

Research Track I – Course Introduction

Carmine Tommaso Recchiuto

Research Track

Why?

- ✓ Hands-on experiences are required for engineering education.
- ✓ It may be seen how the first step of your Master Thesis work. Research Track I will put the basis for letting you working with robots, giving practical fundamentals in different contexts.
- ✓ Research Track I & Research Track 2.



Course Contents (RT1)



- ✓ Basic skills (I): Linux, git, documentation, Docker
- ✓ Basic skills (II): C++, Python

[Assignment I: 08/10] Deadline: 05/11

- ✓ Robot Operating System (I): nodes, topics, messages, services
- ✓ Robot Operating System (II): additional functionalities, custom messages and services, actions.

[Assignment I: 05/11] Deadline: 03/12

- ✓ Simulation environments: Gazebo
- ✓ Visualization tool: RViz
- ✓ ROS and navigation – mapping and motion planning: gmapping, MoveBase, laser sensors, obstacle avoidance

[Assignment III: 03/12] Deadline: before the exam!

Methodology

Teaching methods consist of:

- Frontal lessons with practical examples (Thursday 13:00 – 14:00 and Friday 10:00 – 12:00) in presence and online via the TEAMS platform.
- Guided exercises (Friday 10:00-12:00)



Assignments

- 1st assignment: an exercise involving a simple robot simulator in python
- 2nd assignment: a simple software architecture in ROS, involving the usage of the publish/subscribe and client/server architecture
- 3rd assignment: a complete simulation of a mobile robot in ROS, consisting in different modules.

The final system will navigate in a complex environment, possibly guided by the user or in a complete autonomous way.

Evaluation

Each assignment must be done individually and it should be:

- submitted as a separate Git repository with:
 - the ROS-based code
 - a report as a README file
- There will be an oral exam after that all the assignments have been submitted and evaluated.
- The exam concerns the discussion about the assignments and different aspects of the development of the systems (architecture design, robot behaviour, structure and clarity of the code, structure and clarity of the repository, the ability to motivate the proposed solutions and discuss drawbacks and benefits)



GIT – Distributed Version Control Systems

Carmine Tommaso Recchiuto

Version Control System?

A version control system, or VCS, tracks the history of changes as people and teams collaborate on projects together. As the project evolves, teams can run tests, fix bugs, and contribute new code with the confidence that any version can be recovered at any time. Developers can review project history to find out:

Which changes were made?
Who made the changes?
When were the changes made?
Why were changes needed?



Distributed Version Control System?

A distributed version control system (DVCS) allow full access to every file, branch, and iteration of a project, and allows every user access to a full and self-contained history of all changes.

Unlike once popular centralized version control systems, DVCSs don't need a constant connection to a central repository.

Developers can work anywhere and collaborate asynchronously from any time zone.



Distributed Version Control System!

Without version control, team members are subject to redundant tasks, slower timelines, and multiple copies of a single project.

To eliminate unnecessary work, DVCSs give each contributor a unified and consistent view of a project, surfacing work that's already in progress.

Seeing a transparent history of changes, who made them, and how they contribute to the development of a project helps team members stay aligned while working independently.



GIT

Git is an example of a distributed version control system (DVCS) commonly used for open source and commercial software development.

Git is the most-used VCS in the world!

Git is commonly used for both open source and commercial software development, with significant benefits for individuals, teams and businesses.



Some Terminology

Repository: encompasses the entire collection of files and folders associated with a project, along with each file's revision history.

Branching: divergence from the main line of development. It's a method to update the code without messing with the one of the main line.

Merge: the contents of a source branch are integrated with a target branch. In this process, only the target branch is changed.

Fork: copy of the repository. Forking a repository allows you to freely experiment with changes without affecting the original project.



Some History

1990 – Concurrent Versions System (CVS): revision control system with a client-server architecture.

The server stores the current version of the project and its history, and clients connect to the server in order to “check out” a copy of the project.

It supported distributed operations (the server only accepts changes made to the most recent version of a file). The CVS client may handle conflicts.



Some History

2000 – Subversion (SVN): open source version control system (it fixed the bugs and supplied some features missing in CVS).

New features:

- ***Branching***
- Possibility to rename or move files
- Versioning of symbolic links
- Versioning for directories, renames and file metadata



Some History

2000 – Bitkeeper: commercial software for version control.

The first, real, distributed system: repositories could be **forked** and **merged** easily (Subversion could do forks and merges only as major, time-consuming operations)

It has been used for the development of the source code of the Linux kernel.



Git

2006 – GIT: free and open source DCVS

Git lets developers see the entire timeline of their changes, decisions, and progression of any project in one place. From the moment they access the history of a project, the developer has all the context they need to understand it and start contributing.



Why Git?

With Subversion, you have a Problem: if the SVN Repository is in a location you can't reach (e.g., you don't have internet at the moment), you cannot commit. If you want to make a copy of your code, you have to literally copy/paste it.

With Git, you do not have this problem. Your local copy is a repository, and you can commit to it and get all benefits of source control. When you regain connectivity to the main repository, you can commit against it.

Git

2006 – GIT: free and open source DCVS

Git tracks **content rather than files**.

Branches are lightweight and merging is *easy*,

Git repositories are much **smaller in file size** than Subversion repositories.
There's only one ".git" directory, as opposed to dozens of ".svn" Repositories.

Git facilitates this through the use of topic branches: lightweight pointers to commits in history that can be easily created and deprecated when no longer needed.



Git - Handbook



Basic GIT Commands:

git init initializes a brand new Git repository and begins tracking an existing directory. It adds a hidden subfolder within the existing directory that houses the internal data structure required for version control

git clone creates a local copy of a project that already exists remotely. The clone includes all the project's files, history, and branches

git add stages a change. Git tracks changes to a developer's codebase, but it's necessary to stage and take a snapshot of the changes to include them in the project's history. This command performs staging, the first part of a two-step process. Any changes that are staged will become a part of the next snapshot and a part of the project's history. Staging and committing separately gives developers complete control over the history of their project without changing how they code

Git - Handbook



Basic GIT Commands:

git commit saves the snapshot to the project history and completes the change-tracking process. In short, a commit functions like taking a photo. Anything that's been staged with `git add` will become a part of the snapshot with `git commit`

git status shows the status of changes as untracked, modified, or staged.

git branch shows the branches being worked on locally.

git merge merges lines of development together. This command is typically used to combine changes made on two distinct branches. For example, a developer would merge when they want to combine changes from a feature branch into the main branch for deployment.

Git - Handbook



Basic GIT Commands:

git pull updates the local line of development with updates from its remote counterpart. Developers use this command if a teammate has made commits to a branch on a remote, and they would like to reflect those changes in their local environment.

git push updates the remote repository with any commits made locally to a branch

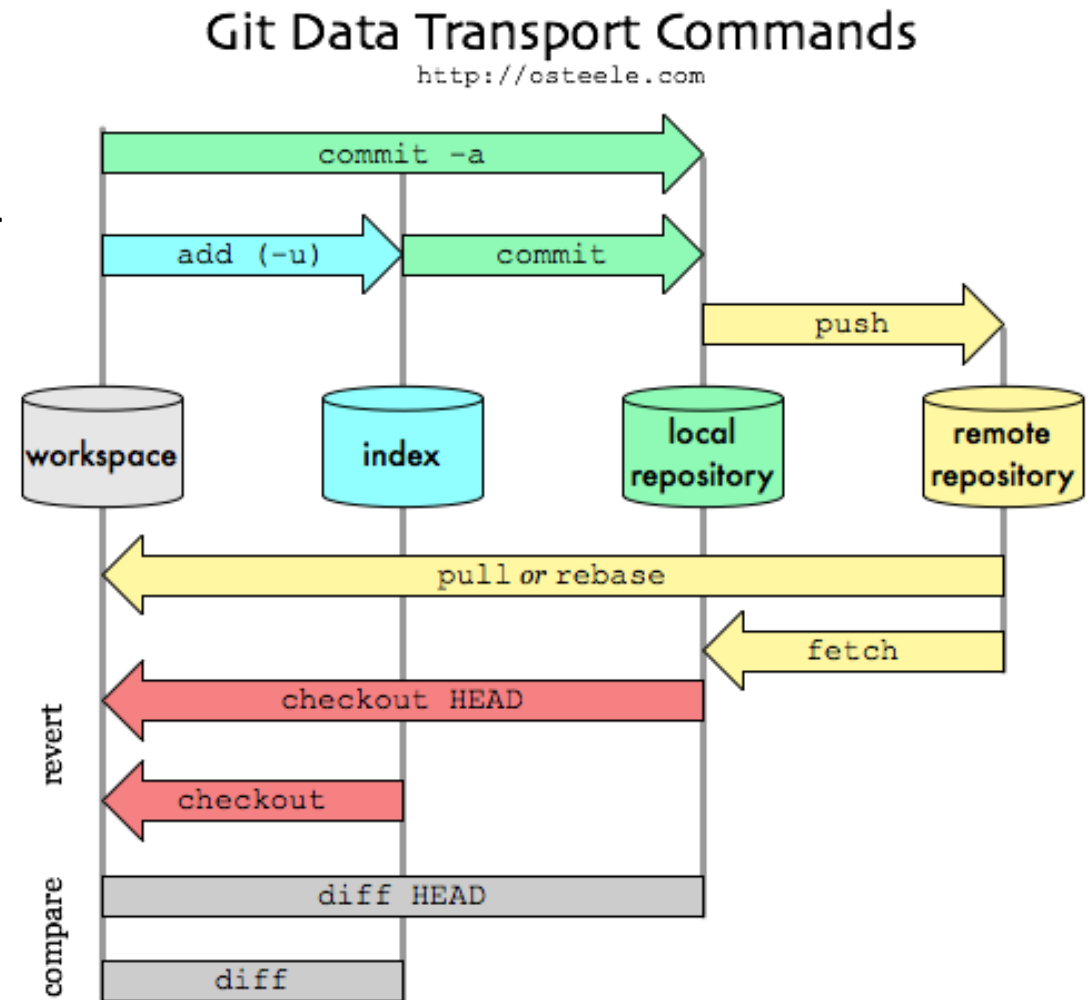
git fetch gathers any commits from the target branch that do not exist in your current branch and stores them in your local repository. However, it does not merge them with your current branch.

Git - Handbook

git checkout 'checks out' of an existing branch and view another branch of code. Checkout is essential for working on a new branch, existing branch, or remote branch

git diff runs a diff functions between different data sources

Please refer to <https://git-scm.com/docs> for a full handbook about all git commands



Example

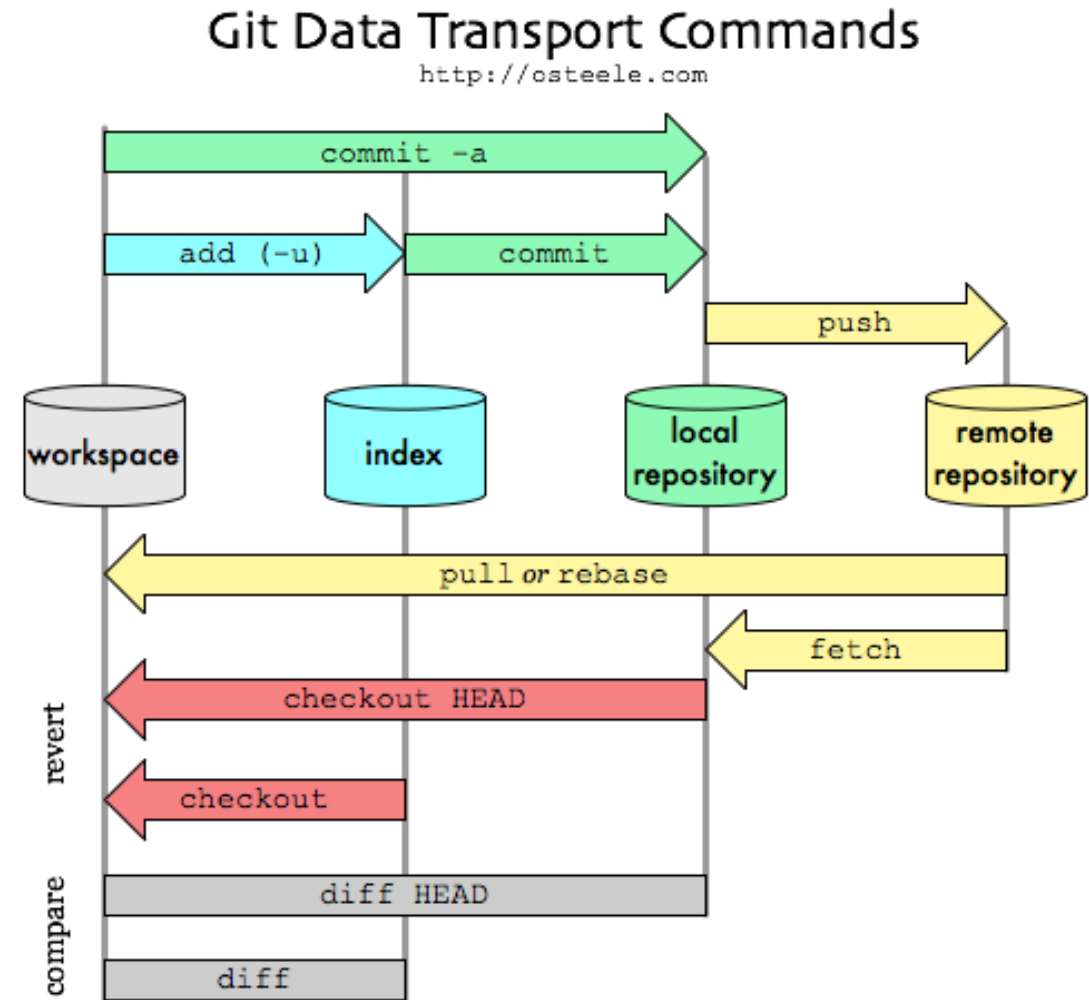
<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git> -> first install Git!

```
$ cd project  
$ git init
```

Git will reply
Initialized empty Git repository in .git/

Next, tell Git to take a snapshot of the contents of all files under the current directory (note the .), with git add:

```
$ git add .
```



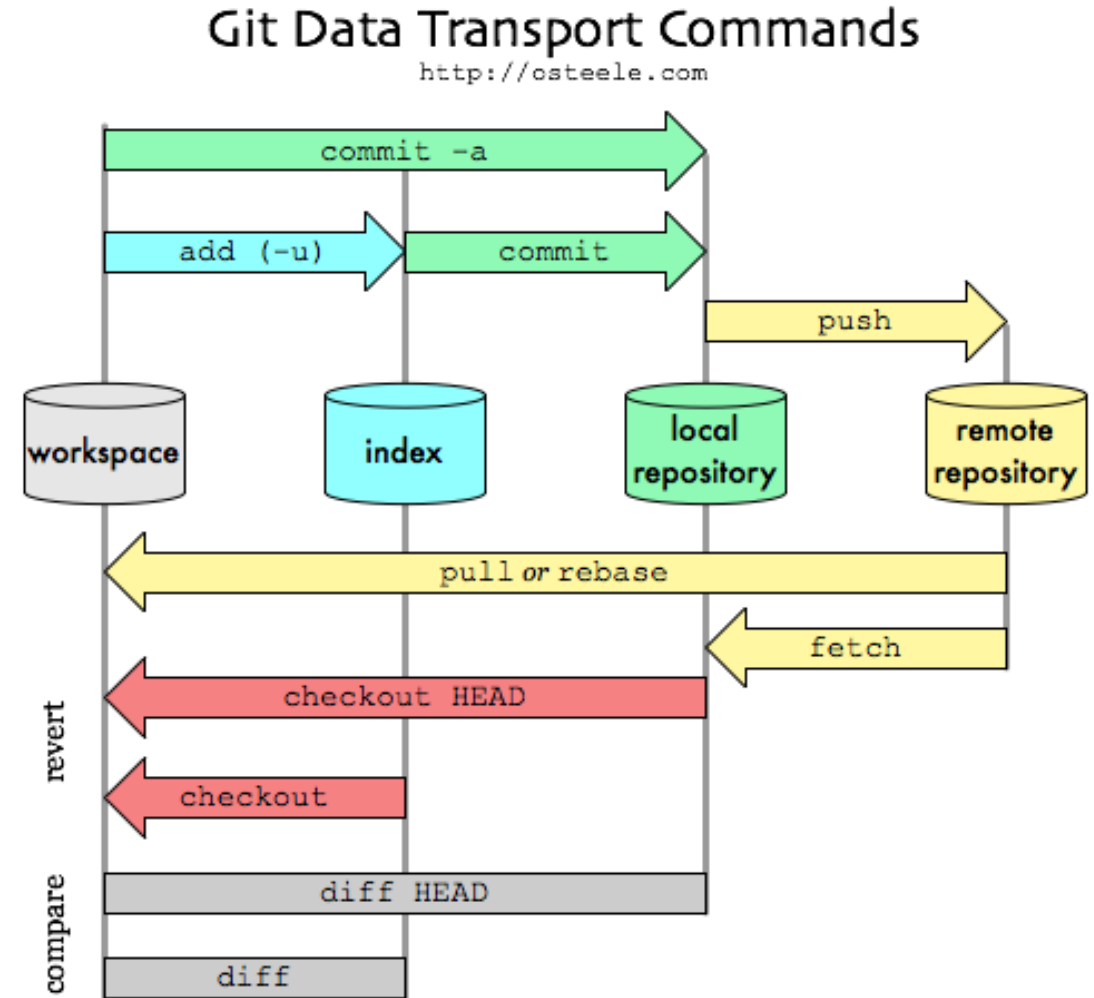
Example

This snapshot is now stored in a temporary staging area which Git calls the "index". You can permanently store the contents of the index in the repository with git commit:

\$ git commit

This will prompt you for a commit message. By default, the vi editor will be used to write the commit message.

