Digital Career Institute

Basic Digital Literacy

Course introduction

DCI

Today

- Course + Curriculum introduction
- Instructor introduction
- Laptop setup
- Installed programs
- GitHub
- Slack Workspace
- Structure of a typical lesson day

Course introduction
Main modules

Basic Digital Literacy → UI Basics → Programming Basics → Single Page Application → Backend

Instructor introduction

Hi!

I’m <Max Mustermann>.

Nice to meet you!

Basic Digital Literacy

1st week	2nd week
Overview	Versioning (Branches)
View, Navigate, Create, Change	Publishing
Installation	Collaboration
Versioning (Basics)	Review

Assessments

1st week	2nd week
Overview	Versioning (Branches)
View, Navigate, Create, Change	Publishing
Installation **Assessment I**	Collaboration
Versioning (Basics)	Review **Assessment II**

BDL Goals

- Work as a **Web Developer**
- Using **Linux**
- Using **git** and **GitHub**
- Authoring with **Markdown**
- Fundamentals of how the **Internet** works

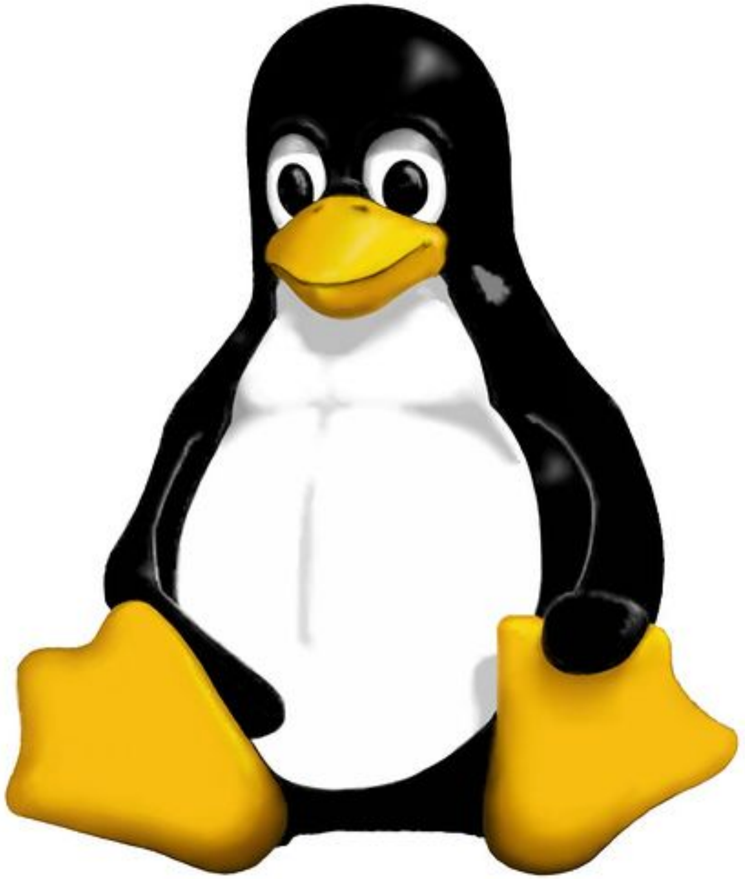
Digital Career Institute

Introduction to Linux

Linux

On this course we will use Linux!

Different kinds of Linuxes exist and we use Ubuntu.



Tux, the Linux mascot

[TUX]: https://en.wikipedia.org/wiki/Tux_(mascot)

Linux

Different Linuxes are called distributions or **distros**

Ubuntu, Debian, CentOS...

A distro is a combination of the Linux **kernel** with software, often combined with a package management system.

The Linux kernel was developed by Linus Torvalds.

Linux

Different Linuxes are called distributions or **distros**

Ubuntu, Debian, CentOS...

A distro is a combination of the Linux **kernel** with software,
often combined with a package management system.

Ubuntu uses a package management system called **apt**

Distros have a *Desktop Environment* which provides the graphical user interface.

Ubuntu uses the **Gnome** desktop environment.

Linux

Why Linux?

1. **The terminal**
   A powerful and fast user interface
   Base commands don't change fast
   Base commands shared by MacOS (and Windows via WSL and gitbash!)

2. **Servers**
   Most web servers run Linux
   Similar development and production environments → less problems
   Containerized environments are practically all Linuxes

3. **Widely used**
   Many companies use Linux or MacOS workstations
   Linux skills transfer very well to MacOS

4. **It's fast and free**

Web Development

Web Development

A day in my life as a Web Developer

09:00	Start Work → check emails and chat, preparation
10:00	Daily project status meeting
10:15	Work
12:00	Lunch
17:00	Done for the day

Web Development

Some of this might be already familiar to you

Web Development

Work means:

- Working with designs
- Understanding requirements
- Testing
- Collaborating
- Planning and estimating
- Coding
    - Writing code
    - Lots of research

+ emails
+ chats
+ meetings
+ reviews
+ much more

Web Development

Main areas of focus

- **Frontend**
    - User interface and experience

Web Development

Main areas of focus

- **Frontend**
  - User interface and experience

- **Backend**
  - Logic, security

Web Development

Main areas of focus

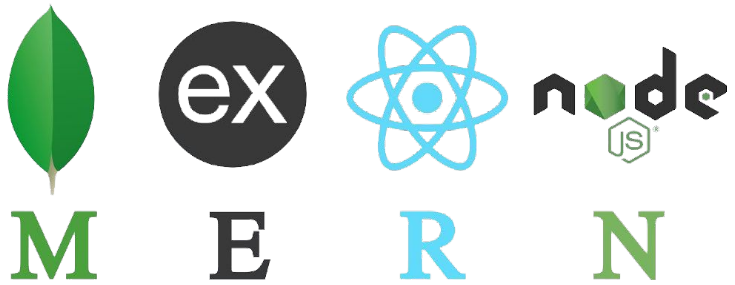
- **Frontend**
  - User interface and experience
- **Backend**
  - Logic, security
- **Full stack**
  - Frontend and Backend

Technology stack → set of technologies used

Web Development

We cover the full stack

- ***M**ongoDB*
- ***E**xpress*
- ***R**eact*
- ***N**ode*



https://commons.wikimedia.org/wiki/File:MERN-logo.png

Web development is also

- Testing
- Database admin
- Operations
- etc

Web Development

We can't learn *everything* in one year

We **can**

- Learn how to learn
- Practice the fundamentals
- Practice the language
- Practice the tools
- Practice the workflow

At the core of the lesson

Linux

- Powerful, fast, stable
- Linux kernel: the system core
- Distro: the kernel + software
- Desktop environment: graphical user interface

Web Development

- More than writing code
- Frontend, Backend, Full Stack
- MERN stack

DCI

View, Navigate, Create, Change

Using the terminal

1st week	2nd week
Overview	Versioning (Branches)
View, Navigate, Create, Change	Publishing
Installation	Collaboration
Versioning (Basics)	Review

DCI

Using the terminal
Viewing & Navigating

The Terminal

Open the terminal – what do you see?

- You are always working in some specific directory
- You use it with text commands
- Try out the first command

```
$ ls
```

The Linux Filesystem

Files and folders (directories) exist in a hierarchy.

Each file and folder has a unique path, folders are separated by **/**.

```
/tmp/notes.txt          File notes.txt inside the tmp folder
/home/dci/cv.pdf        File cv.pdf inside dci, inside home
```

These are examples **absolute** (full) paths.

