OUTLINE

• Version Control System
• VCS Types & Goals
• VCS software tools
• Git History & Features
• Git -- Installation & Configuration
• Git Repository and States
• Git data Integrity
• Some more git concepts....(Pulling vs cloning Repository, Branching, Forking, Merging, Rebase)
• GitLab (User types, Permissions and roles, namespace)
A quick recap...
LTAT.06.015 : Lec-03 : Version Control System (VCS)

Software Development Life Cycle (SDLC)

Why DevOps?

Disadvantages:

Developers team

Developed product

Product deployment

Operation team

Send product back to developer

Failed Product

Production environment

Why are we jumping to Cloud computing now?

4% of respondents currently don’t use a cloud provider:
- DevSecOps Survey 2022 by GitLab

21% of respondents indicated not using the cloud, and instead used a data center or on-premises solution:
- State of DevOps 2021 by Google

DevOps Phases

CODE

MONITORING

BUILD

OPERATE

TEST

RELEASE

DEPLOY

Continuous Integration

Develop Code → Build → Unit Test → Deploy to Stage → Functional Test → Deploy to PROD

Continuous Delivery

Develop Code → Build → Unit Test → Deploy to Stage → Functional Test → Deploy to PROD

Continuous Deployment

Develop Code → Build → Unit Test → Deploy to Stage → Functional Test → Deploy to PROD

Img src: https://medium.com/ibm-garage/devops- adoption-approach-plan-and-design-be3d1ba67c8
DevOps Phases

LTAT.06.015 : Lec-02 : Introduction to DevOps Part II + Cloud Computing
Version Control System
Version Control System - Introduction

**LTAT.06.015 : Lec-03 : Version Control System (VCS)**

**File: Help.html**

- **V1:** Hello from DevOps course PI.
- **V2:** Hello from DevOps course PI.
- **V3:** Hello from DevOps course PI.

---

**Local Computer**

- **Checkout**
  - **File**
    - **Version 3**
    - **Version 2**
    - **Version 1**

---

Version Control System - Introduction

“Version control, also known as source control, is the practice of tracking and managing changes to software code.”

“Version control systems are software tools that help software teams manage changes to source code over time.”

- Bitbucket

Create a file, edit the file, save the file, modify the file, delete the file.....and many more operations

Using a VCS also generally means that if you lose files, you can easily recover

- a.k.a.
  - Source Code Management (SCM)
  - Revision Control System (RCS)
But why to manage changes?

“Version control systems are software tools that help software teams manage changes to source code over time.”

- Bitbucket

Answering *Who, Why, What, When, for Whom* ….

Some benefits of such VCS
- Concurrency (e.g. Google Doc vs working on same docx file through email)
- Preserving collaborative history
- Rollback and undo changes
- Offsite source code backup
- Branching and merging
- Traceability
- Conflict resolution
VCS Goals

Some goals of version control system:

• Speed
• Simple design
• Strong support for non-linear development (thousands of parallel branches)
• Fully distributed
• Able to handle large projects like the Linux kernel efficiently (speed and data size)
• Data integrity

VCS types

Local Version Control Systems

Centralized Version Control Systems

Distributed Version Control Systems

Src: https://www.htown-tech.com/blogs/centralized-vs-distributed-version-control-system
VCS types

Centralized Version Control Systems

- **Location**: central server
- Need to be online, not locally available
- Need to connect to the central server for every command
- Single point of failure
- Rely on the backups

Distributed Version Control Systems

- **Location**: Distributed in nature with local copy of entire history
- Not necessary to be online
- Only required during pushing or pulling the code
- Better performance
- Local backup, if remote crashes
- Remote backup, if local crashes

Src: https://www.htown-tech.com/blogs/centralized-vs-distributed-version-control-system
# VCS software tools

<table>
<thead>
<tr>
<th>Software</th>
<th>Network architecture</th>
<th>Conflict resolution</th>
<th>Development status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Git</td>
<td>Distributed</td>
<td>Merge</td>
<td>Active</td>
</tr>
<tr>
<td>Mercurial</td>
<td>Distributed</td>
<td>Merge</td>
<td>Active</td>
</tr>
<tr>
<td>SVN</td>
<td>Client-server</td>
<td>Merge or lock</td>
<td>Active</td>
</tr>
<tr>
<td>CVS</td>
<td>Client-server</td>
<td>Merge</td>
<td>Maintenance only</td>
</tr>
</tbody>
</table>

