Day 21 Task: Docker Important interview Questions:

Questions:

1. What is the Difference between an Image, Container and Engine?

An **Image** is a pre-configured file system containing all required files, libraries, dependencies to run an application.

A **Container** is a running instance of an Image, which runs in an isolated environment with its own file system, networking, and resources.

An **Engine** is a platform for running containers, it provides a layer of abstraction between the host system and containers, managing containers' lifecycle, networking, storage, and resource allocation. The most popular engine is Docker.

2. What is the Difference between the Docker command COPY vs ADD?

The **COPY** and **ADD** commands in Dockerfiles serve the same purpose, which is to copy files from the host system into a Docker image. The main difference is that **ADD** has some additional functionality:

- **ADD** supports automatically unpacking compressed files (such as tar files) from the host into the Docker image.
- **ADD** can fetch files from a URL and copy them into the Docker image.

Otherwise, **COPY** is the preferred command in modern Dockerfiles because it is more transparent and predictable than **ADD**.

3. What is the Difference between the Docker command CMD vs RUN?

The **CMD** and **RUN** commands in Dockerfiles serve different purposes:

RUN command is used to execute a command during the image building process. The command gets executed and the result is committed to the image.

CMD command specifies the command that should be run when a container is started from the image. It provides default arguments for an executing container. If a user specifies command line arguments when starting a container, the **CMD** value will be overwritten.

In summary, **RUN** is used to build the image and **CMD** is used to specify the default command that should be run when a container is started from the image.

4. How Will you reduce the size of the Docker image?

There are several ways to reduce the size of a Docker image:

- **Multi-stage builds:** Use multiple Dockerfiles in a multi-stage build process to only include the final required artifacts in the final image, discarding intermediate build artifacts and unused files.
- **Use smaller base images:** Select base images with the smallest possible size, for example, using Alpine Linux instead of full-featured distributions like Ubuntu.
- **Minimize the number of installed packages:** Only install the packages required for the application to run, removing unnecessary packages and dependencies.
- **Remove unnecessary files:** Remove files like test files, debug symbols, and documentation that are not needed in production.
- **Use image optimization tools:** There are several image optimization tools that can further reduce the image size by removing duplicates, compressing files, and optimizing file systems.

By following these best practices, you can reduce the size of a Docker image and improve its performance.

5. Why and when to use Docker?

Docker is used to containerize applications, providing a consistent and reproducible environment for development, testing, and deployment. Here are some reasons why and when to use Docker:

- **Isolation:** Docker containers provide isolated environments for each application, ensuring that applications run consistently across different environments.
- **Portability:** Docker images can be easily moved between environments, allowing developers to quickly spin up new instances of an application on any platform that supports Docker.
- **Scalability:** Docker containers can be easily scaled horizontally by adding more instances, making it simple to accommodate increased traffic or demand.
- **Ease of deployment:** Docker containers can be deployed on any platform that supports Docker, making it easy to deploy an application consistently across different environments.
- **Environment consistency:** Docker containers ensure that an application runs consistently across different environments, making it easier to test and deploy applications.

In summary, use Docker when you need to run an application consistently across different environments, or when you need to easily scale an application to accommodate increased demand. Docker is also useful for deploying applications quickly and consistently across different environments.

6. Explain the Docker components and how they interact with each other.

Docker has several components that interact with each other to provide a complete platform for building, deploying, and running containers. These components are:

- **Docker Daemon:** The Docker Daemon is the core component of Docker, responsible for building, running, and managing containers. It communicates with other components through the Docker API.
- **Docker CLI:** The Docker CLI is a command-line interface for interacting with the Docker Daemon. Users can use the CLI to build, run, and manage containers.
- **Docker Images:** Docker Images are snapshots of a file system that contain all the files and dependencies needed to run an application. They are stored in a registry and can be used to create new containers.
- **Docker Registries**: Docker Registries are repositories for storing and distributing Docker Images. The most well-known registry is Docker Hub, but users can also set up their own private registries.
- **Docker Containers:** Docker Containers are instances of Docker Images that run as isolated processes on a host system. They provide a consistent environment for running applications.

In summary, these components interact with each other to provide a complete platform for building, deploying, and running containers. The Docker CLI communicates with the Docker Daemon to build and manage containers, using images stored in a registry. The Docker Daemon runs containers, providing isolated environments for running applications.

