSA 2048 ]-----+
|      o+o      |
|    o + o     |
|  o o -      |
| ..o o  - S   |
|  .o.E       |
|   o.*        |
|    B.        |
|   o=         |
+-----+
C:\>
```

Cd to (//.ssh/id_rsa) and do

\$ cat id_rsa.pub

Copy the id_rsa.pub key

Login to GitHub account and go to the settings > SSH and GPG Keys.

Paste the id_rsa in the Keys section as shown in the below image.

Personal settings

- Profile
- Account
- Emails
- Notifications
- Billing
- SSH and GPG keys 2**
- Security
- Sessions
- Blocked users
- Repositories
- Organizations
- Saved replies
- Applications

SSH keys / Add new

Title

Mylaptop **3**

Key **4**

Begins with 'ssh-rsa', 'ssh-dss', 'ssh-ed25519', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', or 'ecdsa-sha2-nistp521'

Add SSH key

Signed in as arshad75

- Your profile
- Your repositories
- Your stars
- Your gists
- Help
- Settings 1**
- Sign out


Personal settings

- Profile
- Account
- Emails
- Notifications
- Billing
- SSH and GPG keys**
- Security
- Sessions
- Blocked users
- Repositories
- Organizations
- Saved replies
- Applications

SSH keys

New SSH key

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

	Mylaptop Fingerprint: 64:28:78:6a:e3:89:5c:87:ce:90:89:31:ce:dc:a9:62 Added on Oct 2, 2018 Never used — Read/write	Delete
---	--	--------

Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#).

GPG keys

New GPG key

There are no GPG keys associated with your account.

Learn how to [generate a GPG key and add it to your account](#).

GIT FORK VS CLONE

Fork, in the GitHub context, only allows clone on the server side.

When you clone a GitHub repo on your local workstation, you cannot contribute back to the upstream repo unless you are explicitly declared as "contributor". That's because your clone is a separate instance of that project. If you want to contribute to the project, you can use forking to do it, in the following way:

- clone that GitHub repo on your GitHub account (that is the "fork" part, a clone on the server side)
- contribute commits to that GitHub repo (it is in your own GitHub account, so you have every right to push to it)
- signal any interesting contribution back to the original GitHub repo (that is the "pull request" part)

If you want to keep a link with the original repo (also called upstream), you need to add a remote referring that original repo.

