C++ Software Engineering

for engineers of other disciplines

Module 6

"Software Development Essentials"

*1st Lecture: *nix*

ALTEN

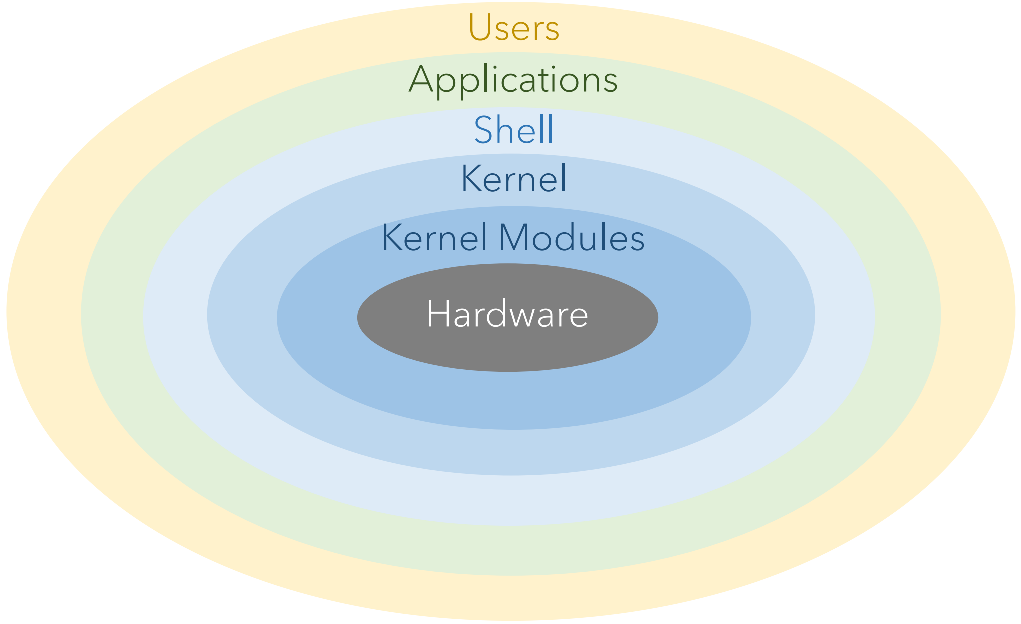
Spring 2022

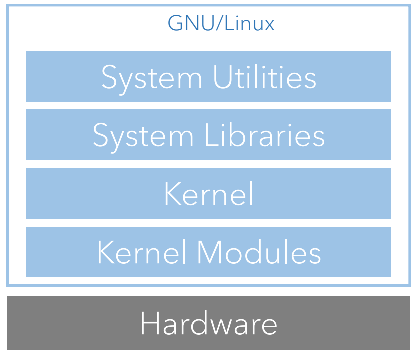
Gothenburg, Sweden

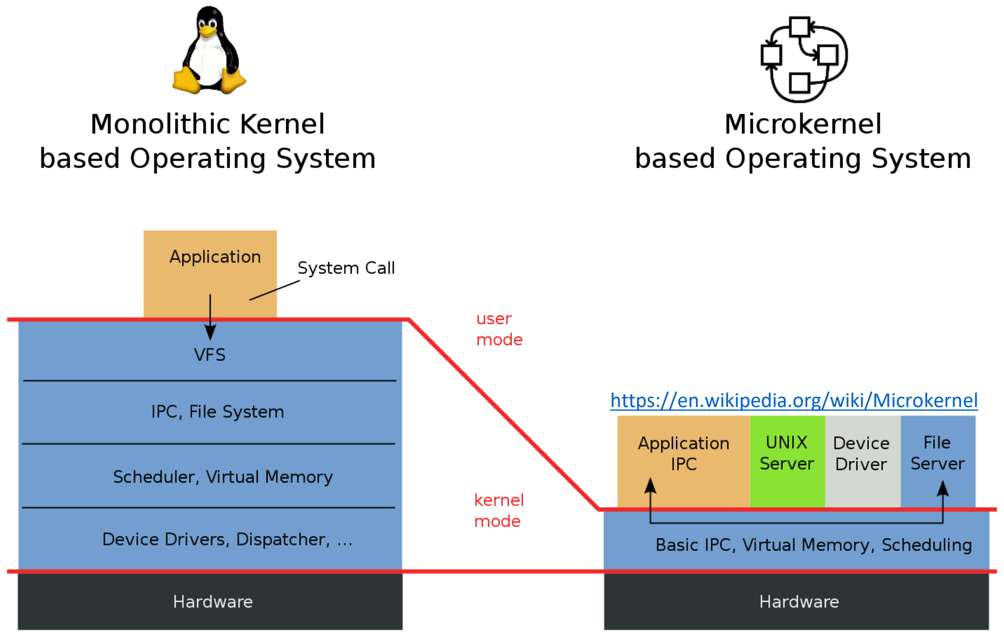
*nix

- ***Unix*** developed in 1970s, lead by the same people who invented C programming language.

- Its design is based on *Unix Philosophy* to implement *minimalist, and modular* software.







https://en.wikipedia.org/wiki/Microkernel

"The group coined the name Unics for Uniplexed Information and Computing Service (pronounced "eunuchs"), as a pun on Multics (an influential early operating system) […] "no one can remember" the origin of the final spelling Unix […] In 1983, Richard Stallman announced the GNU (short for "GNU's Not Unix") project, an ambitious effort to create a free software Unix-like system; "free" in the sense that everyone who received a copy would be free to use, study, modify, and redistribute it. The GNU project's own kernel development project, GNU Hurd, had not yet produced a working kernel, but in 1991 Linus Torvalds released the kernel Linux as free software." https://en.wikipedia.org/wiki/Unix

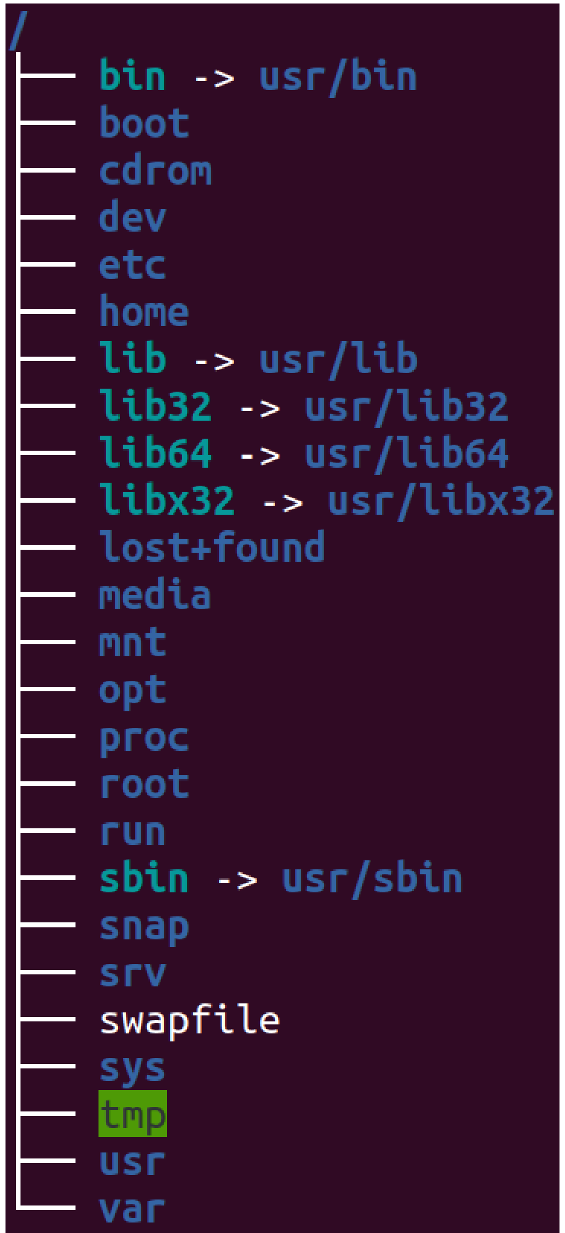
FHS

- Directory structure in *Linux* is defined by **F**ile **H**ierarchy **S**tandard.

Folder	Description
`/`	Primary hierarchy root and root directory of the entire file system hierarchy.
`/etc`	Host-specific system-wide configuration files
`/home`	Users' home directories, containing saved files, personal settings, etc.
`/lib`<*qual*>	Libraries essential for the binaries in **`/bin`** and **`/sbin`**. <qual> represents alternate format essential libraries. These are typically used on systems that support more than one executable code format.
`/usr`	Secondary hierarchy for read-only user data; contains the majority of (multi-)user utilities and applications. Should be shareable and read-only.

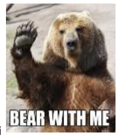
https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard

- *Root directory* is the "top-most" directory in the hierarchy.
- **`swapfile`** is file on the storage which Linux kernel could use as *virtual memory* usually when *RAM* space is low.
- In Linux, it is possible to make both *hard* and *soft links* to files and folders using **`ln`**.
- When **`make install`** is called, the compile output and other necessary artefacts would be *moved* to appropriate folders.