7. Explain the terminology: Docker Compose, Docker File, Docker Image, Docker Container?

The following are common terms used in Docker:

- **Docker File:** A Docker File is a script that contains instructions for building a Docker Image. It specifies the base image, files to be copied, and commands to run when building the image.
- **Docker Image:** A Docker Image is a snapshot of a file system that contains all the files and dependencies needed to run an application. It is created from a Docker File and can be used to create new containers.
- **Docker Container:** A Docker Container is an instance of a Docker Image that runs as an isolated process on a host system. It provides a consistent environment for running applications.
- **Docker Compose:** Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file to define the services, networks, and volumes for an application, and it can be used to start and stop the application with a single command.

8. In what real scenarios have you used Docker?

I have used Docker in a variety of scenarios including microservices architecture, continuous integration and delivery, and development environments.

9. Docker vs Hypervisor?

Docker and hypervisors are both technologies for virtualization, but they have different design goals and use cases:

- **Docker:** Docker is designed for application virtualization, where each application runs in its own container. Containers share the host operating system, making them lightweight and efficient. Docker is ideal for microservices and cloud-based deployments.
- **Hypervisor:** A hypervisor is a layer of software that allows multiple virtual machines to run on a single physical machine. Each virtual machine runs its own operating system and has its own resources, making it ideal for server virtualization and multi-tenant environments.

In summary, Docker and hypervisors are both virtualization technologies, but they have different design goals and use cases. Docker is designed for application virtualization, while hypervisors are designed for server virtualization. The choice between the two depends on the specific requirements of the application and the environment in which it runs.

10. What are the advantages and disadvantages of using docker?

Advantages of using Docker:

- **Portability:** Docker containers can run on any host with a Docker engine, making it easy to move applications between development, testing, and production environments.
- **Isolation:** Docker containers run in isolated environments, ensuring that applications do not interfere with each other and reducing the risk of security vulnerabilities.
- **Scalability:** Docker containers can be easily scaled up or down to meet changing demands, making it easy to handle spikes in traffic or changing workloads.
- **Efficiency:** Docker containers are lightweight, using fewer resources than traditional virtual machines.
- **Ease of use**: Docker provides an easy-to-use platform for building, shipping, and running applications, making it accessible to a wide range of users.

Disadvantages of using Docker:

- **Complexity:** Docker can add complexity to an application, especially when deploying multi-container applications.
- **Performance overhead:** The overhead of running containers can be higher than running applications directly on a host, especially for resource-intensive applications.

- **Security concerns:** Docker containers share the host operating system, making it important to secure the host and to properly isolate containers from each other to reduce the risk of security vulnerabilities.
- **Learning curve:** Docker requires a certain level of knowledge and understanding to use effectively, making it a steep learning curve for some users.

11. What is a Docker namespace?

A Docker namespace is a feature of the Docker engine that provides isolated environments for running containers. Each namespace provides a separate view of system resources, such as the network, process tree, and file system, allowing multiple containers to run on the same host without interfering with each other.

Docker provides several types of namespaces, including the following:

- **PID namespace:** The PID namespace isolates the process tree, allowing each container to have its own process hierarchy.
- **Network namespace:** The network namespace isolates network resources, such as network interfaces and IP addresses, allowing each container to have its own network configuration.
- **Mount namespace:** The mount namespace isolates the file system, allowing each container to have its own file system hierarchy.
- **User namespace:** The user namespace isolates user and group IDs, allowing containers to run with different user and group permissions.

By using namespaces, Docker provides a secure and isolated environment for running containers, allowing multiple containers to run on the same host without interfering with each other. This makes it easier to run multiple applications on a single host, improving resource utilization and reducing the need for additional hardware.

12. What is a Docker registry?

A Docker registry is a repository for storing and distributing Docker images. A Docker registry acts as a central location for storing and sharing Docker images, making it easy to share and distribute images between different environments and teams.

Docker provides a public registry, Docker Hub, where users can store and share images with the wider Docker community. Docker Hub is free to use, but it has limited storage and bandwidth capabilities. There are also several commercial registry services available, such as AWS Elastic Container Registry (ECR), Google Container Registry (GCR), and Azure Container Registry (ACR), which provide additional storage and bandwidth capabilities, as well as advanced security and management features.

To use a Docker registry, images can be pushed to the registry from a local machine or build environment, and then pulled from the registry and run on any host with a Docker engine. This makes it easy to share and distribute images, as well as automate the deployment of applications.

13. What is an entry point?

An entry point in Docker is a command that is automatically executed when a container is started. The entry point command is specified in the Dockerfile using the **CMD** or **ENTRYPOINT** instructions.