Some other alternatives:

* Azure DevOps Server, Helix Core, Subversion, AWS CodeCommit, Rational ClearCase, Micro Focus AccuRev, etc.*
Git

- A good implementation of above VCS idea
A bit of history…

A BRIEF HISTORY OF
VERSION CONTROL

Early 1960s
IEBUPDTE for IBM OS/360
(punched card system)

1972
SCCS (Source Code Control System) created by Marc Rochkind

1982
RCS (Revision Control System) created by Walter Tichy

1986
CVS (Concurrent Versions Systems) created by Dick Grune

2000
Subversion (SVN) created by CollabNet

2005
Git created by Linus Torvalds

Img src: https://blog.tarynmcmillan.com/a-history-of-version-control
A bit of history…

What's the difference between git, GitHub, and GitLab?
Git - Features

• Distributed Version Control System
• Great branching model that adapts well to most workflows
• Fast and reliable
• Keeps a complete history
• Disaster recovery friendly
• Open Source

Src: https://gitlab-org.gitlab.io/
Git -- Installation

- Windows OS:
  - [https://git-scm.com/download/win](https://git-scm.com/download/win)

- macOS:
  - brew install git
  - ref: [https://git-scm.com/download/mac](https://git-scm.com/download/mac)

- Linux and Unix
  - Ubuntu:
    - apt-get install git
  - CentOS:
    - yum install git
  - Ref: [https://git-scm.com/download/linux](https://git-scm.com/download/linux)

- git version with `git --version` command
- Enter `git` to see the list of available options/commands
- For any help enter `git help <command name>`
  - `git help init`
Git -- Configuration

Project level
- only available for the current project
- stored in .git/config in the project's directory

Global level
- Available for all projects for the current user
- Stored in ~/.gitconfig

System level
- Available for all the users/projects
- Stored in /etc/gitconfig

Src: https://gist.github.com/lifuzu/9490352
Git – Configuration commands

Project level
- only available for the current project
- stored in .git/config in the project's directory

Global level
- Available for all projects for the current user
- Stored in ~/.gitconfig

System level
- Available for all the users/projects
- Stored in /etc/gitconfig

Sample command:
git config user.name "Chinmaya Dehury"

Override
Git – Configuration commands

Some more configuration related commands

• list all
  • git config –l

• open an editor to edit the config file
  • git config –e

• Set the value of a specific key:
  • For local: git config user.name “Chinmaya Dehury”
  • For Global level: git config --global user.name “Chinmaya Dehury"
  • For system level: git config --system user.name “Chinmaya Dehury"

• get value: name [value-pattern]
  • git config <key>
Distributed Version Control Systems

Main Server
- e.g. https://gitlab.cs.ut.ee

Local
- Repository
  - Push
  - Pull

Workstation
- Repository
  - Commit
  - Update
  - Working copy #1
  - Working copy #2
  - Working copy #3

- e.g. https://gitlab.cs.ut.ee/dehury/DevOps22FallPub
- e.g. D:/gitlab/DevOps22FallPub

Git Repository
Git Repository

Distributed Version Control Systems

- virtual storage of your project
- Save your code
- Keep meta info
  - What you change
  - Change time
  - Who changed
  - Etc....
- New repo can be created
  - on Remote server
  - Or Local storage system
  - Local: use `git init` command
  - Remote: using GUI
- Cloning an existing repository
  - Using `git clone <repo url>`
  - `git clone`
  https://gitlab.cs.ut.ee/dehury/firstrepo
Modified means that you have changed the file but have not committed it to your database yet.

Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

Committed means that the data is safely stored in your local database.
Git states - Staging

Create a new repository:

```
mkdir testRepo
cd testRepo/
git init
```

Check the status:

```
git status
```

Check the status of the repo:

```
git status
```

Create a new file with some random text:

```
echo "this is the first change." > readme.md
```

Let's have a quick walkthrough...
Git states - Staging

Adding the changes to staging area

- 12/19/2021 15:54:14 /mnt/d/testRepo master git add .

