# Preparation of the working 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 instaFdockerfilellation provides almost all the software needed for work.

## Installation

Taking an old, unused laptop and installing Linux there 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.

Remember to refresh your system from time to time by running two commands from the terminal:

sudo apt update

sudo apt upgrade

The first command updates the internal directory of the **apt** system, the second one installs the updates.

## 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

#### Mv

Command **mv** renames the file. The command requires two parameters:

* **file\_from** – original name of the file,
* **file\_to** – new name.

git mv file\_from file\_to

After executing the **commit** and **push** commands the file name will be changed in the remote and local archives.

#### Reset

To remove a file from the uncommitted list, use the **reset** command:

git reset file\_name

If GIT doesn't want to execute this command, use a more brutal one:

git rm --cached file\_name

#### 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:

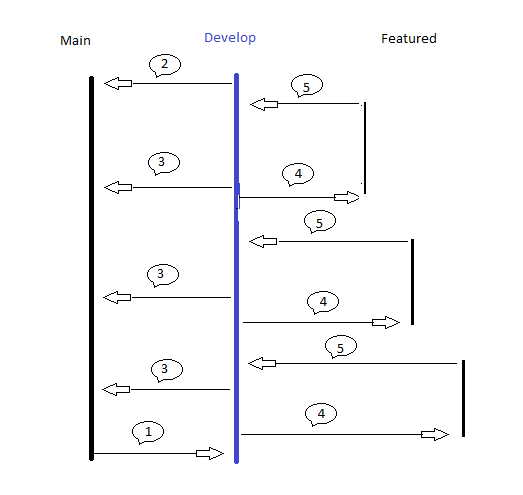


Fig. 4: Branching structure in a typical programming project.

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

## Remote Access (SSH)

Currently, there are many systems that allow you to connect to another computer remotely. Here I will cover SSH, which all Linuxes inherit from the UNIX system. The client part is usually installed on Ubuntu Linux. You can check this by running the command:

apt list –installed | grep ssh

You should see three packages in the list: **openssh-client**, **openssh-server**, **openssh-sftp-server**. Use these commands if something is missing from your list:

sudo apt install openssh-client

sudo apt install openssh-server

The **openssh-server** package installs **openssh-server** and **openssh-sftp-server**.

#### SSH client

Once the installation is complete, the SSH service will start automatically. You can verify that SSH is running by typing:

sudo systemctl status ssh

You should see **active (running)** in the response. The **systemctl** command allows you to manage **ssh** with commands:

* **start** - start OpenSSH Server,
* **stop** - stop OpenSSH server,
* **restart** - restart OpenSSH server.

Format of all three commands is identical:

sudo systemctl {start | stop | restart} ssh

When SSH is active, you can connect to your Linux box from other computers or vice versa - from your Linux to other computers. Type the command:

ssh user\_name@ip\_address

**user\_name** is a name of the user on the remote computer,

**ip\_address** - IP address of the remote computer. A computer has two IP addresses: one on the local network, and external, accessible via the Internet. Use a local IP when both computers are on the same local network.

The secure copy command (**scp**) allows you to securely copy files to and from the remote box;

scp filename.extension [remoteuser@remotebox](mailto:remoteuser@remotebox):/directory

To copy an entire directory (and all of its contents) from the local machine to the remote server, use the recursive -r switch:

scp -r /local/directory [remoteuser@remotebox](mailto:remoteuser@remotebox):/remote/directory

Install **FileZilla** if you prefer a graphical interface:

sudo apt install filezilla

Contact from Windows

Windows 10/11 comes with SSH client but SSH server needs to be installed. Installation is described on the [wincp.net](https://winscp.net/eng/docs/guide_windows_openssh_server) link. You will need the [WinSCP](https://winscp.net/eng/download.php) program for transfering files using the SFTP protocol. Download and install by following this [guide](https://winscp.net/eng/docs/guide_install).

Contact from IOS (IpadOS)

IOS does not have an SSH client, but it can be downloaded from the APP store. The '[Best SSH Terminal Apps](https://www.noupe.com/inspiration/5-best-ssh-terminal-apps-for-iphone.html)' site lists five of the best applications that you can throw away or buy the 'professional' version. I tried the free version of [iTerminal](https://apps.apple.com/us/app/iterminal-ssh-telnet-client/id581455211). I can't say that it's the best, but it works and meets all my needs.

Contact from Android

Android does not have an ssh client in the standard package, but it can be installed from googlePlay. I installed [Termius](https://play.google.com/store/apps/details?id=com.server.auditor.ssh.client&hl=en_US). It works but font size on mobile is too small. I had to increase the font. Second problem: there is no $ sign on virtual keyboard. I haven't found a way to enter the **prompt $g** command.

## 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.

[FreeCodeCamp](https://www.freecodecamp.org/news/the-docker-handbook/" \l "network-manipulation-basics-in-docker) has a great book about docker. [Syncfusion](https://www.syncfusion.com/succinctly-free-ebooks/docker-succinctly) site published free e-book in PDF format. Be sure to read one or another 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

**Docker desktop** application will greatly help you in your daily tasks of working with docker containers, but it requires a processor that supports virtualization. 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.

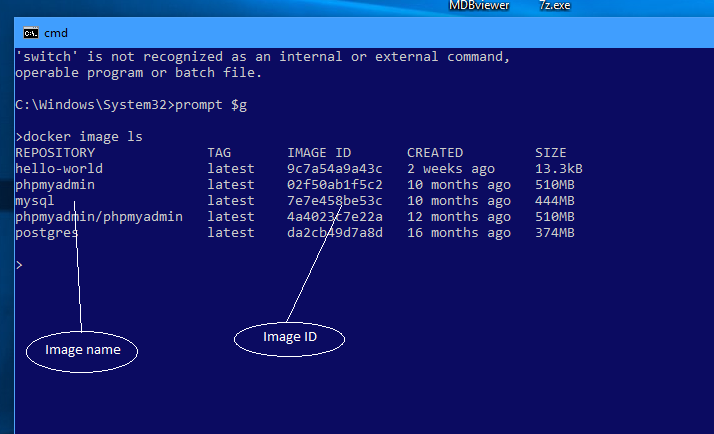
The computer from which you issue the **run** command is called the **host** computer. Linux terminal can be connected to a remote computer using [SSH](https://opensource.com/article/20/9/ssh). Docker commands will run on remote computer after that. In this case, the host is the remote computer.

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]

#### Create volume

* + - 1. Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker.

docker volume create volumeName

This command will create the volume and give it the name specified in the last parameter. The system will generate a unique name by itself if it is not specified in the command.

docker volume ls

The **ls** command will print a list of volumes.

docker volume inspect volumeName

The command will provide information about the volume.

Volumes are stored on the host computer, so the data written to the volume is not lost after stopping or even deleting the container. You can read them after starting this or another container.

