**Git basics**

**Version Control System:**

version control system is a software that helps software developpers to work together and maintain a complete history of their work.

Listed below are the functions of a VCS:

* Allows developers to work simultaneously.
* Does not allow overwriting each other’s changes.
* Maintains a history of every version

Following are the types of VCS:

* Centralized version control system (CVCS).
* Distributed/Decentralized version control system (DVCS).

In this chapter, we will concentrate only on distributed version control system and especially on Git. Git falls under distributed version control system.

## **Distributed Version Control System**

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, distributed version control system (DVCS) comes into picture.

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

## Advantages of Git

### Free and open source

Git is released under GPL’s open source license. It is available freely over the internet. You can use Git to manage property projects without paying a single penny. As it is an open source, you can download its source code and also perform changes according to your requirements.

### Fast and small

As most of the operations are performed locally, it gives a huge benefit in terms of speed. Git does not rely on the central server; that is why, there is no need to interact with the remote server for every operation. The core part of Git is written in C, which avoids runtime overheads associated with other high-level languages. Though Git mirrors entire repository, the size of the data on the client side is small. This illustrates the efficiency of Git at compressing and storing data on the client side.

### Implicit backup

The chances of losing data are very rare when there are multiple copies of it. Data present on any client side mirrors the repository, hence it can be used in the event of a crash or disk corruption.

### Security

Git uses a common cryptographic hash function called secure hash function (SHA1), to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git.

### No need of powerful hardware

In case of CVCS, the central server needs to be powerful enough to serve requests of the entire team. For smaller teams, it is not an issue, but as the team size grows, the hardware limitations of the server can be a performance bottleneck. In case of DVCS, developers don’t interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.

### Easier branching

CVCS uses cheap copy mechanism, If we create a new branch, it will copy all the codes to the new branch, so it is time-consuming and not efficient. Also, deletion and merging of branches in CVCS is complicated and time-consuming. But branch management with Git is very simple. It takes only a few seconds to create, delete, and merge branches.

## DVCS Terminologies

### Local Repository

Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more.

### Working Directory and Staging Area or Index

The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit their changes directly to the repository. But Git uses a different strategy. Git doesn’t track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

Let us see the basic workflow of Git.

**Step 1** : You modify a file from the working directory.

**Step 2** : You add these files to the staging area.

**Step 3** : You perform commit operation that moves the files from the staging area. After push operation, it stores the changes permanently to the Git repository.

Suppose you modified two files, namely “sort.c” and “search.c” and you want two different commits for each operation. You can add one file in the staging area and do commit. After the first commit, repeat the same procedure for another file.

# First commit

[bash]$ git add sort.c

# adds file to the staging area

[bash]$ git commit –m “Added sort operation”

# Second commit

[bash]$ git add search.c

# adds file to the staging area

[bash]$ git commit –m “Added search operation”

### Blobs

Blob stands for **B**inary **L**arge **Ob**ject. Each version of a file is represented by blob. A blob holds the file data but doesn’t contain any metadata about the file. It is a binary file, and in Git database, it is named as SHA1 hash of that file. In Git, files are not addressed by names. Everything is content-addressed.

### Trees

Tree is an object, which represents a directory. It holds blobs as well as other sub-directories. A tree is a binary file that stores references to blobs and trees which are also named as **SHA1** hash of the tree object.

### Commits

Commit holds the current state of the repository. A commit is also named by **SHA1** hash. You can consider a commit object as a node of the linked list. Every commit object has a pointer to the parent commit object. From a given commit, you can traverse back by looking at the parent pointer to view the history of the commit. If a commit has multiple parent commits, then that particular commit has been created by merging two branches.

### Branches

Branches are used to create another line of development. By default, Git has a master branch, which is same as trunk in Subversion. Usually, a branch is created to work on a new feature. Once the feature is completed, it is merged back with the master branch and we delete the branch. Every branch is referenced by HEAD, which points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit.

### Tags

Tag assigns a meaningful name with a specific version in the repository. Tags are very similar to branches, but the difference is that tags are immutable. It means, tag is a branch, which nobody intends to modify. Once a tag is created for a particular commit, even if you create a new commit, it will not be updated. Usually, developers create tags for product releases.

### Clone

Clone operation creates the instance of the repository. Clone operation not only checks out the working copy, but it also mirrors the complete repository. Users can perform many operations with this local repository. The only time networking gets involved is when the repository instances are being synchronized.

### Pull

Pull operation copies the changes from a remote repository instance to a local one. The pull operation is used for synchronization between two repository instances. This is same as the update operation in Subversion.

### Push

Push operation copies changes from a local repository instance to a remote one. This is used to store the changes permanently into the Git repository. This is same as the commit operation in Subversion.

### HEAD

HEAD is a pointer, which always points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit. The heads of the branches are stored in **.git/refs/heads/** directory.

[CentOS]$ ls -1 .git/refs/heads/

master

[CentOS]$ cat .git/refs/heads/master

570837e7d58fa4bccd86cb575d884502188b0c49

### Revision

Revision represents the version of the source code. Revisions in Git are represented by commits. These commits are identified by **SHA1** secure hashes.

### URL

URL represents the location of the Git repository. Git URL is stored in config file.

[tom@CentOS tom\_repo]$ pwd

/home/tom/tom\_repo

[tom@CentOS tom\_repo]$ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

[remote "origin"]

url = gituser@git.server.com:project.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

Before you can use Git, you have to install and do some basic configuration changes. Below are the steps to install Git client on Ubuntu and Centos Linux.

## Installation of Git Client

If you are using Debian base GNU/Linux distribution, then **apt-get** command will do the needful.