You've now stored the first version of your project in Git!



Git Hosting Repositories

However, everything is still in your local repo! To work with a remote repository, we need a Git hosting repository.

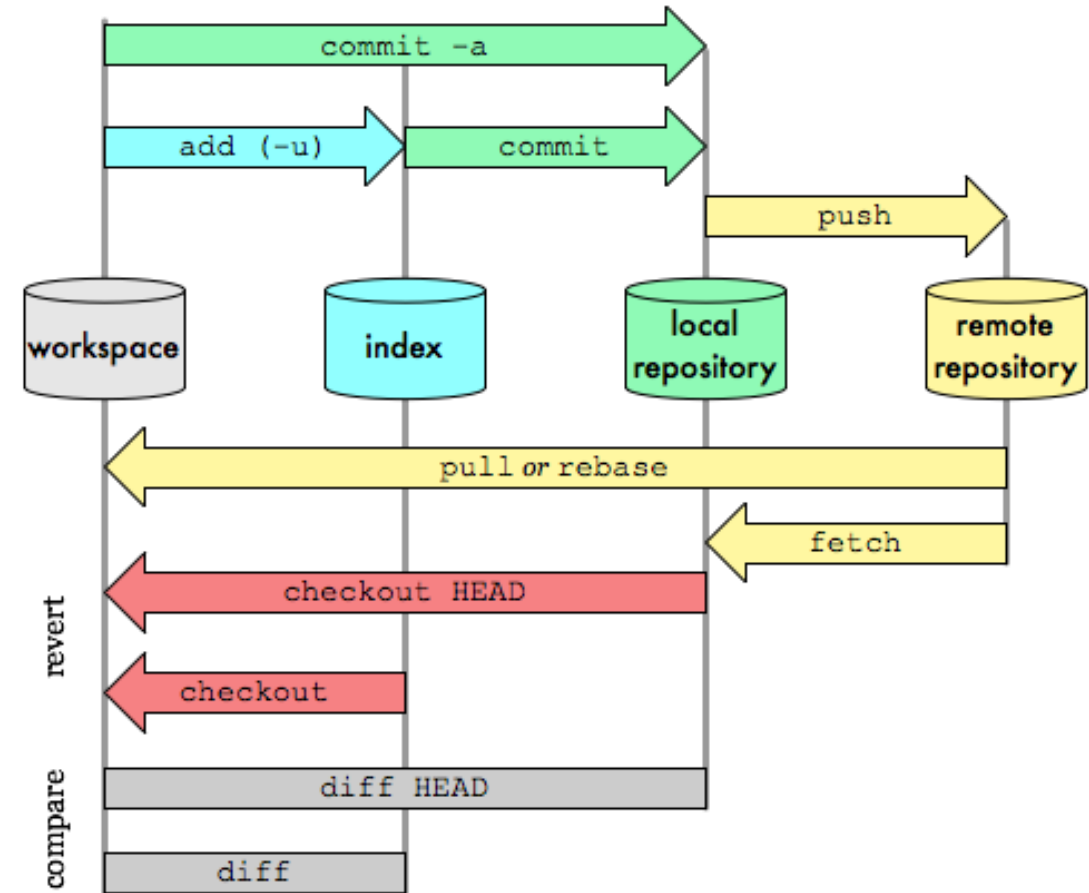
Git hosting repositories provide developers with tools to ship better code through command line features, issues (threaded discussions), pull requests, code review, or the use of a collection of free and for-purchase apps in the Marketplace.

Some examples are GitHub, GitLab, BitBucket, ...

Let's give a look to GitHub.

Git Data Transport Commands

<http://osteele.com>



GitHub

A repository is usually used to organize a single project. Repositories can contain folders and files, images, videos, spreadsheets, and data sets – anything your project needs. A README, or a file with information about your project, IS HIGHLY RECOMMENDED!

Create your account at <https://github.com/>

Add a new repository: <https://github.com/new>

Now we can add the contents of our local repository



Example

We can also add a README.md (md stands for MarkDown, a markup language)

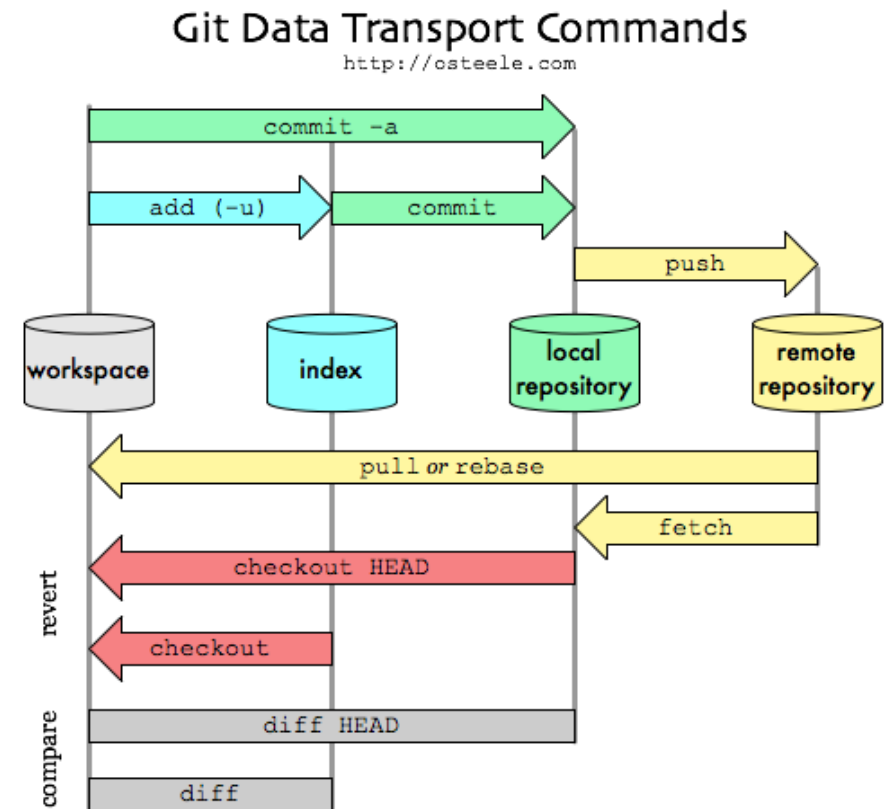
```
$ echo "# fantastic-guide" >> README.md
$ git add README.md
$ git commit -m "first commit"
```

Now let's set the current branch where we are working as the Master branch

```
$ git branch -M main
```

Finally, we need to associate a remote repository to our local repository. We use the command:

```
$ git remote add origin <remote_repo_url>
```



Example

Now we need to update the remote repository with the content of the local repository

\$ git push -u origin main

Important: to push something to your repo, you need at first to generate a token: <https://github.com/settings/tokens>

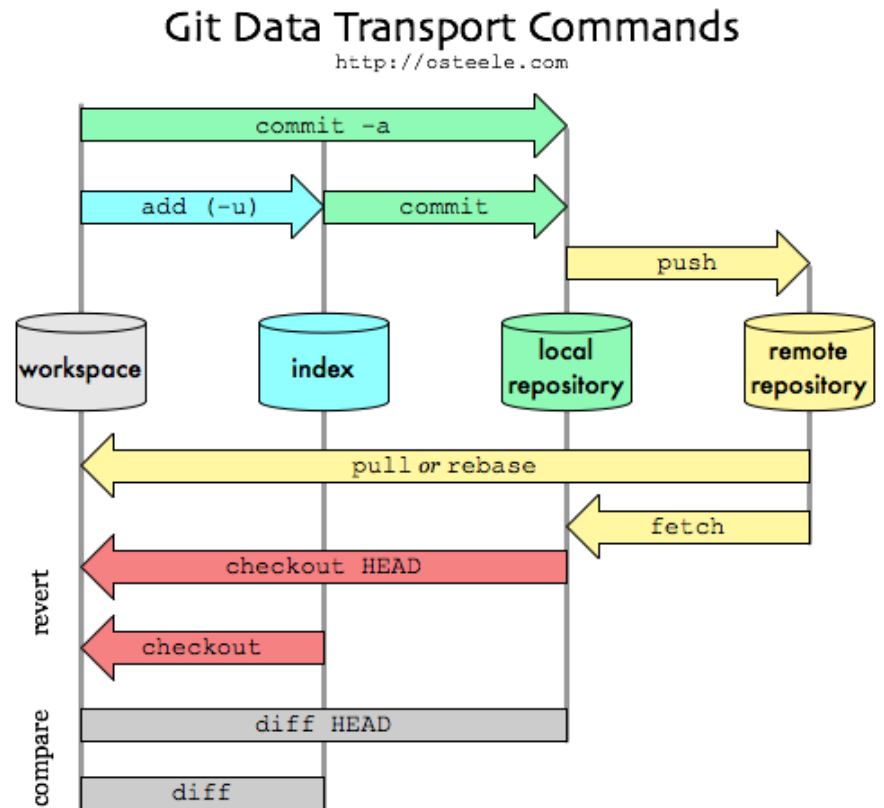
The remote repository is now updated. Every collaborators may consequently upload its local workspace by doing

\$ git clone <remote_repo_url>

Or just

\$ git pull

After the clone, a plain git fetch without arguments will update all the remote-tracking branches, and a git pull without arguments will in addition merge the remote master branch into the current master branch.



Example (2)

So, what if I want to contribute to an existing Git repository?

- Download the repository to your machine

```
$ git clone <remote_repo_url>
```

- Move into the local workspace

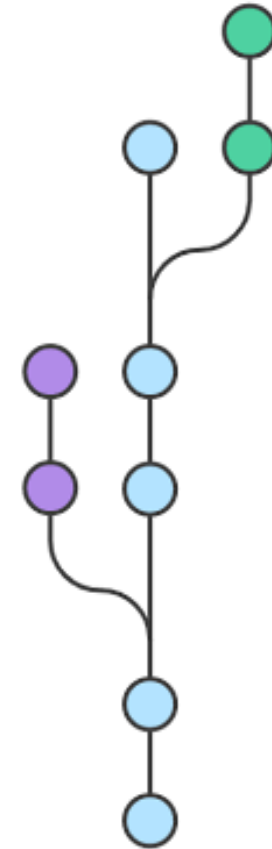
```
$ cd <local_dir_name>
```

- Create a new branch to store any new changes

```
$ git branch my-branch
```

- Switch to that branch (line of development)

\$ git checkout my-branch



Example (2)

Make your changes!

- Stage the changed files

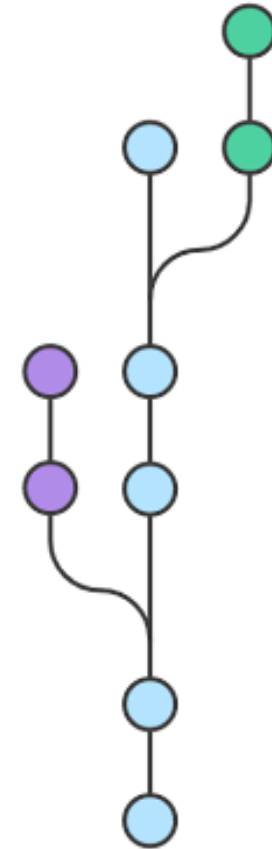
```
$ git add <file1_name> <file2_name>
```

- Take a snapshot of the staging area (and consequently update your local repository)

```
$ git commit -m "my_commit"
```

- Push changes to github

```
$ git push --set-upstream origin my-branch
```



Example (3)

What if I eventually want to merge my changes into the Master branch?

- Switch to the Master branch

\$ git checkout main

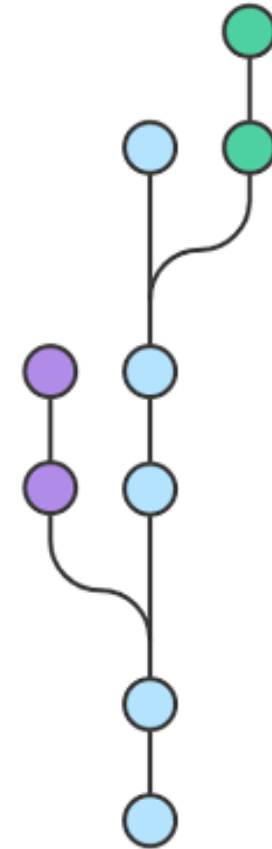
- Merge the branch my-branch on the Master branch

\$ git merge my-branch

- Push changes to github

\$ git push

Now all changes are also visible in the Master branch.



Git Workflows

Git Workflows: recipes or recommendation for how to use Git to accomplish work in a consistent and productive manner.

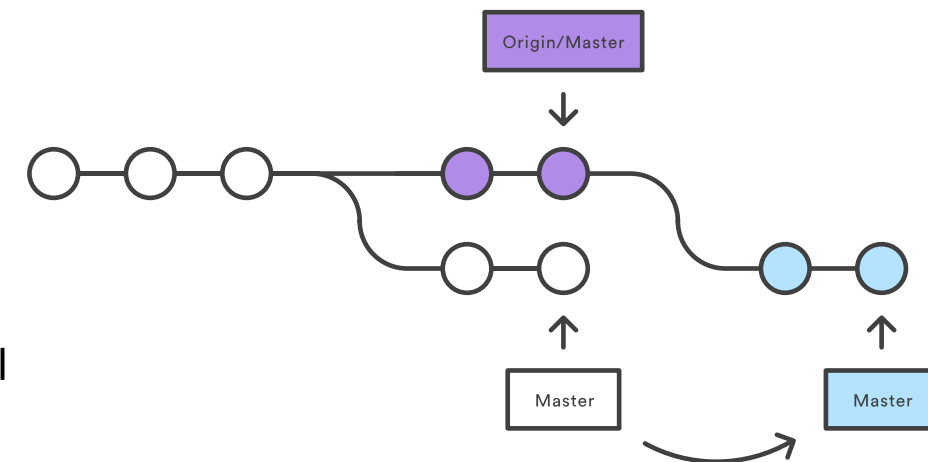
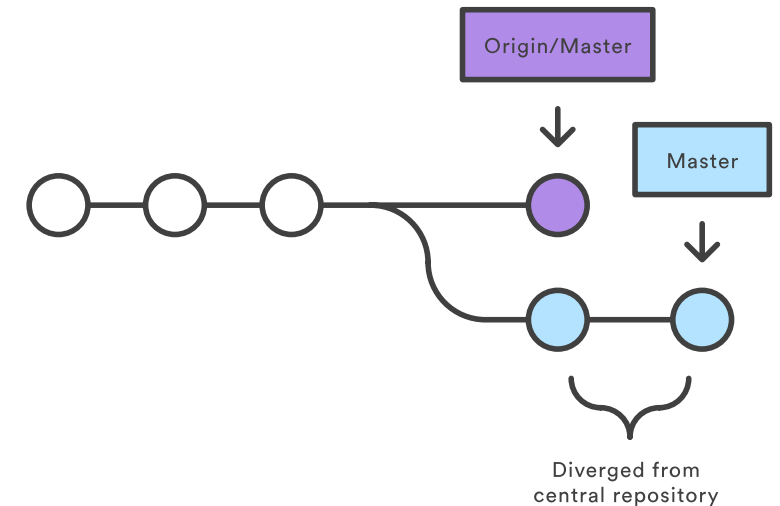
<https://www.atlassian.com/git/tutorials/comparing-workflows>

Examples:

Centralized Workflow

The Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project (only one branch: Master)

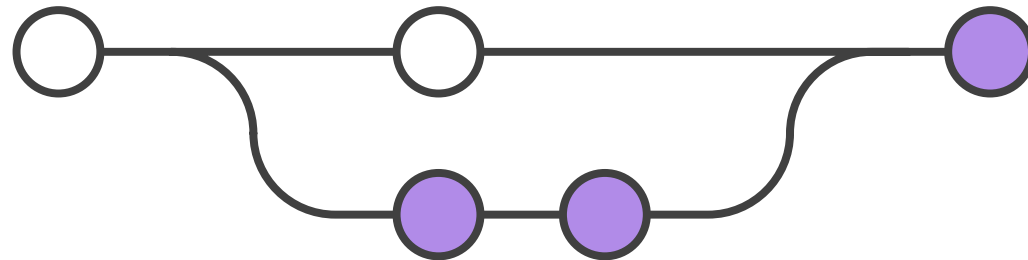
Developers start by cloning the central repository. In their own local copies of the project, they edit files and commit changes; however, these new commits are stored locally - they're completely isolated from the central repository. This lets developers defer synchronizing upstream until they're at a convenient break point.