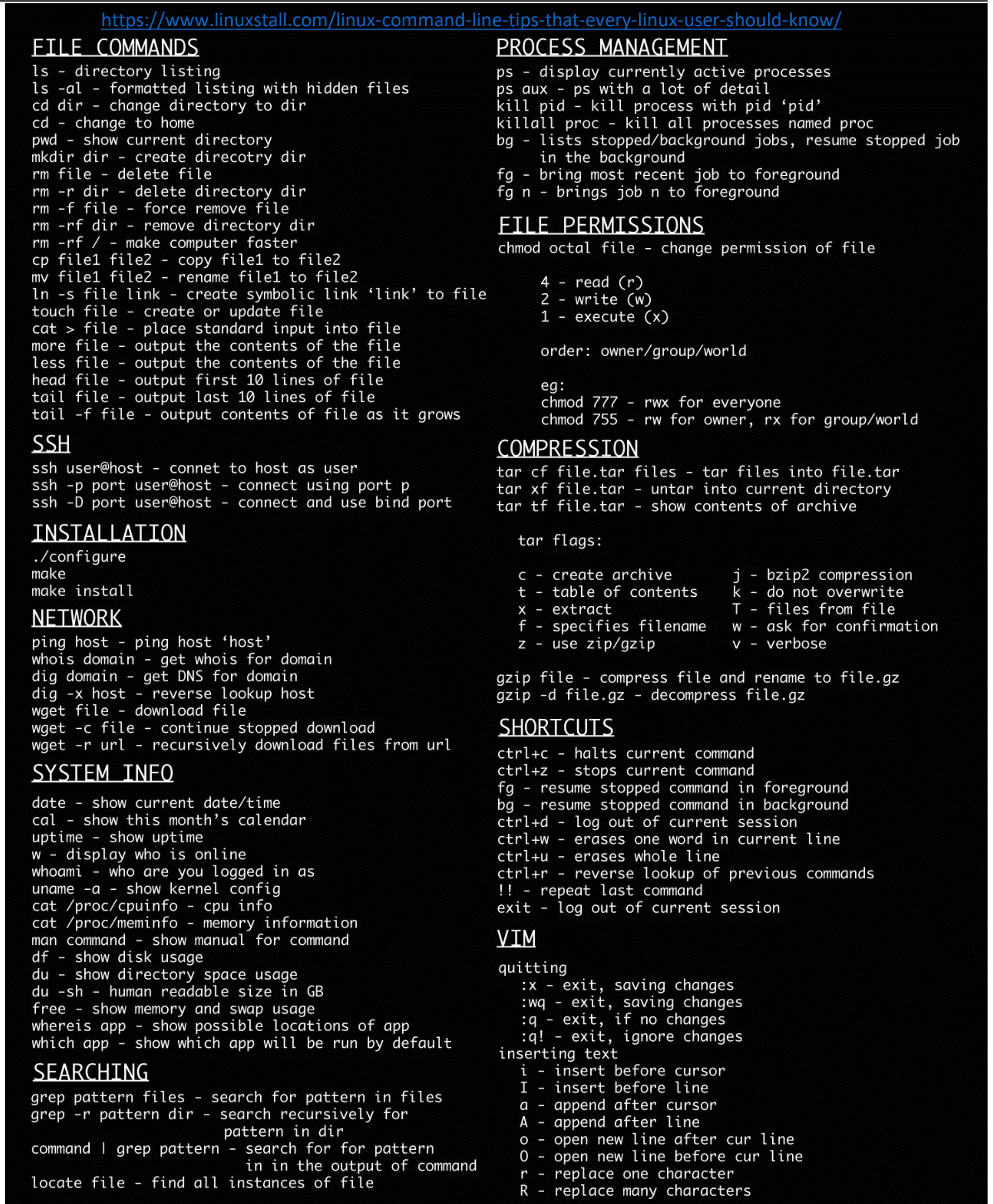
```
/
├── bin -> usr/bin
├── boot
├── cdrom
├── dev
├── etc
├── home
├── lib -> usr/lib
├── lib32 -> usr/lib32
├── lib64 -> usr/lib64
├── libx32 -> usr/libx32
├── lost+found
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin -> usr/sbin
├── snap
├── srv
├── swapfile
├── sys
├── tmp
├── usr
└── var
```

Terminal Emulator



- *“Emulates a video terminal within some other architecture.”*
- *“Allows users to access a UNIX shell while remaining on their graphical desktop.”*
- Multi-user, Multi-session.
- Each session is a separate *environment.*
- ***Bash*** and *Z are* the most common command languages used for *Shell Scripting – they are very compatible.*

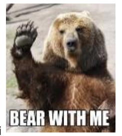
“A command language is a language for job control in computing. It is a domain-specific and interpreted language; common examples of a command language are shell or batch programming languages. ” https://en.wikipedia.org/wiki/Command_language



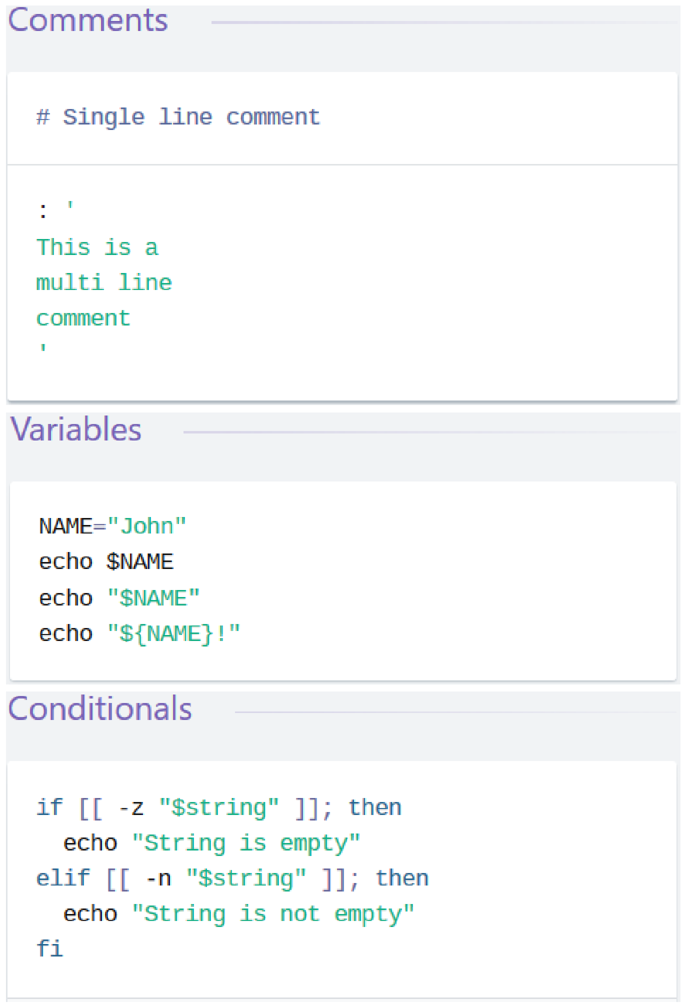
https://www.linuxstall.com/linux-command-line-tips-that-every-linux-user-should-know/

```
FILE COMMANDS
ls - directory listing
ls -al - formatted listing with hidden files
cd dir - change directory to dir
cd - change to home
pwd - show current directory
mkdir dir - create direcotry dir
rm file - delete file
rm -r dir - delete directory dir
rm -f file - force remove file
rm -rf dir - remove directory dir
rm -rf / - make computer faster
cp file1 file2 - copy file1 to file2
mv file1 file2 - rename file1 to file2
ln -s file link - create symbolic link ‘link’ to file
touch file - create or update file
cat > file - place standard input into file
more file - output the contents of the file
less file - output the contents of the file
head file - output first 10 lines of file
tail file - output last 10 lines of file
tail -f file - output contents of file as it grows

SSH
ssh user@host - connet to host as user
ssh -p port user@host - connect using port p
ssh -D port user@host - connect and use bind port

INSTALLATION
./configure
make
make install

NETWORK
ping host - ping host ‘host’
whois domain - get whois for domain
dig domain - get DNS for domain
dig -x host - reverse lookup host
wget file - download file
wget -c file - continue stopped download
wget -r url - recursively download files from url

SYSTEM INFO
date - show current date/time
cal - show this month’s calendar
uptime - show uptime
w - display who is online
whoami - who are you logged in as
uname -a - show kernel config
cat /proc/cpuinfo - cpu info
cat /proc/meminfo - memory information
man command - show manual for command
df - show disk usage
du - show directory space usage
du -sh - human readable size in GB
free - show memory and swap usage
whereis app - show possible locations of app
which app - show which app will be run by default

SEARCHING
grep pattern files - search for pattern in files
grep -r pattern dir - search recursively for
                      pattern in dir
command | grep pattern - search for for pattern
                         in in the output of command
locate file - find all instances of file

PROCESS MANAGEMENT
ps - display currently active processes
ps aux - ps with a lot of detail
kill pid - kill process with pid ‘pid’
killall proc - kill all processes named proc
bg - lists stopped/background jobs, resume stopped job
     in the background
fg - bring most recent job to foreground
fg n - brings job n to foreground

FILE PERMISSIONS
chmod octal file - change permission of file

     4 - read (r)
     2 - write (w)
     1 - execute (x)

     order: owner/group/world

     eg:
     chmod 777 - rwx for everyone
     chmod 755 - rw for owner, rx for group/world

COMPRESSION
tar cf file.tar files - tar files into file.tar
tar xf file.tar - untar into current directory
tar tf file.tar - show contents of archive

  tar flags:

  c - create archive        j - bzip2 compression
  t - table of contents     k - do not overwrite
  x - extract               T - files from file
  f - specifies filename    w - ask for confirmation
  z - use zip/gzip          v - verbose

gzip file - compress file and rename to file.gz
gzip -d file.gz - decompress file.gz

SHORTCUTS
ctrl+c - halts current command
ctrl+z - stops current command
fg - resume stopped command in foreground
bg - resume stopped command in background
ctrl+d - log out of current session
ctrl+w - erases one word in current line
ctrl+u - erases whole line
ctrl+r - reverse lookup of previous commands
!! - repeat last command
exit - log out of current session

VIM
quitting
  :x - exit, saving changes
  :wq - exit, saving changes
  :q - exit, if no changes
  :q! - exit, ignore changes
inserting text
  i - insert before cursor
  I - insert before line
  a - append after cursor
  A - append after line
  o - open new line after cur line
  O - open new line before cur line
  r - replace one character
  R - replace many characters
```

Bash Scripting



- ***Bash*** *"Bourne-Again SHell"* is the command language interpreter for GNU – it is also a programming language.



