# Prepare the work environment

## Operating system

These tutorials will be done in a Linux environment primarily because all the necessary tools can be downloaded from the Internet for free and no piracy is required. The next reason is that Linux requires much less resources and demonstrates normal speed on computers where even Windows 10 is too slow.

Nowadays, there are an endless number of Linux distributions, and if you read the advertisement for any variant, you will understand that this particular variant is the best. I’d recommend choosing Ubuntu: the constructors of this distribution paid a lot of attention to simplifying the installation and support. Even novice users will easily install this system in the computer. Ubuntu desktop is built for individual users and the standard installation provides almost all the software needed for work.

## Installation

Taking an old, unused laptop and installing Linux on it is the easiest way to do these lessons. And do not be afraid that Windows 10 works too slowly there: I installed Xubuntu on an Acer TravelMate with just a 2.2GHz Intel processor.

Now that professional programmers are forced to switch on Windows 11, you can buy a good enough computer at an affordable price. When buying a used one, make sure that it has a DVD drive or the BIOS allows you to boot the system from a USB stick.

You can build a dual boot system if you have a good computer and plenty of space on a hard drive or SSD. Although Linux installations are well-tested and usually uneventful, make sure to create a Windows recovery disc and backup data before installing Linux next to Windows.

Installation is described on site [Install Ubuntu desktop | Ubuntu](https://ubuntu.com/tutorials/install-ubuntu-desktop" \l "1-overview). This link describes the installation in great detail, so there's no need to repeat it here: just follow the instructions and you'll have a working operating system after the process is complete. The site describes installation from USB stick, installation from DVD is almost the same: you will need to create bootable DVD instead of the stick. I'll just add a few comments:

* Drop the [Xubuntu](https://xubuntu.org/) ISO file instead of [Ubuntu](https://ubuntu.com/download). Xubuntu installs [xfce](https://xfce.org/) desktop environment and requires much less resources from the computer, the graphical interface is very simple and you will be able to master it in a few hours. When pressed, you can always expand the system by installing additional software from the Ubuntu archives. You can also reinstall Linux and switch to GNOME or KDE, as long as your computer is powerful enough.
* The system will ask for root username and password during installation. Write down these values somewhere, as you will need them when upgrading the system.
* Choose your native language during installation - this way you will avoid many misunderstandings.

## XFCE desktop environment

Xfce is a lightweight and configurable desktop environment for Unix-like operating systems. After booting XUbuntu, the screen will look something like this:

Fig. 1: Slightly adjusted XFCE screen. I added language switching and screenshot icons there.

At the site [XFCE desktop](https://docs.openeuler.org/en/docs/21.09/docs/desktop/Xfce-user-guide.html" \l "4-shortcut-operation-bar) you will find a brief description of what can be gutted in this environment. For a complete description of the environment, see the [XFCE home site](https://docs.xfce.org/).

Right click on free space in the task bar or desktop and follow popup menu for configuring the region.



Fig. 2: Expanded taskbar management menu.

Right-click on an icon and follow the context menu instructions for updating or removing it from the taskbar. Modifying the desktop icon is exactly the same.

There is a lot of material on YouTube about configuring XFCE. I recommend watching [How to Customize XFCE](https://www.youtube.com/watch?v=mgyTCqr51iI): the tricks shown there will really impress Windows users. Don't mess around with the configuration too long: there are many free XFCE themes in the [Xfce-look](https://www.xfce-look.org/browse?cat=138&ord=latest&ref=itsfoss.com) portal. Choose the right theme for you and install it according to the instructions in the [themes](https://itsfoss.com/install-themes-xfce-xubuntu/) site.

Add a keyboard switch button to the taskbar if you need to work with multiple languages. Also a very useful thing is the workspace switcher. Activate 2 workspaces to begin with, you can increase the number later when you see benefits of this feature.