Git Workflows

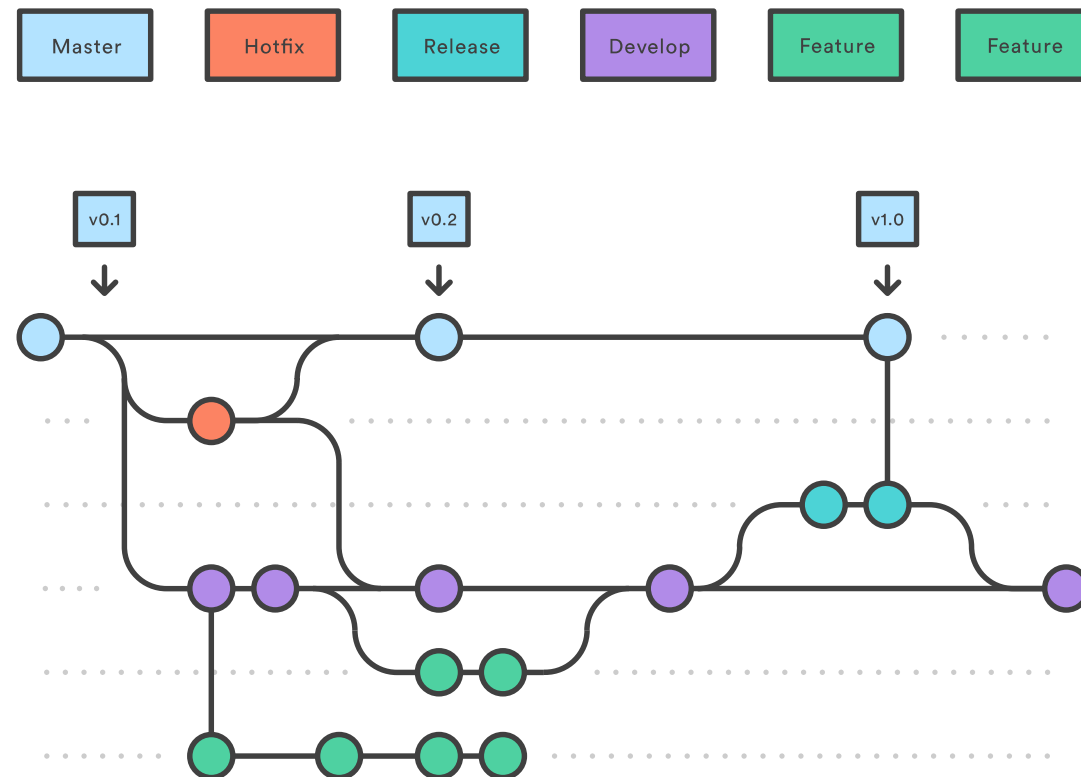
Feature Branch Workflow: all feature development should take place in a dedicated branch instead of the Master branch. This makes it easy for multiple developers to work on a particular feature without disturbing the main codebase.

A merge with the Master branch is done by means of a pull request (in the Git GUI). When a pull request is accepted, the feature is merged into the stable project (Master branch).



Git Workflows

Gitflow Workflow: based on the concept of «releases». It assigns very specific roles to different branches and defines how and when they should interact.



Git GUI Client

In particular when using a workflow involving branches, it may be useful to adopt a Git GUI Client, to have a clear idea of what it is going on.



A GIT GUI Client may be extremely useful to manage multiple branches and to solve conflicts.

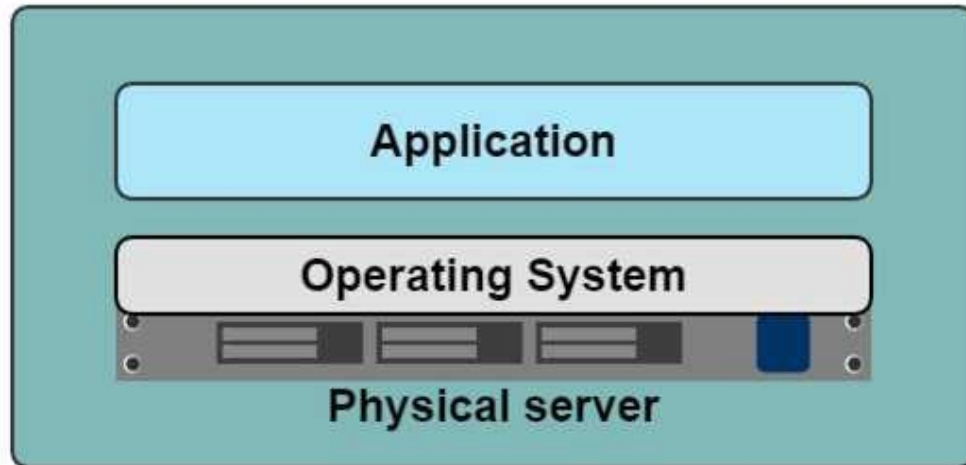
Docker – Virtual Machines and Containers

Carmine Tommaso Recchiuto

Limitations of application deployment

In the Dark Ages

One application on one physical server

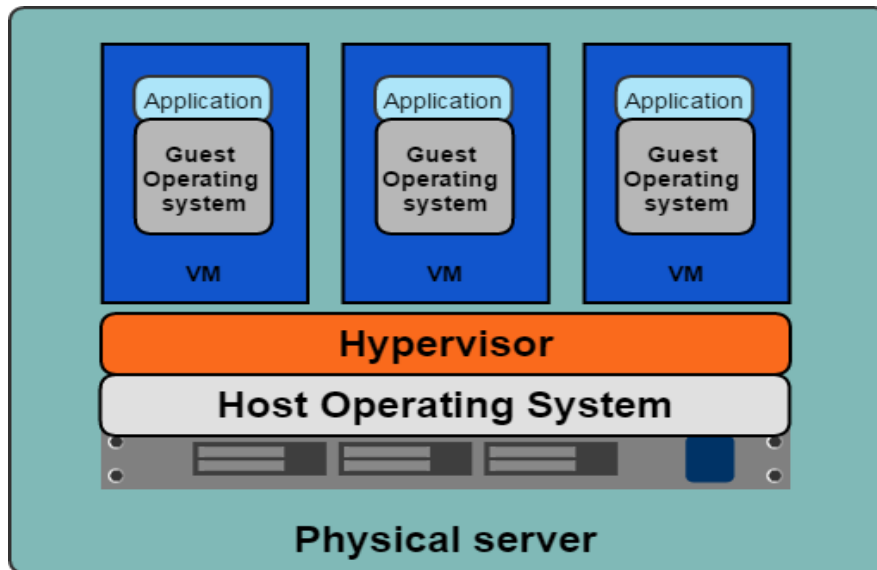


- Slow deployment times
- Huge costs
- Wasted resources
- Difficult to scale
- Difficult to migrate
- Vendor lock in

Hypervisor-based Virtualization

Hypervisor-based Virtualization

- One physical server can contain multiple applications
- Each application runs in a virtual machine (VM)



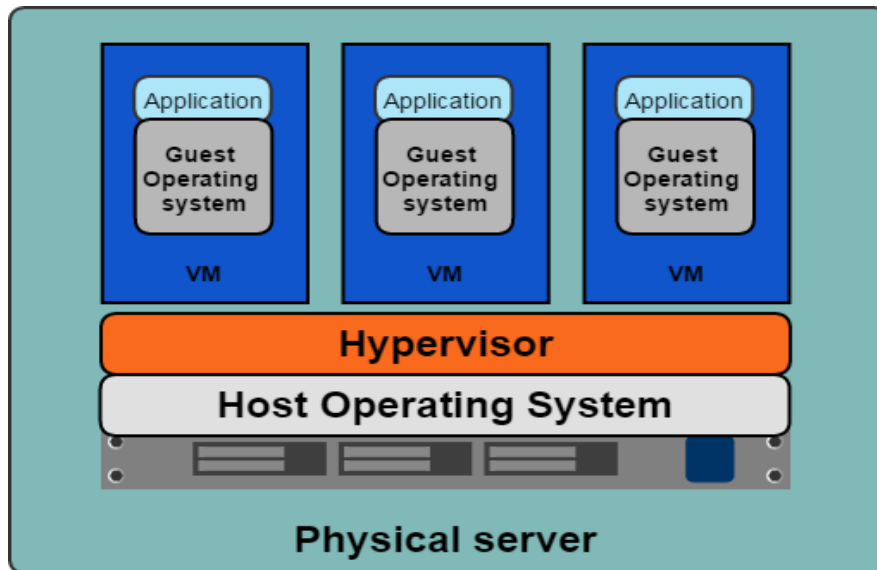
- Better resource pooling
 - One physical machine divided into multiple virtual machines
- Easier to scale
- VMs in the cloud
 - Rapid elasticity
 - Pay as you go model



Hypervisor-based Virtualization

Hypervisor-based Virtualization

- One physical server can contain multiple applications
- Each application runs in a virtual machine (VM)

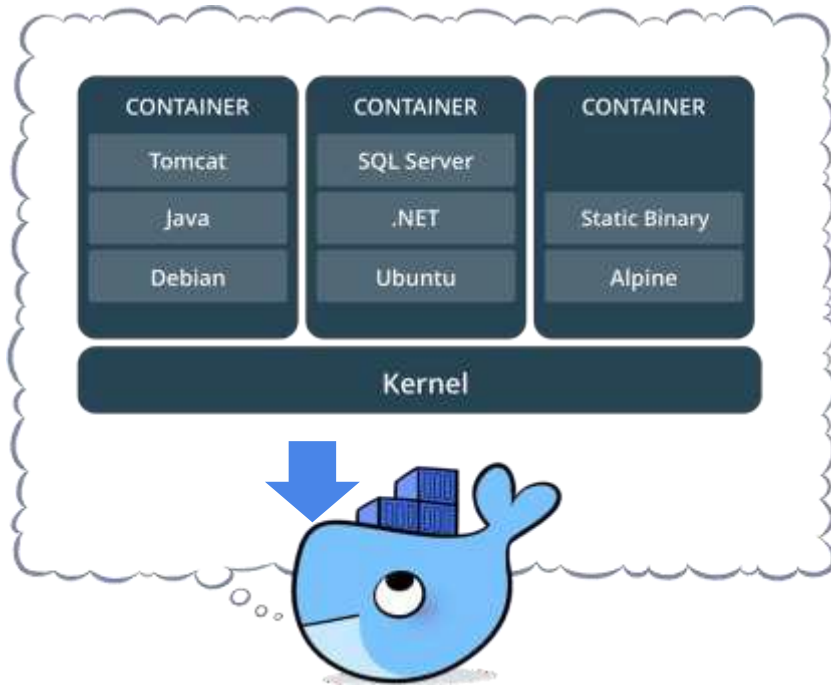


Limitations!

- Each VM stills requires
 - CPU allocation
 - Storage
 - RAM
 - An entire guest operating system
- The more VMs you run, the more resources you need
- Guest OS means wasted resources

Containers

- Standardized packaging for software and dependencies
- Isolate apps from each other
- Share the same OS kernel
- Works with all major Linux and Windows Server



Containers sit on top of a physical server and its host OS—for example, Linux or Windows.

Each container shares the host OS kernel and, usually, the binaries and libraries, too.

Shared components are read-only.

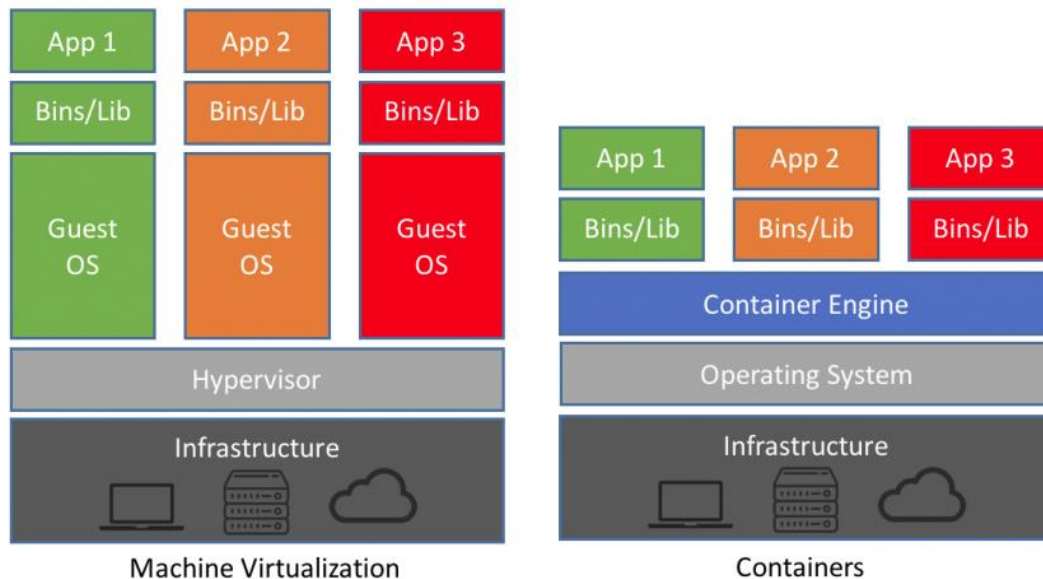
Containers are thus exceptionally “light”—they are only megabytes in size and take just seconds to start, versus gigabytes and minutes for a VM.

Containers

Each virtual machine runs a unique guest operating system, thus VMs with different operating systems can run on the same physical server

Each VM has its own binaries, libraries, and applications that it services, and the VM may be many gigabytes in size.

Containers reduce management overhead. Because they share a common operating system, only a single operating system needs care and feeding for bug fixes, patches, and so on.



Docker

- ✓ Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. In other words, it is a software platform for building containers: it allows applications to use the same kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer.
- ✓ Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.



Docker



Image

The basis of a Docker container. The content at rest.



Container

The image when it is 'running.' The standard unit for app service



Engine

The software that executes commands for containers. Networking and volumes are part of Engine. Can be clustered together.



