













Collabnix

Docker Cheatsheet [2023 Updated]

A cheat-sheet is a concise summary of important information that is meant to be used as a quick reference. Cheat-sheets are often used in the form of a list or a table, and they typically cover a specific topic or subject area. In the context of Docker, a Docker cheat-sheet is a summary of commonly used Docker commands and their options, as well as other useful information related to Docker.

Cheat-sheets can be particularly helpful when learning a new tool or technology, as they provide a convenient way to quickly look up and remind oneself of key concepts and commands. They can also be useful for experienced users who need to recall a specific command or option but may not remember all the details.

Table of Contents

- Categories
 -  Basic Docker CLIs
 -  Container Management CLIs
 -  Inspecting the Container
 -  Interacting with Container
 -  Image Management Commands
 -  Image Transfer Commands
 -  Builder Main Commands
 -  The Docker CLI
 -  Docker Security (Scout, SBOM)
-  Contributors
-  Support and Community
-  References

Basic Docker CLIs

Command	Description
<code>docker run</code>	Run a command in a new container

Command	Description
<code>docker build</code>	Build an image from a Dockerfile
<code>docker push</code>	Push an image to a registry
<code>docker pull</code>	Pull an image from a registry
<code>docker stop</code>	Stop a running container
<code>docker rm</code>	Remove one or more containers

Basic Docker CLIs

Container Management CLIs

Command	Description
<code>docker create image[command]</code> <code>docker run image[command]</code>	create the container = create + start
<code>docker start container...</code> <code>docker stop container...</code> <code>docker kill container...</code> <code>docker restart container...</code>	start the container graceful ² stop kill (SIGKILL) the container = stop + start
<code>docker pause container...</code> <code>docker unpause container...</code>	Suspend the container Resume the container
<code>docker wait container...</code>	Block until one or more containers stop, then print their exit codes
<code>docker top container...</code>	Display the running processes of a container
<code>docker rm[-f³] container...</code>	destroy the container

Container Management CLIs

²send SIGTERM to the main process + SIGKILL 10 seconds later

³`-f` allows removing running containers (=docker kill + docker rm)

Let's break down each of these commands and provide examples for better understanding.

1. Create a Container from an Image using `docker run` :

Before you can execute commands inside a container, you typically need to create a container from an image. This is usually done using the `docker run` command. Here's a basic example:

```
docker run -d --name my_container_name my_image:tag
```

Replace `my_container_name` with the desired name for your container, `my_image:tag` with the name and tag of the Docker image you want to use.

2. Execute Commands in a Running Container using `docker exec` :

You can execute commands in a running container using the `docker exec` command. The syntax is as follows:

```
docker exec [options] CONTAINER COMMAND
```

Here's an example:

```
docker exec -it my_container_name ls /app
```

This would execute the `ls /app` command inside the container named `my_container_name`.

3. Start a Stopped Container using `docker start` :

If you have a stopped container, you can start it again using the `docker start` command. Here's an example:

```
docker start my_container_name
```

This will start the container named `my_container_name`.

4. Stop a Running Container using `docker stop` :

You can stop a running container using the `docker stop` command. Here's an example:

```
docker stop my_container_name
```

This will gracefully stop the container named `my_container_name`.

Remember that you need to replace `my_container_name` with the actual name of your container. Additionally, you can add flags like `-i` (interactive) and `-t` (tty) to the `docker exec` command to make the interaction more user-friendly, especially when running commands that require user input.

For more advanced usage and options, you can refer to the official Docker documentation:

- [Docker run](#)
- [Docker exec](#)
- [Docker start](#)
- [Docker stop](#)

Inspecting The Container

Here's the list of the basic Docker commands that helps you inspect the containers seamlessly:

Command	Description
<code>docker ps</code>	list running container
<code>docker ps -a</code>	list all containers
<code>docker logs [-f⁶] container</code>	Show the container output(<i>stdout+stderr</i>)
<code>docker diff container</code>	Show the differences with the image (modified files)
<code>docker top container...</code>	Display the running processes of a container
<code>docker inspect --format '{{.Config.Image }}' my_container_name..</code>	Inspect a container and extract specific information using <code>--format</code> :

Container Management CLIs

Interacting with Container

Do you want to know how to access the containers? Check out this fundamental command below

