Developing Applications with vSphere Integrated Containers

VMware vSphere Integrated Containers 1.4.x

Table of Contents

Developing Applications with vSphere Integrated Containers

Developing Container Applications with vSphere Integrated Containers provides information about how to use VMware vSphere® Integrated Containers™ virtual container hosts (VCHs) as the endpoints for Docker container application development.

Product version: 1.4

This documentation applies to all 1.4.x releases.

Intended Audience

This information is intended for container application developers whose development environment uses vSphere Integrated Containers. Knowledge of container technology and Docker is assumed.

In particular, these topics explain how developing with vSphere Integrated Containers differs from development in a regular Docker environment, and often enhances it.

VMware, Inc. 3401 Hillview Ave. Palo Alto, CA 94304

www.vmware.com

Supported Docker Commands

vSphere Integrated Containers Engine 1.4 supports Docker client 1.13.0. The supported version of the Docker API is 1.25.

- Docker Management Commands
- Image Commands
- Container Commands
- Hub and Registry Commands
- Network and Connectivity Commands
- Shared Data Volume Commands
- Docker Compose Commands
- Swarm Commands

Docker Management Commands

Command	Docker Reference	Supported
`dockerd`	Launch the Docker daemon	Not applicable. This construct does not exist in vSphere Integrated Containers
`info`	Docker system information	Yes, since 1.0. Provides Docker-specific data, basic capacity information, lists configured volume stores, and virtual container host information. Does not reveal vSphere datastore paths that might contain sensitive vSphere information.
`inspect`	Inspect a container or image	Yes, since 1.0. Includes information about the container network.
`version`	Docker version information	Yes, since 1.0

Image Commands

Command	Docker Reference	Supported
`build`	Build an image from a Dockerfile	No
`commit`	Create a new image from a container's changes	Yes, since 1.2. You can only run `docker commit` on stopped containers.
`history`	Show the history of an image	No
`images`	Images	Yes, since 1.0. Supports `--filter`, `--no-trunc`, and `--quiet`

`import`	Import the contents from a tarball to create a filesystem image	No
`load`	Load an image from a tar archive or STDIN	No
`rmi`	Remove a Docker image	Yes, since 1.0
`save`	Save images	No
`tag`	Tag an image into a repository	Yes, since 1.0

Container Commands

Command	Docker Reference	Supported
`attach`	Attach to a container	Yes, since 1.0
`container list`	List Containers	Yes, since 1.0
`container resize`	Resize a container	Yes, since 1.0
`cp`	Copy files or folders between a container and the local filesystem	Yes, since 1.2. You cannot copy to an NFS volume that is not in use by a running container. You cannot copy from an unstarted container that uses NFS volumes.
`create`	Create a container	Yes, since 1.0. `--cpuset-cpus` in Docker specifies CPUs the container is allowed to use during execution (0-3, 0,1). In vSphere Integrated Containers Engine, this parameter specifies the number of virtual CPUs to allocate to the container VM. Minimum CPU count is 1, maximum is unlimited. Default is 2. `--ip` allows you to set a static IP on the container. By default, the virtual container host manages the container IP. Minimum value for `--memory` is 512MB, maximum unlimited. If unspecified, the default is 2GB. Supports the `--attach`, `--cidfile`, `--cpuset-cpus`, `--entrypoint`, `--env`, `--env-file`, `--help`, `--interactive`, `--ip`, `--link`, `--memory`, `--name`, `--net`, `--net-alias`, `--publish`, `--rm`, `--stop-signal`, `--stop-timeout`, `--tty`, `--user`, `--volume`, and `--workdir` options.
`diff`	Inspect changes on a container's filesystem	Yes, since 1.2
`events`	Get real time events from the server	Yes, since 1.0. Supports passive Docker events for containers and images. Does not yet support events for volumes or networks.

`exec`	Run a command in a running container	Yes, since 1.2
`export`	Export a container	No
`kill`	Kill a running container	Yes, since 1.0. Docker must wait for the container to shut down.
`logs`	Get container logs	Yes, since 1.0. Supports `--since` and `--timestamps` since 1.2.
`pause`	Pause processes in a container	No
`port`	Obtain port data	Yes, since 1.0. Displays port mapping data. Supports mapping a random host port to the container when the host port is not specified.
`ps`	Show running containers	Yes, since 1.0. Supports the `-a/--all` , `-f/--filter` , `--no-trunc` , and `-q/--quiet` options. Filtering by network name is supported, but filtering by network ID is not supported.
`rename`	Rename a container	Yes, since 1.1. Name resolution for renamed running containers is not supported, but if you restart the container the new name is resolved.
`restart`	Restart a container	Yes, since 1.0
`rm`	Remove a container	Yes, since 1.0. Supports the `--force` option and the `name` parameter. To view volumes attached to a container that is removed, use `docker volume ls` and `docker volume inspect <id>` . If you continually invoke `docker create` to make more anonymous volumes, those volumes are left behind after each subsequent removal of that container. Supports `docker rm -v` since 1.3. Running the command removes the container and any anonymous volumes joined to that container. If an anonymous volume is in use by another container, it is not removed. Named volumes that you specify by name in the create/run command are not deleted.
`run`	Run a command in a new container	Yes, since 1.0. Supports mapping a random host port to the container when the host port is not specified. Supports running images from private and custom registries. `docker run -h` is supported since 1.3.0. You can specify a container network by using the `--container-network` option when you deploy a virtual container host. Supports the `--attach` , `--cidfile` , `--cpuset-cpus` , `--detach` , `--detach-keys` , `--entrypoint` , `--env` , `--env-file` , `--help` , `--interactive` , `--ip` , `--link` , `--memory` , `--name` , `--net` , `--net-alias` , `--publish` , `--rm` , `--stop-signal` , `--stop-timeout` , `--tty` , `--user` , `--volume` , and `--workdir` options.
`start`	Start a container	Yes, since 1.0. Supports the `--attach` and `--interactive` options.
	Get container stats	Yes. Provides statistics about CPU and memory usage since 1.1. Provides

	based on resource usage	statistics about network or disk usage since 1.2.
`stop`	Stop a container	Yes, since 1.0. Attempts to politely stop the container. If that fails, powers down the VM.
`top`	Display the running processes of a container	No
`unpause`	Unpause processes within a container	No
`update`	Update a container	No
`wait`	Wait for a container	Yes, since 1.0