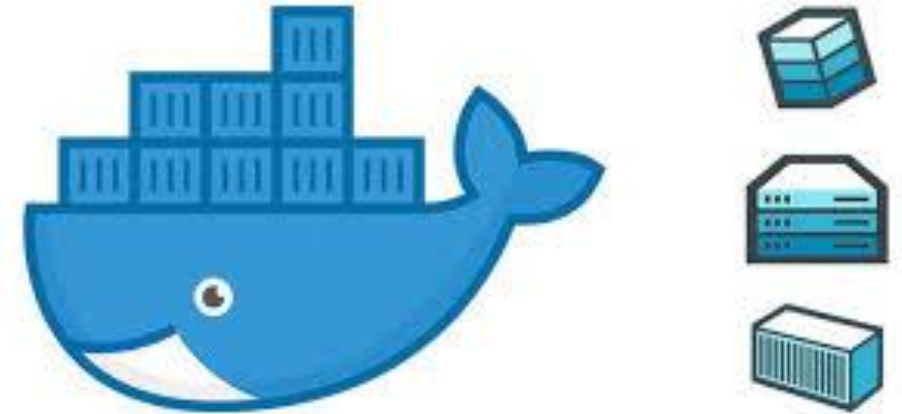
Registry

Stores, distributes and manages Docker images



Control Plane

Management plane for container and cluster orchestration



Its mission is basically: make it easy to package and ship code!

Dockerfile

- ✓ A *Dockerfile* is a text file that contains a bunch of instructions which informs Docker HOW the Docker image should get built.
- ✓ You can relate it to cooking. In cooking you have recipes. A recipe lets you know all of the steps you must take in order to produce whatever you're trying to cook.
 - The act of cooking is building the recipe.
 - A Dockerfile is a recipe for building Docker images



Dockerfile - example

```
$ git clone https://github.com/dockersamples/node-bulletin-board
```

```
$ cd node-bulletin-board/bulletin-board-app
```

```
$ docker build --tag bulletinboard:1.0 .
```

You'll see Docker step through each instruction in your Dockerfile, building up your image as it goes. If successful, the build process should end with a message ***Successfully tagged bulletinboard:1.0***

We can now run the following command to start a container based on this new image:

```
$ docker run --publish 8000:8080 --detach --name bb bulletinboard:1.0
```



Dockerfile - example

The dockerfile defined in this example takes the following steps:

Start FROM the pre-existing node:current-slim image. This is an official image, built by the node.js vendors and validated by Docker to be a high-quality image containing the Node.js Long Term Support (LTS) interpreter and basic dependencies.

Use WORKDIR to specify that all subsequent actions should be taken from the directory /usr/src/app in your image filesystem (never the host's filesystem).

COPY the file package.json from your host to the present location (.) in your image (so in this case, to /usr/src/app/package.json)

RUN the command npm install inside your image filesystem (which will read package.json to determine your app's node dependencies, and install them)

COPY in the rest of your app's source code from your host to your image filesystem.



Dockerfile - example

```
$ docker run --publish 8000:8080 --detach --name bb bulletinboard:1.0
```

--publish asks Docker to forward traffic incoming on the host's port 8000 to the container's port 8080. Containers have their own private set of ports, so if you want to reach one from the network, you have to forward traffic to it in this way. Otherwise, firewall rules will prevent all network traffic from reaching your container, as a default security posture.

--detach asks Docker to run this container in the background.

--name specifies a name with which you can refer to your container in subsequent commands, in this case bb.

Visit your application in a browser at localhost:8000. You should see your bulletin board application up and running.

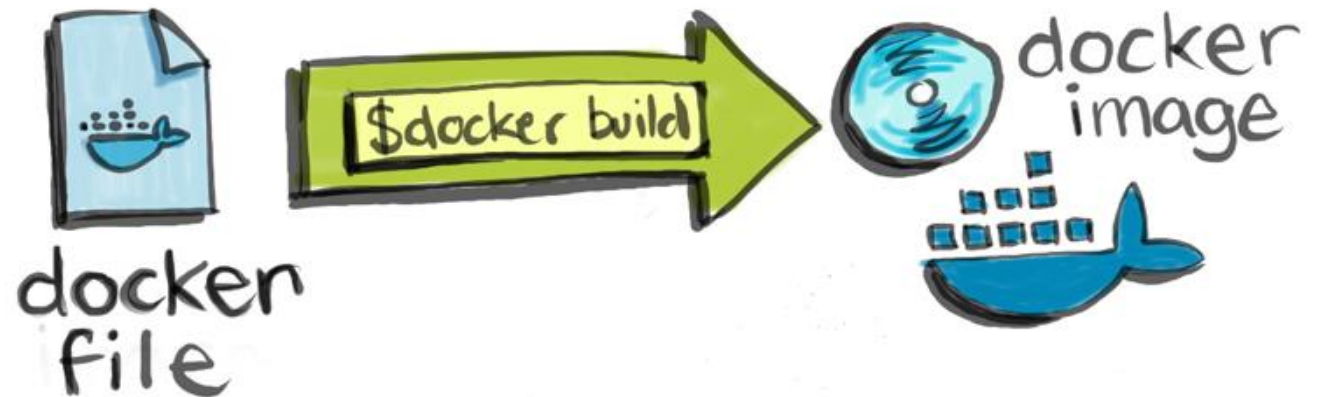


Images and Containers

- ✓ A Docker image gets built by running a Docker command (which uses a Dockerfile)
- ✓ A Docker image is a file, comprised of multiple layers, used to execute code in a Docker container.

\$ docker images lists all the images in your system

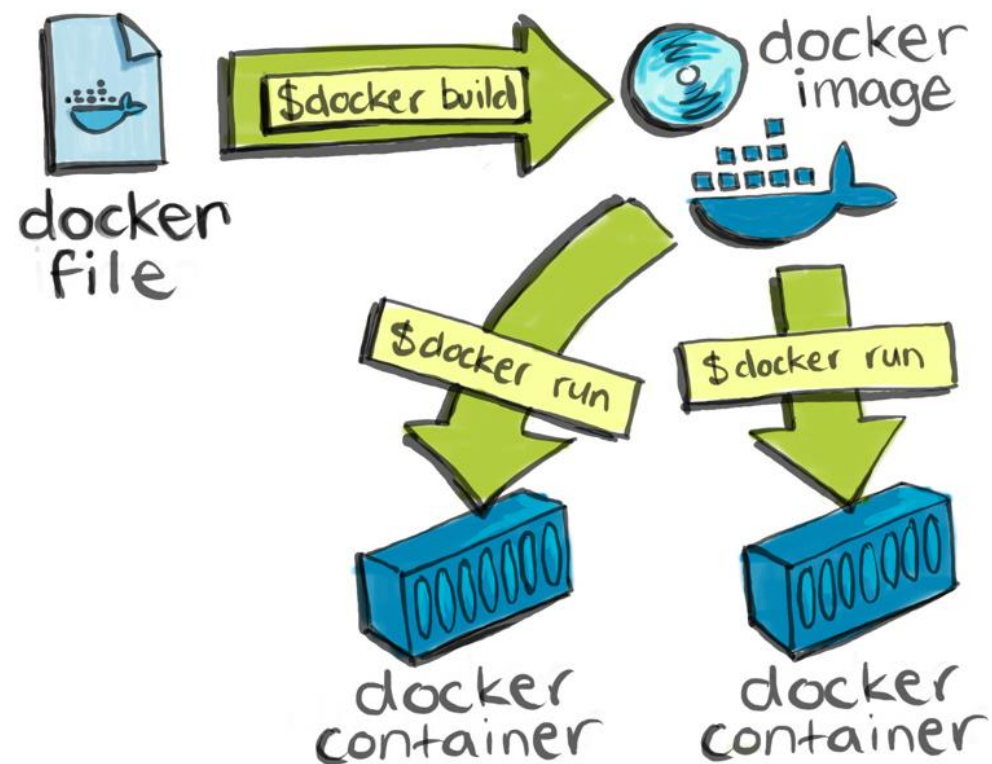
\$ docker rmi <IMAGE ID> may be used to remove a image



Images and Containers

- ✓ The act of running a Docker image creates a Docker container
- ✓ You can think of a Docker image as a class, where as a Docker container is an instance of that class.
- ✓ When a new container is created from an image, a writable layer is also created. This layer is called the container layer, and it hosts all changes made to the running container.

\$ docker run ... creates a container starting from one image



Images and Containers

How to save a modified containers?

Well, you don't need to do that. Once you have done, you may run

\$ docker stop <container_name>

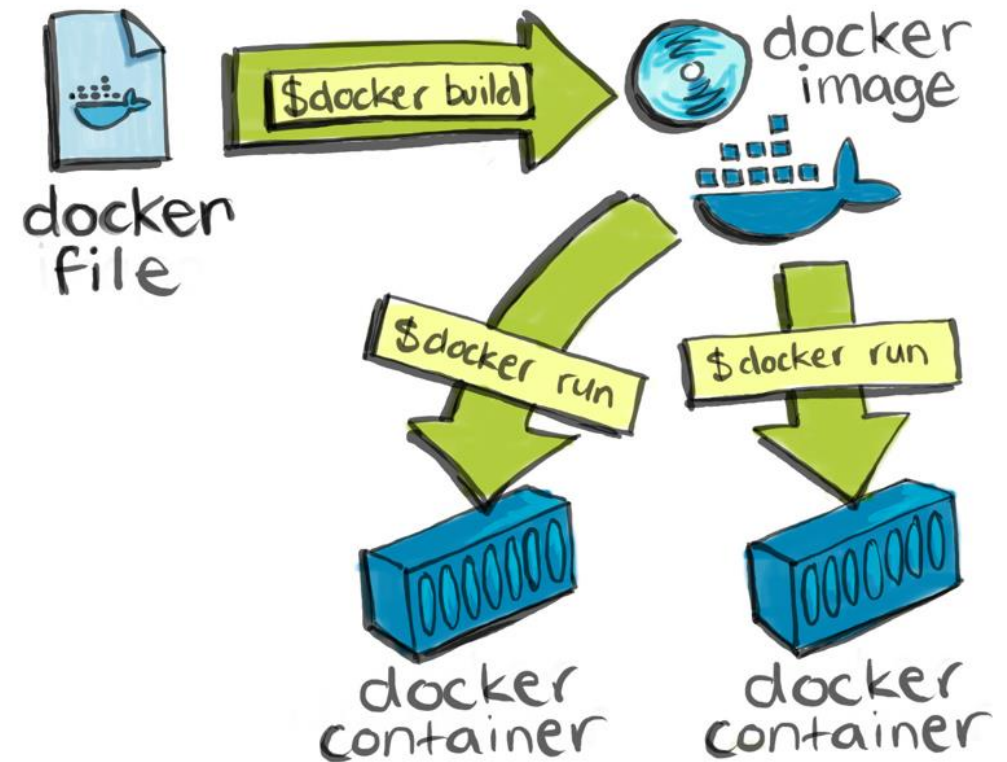
to stop the execution of your container.

When you need again your *living* container, run

\$ docker start <container_name>

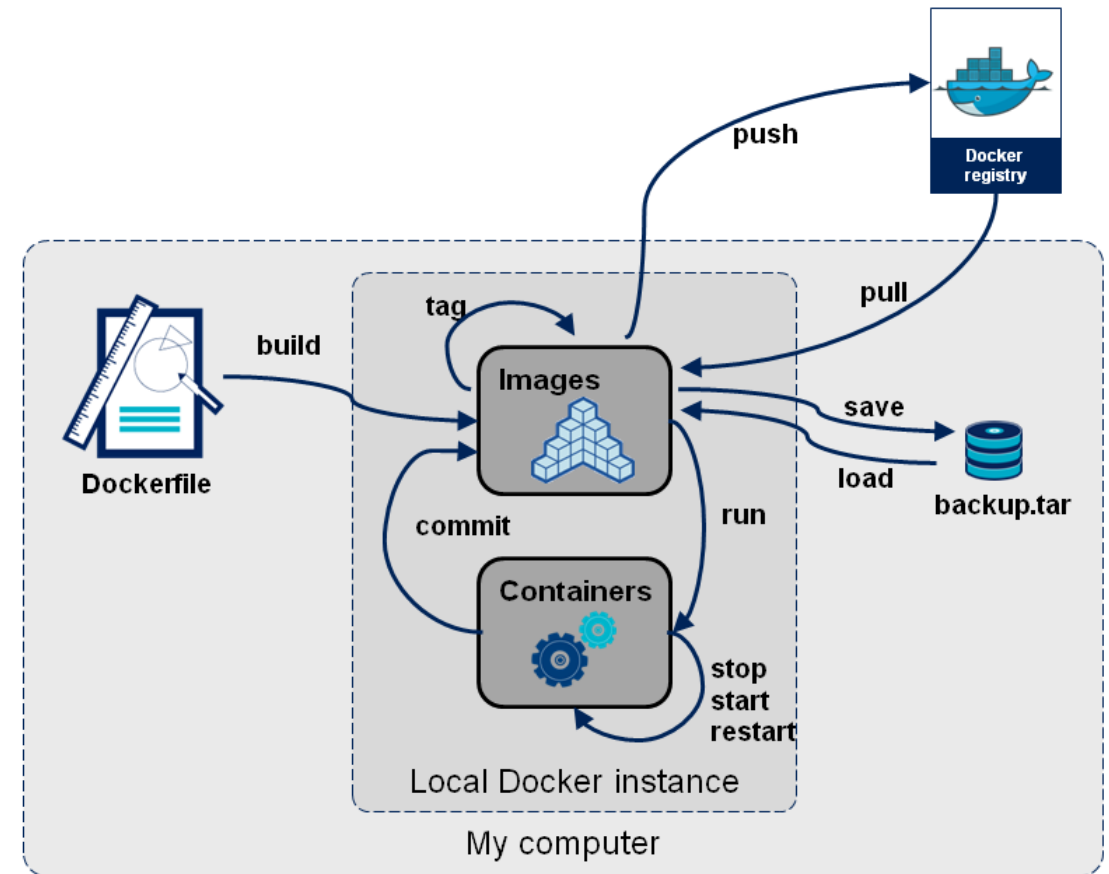
You can also list all containers (active or not)

\$ docker ps -a (docker ps lists only active containers)



Docker Image Repository

- ✓ Docker users store images in private or public repositories and from there can deploy containers, test images and share them. Docker offers Docker Hub, which is a cloud-based registry service that includes private and public image repositories..
- ✓ A user can upload their own custom image to the Docker Hub by using the docker push command. To ensure the quality of community images, Docker reviews the image and provides feedback for the image author before publishing.



References

- ✓ If you are interested in learning more:

<https://docs.docker.com/>

<https://docker-curriculum.com/>




Research Track - Docker Image


- ✓ Install Docker
 - Follow instructions from:
<https://docs.docker.com/install/>
- ✓ Get the image
 - Run `docker pull carms84/rpr`
- ✓ Run a container


Follow instructions from
https://hub.docker.com/repository/docker/carms84/noetic_ros2




https://hub.docker.com/repository/docker/carms84/noetic_ros2

 carms84 / noetic_ros2

This repository does not have a description 



 Last pushed: an hour ago

Tags and Scans



 VULNERABILITY SCANNING - DISABLED

[Enable](#)

This repository contains 1 tag(s).

TAG	OS	PULLED	PUSHED
 latest		an hour ago	an hour ago

[See all](#)

Readme  

Docker container used in the context of robotic programming courses @ Unige

The image has been developed starting from a docker image providing HTML5 VNC interface to access ROS Noetic on Ubuntu 20.04 with the LXDE desktop environment.

Run it by: `docker run -it --name my_ros -p 6080:80 -p 5900:5900 carms84/noetic_ros2`

You may then open a browser and type `http://127.0.0.1:6080/` or use a VNC client (port 5900) to connect to the VNC server.