Comments

```
# Single line comment
```

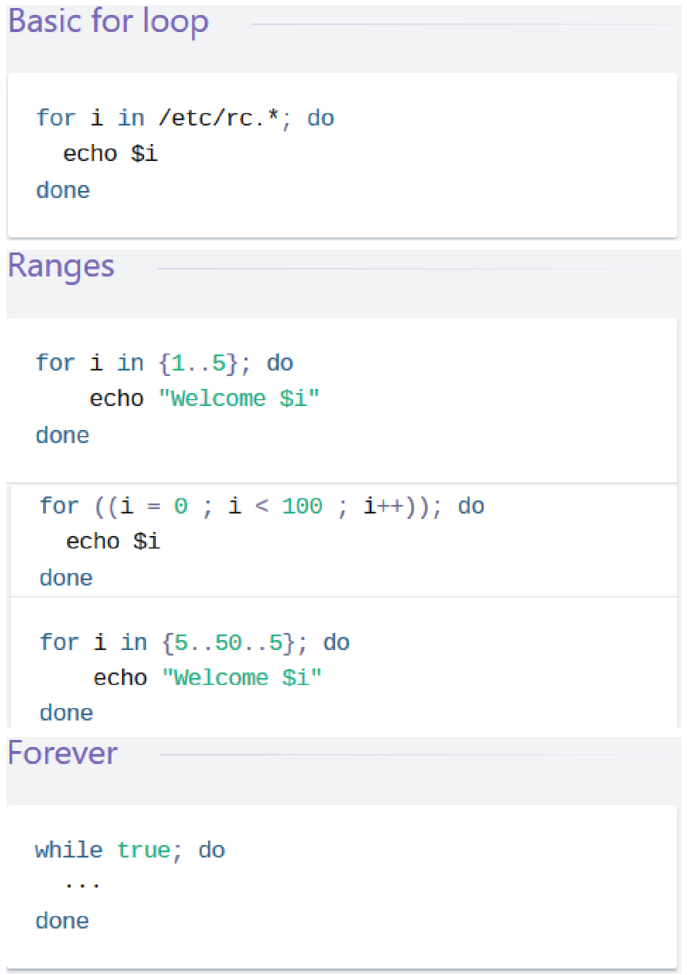
```
: '
This is a
multi line
comment
'
```

Variables

```
NAME="John"
echo $NAME
echo "$NAME"
echo "${NAME}!"
```

Conditionals

```
if [[ -z "$string" ]]; then
  echo "String is empty"
elif [[ -n "$string" ]]; then
  echo "String is not empty"
fi
```



Basic for loop

```
for i in /etc/rc.*; do
  echo $i
done
```

Ranges

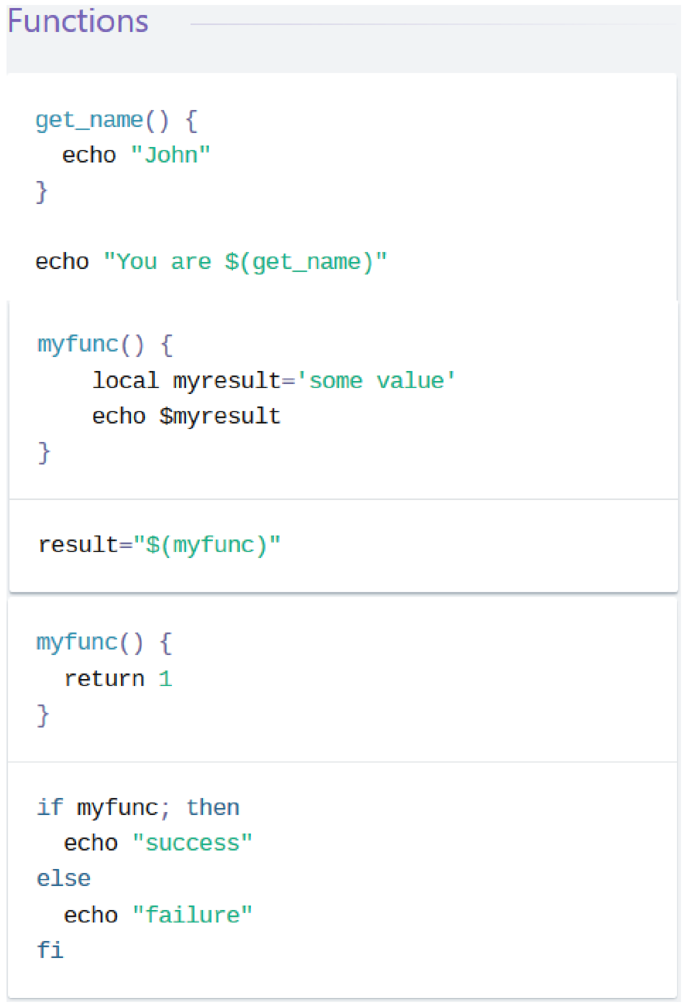
```
for i in {1..5}; do
    echo "Welcome $i"
done
```

```
for ((i = 0 ; i < 100 ; i++)); do
  echo $i
done
```

```
for i in {5..50..5}; do
    echo "Welcome $i"
done
```

Forever

```
while true; do
  ...
done
```



Functions

```
get_name() {
  echo "John"
}

echo "You are $(get_name)"
```

```
myfunc() {
    local myresult='some value'
    echo $myresult
}
```

```
result="$(myfunc)"
```

```
myfunc() {
  return 1
}
```

```
if myfunc; then
  echo "success"
else
  echo "failure"
fi
```

https://devhints.io/bash

Software Development In Linux

- Not very different from any other operating system, except:
  - Location for system libraries,
  - Interfaces for system libraries,
  - Interfaces to access hardware, and other *“system dependencies”*.
- Most of software development tools, like compilers, IDEs, *analyzing* tools and many more are *cross-platform.*
- Some software development tools are not hosted on the developer machines (server back-end), and they provide applications for interaction (front-end) for different operating systems.
- There are OS specific tools, we cover some of them in future lectures.

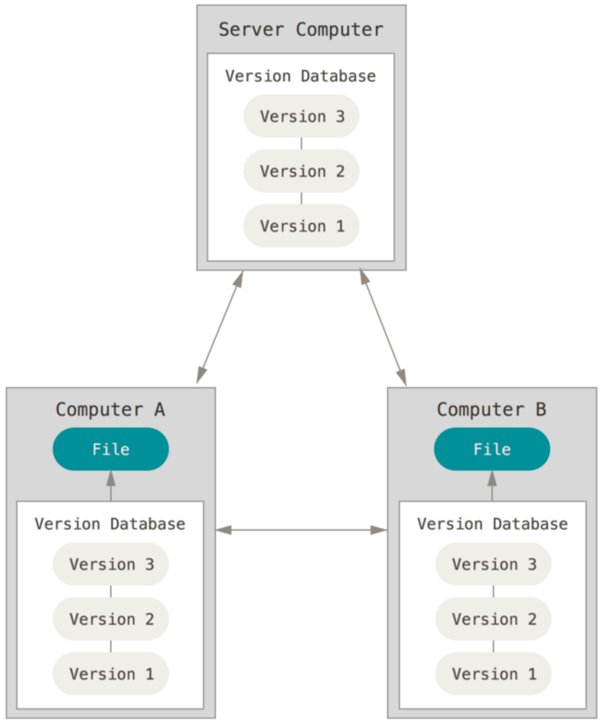
- *Version control* and *build automation* as a part of *continuous integration* are of the most common tools used while developing software.
- Implementing software interacting with the OS, such as drivers, are naturally very OS dependent.

Git

- Git is a distributed version control -- it *controls* changes to source codes and *other documents* in software development projects.

- In the usual setup, Git hosts the code on a server a.k.a. *remote (global, or central) repository*, while a copy exist on every developer's machine a.k.a. *local repository* – **it is the developer's responsibility to keep both in sync** for *very good* reasons.