In Linux, the terminal plays a much more important role than in the Windows operating system. Watch the [video](https://www.youtube.com/watch?v=BFMyUgF6I8Y) if you have never worked with Linux or rarely used Linux terminal.

## Package manager

A package manager or package-management system is a collection of software tools that automates the process of installing, upgrading, configuring, and removing computer programs for a computer in a consistent manner ([Wikipedia](https://en.wikipedia.org/wiki/Package_manager)). Ubuntu inherited Advanced Package Tool (or APT), the main command-line package manager from Debian. The apt system consists of three console applications: apt, apt-get, apt-cache. They require admin rights in most cases thus add magical word **sudo** in front of command.

A typical Ubuntu installation has a graphical tool ([Synaptic](https://www.ubuntugeek.com/synaptic-package-manager-beginners-guide-for-ubuntu-users.html)) for working with packages. Read this article and use its recommendations for managing your system. I provide **apt** management from the terminal, since theese commands may be used even on Linux having no Xwindow. Here is an abbreviated description of the **apt** command. You will receive complete and accurate information by typing

man 8 apt

in the terminal.

The **apt** commandline is designed as end user tool and you can use it instead of specializated tools **apt-get** or **apt-cache**. Type in terminal command

*apt help*

and you will get a list of commands. Here is short description some of them.

#### List

The command **apt list** types long list of packages thus paginate output with **less** command or filter it with **grep** command:

apt list | less

apt list | grep chess

The first commmand displays paginated list, second one displays packages containing phrase chess.

#### Search

The command outputs packages with phrase inside description. Once again, you can send an output to the less or grep commands:

apt search chess | less

apt search chess | grep gnu

#### Show

The command shows package details:

apt show gnuchess-book

#### Install, Reinstall, Remove, Autoremove, Purge

Performs the requested action on one or more packages. All of these commands require administrator privileges, so start the command with the word **sudo**. Packets are separated from each other by a space. The **remove** command leaves the configuration files behind, while **purge** discards everything. By the way, **purge** can also be used for packages that were discarded with **remove**.

The listed commands may be remitted via **apt** or via **apt-get:**

sudo apt install gnuchess

sudo apt purge gnuchess

The first command installs gnuchess, second one removes it with all dependencies and configuration files.

Different packages may be installed and removed in a single command. Add the installable packages marked with a plus sign (+) in the **remove** command. Packages may be removed in the **install** command adding them with a minus sign (-).

sudo apt install tilde -vim

sudo apt remove vim +tilde

Both commands delete the text editor **vim** and install **tilde**.

After executing the **full-upgrade** command, sometimes unnecessary packages remain in the system. The **full-upgrade** command will notify you about this. In this case, run the autoremove command:

sudo apt autoremove

* + - 1. The install command allows you to install local \*.deb files as well. Type the command
      2. sudo apt install <path\_and\_name\_of\_the\_file>.deb

#### Update, Upgrade, Full-upgrade

[Canonical](https://canonical.com/), the developer and maintainer of Ubuntu, periodically publishes a new release of their operating system. The site maintains a large online repository with tens of thousands of software packages for each Ubuntu release. Before you can update the software packages installed on your Ubuntu system, you first download the latest software package information from this online repository. Your Ubuntu system needs this information to detect the availability of an upgrade for an already installed software package.

To update the software package information from the online repository, run **update** command in the terminal:

sudo apt update

Once the command completed, the last line in the output shows if updates are available for installed software packages on your Ubuntu system.

The **upgrade** command does the following:

* it upgrades a software package and even installs new packages, if its dependencies require this,
* it will never remove packages. If a package removal is required, the upgrade is not performed.

In contrast to this, the **full-upgrade** command does the same as **upgrade**, but will also remove packages if needed. Use this command upgrading from one major operating system release to the next.

Some software packages require a system reboot to complete the update. You can verify that writing command

cat /var/run/reboot-required

Reboot your system if this command reports „System restart required“. The command will find no file if your system does not require rebooting. The **reboot** command can be run from the main menu or the terminal.