Docker commands

[Public View](#)

To push a new tag to this repository,

`docker push carms84/noetic_ros2:tagname`

Automated Builds

Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

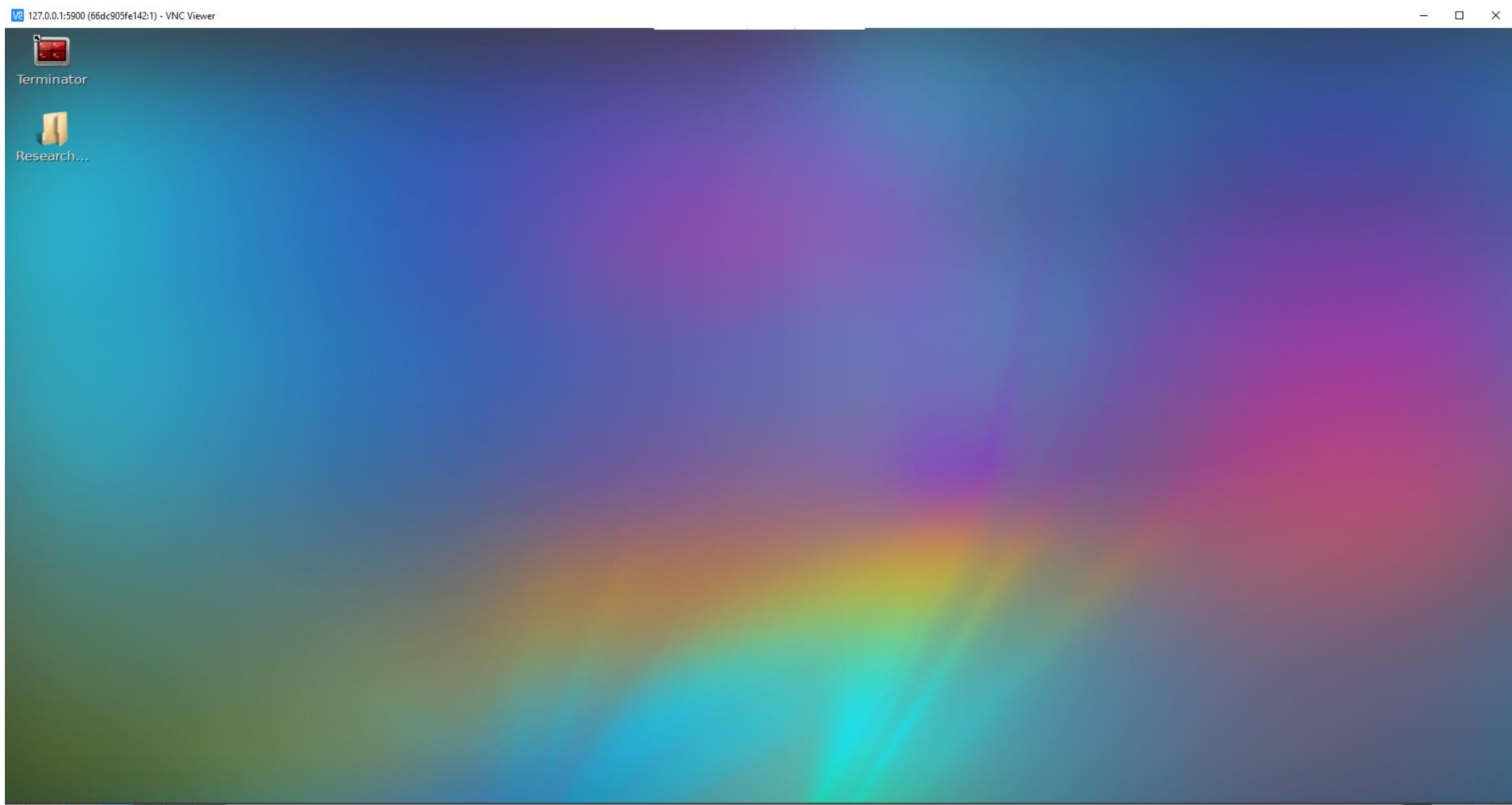
Available on Pro and Team plans.

[Upgrade to Pro](#)

[Learn more](#)

Research Track I – 23-24/09/2021

Carmine Tommaso Recchiuto



How to start

✓ If you have already:

- a Native Linux System, with ROS installed
- a Virtual Machine, with Linux-ROS
- a Docker image with Linux-ROS

you may use that.

✓ Otherwise, I suggest to:

- On Windows 10 Pro – Enterprise, Linux, MacOS, install Docker and pull the Docker image of the course
 - For other Windows versions, you may:
 - Use Docker Toolbox and the Docker image of the Course
- OR**
- Use a Virtual Machine with Linux and run Docker inside the Virtual Machine.

✓ In this last scenario, follow the instructions to install Docker Toolbox here:

https://docs.docker.com/toolbox/toolbox_install_windows/

✓ Or you may install VirtualBox (<https://www.virtualbox.org/>) and a Linux image for Virtual Box

https://www.virtualbox.org/wiki/Linux_Downloads

Known Issue

If executing: **docker run -it -p 6080:80 -p 5900:5900 --name my_ros carms84/noetic_ros2**, you get an error as :

```
INFO exited: xvfb (exit status 1; not expected)
INFO gave up: xvfb entered FATAL state, too many start retries too quickly
INFO exited: pcmanfm (exit status 1; not expected)
INFO gave up: pcmanfm entered FATAL state, too many start retries too quickly
INFO exited: lxpanel (exit status 1; not expected)
INFO gave up: lxpanel entered FATAL state, too many start retries too quickly
INFO exited: lxsession (exit status 1; not expected)
```

run the following commands from your terminal:

```
$ docker start my_ros
$ docker exec -it my_ros /bin/bash
$ rm /tmp/.X1-lock
$ /usr/bin/Xvfb :1 -screen 0 1024x768x16 &
$ export DISPLAY=":1"
$ export HOME ="/root"
$ export USER="/root"
$/usr/bin/openbox &
$/usr/bin/lxpanel --profile LXDE &
$/usr/bin/pcmanfm --desktop --profile LXDE &
$ x11vnc -display :1 -xkb -forever -shared -repeat &
```


Images and Containers

Once done all this commands, your container will be visible using a vnc viewer or a browser.

Please notice that this is required only the first time you launch the container. When finished working with it, you can just close it

\$ docker stop my_ros

And start it when you need it again

\$ docker start my_ros

Software Documentation

Carmine Tommaso Recchiuto

Software Documentation

Writing a reliable documentation is a must for any programmer.

The presence of documentation helps keep track of all aspects of an application and it improves on the quality of a software product.

A good software documentation is fundamental for:

- development
- maintenance
- knowledge transfer to other developers

A successful documentation will make information easily accessible, help new users learn quickly, simplify the product and help cut support costs (if any!).



Software Documentation

- ✓ For all these reasons, you may guess that Software documentation is a critical activity in engineering, which may drastically improve the quality of a software product.
- ✓ Systematic approaches to documentation increase the level of confidence of the end deliverable as well as enhance and ensure product's success through its usability, marketability and ease of support
- ✓ Documentation is an activity that needs to commence early in development and continue throughout the development lifecycle. It acts as a tool for planning and decision making.



Seven Golden Rules

1. Documentation should be written from the point of view of the reader, not the writer.
2. Avoid repetition.
3. Avoid unintentional ambiguity.
4. Use a standard organization.
5. Record rationale
6. Keep it current
7. Review documentation for fitness of purpose



F. Bachmann, L. Bass, J. Carriere, P. Clements, D. Garlan, J. Ivers, R. Nord, R. Little, Software Architecture Documentation in Practice: Documenting Architectural Layers. CMU/SEI-2000- SR-004. Carnegie. Mellon University. 2000.

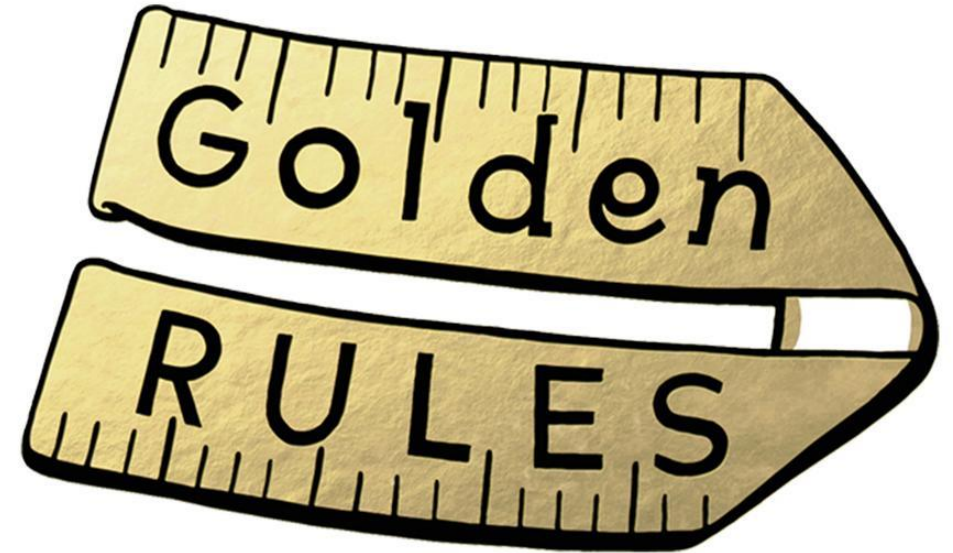
Other four golden rules

Keep comments as close to the code being described as possible. Comments that aren't near their describing code are frustrating to the reader and easily missed when updates are made.

Don't use complex formatting (such as tables or ASCII figures). Complex formatting leads to distracting content and can be difficult to maintain over time.

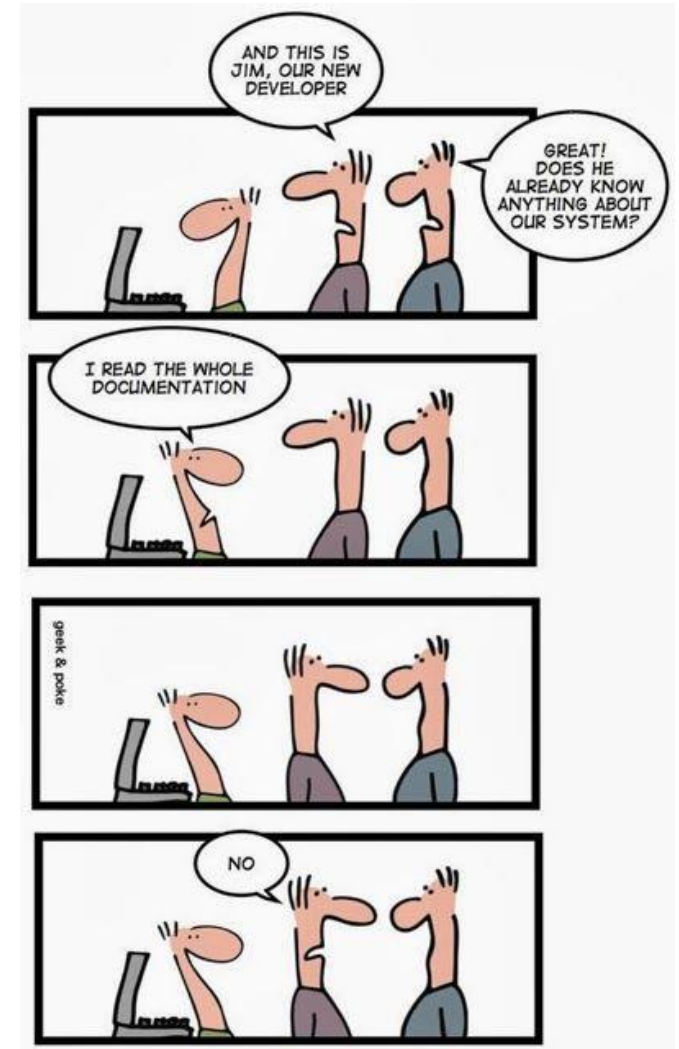
Don't include redundant information. Assume the reader of the code has a basic understanding of programming principles and language syntax.

Design your code to comment itself. The easiest way to understand code is by reading it. When you design your code using clear, easy-to-understand concepts, the reader will be able to quickly conceptualize your intent.



Documentation?

- ✓ API (functions, methods) documentation – Reference documentation regarding making calls and classes
- ✓ README – A high-level overview of the software, usually alongside the source code
- ✓ Release notes - Information describing the latest software or feature releases, and bug fixes
- ✓ System documentation – Documents describing the software system, including technical design documents, software requirements, and UML diagrams



Examples –Python

Comments should have a maximum length of 72 characters. This is true even if your project changes the max line length to be greater than the recommended 80 characters. If a comment is going to be greater than the comment char limit, using multiple lines for the comment is appropriate.

Planning and Reviewing: When you are developing new portions of your code, it may be appropriate to first use comments as a way of planning or outlining that section of code. Remember to remove these comments once the actual coding has been implemented and reviewed/tested:

First step

Second step

Third step

Examples –Python

Code Description: Comments can be used to explain the intent of specific sections of code:

***# Attempt a connection based on previous settings. If unsuccessful,
prompt user for new settings.***

Algorithmic Description: When algorithms are used, especially complicated ones, it can be useful to explain how the algorithm works or how it's implemented within your code. It may also be appropriate to describe why a specific algorithm was selected over another.

Using quick sort for performance gains

Tagging: The use of tagging can be used to label specific sections of code where known issues or areas of improvement are located. Some examples are: BUG, FIXME, and TODO.

TODO: Add condition for when val is None

Examples – Docstrings in Python

Documenting your Python code is all centered on docstrings. These are built-in strings that, when configured correctly, can help your users and yourself with your project's documentation. They are stored in the property `__doc__`:

```
def say_hello(name):  
    print(f"Hello {name}, is it me you're looking for?")  
  
say_hello.__doc__ = "A simple function that says hello! "
```

Now try opening a python shell and typing

```
$ from test_docstrings import say_hello  
$ help(say_hello)
```

Examples – Docstrings in Python

Python has one more feature that simplifies docstring creation. Instead of directly manipulating the `__doc__` property, the strategic placement of the string literal directly below the object will automatically set the `__doc__` value.

```
def say_hello(name):  
    """A simple function that says hello... """  
    print(f"Hello {name}, is it me you're looking for?")
```

Examples – Docstrings in Python

Docstring conventions are described at <https://www.python.org/dev/peps/pep-0257/> . Their purpose is to provide your users with a brief overview of the object. They should be kept concise enough to be easy to maintain but still be elaborate enough for new users to understand their purpose and how to use the documented object.

In all cases, the docstrings should use the triple-double quote (""") string format. This should be done whether the docstring is multi-lined or not. At a bare minimum, a docstring should be a quick summary of whatever is it you're describing and should be contained within a single line

"""This is a quick summary line used as a description of the object."""

Multi-lined docstrings are used to further elaborate on the object beyond the summary. All multi-lined docstrings have the following parts:

- A one-line summary line
- A blank line preceding the summary
- Any further elaboration for the docstring
- Another blank line

Examples – Docstrings in Python

```
"""This is the summary line
```

This is the further elaboration of the docstring. Within this section, you can elaborate further on details as appropriate for the situation. Notice that the summary and the elaboration is separated by a blank new line.

```
"""
```

Notice the blank line above. Code should continue on this line.

Docstrings can be further broken up into three major categories:

- **Class Docstrings:** Class and class methods
- **Script Docstrings:** Scripts and functions



Python Documentation

The way you document your project should suit your specific situation. Keep in mind who the users of your project are going to be and adapt to their needs. Depending on the project type, certain aspects of documentation are recommended. The general layout of the project and its documentation should be as follows:

project_root/

- project/ # Project source code
- docs/
- README
- HOW_TO_CONTRIBUTE
- CODE_OF_CONDUCT
- examples.py



Python Documentation

Readme: A brief summary of the project and its purpose. Include any special requirements for installing or operating the projects. Additionally, add any major changes since the previous version. Finally, add links to further documentation, bug reporting, and any other important information for the project. Dan Bader has put together a great tutorial on what all should be included in your readme: <https://dbader.org/blog/write-a-great-readme-for-your-github-project>

How to Contribute: This should include how new contributors to the project can help. This includes developing new features, fixing known issues, adding documentation, adding new tests, or reporting issues.

Code of Conduct: Defines how other contributors should treat each other when developing or using your software. This also states what will happen if this code is broken. If you're using Github, a Code of Conduct template can be generated with recommended wording. For Open Source projects especially, consider adding this.



Documentation Tools

Documenting your code, especially large projects, can be daunting. Thankfully there are some tools out and references to get you started:

Sphinx: <https://www.sphinx-doc.org/en/master/> A collections of tools to auto-generate documentation in multiple formats

Doxygen: <https://www.doxygen.nl/manual/docblocks.html> A tool for generating documentation that supports Python as well as multiple other languages



BE AWARE!!!



SOMEBODY MAY ACTUALLY READ IT!

Sphinx

Sphinx was originally created for Python, but it has now facilities for the documentation of software projects in a range of languages.