The **CMD** instruction specifies the default command to be executed when a container is started. If a different command is specified when starting the container, it will override the default command specified in the **CMD** instruction.

The **ENTRYPOINT** instruction, on the other hand, specifies a command that must be executed when a container is started, and any command specified when starting the container will be passed as arguments to the **ENTRYPOINT** command.

The entry point is used to specify the default behavior of a container, and it can be used to configure the container to run a specific application or service when it is started. The entry point also provides a convenient way to run multiple commands or start a service in the background when a container is started.

14. How to implement CI/CD in Docker?

CI/CD (Continuous Integration and Continuous Deployment) in Docker involves automating the process of building, testing, and deploying Docker applications. Here are the general steps to implement CI/CD in Docker:

- **Source control:** Store the code and related files in a source control repository such as Git. This allows for version control and collaboration between team members.
- **Continuous Integration (CI):** Automate the build process using a CI tool such as Jenkins or TravisCI. The CI tool will build the Docker image from the code in the source control repository and run automated tests on the image.
- **Automated testing:** Automate the testing of the Docker image using a testing framework such as JUnit or TestNG. The tests can be run in parallel on multiple containers to validate the application.
- **Docker registry:** Store the built Docker images in a Docker registry, such as Docker Hub or a private registry. This makes it easy to distribute the images to other environments for deployment.
- **Continuous Deployment (CD):** Automate the deployment process using a CD tool such as Jenkins, TravisCI, or AWS CodeDeploy. The CD tool will deploy the Docker images stored in the registry to the production environment.
- **Monitoring:** Monitor the deployed application and containers to ensure that they are running as expected.

15. Will data on the container be lost when the docker container exits?

By default, data in a Docker container is not persisted when the container exits. When a container stops, the file system, network configuration, and other state information are deleted, and all changes made to the container are lost.

However, there are ways to persist data even after a container has exited. One way to do this is to use Docker volumes, which allow you to store data on the host file system and mount it into the container. When a container is deleted, the volume still exists and can be used by another container.

Another way to persist data is to use data volumes in the Dockerfile. You can use the **VOLUME** instruction to specify which directories in the container should be treated as data volumes, and then use the **--mount** or **-v** option when starting the container to map the data volume to a directory on the host file system.

In summary, data in a Docker container is not persisted by default when the container exits, but you can use Docker volumes or data volumes to persist data even after a container has exited.

16. What is a Docker swarm?

Docker Swarm is a native orchestration solution for Docker containers, providing the ability to manage and deploy containers at scale. It allows you to turn a group of Docker engines into a single virtual Docker engine, making it easier to manage and orchestrate containers across multiple hosts.

A Docker Swarm cluster is composed of a set of Docker engines, each running on its own host. The Docker engines work together to form a swarm, and the swarm manager nodes coordinate the deployment and management of containers across the swarm.

With Docker Swarm, you can:

- **Easily scale your applications**: You can easily scale your applications by adding or removing nodes to the swarm, and the swarm manager will automatically distribute containers across the available nodes.
- **Load balance containers:** Docker Swarm automatically load balances traffic between containers, ensuring that your applications are always available and responsive.
- **Manage the entire lifecycle of containers:** You can use Docker Swarm to manage the entire lifecycle of containers, from deployment to scaling to removal.
- **Offer high availability:** Docker Swarm ensures high availability by automatically rescheduling containers on healthy nodes in the event of a node failure.

In summary, Docker Swarm is a native orchestration solution for Docker containers that makes it easier to manage and deploy containers at scale, offering features such as automatic scaling, load balancing, and high availability.

17. What are the docker commands for the following:

- **view running containers**

- docker ps

- **command to run the container under a specific name**

- docker run –name <name> <image>

- **command to export a docker**

- docker export <container> <filename>

- **command to import an already existing docker image**

- docker import <filename> <repository:tag>

- **commands to delete a container**

- docker rm <container>

- **command to remove all stopped containers, unused networks, build caches, and dangling images?**

- Docker system prune

18. What are the common docker practices to reduce the size of Docker Image?

Common practices to reduce the size of Docker images include: using multi-stage builds, removing unnecessary files and dependencies, using smaller base images, using a .dockerignore file, squashing multiple layers, using a caching

mechanism, version pinning and using image scanning and vulnerability analysis tools. While these practices can help reduce the size of images, it’s important to consider the impact on the functionality of the application.

Thanks for reading!

Happy Learning 😊