#### Edit-sources

Packets can be dropped from different links (sources). Information about this is stored in the /etc/apt/sources.list file. The edit-sources command allows you to select a text editor and start editing this file.

Sudo apt edit-sources

After editing the sources, it is necessary to run the **update** and **upgrade** commands.

## GitHub, GitLab

Storing your code in external repositories is useful even when you're working alone. When working in a team, it is simply necessary. The GitHub and GitLab repositories described here are good because they work with the standard [git](https://git-scm.com/) program available on Linux, Windows and MAC OS.

A standard XUbuntu installation should have git. You can check by running

git --version

in the terminal. Install **git** if this command says it didn't find the **git** command:

sudo apt install git

GitHub and GitLab use a public/private key for authentication, so you will need to configure a local git and register the public key with the repository. Generating a new key and registering it in the GitHub repository is described [here](https://dev.to/kellycarvalho/how-to-configure-git-on-ubuntu-and-adding-ssh-key-to-github-4h5d). Read this article please.

If you already have a GitHub or GitLab account and have installed GIT on another computer, then go to that computer and find out what credentials you have registered with. This will be told by the following two commands that you need to run from the terminal (Windows users refer to the terminal as "Command Prompt"):

git config --global user.name

git config --global user.email

Use the responses from these two commands for configuring git in a Linux environment:

git config --global user.name "your\_user\_name"

git config --global user.email "your\_email"

Now generate the public/private key pair:

ssh-keygen -t ed25519 -C "your\_email"

The **ssh-keygen** command will ask for a password. You can come up with whatever you want, but not too long, because you will need to enter this password when you commit the code into repository (**push** command).

**Ssh-keygen** command will not only generate the keys, but also tell you the directory where they are placed. They will probably be in the ~/.ssh directory. This directory is hidden and the file manager will show it only after enabling the checkbox "Show hidden files" in the view menu. Open the file id\_ed25519.pub with mousepad or nano and upload this text to GitHub. The upload procedure is described on site [How to configure Git on Ubuntu](https://dev.to/kellycarvalho/how-to-configure-git-on-ubuntu-and-adding-ssh-key-to-github-4h5d).

Join some project on github and copy the ssh link:

Fig. 3: Project URL on GitHub site.

Now in your terminal navigate to the directory where you want to have the project and type the command:

git clone ssh\_address-from\_git

Git will ask for confirmation the first time you run this command with new keys. Answer "yes" and you will work without any problems later.

GitLab configuration is almost the same. Read a [book](https://about.gitlab.com/handbook/) or watch a [video](https://www.youtube.com/watch?v=8aV5AxJrHDg).

There is a very good book on [git](https://git-scm.com/book/en/v2) commands. Sooner or later you will have to read it, but for now I will only present here the most important commands needed for daily work.

#### Clone

The command copies the archive. The third optional parameter of the command specifies where to copy. This is usually a dot (current directory), but any other value may be specified.

git clone https://github.com/linuxacademy/content-source-control-git.git .

The command also allows you to copy a local archive:

git clone --local /mnt/baserepo .

The target directory must be empty. Both commands will not be executed if the current directory contains any file or directory. You can read more about the clone command at the [Git Guides](https://github.com/git-guides/git-clone) site.

#### Init

If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project’s directory and emit init there:

cd /home/user/my\_project

git init

At this point, nothing in your project is tracked yet. You will need to add the files with the following commands:

git add -A -v

git commit -m 'Initial project version'

The archive created in this way will work perfectly, but it will not be connected to remote archives (GitHub, GitLab, BitBucket, ...). The [kbroman](https://kbroman.org/github_tutorial/pages/init.html) link describes how to connect such a local archive to GitHub. Follow the instructions in the section "Connect it to github".

