

APPS@UCU

# Linux course

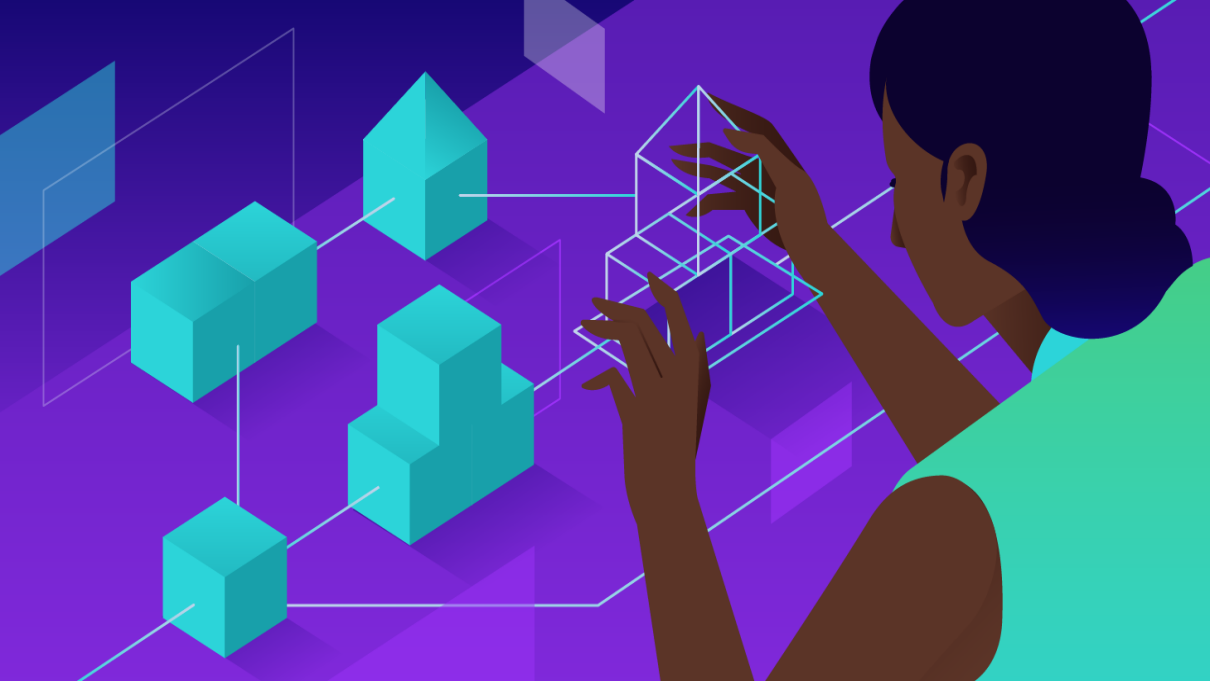
Version control systems

Morhunenko Mykola



# Contents

- 1 Version control
- 2 git
- 3 GitHub
- 4 GitLab
- 5 Git usage
- 6 GitHub features
- 7 How to start?
- 8 Sources



# Version control systems

- Sooner or later, during the development process, it is necessary to check, what was before, how it became broken
- Maybe it is easier to use `Ctrl+z` , but it's impossible to check what was in the code three weeks ago with any keybinding
- In 1972 people started to think about `version control systems`
- Firstly, it had been just a tool for saving a history of binary files, but in 1977 the first `source code control system` was introduced
- The main idea behind - to save the program source code on some checkpoints (`commits`) , add features, develop them leaving trunk untouchable (`branches`) , `merge` new features with a trunk, and release some `tags`
- Since then, the concept itself was developed, and many version control systems have appeared