The Linux Filesystem

Notable directories

`/`	the root folder
`/etc`	configuration files
`/var`	log files, other variable files
`/home`	ome folders
`/home/dci`	home folder for user dci
`~`	shortcut to *your* home folder

DCI

The Linux Filesystem

Notable directories

`/`	the root folder
`/etc`	configuration files
`/var`	log files, other variable files
`/home`	ome folders
`/home/dci`	home folder for user dci
`~`	shortcut to *your* home folder

Try

```
$ ls /
$ ls /var
$ ls /etc
$ ls /home
```

The Linux Filesystem

Commands can have *options* (sometimes called *flags* or *parameters*)

```
$ ls --help      # show help for ls
$ ls -l          # show long (detailed) listing
```

The Linux Filesystem

On Linux files starting with a “.” are considered hidden files

Try

```
$ ls -l          # long listing
$ ls -a          # show hidden files
$ ls -la         # long listing, show hidden
$ ls -lah        # long listing, show hidden, human readable sizes
```

The Linux Filesystem

Relative paths are paths relative to the current working directory.
Check the current working directory:

```
    $ pwd
```

Use “.” to refer to the current directory and “..” for the parent directory.

```
    $ ls .                  # contents of current folder
    $ ls ..                 # contents of parent  folder
    $ ls Documents          # contents of “Documents” folder
    $ ls ./Documents        # . means current folder
    $ ls ../Documents       # .. means parent folder
```

The Linux Filesystem

Change the current working directory with **cd**

```
$ cd ..                # go to parent folder
$ cd ~                 # go to home folder
$ cd Documents         # go to “Documents” folder
$ cd ~/Downloads       # go to the “Downloads” folder in your home
```

What next?

```
$ less file.txt  # open a text file for viewing (q to quit)
```

Awesome terminal feature: **tab autocompletion**

At the core of the lesson

- Issue commands in the terminal
- Your home folder is ~
- Absolute and relative paths

Useful commands

```
$ ls          # list
$ cd          # change directory
$ pwd         # print working directory
$ less        # view text file
```

Command options like –help

```
$ ls --help      # show help for ls
$ cd Downloads   # change directory
```

Using the terminal
Creating & Manipulating

Working with files

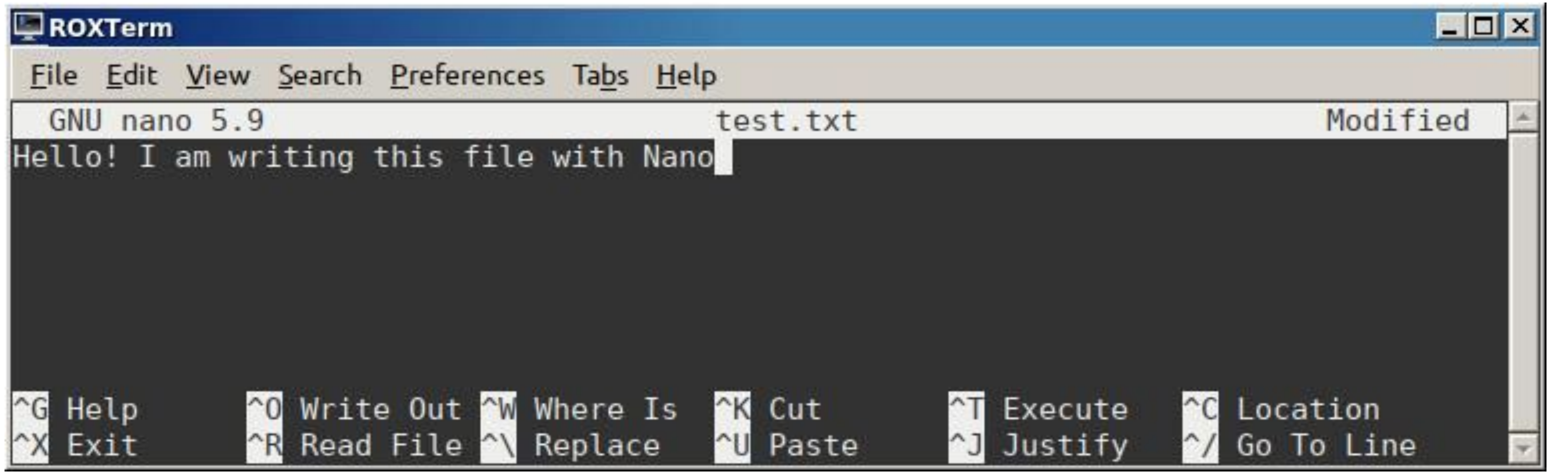
We will have **lots** of files and folders!

Be organized and systematic from the start

```
$ mkdir projects              # make projects directory
$ cd projects                 #
$ mkdir test-project          # make test-project directory
$ cd test-project             #
$ pwd                         # print working directory
$ touch plan.txt              # create empty file plan.txt
$ ls                          #
```

Working with files

```
$ nano test.txt     # introducing the nano text editor
```



```
  GNU nano 5.9                     test.txt                         Modified
Hello! I am writing this file with Nano







^G Help       ^O Write Out ^W Where Is  ^K Cut        ^T Execute    ^C Location
^X Exit       ^R Read File ^\ Replace   ^U Paste      ^J Justify    ^/ Go To Line
```

```
^X   means press Ctrl+x
M-C  means press Alt-c
```

DCI

Working with files

Practice commands with me!

```
$ ls -la                  # long listing, show hidden
$ ls --help               # help for ls
$ mkdir --help            # help for mkdir
$ mkdir projects          # make projects folder
$ cd projects             # change to projects
$ pwd                     # print working directory
$ less file.txt       # view text file
$ touch plan.txt          # create empty file plan.txt
$ man ls                  # show manual for ls (q to quit!)
```

So many… let’s find a *cheat sheet*!

Working with files

VS Code

- Lightweight but powerful IDE
  - Integrated Development Environment
- Super popular
- Supports many languages
- Extensible
- Has a terminal *built in* ❤️

DCI

Working with files

VS Code

- One of our most important tools
- Learn it well
- Train usage in your free time
- We will practice with **VS Code** + **Markdown**

Working with files

Markdown files have the **.md** extension

- Simple markup language
  - Not a programming language
  - Clear *syntax*
- Understandable as plain text
- Can be *rendered*

Markdown in action
(main documentation file for VS Code)

Working with files

```
$ cd ~/projects
$ mkdir markdown-test
$ code markdown-test
```

Working with files

~~~
# Main heading
## Second level heading
### Third level heading

Normal, **bold**, *italic*, `hilighted` text!

1. List item
2. List item

- List item
- List item

> This here is a quote

[A link](https://www.google.com)

![An image](https://placekitten.com/100 "Cat")

```html
  <main>
    <p>HTML inside Markdown!</p>
  </main>
```
~~~

Working with files

Deleting

```
$ rm test.txt                # delete file
$ rm *.pdf                       # delete all .pdf files
$ rm -r old-project              # delete folder
$ rm -rf old-project             # force delete folder (careful!)
```

The * (asterisk, wildcard) matches multiple files!

Working with files

Moving

```
$ mv test.txt newname.txt          # move file (rename)
$ mv my-project ~/backup           # move folder to ~/backup
```

Working with files

Copying

```
$ cp test.txt test2.txt            # copy file
$ cp -r project backup             # copy folder

$ cp --help                        # show help for cp

$ cp * ~/backup                    # copy all files to ~/backup
$ cp -r * ~/backup                 # copy all files & folders
```

At the core of the lesson

- Visual Studio Code is our main editor
- Markdown is a markup language

- The terminal can do a lot
  - Edit text
  - Create and manage files
  - Create and manage folders
  - Start programs

DCI

Installation

file system, package manager, version control

1st week	2nd week
Overview	Versioning (Branches)
View, Navigate, Create, Change	Publishing
Installation	Collaboration
Versioning (Basics)	Review

Installing

Package managers

- Package managers connect to online *repositories*
  - Repository = list of software
  - Can list multiple versions of an package
  - Updated by maintainers

```
$ sudo apt update                  # refresh list of latest packages
$ sudo apt search neofetch         # search repositories
$ sudo apt install neofetch        # install neofetch
$ neofetch                         # run neofetch
```

Installing

Package managers

- Can remove and update too

```
$ sudo apt remove neofetch              # remove neofetch
$ sudo apt upgrade                  # update installed packages
$ sudo apt autoremove                   # remove unused packages
```

Installing

We had to use **sudo**

sudo → **su**peruser **do**

- Some things need *elevated permissions*
  → more permissions than normal users
- The linux super user is **root**
- Don’t work as root, use sudo
- Root/sudo can do anything - including destroy your system... careful 😉

⚠️
```
$ sudo danger
```
⚠️

Installing

We will be using the **Node** program later in the course

Node has its own package manager!

npm

(Node Package Manager)

Installing

Usually npm manages packages for a project

We can install globally with npm too

```
$ npm list                               # list installed packages
$ npm install --global batmansay         # install batmansay
$ batmansay -f default
```

At the core of the lesson

- Package managers manage software
    - Install
    - Uninstall
    - Update

- Packages come from online repositories
- Often used from the command line

Popular package managers

- apt
- npm
- snap
- dpkg
- brew (for Macs only)

BDL Assessment I

DCI

Basics of version control system git

1st week	2nd week
Intro and Overview	Versioning (Branches)
View, Navigate, Create, Change	Publishing
Installation	Collaboration
Versioning (Basics)	Review

Versioning

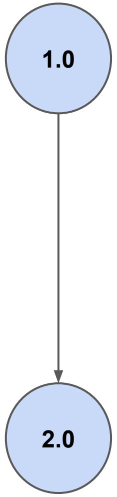
When developing software you **need** a versioning system

- Big teams working on one project together
- Maintaining old versions and developing new ones
- Fixing problems in multiple versions
- Multiple parallel versions before release
    - one version in development
    - one version in testing
    - one version in marketing

Versioning

You are developing a mobile game “**Cyborg**” 🤖.