The --bare flag allows you to create a local git server and not use online archives:

git init --bare <directory>

You would create a bare repository to **git push** and **git pull** from, but never directly commit to it. Conventionally, repositories initialized with the **--bare** flag end in **.git**. For example, the bare version of a repository called my-project should be stored in a directory called **my-project.git**.

[Atlassian](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) describes creating a shared repository under the "Bare repositories" section.

#### Config

The **git config** command is a convenience function that is used to set Git configuration values on a global or local project level. The use of the **config** command has already been demonstrated by entering the username and email. On the [Atlassian](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) site you will find a short but sufficiently detailed description of this command. Here I will mention that the config command allows you to set:

* user’s **name**,
* **email**,
* text **editor** - the **commit** command opens a text editor if the **-m** parameter is not specified. By default this is the **vim** editor, but you can change it with one of the 8 available editors (see the [Atlassian](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) site),
* In the event of a **merge** conflict, Git will launch a "merge tool." By default, Git uses an internal implementation of the common Unix **diff** program. **Branches**, **merge conflicts** and their resolution methods will be explained in the following sections. Now it is enough to know that you can set one of 13 programs: **meld, opendiff, kdiff3, tkdiff, xxdiff**, **tortoisemerge, gvimdiff, diffuse, ecmerge, p4merge, araxis, vimdiff, emerge**.

git config --global merge.tool kdiff3

This command sets the kdiff3 program to be used for conflict resolution,

* Git supports colored terminal output and config command configures colors

git config --global color.ui false

The given command disables coloring.

The **--global** parameter in the examples restricts the scope of the settings. You can use the following three values:

* **--local** - the settings only apply to the project where git config was used,
* **--global** - the settings are valid for all projects of the currently logged-in user,
* **--system** - system-level configuration is applied across an entire machine. This covers all users on an operating system and all repos.

#### Branch

GIT is rightly proud of its branching system. Branching and merging have made GIT the primary tool for programming in the team. The GIT system always creates a **main** branch that you work on, even when you don't create new branches. Branching and Merging are well explained in the [git-book](https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging) link. Read this article carefully: the knowledge there will be useful for you when working with git or its competitors.

According to the agreement, the development of new code is carried out in the **develop** branch, you can register changes in the main branch only when working alone and the project is quite small. Command

git branch

lists branches in your local repository. You can see all project branches (local and remote) by adding the -a option to this command:

git branch -a

The **checkout** command allows you to switch from one branch to another:

git checkout gb\_01

A new local branch can be created by adding the -b option:

git checkout -b gb\_02

It will be written to server after the first **commit**. Local branch may be deleted with option delete.

git branch -d localBranchName

#### Add, Stash, Commit

The **commit** command writes modifications into a local repository but you must specify which files should be saved, so **commit** is always executed in pair with the **add** command. The **add** command includes staging list all modified files. After adding the **-A** option the command will also include new files. **Add** command with the **-v** option will print the names of the included files. Therefore, write

git add -A -v

git commit -m “some\_message“

for saving your code. The -m parameter allows you to write a short text. To record longer text, do not specify this parameter. Then git will launch a text editor (see config description).

There are often files or parts of your project, you do not want to store in the repozitory. You can list such files adding **.gitignore** file to your working directory. **.gitignore** is a simple text file, its structure is described on the [w3schools](https://www.w3schools.com/git/git_ignore.asp?remote=github) site. The **add** command can specify a file or a directory. In this case, the specified file or files belonging to the specified directory will be included into staging list (list of files that will be processed by **commit** command).

git add .

git add ../doc/letter.docx

All files from current directory and letter.docx will be included into **staging** list (list of files that the **commit** command will store in the repository).

The **checkout** command will prevent you from jumping to another **branch** if the current branch contains modified files. You can use the advice of the **commit** command and save the modified files with **add**, **commit** and **push**. Use the **stash** command if you don't want to save files for some reason:

git stash

Once you're back on your branch, be sure to restore it with the help of the stash pop command:

git stash pop

Learn more about the stash command at the [Atlassian](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) link.