Hub and Registry Commands

Command	Docker Reference	Supported
`login`	Log into a registry	Yes, since 1.0
`logout`	Log out from a registry	Yes, since 1.0
`pull`	Pull an image or repository from a registry	Yes, since 1.0. Supports pulling from secure or insecure public and private registries.
`push`	Push an image or a repository to a registry	No
`search`	Search the Docker hub for images	No

Network and Connectivity Commands

For more information about network operations with vSphere Integrated Containers Engine, see Container Networking with vSphere Integrated Containers Engine.

Command	Docker Reference	Supported
`network connect`	Connect to a network	Yes, since 1.0. Not supported for running containers. You can specify the `--ip` option to assign a static IP address to a container. If you do not specify `--ip` , the VCH assigns an IP address from the provided range of addresses for the container network. Using the `--ip` option on container networks with DHCP enabled is not supported.

		container networks with DHCP enabled is not supported.
`network create`	Create a network	Yes, since 1.1. See the use case to connect a container to an external network in Container Networking with vSphere Integrated Containers Engine. Bridge is also supported.
`network disconnect`	Disconnect a network	No
`network inspect`	Inspect a network	Yes, since 1.0
`network ls`	List networks/	Yes, since 1.0
`network rm`	Remove a network	Yes, since 1.0. Network name and network ID are supported.

Shared Data Volume Commands

For more information about volume operations with vSphere Integrated Containers Engine, see Using Volumes with vSphere Integrated Containers Engine.

Command	Docker Reference	Supported
`volume create`	Create a volume	Yes, since 1.0. Supports the `--opt Capacity` and `--opt VolumeStore` options, and ignores any other options that you might specify. Currently only supports `ext4` file systems for volume stores.
`volume inspect`	Information about a volume	Yes, since 1.0
`volume ls`	List volumes	Yes, since 1.0
`volume rm`	Remove or delete a volume	Yes, since 1.0

Docker Compose Commands

vSphere Integrated Containers Engine 1.4 supports Docker Compose version 1.11.2.

For more information about using Docker Compose with vSphere Integrated Containers Engine, see Creating a Containerized Application with vSphere Integrated Containers Engine.

For information about Docker Compose file support, see Supported Docker Compose File Options.

Command	Docker Reference	Supported
`build`	Build or rebuild service	No. Depends on `docker build`.
	Generate a Distributed	

	the Compose file	
`config`	Validate and view the compose file	Yes, since 1.0
`create`	Create services	Yes, since 1.0
`down`	Stop and remove containers, networks, images, and volumes	Yes, since 1.0
`events`	Receive real time events from containers	Yes, since 1.0. Supports passive Docker events for containers and images. Does not yet support events for volumes or networks.
`exec`	Run commands in services	No. Depends on `docker exec` .
`help`	Get help on a command	Yes, since 1.0
`kill`	Kill containers	No, but `docker kill` works.
`logs`	View output from containers	Yes, since 1.0
`pause`	Pause services	No. Depends on `docker pause` .
`port`	Print the public port for a port binding	Yes, since 1.0
`ps`	List containers	Yes, since 1.0
`pull`	Pulls service images	Yes, since 1.0
`push`	Pushes images for service	No. Depends on `docker push`
`restart`	Restart services	Yes, since 1.0
`rm`	Remove stopped containers	Yes, since 1.0
`run`	Run a one-off command	Yes, since 1.0
`scale`	Set number of containers for a service	Yes, since 1.0
`start`	Start services	Yes, since 1.0
`stop`	Stop services	Yes, since 1.0
`unpause`	Unpause services	No. Depends on `docker unpause` .
`up`	Create and start containers	Yes, since 1.1
`version`	Show Docker Compose version information	Yes, since 1.0

Swarm Commands

This version of vSphere Integrated Containers Engine does not directly support Docker Swarm. However, you can use the `dch-photon` Docker Engine to instantiate a Docker swarm for use with vSphere Integrated Containers.

NOTE: Using `dch-photon` to instantiate Docker swarm is not officially supported.

NOTE: Using `dch-photon` to instantiate Docker swarm is not officially supported.

Supported Docker Compose File Options

vSphere Integrated Containers Engine 1.4 supports Docker Compose file version 2, 2.1, and 2.2.

This topic provides information about the Docker Compose file options that vSphere Integrated Containers Engine 1.4 supports.

- Service Configuration Options
- Volume Configuration Options
- Network Configuration Options

Service Configuration Options

Option	Compose File Reference	Supported
`build`	Options applied at build time	No
`cap_add` , `cap_drop`	Add or drop container capabilities	No. Depends on `docker run --cap-add` and `docker run --cap-drop`
`command`	Override the default command	Yes
`cgroup_parent`	Specify an optional parent `cgroup` for the container.	No; need docker run --cgrop_parent
`container_name`	Specify a custom container name	Yes
`devices`	List of device mappings	No. Depends on `docker create --device` .
`depends_on`	Express dependency between services	Yes
`dns`	Custom DNS servers	Yes
`dns_search`	Custom DNS search domains	No. Depends on `docker run --dns-search` .
`tmpfs`	Mount a temporary file system inside the container	No. Depends on `docker run --tmpfs` .
`entrypoint`	Override the default entry point	No. Depends on `docker run --entrypoint` .
`env_file`	Add environment variables from a file	Yes
`environment`	Add environment variables	Yes
`expose`	Expose ports without publishing them to the host machine	No. Depends on `docker run --expose` .
`extends`	Extend another service	Yes
`external_links`	Link to containers started outside this YML	Yes
`extra_hosts`	Add hostname mappings	No. Depends on