- 12/19/2021 15:57:17 /mnt/d/testRepo master git status

No commits yet

Changes to be committed:
(use "git rm --cached <file>..." to unstage)

new file: readme.md

Lets have a quick walkthrough
Git states - Commit

Let's have a quick walkthrough

Committing the changes

```
$ 12/10/2021  16:01.22  /mnt/d/testRepo P master  git commit -m "first change."
[master (root-commit) lelidaa] first change.
1 file changed, 1 insertion(+)
create mode 160644 readme.md
```

Let's again see the status

```
$ 12/16/2021  16:35.10  /mnt/d/testRepo P master  git status
On branch master
nothing to commit, working tree clean
```
Git states – modify and Commit again

Let's have a quick walkthrough

First commit
Second commit

Committing the changes

The second commit stores a pointer to the first commit.
Git logs – Commit history

Let’s see the commits

```bash
$ git log
commit 1e11daad0cbb5c074ec5433df8c9a066393ea318 (HEAD -> master)
Author: chinmaya dehury <chinmaya.dehury@ut.ee>
Date:   Tue Oct 12 16:01:28 2021 +0300

  first change..
```

Will discuss in details in later slides
Branching
Branching

Branch

• A branch is a version of a project’s working tree.
• diverge from the main line of development and continue to do work without messing with that main line.
• The default branch name in Git is master
• A branch in Git is simply a lightweight movable pointer to one of the commits.
• The master branch always points to the last commit you made.
• upon each new commit, the master branch pointer moves forward automatically.
• master branch in Git is not a special branch. It is exactly like any other branch.
Branching – create a new one

Branch

• A new branch means -> a new pointer

Command to create a new branch:
git branch Branch1

Branch1 is a new pointer
Branching – working on new branch

Branch

• Now how to switch to that new branch i.e. *Branch1*?
• *git checkout* command can be used to do so.
  • *git checkout Branch1*
Branching – working on new branch

**Branch**

- **HEAD** is a *special pointer* to know what branch you are currently working on?
- When you start a new repo, **HEAD** pointer always points to *master* branch.
- Upon checkout to different branch, **HEAD** pointer points to the new branch.

**Diagram:**

- Master
- HEAD
- Branch1

Command to checkout *Branch1*:

```
git checkout Branch1
```
Branching – working on new branch

**Branch**

What happen when we make some changes to *Branch1*?
Branching – working on new branch

Branch

What happen when we make some changes to Branch1?

DIY: Try running (with multiple branches and multiple commits)

```
$ git log --all --graph
```
Pulling vs Cloning
Pulling vs cloning Repository

- **git clone:**
  - To get a local copy of an existing repository to work on
  - Clones a repository into a newly created directory, creates remote-tracking branches for each branch in the cloned repository
  - usually only used once for a given repository
  - usually to get a clean copy

- **git pull command:**
  - Incorporates changes from a remote repository into the current branch
  - *update* the local copy with new commits from the remote repository.
Fork and Merge
Forking

- Branching and forking provide two ways of diverging from the main code line.
- *Fork* is another way of saying clone or copy.
- Fork is a clone on the GitLab (or GitHub) side (it clones everything).
- If you fork a repository, you get that repository and all of its branches.
- A fork is independent from the original repository
- If the original repository is deleted, the fork remains.
Merging

In this case, let's merge *Branch1* and *Master* branch.

```
git checkout master
```
Merging

In this case, let's merge *Branch1* and *Master* branch.

- `git checkout master`
- `git merge Branch1`

*Also known as fast-forward merging*
Merging

In this case, let's merge *Branch1* and *Master* branch.

```
git checkout master

git merge Branch1
```
Merging

In this case, let's merge *Branch1* and *Master* branch.