#### Push, pull

The **commit** command writes modifications into your local repository, other project participants do not see these changes. You commit them to the remote repository with the push command:

git add -A -v

git commit -m "Some message"

git push

Git requires additional parameters when writing a new branch for the first time. Don't worry about it, when you get an error message, copy the command from the message and run it.

Additional information about the **push** command can be found at the [GitHub](https://github.com/git-guides/git-push) site.

The **pull** command command fetches and downloads content from the remote repository and integrates changes into your local repository. When working in a team, this command MUST be executed before making any changes to a branch that other developers are using:

git pull

Additional explanations can be found at the [W3docs](https://www.w3docs.com/learn-git/git-pull.html) link

#### Merge

**Target** is the branch into which we are going to insert the code, i.e. the branch we are in now. **Source** is the branch from which we transfer modifications of code into the target branch. Follow the steps listed below to merge the code:

1. Commit the code of the source branch to the repository, i.e. execute the

git add -A -v

git commit -m “some-message”

git push

commands there.

1. Switch to the target branch, dump the latest version of the code and merge source branch

git checkout target\_branch

git pull

git merge source\_branch

1. Push target branch

git push

Don't worry if the push command can't find anything to push. This indicates that the **merge** command performed this step itself.

#### Workflow

Use this git branching structure in your programming project:

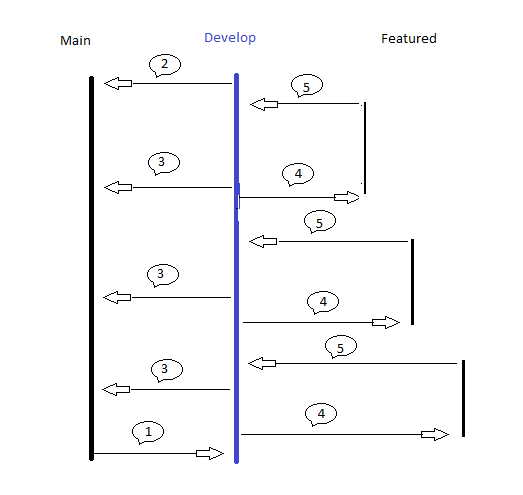


Fig. 4: Branching structure in a typical programming project.

The sequence of steps is explained below.

1. The main branch will be created for you by the repository site. Create a develop branch before starting programming. The main and develop branches remain alive for the duration of the work on the project.
2. After finishing the project, be sure to merge the develop branch into main.
3. Merge the develop branch into main before each release of the project. Those who work in Agile technology do this after completing each stage.
4. Create a featured branch before creating a new project feature. In the text of the program, it can be several interconnected functions and unit tests.
5. After completing and testing the task merge the featured branch into the develop branch.

When working in a team, it is essential to make sure that your merging does not break down due to conflicts. Follow the process a slight differently:

1. Make a commit and push on your featured branch.
2. Merge the **develop** branch into your **featured** branch.
3. Resolve conflicts if any.
4. Commit and push featured branch again.
5. Merge featured branch into develop.

Sometimes the project manager does not allow regular programmers to merge the featured branch into develop branch. In this case you have issue a merge request instead of final merging. The [Gitlab](https://docs.gitlab.com/ee/user/project/merge_requests/) site explains in detail how to create a merge request. On GitHub, this process is named a [pull request](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/incorporating-changes-from-a-pull-request/merging-a-pull-request-with-a-merge-queue). Read one or the other instruction depending on which repository you are working with.

Featured branches do not need to be kept for the entire life of the project. You can remove the featured branch after the merging.

#### Resolve conflicts

Merge conflicts happen when you merge branches that have competing commits, and Git needs your help to decide which changes to incorporate in the final merge. When working on a project alone, it is quite difficult to get a merge conflict, but when working in a team, it is a fairly common situation.