- Git allows developers to:
  - **`pull`** content from a repository
  - **`commit`** changes to a repository
  - **`push`** to certain a repository
  - **`clone`** from a repository, make a **`branch`** or **`merge`** two.
  - **`checkout`** certain version of a file.

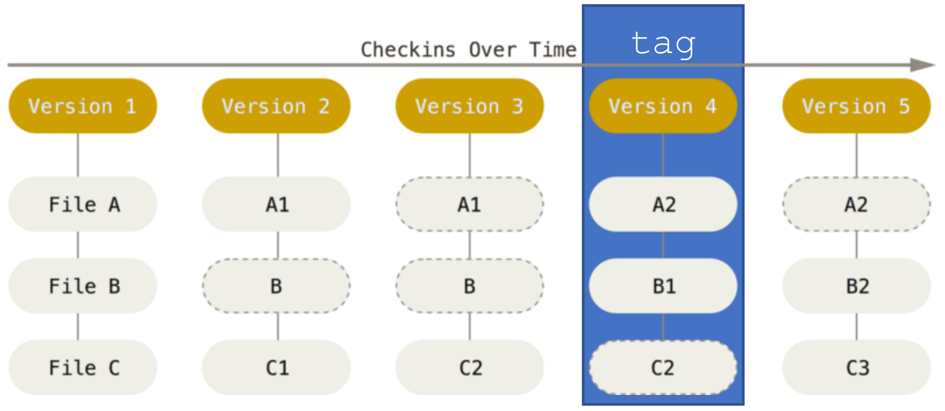


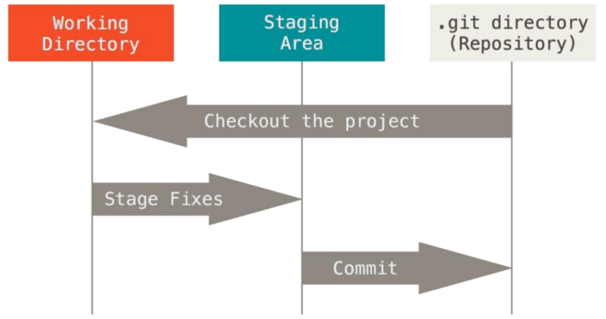
https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

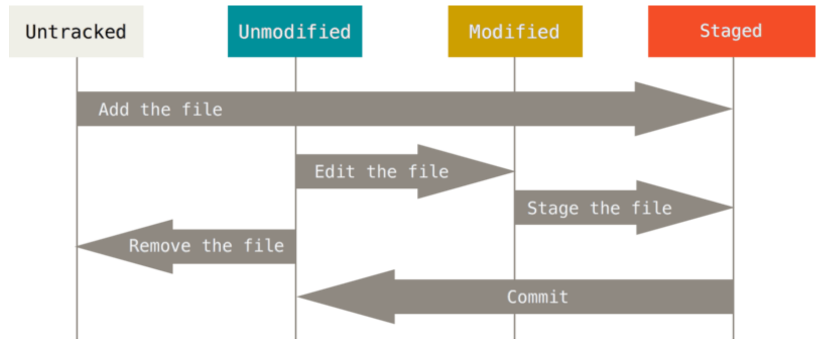
- Git was initially designed as a low-level version control system engine, yet it has since become a complete version-control system that is usable directly. There are other opensource implementations of Git as well.
- **`git`** application could be installed from **`apt`** repository.

Git -- modifications

- Given there is a local repository, all the modifications are local unless explicitly mentioned otherwise.
- Every modification and interaction is logged. *Integrity* of everything (files and meta information) is *guaranteed*.
- Each file could be either:
  - **`modified`**: changes to the file has not been *committed* to the local repository.
  - **`staged`**: changes are *staged* to be *committed* to your local repository.
  - **`committed`**: the changes are *committed;* the files is the same as the one in your local repository.
  - Untracked: these are the files git ignores.





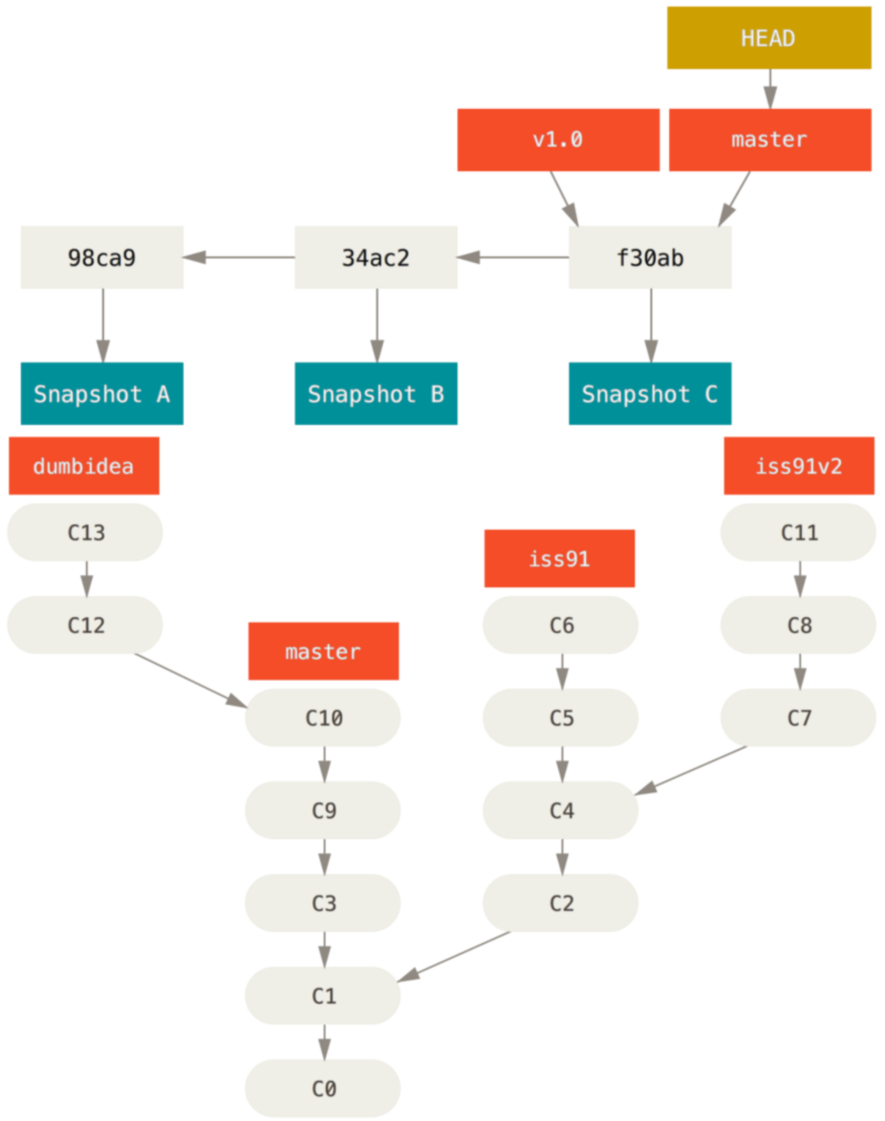


https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Git -- branch

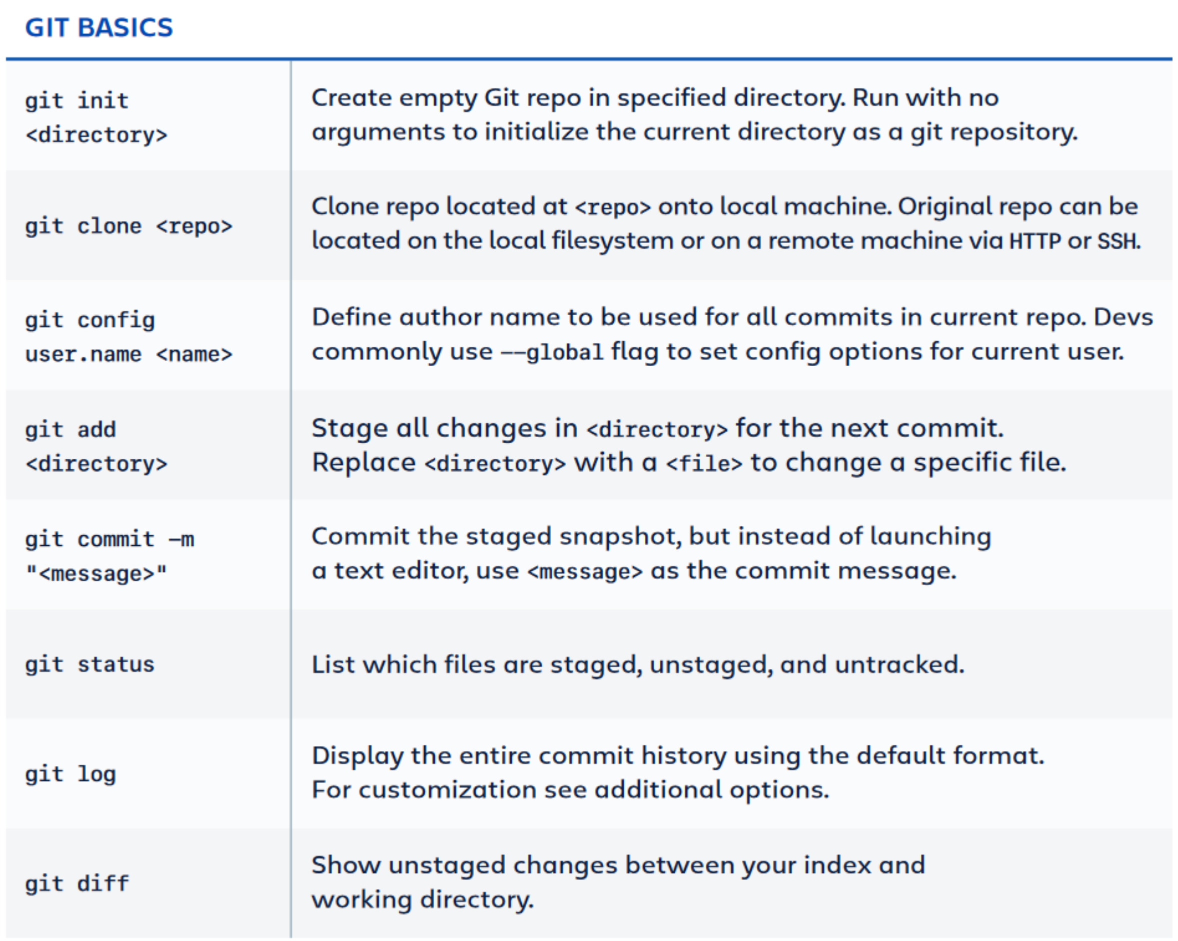
- Branch is a *lightweight* pointer to a commit – the default is **`master`**.
- **`HEAD`** is a pointer to the *working* branch. It could point to any other branch.
- **`origin`** is the identifier for the remote repository of the project, if any.
- Branches could be merged later if necessary, should there be any *conflicts* it should be *resolved.*