#### Create network

Docker networking is used for communication between a host and a container or between multiple containers connected to the same network.

docker network ls

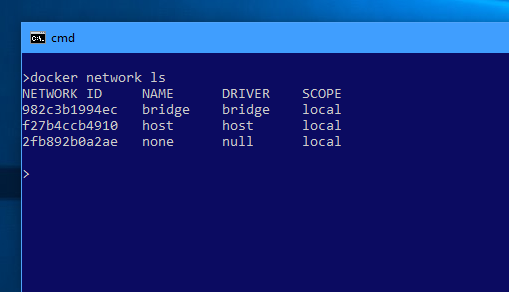


Fig. 6. List of default networks.

This command will show three networks even without running any container. Each network has a unique ID and name. There are six different drivers, three of which are important for the average programmer:

1. **bridge** - The default networking driver in Docker. This can be used when multiple containers are running in standard mode and need to communicate with each other.
2. **none** - This driver disables networking for containers altogether.
3. **host** - Removes the network isolation completely. Any container running under a host network is basically attached to the network of the host system.

You can read about the remaining three drivers (**overlay**, **ipvlan**, **macvlan**) in the [docker](https://docs.docker.com/network/) documentation.

Any container you run will be automatically attached to the bridge network. The **inspect** command shows the IP address of the container in this network. The inspect command prints a lot of information, use the --format option and output only what you are interested in:

docker inspect --format='{{range .NetworkSettings.Networks}} {{.IPAddress}} {{end}}' mysql-server

Suppose this command prints 172.17.0.2. Then, with this IP address and port number (3306), you can call the mySql server from another container using IP address 172.17.0.2:3306. The problem is that this address can change when the container is destroyed and recreated. You solve the problem by creating your own network in docker and connecting the container to that network:

docker network create [OPTIONS] NETWORK\_NAME

If you don’t specify the --driver option, the command automatically creates a bridge network for you. The **network create** command is described on the [docker](https://docs.docker.com/engine/reference/commandline/network_create/) site.

You can connect the container to a network in two ways:

1. use the **network connect** command to attach a container to a network

docker network connect NETWORK CONTAINER

1. using the --network option in the **container run** or **container create** commands.

The article '[Networking with standalone containers](https://docs.docker.com/network/network-tutorial-standalone/)' provides very detailed information about bridge networking.

#### cp

The information inside the container is completely isolated from the host computer. This rule also applies to information recorded in the volume. Although the '**docker volume inspect**' command tells you in which directory the volume information is stored, it is quite difficult to access it from the host computer. Use the **cp** command to copy information from the container to the host or back (from the host to the container):

docker cp source\_path destination\_path

Source\_path or destination\_path may be prefixed with container’s ID or name with colon (:).

docker cp my\_container:source\_path host\_destination\_path

docker cp host\_source\_path my\_container:pth\_inside\_container

See the [Docker](https://docs.docker.com/engine/reference/commandline/cp/) site for a detailed description of the **cp** command.

#### Run create

Use these two commands for creating container from an image:

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

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

The **create** command creates a container, the **run** command creates a container and starts it for execution. In the **run** (or **create**) command, everything written after the image name or ID is interpreted as a command to be executed after the container is started. That property can be used to provide initial values. There is a nice example of this usage on the [internet](https://gnisitricks.de/2020/09/Run/Start-Docker-Container-and-execute-command/) (bash -c "some\_string" reads its parameters from the provided text constant).

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.

docker run -p HostPort:ContainerPort ...

In this command, HostPort and ContainerPort are numeric port numbers. These values can be the same: **-p 5432:5432**. This mapping makes the postgreSQL port (5432) visible in the host computer.

docker run -v HostDirectory:ContainerDirectory:Options …

**HostDirectory** and **ContainerDirectory** are absolute paths to directories (on the host computer and inside the container). The path to the file in the container is always calculated from root, so the initial / is not necessary: /etc/data and etc/data points to the same directory. A **volume name** can be specified instead of HostDirectory. In this case, the data in the volume will be visible inside container’s directory. The last, optional part (Options) is not mandatory. Don't write a trailing colon when the mapping has no options. If multiple options are specified in a command, they are separated from each other by a comma. Current versions of docker recommend using **--mount** instead of **-v**. This command requires more typing, but provides more configuration options. Read about –mount on [docker](https://docs.docker.com/storage/volumes/) site (section "Choose the -v or --mount flag").

The container can be connected to the network using **–network** option:

docker run --network NETWORK …

Network name or ID can be used here. The name of the container can be specified in the --name option:

docker run --name test -it debian

All options for the **RUN** command are listed on the [docker](https://docs.docker.com/engine/reference/commandline/run/) site.

#### 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. It stops a running container and immediately starts it. For a stopped container, this command will work the same as the start command.

docker container restart mysql-server

#### Create image

DockerHub offers quite a lot of different images, but sometimes all of them do not fully meet our requirements. In this case, you have to take the most suitable candidate and modify it creating **Dockerfile** in some directory. Docker will build images automatically by reading the instructions from a **Dockerfile**. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

Here’s a very simple Dockerfile that shows all the main instructions—this image is for a basic app that listens for input on a specific port and echoes out any input it receives to a file.

FROM ubuntu

RUN apt-get update && \

apt-get install -y netcat-openbsd

ENV LOG\_FILE echo.out

COPY ./echoserver.sh /echoserver.sh

RUN chmod +x /echoserver.sh

EXPOSE 8082

VOLUME /server-logs

CMD /echoserver.sh

Once the **Dockerfile** is created, the image can be produced using the [build](https://docs.docker.com/engine/reference/commandline/build/) command:

docker build [OPTIONS] PATH | URL | -

The standard **Dockerfile** name may be changed, but then you will have to specify the actual name of this file in the **-f** option of the **build** command. A full description of the **Dockerfile** is provided on the [docker](https://docs.docker.com/engine/reference/builder/) site. Here is the format of the Dockerfile:

# Comment

INSTRUCTION arguments

INSTRUCTION arguments

…

Docker treats lines that begin with # as a comment, unless the line is a valid parser directive. A # marker anywhere else in a line is treated as an argument.

Directives

Parser **directives** are optional, and affect the way in which subsequent lines in a Dockerfile are handled. Parser directives do not add layers to the build, and will not be shown as a build step. Parser directives are written as a special type of comment in the form **# directive=value**. A single directive may only be used once. The unknown directive is treated as a comment. Two directives can be used in Dockerfile: **syntax** and **escape**.

The **syntax** directive is only available when using the **BuildKit** backend, and is ignored when using the classic builder backend. The escape directive has two forms:

# escape=\ (backslash)

# escape=` (backtick)

FROM

A Dockerfile must begin with a **FROM** instruction. This may be after parser directives, comments, and globally scoped ARGs. The **FROM** instruction specifies the Parent Image from which you are building:

FROM [--platform=<platform>] <image> [AS <name>]

FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]

FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]

The **FROM** instruction initializes a new build stage and sets the Base Image for subsequent instructions.