Command	Description
<code>docker attach container</code>	attach to a running container(stdin/stdout/stderr)
<code>docker cp container:path hostpath/</code>	copy files from the container
<code>docker cp hostpath/ - container: path</code>	copy files into the container
<code>docker export container</code>	export the content of the container (<i>tar archive</i>)
<code>docker exec container args ...</code>	run a command in an existing container (useful for debugging)
<code>docker wait container...</code>	wait until the container terminates and return the exit code
<code>docker commit container image</code>	commit a new docker image (snapshot of the image)

Interacting with the container

Image Management Commands

Docker provides a set of commands to manage Docker images efficiently. Here is a list of some important Docker image management commands:

Command	Description
<code>docker images</code>	list all local images
<code>docker history image</code>	show the image history(list of ancestors)
<code>docker inspect image...</code>	show level-info's(in json format)
<code>docker tag image tag</code>	<i>tag an image</i>
<code>docker commit container image</code>	Create an image(from a container)
<code>docker import url/ -[tag]</code>	Create an image(from a tarball)
<code>docker rmi image.. .</code>	delete images

Image Management Commands

Image Transfer Commands

To transfer Docker images between different environments or systems, you have a few options. Here are some commonly used methods to transfer Docker images:

Using the registry API	Description
<code>docker push repo[:tag]...</code>	push an image /repository from a registry
<code>docker pull repo[:tag]..</code>	pull an image /repository from a registry
<code>docker search text</code>	search an image on the official registry
<code>docker login ...</code>	login to a registry
<code>docker logout ...</code>	logout from a registry
<code>docker export <container_id> > image.tar</code>	exporting: You can export a Docker image as a tarball file using the <code>docker export</code> command
<code>docker import image image.tar <image_name>:<tag></code>	importing: On the target system, you can import the Docker image from the tarball using the <code>docker import</code> command

Image Transfer Commands

Manual Transfer	Description
<code>docker save repo/ [:tag]...</code>	export an image/repo as a tarball
<code>docker load</code>	load images from a tarball
<code>docker-ssh¹⁰</code>	proposed script to transfer images between two daemons over ssh

Contd'

Builder Main Commands

Want to know how to build Docker Image? Do check out the list of image commands below:

Command	Description
<code>FROM image/scratch</code>	base image for the build
<code>MAINTAINER email</code>	name of the maintainer(metadata).
<code>COPY path dst</code>	copy path from the context into the container at location destination(dst)
<code>ADD src dst</code>	same as <code>COPY</code> but untar archives and accepts https urls
<code>RUN args..</code>	run an arbitrary command inside the container
<code>USER name</code>	set the default username
<code>WORKDIR path</code>	set the default working directory
<code>CMD args..</code>	set the default command

Command	Description
<code>ENV name value</code>	set an environment variable

Builder Main Commands

In Docker, the “builder pattern” is a technique used to create optimized and efficient Docker images by separating the build environment from the runtime environment. It involves using a multi-stage build process, where one stage is used for building the application, and another stage is used to package the application for runtime. This pattern helps to reduce the size of the final Docker image and ensures that only the necessary runtime dependencies are included.

Here are the main commands used in the builder pattern in Docker:

1. **FROM:** The **FROM** instruction specifies the base image for the build stage. It sets up the build environment and contains all the tools and dependencies needed for building the application.

```
FROM base_image_for_building AS build_stage
```

2. **RUN:** The **RUN** instruction executes commands inside the build stage container. It is used to install dependencies, compile code, or perform any other build-related tasks.

```
RUN apt-get update && apt-get install -y build-essential
RUN npm install
RUN go build -o my_app
```

3. **COPY/ADD:** The **COPY** or **ADD** instruction is used to copy files from the local directory (context) into the build stage container.

```
COPY . /app
```

4. **WORKDIR:** The **WORKDIR** instruction sets the working directory inside the build stage container where subsequent commands will be executed.

```
WORKDIR /app
```

5. **FROM (second stage)**: After the build stage is complete, you use another **FROM** instruction to start a new stage for the runtime environment. This new stage starts from a smaller base image, typically one containing only the necessary runtime dependencies.

```
FROM base_image_for_runtime AS runtime_stage
```

6. **COPY --from**: The **COPY --from** instruction copies files from the build stage container to the runtime stage container.

```
COPY --from=build_stage /app/my_app /app/my_app
```

7. **CMD/ENTRYPOINT**: The **CMD** or **ENTRYPOINT** instruction specifies the default command or entry point to run when the container starts. It is used in the runtime stage to define the application that should be executed when the container runs.

```
CMD ["/app/my_app"]
```

Here's a complete example of a multi-stage Dockerfile using the builder pattern:

```
# Build stage
FROM golang:1.16 AS build_stage
WORKDIR /app
COPY . .
RUN go build -o my_app

# Runtime stage
FROM alpine:latest AS runtime_stage
WORKDIR /app
COPY --from=build_stage /app/my_app /app/my_app
CMD ["/app/my_app"]
```


In this example, the first stage uses the Go programming language image as the base image for building the application, compiles the Go code, and generates the `my_app` executable. In the second stage, the Alpine Linux image is used as the base image for the runtime, and the `my_app` binary is copied from the first stage. The final image will only include the compiled `my_app` binary and not the entire build environment, resulting in a smaller and more optimized image for runtime.

