# **Docker QA & Commands Cheat Sheet**

# **Docker Questions and Answers**

1. What is the Difference between an Image, Container, and Engine?
   1. Image: An image is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, and system tools.
   2. Container: A container is a running instance of an image that runs the application in an isolated environment.
   3. Engine: The Docker Engine is a client-server application that provides the necessary tools for managing and running Docker containers. It includes a daemon, REST API, and a command-line interface (CLI)
2. What is the Difference between the Docker command COPY vs ADD?
   1. COPY: Copies files and directories from the host machine to the container image.
   2. ADD: Used to Copy files from and directories from the host machine to the docker image. you can also use this command to download files and folders from from URL. Docker will download the file to the destination directory and if a file is a tar archive then it is automatically unpacked into the docker image/
3. What is the Difference between the Docker command CMD vs RUN?
   1. CMD: It is used to specify the default command and arguments that should be executed when a container is started. CMD is the default set of arguments that are supplied to the ENTRYPOINT process.
   2. RUN: This instruction is used to execute commands during the image build process. It is commonly used for installing dependencies, setting up the environment, and other build-time tasks.
4. How Will you reduce the size of the Docker image?
   1. Use a smaller base image.
   2. Minimize the number of layers in your image.
   3. Combine multiple RUN commands into a single layer.
   4. Avoid unnecessary packages and dependencies.
   5. Clean up temporary files and caches after installing software.
   6. Use multi-stage builds to create smaller final images.
   7. Use Docker's build context exclusion to limit the files included in the build.
5. Why and when to use Docker?
   1. Consistency Across Environments: Docker ensures consistency between development, testing, and production environments. The "it works on my machine" problem is mitigated because the application and its dependencies are packaged into a container.
   2. Isolation of Dependencies: Containers encapsulate the application and its dependencies, providing isolation. This isolation prevents conflicts between different versions of libraries or dependencies, making it easier to manage dependencies across projects.
   3. Portability: Docker containers can run on any system that supports Docker, regardless of the underlying infrastructure. This portability simplifies deployment across different environments, including local development machines, on-premises servers, or cloud platforms.
   4. Microservices Architecture: Docker facilitates the implementation of microservices architecture. Each microservice can run in its own container, allowing for easy scaling, updating, and maintenance of individual services without affecting the entire application.
   5. Rapid Deployment: Docker containers can be started and stopped quickly, enabling rapid deployment and scaling. This agility is crucial for modern development practices such as continuous integration and continuous deployment (CI/CD).
   6. Resource Efficiency: Containers share the host OS kernel and use resources more efficiently than traditional virtual machines. This results in faster startup times and lower overhead, allowing for more efficient utilization of resources.
   7. Versioning and Rollbacks: Docker images can be versioned, making it easy to roll back to a previous version if a newer release introduces issues. This versioning ensures that the deployed application is consistent across different stages of the development lifecycle.
   8. Easier Dependency Management: Docker simplifies dependency management by packaging applications and their dependencies into containers. This reduces the "it works on my machine" problem and streamlines the process of moving applications between different environments.
   9. Scalability: Docker Swarm and Kubernetes, which work with Docker, provide native support for orchestrating the deployment and scaling of containerized applications. This makes it easier to scale applications horizontally and manage clusters of container
6. Explain the Docker components and how they interact with each other.  
   Docker is composed of several components that work together to enable containerization and manage containerized applications. Here's an overview of the main Docker components and how they interact with each other:
   1. Docker Daemon (dockerd):
      * The Docker Daemon is a background process that runs on the host machine. It manages Docker containers, images, volumes, and networks. It listens for Docker API requests and communicates with the Docker CLI (Command-Line Interface) to execute commands.
   2. Docker CLI (docker):
      * The Docker CLI is the command-line tool used to interact with the Docker Daemon. Users issue commands to the Docker CLI to build, run, and manage containers. The CLI communicates with the Docker Daemon over a RESTful API.
   3. Docker Images:
      * Docker Images are the blueprints for containers. They are read-only templates containing the application code, libraries, dependencies, and runtime. Images are created from a set of instructions defined in a Dockerfile. Images can be stored locally or pushed to a Docker Registry for sharing.
   4. Docker Containers:
      * Docker Containers are instances of Docker Images. Containers are lightweight, runnable units that encapsulate the application and its dependencies. They run in isolated user spaces on the host system. Multiple containers can run concurrently on the same host, each with its own isolated environment.
   5. Docker Registries:
      * Docker Registries are centralized repositories for storing and sharing Docker Images. Docker Hub is the default public registry, and it hosts a vast collection of pre-built images. Organizations can set up private registries for their own images, ensuring control over access and distribution.
   6. Docker Hub:
      * Docker Hub is the default public registry provided by Docker. It hosts a wide range of official and community-contributed images. Users can pull images from Docker Hub to their local machine or push their images for public or private sharing.
   7. Docker Compose:
      * Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file (docker-compose.yml) to define the services, networks, and volumes required for the application. Docker Compose simplifies the management of complex, multi-container applications.
   8. Docker Swarm:
      * Docker Swarm is Docker's native clustering and orchestration solution. It enables the creation and management of a swarm of Docker nodes, turning them into a single, virtual Docker host. Docker Swarm provides built-in orchestration features for deploying and managing services across a cluster.
      * Swarm Manager:
        + The Swarm Manager is responsible for maintaining the desired state of the swarm. It orchestrates the deployment and scaling of services, monitors node health, and handles task scheduling.
      * Swarm Worker:
        + Swarm Workers are nodes in the swarm that execute tasks delegated by the Swarm Manager. They run containers and report their status back to the manager.
   9. Docker Networks:
      * Docker Networks enable communication between containers and between containers and the external network. Docker supports various network drivers, including bridge networks (default), overlay networks, host networks, and macvlan networks. Networks facilitate secure communication and isolation between containers.
   10. Docker Volumes:
       * Docker Volumes provide persistent and shared storage for containers. Volumes can be mounted into containers, allowing data to persist even if the container is stopped or removed. Volumes facilitate data management and sharing between containers.
   11. Dockerfile:
       * A Dockerfile is a script that contains a set of instructions for building a Docker Image. It specifies the base image, dependencies, and steps to configure the containerized environment. Dockerfiles are used in conjunction with the docker build command to create Docker Images.