* **ARG** is the only instruction that may precede **FROM** in the Dockerfile.
* **FROM** can appear multiple times within a single **Dockerfile** to create multiple images or use one build stage as a dependency for another.
* The optional **name** can be used in subsequent **FROM** and **COPY** --from=<name> instructions to refer to the image built in this stage.
* The **tag** or **digest** values are optional. If you omit either of them, the builder assumes a **latest** tag by default.

ENV

The **ENV** instruction sets the environment variable <key> to the value <value>.

ENV <key>=<value> ...

ENV <key> <value>

The first version allows you to set multiple values in one line. You must write a new ENV instruction for each variable when using the second form.

Environment variables are notated in the **Dockerfile** either with **$variable\_name** or **${variable\_name}**. They are treated equivalently and the brace syntax is typically used to address issues with variable names with no whitespace, like ${foo}\_bar.

Variables are supported by the following list of instructions in the Dockerfile:

ADD COPY ENV

EXPOSE FROM LABEL

USER VOLUME STOPSIGNAL

WORKDIR

RUN

The **RUN** instruction has 2 forms:

RUN <command>

RUN ["executable", "param1", "param2"]

First type of the instruction is named **shell form**, second one – **exec form**. The **RUN** instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.

In the shell form you can use a \ (backslash) to continue a single RUN instruction onto the next line.

RUN /bin/bash -c 'source $HOME/.bashrc && \

echo $HOME'

The exec form does not invoke a command shell. This means that normal shell processing does not happen. If you want shell processing then either use the shell form or execute a shell directly:

RUN [ "sh", "-c", "echo $HOME" ]

A whole series of parameters can be specified in the RUN command: **mount**, **network**, **security**. A description of these parameters can be found in the [documentation](https://docs.docker.com/engine/reference/builder/) site.

CMD

The **CMD** instruction has three forms:

CMD ["executable","param1","param2"] (exec form, this is the preferred form)

CMD ["param1","param2"] (as default parameters to ENTRYPOINT)

CMD command param1 param2 (shell form)

There can only be one **CMD** instruction in a Dockerfile. If you list more than one CMD then only the last CMD will take effect. The main purpose of a **CMD** is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an **ENTRYPOINT** instruction as well. If **CMD** is used to provide default arguments for the **ENTRYPOINT** instruction, both the **CMD** and **ENTRYPOINT** instructions should be specified with the JSON array format.

The exec form is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').

**CMD** does not execute anything at build time, but specifies the intended command for the image.

LABEL

The **LABEL** instruction adds metadata to an image. A **LABEL** is a key-value pair. To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing. An image can have more than one label.

LABEL "com.example.vendor"="ACME Incorporated"

LABEL com.example.label-with-value="foo"

LABEL version="1.0"

LABEL description="This text illustrates \

that label-values can span multiple lines."

To view an image’s labels, use the **docker image inspect** command. You can use the --format option to show just the labels:

docker image inspect --format='{{json .Config.Labels}}' myimage

EXPOSE

The **EXPOSE** instruction informs Docker that the container listens on the specified network ports at runtime.

EXPOSE <port> [<port>/<protocol>...]

You can specify whether the port listens on **TCP** or **UDP**, and the default is **TCP** if the protocol is not specified. The **EXPOSE** instruction does not actually publish the port. To actually publish the port when running the container, use the **-p** flag on docker run to publish and map one or more ports. By default, **EXPOSE** assumes **TCP**. You can also specify **UDP**:

EXPOSE 80/udp

To expose on both TCP and UDP, include two lines:

EXPOSE 80/tcp

EXPOSE 80/udp

Regardless of the EXPOSE settings, you can override them at runtime by using the -p flag:

docker run -p 80:80/tcp -p 80:80/udp …

The docker network command supports creating networks for communication among containers without the need to expose or publish specific ports.

ADD

The ADD instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>. ADD has two forms:

ADD [--chown=<user>:<group>] [--chmod=<perms>] [--checksum=<checksum>] \

<src>... <dest>

ADD [--chown=<user>:<group>] [--chmod=<perms>] ["<src>",... "<dest>"]

The latter form is required for paths containing whitespace. The **--chown** and **--chmo**d features are only supported on Dockerfiles used to build Linux containers. Only octal notation is currently supported in **chmod** option.

Multiple <src> resources may be specified but if they are files or directories, their paths are interpreted as relative to the source of the context of the build. Each <src> may contain wildcards and matching will be done using Go’s [filepath.Match](https://pkg.go.dev/path/filepath" \l "Match) rules. The <src> path must be inside the context of the build; you cannot ADD ../something /something.

The <dest> is an absolute path, or a path relative to **WORKDIR**, into which the source will be copied inside the destination container.

All new files and directories are created with a **UID** and **GID** of 0, unless the optional **--chown** flag specifies a given username, groupname, or UID/GID combination to request specific ownership of the content added. The format of the --chown flag allows for either username and groupname strings or direct integer UID and GID in any combination. Using numeric IDs requires no lookup and will not depend on container root filesystem content.

ADD --chown=55:mygroup files\* /somedir/

ADD --chown=bin files\* /somedir/

ADD --chown=1 files\* /somedir/

ADD --chown=10:11 files\* /somedir/

ADD --chown=myuser:mygroup --chmod=655 files\* /somedir/

In the case where <src> is a remote file URL, the destination will have permissions of 600. If the remote file being retrieved has an HTTP Last-Modified header, the timestamp from that header will be used to set the mtime on the destination file.

Additional features of this command are described on [docker](https://docs.docker.com/engine/reference/builder/" \l "add) site.

COPY

COPY and ADD are both Dockerfile instructions that serve a similar purpose. They let you copy files from a specific location into a Docker image.

COPY [--chown=<user>:<group>] [--chmod=<perms>] <src>... <dest>

COPY [--chown=<user>:<group>] [--chmod=<perms>] ["<src>",... "<dest>"]

The **COPY** instruction copies file or directory from your host (the machine-building the Docker image). **ADD** does that same but in addition, it also supports 2 other sources:

* A URL instead of a local file/directory.
* Compressed archyve

If you are copying local files to your Docker image, always use **COPY**.

