Basic (1-10)

1. Question: What is Docker, and what problem does it solve?

Expected Answer: Docker is a platform for developing, shipping, and running applications inside containers. It solves the problem of inconsistent environments across different stages (development, testing, production). By packaging an application with all its dependencies into a container, Docker ensures that the application runs the same way, regardless of where it's deployed. This eliminates "it works on my machine" issues.

2. Question: Explain the difference between a Docker image and a Docker container.

Expected Answer: A Docker image is a read-only template containing instructions for creating a Docker container. Think of it as a blueprint. A Docker container is a running instance of a Docker image. You can have multiple containers running from the same image. Example: You have a nginx image. You can run multiple nginx containers from that single image, each serving potentially different configurations or virtual hosts.

3. Question: What is a Dockerfile? Provide a simple example.

Expected Answer: A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. It's the recipe for building a Docker image.

```
   FROM ubuntu:latest  # Base image
RUN apt-get update && apt-get install -y nginx  # Install nginx
COPY index.html /var/www/html  # Copy web content
EXPOSE 80  # Expose port 80
CMD ["nginx", "-g", "daemon off;"]  # Start nginx
```

4. Question: Describe the basic Docker architecture.

Expected Answer: The core components are:

- **Docker Client:** The interface for interacting with the Docker Daemon (command line, API).
- **Docker Daemon (dockerd):** The background process that manages Docker images, containers, networks, and volumes. It listens for Docker API requests.
- **Docker Registry (Docker Hub, private registries):** A storage location for Docker images. Docker Hub is the default public registry.
- **Docker Images:** Templates used to create containers.
- **Docker Containers:** Running instances of images.

5. Question: What are some common Docker commands you've used?

Expected Answer: Examples include:

- docker run: Runs a container from an image. (e.g., docker run -d -p 80:80 nginx)
- docker build: Builds an image from a Dockerfile. (e.g., docker build -t my-nginx .)
- docker ps: Lists running containers.
- docker images: Lists available images.
- docker stop: Stops a running container. (e.g., docker stop <container_id>)
- docker rm: Removes a stopped container. (e.g., docker rm <container_id>)
- docker rmi: Removes an image. (e.g., docker rmi <image_id>)
- docker pull: Downloads an image from a registry. (e.g., docker pull ubuntu)
- docker push: Uploads an image to a registry. (requires login).

6. Question: How do you run a Docker container in detached mode? What is the advantage?

Expected Answer: Use the -d flag with docker run. Example: docker run -d nginx. Detached mode runs the container in the background, freeing up your terminal. This is crucial for production deployments, allowing the container to run independently.

7. Question: How do you expose a port from a Docker container to the host machine?

Expected Answer: Use the -p flag with docker run. It maps a port on the host machine to a port inside the container. Example: docker run -p 8080:80 nginx. This maps port 8080 on the host to port 80 inside the container, allowing access to the nginx web server from the host machine's browser.

8. Question: How can you view the logs of a running Docker container?

Expected Answer: Use the docker logs command followed by the container ID or name. Example: docker logs <container_id>. This outputs the container's standard output and standard error streams.

9. Question: What is the purpose of the .dockerignore file?

Expected Answer: The .dockerignore file specifies files and directories that should be excluded from the Docker build context. This reduces the size of the build context, speeding up the build process and preventing sensitive files from being included in the image. Example: ignoring .git/, node_modules/

10. Question: Explain the CMD and ENTRYPOINT instructions in a Dockerfile. What is the difference?

Expected Answer: Both specify the command to run when the container starts.

- CMD: Provides defaults for an executing container. It can be overridden when running the container. If the image is executed, CMD will be the last instruction executed.

- ENTRYPOINT: Configures a container that will run as an executable. It is less likely to be overridden, especially when used in conjunction with CMD.