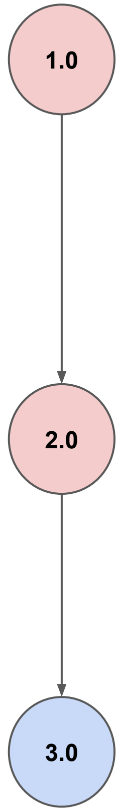
You release 1.0, and then 2.0



Versioning

You are developing a mobile game “**Cyborg**” 🤖.

You release 1.0, and then 2.0
You start 3.0 and find a small **bug** in 1.0 & 2.0

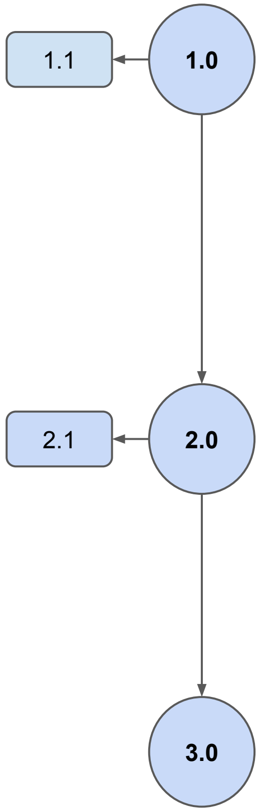


Versioning

You are developing a mobile game “**Cyborg**” 🤖.

You release 1.0, and then 2.0
You start 3.0 and find a small **bug** in 1.0 & 2.0

Some still use 1.0 (old phones can’t use 2.0) so fixes are made → 1.1 & 2.1 & 3.0



Versioning

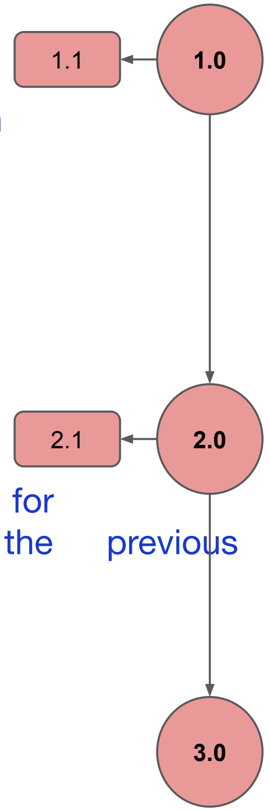
You are developing a mobile game “**Cyborg**” 🤖.

You release 1.0, and then 2.0
You start 3.0 and find a small **bug** in 1.0 & 2.0

Some still use 1.0 (old phones can’t use 2.0) so fixes are made → 1.1 & 2.1 & 3.0

You find a **critical bug** affecting 1.0 / 1.1 / 2.0 / 2.1 / 3.0

The fix is the same for all
you need an emergency fix for **all** versions, even ones without the previous fix



Versioning

You are developing a mobile game “**Cyborg**” 🤖.

You release 1.0, and then 2.0
You start 3.0 and find a small **bug** in 1.0 & 2.0

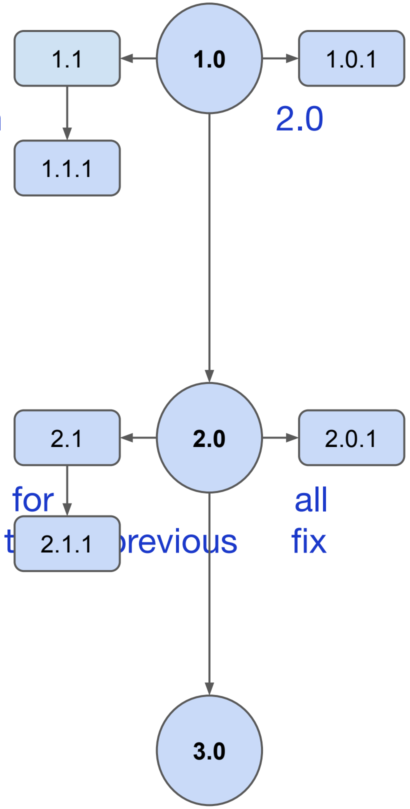
Some still use 1.0 (old phones can’t use 2.0) so fixes are made → 1.1 & 2.1 & 3.0

You find a **critical bug** affecting 1.0 / 1.1 / 2.0 / 2.1 / 3.0

The fix is the same for all
you need an emergency fix for **all** versions, even ones without t previous fix

Now you have 1.0.1, 1.1.1, 2.0.1, 2.1.1 and 3.0. 😨

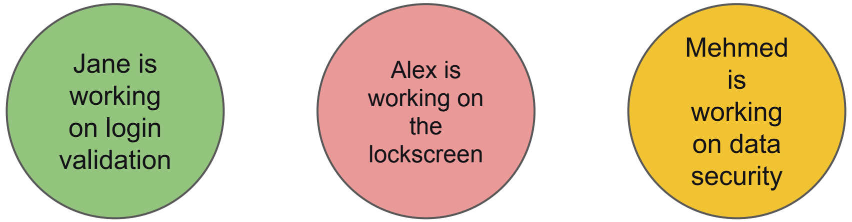
Clearly, a system is needed.

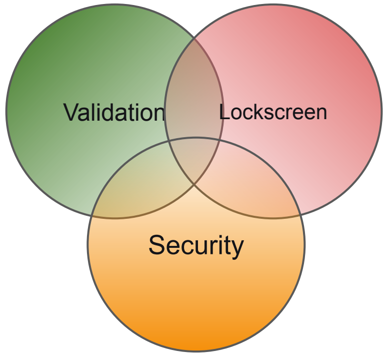


Versioning

Another scenario for **Cyborg**!

What if there are multiple developers?

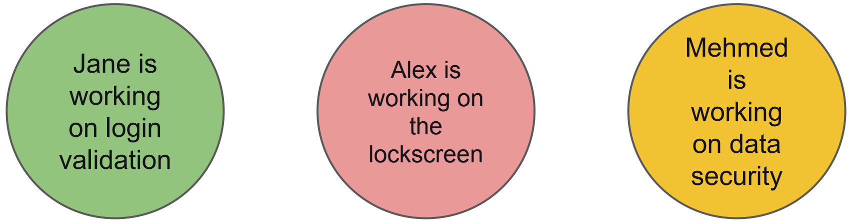

Jane is working on login validation
Alex is working on the lockscreen
Mehmed is working on data security



Versioning

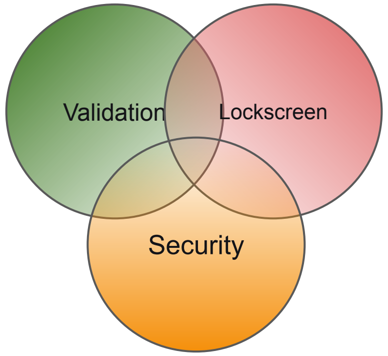
Another scenario for **Cyborg**!

What if there are multiple developers?


Jane is working on login validation
Alex is working on the lockscreen
Mehmed is working on data security

Three versions with overlapping content!

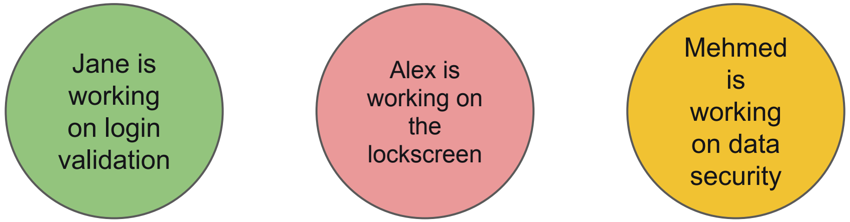
Mehmed finishes his work → Jane and Alex must update their versions
Jane and Alex merge work from Mehmed → conflicting code

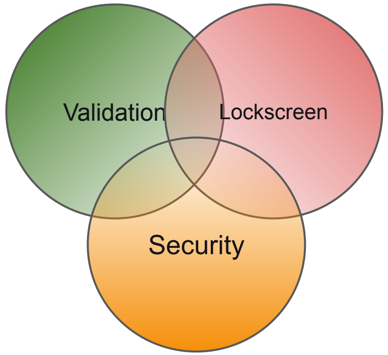


Versioning

Another scenario for **Cyborg**!

What if there are multiple developers?


Jane is working on login validation
Alex is working on the lockscreen
Mehmed is working on data security



Three versions with overlapping content!