Sphinx uses [reStructuredText](#) as its markup language

- **Output formats:** HTML (including Windows HTML Help), LaTeX (for printable PDF versions), ePub, Texinfo, manual pages, plain text
- **Extensive cross-references:** semantic markup and automatic links for functions, classes, citations, glossary terms and similar pieces of information
- **Hierarchical structure:** easy definition of a document tree, with automatic links to siblings, parents and children
- **Automatic indices:** general index as well as a language-specific module indices
- **Code handling:** automatic highlighting using the [Pygments](#) highlighter
- **Extensions:** automatic testing of code snippets, inclusion of docstrings from Python modules (API docs)
- **Contributed extensions:** more than 50 extensions [contributed by users](#) in a second repository

Doxygen

Doxygen is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL

- It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
- You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. Doxygen can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically
- You can also use doxygen for creating normal documentation

Markdown Format

Markdown is intended to be as easy-to-read and easy-to-write as is feasible.

Readability, however, is emphasized above all else. A Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. While Markdown's syntax has been influenced by several existing text-to-HTML filters, the single biggest source of inspiration for Markdown's syntax is the format of plain text email.

You can use Markdown most places around GitHub:

Comments in Issues and Pull Requests

Files with the .md or .markdown extension (README!)

Markdown Format

Github Flavored Markdown: dialect of Markdown that is currently supported for user content on GitHub.com and GitHub Enterprise.

<https://github.github.com/gfm/#what-is-github-flavored-markdown->

Here's an overview of Markdown syntax that you can use anywhere on GitHub.com or in your own text files.

<https://docs.github.com/en/github/writing-on-github/basic-writing-and-formatting-syntax>

Linux (for Robotics)

Carmine Tommaso Recchiuto

Linux



Why Linux for Robotics?

- The focus will be always on robotics developers!

Yes, but why Linux?

- Have you ever heard of ROS? (Yes, actually few slides ago). The Robotics Operating System, at present, **ROS only fully supports Linux systems.**

As a matter of fact, Linux is much more widespread than you may think: Google, Facebook, or any major internet site uses Linux servers. If you are using an Android phone, you are using Linux. 498 out of 5000 world's speediest supercomputers use Linux. And of course, almost all robots uses Linux.

Linux

Versatility

Linux is available under the GNU GPL license, which means it can be freely used on almost any product or service you're developing, as long as the license terms are respected. Also, Linux development is community-based. This means that you can work with other Linux developers to share knowledge and learning.

Security

Linux is one of the most secure operating systems around, from devices/files to programs, access mechanisms, and secure messaging.

Real-time

RTLinux is a hard realtime operating system microkernel. Many RTOS (RTAI,...) are based on the Linux kernel



Linux

While there are hundreds of Linux commands, there are really only a handful that you will use repeatedly. Do you know the Pareto Principle (also known as the 80/20 rule)? For many event, roughly 80% of the effects come from 20% of the causes.

We will focus on learning the fundamentals, the handful of commands and tools that you will use most frequently.

To follow the examples that will be proposed during the course, you should have a linux distribution:

- Native linux OS
- Linux subsystem for windows (WSL) (not recommended)
- Virtual Machine
- Docker!



Linux

Distributions?



A **Linux distribution** (often abbreviated as **distro**) is an operating system made from a software collection that is based upon the Linux kernel

A typical Linux distribution comprises a Linux kernel, GNU tools and libraries, additional software, documentation, a window system (the most common being the X Window System), a window manager, and a desktop environment.

Which one? One of the most popular desktop Linux distribution, and also well compatible with ROS, is **Ubuntu**

In the following, we will always refer to the Ubuntu distribution for the commands description.

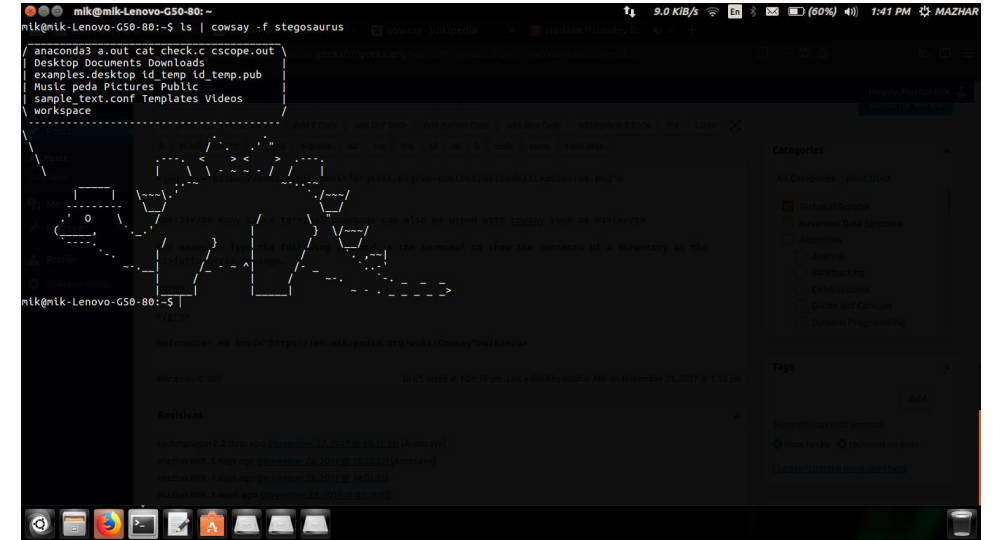
Linux Shell

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

The shell is an environment in which we can run our commands, programs, and shell scripts.

The prompt, \$, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command. It contains some basic information like the current user or the current path you are on.

Es. \$date



Linux Shell – apt-get

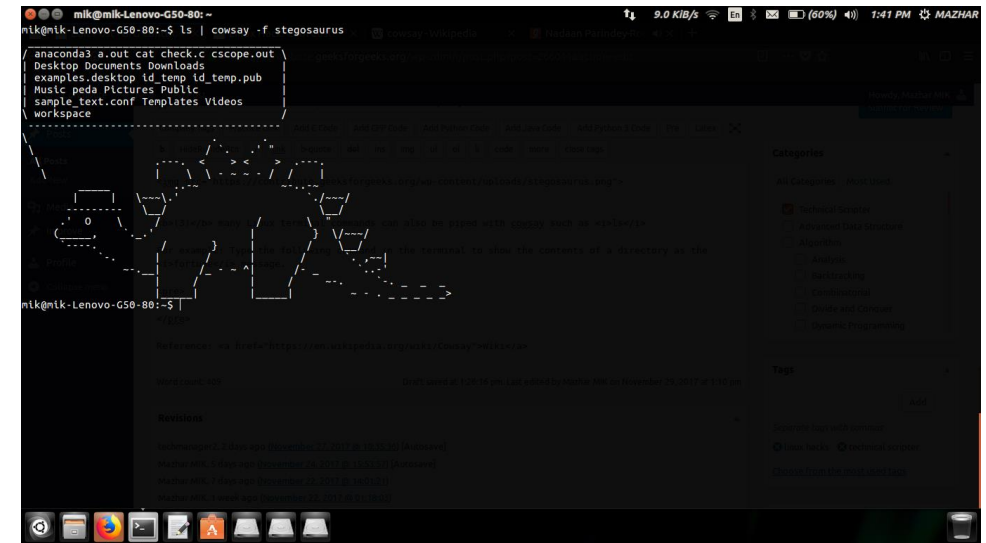
apt-get

APT (Advanced Package Tool) is the command line tool to interact with the Ubuntu packaging system. What is a packaging system??

A packaging system is a way to provide programs and applications for installation. This way, you don't have to build a program from the source code.

apt-get basically works on a database of available packages. If you don't update this database, the system won't know if there are newer packages available or not. In fact, this is the first command you need to run on any Debian-based Linux system after a fresh install.

\$ sudo apt-get update



Superuser privileges

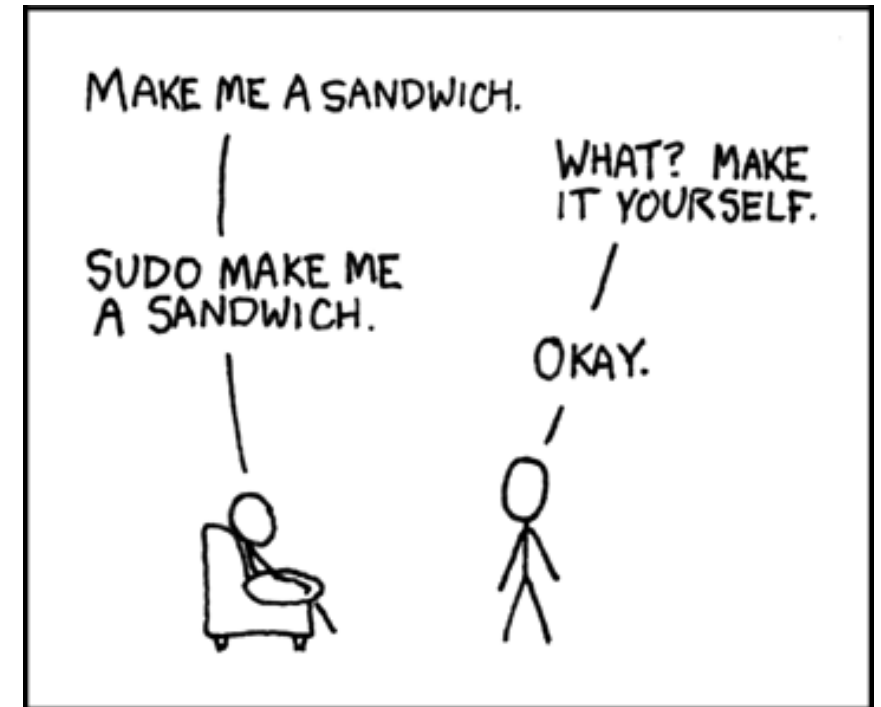
Wait, what is sudo?

It stands for *superuser do*: it allows users to run programs with the security privileges of another user, by default the superuser.

Who is the superuser?

A **superuser** is a special user account used for system administration which is capable of making unrestricted, potentially adverse, system-wide changes. In order to possibly become a “superuser”, the user should be listed in the **/etc/sudoers** file.

By default, **sudo** requires that users authenticate themselves with a password. By default, this is the user's password,



Explore the folder Hierarchy

Let's go back to the Linux shell. As many OS, Linux systems are composed of 2 main elements: **folders** and **files**. The basic difference between the two is that files store data, while folders store files and other folders. The folders, often referred to as directories, are used to organize files on your computer. The folders themselves take up virtually no space on the hard drive.

Let's check out the directory (i.e. **folder**) structure of the **my_ros/src** folder, the workspace folder for ROS. We want to do that using the command `$ tree`

If the program is not installed you can run the command

```
$sudo apt-get install tree
```

Now, how to reach the correct directory?

```
$cd /home/my_ros/src
```

The **cd** command is one of the most important ones in Linux. It allows you to go into a specific directory. When using commands like "cd" you can use the Tab key to autocomplete file and directory names

Explore the folder Hierarchy

Once you reach the correct folder, you may execute the command

\$tree

to get a snapshot of the folder hierarchy

How we move up of one folder?

\$cd ..

How to list all files in a directory?

\$ls

Directories and files will be listed on screen, with different colours depending on the read, write and execution permission.

```
root@193a91ce3203:/home/my_ros/src# tree
.
├── CMakeLists.txt -> /opt/ros/kinetic/share/catkin/cmake/toplevel.cmake
├── ExpRobolutions
│   ├── README.md
│   └── exercisel
│       ├── README.md
│       ├── custom
│       │   ├── CMakeLists.txt
│       │   ├── package.xml
│       │   ├── src
│       │   │   └── PositionServer.cpp
│       │   └── srv
│       │       └── Position.srv
│       └── exercisel
│           ├── CMakeLists.txt
│           ├── package.xml
│           ├── src
│           │   └── exercisel.cpp
│           └── world
│               ├── exercise.world
│               └── uoa_robotics_lab.png
├── custom_messages
│   ├── CMakeLists.txt
│   ├── include
│   │   └── custom_messages
│   ├── msg
│   │   └── Two.msg
│   ├── package.xml
│   ├── src
│   └── srv
│       └── Sum.srv
└── first_package
    ├── CMakeLists.txt
    ├── package.xml
    └── src
        └── example.cpp

16 directories, 19 files
```

Explore the folder Hierarchy

\$ls does not list hidden files. To list hidden files, you should run:

\$ls -a

explicitly lists all the files. You will see a . in front of the name of the hidden files.

~

But how to know all the options for a command??

\$man ls or **\$ls --help**

Other must-know things:

\$pwd (it returns the current path)

\$~ (it's equivalent to the home folder of your system)

```
-bash-4.1# cd /home/ /magento
-bash-4.1# ls -alh
total 732K
drwxr-xr-x 13 root root 4.0K Sep 7 11:07 .
drwxr-xr-x 3  4.0K Sep 7 11:07 ..
-rw-rw-rw- 1 root root 3.1K Feb 17 2016 api.php
drwxr-xr-x 6 root root 4.0K Sep 7 11:07 app
-rw-rw-rw- 1 root root 2.9K Feb 17 2016 cron.php
-rw-rw-rw- 1 root root 1.7K Feb 17 2016 cron.sh
drwxr-xr-x 3 root root 4.0K Sep 7 11:07 dev
drwxr-xr-x 7 root root 4.0K Sep 7 11:07 downloader
drwxr-xr-x 3 root root 4.0K Sep 7 11:07 errors
-rw-rw-rw- 1 root root 1.2K Feb 17 2016 favicon.ico
-rw-rw-rw- 1 root root 5.9K Feb 17 2016 get.php
-rw-rw-rw- 1 root root 6.3K Feb 17 2016 .htaccess
-rw-rw-rw- 1 root root 5.3K Feb 17 2016 .htaccess.sample
drwxr-xr-x 2 root root 4.0K Sep 7 11:07 includes
-rw-rw-rw- 1 root root 2.6K Feb 17 2016 index.php
-rw-rw-rw- 1 root root 2.3K Feb 17 2016 index.php.sample
-rw-rw-rw- 1 root root 6.4K Feb 17 2016 install.php
drwxr-xr-x 12 root root 4.0K Sep 7 11:07 js
drwxr-xr-x 16 root root 4.0K Sep 7 11:07 lib
-rw-rw-rw- 1 root root 11K Feb 17 2016 LICENSE_AFL.txt
-rw-rw-rw- 1 root root 11K Feb 17 2016 LICENSE.html
-rw-rw-rw- 1 root root 11K Feb 17 2016 LICENSE.txt
-rw-rw-rw- 1 root root 2.2K Feb 17 2016 mage
drwxr-xr-x 6 root root 4.0K Sep 7 11:07 media
-rw-rw-rw- 1 root root 886 Feb 17 2016 php.ini.sample
-rw-rw-rw- 1 root root 577K Feb 17 2016 RELEASE_NOTES.txt
drwxr-xr-x 2 root root 4.0K Sep 7 11:07 shell
drwxr-xr-x 5 root root 4.0K Sep 7 11:07 skin
drwxr-xr-x 3 root root 4.0K Sep 7 11:07 var
-bash-4.1#
```


Create Files and Directories

So, up until this point, you've been learning about some very important tools in order to be able to navigate around any Linux-based machine. In the following section, though, you are going to start learning about some tools that will allow you to actually interact with the system, which basically means that you will be able to modify it.

The **mkdir** command is another essential tool in Linux. It allows you to create a new directory. For instance, try executing the following commands:

```
$cd ~ (it will bring you to the home directory)
$mkdir my_folder
```

Now, if you execute the **ls** command again, you will be able to visualize your new folder.

```
root@193a91ce3203:~# ls -l
total 24
drwxr-xr-x 1 root root 4096 Apr 19 2018 Desktop
drwx----- 1 root root 4096 Jul 2 2019 Downloads
drwxr-xr-x 1 root root 4096 Jun 13 2019 model_editor_models
drwxr-xr-x 2 root root 4096 Sep 28 15:27 my_folder
-rw-r--r-- 1 root root 0 Sep 21 14:18 myfile
root@193a91ce3203:~#
```


Create Files and Directories

On the other hand, you can also create new files. There are several ways in which you can create a new file in Linux. I would say that the most commonly used is with the **touch** command. Go inside the folder you created in the previous section, and create a new file named **my_file.txt**.

```
$cd ~/my_folder  
$touch my_file.txt
```

Now, you will be able to visualize our new file using the **ls** command.