`extra_hosts`	Add hostname mappings	No. Depends on `docker run --add-host` .
`group_add`	Specify additional groups for the user inside the container	Yes
`healthcheck`	Check container health	No. Depends on `docker run --health-cmd` .
`image`	Specify container image	Yes
`isolation`	Specify isolation technology	No. Depends on `docker run --isolation` .
`labels`	Add metadata by using labels	Yes
`links`	Link to containers in another service	Yes
`logging` , `log_driver` , `log_opt`	Logging configuration	No. Depends on `docker run --log-driver` and `--log-opt` .
`net`	Network mode (version 1)	Yes
`network_mode`	Network mode (version 2)	Yes
`networks`	Networks to join	Yes
`aliases`	Aliases for this service on the network	Yes
`ipv4_address` , `ipv6_address`	Static IP address for containers	Yes for IPv4. IPv6 is not supported.
`link_local_ips`	List of link-local IPs	No. Depends on `docker run --link-local-ip`
`pid`	Sets PID mode	No. Depends on `docker run --pid` .
`ports`	Expose ports	Yes
`security-opt`	Override the default labeling scheme for containers	No. This option only applies to Windows containers, which are not supported.
`stop-signal`	Sets an alternative signal to stop the container.	Yes
`stop-grace-period`	Specify how long to wait stopping a container	No
`sysctls`	Kernel parameters to set in the container	No
`ulimits`	Override the default ulimits for a container	No
`userns_mode`	Disables the user namespace	No
`volumes` , `volume_driver`	Mount paths or named volumes	Yes
`volumes_from`	Mount volumes from another service or container	No

The following Docker run options are supported if their `docker run` counterpart is supported: `security_opt` , `stop_grace_period` , `stop_signal` , `sysctls` , `ulimits` , `userns_mode` , `cpu_shares` , `cpu_quota` , `cpuset` , `domainname` , `hostname` , `ipc` , `mac_address` , `mem_limit` , `memswap_limit` , `oom_score_adj` , `privileged` , `read_only` , `restart` , `shm_size` , `stdin_open` , `tty` , `user` , `working_dir` .

Volume Configuration Options

Option	Compose File Reference	Supported
`driver`	Specify driver to use for this volume	Yes
`driver_opts`	Specify options to pass to the driver for this volume	Yes
`labels`	Add metadata to containers	Yes
`external`	Specify that volume has been created outside of Compose	Yes

Network Configuration Options

Option	Compose File Reference	Supported
`driver`	Specify driver to use for this network	Yes
`driver_opts`	Specify options to pass to the driver for this network	No
`enable_ipv6`	Enables IPv6	No. IPv6 is not supported.
`ipam`	Specify custom IPAM configuration	Yes
`internal`	Create an externally isolated overlay network	Yes
`labels`	Add metadata to containers	Yes
`external`	Specify that network has been created outside of Compose	Yes

Supported Dockerfile Instructions

Some Dockerfile instructions are directives to the build process and a subset of them are directives to the container engine when a container is run. The latter is an important consideration when it comes to putting a Docker image into production.

For more information on Dockerfile instructions, see the Dockerfile reference here.

This topic provides information about which of the runtime Dockerfile instructions that vSphere Integrated Containers Engine 1.4 supports.

Option	Dockerfile Reference	Supported
`LABEL`	Add metadata to an image	Yes
`EXPOSE`	Expose a port	Not yet supported. Port mappings need to be explicitly declared with `docker run -p`
`ENV`	Set an environment variable	Yes
`ENTRYPOINT`	Set the executable to be run on start	Yes
`CMD`	Set commands to be run on start	Yes
`USER`	Set the user that runs the main process	Yes
`WORKDIR`	Set the working directory	Yes
`STOPSIGNAL`	Set a stop signal for the container	Not yet supported. A stop signal can be explicitly declared with `docker run --stop-signal`
`HEALTHCHECK`	Set a health check process	No health check options supported yet.
`SHELL`	Set a default shell	Yes

Obtain a Virtual Container Host

vSphere Integrated Containers Engine does not currently provide an automated means of obtaining virtual container hosts (VCHs).

When the vSphere administrator uses `vic-machine create` to deploy a VCH, the VCH endpoint VM obtains an IP address. The IP address can either be static or be obtained from DHCP. As a container developer, you require the IP address of the VCH endpoint VM when you run Docker commands.

You can see the addresses of the VCHs that are associated with your project by logging in to vSphere Integrated Containers Management Portal and selecting **Home** > **Infrastructure** > **Container Hosts**.

If the vSphere administrator deploys VCHs with TLS authentication, `vic-machine create` generates a file named `vch_name.env`. The `env` file contains Docker environment variables that are specific to the VCH. You can use the contents of the `env` file to set environment variables in your Docker client. Similarly, if the vSphere administrator deployed the VCH with TLS authentication of clients, you must obtain the client certificates. The vSphere administrator or an automated provisioning service for VCHs could potentially provide the `env` file to you when you request a VCH. For more information about setting environment variables and client certificates for VCHs in your Docker client, see Configure the Docker Client for Use with vSphere Integrated Containers.

Configure the Docker Client for Use with vSphere Integrated Containers

If your container development environment uses vSphere Integrated Containers, you must run Docker commands with the appropriate options, and configure your Docker client accordingly.

vSphere Integrated Containers Engine 1.4 supports Docker client 1.13.0. The supported version of the Docker API is 1.25.

Connecting to the VCH

How you connect to your virtual container host (VCH) depends on the security options with which the vSphere administrator deployed the VCH.