[ubuntu ~]$ sudo apt-get install git-core

[sudo] password for ubuntu:

[ubuntu ~]$ git --version

git version 1.8.1.2

And if you are using RPM based GNU/Linux distribution, then use **yum** command as given.

[CentOS ~]$

su -

Password:

[CentOS ~]# yum -y install git-core

[CentOS ~]# git --version

git version 1.7.1

## Customize Git Environment

Git provides the git config tool, which allows you to set configuration variables. Git stores all global configurations in **.gitconfig** file, which is located in your home directory. To set these configuration values as global, add the **--global** option, and if you omit **--global** option, then your configurations are specific for the current Git repository.

You can also set up system wide configuration. Git stores these values in the **/etc/gitconfig** file, which contains the configuration for every user and repository on the system. To set these values, you must have the root rights and use the **--system** option.

When the above code is compiled and executed, it produces the following result:

### Setting username

This information is used by Git for each commit.

[jerry@CentOS project]$ git config --global user.name "Jerry Mouse"

### Setting email id

This information is used by Git for each commit.

[jerry@CentOS project]$ git config --global user.email "jerry@tutorialspoint.com"

### Avoid merge commits for pulling

You pull the latest changes from a remote repository, and if these changes are divergent, then by default Git creates merge commits. We can avoid this via following settings.

jerry@CentOS project]$ git config --global branch.autosetuprebase always

### Color highlighting

The following commands enable color highlighting for Git in the console.

[jerry@CentOS project]$ git config --global color.ui true

[jerry@CentOS project]$ git config --global color.status auto

[jerry@CentOS project]$ git config --global color.branch auto

### Setting default editor

By default, Git uses the system default editor, which is taken from the VISUAL or EDITOR environment variable. We can configure a different one by using git config.

[jerry@CentOS project]$ git config --global core.editor vim

### Setting default merge tool

Git does not provide a default merge tool for integrating conflicting changes into your working tree. We can set default merge tool by enabling following settings.

[jerry@CentOS project]$ git config --global merge.tool vimdiff

### Listing Git settings

To verify your Git settings of the local repository, use **git config –list** command as given below.

[jerry@CentOS ~]$ git config --list

The above command will produce the following result.

user.name=Jerry Mouse

user.email=jerry@tutorialspoint.com

push.default=nothing

branch.autosetuprebase=always

color.ui=true

color.status=auto

color.branch=auto

core.editor=vim

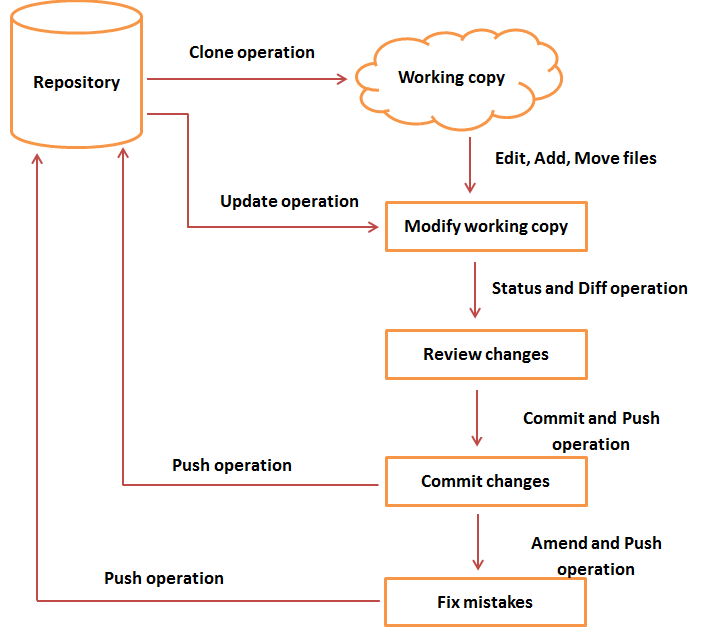
merge.tool=vimdiff

n this chapter, we will discuss the life cycle of Git. In later chapters, we will cover the Git commands for each operation.

General workflow is as follows:

* You clone the Git repository as a working copy.
* You modify the working copy by adding/editing files.
* If necessary, you also update the working copy by taking other developer's changes.
* You review the changes before commit.
* You commit changes. If everything is fine, then you push the changes to the repository.
* After committing, if you realize something is wrong, then you correct the last commit and push the changes to the repository.

Shown below is the pictorial representation of the work-flow.



In this chapter, we will see how to create a remote Git repository; from now on, we will refer to it as Git Server. We need a Git server to allow team collaboration.

## Create New User

# add new group

[root@CentOS ~]# groupadd dev

# add new user

[root@CentOS ~]# useradd -G devs -d /home/gituser -m -s /bin/bash gituser

# change password

[root@CentOS ~]# passwd gituser

The above command will produce the following result.

Changing password for user gituser.

New password:

Retype new password:

passwd: all authentication token updated successfully.

## Create a Bare Repository

Let us initialize a new repository by using **init** command followed by **--bare** option. It initializes the repository without a working directory. By convention, the bare repository must be named as **.git**.

[gituser@CentOS ~]$ pwd

/home/gituser

[gituser@CentOS ~]$ mkdir project.git

[gituser@CentOS ~]$ cd project.git/

[gituser@CentOS project.git]$ ls

[gituser@CentOS project.git]$ git --bare init

Initialized empty Git repository in /home/gituser-m/project.git/

[gituser@CentOS project.git]$ ls

branches config description HEAD hooks info objects refs

## Generate Public/Private RSA Key Pair

Let us walk through the process of configuring a Git server, **ssh-keygen** utility generates public/private RSA key pair, that we will use for user authentication.

Open a terminal and enter the following command and just press enter for each input. After successful completion, it will create a **.ssh** directory inside the home directory.

tom@CentOS ~]$ pwd

/home/tom

[tom@CentOS ~]$ ssh-keygen

The above command will produce the following result.