Mehmed finishes his work → Jane and Alex must update their versions
Jane and Alex merge work from Mehmed → conflicting code

Imagine this with 18 developers!

Clearly, a system is needed!

Versioning

These issues are solved by a **Version Control System** (VCS)

- version control
- revision control
- source control
- source code management

Versioning

These issues are solved by a **Version Control System** (VCS)

- version control
- revision control
- source control
- source code management

Mainly used for source code, but can also be used for almost any versioning

- Documents like markdown files
- Documentation
- Data files - like language translations
- Configuration files

Versioning

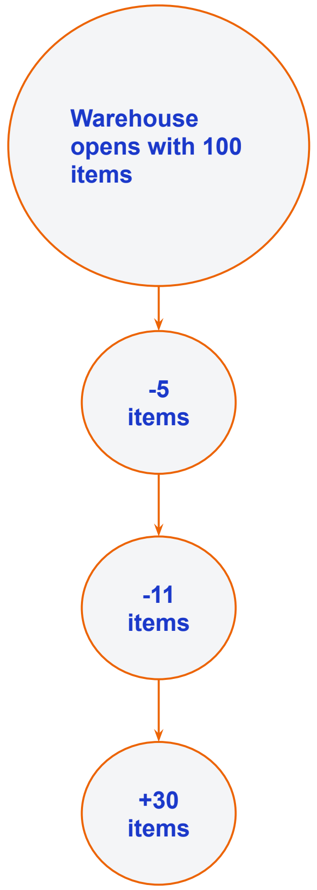
A **Version Control System** is kind of like a warehouse log keeper

Imagine “**Antero**” working in a warehouse

Antero keeps an inventory log

When clients buys items, Antero logs the change!

When a shipment comes in, Antero logs the change!



Versioning

One early version control systems was CVS *(1986)*

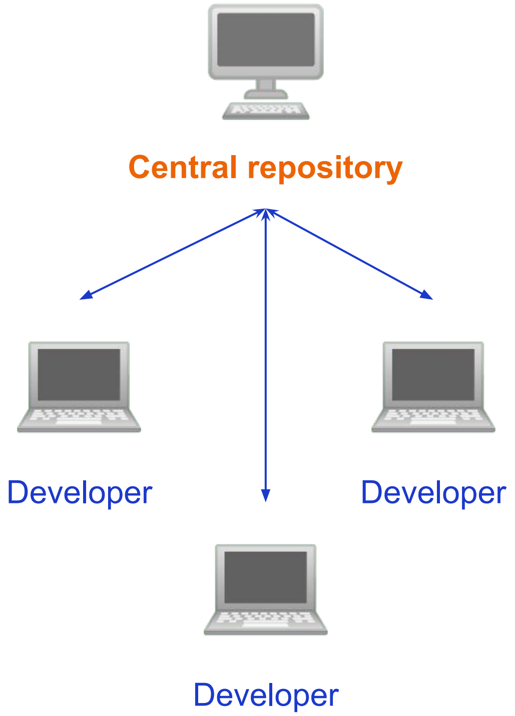
CVS had a **client-server model**

A server stores files and file history in a central *repository*

(repo)

Developers *check out* copies of the central repo

After working, developers *check in* their changes



DCI

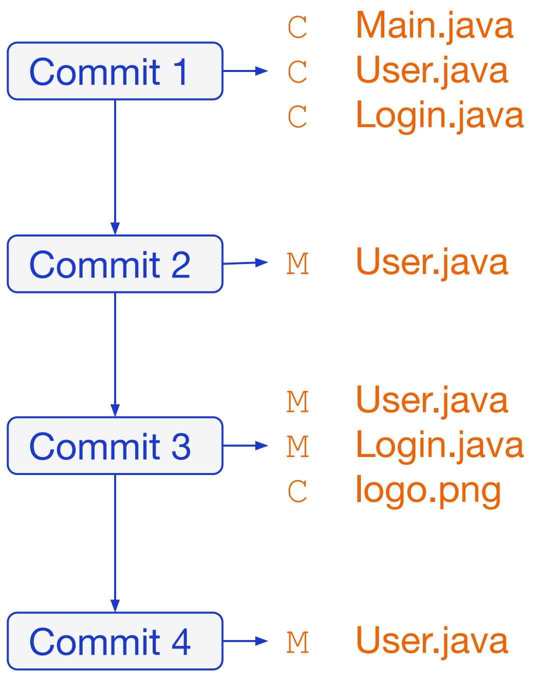
Versioning

CVS tracked changes by file
→ only knew the changes of individual files

The next generation was defined by **Subversion** (svn)

SVN tracked sets of changes, called **commits**

You track the history of your whole project
change by change



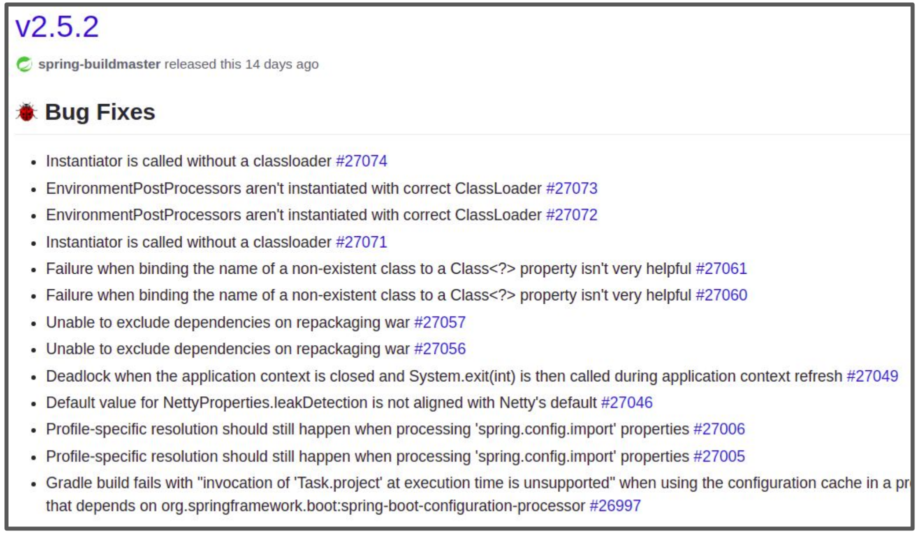
DCI

Versioning

For each commit,
you add a **commit message**

It's easy to generate a list of changes between versions with the messages

The list of commit messages between versions is called a *changelog*



v2.5.2

spring-buildmaster released this 14 days ago

Bug Fixes

- Instantiator is called without a classloader #27074
- EnvironmentPostProcessors aren't instantiated with correct ClassLoader #27073
- EnvironmentPostProcessors aren't instantiated with correct ClassLoader #27072
- Instantiator is called without a classloader #27071
- Failure when binding the name of a non-existent class to a Class<?> property isn't very helpful #27061
- Failure when binding the name of a non-existent class to a Class<?> property isn't very helpful #27060
- Unable to exclude dependencies on repackaging war #27057
- Unable to exclude dependencies on repackaging war #27056
- Deadlock when the application context is closed and System.exit(int) is then called during application context refresh #27049
- Default value for NettyProperties.leakDetection is not aligned with Netty's default #27046
- Profile-specific resolution should still happen when processing 'spring.config.import' properties #27006
- Profile-specific resolution should still happen when processing 'spring.config.import' properties #27005
- Gradle build fails with "invocation of 'Task.project' at execution time is unsupported" when using the configuration cache in a pr
  that depends on org.springframework.boot:spring-boot-configuration-processor #26997

Companies often have rules what commit messages should look like

At the core of the lesson

- Version Control Systems help with
  - Collaboration
  - Release management

- Often there is a central repository

- Modern systems track commits
- A commit is a set of file changes
- Commits have a commit message