- Conflicts are different modifications on the same file. Resolving conflicts could be a tedious procedure. Apart from branch merging, modification on the remote repository could yield in conflicts, if not perform carefully. Prior to any commit to the remote repository, it is almost mandatory to first pull the latest version to avoid conflicts.
- Through out the conflict resolution process, versions of different branch could be chosen to be included in the *merged* branch.



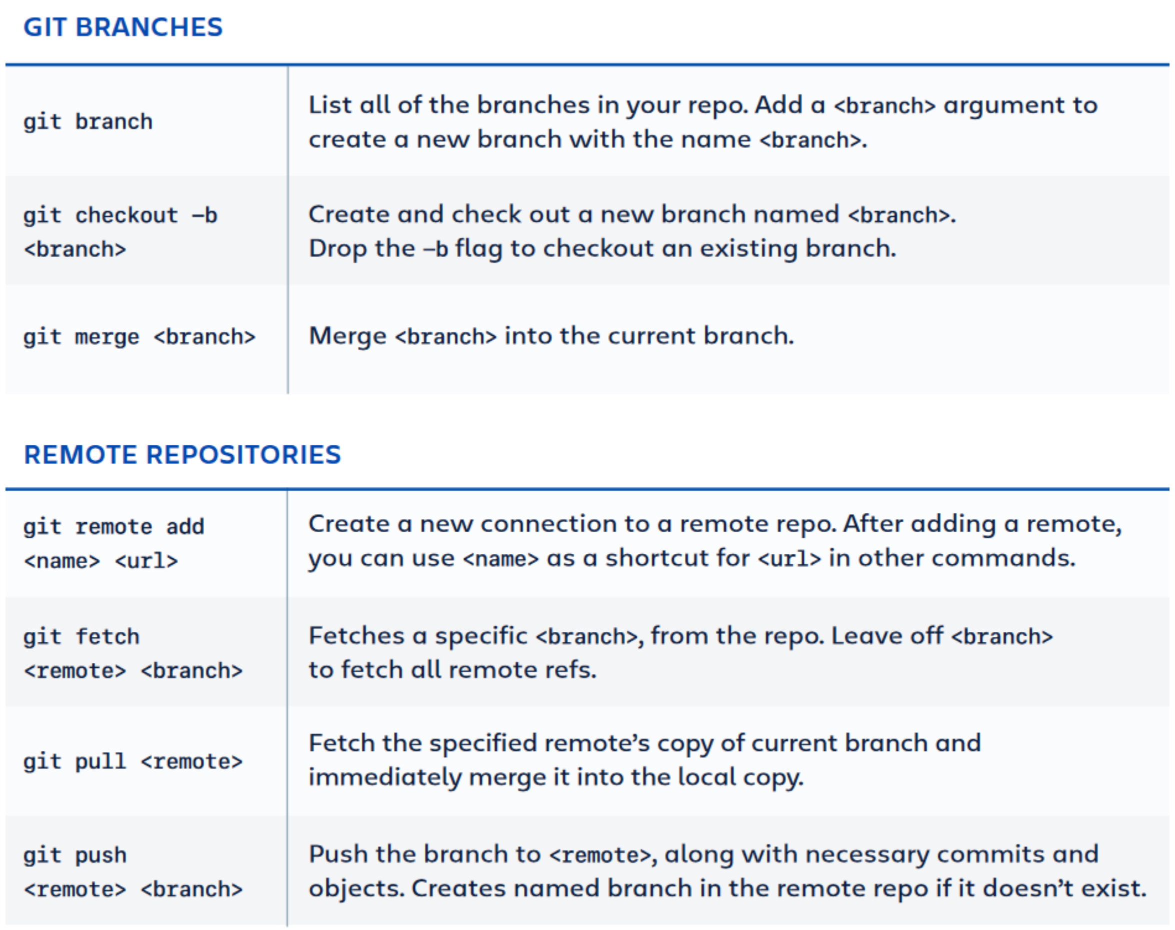
https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Git – cheat sheet



GIT BASICS

`git init <directory>`	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
`git clone <repo>`	Clone repo located at `<repo>` onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
`git config user.name <name>`	Define author name to be used for all commits in current repo. Devs commonly use `--global` flag to set config options for current user.
`git add <directory>`	Stage all changes in `<directory>` for the next commit. Replace `<directory>` with a `<file>` to change a specific file.
`git commit -m "<message>"`	Commit the staged snapshot, but instead of launching a text editor, use `<message>` as the commit message.
`git status`	List which files are staged, unstaged, and untracked.
`git log`	Display the entire commit history using the default format. For customization see additional options.
`git diff`	Show unstaged changes between your index and working directory.



GIT BRANCHES

`git branch`	List all of the branches in your repo. Add a `<branch>` argument to create a new branch with the name `<branch>`.
`git checkout -b <branch>`	Create and check out a new branch named `<branch>`. Drop the `-b` flag to checkout an existing branch.
`git merge <branch>`	Merge `<branch>` into the current branch.

REMOTE REPOSITORIES

`git remote add <name> <url>`	Create a new connection to a remote repo. After adding a remote, you can use `<name>` as a shortcut for `<url>` in other commands.
`git fetch <remote> <branch>`	Fetches a specific `<branch>`, from the repo. Leave off `<branch>` to fetch all remote refs.
`git pull <remote>`	Fetch the specified remote's copy of current branch and immediately merge it into the local copy.
`git push <remote> <branch>`	Push the branch to `<remote>`, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet

Git – cheat sheet

GIT CONFIG

`git config --global user.name <name>`	Define the author name to be used for all commits by the current user.
`git config --global user.email <email>`	Define the author email to be used for all commits by the current user.
`git config --global alias. <alias-name> <git-command>`	Create shortcut for a Git command. E.g. `alias.glog "log --graph --oneline"` will set `"git glog"` equivalent to `"git log --graph --oneline`.
`git config --system core.editor <editor>`	Set text editor used by commands for all users on the machine. `<editor>` arg should be the command that launches the desired editor (e.g., vi).
`git config --global --edit`	Open the global configuration file in a text editor for manual editing.

REWRITING GIT HISTORY

`git commit --amend`	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
`git rebase <base>`	Rebase the current branch onto `<base>`. `<base>` can be a commit ID, branch name, a tag, or a relative reference to `HEAD`.
`git reflog`	Show a log of changes to the local repository's `HEAD`. Add `--relative-date` flag to show date info or `--all` to show all refs.

UNDOING CHANGES

`git revert <commit>`	Create new commit that undoes all of the changes made in `<commit>`, then apply it to the current branch.
`git reset <file>`	Remove `<file>` from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
`git clean -n`	Shows which files would be removed from working directory. Use the `-f` flag in place of the `-n` flag to execute the clean.

GIT DIFF

`git diff HEAD`	Show difference between working directory and last commit.
`git diff --cached`	Show difference between staged changes and last commit

https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet

Gerrit

- Web-based code (*commit*) review tool – *reviewers* approve changes to be committed.

- Gerrit *integrates* with Git and provides a *richer graphical* experience to view commits.

- Gerrit also provides command line interface tool called **`repo`**.