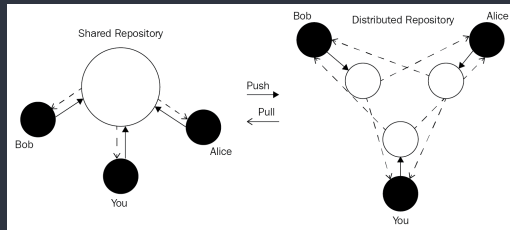
git

## Linus again...

- **Linux kernel** is a huge project, and it is important to have some source-control management system (SCMs) to maintain it
- From 2002 to 2005 BitKeeper, a proprietary SCMs was used to maintain the project
- At some point (3 April 2005), Linus Torvalds realized that existing tools are not suitable for Linux development, so in three days, he announced a project and became a self-hosting of **Git** on the next day
- It was different SCMs. Linus maintained it for half a year, and Junio Hamano has been the core maintainer since then
- It was open-source, free software, with robust safeguards against corruption, either accidental or malicious
- Torvalds sarcastically quipped about the name **Git**, means **unpleasant person** in British English
- He said: *"I am an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'."* =)

# What makes Git so good

- Strong support for non-linear development
- Distributed development
- Efficient handling of large projects
- Toolkit-based design
- Pluggable merge strategies
- And more other features
- It's hard to find any statistics, but that is clear - Git is the most popular SCMs of our days



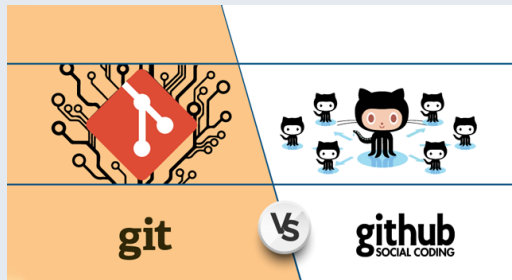


github  
SOCIAL CODING



# Git is not GitHub

- **GitHub, Inc** - provifer of internet hosting for software development and version control using git
- It offers all the functionality of **Git** + it's own features
- Since 2018 - subsidiary of **Microsoft**
- **Not an Open Source project** , but there is a forum for feature requests...
- As of January 2020, GitHub reports having over 40 million users
- More about it's features after **Git usage** part





```
$ git push
```

# Git is not GitHub

- First difference - GitLab was created by **Ukrainian** people, in Ukraine =)
- It has more features, than **GitHub** , but there is no critical difference
- It is **Open Source** , unlike Github
- It is possible to have the same repository on both servers, and I do it sometimes
- So as for 2021, it is just a question of taste



# **GIT USAGE**



# IN CASE OF FIRE



Git Commit



Git Push



Git Out

# Creating a repo

- A **repository** contains all of your project's files and each file's revision history. You can discuss and manage your project's work within the repository.
- **Repository** is NOT a project folder. Repository is *a data structure that stores metadata for a set of files or directory structure*
- Command to initialize a repo in your current folder

```
git init
```

- use **git add** command to specify files you want to track, followed by **git commit** - add a specific message to your commit

```
git add *.sh
```

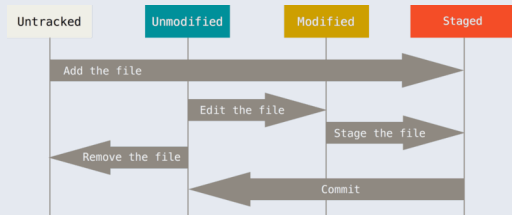
```
git add .gitignore
```

```
git commit -m "add gitignore file; add scripts for some task"
```

- How to write correct commit messages is another art, but remember to write meaningful messages
- So at this point, we have a Git repository in our project directory with tracked files and an initial commit

# Changes to the repo

- At this point you have a git **repo** with scripts and some files
- Each file in your working directory can be in one of two states: tracked or untracked
- Tracked files are files that were in the last snapshot, as well as any newly staged files; they can be unmodified, modified, or staged
- , In short, tracked files are files that Git knows about
- Untracked files are everything else
- Use **git status** to check the status of each file in the current directory
- Files in a **.gitignore** are ignored by git repo



## Manipulations with repo

- As far as `git` is a `decentralized` system, you already have your repo with all version control features
- But now about the most powerful `git` feature and why we use it - `remote repo`
- You can either `clone` existing repo, or add a `remote` to local one
- To `remove` a file from both repo and directory, use `git rm`
- To `rename` a file, use `git mv`
- To add a `remote` to your repo, use `git remote add <name> <url>`
- To `push` your updates to remote, use `git push <remote> <branch>`