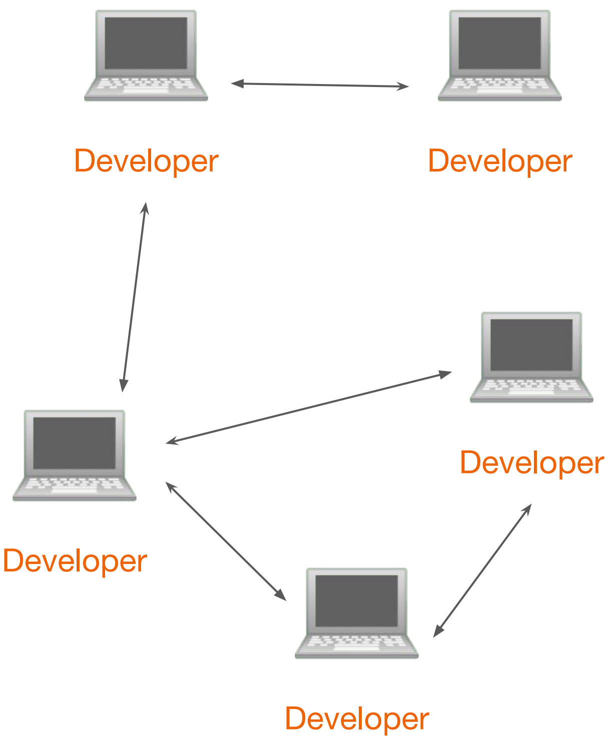
GIT

Versioning

Subversion was the previous generation of VCS

And **git** is the current generation

- Concepts are similar, like commits
- Git is *distributed* (DVCS)
    - Traditionally there was one central repo
    - In git you *just* have repos
    - You don't check out a repo, you clone it



Versioning

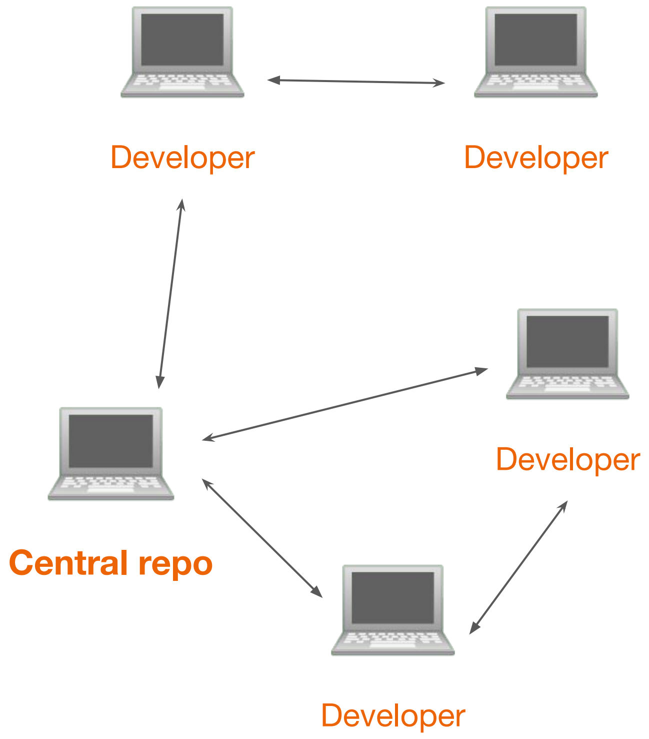
You can transfer work directly between developers

Usually there still is a central repository

Central repositories can be internal or, provided by external service providers

There are many git services

- GitHub
- GitLab
- Bitbucket
- SourceForge



Versioning

- Used in practically all modern software development
- Decentralized / Distributed
- Free and open source
- Scales well for large projects
  - Linux 🐧
  - Visual Studio Code
  - React
- Many tools and services
  - Hosting services
  - Visualization tools
  - Integrations
- Rapid branching *(we will discuss branching soon)*

Versioning

The main way to use git is the git CLI

- Command Line Interface
- used purely in the terminal, using the **git** program

Important concepts

a. Any folder can be made into a new git repository
b. Or repositories can be cloned from another repository
c. Avoid having repositories in repositories (nesting)
d. Do not use **`sudo`** with git

Be aware of what directory you are working in!
Have a place just for all your git repositories ~/projects/

Versioning

We need to configure git

Git configurations are saved in **~/.gitconfig**

You can edit the file or you can use **$ git config**

```
git config --global user.name "Joel Peltonen"
git config --global user.email "joel.peltonen@example.com"
```

Note: many other config options exist too, such as which text editor git uses!

Versioning

You can transform your working directory into a repository

```
$ git init
```

This creates a directory called ***.git***

```
$ ls -a
```

To “unmake” a repository, delete the .git directory

```
$ rm -rf .git
```

Don’t run *git init* inside an old repo
If this happens, undo by deleting the invalid .git directories.
Make sure you delete the right one!

Versioning

The .git contains:

- All commits in all branches
- Connections to other repositories

Browse .git folder, but don’t edit the files
It is easy accidentally break something.

Versioning

The .git directory will only be found in the top level (root) folder of your repository

A directory listing for your project might look like this - note only one .git directory

```
projects/
    my-application/
        .git/
        src/
        public/
        images/
        docs/
```

At the core of the lesson

- Git is used with the **git** command
- Git is distributed
  - usually there’s a central repository
- Any folder can be a repository
- **git init** to create a repository
- Repositories are called repo for short
- Configure git with email and name

Basic git commands & workflow

Git workflow

Note: git is only interested in files

- Create a file → git is interested
- Create a folder → git is not interested
- Create a file inside a folder → ***wow*** git is interested

Your way of using git is called a workflow

- Workflows can differ from company to company
- Workflows can differ from project to project

Git workflow

The most important git command

```
git status
```

This tells you

- Are you in a git repository
- What is the repository status
  - Are there changes to the files
  - What branch are you on

Git workflow

```
dci@dci-laptop:~/projects/not-awesome-project $ git status
fatal: not a git repository (or any of the parent directories): .git
```

Git workflow

```
dci@dci-laptop:~/projects/not-awesome-project $ git status
fatal: not a git repository (or any of the parent directories): .git
```

```
dci@dci-laptop:~/projects/my-awesome-project $ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Git workflow

```
dci@dci-laptop:~/projects/my-awesome-project $ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.adoc

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    newfile

no changes added to commit (use "git add" and/or "git commit -a")
```

Git workflow

When you are ready to commit

You then need to tell git which changes to include in the commit

You do this by **staging** those changes

You do this with

```
$ git add <path>
```

Git workflow

```
dci@dci-laptop:~/projects/my-awesome-project$ git add README.adoc
dci@dci-laptop:~/projects/my-awesome-project$ git add newfile
dci@dci-laptop:~/projects/my-awesome-project$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
	modified:   README.adoc
	new file:   newfile
```

DCI

Git workflow

The *git add* command accepts any path - including folders

What did the . and .. shortcuts mean?

```
dci@dci-laptop:~/projects/my-awesome-project$ git add .
dci@dci-laptop:~/projects/my-awesome-project$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
	modified:   README.adoc
	new file:   newfile
```

Git workflow

Remember commits need a commit message

```
$ git commit
```

You can provide an *inline* commit message directly **-m**

```
dci@dci-laptop:~/projects/test$ git commit -m “Add demo feature”

[main 7e1cf288f0] Add demo feature
 2 files changed, 1 deletion(-)
 create mode 100644 newfile
```

Git workflow

Your commit now exists **only** in your local repository

You can commits many times - projects or companies can have rules for commits

- only commit working code
- only commit a logical set of changes
- max 50 characters
- use complete sentences
- start your message with an issue ID
- write in imperative: **Fix bug** not **Fixed bug**

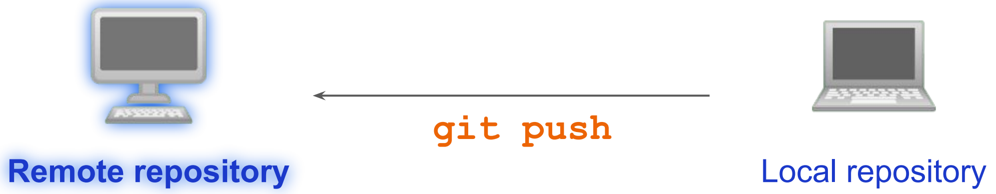
AD-192 Add autosuggestions to search

DCI

Git workflow

If your repository is cloned or connected to another repository
You can send your commits to the remote with

```
$ git push
```

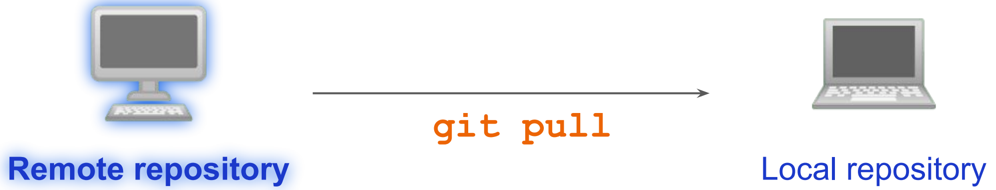


Git workflow

To updates your local repository with from the remote repository

```
$ git pull
```

There is a conflict if you changed a file someone has changed in the remote.
It's best to handle the conflicts as soon as possible!



Git workflow

Read the commit history with **git log**

Each commit has an id, an author, a date and a message
To exit the log, press **q**