Example:

```
    ENTRYPOINT ["/usr/bin/java", "-jar", "app.jar"]
CMD ["--spring.profiles.active=dev"]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.Dockerfile
IGNORE_WHEN_COPYING_END

When running the container, you can override the CMD using docker run <image_name> --spring.profiles.active=prod. ENTRYPOINT is harder to override but possible by using --entrypoint.

Intermediate (11-20)

11. Question: How do you create a custom Docker network, and why would you want to do so?

Expected Answer: Use the docker network create command. Example: docker network create my-network.

Reasons for creating custom networks:

- **Isolation:** Containers on different networks are isolated from each other.
- **Service Discovery:** Docker provides built-in DNS resolution on user-defined networks. Containers can communicate using their service names (container names).
- **Linking Containers:** Although deprecated and largely replaced by user-defined networks, it provided a way for containers to communicate. User-defined networks are generally better for service discovery and isolation.

12. Question: Explain Docker volumes and the different types. When would you use each type?

Expected Answer: Docker volumes are the preferred mechanism for persisting data generated by and used by Docker containers. Types include:

- **Named Volumes:** Created and managed by Docker. Stored in a location managed by Docker (usually /var/lib/docker/volumes). Use when you need persistent storage that is managed by Docker. Example: Database data.
- **Bind Mounts:** Mount a file or directory from the host machine into the container. Use when you need to share files between the host and container or when you need to access host files from the container. Example: Mounting a configuration file from the host into the container.

- **tmpfs Mounts:** Stored in the host's memory. Data is not persisted when the container stops. Use for sensitive data or temporary files that don't need to be persisted. Example: storing temporary session data.

13. Question: How do you define and manage multi-container applications using Docker Compose? Provide an example docker-compose.yml file.

Expected Answer: Docker Compose is a tool for defining and running multi-container Docker applications. You define the services in a docker-compose.yml file.

```
   version: "3.9"
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./html:/usr/share/nginx/html
    depends_on:
      - app

  app:
    image: my-python-app:latest # Replace with your app's image
    environment:
      - DATABASE_URL=postgres://user:password@db:5432/mydb
    depends_on:
      - db

  db:
    image: postgres:13
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=mydb
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.Yaml
IGNORE_WHEN_COPYING_END

Commands:

- docker-compose up: Builds and starts the services defined in the docker-compose.yml file.
- docker-compose down: Stops and removes the containers, networks, and volumes created by docker-compose up.

14. Question: What are multi-stage builds in Docker, and why are they useful? Give an example.

Expected Answer: Multi-stage builds allow you to use multiple FROM statements in a Dockerfile. Each FROM instruction starts a new stage. You can selectively copy artifacts from one stage to another, resulting in smaller, leaner images.

```
    # Stage 1: Build the application
FROM maven:3.8.1-openjdk-17 AS builder
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean install -DskipTests

# Stage 2: Create the final image
FROM openjdk:17-jre-slim
WORKDIR /app
COPY --from=builder /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.Dockerfile
IGNORE_WHEN_COPYING_END

Benefits: Reduces image size by excluding build tools and intermediate artifacts from the final image.

15. Question: How can you limit the resources (CPU, memory) that a Docker container can use?

Expected Answer: Use the --cpus and --memory flags with docker run.

Example: docker run --cpus="0.5" --memory="512m" nginx. This limits the container to 0.5 CPUs and 512MB of memory. You can also define resource limits in Docker Compose files using the deploy section for each service.

16. Question: How do you handle logging in Docker containers? What are some best practices?

Expected Answer: Docker captures the standard output and standard error of containers. You can view these logs using docker logs. Best practices include:

- **Logging Driver Configuration:** Configure logging drivers (e.g., json-file, syslog, fluentd) to send logs to a central logging system.
- **Avoid Logging to Files Inside the Container:** Writing to files inside the container makes log management difficult. Use standard output/error instead.
- **Structured Logging:** Use a structured logging format (e.g., JSON) to make log data easier to parse and analyze.
- **Log Rotation:** Implement log rotation to prevent logs from consuming excessive disk space.

17. Question: What are some common Docker security concerns, and how can you mitigate them?

Expected Answer:

- **Image Vulnerabilities:** Use trusted base images, regularly scan images for vulnerabilities (using tools like docker scan, Clair, or Trivy), and keep images updated.
- **Container Isolation:** Use namespaces and cgroups to isolate containers. Avoid running containers as root. Use user namespaces.
- **Secrets Management:** Avoid storing secrets (passwords, API keys) in Dockerfiles or images. Use Docker Secrets, environment variables, or dedicated secrets management solutions (e.g., HashiCorp Vault).
- **Network Security:** Restrict network access to containers using Docker networks and firewalls.
- **Resource Limits:** Limit container resources (CPU, memory) to prevent denial-of-service attacks.

18. Question: How do you pass environment variables to a Docker container?

Expected Answer:

- -e flag with docker run: Example: docker run -e MY_VARIABLE=value nginx.
- --env-file flag with docker run: Specifies a file containing environment variables.
- environment section in docker-compose.yml: Define environment variables for each service.
- ARG and ENV inside the Dockerfile. ARG variables are only available during build time while ENV variables are available during runtime.

19. Question: What is a health check in Docker, and how do you configure one?

Expected Answer: A health check is a command that Docker runs periodically inside a container to determine if the container is healthy and ready to serve traffic. It is defined using the HEALTHCHECK instruction in a Dockerfile.