If there is a conflict, the files are merged, but you will not be able to use **commit** or **push** commands if at least one file has a conflict. The **merge** command will list files that have conflicts that you must resolve. The conflict can be resolved by opening the conflicting file in any text editor. Visual Studio Code does this especially well. Resolving conflicts with a text editor is well explained on the [Atlassian](https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts) site. If there are several conflicting files in a branch, and after fixing one file you forgot the names of the other files, the grep command will list them in the console:

grep -Hrn "<<<<<<< HEAD" .

The dot at the end of the command forces the grep command to look for files in the current directory. You can specify a directory path instead of a dot.

Git's **mergetool** command will automatically launch the conflict resolver which was specified configuring git.

git mergetool

This command will run all the conflicting files in the current directory one by one for repair. Command

git mergetool file\_name

will allow you to edit one file. You will find out all the command parameters on the man page

man git mergetool

[Ruslan Osipov](https://www.rosipov.com/blog/use-vimdiff-as-git-mergetool/) described how to fix conflicts with **vimdiff** and **mergetool**. Great article, but you need to read about text editing with the vim editor. See the [beginner's guide](https://www.youtube.com/watch?v=bR5bZriaOVU) if you have never worked with vim.

After fixing all conflicts, store your merged branch to the repository

git add -A -v

git commit -m "some message"

git push

## Docker

Different databases caused me a lot of problems when I was working as a freelancer. Some customers needed a PostgreSQL, others – MySQL or MongoDB. After installing all engines, the system consumed too many resources. After uninstalling DB engine that was not needed, in the next contract I had to install it again. Docker helped solve this problem: now I have containers with different databases and start the one needed for work in current contract.

When working in a team, Docker helps you communicate with testers: after building your fetured branch and placing software into a container, you can present your version to testers without breaking the develop branch.

[Free](https://www.freecodecamp.org/news/the-docker-handbook/" \l "hello-world-in-docker-intro-to-docker-basics)CodeCamp has a great book about docker. Be sure to read it before reading any more of my material.There are many video courses on YouTube that explain very well what Docker is and how to use it. Just type "Docker YouTube" into your browser's search box and you'll get a long list of courses. Choose one of them and listen. I recommend the "[Docker tutorial for beginners](https://www.youtube.com/watch?v=3c-iBn73dDE)". Watch it, I will provide only notes for this course.

1. The [docker hub](https://hub.docker.com/) link now requires authentication. Register and choose the free personal plan. You can always choose a better but paid plan later. The link opens a page with the imagies you have created and registered. You can see all available imagies by clicking "Explore" in the main menu.
2. The course suggested **run** command which dumps the image, creates a container with standard parameters, and runs it. Instead, use two commands: **pull** and **create**. In this case, you will be able to specify additional parameters in the **create** command.
3. You can search for the required image by connecting to the [dockerhub](https://hub.docker.com/) link or by running the **docker search** command from the terminal.
4. You can check if docker is already installed on your system by running the command

docker –version

If this command does not find **docker**, run the installation described in the [docker](https://docs.docker.com/engine/install/ubuntu/) site. Versions older than 24.0 should be updated. Follow instructions given in the [itslinuxfoss](https://itslinuxfoss.com/upgrade-docker-ubuntu/) site. After installation or updating, run the **hello-world** command from the terminal:

docker run hello-world

Install **cpu-checker** and check if your CPU supports kvm by running the commands

sudo apt install cpu-checker

kvm-ok

You will not be able to install **docker desktop** if this program says "Your CPU does not support KVM extensions". No problem, this app is just a graphical extension for the **docker**. All commands may be done from the console. It is possible that the CPU supports KVM, but it is not enabled. In this case, reboot the system, enter BIOS and activate virtualization technology (VT). Install docker desktop by following the instructions on the [ubuntu](https://docs.docker.com/desktop/install/ubuntu/) site.