```
dci@dci-laptop:~/projects/my-awesome-project$ git log
commit 7e1cf288f03e8481b77d9505d9098a994b2bce9e (HEAD -> main)
Author: Joel Peltonen <joel.peltonen@digitalcareerinstitute.org>
Date:   Fri Jul 9 17:45:11 2021 +0200

    Add demo feature

commit 7a1c923fecacd4abafda82fa8c2fc6be3bc4e761 (origin/main, origin/HEAD)
Merge: 0b604f5e3b 3de58c2340
Author: Andy Example <example@example.org>
Date:   Fri Jul 9 14:18:18 2021 +0100

    Merge branch '2.5.x'

     Closes gh-27226
```

At the core of the lesson

```
$ git status        # what is happening
$ git add           # stage changes
$ git commit        # commit changes
$ git push          # push to remote
$ git pull          # pull from remote
$ git log           # view history
```

Branching

1st week	2nd week
Intro and Overview	Versioning (Branches)
View, Navigate, Create, Change	Publishing
Installation	Collaboration
Versioning (Basics)	Review

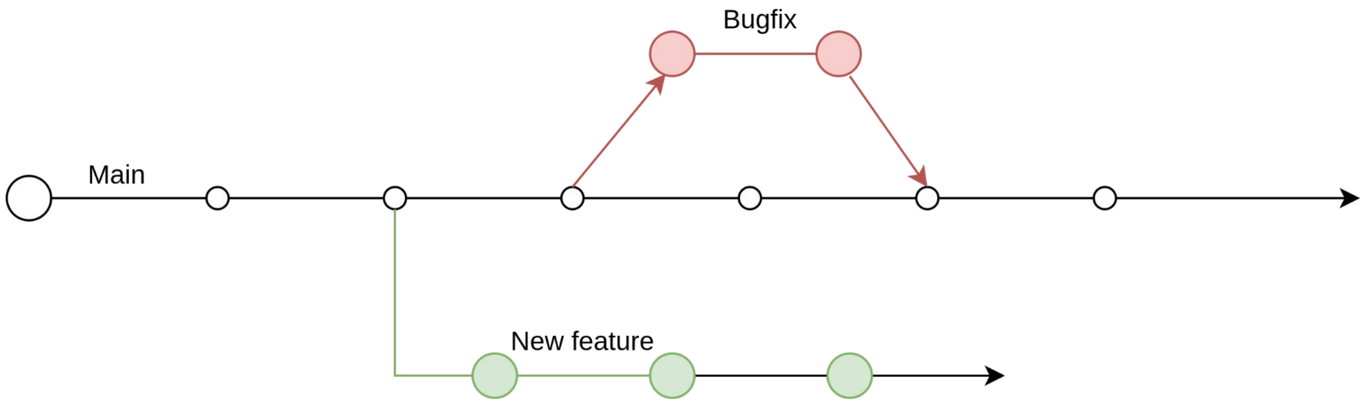
Class Room Rules!

Code of conduct workshop!

Branching

VCS systems use **branching** - parallel versions of the codebase

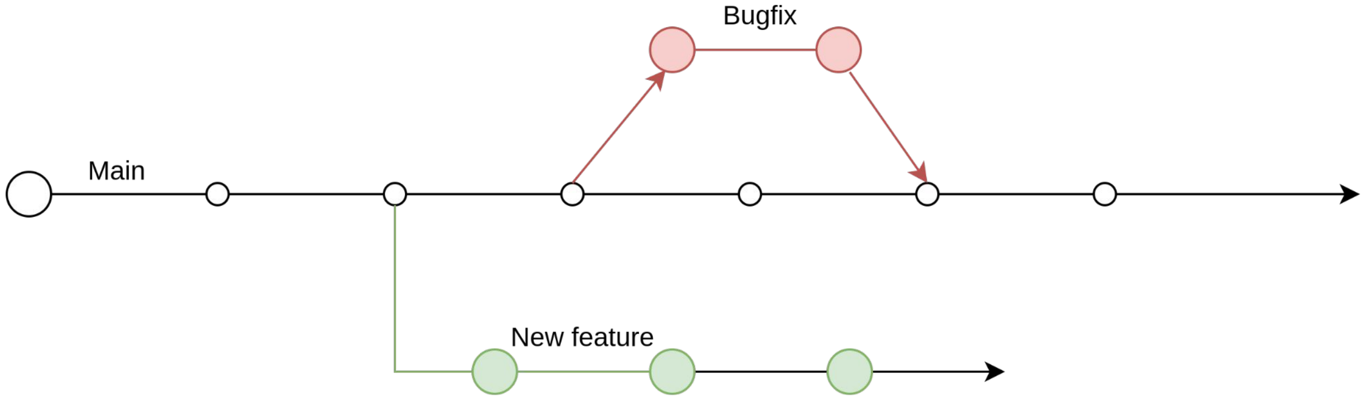
- Often one Main branch
- Branches start from other branches
- Branches merge with other branches



Branching

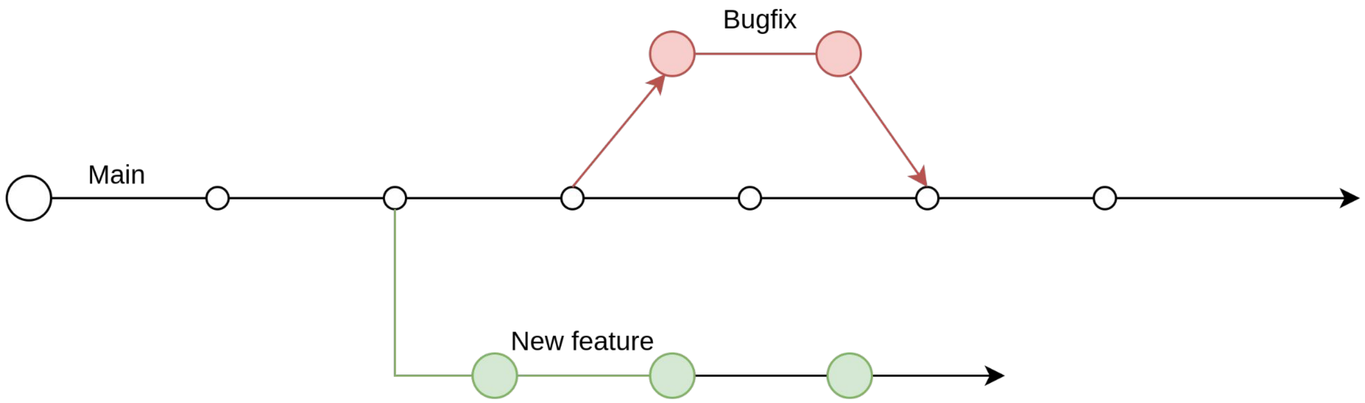
- Often there is a branching strategy for a project, **for example**
  - One main branch
  - One branch per feature/bugfix
  - One branches for every published version

- Often one person works on one issue
  - Otherwise; one branch per person

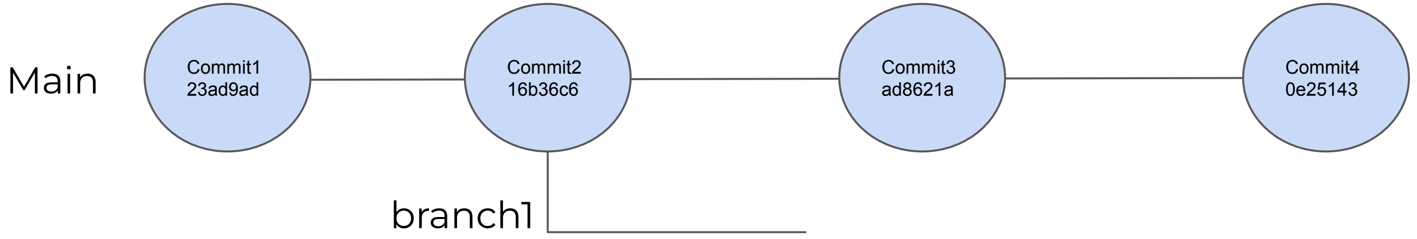


Branching

- When you create a branch, only you have it…
- …until you push it to another repository



Git branch

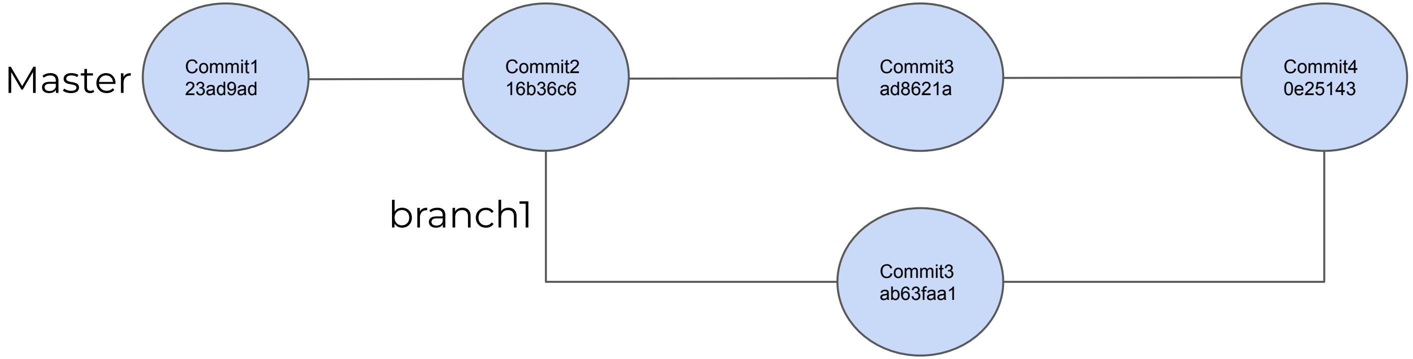


Create new branch called branch1 start from commit2

git checkout -b <branch name> [start point]