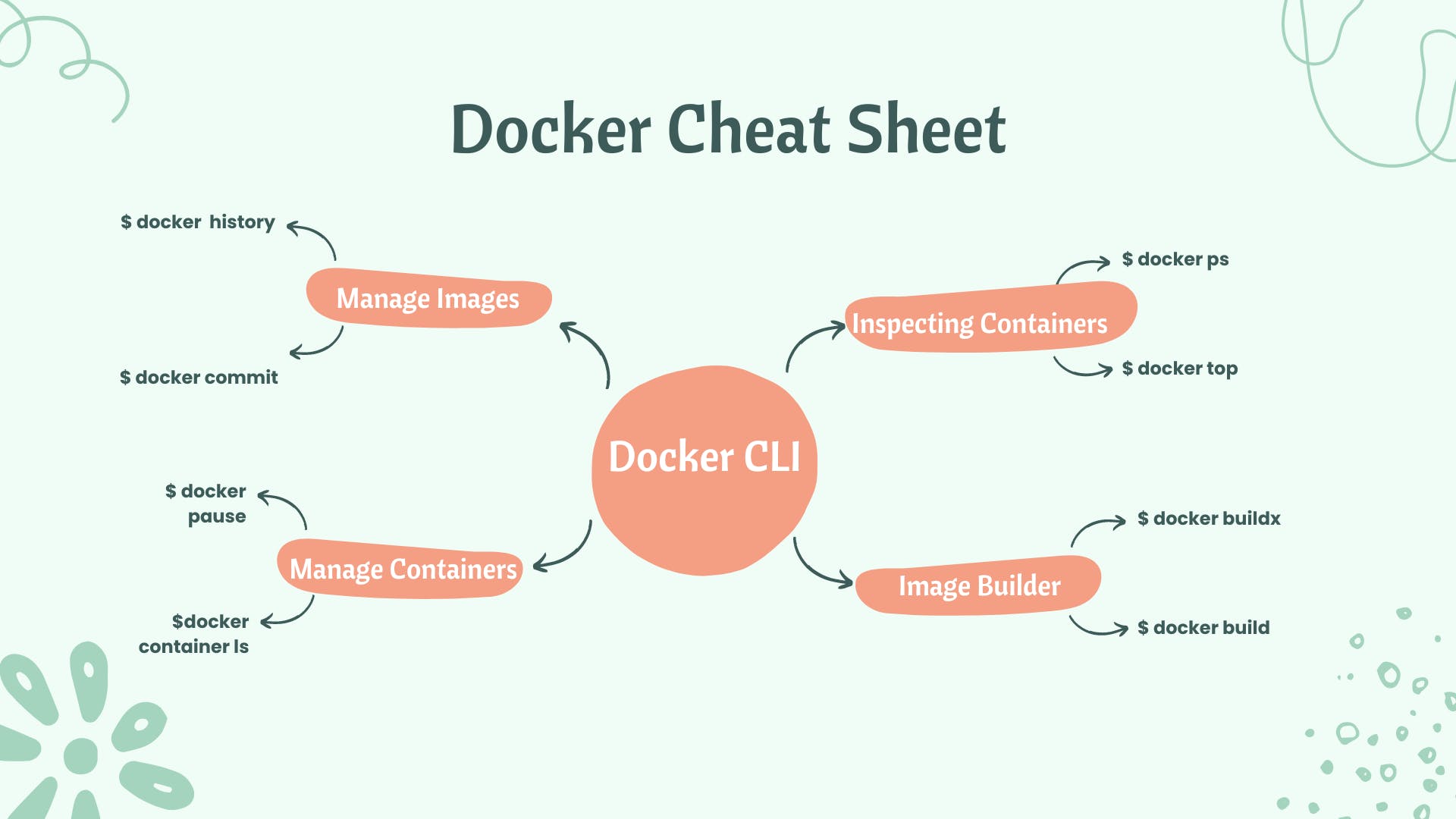
Interactions:

