Dockerfile Cheat Sheet

A comprehensive cheat sheet for Dockerfiles, covering essential commands, syntax, and best practices for building efficient Docker images. Includes examples and explanations for common use cases.

Dockerfile Basics

Base Image Instruction

`FROM`	Sets the base image for subsequent instructions. It's the foundation of your image. **Example:** `FROM ubuntu:latest`
Syntax	`FROM <image>[:<tag>] [AS <name>]` `<image>`: The name of the image (e.g., ubuntu, node). `<tag>`: (Optional) A specific version or label (e.g., 16.04, latest). `<name>`: (Optional) Assigns an alias if using multi-stage builds.
Usage	Always start with a `FROM` instruction. Use specific tags for reproducibility.

Multi-stage Builds	Use `AS` to name a stage and reference it later.
	
	Example:
	`FROM maven:3.6.3-jdk-11 AS builder`
	`WORKDIR /app`
	`COPY pom.xml .`
	`COPY src ./`
	`RUN mvn clean install`
	
	`FROM openjdk:11-jre-slim`
	`COPY --from=builder /app/target/my-app.jar app.jar`
	`ENTRYPOINT ["java", "-jar", "app.jar"]`

Metadata Instructions

`LABEL`	Adds metadata to the image in key-value pairs.
	
	Example:
	`LABEL maintainer="john.doe@example.com"`
	`LABEL version="1.0.1"`
	`LABEL description="A simple web application image"`

Syntax	`LABEL <key>=<value> <key>=<value> ...` Keys and values should be properly quoted if they contain spaces.
Best Practices	Use reverse DNS notation for keys to avoid conflicts (e.g., `com.example.version`). Combine multiple labels in a single instruction for efficiency.

Multiline Labels	Labels can span multiple lines using backslashes.
	
	Example:
	`LABEL description="This text illustrates \`
	`that label-values can span multiple lines."`

Environment Variables

`ENV`	Sets environment variables inside the container.
	
	Example:
	`ENV APP_HOME /app`
	`ENV PORT 8080`

Syntax	`ENV <key> <value>` or `ENV <key>=<value>` The first form allows setting multiple variables at once. The second is more readable for single variables.

Variable Usage	Environment variables can be used in other instructions.
	
	Example:
	`ENV APP_HOME /app`
	`WORKDIR $APP_HOME`

Best Practices	Use `ENV` to define variables that should be configurable at runtime.

File Management and Execution

File Operations

`COPY`	Copies files or directories from the host to the container. **Example:** `COPY ./app /app`
Syntax	`COPY [--chown=<user>:<group>] <src>... <dest>` `--chown` : (Optional) Changes the ownership of the copied files. `<src>` : Source file or directory on the host. `<dest>` : Destination path inside the container.

`ADD`	Similar to `COPY`, but also supports extracting compressed files and fetching remote URLs.
	
	Example:
	`ADD ./app.tar.gz /app/`
	`ADD https://example.com/app.zip /app/`

Best Practices	Prefer `COPY` over `ADD` unless you need the extra features of `ADD`. This makes the build more predictable.

Execution Instructions

`RUN`	Executes commands inside the container during the build process. **Example:** `RUN apt-get update && apt-get install -y --no-install-recommends nginx`
Syntax	`RUN <command>` (shell form) or `RUN ["executable", "param1", "param2"]` (exec form) The exec form avoids shell interpretation and is generally preferred.
`CMD`	Specifies the command to run when the container starts. Can be overridden by `docker run` arguments. **Example:** `CMD ["nginx", "-g", "daemon off;"]`
Entrypoint	Configures a container that will run as an executable. **Example:** `ENTRYPOINT ["executable", "param1", "param2"]`

Directory Management

`WORKDIR`	Sets the working directory for subsequent instructions.
	