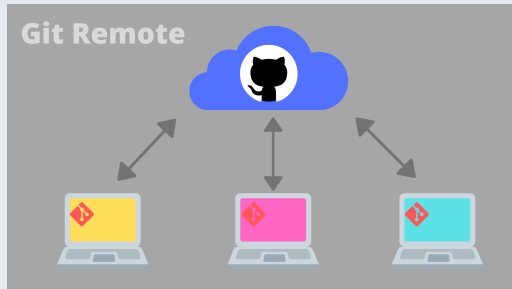


## Manipulations with repo (examples)

```
mkdir test_directory
cd test_directory
echo "empty_readme" >> README.md # initialize new readme
git init
git add README.md
git commit -m "initialized new repo with readme"
# go to https://github.com, create a new repo
# go back to terminal and push your changes to remote
git remote add origin git@github.com:username/reponame.git
git commit -m "add gitignore file; add scripts for some task"
git push origin master
```

# Remotes

- **Remote** - link to remote "versions" of your repository. They are stored on some kind of service (GitHub, GitLab, BitBucket etc.) or on private server
- You can think of them as global versions of your repos (your repo can be stored on few of them simultaneously)
- To see your remotes use **git remote show** , by default there is only **origin**
- For more info on a specific remote, use **git remote show remotename**
- Use **git pull** to make a code from **local** up to date with **origin**

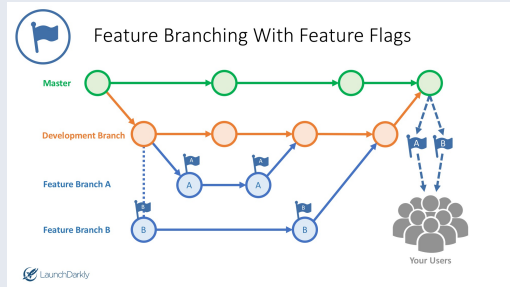


# Branches



# Branches

- **Branch** is a divergence from the one development line to fulfil some specific purpose
- For example - stable branch (usually master), development (dev), features, bug fixing
- Used for concurrent work of different developers on the same project
- Helps to avoid often conflicts while working on fairly independent parts
- Easier to access code for a specific feature and explore its history



## Branches (examples)

```
# we already have a test_dir. continue working with it
cd test_directory
ls
> README.md
git checkout -b dev # same as git branch dev; git checkout dev
echo 'print("Hello world")' >> new.py
git add new.py
git commit -m "add new.py file"
git push origin dev
git checkout master # checkout back to master
ls
> README.md
```

You are back at **master** branch where there is no **new.py** file, it is on the **dev** branch

Now it is possible to merge with

```
git merge dev # about merge hell read yourself
```



GitHub Presents:

**GAME OFF**

- **GitHub repository** - remote “versions” of your repository, it's **origin** , but also much more...
- **GitHub** provide users with such a nice tools as **Issues, Pull requests, Forks, Actions** , also **followers and followings, their activity, stars for projects, Projects** tool for managing project development workflow, **Marketplace, Insights**(some statistics), **Wiki, even Discussions**
- So developers do a **GitHub** as a big
- Starting with 2021, there is such thing as **GitHub CLI** !

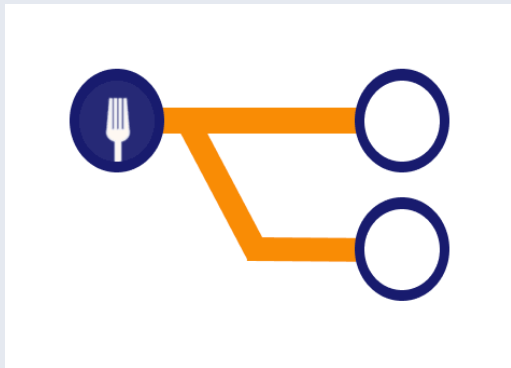
# Issues

- If you use some app stored on **GitHub** and it works with bugs - you have two options: report the bug or fix it and send a **bugfix** to maintainers
- For the first thing **Issues** could be helpful (For the second - **Pull requests** )
- All GitHub users can create an issue on some projects pages, comment it, react on it with some emoji etc
- Issue itself - a conversation-starting with user's message, that (for example) he found a bug, and some information about how you got this bug
- Also there can be **feature request bugs** or even **improvements** or other. There are dozens of such labels
- Often Issues either has a solution written as the last message or a pull request that solves the problem
- After that **issue** can be marked as solved
- **Issues** are part of the development workflow, so it is important to keep them as much as keep source code of the program itself and its commits history