- If the VCH implements any level of TLS authentication, you connect to the VCH at *vch_address*:2376 when you run Docker commands.
- If the VCH implements mutual authentication between the Docker client and the VCH by using both client and server certificates, you must provide a client certificate to the Docker client so that the VCH can verify the client's identity. This configuration is commonly referred to as `tlsverify` in documentation about containers and Docker. You must obtain a copy of the client certificate that was either used or generated when the vSphere administrator deployed the VCH. You can provide the client certificate to the Docker client in either of the following ways:
  - By using the `--tlsverify`, `--tlscert`, and `--tlskey` options when you run Docker commands. You must also add `--tlscacert` if the server certificate is signed by a custom Certificate Authority (CA). For example:

    ```
    docker -H vch_address:2376
    --tlsverify
    --tlscert=path_to_client_cert/cert.pem
    --tlskey=path_to_client_key/key.pem
    --tlscacert=path/ca.pem
    info
    ```

  - By setting Docker environment variables:

    ```
    DOCKER_CERT_PATH=client_certificate_path/cert.pem
    DOCKER_TLS_VERIFY=1
    ```

- If the VCH uses server certificates but does not authenticate the Docker client, no client certificate is required and any client can connect to the VCH. This configuration is commonly referred to as `no-tlsverify` in documentation about containers and Docker. In this configuration, the VCH has a server certificate and connections are encrypted, requiring you to run Docker commands with the `--tls` option. For example:

  ```
  docker -H vch_address:2376 --tls info
  ```

  In this case, do not set the `DOCKER_TLS_VERIFY` environment variable. Setting `DOCKER_TLS_VERIFY` to 0 or to `false` has no effect.
- If TLS is completely disabled on the VCH, you connect to the VCH at *vch_address*:2375. Any Docker client can connect to the VCH and communications are not encrypted. As a consequence, you do not need to specify any additional TLS options in Docker commands or set any environment variables. This configuration is not recommended in production environments. For example:

  ```
  docker -H vch_address:2375 info
  ```

Using Docker Environment Variables

If the vSphere administrator deploys the VCHs with TLS authentication, `vic-machine create` generates a file named `vch_name.env`. The `env` file contains Docker environment variables that are specific to the VCH. You can use the `env` file to set environment variables in your Docker client.

The contents of the `env` files are different depending on the level of authentication with which the VCH was deployed.

- Mutual TLS authentication with client and server certificates:

  ```
  DOCKER_TLS_VERIFY=1
  DOCKER_CERT_PATH=client_certificate_path\vch_name
  DOCKER_HOST=vch_address:2376
  ```

- TLS authentication with server certificates without client authentication:

  ```
  DOCKER_HOST=vch_address:2376
  ```

- No `env` file is generated if the VCH does not implement TLS authentication.

For information about how to obtain the `env` file, see Obtain a VCH. For information about the `env` files in Docker, see `docker-machine env` in the Docker documentation.

Install the vSphere Integrated Containers Registry Certificate

If your development environment uses vSphere Integrated Containers Registry or another private registry server that uses CA server certificates, you must pass the registry's CA certificate to the Docker client. The vSphere administrator must also have configured the VCH to access the registry.

For information about how vSphere administrators deploy VCHs so that they can access a private registry, see Connect Virtual Container Hosts to Registries.

The level of security of the connection between the Docker client and the VCH is independent from the level of security of the connection between the Docker client and the registry. Connections between the Docker client and the registry can be secure while connections between the Docker client and the VCH are insecure, and the reverse.

NOTE: VCHs cannot to connect to vSphere Integrated Containers Registry instances as insecure registries. Connections to vSphere Integrated Containers Registry always require HTTPS and a certificate.

Obtain the vSphere Integrated Containers Registry CA Certificate

To access the vSphere Integrated Containers Registry CA certificate, log in to vSphere Integrated Containers Management Portal with an account that has at least the Management Portal administrator role. For information about logging in to vSphere Integrated Containers Management Portal, see Logging In to the Management Portal.

1. Go to **Administration** -> **Configuration**.
2. Click the download link for **Registry Root Certificate**.

Configure the Docker Client on Linux

This example configures a Linux Docker client so that you can log into vSphere Integrated Containers Registry by using its IP address.

1. Copy the certificate file to the Linux machine on which you run the Docker client.
2. Switch to `sudo` user.

   ```
   $ sudo su
   ```

3. Create a subfolder in the Docker certificates folder, using the registry's IP address as the folder name.

   ```
   $ mkdir -p /etc/docker/certs.d/registry_ip
   ```

4. Copy the registry's CA certificate into the folder.

   ```
   $ cp ca.crt /etc/docker/certs.d/registry_ip/
   ```

5. Open a new terminal and attempt to log in to the registry server, specifying the IP address of the registry server.

   ```
   $ docker login registry_ip
   ```

6. If the login fails with a certificate error, restart the Docker daemon.

   ```
   $ sudo systemctl daemon-reload
   ```

```
$ sudo systemctl restart docker
```

Configure the Docker Client on Windows

To pass the registry's CA certificate to a Docker client that is running on Windows 10, use the Windows Certificate Import Wizard.

1. Copy the `ca.crt` file to the Windows 10 machine on which you run the Docker client.
2. Right-click the `ca.crt` file and select **Install Certificate**.
3. Follow the prompts of the wizard to install the certificate.
4. Restart the Docker daemon:
    - Click the up arrow in the task bar to show running tasks.
    - Right-click the Docker icon and select **Settings**.
    - Select **Reset** and click **Restart Docker**.
5. Log in to the registry server.

    ```
    docker login registry_ip
    ```

Using vSphere Integrated Containers Registry with Content Trust

vSphere Integrated Containers Registry provides a Docker Notary server that allows you to implement content trust by signing and verifying the images in the registry. Management Portal administrators enable or disable content trust at the project level in vSphere Integrated Containers Management Portal.

If you the project that you are working on implements content trust, you must pass the registry's CA certificate to your Docker client and set up Docker Content Trust. By default, the vSphere Integrated Containers Registry Notary server runs on port 4443 on the vSphere Integrated Containers appliance.