	Example:
	`WORKDIR /app`
	`RUN echo "Hello" > file.txt  # Creates /app/file.txt`

Syntax	`WORKDIR <path>` If the directory doesn't exist, it will be created.
`VOLUME`	Creates a mount point for accessing and storing data outside the container's file system. **Example:** `VOLUME ["/data"]`
Important Notes	Changes to the volume are not included when updating the image. Volumes are designed for persistent storage.

Networking and Build Arguments

Networking Instruction

`EXPOSE`	Declares the ports that the container will listen on at runtime. It's informative but doesn't actually publish the port.
	
	Example:
	`EXPOSE 80`
	`EXPOSE 443`

Syntax	`EXPOSE <port>[/<protocol>] [<port>[/<protocol>]...]` `<protocol>` : Can be `tcp` (default) or `udp`.
Publishing Ports	To actually publish the ports, use the `-p` or `-P` flags with `docker run`.

Build Arguments

`ARG`	Defines variables that can be passed during the build process using the `--build-arg` flag.
	
	Example:
	`ARG version`
	`RUN echo "App Version: $version"`

Syntax	`ARG <name>[=<default value>]` Arguments can have default values.
Usage	Build arguments are useful for passing sensitive information or configuration values at build time. **Example (using docker build):** `docker build --build-arg version=1.2.3 .`
Scope	Arguments are only available during the build process and are not stored in the final image unless explicitly set as environment variables.

User Instruction

`USER`	Sets the user to use when running subsequent `RUN`, `CMD`, and `ENTRYPOINT` instructions. **Example:** `USER nginx`
Syntax	`USER <username>[:<group>]` or `USER <UID>[:<GID>]` Can be a username or a numeric UID.
Best Practices	Avoid running processes as `root` for security reasons. Create a dedicated user and group for your application.

Advanced Dockerfile Concepts

Healthcheck Instruction

`HEALTHCHECK`	Configures a health check command to determine if a container is healthy.
	
	Example:
	
	`HEALTHCHECK --interval=5m --timeout=3s \`
	`CMD curl -f http://localhost/ || exit 1`

Syntax	`HEALTHCHECK [OPTIONS] CMD <command>` (exec form) or `HEALTHCHECK NONE` (disable healthcheck) Options: • `--interval=<duration>`: Time between checks (default: 30s). • `--timeout=<duration>`: Time to wait before considering the check a failure (default: 30s). • `--start-period=<duration>`: Initial startup time to allow the container to initialize (default: 0s). • `--retries=<number>`: Number of consecutive failures needed to consider the container unhealthy (default: 3).
Return Codes	`0`: healthy, `1`: unhealthy, `2`: reserved (don't use).

Onbuild Instruction

`ONBUILD`	Defers the execution of a command until the image is used as the base for another build. It is triggered when a downstream image is built. **Example:** `ONBUILD RUN echo "Running onbuild..."`
Syntax	`ONBUILD <INSTRUCTION>` Any valid Dockerfile instruction can be used.
Use Cases	Useful for creating reusable base images that perform common tasks, such as installing dependencies or setting up configurations in derived images.
Important Considerations	Avoid using `ONBUILD` instructions that depend on specific paths or configurations in the derived image. It can lead to unexpected behavior if not used carefully.

Shell Instruction

`SHELL`	Overrides the default shell used for the shell form of the `RUN`, `CMD`, and `ENTRYPOINT` instructions.
	
	Example:
	
	`SHELL ["/bin/bash", "-c"]`
	`RUN echo "Hello, world!"`

Syntax	`SHELL ["executable", "param1", "param2"]` Specifies the executable to use as the shell. Defaults to `["/bin/sh", "-c"]` on Linux or `["cmd", "/S", "/C"]` on Windows.

Strict Mode Example	Run commands in strict shell.
	
	`ENV my_var`
	`SHELL ["/bin/bash", "-euo", "pipefail", "-c"]`
	
	`# With strict mode:`
	`RUN false # fails build like using &&`
	`RUN echo "$myvar" # will throw error due to typo`
	`RUN true | false # will bail out of pipe`