Generating public/private rsa key pair.

Enter file in which to save the key (/home/tom/.ssh/id\_rsa): **Press Enter Only**

Created directory '/home/tom/.ssh'.

Enter passphrase (empty for no passphrase): **---------------> Press Enter Only**

Enter same passphrase again: **------------------------------> Press Enter Only**

Your identification has been saved in /home/tom/.ssh/id\_rsa.

Your public key has been saved in /home/tom/.ssh/id\_rsa.pub.

The key fingerprint is:

df:93:8c:a1:b8:b7:67:69:3a:1f:65:e8:0e:e9:25:a1 tom@CentOS

The key's randomart image is:

+--[ RSA 2048]----+

| |

| |

| |

|

.

|

| Soo |

| o\*B. |

| E = \*.= |

| oo==. . |

| ..+Oo

|

+-----------------+

**ssh-keygen** has generated two keys, first one is private (i.e., id\_rsa) and the second one is public (i.e., id\_rsa.pub).

**Note:** Never share your PRIVATE KEY with others.

## Adding Keys to authorized\_keys

Suppose there are two developers working on a project, namely Tom and Jerry. Both users have generated public keys. Let us see how to use these keys for authentication.

Tom added his public key to the server by using **ssh-copy-id** command as given below:

[tom@CentOS ~]$ pwd

/home/tom

[tom@CentOS ~]$ ssh-copy-id -i ~/.ssh/id\_rsa.pub gituser@git.server.com

The above command will produce the following result.

gituser@git.server.com's password:

Now try logging into the machine, with "ssh 'gituser@git.server.com'", and check in:

.ssh/authorized\_keys

to make sure we haven't added extra keys that you weren't expecting.

Similarly, Jerry added his public key to the server by using ssh-copy-id command.

[jerry@CentOS ~]$ pwd

/home/jerry

[jerry@CentOS ~]$ ssh-copy-id -i ~/.ssh/id\_rsa gituser@git.server.com

The above command will produce the following result.

gituser@git.server.com's password:

Now try logging into the machine, with "ssh 'gituser@git.server.com'", and check in:

.ssh/authorized\_keys

to make sure we haven't added extra keys that you weren't expecting.

## Push Changes to the Repository

We have created a bare repository on the server and allowed access for two users. From now on, Tom and Jerry can push their changes to the repository by adding it as a remote.

Git init command creates **.git** directory to store metadata about the repository every time it reads the configuration from the **.git/config** file.

Tom creates a new directory, adds README file, and commits his change as initial commit. After commit, he verifies the commit message by running the **git log** command.

[tom@CentOS ~]$ pwd

/home/tom

[tom@CentOS ~]$ mkdir tom\_repo

[tom@CentOS ~]$ cd tom\_repo/

[tom@CentOS tom\_repo]$ git init

Initialized empty Git repository in /home/tom/tom\_repo/.git/

[tom@CentOS tom\_repo]$ echo 'TODO: Add contents for README' > README

[tom@CentOS tom\_repo]$ git status -s

?? README

[tom@CentOS tom\_repo]$ git add .

[tom@CentOS tom\_repo]$ git status -s

A README

[tom@CentOS tom\_repo]$ git commit -m 'Initial commit'

The above command will produce the following result.

[master (root-commit) 19ae206] Initial commit

1 files changed, 1 insertions(+), 0 deletions(-)

create mode 100644 README

Tom checks the log message by executing the git log command.

[tom@CentOS tom\_repo]$ git log

The above command will produce the following result.

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

Tom committed his changes to the local repository. Now, it’s time to push the changes to the remote repository. But before that, we have to add the repository as a remote, this is a one-time operation. After this, he can safely push the changes to the remote repository.

**Note:** By default, Git pushes only to matching branches: For every branch that exists on the local side, the remote side is updated if a branch with the same name already exists there. In our tutorials, every time we push changes to the **origin master** branch, use appropriate branch name according to your requirement.

[tom@CentOS tom\_repo]$ git remote add origin gituser@git.server.com:project.git

[tom@CentOS tom\_repo]$ git push origin master

The above command will produce the following result.

Counting objects: 3, done.

Writing objects: 100% (3/3), 242 bytes, done.

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

To gituser@git.server.com:project.git

\* [new branch]

master −> master

Now, the changes are successfully committed to the remote repository.

We have a bare repository on the Git server and Tom also pushed his first version. Now, Jerry can view his changes. The Clone operation creates an instance of the remote repository.

Jerry creates a new directory in his home directory and performs the clone operation.

[jerry@CentOS ~]$ mkdir jerry\_repo

[jerry@CentOS ~]$ cd jerry\_repo/

[jerry@CentOS jerry\_repo]$ git clone gituser@git.server.com:project.git

The above command will produce the following result.

Initialized empty Git repository in /home/jerry/jerry\_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)

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

[jerry@CentOS jerry\_repo]$ cd project/

[jerry@CentOS jerry\_repo]$ ls

README

erry clones the repository and decides to implement basic string operations. So he creates string.c file. After adding the contents, string.c will look like as follows:

#include <stdio.h>

int my\_strlen(char \*s)

{

char \*p = s;

while (\*p)

++p;

return (p - s);

}

int main(void)

{

int i;

char \*s[] =

{

"Git tutorials",

"Tutorials Point"

};

for (i = 0; i < 2; ++i)

printf("string lenght of %s = %d\n", s[i], my\_strlen(s[i]));

return 0;

}

He compiled and tested his code and everything is working fine. Now, he can safely add these changes to the repository.

Git add operation adds file to the staging area.