[Dockerfile reference | Docker Documentation](https://docs.docker.com/engine/reference/builder/)

#### Compose

Let's say you decide to install the mysql and phpmyadmin containers. You can also do this without using docker-compose:

1. Drop images from sites [mysql - Official Image | Docker Hub](https://hub.docker.com/_/mysql) , [phpmyadmin - Official Image | Docker Hub](https://hub.docker.com/_/phpmyadmin). Both sites provide pull commands in the top right corner of the page. Just copy them to the terminal and run them one by one.

docker pull mysql

docker pull phpmyadmin

1. Now you need to create a network common for both containers and a volume where **mysql** will store its data.

docker network create mySqlNet

docker volume create mySqlData

1. The [mysql](https://hub.docker.com/_/mysql) site provides parameters of a **run** command. I shall use these ones

docker run --name mySqlSrv -e MYSQL\_ROOT\_PASSWORD=123 \

-e MYSQL\_DATABASE=testDB --network=mySqlNet \

-p 3306:3306 -v mySqlData:/var/lib/mysql -d mysql

A backslash (\) at the end of a line indicates the continuation of the command on the next line. Use the caret (^) when working with the Windows terminal. Parameters of the command:

**--name mySqlSrv** – name of the container,

**-e MYSQL\_ROOT\_PASSWORD=123** – password for the root user; you should use a stronger password if the database will be accessed externally,

**-e MYSQL\_DATABASE=testDB** – create empty database testDB after running the container; you will need some kind of database for running the mysql client from the terminal,

**--network=mySqlNet** – connect the container to the **mySqlNet** network,

**-p 3306:3306** – expose default mySql port; without using this parameter you will not be able to call mySql from outside,

**-v mySqlData:/var/lib/mysql** - all database information (directory **/var/lib/mysql**) will be redirected to the **mySqlData** volume,

**-d** – detach from the terminal,

**mysql** – name of the image.

1. After stopping the container, test it by running the commands listed below:

> docker container start mySqlSrv

> docker container exec -it mySqlSrv sh

# mysql -u root -p testDB

mysql> show databases;

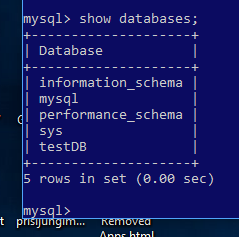


Fig. 7. List of databases.

You have done everything correctly and mySql server is working if you see something like this. Exit the database and the sh command interpreter by typing **\q** and **exit**.

1. Make sure mySqlSrv is running and create phpmyadminSrv with the commands; here I have listed and explained only the most necessary parameters, the rest of the parameters can be found in the [phpadmin](https://hub.docker.com/_/phpmyadmin) site:

docker ps

docker run -d -p 8080:80 -e PMA\_HOST=mySqlSrv \

-e PMA\_USER=root -e PMA\_PASSWORD=123 \

--name phpmyadminSrv --network mySqlNet phpmyadmin

A backslash (\) at the end of a line indicates the continuation of the command on the next line.

**-d** – detach from the terminal,

**-p 8080:80** – expose default Http port (80) as port 8080 on host machine,

**-e PMA\_HOST=mySqlSrv** - a reference to the container where the mySql server is installed,

**-e PMA\_USER=root** - mySql database user; any other user can be used here, if it was created in the **mysql run** command,

**-e PMA\_PASSWORD=123** - password that was created in the **mysql run** command,

**--name phpmyadminSrv** - name of the container,

**--network=mySqlNet** – connect the container to the **mySqlNet** network,

**phpmyadmin** - name of the image.

1. If you did everything right, the **docker ps** command should show you two containers: **phpmyadminSrv** and **mySqlSrv**.
2. When you launch any browser and type **localhost:8080** in the address bar, you will see the phpAdmin page.

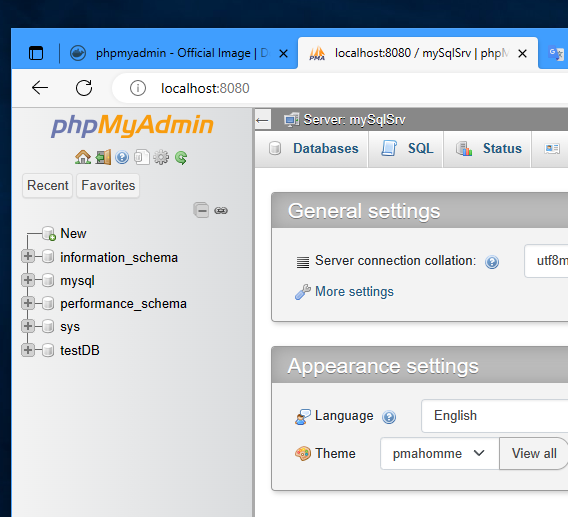


Fig. 8. PhpAdmin main window.

As you can see, networking with two containers is not a big problem, but what will you do when your project grows to five or more containers? Starting them in the right order is quite tricky.

**Docker-compose** provides a solution to this problem: you create a file that lists all the project elements (containers, networks, volumes, ...) and run this file using the **docker-compose** command

docker compose up

This [command](https://docs.docker.com/engine/reference/commandline/compose_up/) will take the **docker-compose.yml** file in your local directory and start building the containers.

Windows users will get **docker-compose** by installing **docker-desktop**, Linux users must install this application by following the instructions on the [DigitalOcean](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-20-04) site. There are many articles about **docker-compose** on the web. You can read [this](https://linuxhint.com/docker-compose-tutorial/) one. Here I will explain how to build docker-compose.yml for mySQL database and phpAdmin program.

The **docker-compose.yml** JSON file shows a tree that contains the same elements we used in the docker run command. The file must start with the version line followed by the top-level elements:

version: "3.8"

services:

...

volumes:

...

networks:

…

All top-level elements are listed on the [docker](https://docs.docker.com/compose/compose-file/) site, three names (services, volumes, networks) are important in the example under consideration. You can arrange them in any order.

version: "3.8"

services:

mySqlService:

image: mysql

container\_name: mySqlSrv

restart: "no"

environment:

- MYSQL\_DATABASE=testDB

- MYSQL\_ROOT\_PASSWORD=123

ports:

- 3306:3306

volumes:

- mySqlData:/var/lib/mysql

networks:

- mySqlNet

phpmyadminService:

image: phpmyadmin

container\_name: phpmyadminSrv

restart: "no"

environment:

- PMA\_HOST=mySqlSrv

- PMA\_USER=root

- PMA\_PASSWORD=123

ports:

- 8080:80

networks:

- mySqlNet

volumes:

mySqlData:

external: true

networks:

mySqlNet:

external: true

This docker-compose.yml file creates the same two containers. You can write this file with any text editor, but Visual Studio Code works best. Enter the commands listed below in the terminal:

cd ~

mkdir composeTest

cd composeTest/

touch docker-compose.yml

code . &

After writing this file, type the command in the terminal:

docker-compose up -d

After launching any WEB browser and typing **localhost:8080** in the address bar, you will see the phpAdmin page.

From the point of view of docker-compose, every container is a service and must be listed inside the **services** element . The service name (mySqlService) can match the container name (mySqlSrv), but the container\_name element must still be provided. Otherwise docker-compose will create its own container name and you will have problems specifying PMA\_HOST. The restart policy is described on the [Baeldung](https://www.baeldung.com/ops/docker-compose-restart-policies) site. Here's how to pass environment variables to a container. You can hide this information by writing these values into a file. You specify the name of this file inside the service.

env\_file: variables.env

The expression **external: tru**e inside the volumes and networks sections forces the docker-compose system to use already created and registered objects.

Application **docker-compose** listens to the same commands as the container: start, stop, ls, rm. All allowed commands will be listed by the help command or the [docker](https://docs.docker.com/compose/reference/) site:

docker compose --help

Run all **docker-compose** commands being in the directory where the **docker-compose.yml** file is written. The Compose file is a YAML file defining **services**, **networks** and **volumes**. The default path for a Compose file is ./docker-compose.yml.