```
   HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution.Dockerfile
IGNORE_WHEN_COPYING_END

This example uses curl to check if the web server is responding. If the check fails, the container is marked as unhealthy. Orchestration tools like Kubernetes use health checks to automatically restart unhealthy containers.

20. Question: You have a Docker image that is too large. How would you troubleshoot and optimize it?

Expected Answer:

1. **Analyze Image Layers:** Use docker history <image_id> to identify the largest layers.
2. **Multi-Stage Builds:** Use multi-stage builds to reduce the final image size.
3. **Use Smaller Base Images:** Choose smaller base images (e.g., Alpine Linux instead of Ubuntu) if possible.
4. **Remove Unnecessary Files:** Delete unnecessary files and dependencies from the image.
5. **Optimize Package Management:** Use package managers efficiently (e.g., combine multiple apt-get install commands into a single command, clean up package caches).
6. **Use .dockerignore:** Ensure the .dockerignore file is properly configured to exclude unnecessary files.
7. **Squash Images:** Although generally not recommended for production due to potential layer caching issues, docker image squash can flatten layers into a single layer (use with caution).

Advanced (21-30)

21. Question: You're running a Docker container in production, and it's experiencing performance issues. How would you approach debugging and optimizing its performance?

Expected Answer:

1. **Resource Monitoring:** Use docker stats to monitor CPU, memory, network, and I/O usage of the container. Use tools like Prometheus and Grafana for persistent monitoring.
2. **Profiling:** Use profiling tools (e.g., perf, jprofiler) to identify performance bottlenecks in the application code running inside the container.

3. **Network Analysis:** Use tcpdump or wireshark to analyze network traffic to and from the container.
4. **Logging Analysis:** Analyze container logs to identify errors, warnings, or slow queries.
5. **Resource Limits:** Ensure the container has sufficient resources allocated (CPU, memory). Adjust resource limits if necessary.
6. **Application Optimization:** Optimize the application code itself to improve performance (e.g., improve database queries, optimize algorithms).
7. **Caching:** Implement caching mechanisms to reduce the load on backend services.
8. **Load Testing:** Perform load testing to identify performance bottlenecks under realistic load conditions.

22. Question: Describe the process of setting up a CI/CD pipeline for Docker images using a tool like Jenkins or GitLab CI.