- `git checkout master`
- `git merge Branch1`

*Also known as three-way merge*
Merge conflict Example

- If you have changed the *same part* of the *same file* differently in the two branches.
Merge + Rebase

- **Merge** when bringing changes from one branch to the master branch
- **Rebase** when updating your branch with master

- Common goal:
  - to integrate changes from one branch into another branch
Rebase

• **Useful**: when updating your branch with master

Take the patch of the change that was introduced in C5 and reapply it on top of C4

commit the changes -> C6
Pull request vs Merge Request
Pull request vs Merge Request

According to GitLab Doc-

“...Tools such as GitHub and Bitbucket choose the name “pull request”, because the first manual action is to pull the feature branch. Tools such as GitLab and others choose the name “merge request”, because the final action is to merge the feature branch....” [src]
Git objects
How git stores objects?

- `mkdir myrepo`
- `cd myrepo`
- `git init`

Let's focus on `.git/objects` directory.
How git stores objects?

- `mkdir myrepo`
- `cd myrepo`
- `git init`
- `echo "File1: This is line 1." > File1.txt`
How git stores objects?

- `mkdir myrepo`
- `cd myrepo`
- `git init`
- `echo "File1: This is line 1." > File1.txt`
- `git add *`

New directory 68 and a file with alpha numeric file name
How git stores objects?

- `mkdir myrepo`
- `cd myrepo`
- `git init`
- `echo "File1: This is line 1." > File1.txt`
- `git add *`
- `git cat-file blob 68dff0e2dcb599a0850d9f2e94e60ebd51e0a480`

Showing the content of blob (or the file)
How git stores objects?

- mkdir myrepo
- cd myrepo
- git init
- echo "File1: This is line 1." > File1.txt
- git add *
- git cat-file blob 68dff0e2dc599a0850d9f2e94e60ebd51e0a480
- git commit -m "Saving first file"

Notice the commit ID.

Notice the new directories and blobs.
How git stores objects?

- `mkdir myrepo`
- `cd myrepo`
- `git init`
- `echo "File1: This is line 1." > File1.txt`
- `git add *`
- `git cat-file blob 68dff0e2dcb599a0850d9f2e94e60ebd51e0a480`
- `git commit -m "Saving first file"`
- `git cat-file -p 3d1020f`

Notice the commit ID.
How git stores objects?

- `mkdir myrepo`
- `cd myrepo`
- `git init`
- `echo "File1: This is line 1." > File1.txt`
- `git add *`
- `git cat-file blob 68dff0e2dcb599a0850d9f2e94e60e65d1e0a480`
- `git commit -m "Saving first file"
- `git cat-file -p 3d1020f`
- `git cat-file -p 9ad7d8a2c46`

Notice the tree (NOT the command in the screenshot) ID.
Notice the blob ID.
How git stores objects?

- mkdir myrepo
- cd myrepo
- git init
- echo "File1: This is line 1." > File1.txt
- git add *
- git cat-file blob 68dff0e2dcb599a0850d9f2e94e60ebd51e0a480
- git commit -m "Saving first file"
- git cat-file -p 3d1020f
- git cat-file -p 9ad7d8a2c46
- git cat-file -p 68dff0e2dc

Notice the blob content.
How git stores objects?

- `mkdir myrepo`
- `cd myrepo`
- `git init`
- `echo "File1: This is line 1." > File1.txt`
- `git add *`
- `git cat-file blob 68dff0e2dcb599a0850d9f2e94e60ebd51e0a480`
- `git commit -m "Saving first file"`
- `git cat-file -p 3d1020f`
- `git cat-file -p 9ad7d8a2c46`
- `git cat-file -p 68dff0e2dc`
How git stores objects?

- `mkdir myrepo2`
- `cd myrepo2`
- `git init`
- `echo "File1: This is line 1." > File1.txt`
- `echo "File2: This is line 1." > File2.txt`
- `git add *`
- `git commit -m "Saving two files for the first time"`
How git stores objects?

- `mkdir myrepo2`
- `cd myrepo2`
- `git init`
- `echo "File1: This is line 1." > File1.txt`
- `echo "File1: This is line 1." > File2.txt`
- `git add *`
- `Git commit -m "files with same content."`
How git stores objects: Modified files?

- `mkdir myrepo4`
- `cd myrepo4`
- `git init`
- `echo "File1: This is line 1." > File1.txt`
- `git add *`
- `Git commit -m "Saving first version."`
How git stores objects: Modified files?