#### Container and image

When working with docker, the user runs an application packaged in a **container**. By definition, a container is lightweight and portable run-time environments in which users can run applications in isolation from the underlying host machine. The container is created from the **image** by running the run command.

docker run [OPTIONS] IMAGE [COMMAND] [ARG…]

The **run** command has many options and arguments which are listed on the [docker](https://docs.docker.com/engine/reference/commandline/run/) site. I will explain some of them later, now we will talk about the IMAGE parameter. The name or ID of the image is specified here. You can download the image from the dockerhub repository or create your own using a text file with the constant name **Dockerfile**. Think of a Docker container as a running image instance. You can create many **containers** from the same **image**, each with its own unique name, data and state.

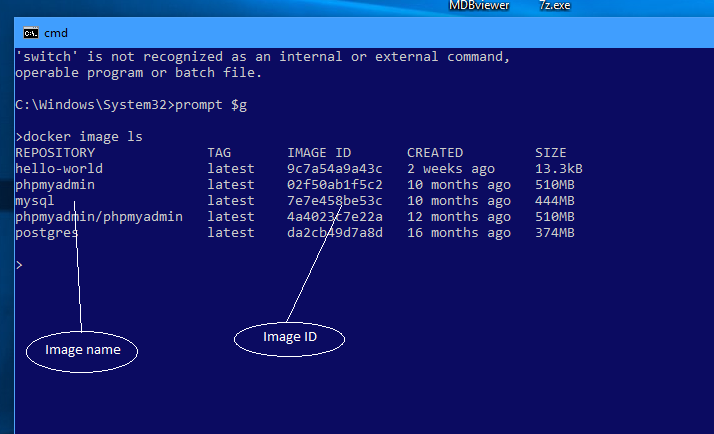
The computer from which you issue the **run** command is called the **host** computer. Linux terminal can be connected to a remote computer using [SSH](https://opensource.com/article/20/9/ssh). Docker commands will run on remote computer after that. In this case, the host is the remote computer.

After emitting the **run** command, the docker **daemon** searches for an **image** on host computer. If it doesn't find such an image, the daemon downloads it from the **dockerhub** repository and stores the image into a special directory. The subsequent **run** commands will use the local copy of the **image**.

A list of the images you will get by running the command

docker image ls

This command will print all images saved on your computer.

Fig. 5. List of images.

The rm command will remove unused images

docker image rm [OPTIONS] IMAGE [IMAGE…]

Only by specifying the **-f** option will you be able to drop the image that was used for creation of a container.

You can see the list of active containers by running one of the two commands:

docker ps

docker container ls

Adding the -a option will show all containers.

docker ps -a

docker container ls -a

Use komand **rm** for removing inactive container

docker rm [OPTIONS] CONTAINER [CONTAINER…]

Read about this command on [docker](https://docs.docker.com/engine/reference/commandline/rm/) site.

[Prune](https://docs.docker.com/engine/reference/commandline/container_prune/) command removes all inactive containers.

docker container prune [OPTIONS]

#### Run

The docker container is completely isolated from the host. You need to map host resources into container’s internal resources. Otherwise, you won't be able to interact with the containerized application. This mapping is specified in parameters of the **run** command.

#### Start Stop Restart

The run command always creates a new container. To start the existing container, use the **start** command, the **stop** command will stop the container.

docker container start mysql-server

docker container stop mysql-server

If the container for some reason does not listen to the **stop** command, then the more brutal **kill** command can be used:

docker container kill mysql-server

The Docker environment sends a SIGTERM signal after using the **stop** command. The kill command sends a SIGKILL signal to the program. The difference between these two commands is described on the [linux](https://linuxhandbook.com/sigterm-vs-sigkill/) site.

Sometimes you want to run a container and delete it immediately after it exits. Docker provides the --rm command line option for this purpose:

docker run --rm <any other options> IMAGE

The container **restart** command follows the exact syntax as the container start command.