[jerry@CentOS project]$ git status -s

?? string

?? string.c

[jerry@CentOS project]$ git add string.c

Git is showing a question mark before file names. Obviously, these files are not a part of Git, and that is why Git does not know what to do with these files. That is why, Git is showing a question mark before file names.

Jerry has added the file to the stash area, git status command will show files present in the staging area.

[jerry@CentOS project]$ git status -s

A string.c

?? string

To commit the changes, he used the git commit command followed by –m option. If we omit –m option. Git will open a text editor where we can write multiline commit message.

[jerry@CentOS project]$ git commit -m 'Implemented my\_strlen function'

The above command will produce the following result:

[master cbe1249] Implemented my\_strlen function

1 files changed, 24 insertions(+), 0 deletions(-)

create mode 100644 string.c

After commit to view log details, he runs the git log command. It will display the information of all the commits with their commit ID, commit author, commit date and **SHA-1** hash of commit.

[jerry@CentOS project]$ git log

The above command will produce the following result:

commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:05:26 2013 +0530

Implemented my\_strlen function

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

After viewing the commit details, Jerry realizes that the string length cannot be negative, that’s why he decides to change the return type of my\_strlen function.

Jerry uses the **git log** command to view log details.

[jerry@CentOS project]$ git log

The above command will produce the following result.

commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:05:26 2013 +0530

Implemented my\_strlen function

Jerry uses the **git show** command to view the commit details. The git show command takes **SHA-1** commit ID as a parameter.

[jerry@CentOS project]$ git show cbe1249b140dad24b2c35b15cc7e26a6f02d2277

The above command will produce the following result:

commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:05:26 2013 +0530

Implemented my\_strlen function

diff --git a/string.c b/string.c

new file mode 100644

index 0000000..187afb9

--- /dev/null

+++ b/string.c

@@ -0,0 +1,24 @@

+#include <stdio.h>

+

+int my\_strlen(char \*s)

+{

+

char \*p = s;

+

+

while (\*p)

+ ++p;

+ return (p -s );

+

}

+

He changes the return type of the function from int to size\_t. After testing the code, he reviews his changes by running the **git diff** command.

[jerry@CentOS project]$ git diff

The above command will produce the following result:

diff --git a/string.c b/string.c

index 187afb9..7da2992 100644

--- a/string.c

+++ b/string.c

@@ -1,6 +1,6 @@

#include <stdio.h>

-int my\_strlen(char \*s)

+size\_t my\_strlen(char \*s)

{

char \*p = s;

@@ -18,7 +18,7 @@ int main(void)

};

for (i = 0; i < 2; ++i)

{

- printf("string lenght of %s = %d\n", s[i], my\_strlen(s[i]));

+ printf("string lenght of %s = %lu\n", s[i], my\_strlen(s[i]));

return 0;

}

## **Modify Existing Function**

Tom performs the clone operation and finds a new file string.c. He wants to know who added this file to the repository and for what purpose, so, he executes the **git log** command.

[tom@CentOS ~]$ git clone gituser@git.server.com:project.git

The above command will produce the following result:

Initialized empty Git repository in /home/tom/project/.git/

remote: Counting objects: 6, done.

remote: Compressing objects: 100% (4/4), done.

Receiving objects: 100% (6/6), 726 bytes, done.

remote: Total 6 (delta 0), reused 0 (delta 0)

The Clone operation will create a new directory inside the current working directory. He changes the directory to newly created directory and executes the **git log** command.

[tom@CentOS ~]$ cd project/

[tom@CentOS project]$ git log

The above command will produce the following result:

commit d1e19d316224cddc437e3ed34ec3c931ad803958

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my\_strlen to size\_t

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

After observing the log, he realizes that the file string.c was added by Jerry to implement basic string operations. He is curious about Jerry’s code. So he opens string.c in text editor and immediately finds a bug. In my\_strlen function, Jerry is not using a constant pointer. So, he decides to modify Jerry’s code. After modification, the code looks as follows:

[tom@CentOS project]$ git diff

The above command will produce the following result:

diff --git a/string.c b/string.c

index 7da2992..32489eb 100644

--- a/string.c

+++ b/string.c

@@ -1,8 +1,8 @@

#include <stdio.h>

-size\_t my\_strlen(char \*s)

+size\_t my\_strlen(const char \*s)

{

- char \*p = s;

+ const char \*p = s;

while (\*p)

++p;

}

After testing, he commits his change.

[tom@CentOS project]$ git status -s

M string.c

?? string

[tom@CentOS project]$ git add string.c

[tom@CentOS project]$ git commit -m 'Changed char pointer to const char pointer'

[master cea2c00] Changed char pointer to const char pointer

1 files changed, 2 insertions(+), 2 deletions(-)

[tom@CentOS project]$ git log

The above command will produce the following result:

commit cea2c000f53ba99508c5959e3e12fff493b

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 08:32:07 2013 +0530

Changed char pointer to const char pointer

commit d1e19d316224cddc437e3ed34ec3c931ad803958

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my\_strlen to size\_t

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

Tom uses git push command to push his changes.

[tom@CentOS project]$ git push origin master

The above command will produce the following result:

Counting objects: 5, done.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 336 bytes, done.

Total 3 (delta 1), reused 0 (delta 0)

To gituser@git.server.com:project.git

d1e19d3..cea2c00 master −> master

## Add New Function

Meanwhile, Jerry decides to implement **string compare** functionality. So he modifies string.c. After modification, the file looks as follows:

[jerry@CentOS project]$ git diff

The above command will produce the following result:

index 7da2992..bc864ed 100644

--- a/string.c

+++ b/string.c

30Git Tutorials

@@ -9,9 +9,20 @@ size\_t my\_strlen(char \*s)

return (p -s );

}

+char \*my\_strcpy(char \*t, char \*s)

+

{

+

char \*p = t;

+

+ while (\*t++ = \*s++)

+ ;

+

+

return p;

+

}

+

int main(void)

{

int i;

+

char p1[32];

char \*s[] =

{

"Git tutorials",

"Tutorials Point"

@@ -20,5 +31,7 @@ int main(void)

for (i = 0; i < 2; ++i)

printf("string lenght of %s = %lu\n", s[i], my\_strlen(s[i]));

+

printf("%s\n", my\_strcpy(p1, "Hello, World !!!"));

+

return 0;

}

}

After testing, he is ready to push his change.