Expected Answer:

1. **Code Repository:** Store application code in a Git repository (e.g., GitHub, GitLab).
2. **Dockerfile:** Create a Dockerfile to build the Docker image.
3. **CI/CD Configuration:** Configure a CI/CD pipeline in Jenkins, GitLab CI, or similar tool. The pipeline should include the following stages:
   - **Build:** Build the Docker image from the Dockerfile.
   - **Test:** Run unit tests and integration tests on the built image.
   - **Scan:** Scan the image for vulnerabilities using tools like docker scan, Clair, or Trivy.
   - **Push:** Push the image to a Docker registry (e.g., Docker Hub, private registry) if the build and tests are successful.
   - **Deploy:** Deploy the new image to the target environment (e.g., Kubernetes cluster).
4.
5. **Triggers:** Configure triggers to automatically start the pipeline when code is pushed to the repository.
6. **Notifications:** Set up notifications to alert developers of build failures or successful deployments.
7. **Versioning:** Tag Docker images with meaningful versions (e.g., semantic versioning) to track changes and facilitate rollbacks.

23. Question: How would you implement rolling updates for a Dockerized application in a Kubernetes cluster?

Expected Answer:

1. **Deployment Object:** Define a Kubernetes Deployment object for the application.
2. **Image Update:** Update the image field in the Deployment object with the new image version.

3. **Rolling Update Strategy:** Kubernetes will automatically perform a rolling update by gradually replacing old pods with new pods, ensuring minimal downtime. The default strategy is RollingUpdate.
4. **Health Checks:** Kubernetes uses health checks to determine if new pods are healthy before routing traffic to them. Configure livenessProbe and readinessProbe in your deployment.
5. **Rollback:** If a rolling update fails, Kubernetes can automatically roll back to the previous version.

24. Question: Explain how you would implement a secure secret management strategy for Docker containers deployed in a production environment.

Expected Answer:

1. **Avoid Embedding Secrets in Images:** Never hardcode secrets in Dockerfiles or images.
2. **Docker Secrets:** Use Docker Secrets to securely store and manage secrets. Docker Secrets are encrypted at rest and in transit. They are available only to services that are granted access.
3. **Environment Variables:** Pass secrets as environment variables to containers. However, environment variables are not encrypted and can be exposed in certain situations.
4. **Dedicated Secrets Management Solutions:** Use dedicated secrets management solutions like HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault. These tools provide robust security features, including encryption, access control, and audit logging.
5. **Least Privilege:** Grant containers only the necessary permissions and access to secrets.
6. **Rotation:** Regularly rotate secrets to minimize the impact of a potential compromise.
7. **Auditing:** Implement auditing to track access to secrets and identify potential security breaches.

25. Question: How do you troubleshoot a "CrashLoopBackOff" error in a Kubernetes pod running a Docker container?

Expected Answer:

1. **Describe the Pod:** Use kubectl describe pod <pod_name> to check the pod's events and identify the reason for the crash.
2. **Check Container Logs:** Use kubectl logs <pod_name> -c <container_name> to view the container's logs and identify any errors or exceptions.
3. **Check Resource Limits:** Ensure the pod has sufficient resources (CPU, memory) allocated. Check the pod's resource requests and limits.
4. **Check Health Checks:** Verify that the pod's health checks (livenessProbe and readinessProbe) are correctly configured and that the application is passing the health checks.

5. **Check Configuration:** Verify that the application's configuration is correct and that it is able to connect to any required services (e.g., database).
6. **Exec into the Container (if possible):** If the container is running (even briefly), use kubectl exec -it <pod_name> -c <container_name> -- bash to exec into the container and troubleshoot the issue interactively.
7. **Examine Application Code:** If the issue is not immediately apparent, examine the application code for errors or exceptions.
8. **Review Kubernetes Events:** Use kubectl get events to review events related to the pod and identify any issues with the pod's deployment.

26. Question: Design a Docker-based architecture for a highly scalable and resilient web application. Consider factors like load balancing, service discovery, and data persistence.