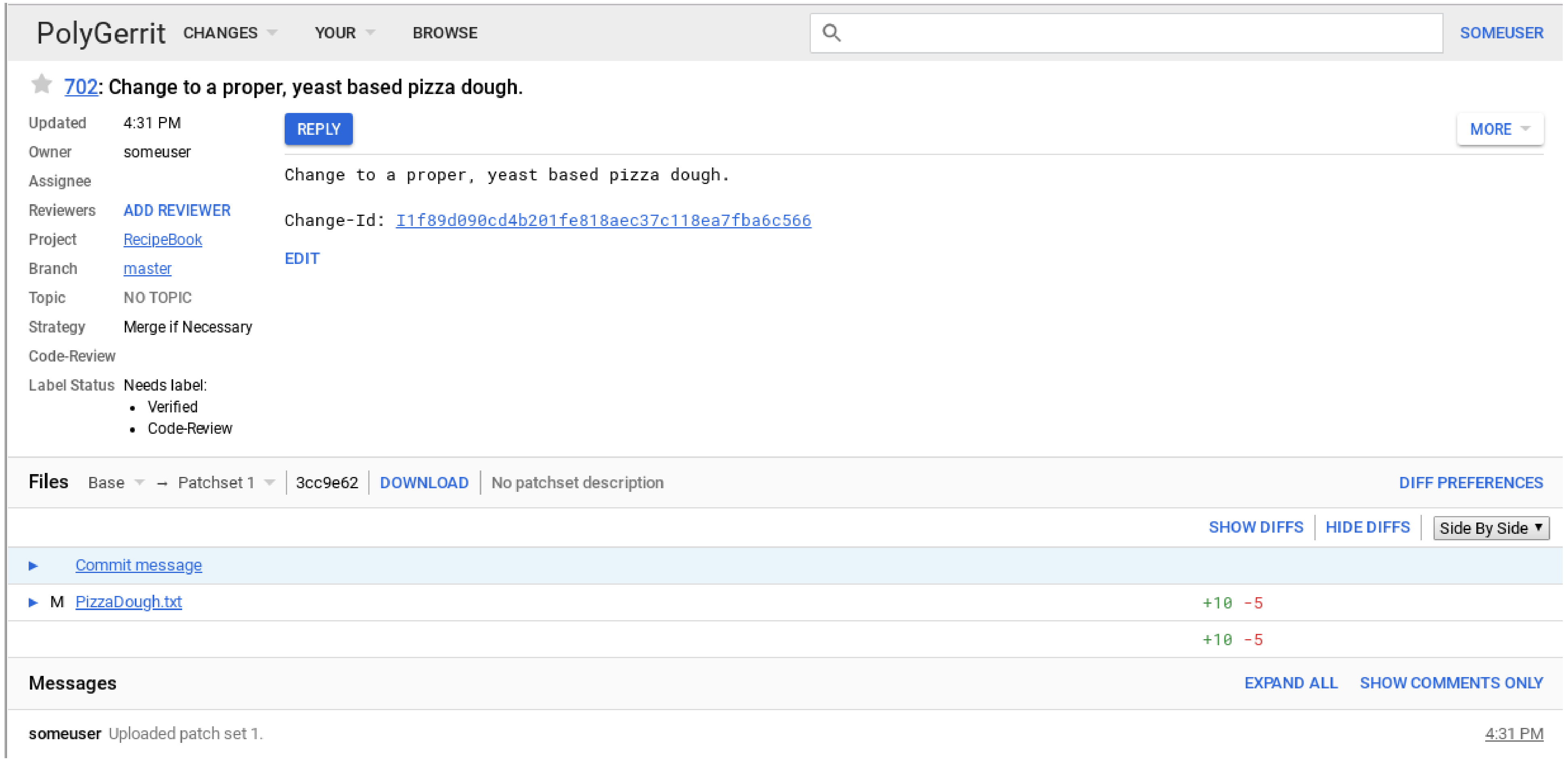
- Gerrit also provides Project Management:
  - Project Configuration
  - Access control
  - Project classification (Superproject, submodules, etc.)
  - Customized *submit rule* (in Prolog)

"Code review (sometimes referred to as peer review) is a software quality assurance activity in which one or several people check a program mainly by viewing and reading parts of its source code, and they do so after implementation or as an interruption of implementation." https://en.wikipedia.org/wiki/Code_review

- Gerrit is developed by Google and is an open source tool. It is possible to customize it to the organization need. Besides, it integrates with many other tools for extra functionalities, such as sending email or other types of notifications, and etc.

Gerrit – request a review

- Once a commit happens to the **`refs/for/`***<branchName>* of the remote repository, Gerrit creates a review.



702: Change to a proper, yeast based pizza dough.

Updated	4:31 PM
Owner	someuser
Assignee	
Reviewers	ADD REVIEWER
Project	RecipeBook
Branch	master
Topic	NO TOPIC
Strategy	Merge if Necessary
Code-Review	
Label Status	Needs label: • Verified • Code-Review

```
Change to a proper, yeast based pizza dough.

Change-Id: I1f89d090cd4b201fe818aec37c118ea7fba6c566
```

Files Base → Patchset 1 | 3cc9e62 | DOWNLOAD | No patchset description

	Commit message	
M	PizzaDough.txt	+10 -5
		+10 -5

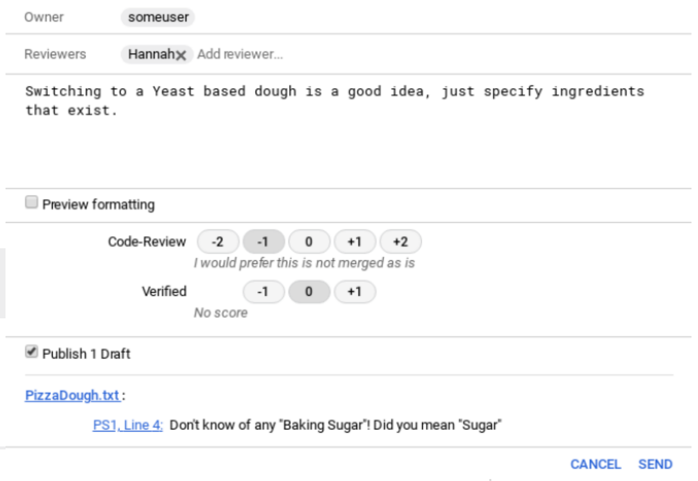
Messages

someuser Uploaded patch set 1. 4:31 PM

https://gerrit-review.googlesource.com/Documentation/intro-gerrit-walkthrough.html#_making_the_change

Gerrit – review a change

- Reviewers could review changes assigned to them by developers or find for themselves.
- Each change undergoes two checks: ***peer review*** and *automated* ***verification*** *step.*



Owner someuser

Reviewers Hannah Add reviewer...

```
Switching to a Yeast based dough is a good idea, just specify ingredients
that exist.
```

☐ Preview formatting

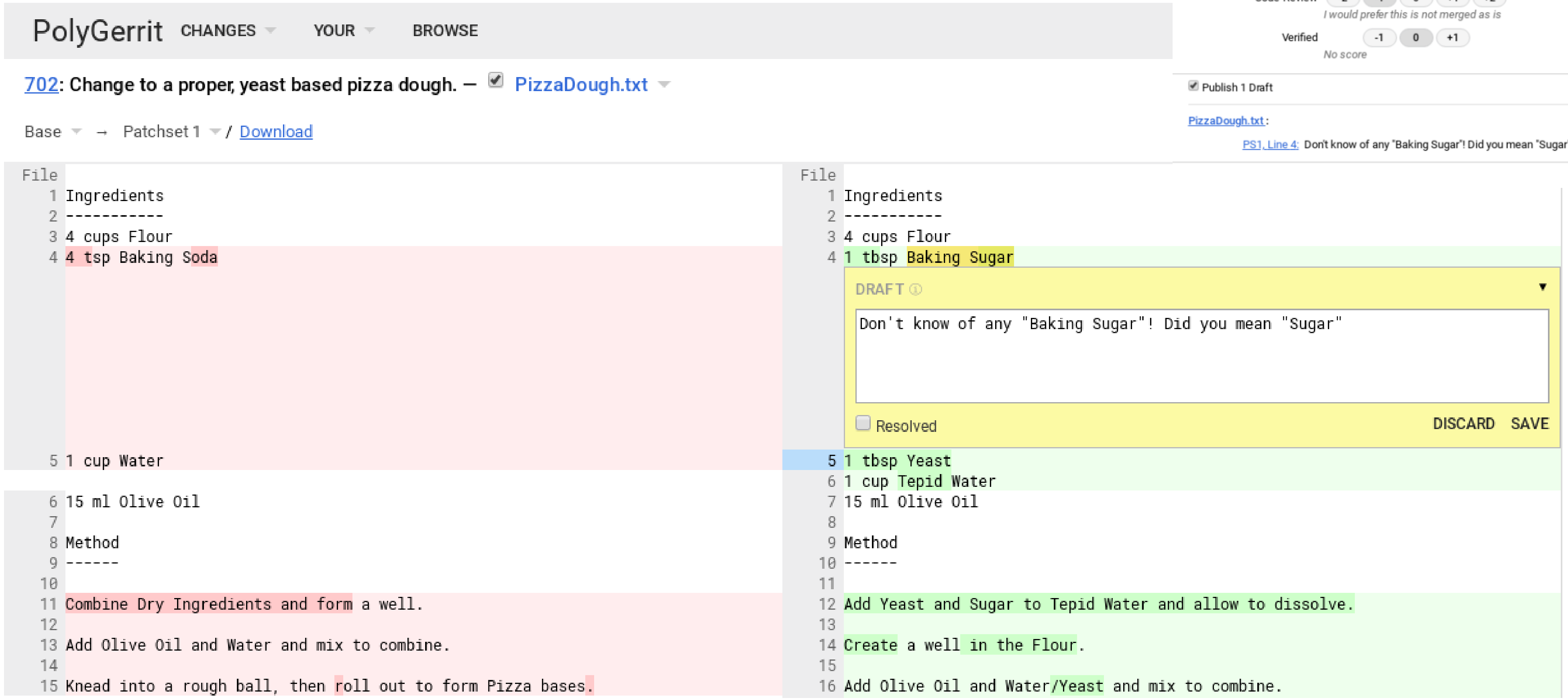
Code-Review -2 [✓ -1] 0 +1 +2
I would prefer this is not merged as is