[jerry@CentOS project]$ git status -s

M string.c

?? string

[jerry@CentOS project]$ git add string.c

[jerry@CentOS project]$ git commit -m "Added my\_strcpy function"

[master e944e5a] Added my\_strcpy function

1 files changed, 13 insertions(+), 0 deletions(-)

Before push operation, he verifies commit by viewing log messages.

[jerry@CentOS project]$ git log

The above command will produce the following result:

commit e944e5aab74b26e7447d3281b225309e4e59efcd

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:41:42 2013 +0530

Added my\_strcpy function

commit d1e19d316224cddc437e3ed34ec3c931ad803958

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my\_strlen to size\_t

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

Jerry is happy with the changes and he wants to push his changes.

[jerry@CentOS project]$ git push origin master

The above command will produce the following result:

To gituser@git.server.com:project.git

! [rejected]

master −> master (non-fast-forward)

error: failed to push some refs to 'gituser@git.server.com:project.git'

To prevent you from losing history, non-fast-forward updates were rejected

Merge the remote changes before pushing again. See the 'Note about

fast-forwards' section of 'git push --help' for details.

But Git is not allowing Jerry to push his changes. Because Git identified that remote repository and Jerry’s local repository are not in sync. Because of this, he can lose the history of the project. To avoid this mess, Git failed this operation. Now, Jerry has to first update the local repository and only thereafter, he can push his own changes.

**Fetch Latest Changes**

Jerry executes the git pull command to synchronize his local repository with the remote one.

[jerry@CentOS project]$ git pull

The above command will produce the following result:

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 3 (delta 1), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From git.server.com:project

d1e19d3..cea2c00 master −> origin/master

First, rewinding head to replay your work on top of it...

Applying: Added my\_strcpy function

After pull operation, Jerry checks the log messages and finds the details of Tom’s commit with commit ID **cea2c000f53ba99508c5959e3e12fff493ba6f69**

[jerry@CentOS project]$ git log

The above command will produce the following result:

commit e86f0621c2a3f68190bba633a9fe6c57c94f8e4f

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:41:42 2013 +0530

Added my\_strcpy function

commit cea2c000f53ba99508c5959e3e12fff493ba6f69

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 08:32:07 2013 +0530

Changed char pointer to const char pointer

commit d1e19d316224cddc437e3ed34ec3c931ad803958

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my\_strlen to size\_t

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

Now, Jerry’s local repository is fully synchronized with the remote repository. So he can safely push his changes.

[jerry@CentOS project]$ git push origin master

The above command will produce the following result:

Counting objects: 5, done.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 455 bytes, done.

Total 3 (delta 1), reused 0 (delta 0)

To gituser@git.server.com:project.git

cea2c00..e86f062 master −> master

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**VCS** are sometimes **known as** SCM (Source Code Management) **tools** or **RCS** (Revision Control **System**). **One** of **the most popular VCS tools** in use today is **called** Git. Git is a Distributed **VCS**, a category **known as** DVCS, **more** on that later. Like many of **the most popular VCS systems** available today, Git is free and open source

### #1) Git



Git is one of the best version control tools that is available in the present market.

Features

* Provides strong support for non-linear development.
* Distributed repository model.
* Compatible with existing systems and protocols like HTTP, FTP, ssh.
* Capable of efficiently handling small to large sized projects.
* Cryptographic authentication of history.
* Pluggable merge strategies.
* Toolkit-based design.
* Periodic explicit object packing.
* Garbage accumulates until collected.

Pros

* Super-fast and efficient performance.
* Cross-platform
* Code changes can be very easily and clearly tracked.
* Easily maintainable and robust.
* Offers an amazing command line utility known as git bash.
* Also offers GIT GUI where you can very quickly re-scan, state change, sign off, commit & push the code quickly with just a few clicks.

Cons

* Complex and bigger history log become difficult to understand.
* Does not support keyword expansion and timestamp preservation.

Open Source: Yes

Cost: Free

Click [here](https://git-scm.com/) for official Website.

### #2) CVS



It is yet another most popular revision control system. CVS has been the tool of choice for a long time.

Features

* Client-server repository model.
* Multiple developers might work on the same project parallelly.
* CVS client will keep the working copy of the file up-to-date and requires manual intervention only when an edit conflict occurs
* Keeps a historical snapshot of the project.
* Anonymous read access.
* ‘Update’ command to keep local copies up to date.
* Can uphold different branches of a project.
* Excludes symbolic links to avoid a security risk.
* Uses delta compression technique for efficient storage.

Pros

* Excellent cross-platform support.
* Robust and fully-featured command-line client permits powerful scripting
* Helpful support from vast CVS community
* allows good web browsing of the source code repository
* It’s a very old, well known & understood tool.
* Suits the collaborative nature of the open-source world splendidly.

Cons

* No integrity checking for source code repository.
* Does not support atomic check-outs and commits.
* Poor support for distributed source control.
* Does not support signed revisions and merge tracking.

Open Source: Yes

Cost: Free

Click [here](http://savannah.nongnu.org/projects/cvs) for official Website.

### #3) SVN



Apache Subversion, abbreviated as SVN aims at to be a best-matched successor to the widely used CVS tool that we just discussed above.

