**What is Dockerfile?**

Before learning about Dockerfiles — what is Docker?

Docker is an open-source platform for building, deploying, and managing containerized applications. It has become the de facto standard to build and share apps, from desktop to cloud, including edge devices like Raspberry Pi.

One of the features that have made Docker so popular among developers is the ability to easily pack, ship, and run applications as lightweight, portable, and self-sufficient containers that can run virtually anywhere.

The instructions to build a Docker container image are stored in files called Dockerfile. This is an example of a typical Dockerfile :



A Dockerfile is a text document that contains all the commands a user can call on the command line to assemble an image.

# Why Dockerfile security?

If you perform a quick search for Dockerfiles in GitHub, you can see that it returns more than 3 million files.

* Dockerfiles are a blueprint for building your Docker container images
* Dockerfiles are a codified version of your application and infrastructure
* Dockerfiles are among the key components in the entire supply chain security
* Dockerfiles need to be part of your security posture to maintain the highest level of security comprehensively
* Insecure Dockerfiles can cause serious security issues

# Best practices to write Dockerfiles

Here is a collection of standard best practices that Docker recommend in their [Documentation](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/), as well as community-driven best practices:

* Start with a small version of the image
* Create ephemeral containers
* Understand the build context
* Exclude files from the image with **.dockerignore** — it works similarly to **.gitignore** in Git
* Use multi-stage builds to reduce the image size and its attack surface
* Create multi-line arguments in a structured way, and reduce the image layers
* Minimize the number of layers
* Leverage the build cache
* Create your own base image like a golden image

The Docker community recommends a number of other best practices when creating Dockerfiles**.** For example:

* Order the steps in the Dockerfile from least to most frequently changing content
* Use the COPY instruction to copy only the necessary files. Avoid executing instructions such as COPY . .
* Only install what you need. For example, use the --no-install-recommendsoption
* Group similar commands. For example: RUN apt-get update && apt-get install -y curl
* Remove the package manager cache: rm -rf /var/lib/apt/lists/\*
* Use a specific image tag; avoid the latest tag
* Set non-root user and group
* Disallow acquiring new privileges
* Use only trusted and official base images
* Don’t store secrets or sensitive information in Dockerfiles
* Don’t install SSH or similar services that may expose your containers
* Apply image lifecycle management updates, if required

So far we explored a number of standard best practices to follow. Now let’s see how we can apply them in practice in our DevOps workflow

**BuildKit — SSH Socket (security use case)**

Organizations hosting their code on private version control systems and running Docker builds in CI/CD pipelines may sometimes use workarounds to pass the SSH authentication credentials to have SSH access to the container build. In the example below, the SSH key is copied to the Dockerfile in an insecure manner.

BuildKit enables passing the SSH socket by mounting it. This forwards the SSH agent from the host in a secure manner.

## hadolint — Haskell Dockerfile Linter

A smarter Dockerfile linter that helps you build best practice Docker images. The linter parses the Dockerfile to an AST, and then it runs rules on top of it.

hadolint is inspired by ShellCheck, which lints Bash code inside RUN instructions.