The Docker CLI

A brief description of some essential Docker CLI commands related to image management and container operations are seen below. Below, I've expanded on the descriptions of each command:

1. Docker build:

```
docker build [options] -t "app/container_name" .
```

This command is used to create a Docker image from a Dockerfile, which is a text file containing instructions to define the image. The `-t` option is used to specify a tag or name for the built image. The `.` at the end of the command refers to the build context, i.e., the current directory where the Dockerfile is located.

2. Docker run:

```
docker run [options] IMAGE
```

This command is used to run a command or application in a Docker container based on the specified IMAGE. It creates a new container from the image and executes the default command specified in the image, or any command provided as arguments to the `docker run` command. This is one of the primary ways to start a container.

3. Docker create:

```
docker create [options] IMAGE
```

`docker create` command creates a new container from the specified IMAGE, but it does not start the container. It prepares the container to be started using the `docker start` command. You can use this command to create a container and then start it later.

4. Manage images:

Docker provides several commands to manage images:

- `docker images` : Lists all Docker images available on your system.
- `docker pull` Downloads Docker images from a registry.
- `docker push` Pushes Docker images to a registry (e.g., Docker Hub) for sharing with others .
- `docker rmi` Removes Docker images from your system .

4. Manage containers:

Docker provides several commands to manage containers:

- `docker ps` : Lists all running containers.
- `docker ps -a` Lists all containers (including stopped ones).
- `docker start` Starts one or more stopped containers .
- `docker stop` Stops one or more running containers.
- `docker restart` : Stops and then starts a container .
- `docker rm` Removes one or more containers (use with caution).

These commands are just the basics of working with Docker images and containers. Docker provides a rich set of commands and options to interact with containers, networks, volumes, and more. You can always explore the official Docker documentation for more in-depth information on each command and its options.

To run a command in a Docker container using the `docker create` command, you first need to create a container from an image. The `docker create` command creates a container but does not start it. It returns a container ID that you can use to manage the container.

Here's the basic syntax of the `docker create` command:

```
docker create [options] IMAGE [COMMAND] [ARG...]
```

Let's break down the options:

- `-a , --attach` : Attach STDOUT/STDERR and forward signals.
- `-i , --interactive` : Keep STDIN open even if not attached.
- `--name NAME` : Assign a name to the container.
- `-p , --publish 5000: 5000` : Publish a container's port to the host. In this example, port 5000 from the container will be mapped to port 5000 on the host.
- `--expose 5432` : Expose a port internally to linked containers. This does not publish the port to the host machine.
- `-P , --publish-all` : Publish all exposed ports to the host.
- `--link container: alias` : Link the container to another container with an optional alias.
- `-v , --volume 'pwd:/app` : Mount a host directory into the container. In this example, the current working directory is mounted to the `/app` directory in the container.
- `-e , --env NAME=hello` : Set an environment variable inside the container.

Here's an example of using the `docker create` command:

```
docker create -a -i --name my_container -p 5000:5000 --expose 5432 -P --link db_container:db_alias -v "$(pwd):/app" -e NAME=hello my_image
```

- `-a -i` : Attach STDOUT/STDERR and keep STDIN open for an interactive session .
- `--name my_container` : Assign the name "my_container" to the created container.
- `-p 5000:5000` : Publish port 5000 from the container to port 5000 on the host.
- `--expose 5432` : Expose port 5432 internally for linking to other containers .
- `-P` : Publish all exposed ports to the host .
- `--link db_container : db_alias` : Link the container to another container named "db_container" with the alias "db_alias".
- `-v "$(pwd):/app"` : Mount the current working directory on the host into the `/app` directory in the container.
- `-e NAME=hello` : Set an environment variable "NAME" with the value "hello" inside the container.

- `my_image` : The name of the Docker image from which to create the container.