Services

A service definition contains configuration that is applied to each container started for that service, much like passing command-line parameters to docker run. First line of this section defines name of the service. Next row is **build** subsection. It can be either specified as a single string defining a context path, or as a detailed build definition. In the former case, the whole path is used as a Docker context to execute a Docker build, looking for a canonical **Dockerfile** at the root of the directory. The path can be absolute or relative. If it is relative, it must be resolved from the **Compose** file parent folder.

version: "3.8"

services:

webapp:

build: ./dir

Subsection **build** may be specified as an object with the path specified under context and optionally **Dockerfile** and args:

version: "3.8"

services:

webapp:

build:

context: ./dir

dockerfile: Dockerfile-alternate

args:

buildno: 1

It is possible to define **image** inside **services** section.

Networks

The top-level networks element lets you configure named networks that can be reused across multiple services. To use a network across multiple services, you must explicitly grant each service access by using the networks attribute. The networks top-level element has additional syntax that provides more granular control.

services:

frontend:

image: awesome/webapp

networks:

- front-tier

- back-tier

networks:

front-tier:

back-tier:

Default **bridge** network is used in this example.

#### Authentication

You can download an image from [docker hub](https://www.docker.com/) without connecting to the site, but when you try to create your own image, you will be told that there are insufficient rights. The problem is that docker now requires authentication. The Personal plan is free, so connect to the [docker](https://hub.docker.com/) link and create your account if you don't already have one.

The docker system stores login credentials in a hidden file **~/.docker/config.json**. Enable **View/Show hidden files** in the file manager for accessing this file. Your operating system was ready to go by installing **docker desktop**. Follow these instructions if you haven't installed **docker desktop**:

1. Open file ~/.docker/config.json in any text editor and remove line

"credsStore": "desktop"

1. Type this command in the terminal

docker login -u loginName -p password

**loginName** and **password** are the login credentials for docker hub. You can ignore the security warning because the password will be saved in a hidden file **~/.docker/config.json**. The Docker system will use the saved password until you delete it with the **logout** command.

docker logout

This method is safe enough when working with an individual computer. Use the [pass](https://www.passwordstore.org/) program if you want to protect your passwords more:

1. Clear the password store entering command:

rm -fr ~/.password-store

1. Check if pass is installed

pass --version

1. Install this command if the **pass** command is not available on your system:

sudo apt install pass

1. Generate a public/private key pair using **gpg**. Generating a key pair is the same as for connecting to GIT, but use the RSA encryption algorithm and keysize of 4,096 bits:

gpg --full-generate-key

This program stores certificates in a hidden directory, which you can view using the **list-keys** command:

gpg –list-keys

You can also generate keys with **ssh-keygen**, which you used for connection to GIT:

ssh-keygen -t rsa -b 4096 -C "your\_email"

1. Tell the system that you fully trust the first key:

gpg --edit-key "your\_email"

gpg> trust

# Select 5: '5 = I trust ultimately'

gpg> quit

1. Init password store with command:

pass init "your\_email"

1. Initialize docker credential helpers, using pass.

pass insert docker-credential-helpers/docker-pass-initialized-check

1. Verify your installation entering command:

docker-credential-pass list

You should see empty list: {}

1. Log out of docker (emit **docker logout** command) and edit the ~/.docker/config.json file. It should be like this:

{

"credsStore": "pass"

}

1. Login into docker:

docker login -u loginName docker.io

This command will ask for a password. After a successful connection, you will see a completely different ~/.docker/config.json:

{  
 "auths": {  
 "https://index.docker.io/v1/": {}  
 },  
 "credsStore": "pass"  
}

It is clear that an experienced hacker will break even this password protection, but he will have to rummage around your computer much longer.

#### Build image

You will definitely need to produce your own docker **image** when publishing your own software. Suppose you need to display a static or dynamic HTML page. Without going into the programming jungle now, I used the following index.html file:

<!DOCTYPE html>

<html>

<body>

<h1>Hello from docker</h1>

<p>Here is my first page.</p>

</body>

</html>

Of course, you can save this file to some directory, drop the nginx image from the repository, mount the directory with the **-v** option and create a container from the nginx image. An example is given in the official [nginx](https://hub.docker.com/_/nginx) site. This method will work, but will complicate publishing the container: you will need to copy not only the container, but also the mounted directory.

Be sure to read the "[Docker Image Manipulation Basics](https://www.freecodecamp.org/news/the-docker-handbook/" \l "how-to-create-a-docker-image)" in the book about docker. The basics of creating an image are perfectly laid out there. I'll show you how to create an image whose container will point to the index.html file already mentioned.

1. Create a directory dockerImageTest and put index.html there. You can use any text editor, but instead practice working with VS Code.
2. Save the text Dockerfile in the same directory. The content of this file is:

FROM ubuntu:latest

EXPOSE 80

RUN apt update && \

apt install nginx -y && \

apt clean && rm -rf /var/lib/apt/lists/\* && \

rm /var/www/html/\*

COPY ./index.html /var/www/html

CMD ["nginx", "-g", "daemon off;"]

1. Build own image with command:

docker image build --tag custom-nginx:packaged .

1. Create the container and run it:

docker container run --detach --name custom-nginx-packaged \

--publish 8080:80 custom-nginx:packaged

1. Open the page **localhost:8080** in any browser and you will see the information written in the **index.html** file.

The contents of Dockerfile should be clear if you have read [Docker Image Manipulation Basics](https://www.freecodecamp.org/news/the-docker-handbook/" \l "how-to-create-a-docker-image). My example differs from the one explained there only in that I used the **apt** command instead of **apt-get** and changed the contents of the **/var/www/html** directory.

#### Dockerfile structure

A complete list of commands you can use writing a Dockerfile is provided at the [docker](https://docs.docker.com/engine/reference/builder/) site. I will explain only most important them here.

DIRECTIVE

A **directive** command can begin a Dockerfile. Two directives are currently supported: **#syntax** and **#escape**:

1. **syntax** – the directive is only available when using the [BuildKit](https://docs.docker.com/build/buildkit/) backend, and is ignored when using the classic builder backend.
2. **Escape** – the directive sets the character used to escape characters in a Dockerfile. If not specified, the default escape character is "\". The escape character is used both to escape characters in a line, and to escape a newline.

Once a comment, empty line or builder instruction has been processed, Docker no longer looks for parser directives.

ARG

ARG <name>[=<default value>]

The **[ARG](https://docs.docker.com/engine/reference/builder/" \l "arg)** instruction defines a variable that users can pass at build-time to the builder with the docker build command using the **--build-arg <varname>=<value>** flag. A Dockerfile may include one or more **ARG** instructions.

An **ARG** declared before a **FROM** is outside of a build stage, so it can’t be used in any instruction after a **FROM**. To use the default value of an **ARG** declared before the first **FROM** use an **ARG** instruction without a value inside of a build stage:

ARG VERSION=latest

FROM busybox:$VERSION

ARG VERSION

RUN echo $VERSION > image\_version

An **ARG** variable definition comes into effect from the line on which it is defined in the Dockerfile:

FROM busybox

USER ${username:-some\_user}

ARG username

USER $username

# ...

A user builds this file by calling:

docker build --build-arg username=what\_user .