Features

* Client-server repository model. However, SVK permits SVN to have distributed branches.
* Directories are versioned.
* Copying, deleting, moving and renaming operations are also versioned.
* Supports atomic commits.
* Versioned symbolic links.
* Free-form versioned metadata.
* Space efficient binary diff storage.
* Branching is not dependent upon the file size and this is a cheap operation.
* Other features – merge tracking, full MIME support, path-based authorization, file locking, standalone server operation.

Pros

* Has a benefit of good GUI tools like TortoiseSVN.
* Supports empty directories.
* Have better windows support as compared to Git.
* Easy to set up and administer.
* Integrates well with Windows, leading IDE and Agile tools.

Cons

* Does not store the modification time of files.
* Does not deal well with filename normalization.
* Does not support signed revisions.

Open Source – Yes

Cost: Free

Click [here](https://subversion.apache.org/) for official Website.

### #4) Mercurial



Mercurial is a [distributed revision-control](https://en.wikipedia.org/wiki/Distributed_revision_control) tool which is written in python and intended for [software developers](https://en.wikipedia.org/wiki/Software_developer). The operating systems that it supports are Unix-like, Windows and macOS.

Features

* High performance and scalability.
* Advanced branching and merging capabilities.
* Fully distributed collaborative development.
* Decentralized
* Handles both plain text and binary files robustly.
* Possesses an integrated web interface.

Pros

* Fast and powerful
* Easy to learn
* Lightweight and portable.
* Conceptually simple

Cons

* All the add-ons must be written in Python.
* Partial checkouts are not allowed.
* Quite problematic when used with additional extensions..

Open Source: Yes

Cost: Free

Click [here](https://www.mercurial-scm.org/) for official Website.

### #5) Monotone



Monotone, written in C++, is a tool for distributed revision control. The OS that it supports includes [Unix](https://en.wikipedia.org/wiki/Unix), [Linux](https://en.wikipedia.org/wiki/Linux), [BSD](https://en.wikipedia.org/wiki/BSD), [Mac OS X](https://en.wikipedia.org/wiki/Mac_OS_X), and Windows.

Features

* Provides good support for internationalization and localization.
* Focuses on integrity over performance.
* Intended for distributed operations.
* Employs cryptographic primitives to track file revisions and authentications.
* Can import CVS projects.
* Uses a very efficient and robust custom protocol called netsync.

Pros

* Requires very low maintenance
* Good documentation
* Easy to learn
* Portable design
* Works great with branching and merging
* Stable GUI

Cons

* Performance issues observed for some operations, most visible was an initial pull.
* Can’t commit or checkout from behind the proxy (this is because of a non-HTTP protocol).

Open Source: Yes

Cost: Free

Click [here](https://www.monotone.ca/) for official Website.

### #6) Bazaar



Bazaar is a version control tool that is based on a distributed and client-server repository model. It provides cross-platform OS support and is written in Python 2, Pyrex and C.

Features

* It has commands similar to SVN or CVS.
* It allows you to be working with or without a central server.
* Provides free hosting services through the websites Launchpad and Sourceforge.
* Supports file names from the entire Unicode set.

Pros

* Directories tracking is supported very well in Bazaar (this feature is not there in tools like Git, Mercurial)
* Its plugin system is fairly easy to use.
* High storage efficiency and speed.

Cons

* Does not support partial checkout/clone.
* Does not provide timestamp preservation.

Open Source: Yes

Cost: Free

Click [here](http://bazaar.canonical.com/en/) for official Website.

### #7) TFS



TFS, an acronym for team foundation server is a version control product by Microsoft. It is based on client-server, distributed repository model and has a proprietary license. It provides Windows, cross-platform OS support through Visual Studio Team Services (VSTS).

Features

* Provides entire application lifecycle support including source code management, project management, reporting, automated builds, testing, release management and requirement management.
* Empowers DevOps capabilities.
* Can be used as a backend for several IDEs.
* Available in two different forms (on-premises and online (known as VSTS)).

Pros

* Easy administration. Familiar interfaces and tight integration with other Microsoft products.
* Allows continuous integration, the team builds and unit test integration.
* Great support for branching and merging operations.
* Custom check-in policies to aid in implementing a steady & stable codebase in your source control.

Cons

* Frequent merge conflicts.
* Connection to the central repository is always required.
* Quite slow in performing a pull, check-in, and branching operations.

Open Source: No

Cost: Free of cost for up to 5 users in the VSTS or for open source projects via codeplex.com; else paid and licensed through MSDN subscription or direct buy.

The server license can be bought for around $500 and the client licenses are also nearly the same.

Click [here](https://www.visualstudio.com/tfs/) for official Website.

### #8) VSTS



VSTS (Visual Studio Team Services) is a distributed, client-server repository model based version control tool provided by Microsoft. It follows the Merge or Lock concurrency model and provides cross-platform support.

Features

* Programming Language: C# & C++
* Changeset storage method.
* File and Tree scope of change.
* Network protocols supported: SOAP over HTTP or HTTPS, Ssh.
* VSTS offers elastic build capabilities thru build hosting in Microsoft Azure.
* DevOps enables

Pros

* All the features that are present in TFS are available in VSTS in the cloud.
* Supports almost any programming language.
* Instinctive User Interface
* Upgrades get automatically installed.
* Git access

Cons

* Signed revisions are not allowed.
* The “work” section is not very well optimized for large teams.

Open Source: No, it is a proprietary software. But, free trial version is available.

Cost: Free for up to 5 users. $30/mo for 10 users. Also offers a lot of free and paid extensions.

Click [here](https://www.visualstudio.com/team-services/) for official Website.