* Image Build and Pull:
  + Users define Docker Images using Dockerfiles. These images can be built using the docker build command. Images can also be pulled from Docker Registries using the docker pull command.
* Container Creation and Execution:
  + Containers are created and executed using the docker run command. Users specify the image, container name, and any required configurations. The Docker Daemon orchestrates the creation and execution of containers.
* Docker Compose Deployment:
  + Docker Compose uses a YAML file to define multi-container applications. The docker-compose CLI is used to deploy and manage these applications. Compose interacts with the Docker Daemon to create and manage the specified services.
* Docker Swarm Deployment:
  + Docker Swarm is used for orchestrating the deployment and scaling of services across a cluster of nodes. The Swarm Manager interacts with the Docker Daemon on worker nodes to schedule and manage tasks.
* Networks and Volumes:
  + Docker Networks and Volumes are used to facilitate communication between containers and to provide persistent storage, respectively. They ensure that containers can interact with each other and share data as needed.
* Docker Registry Interaction:
  + Docker Images can be pushed to or pulled from Docker Registries using the docker push and docker pull commands. Registries play a central role in sharing and distributing Docker Images.

In summary, Docker components work together to provide a comprehensive containerization solution. The Docker Daemon, CLI, Images, Containers, Registries, Compose, Swarm, Networks, Volumes, and Dockerfiles all contribute to the creation, deployment, and management of containerized applications. Understanding how these components interact is essential for effectively using Docker in various scenarios.

1. In what real scenarios have you used Docker?
   * Docker is commonly used in software development, particularly in modern software development practices like DevOps and microservices architecture.
   * It allows for the deployment and scaling of applications in a consistent and portable way across different cloud platforms.
   * Additionally, Docker is often used in testing and continuous integration and delivery (CI/CD) workflows to automate the building, testing, and deployment of software applications
2. Docker vs Hypervisor?
   * Docker: Uses containerization to run applications in isolated environments, sharing the host OS kernel. It is more lightweight, efficient, and offers faster startup times.
   * Hypervisor: Creates virtual machines (VMs) that run complete OS instances. It is less efficient due to the overhead of managing multiple operating systems.
3. What are the advantages and disadvantages of using docker?
   * Advantages: Portability, Consistency, Isolation, Efficiency, Rapid Deployment, Microservices Architecture, Resource Efficiency, Scalability, Versioning and Rollbacks, DevOps Integration and Community and Ecosystem
   * Disadvantages: Complexity for certain applications, Security concerns in multi-tenant environments.
4. What is a Docker namespace?
   * A Docker namespace is a mechanism that isolates and separates resources within a container, providing each container with its own view of system resources like process IDs, network interfaces, and filesystem mounts.
   * It helps avoid conflicts between containers and ensures that each container runs independently.
5. What is a Docker registry?
   * A Docker registry is a repository for storing and sharing Docker images. It allows you to distribute and manage container images, making them accessible to users across different systems
6. What is an entry point?
   * The entry point is the command that specifies what executable should run when a container is started from a Docker image. It defines the default behavior of the container and can be overridden at runtime
7. How to implement CI/CD in Docker?
   * Write your application's code and store it in a Git repository.
   * Create a Dockerfile that describes your application's environment and dependencies.
   * Use a CI/CD tool like GitLab CI/CD or GitHub Actions to automatically build, test, and push the Docker images to a registry.
   * Use a container orchestration tool like Kubernetes or Docker Swarm to manage and deploy the Docker containers in the production environment.
8. Will data on the container be lost when the docker container exits?
   * Yes, by default, data stored in a container's ephemeral filesystem is lost when the container exits. To persist data, you can use Docker volumes or bind mounts.
9. What is a Docker swarm?
   * Docker Swarm is a tool that allows you to manage multiple Docker hosts as a single cluster.
   * It helps you deploy and scale applications across those hosts, providing features such as load balancing and service discovery.
   * Swarm simplifies the deployment of containerized applications.