Verified -1 [✓ 0] +1
No score

☑ Publish 1 Draft

PizzaDough.txt:
PS1, Line 4: Don't know of any "Baking Sugar"! Did you mean "Sugar"

CANCEL SEND



702: Change to a proper, yeast based pizza dough. — ☑ PizzaDough.txt

Base → Patchset 1 / Download

```
File
 1 Ingredients
 2 -----------
 3 4 cups Flour
 4 4 tsp Baking Soda
 5 1 cup Water
 6 15 ml Olive Oil
 7
 8 Method
 9 ------
10
11 Combine Dry Ingredients and form a well.
12
13 Add Olive Oil and Water and mix to combine.
14
15 Knead into a rough ball, then roll out to form Pizza bases.
```

```
File
 1 Ingredients
 2 -----------
 3 4 cups Flour
 4 1 tbsp Baking Sugar
 5 1 tbsp Yeast
 6 1 cup Tepid Water
 7 15 ml Olive Oil
 8
 9 Method
10 ------
11
12 Add Yeast and Sugar to Tepid Water and allow to dissolve.
13
14 Create a well in the Flour.
15
16 Add Olive Oil and Water/Yeast and mix to combine.
```

DRAFT

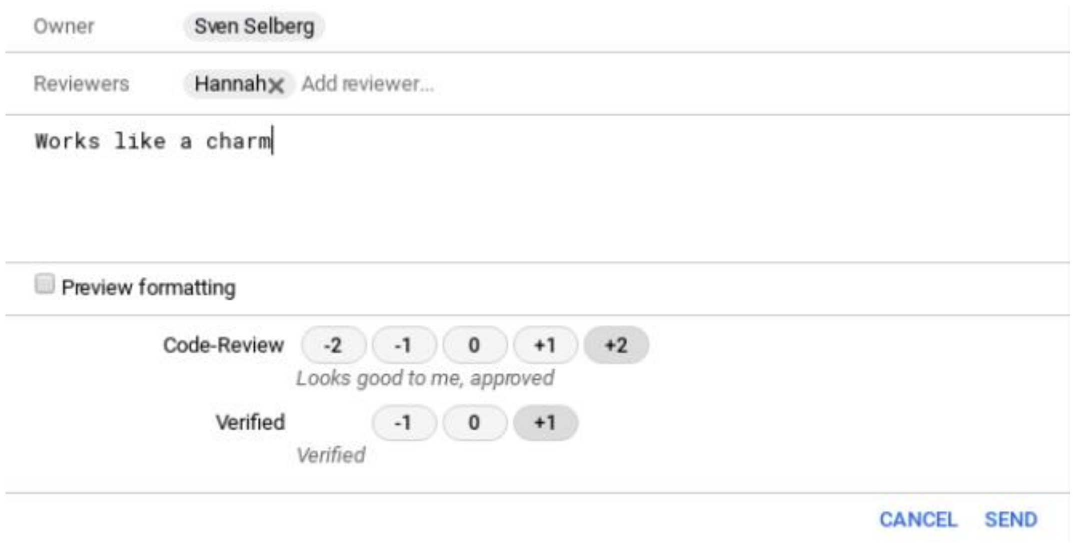
```
Don't know of any "Baking Sugar"! Did you mean "Sugar"
```

☐ Resolved DISCARD SAVE

https://gerrit-review.googlesource.com/Documentation/intro-gerrit-walkthrough.html#_creating_the_review

Gerrit – reworking & submitting

- If reviewers *reject* the changes, the developer shall:
  - Incorporate the comments
  - Checkout the commit
  - *Amend* the commit (*rebase* if necessary)
  - Push the commit to Gerrit
- Once the change are approved by the reviewer it needs to be *verified*.
- Verification is usually an automated step, reviewers with *verification permission* can perform manual verification if needed.



Owner: Sven Selberg

Reviewers: Hannah

Works like a charm

Preview formatting

Code-Review: +2 — Looks good to me, approved

Verified: +1 — Verified

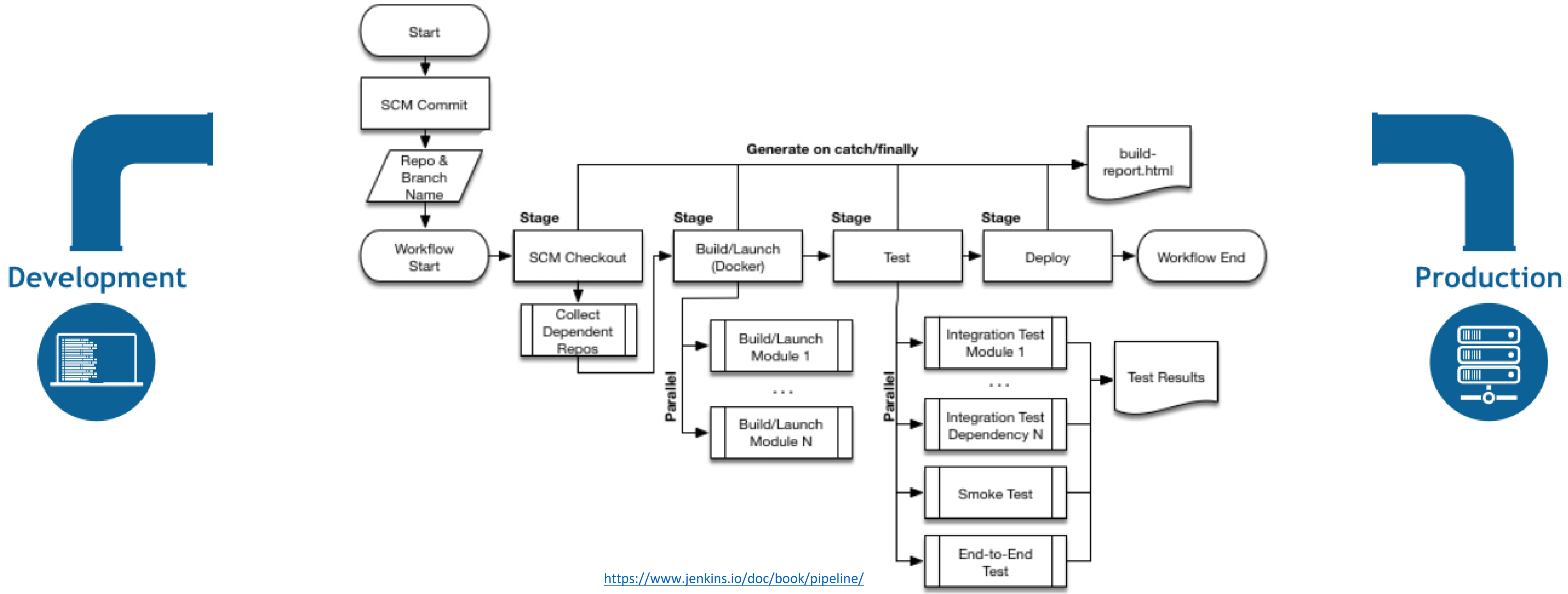
https://gerrit-review.googlesource.com/Documentation/intro-gerrit-walkthrough.html#_creating_the_review

- The verification procedure is usually triggered automatically once a reviewer approves the change. There are *plug-ins* for Gerrit which triggers build automation tools like Jenkins.

Jenkins

- Jenkins is an automation server for software development. It is plug-in based, and support many tools, such as Git, Gerrit, and Bash.