```
$ git checkout -b branch1 16b36c6
```

Git merge



```
$ git merge branch1
```

DCI

Git workflow

Let’s practice working with branches!

- Creating a branch
- Renaming current branch
- Listing branches
- Switching branches
- Deleting a branch
- Merging changes

```
$ git checkout -b <new-branch-name>
$ git branch -m <branch-name>
$ git branch [-a]
$ git checkout <branch-name>
$ git branch -d <branch-name>
$ git merge <branch-name>
```

- Publishing a branch

```
$ git push -u origin <branch-name>
```

`Note: more than one way to pet a cat; there are alternatives to these`

Git workflow

Branches are fundamental to Git and there's lots you can do with them!

create

delete

checkout

rename

compare

DCI

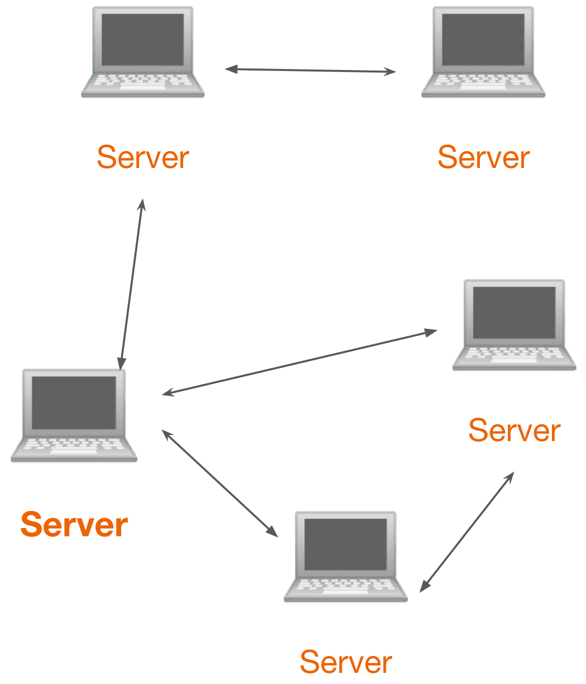
Publishing

1st week	2nd week
Intro and Overview	Versioning (Branches)
View, Navigate, Create, Change	Publishing
Installation	Collaboration
Versioning (Basics)	Review

Publishing

THE INTERNET

- Network of computers
- Computers *speak* TCP/IP
    - Set of protocols
    - Global standard
    - Like a basic language
- Computers have addresses
    - More like phone numbers



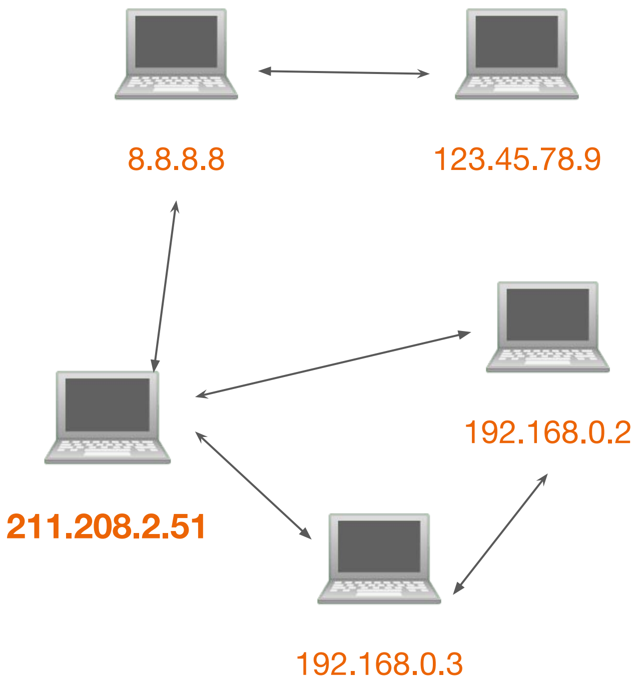
Publishing

The Internet

Computer addresses

- IP (version 4)
  - 8.8.8.8 (public)
  - 211.208.2.51 (public)
  - 192.168.0.2 (private)
  - 127.0.0.1 (private)

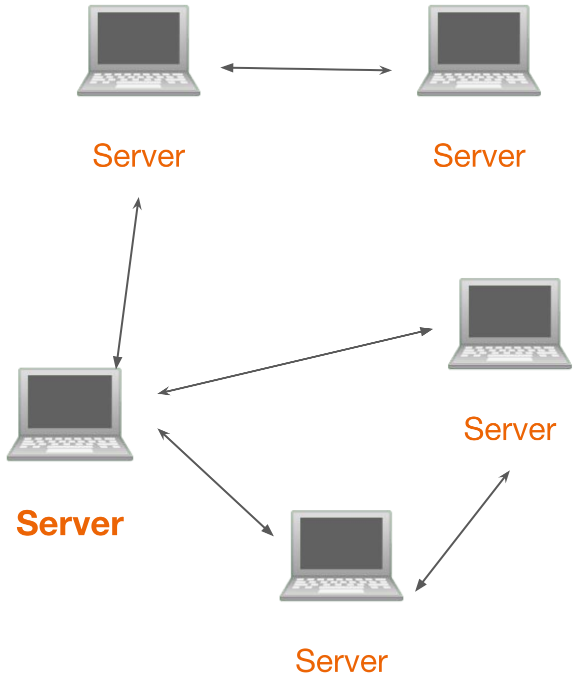
- Four numbers between 0 - 255
- IPv6 also exists…
  - *2001:db8::8a2e:370:733*
  - *2607:f0d0:1002:0051:0000:0000:0000:0004*



Publishing

The Internet

- IP address: hard to remember
  - 211.208.2.51
- Domains: easy
  - example.org
- Domain names are controlled by DNS



DCI **Publishing**

Anatomy of an URL

Protocol **Port** **Query string**