Please note that some of the options like `--link` are deprecated in recent versions of Docker, and it's recommended to use user-defined networks for container communication instead.:

Variable Name	Value
-----	-----
VAR1	value1
VAR2	value2
VAR3	value3

Replace `VAR1` , `VAR2` , and `VAR3` with the actual names of your environment variables, and `value1` , `value2` , and `value3` with their respective values.

Remember, to execute the `docker create` command, you must have Docker installed and running on your system.

Docker Compose

Basic Example

```
version: '2'
services:
  web:
    build:
      context: ./Path
      dockerfile: Dockerfile
    ports:
      - "5000:5000"
    volumes:
      - ./code

  redis:
    image: redis
```

In this configuration:

- The `web` service is defined, which will build an image using the Dockerfile located in the `./Path` directory. It exposes port 5000 from the container to port 5000 on the host and mounts the current directory (`.`) to `/code` inside the container.
- `redis` service is defined, which uses the official Redis image from Docker Hub.

Please ensure that you replace `./Path` with the actual path to your Dockerfile and update any other placeholders with your specific project details. Make sure the indentation is correct as YAML is indentation-sensitive.

References

1. Building Images:

To build images using Docker Compose, you can specify the `build` section within a service definition. You can build from a Dockerfile in a directory (`context`) and specify a custom Dockerfile if needed. Alternatively, you can directly use an existing image.

```
version: '3'
services:
  web:
    build:
      context: ./dir
      dockerfile: Dockerfile.dev
      image: my-custom-image:tag

  database:
    image: postgres:latest
```

In this example, the `web` service is built using a custom Dockerfile located in the `./dir` directory. The resulting image is then tagged as `my-custom-image:tag` . The `database` service uses an existing `postgres` image.

2. Exposing Ports:

To expose ports from containers to the host machine, you can use the **ports** section within a service definition. You can specify both the guest (container) port and the host port.

```
version: '3'
services:
  app:
    image: my-app-image:latest
    ports:
      - "3000"
      - "8000:80"
```

```
#Expose port to linked service (not to host)
Expose :["3000"]
```

In this example, the **app** service exposes port 3000 from the container to a random port on the host. It also maps port 80 from the container to port 8000 on the host.

Please note that these examples are meant to illustrate the structure and syntax of Docker Compose configurations. Make sure to replace **my-custom-image:tag** and **my-app-image:latest** with the actual image names you intend to use.

Remember that proper YAML indentation is crucial in Docker Compose files.

Advanced Features

1. Labels:

Labels allow you to attach metadata to your Docker services, containers, and other objects. Here's how you can define labels in a Docker Compose YAML file:

```
version: '3'
services:
  web:
    image: my_web_app_image
    labels:
      com.example.description: "Accounting web app"
```

In this example, the `web` service is assigned a label `com.example.description` with the value `"Accounting web app"`.

2. DNS Servers:

You can configure DNS servers for your services using the `dns` option. Here's how you can define DNS servers in a Docker Compose YAML file:

```
version: '3'
services:
  web:
    image: my_web_app_image
    dns:
      - 8.8.8.8
      - 8.8.4.4
```

In this example, the `web` service is configured to use the Google DNS servers (8.8.8.8 and 8.8.4.4).

3. Devices:

You can assign devices to a service using the `devices` option. Here's how you can define devices in a Docker Compose YAML file:

```
version: '3'
services:
  web:
    image: my_web_app_image
    devices:
      - "/dev/ttyUSB0:/dev/ttyUSB0"
```

In this example, the `web` service is assigned the device `/dev/ttyUSB0` from the host, and it will be available at the same path inside the container.

Remember to replace `my_web_app_image` with the actual image name you're using.

Make sure to save these configurations in a `.yaml` file (e.g., `docker-compose.yaml`) and use the `docker-compose` command to manage your services:

```
docker-compose up -d # to start your services
docker-compose down # to stop and remove your services
```

Please note that YAML indentation is crucial. Be sure to maintain proper indentation levels as shown in the examples.

Docker Services

1. To view a list of all the services running in the swarm:

```
docker service ls
```

2. To see all running services in a Docker stack:

```
docker stack services STACK_NAME
```

3. To see the logs of a specific service in a Docker stack:

```
docker service logs STACK_NAME_SERVICE_NAME
```

4. To scale a service quickly across qualified nodes in a Docker stack:

```
docker service scale STACK_NAME_SERVICE_NAME=REPLICAS
```

Replace `STACK_NAME` , `SERVICE_NAME` , and `REPLICAS` with the actual names of the Docker stack, service, and the desired number of replicas, respectively.