Expected Answer:

1. **Containerized Application:** Package the web application and its dependencies into Docker containers.
2. **Microservices Architecture:** Decompose the application into smaller, independent microservices. Each microservice should be responsible for a specific function.
3. **Kubernetes Orchestration:** Deploy and manage the containers using Kubernetes. Kubernetes provides features like service discovery, load balancing, auto-scaling, and self-healing.
4. **Load Balancing:** Use a Kubernetes Service of type LoadBalancer or an Ingress controller to distribute traffic across multiple instances of the web application.
5. **Service Discovery:** Use Kubernetes DNS or a service mesh (e.g., Istio, Linkerd) for service discovery.
6. **Data Persistence:** Use persistent volumes (PVs) and persistent volume claims (PVCs) to provide persistent storage for the application's data.
7. **Database Clustering:** Use a database cluster (e.g., PostgreSQL, MySQL, MongoDB) for high availability and scalability.
8. **Caching:** Implement caching mechanisms (e.g., Redis, Memcached) to reduce the load on the database.
9. **Monitoring and Logging:** Implement comprehensive monitoring and logging to track the application's performance and identify potential issues.
10. **Auto-Scaling:** Configure horizontal pod autoscaling (HPA) to automatically scale the number of pods based on CPU or memory utilization.

27. Question: You need to build a Docker image with sensitive data (API keys, passwords). How do you handle this situation securely?

Expected Answer:

1. **Never Hardcode Secrets:** Never include sensitive data directly in the Dockerfile.
2. **Docker Secrets (Preferred):** Use Docker Secrets if running in Swarm mode. These are encrypted at rest and in transit.

3. **Environment Variables with caution:** Pass secrets as environment variables at runtime. Use a mechanism to inject the environment variables securely (e.g., Kubernetes Secrets). Be aware that environment variables are visible in docker inspect, so use caution.
4. **Vault Integration:** Use a secrets management solution like HashiCorp Vault. The container can authenticate to Vault and retrieve secrets at runtime. This is a highly secure approach.
5. **Avoid Storing Secrets on Disk:** Don't write secrets to files on disk inside the container.
6. **Regular Rotation:** Regularly rotate secrets to minimize the risk of compromise.
7. **Build-time Secrets (with ARG and ENV):** While generally discouraged for *sensitive* secrets, if you *must* use a secret during build time (e.g., downloading dependencies from a private repository), use an ARG to pass the secret and then unset it with an ENV after it's used. This still isn't ideal because the ARG value will be in the image history, but it's better than nothing.

28. Question: How can you use Docker to create a development environment that mirrors the production environment as closely as possible?

Expected Answer:

1. **Dockerfile Alignment:** Ensure that the Dockerfile used for the development environment is as close as possible to the Dockerfile used for the production environment.
2. **Docker Compose for Multi-Container Apps:** Use Docker Compose to define the entire development environment, including all dependencies (e.g., database, message queue).
3. **Environment Variables:** Use environment variables to configure the application in both development and production. Use different environment variable values for each environment.
4. **Volumes for Code Sharing:** Use volumes to mount the application's source code into the container, allowing developers to make changes to the code and see the changes reflected in the running application without rebuilding the image.
5. **Network Simulation:** Simulate the production network environment as closely as possible. Use Docker networks to isolate containers and configure network policies to restrict access.
6. **Resource Limits:** Set resource limits (CPU, memory) for the containers in the development environment to match the resource limits in the production environment.
7. **Consistent Base Images:** Use the same base images for both development and production.
8. **Immutable Infrastructure:** Encourage the use of immutable infrastructure practices, where the entire environment is rebuilt for each deployment.

29. Question: You suspect a Docker container is leaking memory. What steps would you take to diagnose the problem and find the root cause?