Great! So we have created a new file but...it's empty!

vi visual editor

vi is the default tool in Linux systems for text editing. Of course there are many tools, offering a nice graphical user interface, but there can be some situations (e.g., you are working on a remote robot) in which you may need to modify text using *vi*.

Basically, **vi** has 2 different modes: **Command** and **Insert** modes:

- *Command Mode*: This mode allows you to use commands in order to interact with the file. For instance, go to a certain line of the file, delete certain lines, etc...
- *Insert Mode*: This mode allows you to insert text into the file.

By default, *vi* opens with the **command** mode activated. In order to switch to the **insert** mode, you will have to type the character **i**. After pressing **i**, any character that you type now will be written into the file. Now, type any phrase or word you want into the file, and save it.

...

How to save it?

vi visual editor

You should go back to the **command** mode, by pressing the **esc** key of your keyboard.

Now for example you may delete one character by pressing the **x** key.

In order to save the file, you have to enter the following sequence of characters -> **:wq**.

Now, just type **enter** on your keyboard, and the file will be saved. The character **w** stands for **write** and the character **q** stands for **quit**. So basically, you are telling your file to save and exit.

Now, you can enter your file again and you should see whatever you've written into the file.

What if you want to exit the text editor without saving?

You may only type **:q**, but if you have added some new test to your file, you should explicitly tell the editor to ignore the changes you have done, by entering **:q!**

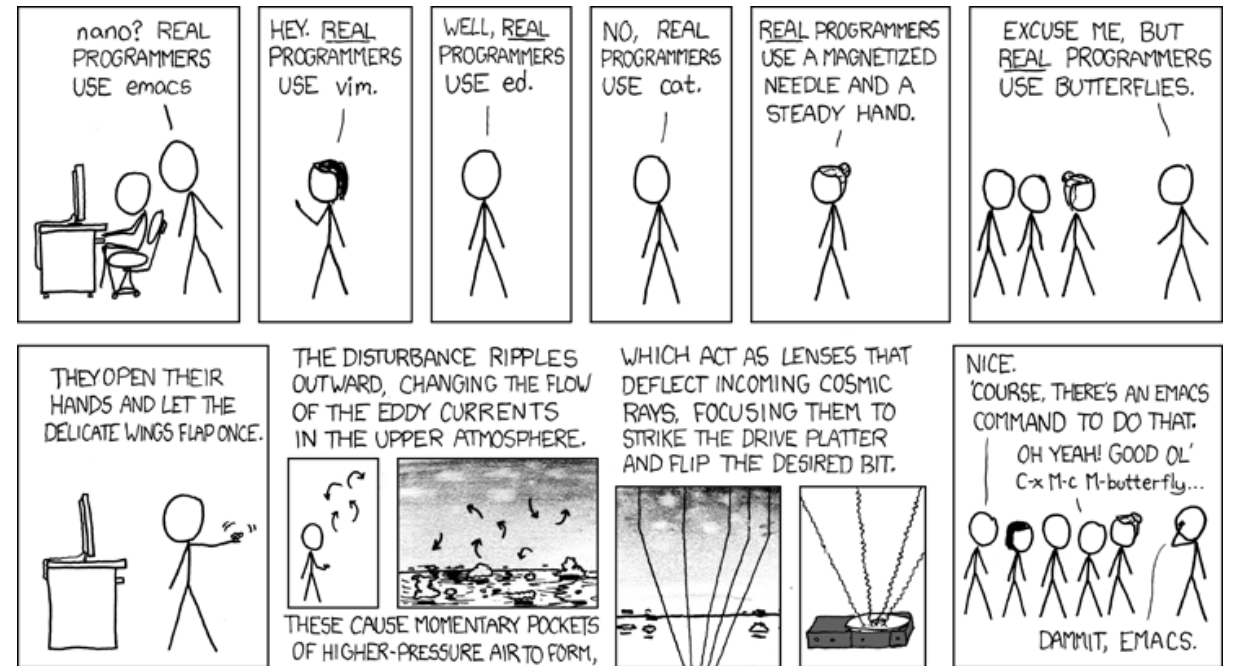
gedit text editor

Don't worry, things may be simpler in most cases.

For example, if you are working in a machine with a graphical user interface and a windows manager, you may use ***gedit***.

\$gedit my_file.txt

Gedit is the default text editor of the GNOME Desktop environment. It includes syntax highlighting for various program code and text markup formats



Commands *move and copy*

The command **mv** in Linux stands for **move**, so it's quite self-explanatory. It allows you to move files or folders from one location to another. For instance, let's try the following commands. First off, let's make sure we are on the directory of the file/folder we want to move.

```
$mv my_file.txt /home/my_ros/my_file.txt
```

In other words, the structure is:

```
mv <file/folder we want to move> <destination>
```

Commands *move and copy*

In Linux systems, the command **cp** stands for **copy**. So basically, it allows you to copy a file or a folder (or multiples) from one location to another one.

```
$cp my_file.txt /root/my_folder/my_file.txt
```

In other words, the structure is:

```
cp <file/folder we want to move> <destination>
```

However, it is NOT possible to copy a folder using the regular **cp** command. In order to copy a folder, you will need to use the **-r** argument.

```
cp -r <folder we want to move> <destination>
```

Remember that you can always run **\$man cp** or **\$cp --help** to list all the available options.

Commands *remove*

In Linux, the **rm** command stands for **remove**. So, you can use it to remove some of the files and folders that you have created:

```
$rm my_file.txt
```

Of course make sure we are in the correct folder.

For removing folders, it works exactly the same as with the **copy** command: you need to add a **-r** flag to the command.

```
$rm -r <folder to remove>
```

Explore permissions

We are going now to explore some advanced utilities that will allow you to interact deeply with a Linux system.

One of these features is related to assigning the correct permissions to file.

Let's go in one folder, and execute the command:

\$ls -la

The **-l** flag allows us to see some basic data related to the files or folders, like the permissions of the files/folders, their creation date/time, etc.

For now, let's just focus on the first part (on the left), These are the **PERMISSIONS** of the file.

```
root@193a91ce3203:/home/my_ros/build# ls -la
total 156
drwxr-xr-x 12 root root 4096 Sep 28 10:13 .
drwxr-xr-x  5 root root 4096 Sep 28 16:44 ..
-rw-r--r--  1 root root   11 Sep 28 10:13 .built_by
-rw-r--r--  1 root root    0 Sep 28 10:13 CATKIN_IGNORE
-rw-r--r--  1 root root 21217 Sep 28 10:13 CMakeCache.txt
drwxr-xr-x  9 root root 4096 Sep 28 10:13 CMakeFiles
-rw-r--r--  1 root root 2317 Sep 24 19:15 CTestConfiguration.ini
-rw-r--r--  1 root root  104 Sep 28 10:13 CTestCustom.cmake
-rw-r--r--  1 root root  397 Sep 28 10:13 CTestTestfile.cmake
drwxr-xr-x  3 root root 4096 Sep 26 00:23 ExpRobolutions
-rw-r--r--  1 root root 55348 Sep 28 10:13 Makefile
drwxr-xr-x  2 root root 4096 Sep 28 10:13 atomic_configure
drwxr-xr-x  3 root root 4096 Sep 24 19:15 catkin
drwxr-xr-x  4 root root 4096 Sep 28 10:13 catkin_generated
-rw-r--r--  1 root root  201 Sep 28 10:13 catkin_make.cache
-rw-r--r--  1 root root 6417 Sep 28 10:13 cmake_install.cmake
drwxr-xr-x  5 root root 4096 Sep 28 10:13 custom_messages
drwxr-xr-x  5 root root 4096 Sep 26 02:09 custom_package
drwxr-xr-x  4 root root 4096 Sep 28 10:13 first_package
drwxr-xr-x  4 root root 4096 Sep 28 10:13 gtest
drwxr-xr-x  2 root root 4096 Sep 24 19:15 test_results
root@193a91ce3203:/home/my_ros/build#
```


Explore permissions

Each file or directory has 3 permissions types:

read: The Read permission refers to a user's ability to read the contents of the file. It is stated with the character **r**.

write: The Write permission refers to a user's ability to write or modify a file or directory. It is stated with the character **w**.

execute: The Execute permission affects a user's ability to execute a file or view the contents of a directory. It is stated with the character **x**.

On the other hand, each file or directory has three user-based permission groups:

owner: The Owner permissions apply only to the owner of the file or directory, and will not impact the actions of other users. They are represented in the first 3 characters.

group: The Group permissions apply only to the group that has been assigned to the file or directory, and will not affect the actions of other users. They are represented in the middle 3 characters.

all users: The All Users permissions apply to all other users on the system, and this is the permission group that you want to watch the most. They are represented in the last 3 characters.

Explore permissions

So, what does **-rw-r--r--** mean?

The **owner** of the file (in this case, it's us) has **read (r*) and *write (w*) permissions, and the *group** and the **rest of users** have only **read (r)** permissions.

So, if permissions appear with a **-** symbol, it means that the permissions are not applied.

In this case, this file has no **execution** permissions. And this, if we want to actually execute the file, could be quite a problem. Then... how can we change this?

```
root@193a91ce3203:/home/my_ros/build# ls -la
total 156
drwxr-xr-x 12 root root 4096 Sep 28 10:13 .
drwxr-xr-x  5 root root 4096 Sep 28 16:44 ..
-rw-r--r--  1 root root   11 Sep 28 10:13 .built_by
-rw-r--r--  1 root root    0 Sep 28 10:13 CATKIN_IGNORE
-rw-r--r--  1 root root 21217 Sep 28 10:13 CMakeCache.txt
drwxr-xr-x  9 root root 4096 Sep 28 10:13 CMakeFiles
-rw-r--r--  1 root root 2317 Sep 24 19:15 CTestConfiguration.ini
-rw-r--r--  1 root root  104 Sep 28 10:13 CTestCustom.cmake
-rw-r--r--  1 root root   397 Sep 28 10:13 CTestTestfile.cmake
drwxr-xr-x  3 root root 4096 Sep 26 00:23 ExpRobolutions
-rw-r--r--  1 root root 55348 Sep 28 10:13 Makefile
drwxr-xr-x  2 root root 4096 Sep 28 10:13 atomic_configure
drwxr-xr-x  3 root root 4096 Sep 24 19:15 catkin
drwxr-xr-x  4 root root 4096 Sep 28 10:13 catkin_generated
-rw-r--r--  1 root root   201 Sep 28 10:13 catkin_make.cache
-rw-r--r--  1 root root 6417 Sep 28 10:13 cmake_install.cmake
drwxr-xr-x  5 root root 4096 Sep 28 10:13 custom_messages
drwxr-xr-x  5 root root 4096 Sep 26 02:09 custom_package
drwxr-xr-x  4 root root 4096 Sep 28 10:13 first_package
drwxr-xr-x  4 root root 4096 Sep 28 10:13 gtest
drwxr-xr-x  2 root root 4096 Sep 24 19:15 test_results
root@193a91ce3203:/home/my_ros/build#
```

Command *chmod*

In Linux, the **chmod** command is used to modify the permissions of a given file or directory (or many of them). There are a couple of ways to use this command, though, so let's go by parts.
Let's first modify **my_file.txt** by writing "echo hello".

Now, let's try the next command:

\$chmod +x my_file.txt

Let's now execute the **ls** command again and see how the permissions have changed.

```
root@193a91ce3203:/home/my_ros# ls -la
total 32
drwxr-xr-x  5 root root 4096 Sep 28 17:49 .
drwxr-xr-x  1 root root 4096 Sep 28 17:26 ..
-rw-r--r--  1 root root   98 Sep 24 19:15 .catkin_workspace
drwxr-xr-x 14 root root 4096 Sep 28 17:46 build
drwxr-xr-x  5 root root 4096 Sep 26 00:23 devel
-rwxr-xr-x  1 root root   13 Sep 28 17:49 my_file.txt
drwxr-xr-x  6 root root 4096 Sep 28 17:42 src
```

Command *chmod*

As you can see, the **execution** permissions, which are represented by the character **x**, have been added to all the permission groups.

What if I only want to modify the permissions for 1 of the groups? Or what if I want to remove a permission instead of assigning it?

chmod <groups to assign the permissions><permissions to assign/remove> <file/folder names>

As for the groups, you can specify them using the following flags:

u: Owner

g: Group

o: Others

a: All users. For all users, you can also leave it blank, as we did in the example command you executed before.

Permissions to assign/remove may be: (+/-) (r/w/x).

Command *chmod*

So, for example:

```
$chmod go-x my_file.txt
```

will remove the permission x (execution), only for the users Groups and Others, of the file my_file.txt.

The same thing can be done using **Binary References**.

Basically, the whole string stating the permissions (*rwxrwxrwx*) is substituted by 3 numbers. The first number represents the Owner permission; the second represents the Group permissions; and the last number represents the permissions for all other users.

Each permission has a number assigned: r = 4, w = 2, x = 1.

Then, you add the numbers to get the integer/number representing the permissions you wish to set.

```
$chmod 740 my_file.txt -> rwxr-----
```

Bash scripts

In one of the previous example, we have executed the file *my_file.txt*

What we have done, it is basically a script: a series of commands within a file capable of being executed

In Linux we have a specific way of writing scripts: **bash scripts**.

A **bash script** is a regular text file that contains a series of commands. These commands are a mixture of commands you would normally type yourself on the command line (such as the ones we have been reviewing, **cd**, **ls**, or **cp**) and also commands we could type on the command line. **Anything you can run normally on the command line can be put into a script and it will do exactly the same thing**. Similarly, anything you can put into a script can also be run normally on the command line and it will do exactly the same thing.

Let's create a *bash scripts*.

Bash scripts

\$touch bash_scripts.sh

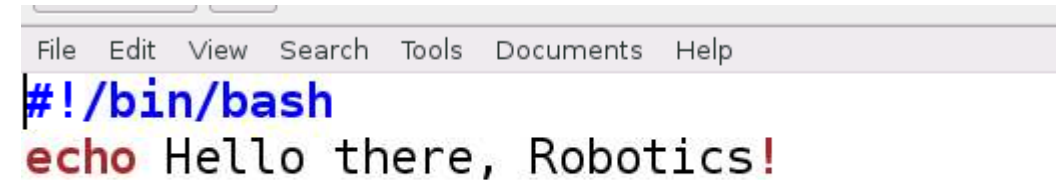
Inside the file, let's place the following contents:

```
#!/bin/bash  
echo Hello there, Robotics!
```

As you can see, the file extension is **.sh**. Usually, this is the extension you will always use when you create a new bash script. Note also that the script starts with the line **#!/bin/bash**. All bash scripts will start with this special line. Basically, it let's the Linux system know that this file is a bash script.

What if I run

\$/bash_scripts?

A screenshot of a terminal window with a menu bar at the top containing 'File', 'Edit', 'View', 'Search', 'Tools', 'Documents', and 'Help'. The terminal displays two lines of text: the first line is '#!/bin/bash' in blue, and the second line is 'echo Hello there, Robotics!' in red. The text is displayed in a monospaced font.

```
File Edit View Search Tools Documents Help  
#!/bin/bash  
echo Hello there, Robotics!
```

Bash scripts

Yes, remember to fix the permissions!

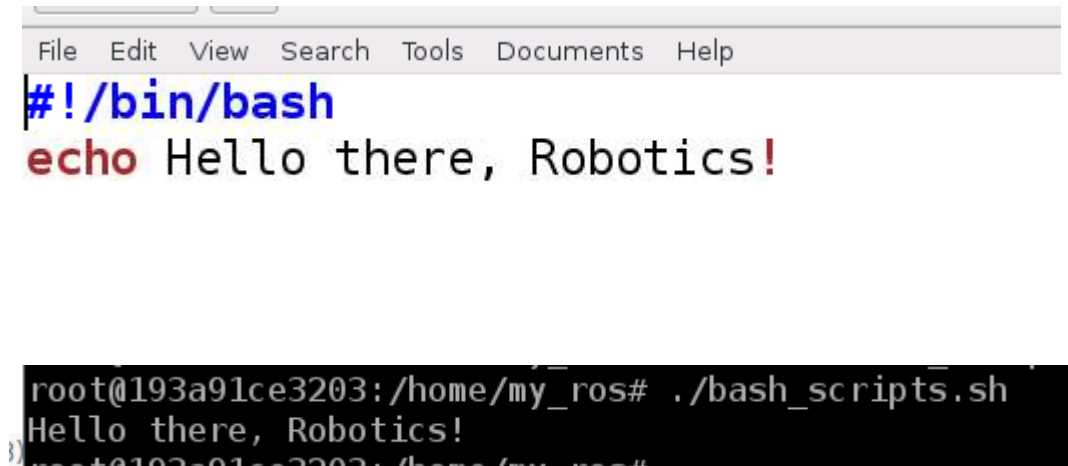
```
$ chmod +x bash_scripts.sh
```

Now you can execute the script!