- `mkdir myrepo4`
- `cd myrepo4`
- `git init`
- `echo "File1: This is line 1." > File1.txt`
- `git add *`
- `Git commit -m “Saving first version.”`
- `echo "File1: This is line 2." >> File1.txt`
- `git add *`
- `Git commit -m “Saving Second version.”`
Git objects: Two-level directory structure

- mkdir myrepo
- cd myrepo
- git init
- echo "File1: This is line 1." > File1.txt
- git add *
- git cat-file blob 68dff0e2dcb599a0850d9f2e94e60ebd51e0a480
git commit -m "Saving first file"
- git cat-file -p 3d1020f
- git cat-file -p 9ad7d8a2c46
- git cat-file -p 68dff0e2dc

Diagram:

- Commit 3d1020f
- Tree 9ad7d8a2c46
- Blob 68dff0e2dc (File1.txt)

Directory name
File Name
Git object: handling branch & merge

DIY: Where is the branch info?
• Look at `.git/refs/heads/` directory

DIY: What happen when you merge two branches without conflict?
• Observe `.git/refs/heads/` directory

DIY: What happen when you merge two branches with conflict?
• Observe `.git/refs/heads/` directory
Git objects: Data Integrity

- Everything in Git is *checksummed* before it is stored and is then referred to by that checksum.

```
$ git log
commit 1e11daad0cbb5c074ec5433df8c9a066393ea318 (HEAD -> master)
Author: chinmaya dehury <chinmaya.dehury@ut.ee>
Date:   Tue Oct 12 16:01:28 2021 +0300

  first change..
```

This the checksum of this commit
Git objects : Data Integrity

- Everything in Git is *checksummed* before it is stored and is then referred to by that checksum.
- Git *stores* everything in its database *not by file name* but by the hash value of its contents.
- SHA-1 (or other SHA256) hash is used for checksumming.
- a 40-character string
- composed of hexadecimal characters (0–9 and a–f)
- calculated based on the contents of a file or directory structure in Git
Git objects : Compression

- Git compresses the content of a blob object (file)
  - Uses zlib tool
- The commit and tree content are very specifically formatted
  - Header of these files usually begins with commit or tree

GitLab
Beyond just a repository

• Repository is mostly about the source code
• But in GitLab (and some other alternate solutions), you get more than just managing the code:
  • Manage permissions and roles
  • Manage groups
  • DevOps - CI/CD pipeline
  • Manage issues
  • Deployments
  • Analytics
  • Wiki
  • Notification integrations
  • Monitoring
  • Graph visualization
  • and many more...

❖ So in GitLab you always create *projects not just a repository that contain only source code*. 
GitLab - User types

There are several types of users in GitLab:

• Regular users and GitLab.com users.

• Groups of users:
  • use groups to manage permissions for your projects
  • access to a group -> access to all the projects in the group
  • can view issues, merge requests for the projects in the group, analytics of the group's activity, etc
  • can communicate with all of the members of the group at once
  • can also create subgroups.

• Subgroups
  • GitLab supports up to 20 levels of subgroups
  • When you add a member to a group, that member is also added to all subgroups
  • Permission level is inherited from the group's parent.
GitLab - User types

There are several types of users in GitLab:

• Group example

- Organization Group - GNU/Linux distro
  - Category Subgroup - Packages
    - (project) Package01
    - (project) Package02
  - Category Subgroup - Software
    - (project) Core
    - (project) CLI
    - (project) Android app
    - (project) iOS app
  - Category Subgroup - Infra tools
    - (project) Ansible playbooks

- Organization Group - GitLab
  - Category Subgroup - Marketing
    - (project) Design
    - (project) General
  - Category Subgroup - Software
    - (project) GitLab CE
    - (project) GitLab EE
    - (project) Omnibus GitLab
    - (project) GitLab Runner
    - (project) GitLab Pages daemon
  - Category Subgroup - Infra tools
    - (project) Chef cookbooks
  - Category Subgroup - Executive team
There are several types of users in GitLab:

• Regular users and GitLab.com users.

• Groups of users:

  • GitLab Administrator with full access to self-managed instances' features and settings:
    • web UI to manage and configure GitLab self-managed instances.
    • statistics and system information about the GitLab instance.
    • administer all jobs in the GitLab instance
    • administer all runners
    • Get System Info, logs

• Internal users.
GitLab - User types

There are several types of users in GitLab:

• Regular users and GitLab.com users.