The USER at line 2 evaluates to **some\_user** as the username variable is defined on the subsequent line 3. The USER at line 4 evaluates to **what\_user**, as the username argument is defined and the what\_user value was passed on the command line.

Docker has a set of predefined ARG variables that you can use without a corresponding ARG instruction in the Dockerfile:

HTTP\_PROXY http\_proxy HTTPS\_PROXY https\_proxy

FTP\_PROXY ftp\_proxy NO\_PROXY no\_proxy

ALL\_PROXY all\_proxy

To use these, pass them on the command line using the **--build-arg** flag, for example:

docker build --build-arg HTTPS\_PROXY=https://my-proxy.example.com .

Additional details can be found on the [docker](https://docs.docker.com/engine/reference/builder/" \l "arg) site.

FROM

FROM [--platform=<platform>] <image> [AS <name>]

The **FROM** instruction initializes a new build stage and sets the Base Image for subsequent instructions. **FROM** can appear multiple times within a single Dockerfile. **FROM** instructions support variables that are declared by any **ARG** instructions that occur before the first FROM:

ARG CODE\_VERSION=latest

FROM base:${CODE\_VERSION}

CMD /code/run-app

FROM extras:${CODE\_VERSION}

CMD /code/run-extras

Additional details can be found on the [docker](https://docs.docker.com/engine/reference/builder/" \l "from) site.

MAINTAINER

The **MAINTAINER** instruction sets the Author field of the generated images. It is deprecated now.

LABEL

The **LABEL** instruction is a much more flexible version of this and you should use it instead.

LABEL <key>=<value> <key>=<value> <key>=<value> ...

An image can have more than one **LABEL**. You can specify multiple labels on a single line or multiple lines. In the latter case, all lines except the last must end with a backslash:

LABEL multi.label1="value1" \

multi.label2="value2" \

other="value3"

To view an image’s labels, use the docker image inspect command.

Additional details can be found on the [docker](https://docs.docker.com/engine/reference/builder/" \l "label) site.

RUN

The **RUN** command has 2 forms:

* **RUN <command>** (shell form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows)
* **RUN ["executable", "param1", "param2"]** (exec form)

The given example uses shell form of the RUN command. The argument to the RUN command written in the Exec form is a JSON array, look at the **CMD** command in the example..

A Dockerfile can have multiple RUN commands. Every RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile. For this reason, commands are chained in the shell form, otherwise we will end up with a lot of unnecessary layers. You can find all parameters and options of the **RUN** command on the [docker](https://docs.docker.com/engine/reference/builder/" \l "run) link.

CMD

The **CMD** command doesn’t execute during the build time it will execute after the creation of the container. The main purpose of a CMD is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an **ENTRYPOINT** instruction as well:

* **CMD ["executable","param1","param2"]** (exec form, this is the preferred form)
* **CMD ["param1","param2"]** (as default parameters to ENTRYPOINT)
* **CMD command param1 param2** (shell form)

There can only be one CMD instruction in a Dockerfile. If you list more than one CMD then only the last **CMD** will take effect. If **CMD** is used to provide default arguments for the **ENTRYPOINT** instruction, both the **CMD** and **ENTRYPOINT** instructions should be specified with the JSON array format. Write **CMD** after the **ENTRYPOINT** in this case.

The command specified in the **CMD** instruction can be replaced by the **docker run** or **docker create** commands.

Additional details can be found on the [docker](https://docs.docker.com/engine/reference/builder/" \l "cmd) site.

ENTRYPOINT

**[ENTRYPOINT](https://docs.docker.com/engine/reference/builder/" \l "entrypoint)** has two forms:

1. **ENTRYPOINT ["executable", "param1", "param2"]**
2. **ENTRYPOINT command param1 param2**

The first form is called the **exec form**, the second is called one - the **shell form**. **ENTRYPOINT**, like **CMD**, specifies the command that is executed when the container is started. The most flexible usage of the command is a pair of two commands (**ENTRYPOINT, CMD)**. Both commands would be written in exec form. **ENTRYPOINT** specifies the command to be executed and some parameters, while **CMD** specifies only parameters. Docker runtime joins both arrays and executes command defined in **ENTRYPOINT**. Consider This Dockerfile:

FROM alpine

ENTRYPOINT [ "echo", "Hello" ]

CMD ["world!"]

After running these commands in the terminal

docker build -t test .

docker run --rm test

we will see **Hello world!** in the terminal. Command echo was executed with two parameters. In the **run** command, you can write several parameters after the image name. These parameters are converted into a JSON array that replaces contents of the CMD command:

docker run --rm test Old John

This container will print **Hello Old John!**.

The [shisho.dev](https://shisho.dev/blog/posts/docker-cmd-entrypoint/) site explains in detail an usage of the **ENTRYPOINT** command and possible pitfalls.

USER

The **[USER](https://docs.docker.com/engine/reference/builder/" \l "user)** instruction sets the user name (or UID) and optionally the user group (or GID) to use as the default user and group for the remainder of the current stage. The specified user is used for RUN instructions and at runtime, runs the relevant ENTRYPOINT and CMD commands. User and group can be queried by names or identifiers:

USER <user>[:<group>]

USER <UID>[:<GID>]

See the [Docker](https://docs.docker.com/engine/reference/builder/" \l "user) site for a detailed description of the command.

VOLUME

The **[VOLUME](https://docs.docker.com/engine/reference/builder/" \l "volume)** instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers. Volumes can be declared in your Dockerfile using the **VOLUME** statement. This statement declares that a specific path of the container must be mounted to a Docker volume. Every **docker** **run** command will create new anonymous volume (volume with a unique id as the name) and mount it to the specified path but **docker container start** will use volume created earlier. You can use this feature to save data after the container is stopped.

VOLUME ["/data"]

Argument of the instruction can be a JSON array, **VOLUME ["/var/log/", "/var/db"]**, or a plain string with multiple arguments, such as **VOLUME /var/log** or **VOLUME /var/log /var/db**. The **docker run** command initializes the newly created volume with any data that exists at the specified location within the base image:

FROM ubuntu

RUN mkdir /myvol

RUN echo "hello world" > /myvol/greeting

VOLUME /myvol

Command **docker run** will create a new mount point at /myvol and copy the greeting file into the newly created volume.

You can create an external named volume and mount it using the **-v** option. An anonymous volume will not be created in this case.

[Howtogeek](https://www.howtogeek.com/devops/understanding-the-dockerfile-volume-instruction/) has a detailed description of this command and as always you can refer to the [docker](https://docs.docker.com/engine/reference/builder/" \l "volume) link.

WORKDIR

The **[WORKDIR](https://docs.docker.com/engine/reference/builder/" \l "workdir)** instruction sets the working directory for any **RUN**, **CMD**, **ENTRYPOINT**, **COPY** and **ADD** instructions that follow it in the Dockerfile. If the WORKDIR doesn’t exist, it will be created. The **WORKDIR** instruction can be used multiple times in a Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction.