Enabling content trust on a project automatically modifies the registry whitelist settings of any VCHs that are registered with the project. Consequently, when content trust is enabled, the VCHs in the project can only pull signed and verified images from the registry instance that is running in the vSphere Integrated Containers appliance.

- For general information about Docker Notary and content trust, see Content trust in Docker in the Docker documentation.
- For information about content trust in vSphere Integrated Containers, see Enabling Content Trust in Projects in *vSphere Integrated Containers Management Portal Administration*.
- For information about how enabling content trust affects VCHs, see VCH Whitelists and Content Trust in *vSphere Integrated Containers for vSphere Administrators*.

Procedure

1. If you are using a self-signed certificate, copy the CA root certificate to the Docker certificates folder.

    To pass the certificate to the Docker client, follow the procedure in Using vSphere Integrated Containers Registry above.

2. If you are using a self-signed certificate, copy the CA certificate to the Docker TLS service.

```
$ cp ca.crt ~/.docker/tls/registry_ip:4443/
```

3. Enable Docker Content Trust by setting environment variables.

```
export DOCKER_CONTENT_TRUST=1
export DOCKER_CONTENT_TRUST_SERVER=https://registry_ip:4443
```

4. (Optional) Set an alias for Notary.

   By default, the local directory for storing meta files for the Notary client is different from the folder for the Docker client. Set an alias to make it easier to use the Notary client to manipulate the keys and meta files that Docker Content Trust generates.

```
alias notary="notary -s https//registry_ip:4443 -d ~/.docker/trust --tlscacert
/etc/docker/certs.d/registry_ip/ca.crt"
```

5. When you push an image for the first time, define and confirm passphrases for the root key and the repository key for that image.

   The root key is generated at:

```
/root/.docker/trust/private/root_keys
```

   The repository key is generated at:

```
/root/.docker/trust/private/tuf_keys/[registry_name]/[image_path]
```

You can see that the signed image that you pushed is marked with a green check on the Project Repositories page in the Management Portal.

Use and Limitations of vSphere Integrated Containers Engine

vSphere Integrated Containers Engine currently includes the following capabilities and limitations:

Supported Docker Features

This version of vSphere Integrated Containers Engine supports these features:

- `docker-compose`
- Pulling images from Docker hub and private registries
- Named data volumes
- Anonymous data volumes
- Sharing concurrent NFS share points between containers
- Bridged networks
- External networks
- Port mapping
- Network links/aliases

Unsupported Docker Features

This version of vSphere Integrated Containers Engine does not support these features:

- Pulling images via image digest
- Mapping a local host folder to a container volume
- Mapping a local host file to a container
- `docker push`
- `docker build`

For limitations of using vSphere Integrated Containers with volumes, see Using Volumes with vSphere Integrated Containers Engine.

Limitations of vSphere Integrated Containers Engine

vSphere Integrated Containers Engine includes these limitations:

- If you do not configure a `PATH` environment variable, or if you create a container from an image that does not supply a `PATH`, vSphere Integrated Containers Engine provides a default `PATH`.
- You can resolve the symbolic names of a container from within another container, except in the following cases:
    - Aliases
    - IPv6
    - Service discovery

- Containers can acquire DHCP addresses only if they are on a network that has DHCP.
- When you use a standard Docker Engine, an image can have a maximum of 120 layers. When you use a vSphere Integrated Containers Engine virtual container host (VCH), an image can have a maximum of 90 layers. For more information, see Pulling Images into VCHs Fails with Image Store Error in the Troubleshooting section.

Using `docker-compose` with TLS

vSphere Integrated Containers supports TLS v1.2, so you must configure `docker-compose` to use TLS 1.2. However, `docker-compose` does not allow you to specify the TLS version on the command line. You must use environment variables to set the TLS version for `docker-compose`. For more information, see `docker-compose` issue 4651. Furthermore, `docker-compose` has a limitation that requires you to set TLS options either by using command line options or by using environment variables. You cannot use a mixture of both command line options and environment variables.

To use `docker-compose` with vSphere Integrated Containers and TLS, set the following environment variables:

```
COMPOSE_TLS_VERSION=TLSv1_2
DOCKER_TLS_VERIFY=1
DOCKER_CERT_PATH="path to your certificate files"
```

The certificate file path must lead to `CA.pem`, `key.pem`, and `cert.pem`. You can run `docker-compose` with the following command:

```
docker-compose -H vch_address up
```

Building and Pushing Images with the `dch-photon` Docker Engine

vSphere Integrated Containers Engine is an enterprise container runtime that you use as a deployment endpoint for container VMs. As such, it does not have native `docker build` or `docker push` capabilities. The job of building and pushing container images is typically part of a continuous integration (CI) pipeline, that does this by using standard Docker Engine instances.

vSphere Integrated Containers can deploy standard Docker Engine instances for you, in the form of a container image repository named `dch-photon`. The `dch-photon` image allows you to deploy container VMs that run a Docker Engine instance, known as a Docker container host (DCH), that runs on Photon OS. You can deploy any number of these `dch-photon` Docker Engine instances to perform `docker build` and `docker push` operations as part of your CI infrastructure.

vSphere Integrated Containers 1.4.x supports `dch-photon` version 1.13. The `dch-photon` image is pre-loaded in the `default-project` in vSphere Integrated Containers Registry, or you can pull it from Docker Hub.

Advantages of Using `dch-photon`

Virtual container hosts (VCHs) focus on running pre-existing images in production. An advantage of using VCHs over standard Docker Engine instances is the opinionated, strongly isolated provisioning model of container VMs as compared to standard containers. VCHs assume that image creation happens elsewhere in the CI process. vSphere Integrated Containers provides the `dch-photon` Docker Engine as a container image so that you can easily deploy Docker Engine instances to act as build slaves in your CI infrastructure.

By bringing the ephemeral quality of running the Docker Engine itself as a container VM, `dch-photon` provides the following advantages:

- Eliminates snowflake deployments of Docker Engine.
- Promotes efficient use of resources by providing an easy mechanism for provisioning and removing Docker Engine instances that fits well with CI automation.