Expected Answer:

1. **Monitor Memory Usage:** Use docker stats <container_id> to observe the container's memory usage over time. Look for a steadily increasing memory footprint that doesn't decrease.
2. **cAdvisor:** Use cAdvisor, a resource container and machine analyzer, to get more detailed metrics on the container's memory usage.
3. **Heap Dump:** If the application is running in a language with garbage collection (e.g., Java, Node.js), take a heap dump of the application. Analyze the heap dump to identify memory leaks.
4. **Profiling Tools:** Use profiling tools (e.g., perf, jprofiler) to identify memory allocation patterns in the application code.
5. **Application Code Review:** Review the application code for potential memory leaks, such as unclosed connections, large data structures that are not being released, or inefficient caching mechanisms.
6. **Memory Leak Detection Tools:** Use memory leak detection tools specific to the application's language (e.g., Valgrind for C/C++, Memcheck for Java).
7. **strace:** Use strace to trace system calls made by the container. This can help identify excessive memory allocation or other system-level issues.
8. **Restart the Container:** While it doesn't fix the *root cause*, restarting the container will often temporarily alleviate the symptoms, giving you more time to investigate.
9. **Isolate the Problem:** Try to isolate the problem by running a smaller, simpler version of the application in a container to see if the memory leak still occurs. This can help narrow down the scope of the problem.

30. Question: You're designing a system where you need to share a large dataset between multiple Docker containers running on different hosts. What strategies could you use? What are the trade-offs?

Expected Answer:

1. **Network File System (NFS):**
   - **Pros:** Simple to set up, widely supported.
   - **Cons:** Potential performance bottleneck, security concerns if not configured properly, requires network access between hosts. Single point of failure (the NFS server).
   - **Implementation:** Set up an NFS server and mount the shared directory on each host. Then, use bind mounts to mount the NFS directory into the containers.
2.
3. **Object Storage (Amazon S3, Google Cloud Storage, Azure Blob Storage):**
   - **Pros:** Highly scalable, durable, and cost-effective. Object storage is designed for large-scale data storage.
   - **Cons:** Requires network access, data must be uploaded and downloaded, potential latency. Requires application code to interact with the object storage API.
   - **Implementation:** Store the dataset in object storage. Each container can then download the data as needed.

4.
5. **Distributed File System (GlusterFS, Ceph):**
    - **Pros:** Highly scalable, fault-tolerant, and distributed. Designed for large-scale data storage and sharing.
    - **Cons:** More complex to set up and manage, requires significant resources. Can have higher latency than NFS.
    - **Implementation:** Set up a distributed file system cluster and mount the shared directory on each host. Then, use bind mounts to mount the distributed file system directory into the containers.
6.
7. **Shared Volume with Kubernetes:** (If using Kubernetes)
    - **Pros:** Kubernetes-managed, integrates well with container orchestration. Can leverage cloud provider-specific storage solutions.
    - **Cons:** Tied to Kubernetes infrastructure. May require specific storage drivers.
    - **Implementation:** Use a Kubernetes PersistentVolumeClaim to request storage and have the containers mount the same volume.
8.
9. **rsync/scp:**
    - **Pros:** Simple, suitable for smaller datasets and less frequent synchronization.
    - **Cons:** Not suitable for large datasets or frequent updates. Requires setting up SSH keys for automated transfer.
    - **Implementation:** Use a script to regularly synchronize data from a central location to the worker nodes.
10.

Trade-offs to Consider:

- **Performance:** NFS and shared volumes are generally faster for local access, but object storage can be faster for distributed access. Distributed file systems offer a balance between performance and scalability.
- **Scalability:** Object storage and distributed file systems are more scalable than NFS.
- **Durability:** Object storage is highly durable. Distributed file systems can provide good durability with replication.
- **Complexity:** NFS is the simplest to set up, while distributed file systems are the most complex.
- **Cost:** Object storage is generally the most cost-effective for large datasets.

The best strategy depends on the specific requirements of the system, including the size of the dataset, the frequency of updates, the performance requirements, and the available infrastructure. Remember to consider security implications when choosing a solution.