# **Docker Cheat Sheet**

****

### **✅ Docker Run Commands**

Docker uses the run command to create containers from provided images. The default syntax for this command looks like this:

COPY

COPY

docker run (options) image (command) (arg...)



| Flag | Explanation |
| --- | --- |
| --detach , -d | Runs a container in the background and prints the container ID |
| --env , -e | Sets environment variables |
| --hostname , -h | Sets a hostname to a container |
| --label , -l | Creates a metadata label for a container |
| --name | Assign a name to a container |
| --network | Connects a container to a network |
| --rm | Removes container when it stops |
| --read-only | Sets the container filesystem as read-only |
| --workdir , -w | Sets a working directory in a container |

### **✅ Docker Container Interaction Commands**

| Command | Explanation |
| --- | --- |
| docker start container | Starts a new container |
| docker stop container | Stops a container |
| docker pause container | Pauses a container |
| docker unpause container | Unpauses a container |
| docker restart container | Restarts a container |
| docker export container | Exports container contents to a tar archive |
| docker attach container | Attaches to a running container |
| docker commit -m “commit message” -a “author” container username/image\_name: tag | Saves a running container as an image |
| docker logs -ft container | Follows container logs |
| docker exec -ti container [script.sh](https://www.hostinger.in/tutorials/docker-start-a-container/) | Runs a command in a container |
| docker commit container image | Creates a new image from a container |
| docker create image | Creates a new container from an image |

### **✅ Docker Container Inspection Commands**

| Command | Explanation |
| --- | --- |
| docker ps | Lists all running containers |
| docker -ps -a | Lists all containers |
| docker diff container | Inspects changes to directories and files in the container filesystem |
| docker top container | Shows all running processes in an existing container |
| docker inspect container | Displays low-level information about a container |
| docker logs container | Gathers the logs for a container |
| docker stats container | Shows container resource usage statistics |

### **✅ Docker Images Commands**

| Command | Explanation |
| --- | --- |
| docker images | Lists images |
| docker image rm mysql | Removes an image |
| docker tag image tag | Tags an image |
| docker history image | Displays the image history |
| docker inspect image | Displays low-level information about an image |

### **✅ Docker Build Commands**

| Command | Explanation |
| --- | --- |
| docker build -t <imagename> . | Builds an image from a Dockerfile in the current directory |
| docker build -t <imagename> . -–no-cache | Builds an image from a Dockerfile in the current directory without cache |
| docker build -t imagename/tag | Builds and tags an image for easier tracking |
| docker build <https://yourserver/file.tar.gz> | Builds an image from a remote tar archive |

### **✅ Docker Clean Up Commands**

| Command | Explanation |
| --- | --- |
| docker image prune | Clears an unused image |
| docker image prune -a | Clears all images that are not being used by containers |
| docker system prune | Removes all stopped containers, all networks not used by containers, all dangling images, and all build cache |
| docker image rm image | Removes an image |
| docker rm container | Removes a running container |
| docker swarm leave | Leaves a swarm |
| docker stack rm stackname | Removes a swarm |
| docker volume rm $(docker volume ls -f dangling=true -q) | Removes all dangling volumes |
| docker rm $(docker ps -a -q) | Removes all stopped containers |
| docker kill $ (docker ps -q) | Stops all running containers |

### **✅ Docker Network Commands**

| Command | Explanation |
| --- | --- |
| docker network create networkname | Creates a new network |
| docker network rm networkname | Removes a specified network |
| docker network ls | Lists all networks |
| docker network connect networkname container | Connects a container to a network |
| docker network disconnect networkname container | Disconnects a container from a network |
| docker network inspect networkname | Displays detailed information about a network |
| docker network prune | Remove unused network |

### **✅ Docker Volume Commands**

| Command | Explanation |
| --- | --- |
| docker volume ls | List Volumes |
| docker volume create volumename | Create a volume |
| docker volume inspect volumename | Display volume details |
| docker volume rm volumename | Remove a volume |
| docker volume prune | Remove unused volume |

### **✅ Docker Registry Commands**

| Command | Explanation |
| --- | --- |
| docker login | Logs in to a registry |
| docker logout | Logs out from a registry |
| docker pull mysql | Pulls an image from a registry |
| docker push repo/ rhel-httpd:latest | Pushes an image to a registry |
| docker search term | Searches Docker Hub for images with the specified term |

### **✅ Docker Service Commands**

| Command | Explanation |
| --- | --- |
| docker service ls | Lists all services running in a swarm |
| docker stack services stackname | Lists all running services |
| docker service ps servicename | Lists the tasks of a service |
| docker service update servicename | Updates a service |
| docker service create image | Creates a new service |
| docker service scale servicename=10 | Scales one or more replicated services |
| docker service logs stackname servicename | Lists all service logs |