The workflow for using `dch-photon` Docker Engines is as follows:

1. Pull the `dch-photon` image from vSphere Integrated Containers Registry and instantiate it.
2. Use the Docker Engine running in `dch-photon` to build and push an image to vSphere Integrated Containers Registry.
3. Remove the `dch-photon` container VM.
4. Pull the new image from vSphere Integrated Containers Registry into a VCH and run it in production.

Because of the ephemeral quality of the `dch-photon` Docker Engine and because it is itself a container image, this process can be scripted or integrated with an existing CI tool, such as Jenkins.

Requirements for Using `dch-photon`

To use `dch-photon`, your environment must satisfy the following conditions:

- Configure your local Docker client to use the vSphere Integrated Containers Registry certificate. For information about how to obtain the registry certificate and pass it to the Docker client, see the section Install the vSphere Integrated Containers Registry Certificate in *Configure the Docker Client for Use with vSphere Integrated Containers*.
- You have access to a VCH that the vSphere administrator configured so that it can connect to the registry to pull the `dch-photon` image. The VCH must also have a volume store named `default`. For information about how deploy a VCH that is suitable for use with `dch-photon`, see the Deploy a Virtual Container Host with a Volume Store and vSphere Integrated Containers Registry Access in *vSphere Integrated Containers for vSphere Administrators*.

Anonymous `dch-photon` Volumes

Each `dch-photon` container VM that you run creates an anonymous volume in the `default` volume store. By default, all of the images you pull into `dch-photon` go into this volume. The anonymous volume has a 2 GB limit. If you require more than 2 GB to store images and container state, you must explicitly specify a volume with a higher limit when you run `dch-photon`. For information about how to specify a larger volume, see Expand the Root Disk on a dch-photon Docker Engine.

The anonymous volumes that `dch-photon` creates are not deleted when you delete a `dch-photon` container VM. This is by design, so that you can persist your image cache and container state beyond the lifespan of an individual `dch-photon` container VM. When you delete `dch-photon` container VMs, you must manually remove the anonymous volume from the volume store if you do not require them.

Using `dch-photon` with vSphere Integrated Containers Registry

For `dch-photon` to be able to authenticate with vSphere Integrated Containers Registry, it needs to have the registry's CA certificate. The purpose of `dch-photon` is primarily to build images and push them to registries, so each `dch-photon` instance must be able to authenticate with the registry to which it pushes. Even if you use the same Docker client to pull and run the `dch-photon` image as you use to push built images back to the registry, the `dch-photon` container VM still needs to have the appropriate registry certificate so that it can successfully push images.

You can provide the certificate to `dch-photon` in one of two ways:

- Build a custom `dch-photon` image that has the certificate embedded in it. This method is preferable since you only need to perform the operation once.
- Manually copy the certificate in to a `dch-photon` container running in a VCH by using `docker cp`.

When you have deployed `dch-photon` with the registry certificate, you can use it to build an image and push that image from `dch-photon` to vSphere Integrated Containers Registry. You can then pull the image from the registry into a VCH for deployment.

For an example of how to deploy a `dch-photon` instance that you can use with vSphere Integrated Containers Registry, see Deploy a Test `dch-photon` Instance.

For more advanced use of `dch-photon`, see Advanced Use of `dch-photon`.

Deploy a Test `dch-photon` Instance

To use `dch-photon` with vSphere Integrated Containers Registry and a VCH, you must perform the following tasks, in order:

1. Obtain an appropriately configured VCH by following the procedure in Deploy a Virtual Container Host with a Volume Store and vSphere Integrated Containers Registry Access.
2. Provide the vSphere Integrated Containers Registry certificate to a `dch-photon` instance in one of the following ways:
   - Add the Registry Certificate to a Custom `dch-photon` Image. This is the recommended method because you only need to perform it once.
   - Manually Add the Registry Certificate to a `dch-photon` VM. This method must be repeated for every `dch-photon` instance that you deploy.
3. Test the dch-photon instance by following the procedure in Build, Push, and Pull an Image with `dch-photon`.

Add the Registry Certificate to a Custom `dch-photon` Image

The recommended method of passing the vSphere Integrated Containers Registry CA certificate to `dch-photon` is to create a custom `dch-photon` image that includes the certificate. You can then push the image to the vSphere Integrated Containers Registry and verify that it works by deploying it to a virtual container host (VCH).

By creating a custom image, you can deploy multiple instances of `dch-photon` that have the correct registry certificate, without having to manually copy the certificate into each `dch-photon` container VM.

Prerequisites

- You have a known user account that has at least the Developer role in the `default-project` in vSphere Integrated Containers Management Portal.
- You have an instance of Docker Engine running on your local sytem.
- You installed the CA certificate for vSphere Integrated Containers Registry in your local Docker client. For information about how to install the registry certificate in a Docker client, see Install the vSphere Integrated Containers Registry Certificate.
- You have access to a VCH that the vSphere administrator configured so that it can connect to the registry to pull the `dch-photon` image. The VCH must also have a volume store named `default`. For information about how deploy a VCH that is suitable for use with `dch-photon`, see the Deploy a Virtual Container Host with a Volume Store and vSphere Integrated Containers Registry Access in *vSphere Integrated Containers for vSphere Administrators*.
- For simplicity, this example uses a VCH that was deployed with the `--no-tlsverify` option. If your VCH implements TLS verification of clients, you must import the VCH certificates into your Docker client and adapt the Docker commands accordingly. For information about how to connect a Docker client to a VCH that uses full TLS authentication, see Connecting to the VCH.

Procedure

1. Log in to vSphere Integrated Containers Registry from your local Docker client.

   ```
   docker login registry_address
   ```

2. Pull the `dch-photon` image into the image cache in your local Docker client.

   vSphere Integrated Containers 1.4.x supports `dch-photon` version 1.13.