• Groups of users:

• GitLab Administrator with full access to self-managed instances' features and settings:

• Internal users:
  • sometimes referred to as "bots"
  • created programmatically
  • example jobs:
    • generating alerts
    • automatic review feedback
GitLab – Permissions and roles

• Each member in a project gets a role
  • -> determines what they can do in the project.

• Role defines the abilities

• List of roles:
  • Guest
    • Creates a new issue
    • Can leave comments
    • Download a project
    • View wiki pages, and more
  • Reporter
    • Manage project labels
    • View project statistics
    • View Error Tracking list, and more
GitLab – Permissions and roles

• Each member in a project gets a role
  • -> determines what they can do in the project.

• Role defines the abilities

• List of roles:
  • Guest
  • Reporter
  • Developer
    • Create new merge request
    • Create new branch
    • Write a wiki, and more
  • Maintainer
    • Manage CI/CD variables
    • Manage job triggers
    • Manage runners, and more
GitLab – Permissions and roles

- Each member in a project gets a role
  - -> determines what they can do in the project.

- Role defines the abilities

- List of roles:
  - Guest
  - Reporter
  - Developer
  - Maintainer
  - Owner
    - Delete pipelines
    - Delete Merge requests
    - Archive project
    - Change project visibility level
    - Delete project
    - Rename project
    - Disable notification emails
    - ... and everything that others can not perform

Details on permission and roles:
- https://gitlab.cs.ut.ee/help/user/permissions.md
GitLab - namespace

• A namespace is a
  • unique name AND
  • URL for a user, a group, or subgroup
    • http://gitlab.example.com/username
    • http://gitlab.example.com/groupname
    • http://gitlab.example.com/groupname/subgroup_name

• Namespace Examples:
  • Chinmaya Dehury creates an account with the username *dehury*:
    https://gitlab.example.com/dehury
  • https://gitlab.cs.ut.ee/devops2021/students
  • https://gitlab.cs.ut.ee/devops2021

• Not to confuse with project url, e.g.
  https://gitlab.cs.ut.ee/dehury/DevOps21FallPub
Summary

- Version Control System
- VCS Types & Goals
- VCS software tools
- Git History & Features
- GitLab (User types, Permissions and roles, namespace)
- Git -- Installation & Configuration
- Git Repository and States
- Git data Integrity
- Some more git concepts... (Pulling vs cloning Repository, Branching, Forking, Merging, Rebase)
Further Reading...

Git Large File System (Git LFS)
  • How to handle large files, for example the assets of Game development

Git Virtual File System (GVFS)
  • Would you prefer to work with a Git repo having more than 3.5M files
  • [https://devblogs.microsoft.com/bharry/the-largest-git-repo-on-the-planet/](https://devblogs.microsoft.com/bharry/the-largest-git-repo-on-the-planet/)

Git for large project:
  • Very old project
  • Large file count, activity, file size
Lab Sessions

- Creating first repository (local & remote)
- Branches and Merging
- Handling Merge Conflicts
- Forking and merging a branch
Next Lecture : Containerization
References

5. https://www.atlassian.com/git/tutorials
Any Question?

THANK YOU
Revert vs. Unstage
Unstage

• To remove files from stage use reset HEAD. Where HEAD is the last commit of the current branch.
  • `git reset HEAD <file>`

• This will unstage the file but maintain the modifications. To revert the file back to the state it was in before the changes we can use:
  • `git checkout -- <file>`

• To remove a file from disk and repo use 'git rm'
  • e.g.
    • `git rm readme.md`
    • `git rm -r feature-dir`

• `git rm <filename> --cache` to remove a file from the repository but keep it on disk
Revert

- **Reset** removes the commit while **revert** removes the changes but leaves the commit
- Revert is safer
  - One can revert a revert
UNDO

• Undo last commit putting everything back into the staging area.
  
git reset --soft HEAD^  

• Undo last and remove changes
  
git reset --hard HEAD^  

• Undo for two commits back
  
git reset --hard HEAD^^