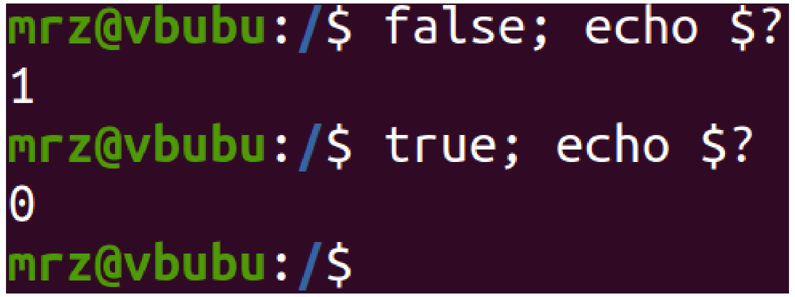
- It can provide a ***continuous delivery pipeline*** from *development* to *production.*



https://www.jenkins.io/doc/book/pipeline/

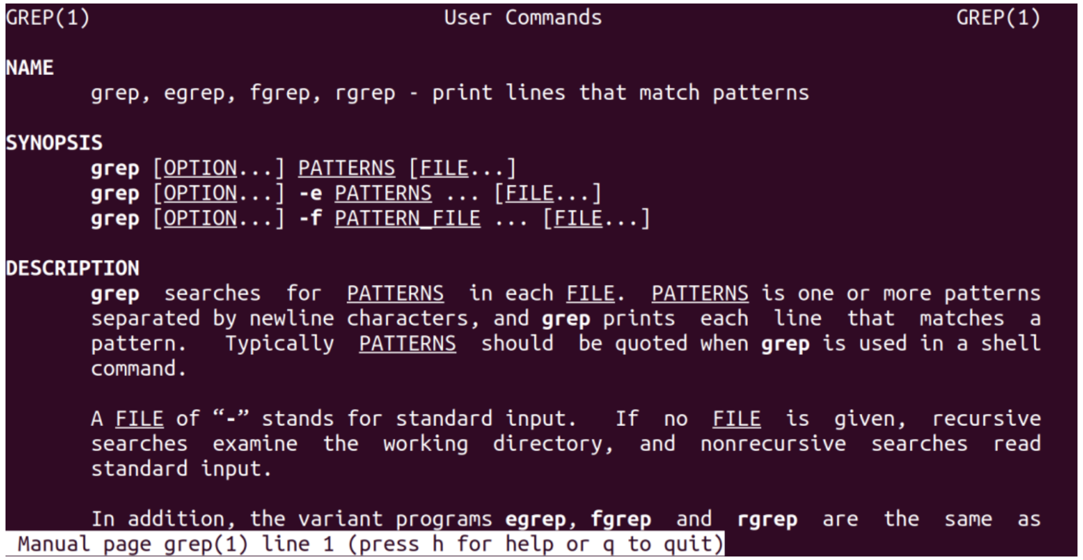
DEMO!

False is 1!

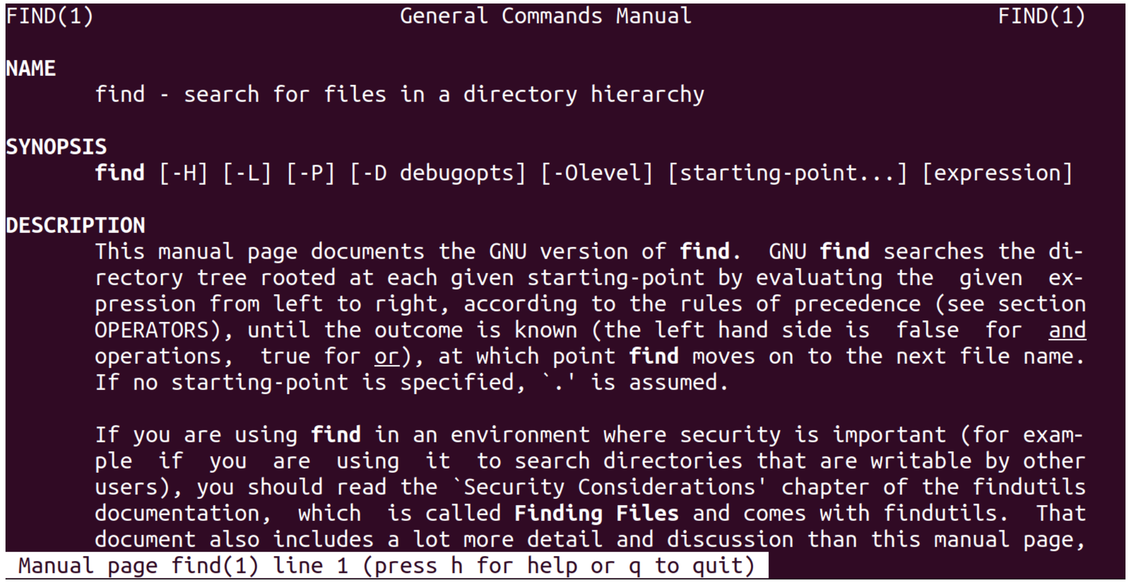


```
mrz@vbubu:/$ false; echo $?
1
mrz@vbubu:/$ true; echo $?
0
mrz@vbubu:/$
```

man

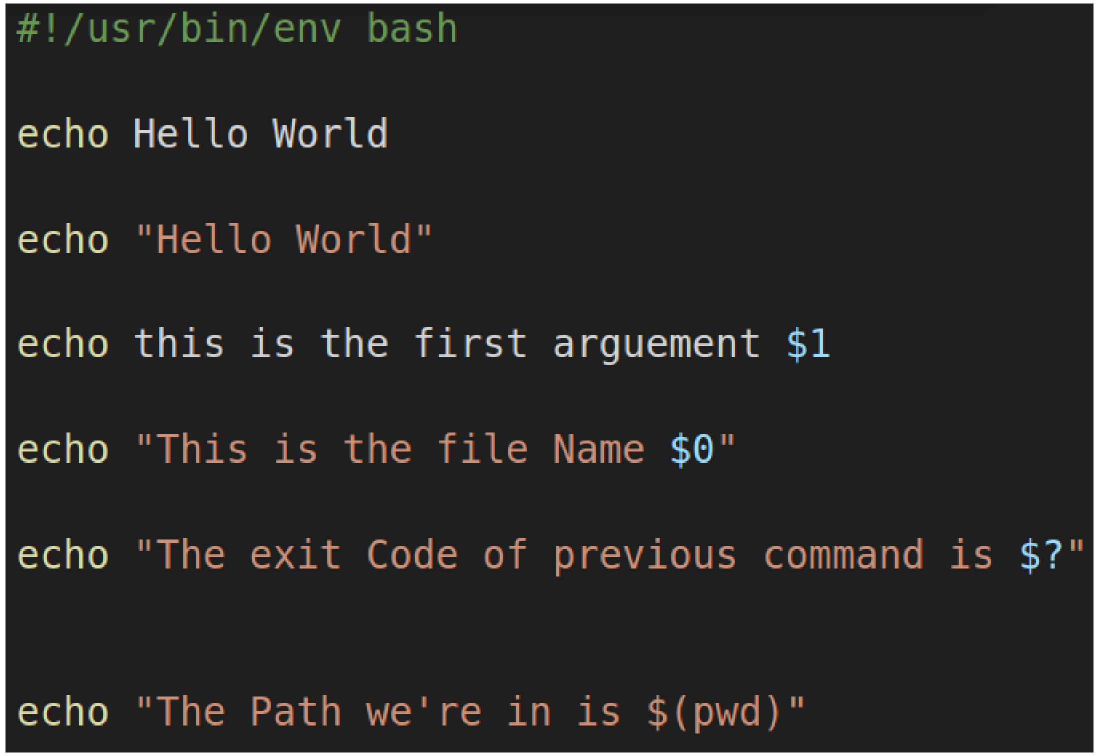


```
GREP(1)                          User Commands                         GREP(1)

NAME
       grep, egrep, fgrep, rgrep - print lines that match patterns

SYNOPSIS
       grep [OPTION...] PATTERNS [FILE...]
       grep [OPTION...] -e PATTERNS ... [FILE...]
       grep [OPTION...] -f PATTERN_FILE ... [FILE...]

DESCRIPTION
       grep  searches  for  PATTERNS  in each FILE.  PATTERNS is one or more patterns
       separated by newline characters, and grep prints  each  line  that  matches  a
       pattern.   Typically  PATTERNS  should  be quoted when grep is used in a shell
       command.

       A FILE of “-” stands for standard input.   If  no  FILE  is  given,  recursive
       searches  examine  the  working  directory,  and  nonrecursive  searches  read
       standard input.

       In addition, the variant programs egrep, fgrep  and  rgrep  are  the  same  as
 Manual page grep(1) line 1 (press h for help or q to quit)
```



```
FIND(1)                      General Commands Manual                     FIND(1)

NAME
       find - search for files in a directory hierarchy

SYNOPSIS
       find [-H] [-L] [-P] [-D debugopts] [-Olevel] [starting-point...] [expression]

DESCRIPTION
       This manual page documents the GNU version of find.  GNU find searches the di-
       rectory tree rooted at each given starting-point by evaluating the  given  ex-
       pression from left to right, according to the rules of precedence (see section
       OPERATORS), until the outcome is known (the left hand side is  false  for  and
       operations,  true for or), at which point find moves on to the next file name.
       If no starting-point is specified, `.' is assumed.

       If you are using find in an environment where security is important (for exam-
       ple  if  you  are  using  it  to search directories that are writable by other
       users), you should read the `Security Considerations' chapter of the findutils
       documentation,  which  is called Finding Files and comes with findutils.  That
       document also includes a lot more detail and discussion than this manual page,
 Manual page find(1) line 1 (press h for help or q to quit)
```

bash



```bash
#!/usr/bin/env bash

echo Hello World

echo "Hello World"

echo this is the first arguement $1

echo "This is the file Name $0"

echo "The exit Code of previous command is $?"


echo "The Path we're in is $(pwd)"
```

Assignemt 1

- Create a public repository in GitHub, push all your assignments in separate folders for each day – your submission is the URL to your repository.

Assignemt 2

- Write a bash script which *pulls* from your repository, the first assignment from yesterday, then it should build and run it.

- *Extra headache? Is there a way to skip authentication for GitHub?*