   ```
   docker pull registry_address/default-project/dch-photon:1.13
   ```

3. Make a new folder and copy the vSphere Integrated Containers Registry certificate into it.

4. In the new folder, create a `Dockerfile` with the following format:

   ```
   FROM registry_address/default-project/dch-photon:1.13

   COPY ca.crt /etc/docker/certs.d/registry_address/ca.crt
   ```

5. In the same folder, build the Dockerfile as a new image and give it a meaningful new tag.

   ```
   docker build -t registry_address/default-project/dch-photon:1.13-cert .
   ```

6. Push the new image into vSphere Integrated Containers Registry.

   ```
   docker push registry_address/default-project/dch-photon:1.13-cert
   ```

7. (Optional) Log in to vSphere Integrated Containers Registry from the VCH.

   If you use the same Docker client as in the preceding steps it is already authenticated with the registry. In this case, you do not need to log in again when you run commands against the VCH. If you use a different Docker client to run commands against the VCH, or you logged out, you must log in to the registry.

   ```
   docker -H vch_address:2376 --tls login registry_address
   ```

8. Pull the image from vSphere Integrated Containers Registry into the VCH and run it with the name `build-slave`.

   This example runs `dch-photon` behind a port mapping, that exposes the HTTP port (2375) of the `dch-photon` instance on port 12375 of the VCH. You can also deploy `dch-photon` on a container network.

   ```
   docker -H vch_address:2376 --tls run --name build-slave -d -p 12375:2375
   registry_address/default-project/dch-photon:1.13-cert
   ```

Result

- You have a custom `dch-photon` image in your vSphere Integrated Containers Registry that contains the correct certificate so that it can build, pull, and push images to and from that registry.
- You deployed a `dch-photon` container VM named `build-slave` from that image, that exposes Docker Engine on port 12375 of your VCH.

What to Do Next

To test the `dch-photon` Docker Engine, see Build, Push, and Pull an Image with `dch-photon`.

Manually Add the Registry Certificate to a `dch-photon` Container VM

To manually add the vSphere Integrated Containers CA certificate to `dch-photon`, you can create a `dch-photon` container VM, then use `docker cp` to copy the certificate into it.

NOTE: This method requires you to copy the certificate to every `dch-photon` container VM that you deploy. To avoid having to copy the certificate every time, the recommended method is to create a custom `dch-photon` image. For information about creating a custom image, see Add the Registry Certificate to a Custom `dch-photon` Image.

Prerequisites

- You have a known user account that has at least the Developer role in the `default-project` in vSphere Integrated Containers Management Portal.
- You have an instance of Docker Engine running on your local sytem.
- You installed the CA certificate for vSphere Integrated Containers Registry in your local Docker client. For information about how to install the registry certificate in a Docker client, see Install the vSphere Integrated Containers Registry Certificate.
- You have access to a virtual container host (VCH) that the vSphere administrator configured so that it can connect to the registry to pull the `dch-photon` image. The VCH must also have a volume store named `default`. For information about how deploy a VCH that is suitable for use with `dch-photon`, see the Deploy a Virtual Container Host with a Volume Store and vSphere Integrated Containers Registry Access in *vSphere Integrated Containers for vSphere Administrators*.
- For simplicity, this example uses a VCH that was deployed with the `--no-tlsverify` option. If your VCH implements TLS verification of clients, you must import the VCH certificates into your Docker client and adapt the Docker commands accordingly. For information about how to connect a Docker client to a VCH that uses full TLS authentication, see Connecting to the VCH in Configure the Docker Client for Use with vSphere Integrated Containers.

Procedure

1. Log in to vSphere Integrated Containers Registry from your VCH.

   ```
   docker -H vch_address:2376 --tls login registry_address
   ```

2. Pull the `dch-photon` image into the image cache in your VCH.

   vSphere Integrated Containers 1.4.x supports `dch-photon` version 1.13.

   ```
   docker -H vch_address:2376 --tls pull registry_address/default-project/dch-photon:1.13
   ```

3. Create a `dch-photon` container VM named `build-slave` in your VCH, but do not start it.

   vSphere Integrated Containers 1.4.x supports `dch-photon` version 1.13.

The container should be stopped because the Docker Engine instance that it runs must restart so that it can recognize the new certificate after you have copied it to the container. If you have already deployed `dch-photon`, use `docker stop` to stop it.

This example runs `dch-photon` behind a port mapping, that exposes the HTTP port (2375) of the `dch-photon` instance on port 12375 of the VCH.

```
docker -H vch_address:2376 --tls create --name build-slave -p 12375:2375
registry_address/default-project/dch-photon:1.13
```

4. Create the required folder structure on your local machine.

```
mkdir -p certs.d/registry_address
```

5. Copy the certificate into the new folder.

```
cp path_to_cert/ca.crt certs.d/registry_address
```

6. Use `docker cp` to copy the certificate from your local system into the `dch-photon` container VM named `build-slave` that is running in the VCH.

   Docker Engine stores registry certificates in a folder named `/etc/docker/certs.d/registry_address`.

```
 docker -H vch_address:2376  --tls cp certs.d build-slave:/etc/docker
```

7. Restart the Docker host to load the certificate.

```
docker -H vch_address:2376 --tls start build-slave
```

Result

You have a running Docker host named `build-slave`, that is exposed on port 12375 of your VCH. You configured `build-slave` to push and pull images to and from vSphere Integrated Containers Registry.

What to Do Next

To test the Docker host, see Build, Push, and Pull an Image with `dch-photon`.

Build, Push, and Pull an Image with `dch-photon`

After you have loaded the vSphere Integrated Containers Registry certificate into a `dch-photon` container VM, you can test the `dch-photon` Docker host by building an image and pushing it to vSphere Integrated Containers Registry. Then, you can pull the image into a virtual container host (VCH) to deploy it.