### #9) Perforce Helix Core

Helix Core is a Client-server and distributed revision control tool developed by Perforce Software Inc. It supports Unix-like, Windows and OS X platforms. This tool is mainly for large-scale development environments.

Features:

* Maintains a central database and a master repository for the file versions.
* Supports all file types and sizes.
* File-level asset management.
* Maintains a single source of truth.
* Flexible branching
* DevOps ready

Pros

* Git accessible
* Lightning fast
* Massively scalable
* Easy to track change list.
* Diff tools make it very easy to identify code changes.
* Works well with the visual studio through the plugin.

Cons

* Does not support interactive commits and external references.
* Managing multiple workspaces is quite difficult.
* Rollbacking changes is troublesome if its split across multiple change-lists.
* UI is quite confusing for new users.

Open Source: No, it’s a proprietary software. But, free trial version for 30 days is available.

Cost: Obtainable as perpetual license and subscriptions; charges differ based on configurations and options.

Click [here](https://www.perforce.com/products/helix-core) for official Website.

### #10) IBM Rational ClearCase



ClearCase by IBM Rational is a client-server repository model based on software configuration management tool. It supports a lot of Operating systems including [AIX](https://en.wikipedia.org/wiki/IBM_AIX),  Windows, [z/OS](https://en.wikipedia.org/wiki/Z/OS) (limited client), [HP-UX](https://en.wikipedia.org/wiki/HP-UX), Linux, [Linux on z Systems](https://en.wikipedia.org/wiki/Linux_on_z_Systems), [Solaris](https://en.wikipedia.org/wiki/Solaris_(operating_system)).

Features:

* Supports two models i.e UCM and base ClearCase.
* UCM stands for Unified Change Management and offers an out-of-the-box model.
* Base ClearCase offers basic infrastructure.
* Capable of handling huge binary files, a large number of files, and big repository sizes.
* Allows branching, labeling, and versioning of directories.

Pros

* Simple UI
* Integrates with Visual Studio.
* Handles parallel development.
* ClearCase Views are very convenient as they allow to switch between projects and configurations as opposed to local workstation model of the other version control tools.

Cons

* Slow recursive operations.
* Evil Twin problem – Here, two files with the same name get added to the location instead of versioning the same file.
* No advanced API

Open Source: No, it is a proprietary tool. But, free trial version is available.

Cost: $4600 for each floating license (detained automatically for a 30-minutes minimum for each user, can be surrendered manually)

Click [here](https://www.ibm.com/in-en/marketplace/rational-clearcase) for official Website.

### #11) Revision Control System

Revision Control system (RCS), developed by Thien-Thi Nguyen works on the local repository model and supports Unix-like platforms. RCS is a very old tool and was first released in 1982. It is an early version of VCS(Version Control System).

Features:

* Was originally intended for programs, but, is also helpful for text documents or config files that often get revised.
* RCS can be considered as a set of Unix Commands that permits various users to build and maintain program code or documents.
* Allows revision of documents, committing changes and merging docs together.
* Store revisions in a tree structure.

Pros

* Simple architecture
* Easy to work with
* It has local repository model, so the saving of revisions is independent of the central repository.

Cons

* Less security, version history is editable.
* At a time, only one user can work on the same file.

Open Source: Yes

Cost: Free

Click [here](https://www.gnu.org/software/rcs/) for official Website.

### #12) Visual SourceSafe(VSS)



VSS by Microsoft is a Shared folder repository model based revision control tool. It supports Windows OS only.

It is intended for small software development projects.

Features

* Creates a virtual library of computer files.
* Capable of handling any file type in its database.

Pros

* Fairly easy to use interface.
* It lets a single user system to be assembled with fewer configurations when compared to any other [SCM](https://en.wikipedia.org/wiki/Source_Code_Management) systems.
* Easy backup process.

Cons:

* Lacks many important features of a multi-user environment.
* Database corruption is one of the serious problems noted with this tool.

Cost: Paid. Nearly $500 for each license or single license which is comprised of every MSDN subscription.

Click [here](https://www.microsoft.com/en-in/download/details.aspx?id=291) for official Website.

### #13) CA Harvest Software Change Manager

This is a revision control tool provided by CA technologies. It supports many platforms including Microsoft Windows, Z-Linux, Linux, AIX, Solaris, Mac OS X.

Features

* Changes are made to a “change package”. Harvest supports both version control as well as change management.
* Has a pre-defined lifecycle from Test to Production stages.
* Fully customizable project environments. Project means ‘entire control framework’ in Harvest.

Open Source: No, this tool comes with Proprietary EULA License. However, a free trial is available.

Pros

* Helps very well in tracking the application flow from dev to prod environments. The biggest asset of this tool is this lifecycle feature.
* Deployment in a safe manner.
* Stable and scalable.

Cons

* Could be more user-friendly.
* Merging feature could be improved.
* Handling Polar Requests For Code Reviews Is challenging.

Cost: Not disclosed by the vendor.

Click [here](https://www.ca.com/us/products/ca-harvest-software-change-manager.html) for official Website.

### #14) PVCS



PVCS (an acronym for Polytron Version Control System), developed by Serena Software is a client-server repository model based version control tool. It supports Windows and Unix-like platforms. It provides version control of source code files. It is mainly intended for small development teams.

Features

* Follows locking approach to concurrency control.
* No built-in merge opera.tor but has a separate merge command.
* Supports multi-user environment.

Pros

* Easy to learn and use
* Manages the file versions regardless of the platforms.
* Gets easily integrated with Microsoft Visual Studio .NET and Eclipse IDEs.

Cons

* Its GUI has some quirks.

Open Source: No, it is a proprietary software.

Cost: Not disclosed by the vendor.

Click [here](https://www.microfocus.com/products/pvcs/) for official Website.