WORKDIR /a

WORKDIR b

WORKDIR c

RUN pwd

The output of the final pwd command in this Dockerfile would be /a/b/c.

ONBUILD

ONBUILD <INSTRUCTION>

The **[ONBUILD](https://docs.docker.com/engine/reference/builder/" \l "onbuild)** instruction adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build (child image). The trigger will be executed in the context of the downstream build, as if it had been inserted immediately after the FROM instruction. A **Dockerfile** can contain several **ONBUILD** instructions. ONBUILD instructions are not inherited by “grand-children” builds.

**ONBUILD** allows you to create base files and use them in more complex files, much like the base classes are used in objective programming languages (C++, Java, C#, Delphi, …). Consider file **my-base-image**

FROM openjdk:16-alpine3.13

WORKDIR /app

ONBUILD COPY ./setup.sh .

ONBUILD RUN ./setup.sh

ONBUILD COPY src ./src

After building this image with command

docker build -t my-base-image

one can use it as creating child image (file **my-child-image**)

FROM my-base-image:latest

WORKDIR /app

CMD ["./mvnw", "spring-boot:run"]

Now run this command:

docker build -t my-child-image

and commands

COPY ./setup.sh .

RUN ./setup.sh

COPY src ./src

will be executed just after FROM.

STOPSIGNAL

STOPSIGNAL signal

When you run docker stop, you are instructing the Docker daemon to send a signal to the process running the container to stop. By default, it does this by sending a **SIGTERM** and then wait a short period so the process can exit gracefully. If the process does not terminate within a grace period (10s by default, customisable), it will send a **SIGKILL**. However, your application may be configured to listen to a different signal - **SIGUSR1** and **SIGUSR2**, for example. In these instances, you can use the **STOPSIGNAL** Dockerfile instruction to override the default.

The **[STOPSIGNAL](https://docs.docker.com/engine/reference/builder/" \l "stopsignal)** instruction sets the system call signal that will be sent to the container to exit. This signal can be a signal name in the format SIG<NAME>, for instance **SIGKILL**, or an unsigned number that matches a position in the kernel’s syscall table, for instance 9.

The image’s default stopsignal can be overridden per container, using the **--stop-signal** flag on **docker run** and **docker create**.

HEALTHCHECK

The [HEALTHCHECK](https://docs.docker.com/engine/reference/builder/" \l "healthcheck) instruction has two forms:

* **HEALTHCHECK [OPTIONS] CMD command** (check container health by running a command inside the container)
* **HEALTHCHECK NONE** (disable any healthcheck inherited from the base image)

The **HEALTHCHECK** instruction tells Docker how to test a container for check that it is still working. This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, even though the server process is still running.

[Ganesh Mani](https://scoutapm.com/blog/how-to-use-docker-healthcheck) wrote a great article about the **HEALTHCHECK** command and how it may be used inside docker containers. **HEALTHCHECK** may also be defined inside **docker-compose** files. Article "[Health Check Command in Docker](https://www.atatus.com/blog/health-check-command-in-docker/)" explains this.

SHELL

The **[SHELL](https://docs.docker.com/engine/reference/builder/" \l "shell)** instruction allows the default shell used for the shell form of commands to be overridden. The default shell on Linux is ["/bin/sh", "-c"], and on Windows is ["cmd", "/S", "/C"].

The **SHELL** instruction must be written in JSON form in a Dockerfile:

SHELL ["executable", "parameters"]

The **SHELL** instruction can appear multiple times. Each **SHELL** instruction overrides all previous **SHELL** instructions, and affects all subsequent instructions.

HERE-DOCUMENTS

[Redirection](https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html" \l "tag_18_07_04) operator is feature of Linux shell thus follow rules of Linux shell writing redirections. The redirection operators "**<<**" and "**<<-**" both allow redirection of subsequent lines read by the shell to the input of a command. The redirected lines are known as a "**here-document**".

**[Here-documents](https://docs.docker.com/engine/reference/builder/" \l "here-documents)** allow redirection of subsequent Dockerfile lines to the input of **RUN** or **COPY** commands. If such command contains a **here-document** the Dockerfile considers the next lines until the line only containing a here-doc delimiter as part of the same command:

# syntax=docker/dockerfile:1

FROM debian

RUN <<EOT bash

set -ex

apt-get update

apt-get install -y vim

EOT

If the command only contains a here-document, its contents is evaluated with the default shell. Shebang header can be used to define an interpreter:

# syntax=docker/dockerfile:1

FROM python:3.6

RUN <<EOT

#!/usr/bin/env python

print("hello world")

EOT

ONBUILD

* + - 1. The **[ONBUILD](https://docs.docker.com/engine/reference/builder/" \l "onbuild)** instruction adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build.
      2. ONBUILD <INSTRUCTION>
      3. The trigger will be executed in the context of the downstream build, as if it had been inserted immediately after the FROM instruction in the downstream Dockerfile. Any build instruction can be registered as a trigger. Here is an example of this instruction:
      4. ONBUILD ADD . /app/src
      5. ONBUILD RUN /usr/local/bin/python-build --dir /app/src

Remember:

1. When docker encounters an **ONBUILD** instruction, the builder adds a trigger to the metadata of the image being built. The instruction does not otherwise affect the current build.
2. At the end of the build, a list of all triggers is stored in the image manifest, under the key OnBuild. They can be inspected with the **docker inspect** command.
3. Later the image may be used as a base for a new build, using the FROM instruction. As part of processing the **FROM** instruction, the downstream builder looks for **ONBUILD** triggers, and executes them in the same order they were registered. If any of the triggers fail, the **FROM** instruction is aborted which in turn causes the build to fail. If all triggers succeed, the **FROM** instruction completes and the build continues as usual.
4. Triggers are cleared from the final image after being executed. In other words they are not inherited by “grand-children” builds.

## Text editor

You will definitely need to edit text files during your work. By default, after installing the Ubuntu system, you’ll get:

1. **Mousepad** – a small and fairly clear text editor suitable for small corrections,
2. **Visual Studio Code** - a basic programmer's tool that we'll explore later,
3. **Libre Office** - a suite of programs that perform the same tasks as Microsoft's MS Office. The only problem is that it is written in Java and is significantly slower than Microsoft's product.

When working remotely, you will need an editor that allows you to edit files on the remote computer. **Nano** and **vim** do that best. Both editors take up little space and work in a terminal window, so you can easily make the necessary corrections by connecting to a remote computer via **ssh**.