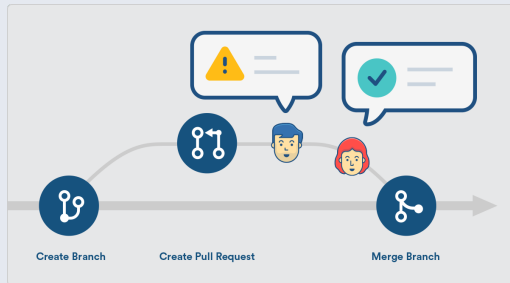
# Fork

- A GitHub **fork** is a copy of a repo that sits in your account rather than the account from which you forked
- Once you have forked a repo, you own your forked copy
- This means that you can edit the contents of your forked repository without impacting the parent repo
- Forked repo can be **detached** (in theory, but it is not possible for now. By hands, the support team can do that) to become an independent project



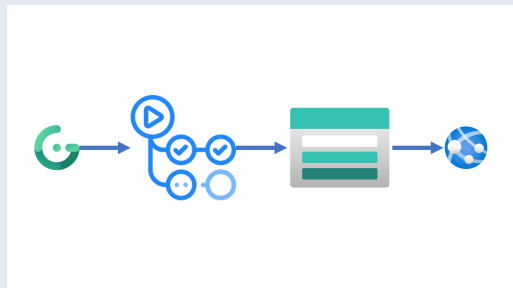
# Pull request

- In most cases, nobody wants all their code to be broken by strangers, so only a few people ( **contributors** ) have the right for changing the repo
- Instead, they should **fork** , implement some feature and make a **pull request** to merge their contribution to your project
- Almost the same (merge request) can be done between branches inside one fork
- Sometimes there is a **CONTRIBUTION.md** describing how to make correct changes with clear commits and acceptable PR message for changes to be merged



# GitHub Actions. CI from GitHub

- More advanced usage of GitHub - use it with **GitHub Actions**
- Continuous integration (CI) - the practice of automating the integration of code changes
- Using this tool, it is easy to test any project for compilation on different compilers, distros, with different configurations, and all of that - automatically
- There are hundreds of existing **actions**, but it is possible (and quite easy) to make your own
- I recommend to use it for lab works and small projects - then it will be easier to use it everywhere



# THE ESSENTIAL GITHUB CLI COMMANDS



## GitHub cli commands

- Starting from 2021 GitHub released it's API and made an amazing tool called **GitHub CLI** !
- They made possible to automate a lot of stuff as creating a repo or read issues, make pr etc.
- Some important commands:
- **gh auth login** - authentication
- **gh config set editor <editor>**
- **gh repo create** - create a repo (allow to skip "going to a browser and do some stuff there...")
- **gh pr create** - create a pr from current branch to master
- **gh workflow run**



GETTING STARTED WITH GIT

## How to get stated?

- First of all you should create an account on GitHub
- Than you can create repositories, clone any open repo, add friends, put starts etc
- GitHub try to improve it's security more and more
- In case if you want to **push** something, firstly you need to authosize yourself (abd create new ssh-key)

```
git config --global user.name "username"  
git config --global user.email "email@example.com"
```

- Then make an ssh key and add it to your gh accoung. More info [here](#)
- Keep using it. Using **Git** is cool!

# Sources



# Sources

- [Version control systems comparison](#)
- [Why Git is Better than X](#)
- [Git Wiki](#)
- [GitHub Wiki](#)
- [GitLab history](#)
- [GitHub documentation](#)
- [Git documentation](#)
- [GitHub CI](#)
- [GitHub CLI](#)