```
https://test.example.org:80/dogs/poodle?color=white&puppy=false#first
```

Domain
(subdomain: test)
(domain name: example)
(TLD / top level domain: org)

Resource path

Hash

Common ports

80 - normal web traffic (http)
443 - encrypted web traffic (https)

22 - SSH access
21 - FTP File transfer protocol

DCI

GitHub

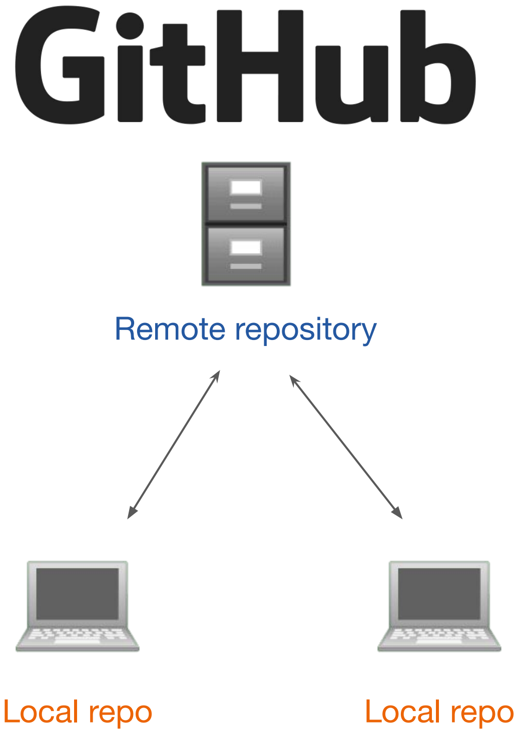
“Usually there is a central repository”

GitHub

www.github.com

Let’s make an account!

(choose your username wisely)

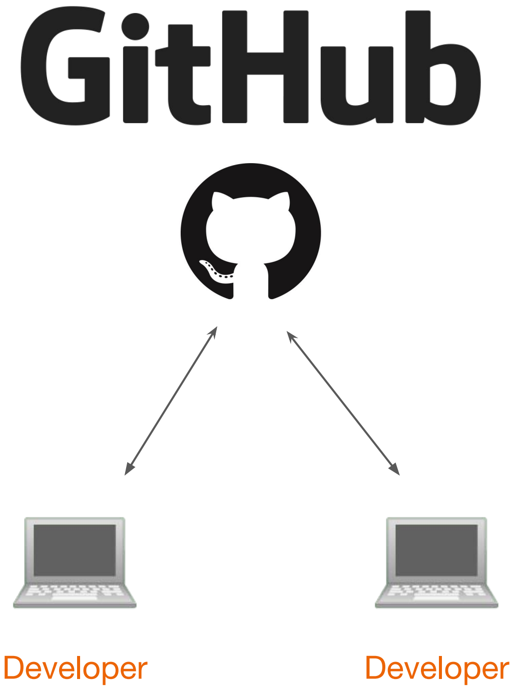


GitHub

Repositories in the cloud

- Public and private repos
    - Open source
    - Closed source
- Project management
    - Issue tracking
    - Pull requests
    - Code reviews
- Automation

https://github.com/torvalds/linux
https://github.com/microsoft/vscode
https://github.com/DigitalCareerInstitute/marketing-website



DCI

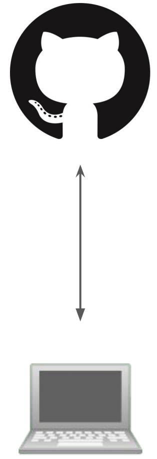
GitHub

To use GitHub, we must **authenticate**

For authentication we normally use **SSH**

- Secure Shell
- Commonly used to connect to servers
- Used for data transfer too

GitHub

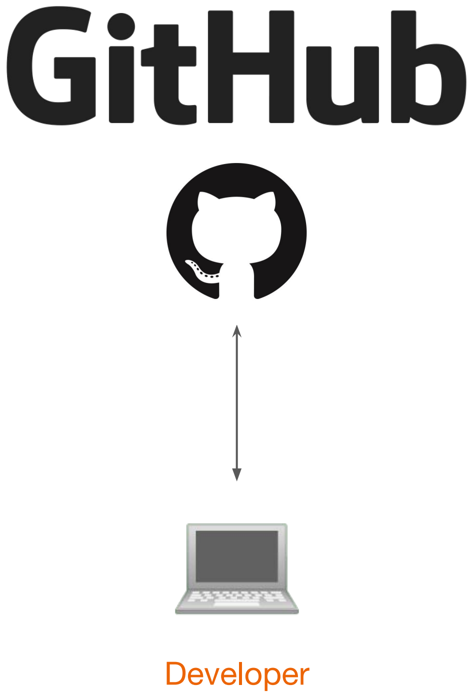


DCI

GitHub

An analogy for **SSH** keys

- You create a lock and a key on your computer
- Your keep the key secret
- You copy the lock to GitHub
- “Hey GitHub, only I can open this lock”
- The key is automatically used when connecting



Let’s make an SSH key

Git remote & clone

Shows all remotes:

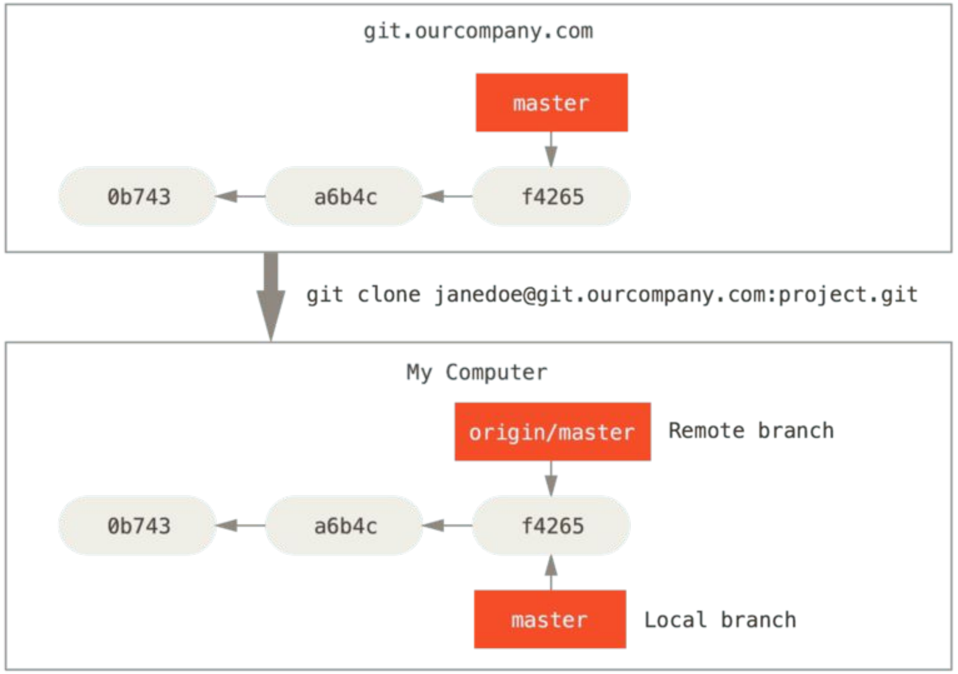
```
$ git remote
```

Add a remote:

```
$ git remote add <shortname> <url>
```

Clone an existing repository

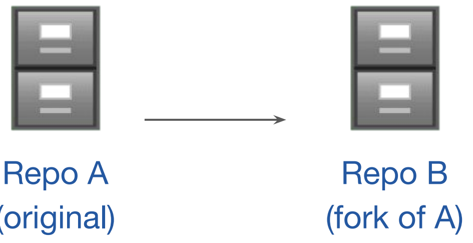
```
$ git clone <url>
```



GitHub

GitHub… forking?

- To “fork” a project is to clone it **in GitHub**
- Possible for open source projects



Let’s practice forking and make a pull request!

At the core of the lesson

GitHub

- Git repositories in the cloud
- Open and closed source
- Public and private repositories
- Tools for project and repo management
- SSH Authentication

Practice time!

1. Create a repository
2. Edit a README in GitHub
3. Clone a repo from GitHub
4. Push a commit
5. Create a branch and push it
6. Fork a repository and then delete the fork

Self Study

https://lab.github.com/curi-holdings/developer-beginner

Good luck, have fun!

Collaborating

1st week	2nd week
Intro and Overview	Versioning (Branches)
View, Navigate, Create, Change	Publishing
Installation	Collaborating
Versioning (Basics)	Review

Introduction:
Working together with git and GitHub

Digital Career Institute

Practice time!

GitHub helps collaboration

- **Pull Requests** & **Code Reviews**
  - Peter makes a branch to fix a bug
  - Peter pushes his branch
  - Peter creates a Pull Request to merging his work
  - Anne reviews Peters code
  - Anne checks the target branch is correct

- With GitHub you can **protect** branches - no pushes to the main without review!

Practice time!

- Many people working on one project causes **Conflicts** - this is normal 🙂
  - Anna and Peter both edited *index.html*
  - Their work (branches) are being merged
  - Conflicts need to be **resolved**
    - Keep the version from Anna?
    - Keep the version from Peter?
    - Or manually edit the code, merging both together

- Often conflicts happen during Code Review

DCI

Open source

Design

Write Documents

Organize Issues

How can you contribute?

Translate

Help others

Code

Practice time!

1. Create & clone a repo from GitHub
2. Make a local branch
3. Change the text in <h1> (index.html)
4. Push your branch
5. Create a Pull Request
6. Let’s review a Pull Request together

Review

1st week	2nd week
Intro and Overview	Versioning (Branches)
View, Navigate, Create, Change	Publishing
Installation	Collaborating
Versioning (Basics)	Review

Review

Discussion

1. **The terminal**
2. **Markdown**
3. **Versioning / git**
4. **GitHub**

DCI Review

Assessment time!

Review

Life after BDL

1.	UIB	-	User Interface Basics
2.	PB	-	Programming Basics
3.	SPA	-	Single Page Application
4.	BE	-	Backend
5.	FP	-	Final Project

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute