DUMPSARENA

Docker Certified Associate (DCA) Exam

Docker DCA

Version Demo

Total Demo Questions: 28

Total Premium Questions: 281

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, Orchestration	80
Topic 2, Image Creation, Management, and Registry	40
Topic 3, Networking	40
Topic 4, Security	58
Topic 5, Installation and Configuration	30
Topic 6, Storage and Volumes	33
Total	281

QUESTION NO: 1

Will a DTR security scan detect this?

Solution: licenses for known third party binary components

A. Yes

B. No

ANSWER: B

Explanation:

“No” is correct because Docker Trusted Registry (DTR) security scanning is designed to identify known security vulnerabilities in container images, rather than to perform software-license discovery or license-compliance analysis. DTR examines image layers and identifies installed operating-system packages and their versions. It then compares those packages against vulnerability data, reporting applicable CVEs and their severity so that teams can assess and remediate security risk before deploying an image. A license for a known third-party binary component is a legal, procurement, or open-source-compliance attribute, not a vulnerability record that DTR’s vulnerability scanner evaluates. Organizations that need to identify third-party component licenses generally use a software composition analysis or dedicated license-management process alongside container vulnerability scanning. DTR’s scanning workflow and its vulnerability-reporting focus are documented in Mirantis’s DTR security scanning documentation: DTR security scanning. Docker’s image-security guidance likewise describes scanning in terms of detecting known vulnerabilities in image dependencies: Docker security scans.

QUESTION NO: 2

Is this statement correct?

Solution: A Dockerfile provides instructions for building a Docker image

A. Yes

B. No

ANSWER: A

Explanation:

Yes is correct because a Dockerfile is a text file containing the instructions Docker uses to build an image. Each instruction defines part of the image build process, such as selecting a base image with `FROM`, copying application files with `COPY`, installing dependencies with `RUN`, setting configuration through `ENV`, and specifying the default process using `CMD` or `ENTRYPOINT`. When a user runs `docker build` and supplies a build context containing the Dockerfile, Docker processes those instructions to produce an image composed of layers. This makes the image definition reproducible and version-controllable: the same Dockerfile and build context can be used to rebuild the application image consistently. Docker’s documentation explicitly describes a Dockerfile as a text document containing all commands a user could call on the command line to assemble an image, and identifies it as the usual way to create images. See the Dockerfile reference and Docker’s guide on writing a Dockerfile.

QUESTION NO: 3

You want to reduce the size of a production image by compiling an application in one build stage and copying only the resulting binary into the final image.

Which Dockerfile statements support this multi-stage build approach? (Choose 2.)

A. `FROM golang:latest AS builder`

B. `COPY --from=builder /app/server /server`

C. `EXPORT --stage=builder /app/server`

D. `IMPORT builder:/app/server /server`

E. `RUN --from=builder /app/server`

ANSWER: A B

Explanation:

A multi-stage Dockerfile can name a build stage by using `AS`, such as `FROM golang:latest AS builder`. Naming the stage makes it possible to refer to that stage later in the Dockerfile.

The final stage can use `COPY --from=builder` to copy selected build artifacts from the named builder stage. Files that are not copied into the final stage, such as source code, package-manager caches, and compiler toolchains, are not included in the resulting final image. This allows the final image to contain only the runtime dependencies required by the application. See Docker's multi-stage builds documentation and the Dockerfile reference for COPY.

QUESTION NO: 4

An application image runs in multiple environments, with each environment using different certificates and ports. Is this a way to provision configuration to containers at runtime?

Solution.Create a Dockerfile for each environment, specifying ports and Docker secrets for certificates.

A. Yes

B. No

ANSWER: B

Explanation:

No. A Dockerfile defines the steps used to build an image, producing an immutable image artifact from instructions such as `FROM`, `RUN`, `EXPOSE`, and `CMD`. Creating one Dockerfile per environment therefore moves environment-specific values into separate image builds rather than supplying configuration when a container is started. This contradicts the goal of promoting the same application image through development, test, and production while varying only deployment-time configuration.

Ports should be selected at runtime through the container's published-port settings, for example with `docker run -p`, rather than by creating an image for each target environment. Likewise, Docker secrets are provided to services at deployment time in Swarm mode and made available to containers as files under `/run/secrets/`; they are not specified as a Dockerfile build-time configuration mechanism. A suitable runtime approach uses deployment configuration to publish the required port and attaches the applicable secret, allowing the application image itself to remain unchanged across environments. Docker documents the distinction between image-building Dockerfile instructions and runtime container options in the Dockerfile reference. For secret delivery, see Manage sensitive data with Docker secrets.

QUESTION NO: 5

Which statements about Kubernetes ServiceAccounts are correct? (Choose 2.)

A. A Pod can specify a ServiceAccount with spec.serviceAccountName.

B. A ServiceAccount can be granted API permissions through RBAC bindings.

C. A ServiceAccount is a cluster-wide replacement for all Kubernetes users.

D. A ServiceAccount automatically grants cluster-admin permissions to its Pods.

E. A ServiceAccount can be used only by Pods in the kube-system namespace.

ANSWER: A B

Explanation:

A Kubernetes ServiceAccount provides an identity for processes that run in Pods. A Pod can specify the ServiceAccount it should use through `spec.serviceAccountName`. If no ServiceAccount is specified, Kubernetes uses the `default` ServiceAccount in the Pod's namespace.

ServiceAccounts can be granted permissions through Kubernetes RBAC objects, such as Roles, ClusterRoles, RoleBindings, and ClusterRoleBindings. The permissions assigned to the ServiceAccount determine which Kubernetes API operations a workload may perform. A ServiceAccount is namespace-scoped, although it can be referenced by a cluster-wide RBAC binding. See the Kubernetes documentation for ServiceAccounts and RBAC authorization.

QUESTION NO: 6

Which of the following commands wifi automatically create a volume when a container is started?

A. 'docker container run --name nginxtest --volumes=/app nginx'

B. 'docker container run --name nginxtest -v /app:mount nginx'

C. 'docker container run --name nginxtest --volumes myvol:/app:new nginx'

D. 'docker container run --name nginxtest -v myvol:/app nginx'

ANSWER: D

Explanation:

`docker container run --name nginxtest -v myvol:/app nginx` is correct because it uses the `-v` (or `--volume`) mount syntax to mount the named volume `myvol` at `/app` in the container. Docker checks whether the named volume exists when creating the container. If `myvol` does not already exist, Docker creates it automatically using the local volume driver by default, then attaches it to the container at the specified target path. The volume is managed independently of the container lifecycle, so its data can persist after the container is removed and the same named volume can later be attached to another container. This is a standard and useful way to provide persistent, Docker-managed storage without first issuing a separate `docker volume create myvol` command. Docker documents that a volume is created automatically when it is specified by name in a `--mount` or `-v` declaration and does not already exist. See Docker's Volumes documentation and the docker container run reference.

QUESTION NO: 7

Are these conditions sufficient for Kubernetes to dynamically provision a persistentVolume, assuming there are no limitations on the amount and type of available external storage?

Solution: A default storageClass is specified, and subsequently a persistentVolumeClaim is created.

A. Yes

B. No

ANSWER: A

Explanation:

Yes is correct. Kubernetes dynamic volume provisioning is driven by a PersistentVolumeClaim together with a StorageClass that identifies a provisioner. When a PersistentVolumeClaim is created without an explicitly specified storage class, Kubernetes can assign the cluster's default StorageClass to that claim. The StorageClass provisioner then creates an appropriate PersistentVolume backed by the configured external storage system, provided the requested access mode, capacity, and other claim requirements can be fulfilled.

Under the question's assumption that suitable external storage is available without capacity or storage-type constraints, specifying a default StorageClass and then creating a PersistentVolumeClaim supplies the normal trigger for dynamic provisioning. The resulting volume is created in response to the claim rather than requiring an administrator to pre-create a PersistentVolume. Kubernetes documents that a default StorageClass is used for PersistentVolumeClaims that do not request a particular storage class, and that dynamic provisioning creates storage volumes on demand based on StorageClass configuration. See the Kubernetes documentation on StorageClasses and dynamic volume provisioning.

QUESTION NO: 8

You need to control where a Docker Swarm service runs. Which placement settings can be specified when creating or updating a service? (Choose 2.)

A. Placement constraints

B. Placement preferences

C. Node tolerations

D. Pod affinity rules

E. Replica selectors

ANSWER: A B

Explanation:

A placement constraint restricts service tasks to nodes that satisfy an expression. For example, `--constraint-add 'node.labels.environment == production'` causes the scheduler to consider only nodes with the matching label. Constraints are appropriate when workloads must run only on a particular operating system, node role, environment, or hardware class.

A placement preference expresses a scheduling preference rather than a strict requirement. For example, `--placement-pref-add spread=node.labels.zone` asks Swarm to spread tasks across values of a node label where possible. A preference does not make a node ineligible in the way that a constraint does. See Docker's service placement documentation and the docker service create reference.

QUESTION NO: 9

Does this describe the role of Control Groups (cgroups) when used with a Docker container?

Solution: user authorization to the Docker API

A. Yes

B. No

ANSWER: B

Explanation:

No is correct. In Docker, control groups (cgroups) are a Linux kernel feature used to organize processes and apply resource-management policies to them. Docker uses cgroups to constrain and account for container resource consumption, including CPU time, memory, block I/O, and process limits. For example, Docker can configure cgroup settings through runtime options such as memory limits, CPU limits, and PID limits, helping ensure that a container cannot consume more than its allocated share of host resources. This resource isolation is a fundamental part of container runtime behavior, alongside namespaces, which provide isolation of system views such as processes and networking. User authorization to the Docker API is instead an access-control concern: it determines which authenticated users or clients may perform Docker daemon operations. Docker documents authorization through access-authorization plugins, which receive requests after authentication and determine whether the request should be allowed. Therefore, cgroups do not provide user authorization

to the Docker API; their Docker role is controlling and accounting for container resource usage. See the Docker documentation on runtime metrics and cgroups and authorization plugins.

QUESTION NO: 10

You are configuring a rolling update for a replicated Docker Swarm service. Which settings can control the update behavior? (Choose 2.)

A. --update-parallelism

B. --update-delay

C. --replica-history-limit

D. --endpoint-mode

E. --swarm-quorum

ANSWER: A B

Explanation:

The `--update-parallelism` option controls how many service tasks are updated at the same time. Setting it to one performs the update one task at a time, while a larger value permits more concurrent task replacements. This can limit the impact of a faulty image or configuration change.

The `--update-delay` option defines the time Docker waits between groups of task updates. It can be used with update parallelism to create a controlled rollout and allow time to observe each update batch. Docker also provides update monitoring, failure action, and order settings for additional rollout control. See the docker service update reference and Docker's rolling update documentation.

QUESTION NO: 11

You need to provide a password to a Docker Swarm service without including it in the image or as an environment variable.

Which statements about Docker secrets are correct? (Choose 2.)

A. Secrets are available only to services granted access to them.

B. Secrets are mounted under `/run/secrets` by default.

C. Secrets are automatically available to every service in the swarm.

D. Secrets must be stored in the Dockerfile with an `ENV` instruction.

E. Secrets are persisted in a container image layer.

ANSWER: A B

Explanation:

Docker secrets are intended for sensitive data such as passwords, TLS private keys, and API credentials. A secret is created in the Swarm and then explicitly granted to a service. Docker makes the secret available only to containers in service tasks that have been granted access to it.

Inside a Linux service container, a secret is mounted as an in-memory file under `/run/secrets` by default. Secrets are not supplied as ordinary environment variables, which helps avoid accidental exposure through environment inspection or process listings. Docker documents secret creation, service access, and the default mount location in its Manage sensitive data with Docker secrets documentation.

QUESTION NO: 12

An application image runs in multiple environments, with each environment using different certificates and ports.

Is this a way to provision configuration to containers at runtime?

Solution: Create images that contain the specific configuration for every environment.

A. Yes

B. No

ANSWER: B

Explanation:

No is correct. Creating separate images that embed environment-specific certificates and port settings does not provision configuration at container runtime. Instead, configuration is baked into each image during the image build process. A container image should generally remain portable so that the same application artifact can be promoted through development, testing, and production environments without requiring a rebuild solely because its deployment configuration differs. Runtime configuration is supplied when the container is started or deployed, for example through environment variables, mounted configuration files, Docker configs or secrets, command arguments, or orchestration-platform configuration mechanisms. This separation lets operators provide the appropriate ports and certificates for the target environment while retaining one consistent application image. It also allows configuration or secret rotation without creating a new application image, subject to the deployment mechanism used. Docker documents that environment variables can be set for containers at runtime using `docker run --env` and through Compose configuration. For sensitive certificate material and similar data, Docker secrets provide a runtime delivery mechanism for services and are mounted into containers rather than embedded in an image. See Docker Docs: environment variables and Docker Docs: manage sensitive data with Docker secrets.

QUESTION NO: 13

Will this configuration achieve fault tolerance for managers in a swarm? Solution: an odd number of manager nodes, totaling more than two

A. Yes

B. No

ANSWER: A

Explanation:

Yes is correct. Docker Swarm manager nodes use the Raft consensus algorithm to maintain a consistent, replicated copy of the swarm's control-plane state. Raft requires a quorum, meaning that more than half of the manager nodes must be available to process management operations and make changes to the swarm. Deploying an odd number of managers greater than two—such as three, five, or seven—provides practical manager fault tolerance while avoiding an extra node that would not increase the number of failures the swarm can tolerate. For example, a three-manager swarm retains quorum with two available managers and can continue operating if one manager fails. A five-manager swarm retains quorum with three available managers and can tolerate two manager failures. Docker recommends an odd number of manager nodes for this reason and advises using three or five managers for most production environments. Manager nodes should also be distributed across failure domains, such as availability zones or physical hosts, so that a single infrastructure failure is less likely to remove enough managers to lose quorum. See Docker's Swarm administration guide and its Raft consensus documentation.

QUESTION NO: 14

Which statements about Docker bind mounts are correct? (Choose 2.)

A. A bind mount uses a file or directory from the Docker daemon host.

B. A bind mount is writable by default unless configured as read-only.

C. A bind mount is always managed under Docker's volume storage directory.

D. A bind mount can be attached only to Swarm services.

E. A bind mount is copied into the image during docker build.

ANSWER: A B

Explanation:

Bind mounts use a path on the Docker daemon host as the source of the mount. The host path is mounted directly into the container, so changes made through the mounted path can be visible on both the host and the container. This makes bind mounts useful when a container needs access to host-managed files, such as source code during development or a host configuration file.

Bind mounts are writable by default. A read-only bind mount can be configured explicitly with `:ro` in `--volume` syntax or with `readonly` in `--mount` syntax. Unlike Docker-managed volumes, bind mounts depend on the filesystem layout of the specific daemon host. See Docker bind mounts documentation.

QUESTION NO: 15

Which of the following modes can be used for service discovery of a Docker swarm service (Pick 2 correct answers)

A. Virtual IP (VIP) with --endpoint-mode vip

B. Overlay with --endpoint-mode overlay

C. DNS Round-Robin with --endpoint-mode dnsrr

D. Ingress with --endpoint-mode ingress

E. Network Address Translation(NAT) with --endpoint-mode nat

ANSWER: A C

Explanation:

Docker Swarm supports two service-discovery endpoint modes: Virtual IP (VIP) and DNS Round-Robin (DNSRR). With `--endpoint-mode vip`, which is the default mode, Docker assigns the service a virtual IP address. Requests sent to the service name resolve to that VIP, and the swarm's internal load balancer distributes connections among the healthy service tasks. This lets clients use one stable service name and address while Docker handles task selection.

With `--endpoint-mode dnsrr`, DNS resolution for the service name returns the individual IP addresses of the running task containers instead of a single virtual IP. The client is then responsible for selecting an address and connecting to a task. DNSRR is especially appropriate when an external load balancer needs to discover task addresses directly, or when application-level client-side load balancing is required. Docker documents both values as the accepted endpoint-mode settings for Swarm services. See the docker service create endpoint-mode reference and Docker's Swarm service-discovery networking documentation.

QUESTION NO: 16

You are pulling images from a Docker Trusted Registry installation configured to use self-signed certificates, and this error appears:

'x509: certificate signed by unknown authority'.

You already downloaded the Docker Trusted Registry certificate authority certificate from https://dtr.example.com/ca.

How do you trust it? (Select two.)

A. Place the certificate in '/etc/docker/dtr/dtr.example.com.crt' and restart the Docker daemon on all cluster nodes.

B. Place the certificate in your OS certificate path, trust the certificate system-wide, and restart the Docker daemon across all cluster nodes.

C. Pass '-trust-certificate ca.crt' to the Docker client.

D. Pass --insecure-registry to the Docker client.

E. Place the certificate in '/etc/docker/certs.d/dtr.example.com/ca.crt' on all cluster nodes.

ANSWER: B E

Explanation:

The Docker Trusted Registry CA certificate must be installed in a trust store that the Docker daemon uses when it establishes the TLS connection to the registry. On Linux hosts, Docker supports a registry-specific CA directory. Installing the CA as `/etc/docker/certs.d/dtr.example.com/ca.crt` causes the daemon to trust certificates issued by that CA when contacting `dtr.example.com`. The directory name must exactly match the registry hostname (and include a port if a non-default port is used). This installation must be performed on every node that will pull from the registry.

Alternatively, the DTR CA can be added to the operating system's trusted root certificate store. Once the system trusts the CA and the Docker daemon is restarted, Docker can use that trusted authority for registry TLS verification. This is especially useful where host-wide certificate trust is managed centrally. Both approaches preserve TLS certificate validation and allow Docker to verify the registry identity rather than bypassing security checks. Docker documents registry-specific CA placement in its daemon certificate documentation; Docker Trusted Registry deployment guidance also covers distributing and trusting the DTR CA certificate: Mirantis DTR documentation.

QUESTION NO: 17

You are pulling images from a Docker Trusted Registry installation

configured to use self-signed certificates, and this error appears:

`x509: certificate signed by unknown authority.

You already downloaded the Docker Trusted Registry certificate authority

certificate from https://dtr.example.com/ca.

How do you trust it? (Select two.)

A. Pass '-trust-certificate ca.crt to the Docker client.

B. Place the CA certificate at /etc/docker/certs.d/dtr.example.com/ca.crt and restart the Docker daemon on all cluster nodes.

C. Place the certificate in /etc/docker/dtr/dtr.example.com.crt on all cluster nodes.

D. Pass -- insecure-registry to the Docker client.

E. Add the certificate to the OS certificate store system-wide, and restart the Docker daemon across all cluster nodes.

ANSWER: B E

Explanation:

Trusting the DTR certificate authority requires installing its CA certificate where the Docker daemon or the host operating system can use it to validate the registry's TLS certificate. For Docker Engine on Linux, the registry-specific approach is to create a directory named for the registry host and place the downloaded CA certificate at `/etc/docker/certs.d/dtr.example.com/ca.crt`. Docker uses certificates in this directory when connecting to that registry. Because every node that pulls images must establish and validate its own TLS connection, the certificate installation and Docker daemon restart must be performed on every cluster node.

A second supported approach is to add the DTR CA certificate to the host operating system's trusted root certificate store system-wide. This allows software using the host trust store, including Docker in applicable configurations, to recognize the self-signed or privately issued registry certificate as trusted. After changing trust material, restarting the Docker daemon ensures it reloads the available certificate authorities. Docker documents both the registry-specific certificate directory and using the host OS trust store for custom CAs. See Docker Engine: Use CA certificates with Docker and Docker Registry deployment and certificate configuration.

QUESTION NO: 18

Can this set of commands identify the published port(s) for a container?

Solution: docker container inspect', 'docker port'

A. Yes

B. No

ANSWER: A

Explanation:

Yes. Both `docker container inspect` and `docker port` can identify a container's published port mappings. `docker port CONTAINER` is specifically intended to list the port mappings for a running container in a concise format, showing the container port/protocol and the host address and port to which it is published. For example, `docker port web` may report a mapping such as `80/tcp -> 0.0.0.0:8080`. `docker container inspect CONTAINER` provides the container's detailed low-level configuration and state as JSON. Its network settings include port-binding data, so users can retrieve published ports directly or use formatting to output only the relevant mapping information, such as `docker container inspect --format '{{json .NetworkSettings.Ports}}' web`. These commands are useful after a container has been created because published host ports can be explicitly selected or dynamically assigned by Docker. Docker documents `docker container port` as listing port mappings or a specific mapping for a container, while `docker container inspect` exposes detailed object information that includes networking configuration. See the Docker container port reference and the Docker container inspect reference.

QUESTION NO: 19

Is this a supported user authentication method for Universal Control Plane?

Solution.LDAP

A. Yes

B. No

ANSWER: A

Explanation:

Yes is correct. Universal Control Plane supports LDAP as an external authentication method, allowing it to authenticate users against an organization's existing LDAP directory rather than requiring separately managed local UCP credentials. LDAP integration is configured by supplying the directory connection and search details that UCP needs, such as the LDAP server URL, base distinguished name, bind credentials where required, and user and group search settings. Once configured, UCP can use directory identities and group membership as inputs to its authorization model, including role-based access control assignments. This centralizes identity management and lets administrators retain the organization's established account lifecycle and directory structure while controlling access to Docker resources through UCP. LDAP support is documented in the UCP administration guidance for configuring authentication backends. See the Mirantis UCP LDAP authentication documentation and the UCP authentication configuration overview.

QUESTION NO: 20

Which statements about an internal Docker bridge network are correct? (Choose 2.)

A. It can be created with docker network create --internal.

B. Containers on the same internal network can communicate with each other.

C. It automatically publishes all container ports on the host.

D. It can be used only with the host network driver.

E. It prevents Docker DNS resolution between attached containers.

ANSWER: A B

Explanation:

An internal network can be created by using the `--internal` option with `docker network create`. Docker restricts external access for containers attached to an internal network by not providing the normal default route that would allow traffic to leave through the host's external network interfaces. This is useful for isolating backend components such as databases.

Containers attached to the same internal user-defined bridge network can still communicate with each other using Docker's embedded DNS and container networking. An internal network controls external connectivity; it does not prevent communication among connected containers. See the docker network create reference and Docker's bridge network documentation.

QUESTION NO: 21

Is this a supported user authentication method for Universal Control Plane?

Solution: PAM

A. Yes

B. No

ANSWER: B

Explanation:

No is correct. Pluggable Authentication Modules (PAM) are a Linux host-level authentication framework, typically used by operating-system services to validate local users or to connect those services to configured identity mechanisms. PAM is not an authentication backend that Universal Control Plane (now Mirantis Kubernetes Engine in current product naming) exposes as a supported user-authentication integration. UCP's supported external identity configuration is performed through its documented directory or federated-identity integrations, rather than by delegating control-plane login directly to a host's PAM stack. This distinction matters because UCP authentication must consistently identify users and groups across managers, provide identity information for role-based access control, and remain independent of the configuration of an individual node's operating system. Administrators should therefore use the product's supported external-authentication configuration when integrating enterprise identities, and manage any locally defined UCP accounts through the control plane itself. Mirantis documentation describes configuring supported LDAP authentication for MKE/UCP at Configure LDAP authentication and documents SAML-based federation at Configure SAML authentication.

QUESTION NO: 22

What is the difference between the ADD and COPY Dockerfile instructions? (Select two.)

A. ADD supports remote URL handling while COPY does not.

B. COPY supports compression format handling while ADD does not.

C. COPY supports regular expression handling while ADD does not.

D. ADD supports regular expression handling while COPY does not.

E. ADD supports compression format handling while COPY does not.

ANSWER: A E

Explanation:

`ADD supports remote URL handling while COPY does not.` is correct because `ADD` can use a remote URL as its source; Docker downloads the referenced file into the image filesystem during the build. This capability is specific to `ADD`. For reproducible builds, Docker generally recommends downloading remote artifacts separately and then using `COPY`, but URL sources remain an `ADD` feature.

`ADD supports automatic extraction of local tar archives while COPY does not.` is also correct. When the source for `ADD` is a local tar archive in a recognized compression format, Docker extracts its contents into the destination directory instead of copying the archive as a single file. This makes `ADD` useful when archive extraction is explicitly required. The Dockerfile reference documents both the remote-URL and local-tar extraction behaviors, while Docker's build guidance recommends choosing `COPY` when those additional `ADD` capabilities are not needed. See the Dockerfile ADD reference and Docker build best practices for ADD or COPY.

QUESTION NO: 23

What is the difference between the ADD and COPY dockerfile instructions? (choosen 2)

A. ADD supports compression format handling while COPY does not.

B. COPY supports regular expression handling while ADD does not.

C. COPY supports compression format handling while ADD does not.

D. ADD support remote URL handling while COPY does not.

E. ADD supports regular expression handling while COPY does not.

ANSWER: A D

Explanation:

`ADD supports compression format handling while COPY does not.` is correct because, when an `ADD` source is a local tar archive in a recognized compressed or uncompressed tar format, Docker automatically extracts its contents into the destination directory in the image. This auto-extraction behavior is specific to local archives supplied to `ADD`; `COPY` transfers files and directories as files, without unpacking an archive. Docker's Dockerfile reference documents this behavior and notes that archive recognition is based on content rather than the filename extension.

`ADD support remote URL handling while COPY does not.` is also correct in the traditional Dockerfile distinction: `ADD` can use an HTTP or HTTPS URL as its source, downloading the referenced resource into the image filesystem during the build. A URL source is copied as a downloaded file and is not automatically extracted merely because its name suggests an archive format. `COPY` is intended for copying files from the build context, another build stage, or a named context; it does not accept a remote HTTP/HTTPS URL as a source. Docker recommends preferring `COPY` for ordinary local-file transfers because its behavior is more explicit, while reserving `ADD` for cases requiring its additional source-handling features. See the Dockerfile ADD reference and the Dockerfile COPY reference.

QUESTION NO: 24

A firewall is being configured between Docker Swarm nodes that participate in an overlay network.

Which ports must be permitted for Swarm overlay network operation? (Choose 2.)

A. `4789/UDP`

B. `7946/TCP and 7946/UDP`

C. `2376/UDP`

D. `6443/TCP`

E. `80/UDP`

ANSWER: A B

Explanation:

Docker Swarm uses port `4789/UDP` for overlay-network data traffic. This traffic is carried with VXLAN, allowing containers attached to the same overlay network to communicate across participating Docker hosts.

Swarm nodes also use port `7946/TCP` and `7946/UDP` for container-network discovery and communication. These ports are distinct from `2377/TCP`, which is used for swarm-management communication between managers and nodes joining or participating in the swarm. Docker documents the required ports in its Swarm mode tutorial and overlay network driver documentation.

QUESTION NO: 25

What is used by the kernel to Isolate resources when running Docker containers?

A. Namespaces

B. Overlay networks

C. Volumes

D. Control groups (also know as cgroups)

ANSWER: A

Explanation:

Namespaces are the Linux kernel feature Docker uses to provide isolation between containers and between a container and the host. When Docker creates a container, it uses namespaces to give that container separate views of key operating-system resources. For example, a process namespace isolates process IDs, a network namespace supplies an independent network stack, a mount namespace isolates filesystem mount points, and a user namespace can map user and group IDs separately from the host. As a result, processes running in one container generally see only the processes, interfaces, mounts, and identities assigned to their own namespace rather than those belonging to other containers or the host system. This kernel-level separation is a foundational part of the container isolation model; a container is not a virtual machine with its own kernel. Docker documentation describes namespaces as the technology that provides isolation, while control groups are used to constrain and account for resource consumption such as CPU and memory. See Docker's containers concepts documentation and the Docker Engine security documentation for further details.

QUESTION NO: 26

Will this Linux kernel facility limit a Docker container's access to host resources, such as CPU or memory?

Solution: seccomp

A. Yes

B. No

ANSWER: B

Explanation:

No is correct. Seccomp (secure computing mode) is a Linux kernel security facility that restricts the system calls a process is permitted to make. Docker uses a default seccomp profile to reduce a container's available kernel attack surface by blocking selected system calls that are not commonly needed by ordinary container workloads. This is a form of syscall filtering and process isolation, rather than a mechanism for allocating, reserving, or limiting hardware resources.

Docker applies CPU, memory, and related host-resource constraints through Linux control groups (cgroups). For example, Docker's resource-management flags can set a memory limit, CPU quota, CPU shares, or a cpuset allocation; Docker translates those settings into cgroup controls enforced by the Linux kernel. Consequently, seccomp can help constrain what a process can ask the kernel to do, but it does not itself impose CPU or memory consumption limits. Docker documentation describes its default seccomp profile as an allowlist of permitted system calls, while its resource-constraint documentation identifies cgroups as the kernel feature used for limiting container resources. See Docker seccomp security documentation and Docker resource constraints documentation.

QUESTION NO: 27

Which Docker restart policy restarts a container after it exits, except when the container was explicitly stopped by an administrator?

A. `no`

B. `on-failure`

C. `always`

D. `unless-stopped`

ANSWER: D

Explanation:

`unless-stopped` is correct. This restart policy causes Docker to restart the container when it exits or when the Docker daemon restarts, unless the container was manually stopped. A manually stopped container remains stopped after a daemon restart until an administrator starts it again.

This differs from `always`, which restarts a container after the daemon restarts even if it had previously been manually stopped. The `on-failure` policy restarts only when a container exits with a nonzero status, while `no` disables automatic restart behavior. See the Docker Start containers automatically documentation.

QUESTION NO: 28

You have deployed a service to swarm. Which command uses the Docker CLI to set the number of tasks of the services to 5? (choose 2)

A. 'docker service update --replicas=5 '

B. 'docker replica update =5'

C. 'docker update service =5'

D. 'docker service replicas =5'

E. docker service scale =5

ANSWER: A E

Explanation:

Both `docker service update --replicas=5 <service-id>` and `docker service scale <service-id>=5` set the desired replica count for an existing replicated Swarm service to five tasks. The `docker service update` command modifies a service's configuration; its `--replicas` flag changes the desired number of replicas. After the update, the Swarm manager reconciles actual state with desired state by creating or removing service tasks as necessary, subject to the service's update settings and available cluster resources. Docker documents `--replicas` as a supported `docker service update` option for replicated services.

The `docker service scale` command is the direct scaling form of the same operation. Its argument uses the required `SERVICE=REPLICAS` syntax, so specifying the service identifier followed by `=5` requests five replicas. This command can also scale multiple services in one invocation, but a single service assignment is sufficient here. These commands apply to replicated services; a global service instead runs one task on each eligible node. See the Docker service update reference and the Docker service scale reference.