In a bash script, the command **echo** is used in order to print something on the Shell.

You can also pass parameters to a bash script. This is used when you want your script to perform in a different way, depending on the values of the input parameters (also called arguments).

How?

A screenshot of a terminal window. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Tools', 'Documents', and 'Help'. The terminal content shows a blue prompt character followed by '#!/bin/bash' and a red 'echo' command followed by the text 'Hello there, Robotics!'. Below this, a black terminal window shows the command './bash_scripts.sh' being executed, resulting in the output 'Hello there, Robotics!'.

Bash scripts

In fact, it's pretty simple. You can access an argument inside a script using the variables **\$1**, **\$2**, **\$3**, and so on. The variable **\$1** refers to the first argument, **\$2** to the second argument, and **\$3** to the third argument.

So, let's change our script in this way:

```
#!/bin/bash
```

```
Echo Hello there, $1!
```

And run the script:

```
$/bash_scripts.sh students
```

```
INSTALL.SH  
#!/bin/bash  
  
pip install "$1" &  
easy_install "$1" &  
brew install "$1" &  
npm install "$1" &  
yum install "$1" & dnf install "$1" &  
docker run "$1" &  
pkg install "$1" &  
apt-get install "$1" &  
sudo apt-get install "$1" &  
steamcmd +app_update "$1" validate &  
git clone https://github.com/"$1"/"$1" &  
cd "$1";./configure;make;make install &  
curl "$1" | bash &
```

The *.bashrc* script

The **.bashrc** file is a special bash script, which Linux executes whenever a new Shell session is initialized. It contains an assortment of commands, aliases, variables, configuration, settings, and useful functions.

As you may have noticed, the **.bashrc** file is a hidden file. It is automatically generated by the Linux system and it is always placed in the HOME folder (in our case, this is **/home/user/**). However, you can still modify it in order to customize your Shell session.

The **.bashrc** script runs automatically any time you open up a new terminal, window or pane in Linux. However, if you have a terminal window open and want to rerun the **.bashrc** script, you have to use the following command:

\$ source .bashrc

Why is it important for Robotics?

It may be used to set-up the ROS environment, to set ROS workspaces, or to set ***environmental variables***.

Environmental variables

An environment variable is a named value that can affect the way running processes will behave on a computer. For example, a running process can query the value of the *HOME* variable to find the directory structure owned by the user running the process.

Environmental variables may be set by using the command ***export***

We can also run the export command by itself in order to see all the environment variables running.

\$ export

But, there are a lot of variables...

In Linux systems, the **grep** command is used in order to filter elements. For, instance, execute the following command.

\$ export | grep ros

You can always use **grep** to filter results.

Understand processes

A process refers to a program in execution. Basically, it's a running instance of a program. It is made up of the program instructions, data read from files, other programs or input from a system user.

Basically, there are 2 types of processes in Linux:

Foreground processes : These are initialized and controlled through a terminal session. In other words, there has to be a user connected to the system to start such processes; they haven't started automatically as part of the system functions/services.

Background processes: These are processes not connected to a terminal. This means that they don't expect any user input. Process may be started in background, by adding **&** after the command.

There exist several different commands that will allow you to visualize the running processes on the system. In this course, though, we are going to focus on 2 of them: **top** and **ps**.

Understand processes

top provides a dynamic real-time view of the running system. Usually, this command shows the summary information of the system and the list of **processes** or threads which are currently managed by the **Linux Kernel**

It also shows some additional information, such as cpu usage and memory occupation

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
26	root	20	0	89100	20120	12964	S	6.3	0.3	6:37.74	x11vnc
22	root	20	0	271368	78640	43028	S	5.0	1.2	4:54.90	Xvfb
12	root	20	0	56184	18436	6904	S	0.3	0.3	0:01.60	python
15	www-data	20	0	125460	3216	1600	S	0.3	0.1	0:01.09	nginx
24	root	20	0	414252	24812	19680	S	0.3	0.4	0:06.81	lxpanel
3705	root	20	0	302220	43052	24804	S	0.3	0.7	0:01.84	terminator
4422	root	20	0	89724	19896	6220	S	0.3	0.3	0:02.04	python
1	root	20	0	4364	632	564	S	0.0	0.0	0:00.34	tini
10	root	20	0	11288	2188	1984	S	0.0	0.0	0:00.00	startup.sh
13	root	20	0	125120	1448	56	S	0.0	0.0	0:00.00	nginx
14	www-data	20	0	125460	3216	1600	S	0.0	0.1	0:13.20	nginx
16	www-data	20	0	125460	3216	1600	S	0.0	0.1	0:00.71	nginx
17	www-data	20	0	125460	3216	1600	S	0.0	0.1	0:00.64	nginx
18	root	20	0	56712	18952	7092	S	0.0	0.3	0:02.88	supervisord
23	root	20	0	624828	26784	20044	S	0.0	0.4	0:02.93	pcmanfm
25	root	20	0	186404	17244	12444	S	0.0	0.3	0:02.76	openbox
27	root	20	0	87432	23324	9816	S	0.0	0.4	0:01.20	python
53	root	20	0	120352	6628	6032	S	0.0	0.1	0:00.00	menu-cached
130	root	20	0	43600	372	12	S	0.0	0.0	0:00.00	dbus-launch
131	root	20	0	42864	3236	2816	S	0.0	0.1	0:00.11	dbus-daemon
141	root	20	0	178532	4612	4156	S	0.0	0.1	0:00.05	dconf-service
3504	root	20	0	301596	43044	24936	S	0.0	0.7	0:02.40	terminator
3508	root	20	0	14872	1704	1556	S	0.0	0.0	0:00.45	gnome-pty-helpe
3513	root	20	0	20792	4508	3188	S	0.0	0.1	0:00.10	bash
3709	root	20	0	14872	1720	1572	S	0.0	0.0	0:00.44	gnome-pty-helpe
3714	root	20	0	20788	4460	3140	S	0.0	0.1	0:00.10	bash
3777	root	20	0	1956436	240696	126012	S	0.0	3.8	0:30.07	firefox
3849	root	20	0	1541092	131240	106472	S	0.0	2.1	0:03.13	Web Content
3918	root	20	0	1498148	104180	81356	S	0.0	1.6	0:07.25	WebExtensions
3961	root	20	0	1473524	73948	59784	S	0.0	1.2	0:00.15	Web Content
4293	root	20	0	640568	39668	29384	S	0.0	0.6	0:08.44	gedit
4429	root	20	0	399776	66192	18708	T	0.0	1.0	0:00.96	python
4445	root	20	0	40460	3520	3072	R	0.0	0.1	0:00.00	top

Understand processes

ps stands for process status. However, if you execute it, you will only see the processes of the user, which are attached to a terminal.

If you want to see **all** active processes, use the command:

\$ps aux

Consider that you can use the **grep** command to filter results:

Es:

\$ps aux | grep python

```
root@193a91ce3203:~/Downloads/robot-sim# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   4364    632 pts/0    Ss   16:23   0:00 /bin/tini -- /usr/bin/supervisord -n
root        10  0.0  0.0  11288   2188 pts/0    S    16:23   0:00 /bin/bash /startup.sh
root        12  0.0  0.2   56184  18436 pts/0    S    16:23   0:01 python ./run.py
root        13  0.0  0.0  125120   1448 ?        Ss   16:23   0:00 nginx: master process nginx -c /etc/nginx/nginx.conf
www-data    14  0.0  0.0  125460   3216 ?        S    16:23   0:13 nginx: worker process
www-data    15  0.0  0.0  125460   3216 ?        S    16:23   0:01 nginx: worker process
www-data    16  0.0  0.0  125460   3216 ?        S    16:23   0:00 nginx: worker process
www-data    17  0.0  0.0  125460   3216 ?        S    16:23   0:00 nginx: worker process
root        18  0.0  0.2   56712  18952 pts/0    S+   16:23   0:02 /usr/bin/python /usr/bin/supervisord -n
root        22  2.1  1.2  271368  78640 pts/0    Sl   16:23   5:10 /usr/bin/Xvfb :1 -screen 0 1920x969x16
root        23  0.0  0.4   624828  26784 pts/0    Sl   16:23   0:02 /usr/bin/pcmanfm --desktop --profile LXDE
root        24  0.0  0.3   414252  24812 pts/0    Sl   16:23   0:07 /usr/bin/lxpanel --profile LXDE
root        25  0.0  0.2   186404  17244 pts/0    S    16:23   0:02 /usr/bin/openbox
root        26  2.9  0.3   89100  20120 pts/0    S    16:23   6:58 x11vnc -display :1 -xkb -forever -shared -repeat
root        27  0.0  0.3   87432  23324 pts/0    S    16:23   0:01 python /usr/lib/noVNC/utils/websockify/run --web /usr/lib/noVNC 6081 localhost:5900
root        53  0.0  0.1  120352   6628 ?        Sl   16:23   0:00 /usr/lib/menu-cache/menu-cached /root/.cache/menu-cached-:1
root       130  0.0  0.0   43600    372 ?        S    16:39   0:00 dbus-launch --autolaunch=40dc81ed31bcebe8a699e4e15ad90669 --binary-syntax --close-std
root       131  0.0  0.0   42864   3236 ?        Ss   16:39   0:00 /usr/bin/dbus-daemon --fork --print-pid 5 --print-address 7 --session
root       141  0.0  0.0   178532   4612 ?        Sl   16:39   0:00 /usr/lib/dconf/dconf-service
root      3504  0.0  0.6  301596  43044 pts/0    Sl+  17:48   0:02 /usr/bin/python /usr/bin/terminator
root      3508  0.0  0.0   14872   1704 pts/0    S+   17:48   0:00 gnome-pty-helper
root      3513  0.0  0.0   20792   4508 pts/1    Ss+  17:48   0:00 /bin/bash
root      3705  0.0  0.6  302220  43052 pts/0    Sl+  18:55   0:02 /usr/bin/python /usr/bin/terminator
root      3709  0.0  0.0   14872   1720 pts/0    S+   18:55   0:00 gnome-pty-helper
root      3714  0.0  0.0   20788   4460 pts/2    Ss   18:55   0:00 /bin/bash
root      3777  0.6  3.8  1958484  242724 pts/0    Sl+  18:58   0:30 /usr/lib/firefox/firefox
root      3849  0.0  2.0  1541092  131240 pts/0    Sl+  18:58   0:03 /usr/lib/firefox/firefox -contentproc -childID 1 -isForBrowser -prefsLen 1 -prefMapSi
root      3918  0.1  1.6  1498148  104180 pts/0    Sl+  18:58   0:08 /usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -prefsLen 5866 -prefMap
root      3961  0.0  1.1  1473524  73948 pts/0    Sl+  18:58   0:00 /usr/lib/firefox/firefox -contentproc -childID 3 -isForBrowser -prefsLen 6588 -prefMap
root      4293  0.2  0.6   640568  39668 pts/0    Sl+  19:19   0:08 gedit file:///root/Downloads/robot-sim/test.py
root      4422  0.2  0.3   89724  19896 pts/0    S    19:58   0:03 python /usr/lib/noVNC/utils/websockify/run --web /usr/lib/noVNC 6081 localhost:5900
root      4429  0.1  1.0  399776  66192 pts/2    Tl   20:11   0:00 python run.py -c games/abc.yaml test.py
root      4456  0.0  0.0   36128   3196 pts/2    R+   20:20   0:00 ps aux
```

Kill processes

Ctrl + C is used to kill a process with the signal **SIGINT**, and can be intercepted by a program executed in a shell, so that it can clean itself up before exiting (in our case, stop the robot), or not exit at all. It depends on how the application is built.

Ctrl + Z is used for suspending a process by sending it the signal **SIGSTOP**, which cannot be intercepted by the program. Basically, it sends **SIGTSTOP** to a **foreground application**, effectively putting it in the background. Thus, the process is still there, running in the background.

How can I stop a process that is running in the background? Well, for this case, Linux provides the **kill** command. It's very easy to use, but you need to know the Process ID (**PID**) of the process you want to terminate. If you check the previous slide, when running **ps**, you get also the list of all PIDs associated to process.

Just execute:

\$ kill <PID_number> to kill the process associated to that PID number.

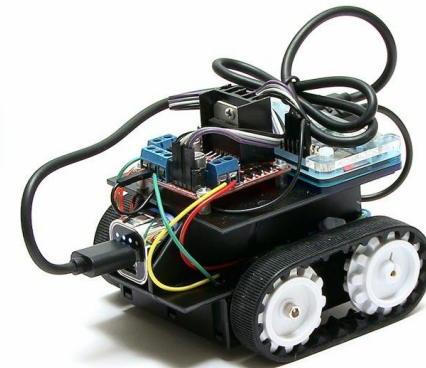
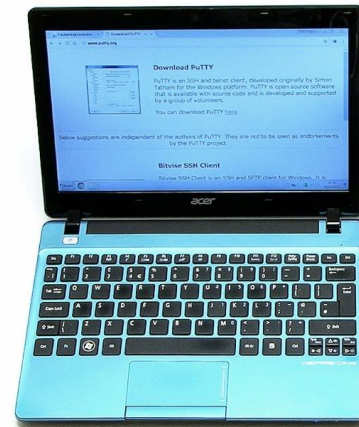
SSH protocol

Secure Shell, most commonly known as **ssh**, is a protocol that allows users to connect to a remote machine in a secure way. It is based on a Client-Server architecture. So, from your local machine (Client), you can log into the remote machine (Server) in order to transfer files between the two machines, execute commands on the remote machine, etc...

In robotics, it is mostly used to access the remote machine that runs in a real (physical) robot from any computer in order to control it and send commands to it.

\$ssh <user>@<host>

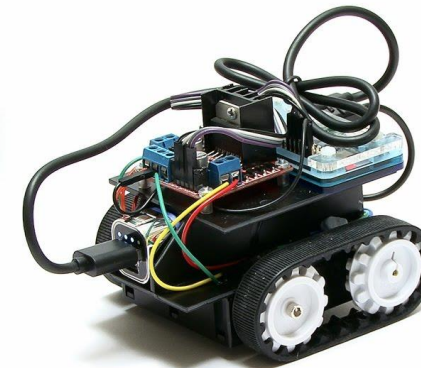
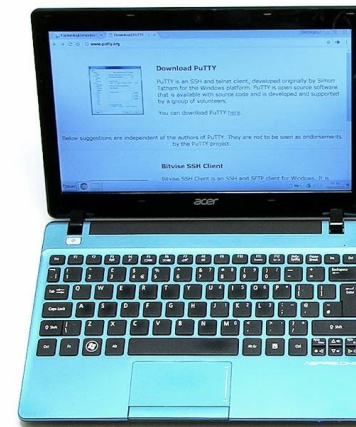
<host> makes reference to the remote machine you want to access (where the SSH Server is running), and **<user>** makes reference to the account in which you want to login from the remote machine.



SSH protocol

Please consider that the host and the client should be in the same LAN, or the host should be in some way accessible from the outside.

Finally, in order to close an ssh session and go back to your local machine, all you have to do is to execute the **exit** command on Shell.



That's all

Of course there are many other commands and things to know about Linux, but these info should be sufficient to start!

Next steps: Python and CPP!

For the next week, try to have a Linux (Ubuntu) system ready because I will propose you some exercises to do.

How?

- Docker image
- Native Ubuntu Linux
- On Windows or Mac-OS
 - Virtual Machine (VirtualBox, Wmware,..) + Ubuntu image