Please note that the command for viewing logs (`docker service logs`) requires both the stack name and the service name as arguments. The scaling command (`docker service scale`) should be used when you want to adjust the number of replicas (instances) of a specific service within a Docker swarm.

Docker Clean

1. To clean or prune unused (dangling) images:


```
docker image prune
```

2. To remove all images which are not in use containers (including dangling images):

```
docker image prune -a
```

3. To prune your entire system (including unused images, containers, networks, and volumes):

```
docker system prune
```

4. To leave a Docker swarm (if you are a node in a swarm):

```
docker swarm leave
```

5. To remove a Docker swarm (delete all services and volumes related to the stack):

```
docker stack rm STACK_NAME
```

6. To kill all running containers:

```
docker kill $(docker ps -q)
```

Note: The “docker swarm leave” command should be executed on a worker or manager node within the swarm. The “docker stack rm” command is used to remove a Docker stack (services) from a swarm cluster. Also, be cautious while using the “docker system prune” command, as it will remove all unused data (images, containers, networks, and volumes) from your system, and the operation cannot be undone. Always review what will be pruned before confirming the action.

Docker Security (Scout,SBOM)

The provided commands are part of a tool called “Docker Scout” used to analyze software artifacts, especially Docker images, for vulnerabilities. Below is a description of each command:

1. Display vulnerabilities from a Docker save tarball:

This command analyzes vulnerabilities in a Docker image saved as a tarball.

```
docker scout cves [OPTIONS] IMAGE | DIRECTORY | ARCHIVE
```

Example usage:

```
docker scout cves redis.tar
```

2. Display vulnerabilities from an OCI directory:

This command analyzes vulnerabilities in a Docker image stored in an OCI (Open Container Initiative) directory.

```
skopeo copy --override-os linux docker://alpine oci:redis
```

Note: Skopeo is a tool for working with container images and can be used to copy an image from one location to another.

3. Export vulnerabilities to a SARIF JSON file:

This command exports vulnerabilities detected in a Docker image to a SARIF (Static Analysis Results Interchange Format) JSON file.

```
docker scout cves --format sarif --output redis.sarif.json redis
```

The above command generates a SARIF JSON file named `redis.sarif.json` containing the vulnerabilities found in the `redis` Docker image.

4. Comparing two images:

This command compares two Docker images to identify any differences in vulnerabilities between them.

```
docker scout compare --to redis:6.0 redis:6-bullseye
```

The above command compares the vulnerabilities in the `redis:6.0` and `redis:6-bullseye` Docker images.

5. Displaying the Quick Overview of an Image:

This command provides a quick overview of the vulnerabilities in a Docker image.

```
docker scout quickview redis:6.0
```

The above command displays a summary of vulnerabilities found in the `redis:6.0` Docker image.

Please note that the commands provided assume you have the “Docker Scout” tool installed and properly configured. The tool may not be a standard Docker command, so you may need to install it separately. Additionally, the options and capabilities of the tool might change over time, so it’s always a good idea to refer to the tool’s documentation for the most up-to-date information.:

Contributors

Join our vibrant community of 8000+ DevOps engineers, including Docker Captain [Ajeet Raina](#). Collaborate and contribute to empower cloud-native education and innovation.

Support and Community

Support & Community: Engage with [Collabnix](#) on various platforms. Access community-contributed blogs, stay updated with announcements, connect on [slack](#) , [Twitter](#) , [Dev](#) , and chat on [Discord](#).

References

[here](#) . Explore [DockerLabs](#) , [KubeLabs](#) , [Kubetools](#) , [DockerTools](#) , and more. Enhance your DevOps and cloud-native skills.

Contributing to Collabnix

If you're excited about cloud-native technologies and have a passion for sharing your knowledge, we welcome you to contribute to Collabnix. Whether you're a seasoned pro or just starting your journey, our community provides a nurturing environment for collaboration and growth.

Get in Touch!

Docker Captain [Ajeet Raina](#) is here to guide you through the Collabnix experience. Also you could reach out to [Adesoji Alu](#) to connect. And don't forget to explore collabnix.com to embark on a journey of learning and innovation.

Join me and the entire Collabnix community as we explore the fascinating world of DevOps, Cloud-Native, AI, and beyond. Let's empower each other, one cloud-native step at a time.

Cheers to the exciting journey ahead! 🚀☀️👤

[Join our Discord Server](#)