#### Nano

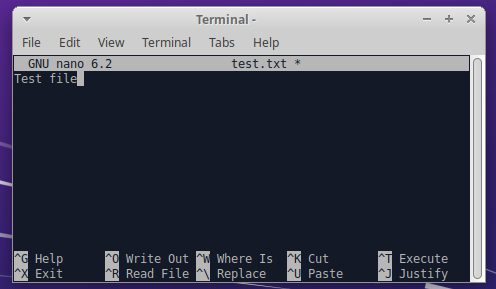
GNU nano is an easy to use command line text editor for Unix and Linux operating systems. It includes all the basic functionality you’d expect from a regular text editor, like syntax highlighting, multiple buffers, search and replace with regular expression support, spellchecking, UTF-8 encoding, and more. Nano text editor is pre-installed on **macOS** and most **Linux** distros. To check if it is installed on your system type:

nano --version

Install it if this command found nothing:

sudo apt install nano

The editor window listens for cursor control keys. Press CTL+G to see a short tutorial. Control keys that are a bit unusual for Windows users are listed in the lower lines of the window.



Users of other text editors will start working with **nano** in a few minutes, but it must be said that this editor is not a tool for professional programming. Small instruction you’ll find on [linuxize](https://linuxize.com/post/how-to-use-nano-text-editor/) site.

#### Vim

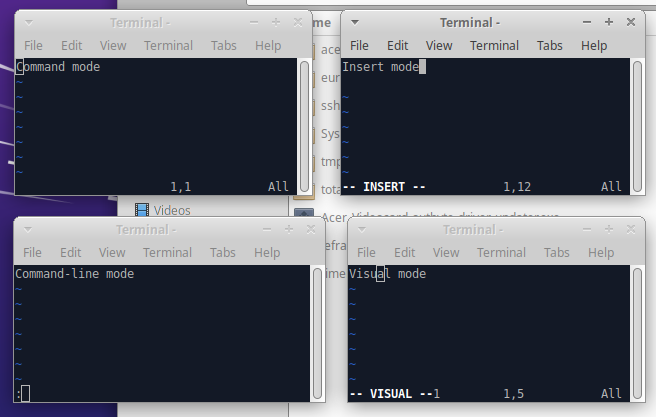
**Vim** is a modified version of **VI** and has been faithfully serving programmers for 40 years. It is a mandatory tool for network and database administrators. Install this editor with apt command:

sudo apt install vim

The editor works in four different modes:

1. Command mode - VIM starts in this mode. Being in this mode you can move across the screen, delete text, copy text and switch to any other mode. Press ESC key for switching to command mode.
2. Insert mode – this mode allows writing the text. You can switch to this mode from command mode. Just press the **i** key.
3. Command-line mode - in this mode, VIM performs various text manipulations, executes external commands, and completes the work of the editor. From command mode, you can go to the command-line by pressing one of the following keys **/ ? : !** .
4. Visual mode - this mode makes it easier to highlight and manipulate text in Vim. You can switch to this mode from command mode. Just press keys **v,** **V** or **CTRL+v** .

Bottom line of the screen indicates a current mode.



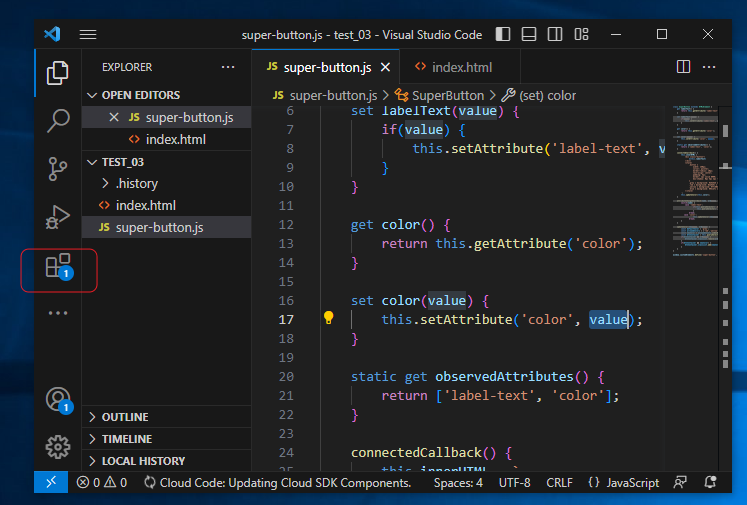
A more or less complete description of the vim editor is provided at the [sci.notbc.org site](./A%20more%20or%20less%20complete%20description%20of%20the%20editor%20is%20provided%20at%20the%20sci.notbc.org%20site.%20Reading%20this%20book%20will%20take%20a%20time%20thus%20start%20with%20smaller%20one.). Reading this book will take a time thus start with smaller [one](https://www.vim-hero.com/). Don't spend a lot of time learning VIM: many of the things it does can be done better and much easier with **Visual Studio Code**.

#### Visual Studio Code

Visual Studio Code (VS Code) was produced as a simplified version of Visual Studio when Microsoft decided to extend .NET to Linux and MAC OS systems. VS Code has evolved very rapidly and is now one of the best free tools for programmers. It works on Windows, Linux and macOS.

In the 2016 Developers Survey of Stack Overflow, Visual Studio Code ranked No. 13 among the top popular development tools, with only 7% of the 47,000 respondents using it. Two years later, however, Visual Studio Code achieved the No. 1 spot, with 35% of the 75,000 respondents using it. In the 2019 Developers Survey, Visual Studio Code was also ranked No. 1, with 50% of the 87,000 respondents using it. In the 2021 Developers Survey, Visual Studio Code continued to be ranked No. 1, with 74.5% of the 71,000 respondents using it, and 74.48% of the 71,010 responses in the 2022 survey ([Wikipedia](https://en.wikipedia.org/wiki/Visual_Studio_Code)).

VS Code has different extensions for different languages. You can download the extension by clicking the corresponding button:



VS Code does not have specialized project files. For this tool, the project is a tree of the directories. in the top directory of which you opened VS Code. Always open the root directory of the tree when working with VS Code. Use "Open Recent" for subsequent openings of the project.

Visual Studio Code project may be linked to any supported version control system (Git, Apache Subversion, Perforce, ...). Therefore, start new projects according to this instruction:

1. Create a new project on GitHub or GitLab,
2. Clone the project to your local disk,
3. Open terminal and go to the root directory (directory where the **.git** hidden directory is created).
4. Create empty file with correct extension (.go, .cs, .js, .ts, …)

touch myFile.go // Linux

type nul > myFile.go // Windows

1. Open the directory in VS code

code . &

VS Code will offer you the extensions you need and you can manage GIT directly from VS Code. After pressing CTRL+Shift+P, you will see a window with all available commands. Over time, you will remember the most frequently used combinations, now just select the required command from the menu. [Alessandro Del Sole](https://www.syncfusion.com/succinctly-free-ebooks/visual-studio-code-succinctly) published free book on WEB. Read it.

Settings

You can configure Visual Studio Code to your liking through its various settings. Nearly every part of VS Code's editor, user interface, and functional behavior has options you can modify. To modify user settings, you'll use the **Settings** editor to review and change VS Code settings. To open the Settings editor, use the following VS Code menu command: **File > Preferences > Settings**. Home site of [VS Code](https://code.visualstudio.com/docs/getstarted/settings) describes all available modifications.