### #15) darcs



darcs (Darcs Advanced Revision Control System), developed by The Darcs team is a distributed version control tool that follows merge concurrency model. This tool is written in Haskell and supports Unix, Linux, [BSD](https://en.wikipedia.org/wiki/BSD), ApplemacOS, MS Windows platforms.

Features

* Capable of selecting which changes to accept from other repositories.
* Communicates with local and remote repositories through SSH, HTTP, email or unusually interactive interface.
* Works on the concept of linearly ordered patches.

Pros

* Has fewer and more interactive commands when compared to other tools like git and SVN.
* Offers send system for direct mailing.

Cons

* Performance issues related to merging operations.
* Installation takes a long time.

Open Source: Yes

Cost: This is a free tool.

Click [here](https://www.microfocus.com/products/pvcs/) for official Website.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Few more Version Control tools which are worth mentioning are:

#16) AccuRev SCM

AccuRev is a proprietary revision control tool developed by AccuRev, Inc. Its main features include streams and parallel development, private developer history, change packages, distributed development and automated merging.

Click [here](https://www.microfocus.com/products/change-management/accurev/) for official Website.

#17) Vault

Vault is a proprietary revision control tool developed by SourceGear LLC that works on CLI platform. This tool is the closest competitor to Microsoft’s Visual Source Safe. The backend database for Vault is Microsoft SQL Server. It supports atomic commits.

Click [here](http://www.sourcegear.com/vault/) for official Website.

#18) GNU arch

GNU arch is a distributed and decentralized revision control tool. It is a free and open source tool. This tool is written in C language and supports GNU/Linux, Windows, Mac OS X Operating systems.

Click [here](https://www.gnu.org/software/gnu-arch/) for official Website.

#19) Plastic SCM

Plastic SCM is a proprietary version control tool that works on.NET/Mono platform. It follows a distributed repository model. The Operating systems that it supports include Microsoft Windows, Linux, Solaris, Mac OS X. It comprises of a command-line tool, a Graphical User Interface, and integration with numerous IDEs.

This tool deals with big projects excellently.

Click [here](https://www.plasticscm.com/) for official Website.

#20) Code Co-op

Code Co-op, developed by Reliable Software is a peer to peer revision control tool. It follows distributed, peer to peer architecture where it creates a replica of its own database on every machine involved in the shared project. One of its interesting distinguishing features is its inbuilt wiki system for documentation.

Click [here](http://relisoft.com/co_op/) for official Website.

Conclusion

In this article, we discussed the best version control software. As we have seen, each tool has its own distinguishing features, pros, and cons. Few of them were open source tools while others were paid. Some suit small enterprise model well while the others suit big enterprise.

So, you need to choose the right tool as per your requirements, after weighing their pros and cons. For paid tools, I would suggest you to first explore their free trial versions before you buy.

\*\*\*\*\*\*\*\*\*\*  
=>> [Contact us to suggest listing here](https://www.softwaretestinghelp.com/contact/)  
\*\*\*\*\*\*\*\*\*\*

**Join Over 200,000+ Testers**

**Get premium ebooks and testing tips.**

## Top FREE Training Tutorials

* [QA Training](https://www.softwaretestinghelp.com/free-online-software-testing-qa-training-course/)
* [Selenium Tutorials](https://www.softwaretestinghelp.com/selenium-tutorial-1/)
* [QTP Tutorials](https://www.softwaretestinghelp.com/qtp-quicktest-professional-tutorial-1/)
* [ISTQB Study Guide](https://www.softwaretestinghelp.com/istqb-free-updates/)
* [ISTQB Premium Study Guide](http://www.softwaretestinghelp.org/istqb-certification-complete-study-package/)
* [QC Tutorials](https://www.softwaretestinghelp.com/learn-hp-quality-center-qc-in-4-days/)
* [HP LoadRunner Tutorials](https://www.softwaretestinghelp.com/hp-loadrunner-load-testing-tool-training-tutorials/)
* [JMeter Tutorials](https://www.softwaretestinghelp.com/jmeter-tutorials/)
* [JIRA Tutorials](https://www.softwaretestinghelp.com/atlassian-jira-tutorial-1/)
* [VBScript Tutorials](https://www.softwaretestinghelp.com/vbscript-tutorial-1/)
* [Best Test Management Tools](https://www.softwaretestinghelp.com/15-best-test-management-tools-for-software-testers/)
* [Appium Studio Tutorials](https://www.softwaretestinghelp.com/appium-studio-tutorial/)
* [Unix Tutorials](https://www.softwaretestinghelp.com/unix-tutorials/)
* [Free DevOps Training](https://www.softwaretestinghelp.com/devops-tutorials/)
* [Free JAVA Tutorials](https://www.softwaretestinghelp.com/java/)
* [Free Python Tutorials](https://www.softwaretestinghelp.com/python/)
* [101+ Interview Questions](https://www.softwaretestinghelp.com/software-testing-interview-questions/)

## Adv







**GitLab, GitHub:**

Git - A source code versioning system that lets you locally track changes and push or pull changes from remote resources

* GitLab, GitHub, and Bitbucket - Are services that provides remote access to Git repositories. In addition to hosting your code, the services provide additional features designed to help manage the software development lifecycle. These additional features include managing the sharing of code between different people, bug tracking, wiki space and other tools for 'social coding'.
* [GitHub](https://www.github.com/) is a publicly available, free service which requires all code (unless you have a paid account) be made open. Anyone can see code you push to GitHub and offer suggestions for improvement. GitHub currently hosts the source code for tens of thousands of open source projects.
* [GitLab](https://www.gitlab.com/) is a github like service that organizations can use to provide internal management of git repositories. SESYNC has setup a GitLab server for our researchers and staff to better collaborate