Prerequisites

- You performed one of the procedures in either Add the Registry Certificate to a Custom Image or Manually Add the Registry Certificate to a `dch-photon` VM to create an instance of the `dch-photon` Docker Engine, named `build-slave`.
  - The `build-slave` container VM includes the CA certificate of your vSphere Integrated Containers Registry instance.
  - The `build-slave` container VM is exposed on port 12375 of the VCH.
- For simplicity, this example uses a VCH that was deployed with the `--no-tlsverify` option. If your VCH implements TLS verification of clients, you must import the VCH certificates into your Docker client and adapt the Docker commands accordingly. For information about how to connect a Docker client to a VCH that uses full TLS authentication, see Connecting to the VCH in Configure the Docker Client for Use with vSphere Integrated Containers.
- This procedure assumes that the VCH uses the same network for the client and public networks. If a VCH is deployed using separate client and public networks, the VCH endpoint is exposed on the client network. When you deploy a `dch-photon` Docker Engine on the VCH, it is exposed on the public network and the commands in the procedure fail.

vSphere Integrated Containers 1.4.x supports `dch-photon` version 1.13.

Procedure

1. Run `docker info` to test that the Docker host running in the `dch-photon` container VM has started correctly.

   By specifying port 12375 you direct the Docker client to the `dch-photon` Docker Engine that is running in the VCH, rather than to the VCH itself.

   ```
   docker -H vch_address:12375 info
   ```

2. Test that you can authenticate with the registry from the `dch-photon` container VM.

   You should not need to log in if your client is already authenticated with the registry, but the `login` command is included here for clarity. You specify port 12375 to run the `login` command on the `dch-photon` Docker Engine, rather than on the VCH.

   ```
   docker -H vch_address:12375 login registry_address
   ```

3. Test that you can pull images from the registry into the `dch-photon` container VM.

   Specify port 12375 to run the `pull` command on the `dch-photon` Docker Engine.

```
docker -H vch_address:12375 pull registry_address/default-project/dch-photon:1.13
```

4. Remove the test image from the `dch-photon` Docker Engine.

   Specify port 12375 to run the `rmi` command on the `dch-photon` Docker Engine.

   ```
   docker -H vch_address:12375 rmi registry_address/default-project/dch-photon:1.13
   ```

5. Create a simple `Dockerfile` and save it in the current directory.

   Copy the following text into `Dockerfile`:

   ```
   FROM debian:latest

   RUN apt-get update -y && apt-get install -y fortune-mod fortunes

   ENTRYPOINT ["/usr/games/fortune", "-s"]
   ```

6. Build an image named `test-container` from the `Dockerfile`, and tag it with the path to a project in vSphere Integrated Containers Registry.

   Specify port 12375 to run the `build` command on the `dch-photon` Docker Engine.

   ```
   docker -H vch_address:12375 build -t registry_address/default-project/test-container .
   ```

7. Push the image from the `dch-photon` Docker host to the registry.

   Specify port 12375 to run the `push` command on the `dch-photon` Docker Engine.

   ```
   docker -H vch_address:12375 push registry_address/default-project/test-container
   ```

8. Pull the image from the registry into the VCH.

   Specify port 2376 to run the `pull` command on the VCH.

   ```
   docker -H vch_address:2376 --tls pull registry_address/default-project/test-container
   ```

9. Instantiate a container from the `test-container` image on the VCH.

   Specify port 2376 to run the test container on the VCH.

   ```
   docker -H vch_address:2376 --tls run --name test-container registry_address/default-
   project/test-container
   ```

10. List the containers that are running and stopped in the VCH.

Specify port 2376 to run the `ps -a` command on the VCH.

```
docker -H vch_address:2376 --tls ps -a
```

11. (Optional) Log in to vSphere Integrated Containers Management Portal.

    You should see the `test-container` image in the list of repositories for `default-project` and the `test-container` container VM in the list of containers.

Result

You built a `test-container` image in a `dch-photon` Docker Engine and pushed it from the `dch-photon` instance to vSphere Integrated Containers Registry. You pulled the `test-container` image from the registry into a VCH and ran it. The resulting `test-container` container VM appears in the list of containers that have run in the VCH.

NOTE: Each `dch-photon` Docker Engine that you run creates an anonymous volume in the `default` volume store. This anonymous volume is not deleted when you delete a `dch-photon` container VM. When you delete `dch-photon` container VMs, you must manually remove the anonymous volume from the volume store.

Advanced Use of `dch-photon`

For information about how to use `dch-photon` with TLS authentication and with other registries than vSphere Integrated Containers Registry, see Advanced dch-photon Deployment Options.

For information about to use `dch-photon` with large images or with large numbers of images, see Expand the Root Disk on a dch-photon Docker Engine.

For information about configuring `dch-photon` to use proxy servers, see Configure `dch-photon` to Use Proxy Servers.

For information about configuring `dch-photon` to connect to registries that use a custom CA, see Add a Custom Registry Certificate Authority to `dch-photon`.

For information about instantiating a Docker swarm with `dch-photon`, see Automating Swarm Creation with vSphere Integrated Containers.

NOTE: Using `dch-photon` to instantiate Docker swarm is not officially supported.

Advanced `dch-photon` Deployment Options

You do not need to specify any options when you use `docker run` to deploy `dch-photon` Docker Engine instances for use with vSphere Integrated Containers Registry. However, you can optionally specify `dch-photon` options in the `docker run` command to run the `dch-photon` Docker Engine with TLS authentication.

You can also specify `dch-photon` options to connect `dch-photon` Docker Engine instances to registries other than vSphere Integrated Containers Registry.

vSphere Integrated Containers 1.4.x supports `dch-photon` version 1.13.

- `dch-photon` Options
- Using `dch-photon` with TLS Authentication
  - With Remote Verification
  - Without Remote Verification
  - With Automatically Generated Certificates

`dch-photon` Options

You can specify the following options when you deploy `dch-photon` Docker Engine instances:

- `-insecure-registry` : Enable insecure registry communication. Set this option multiple times to create a list of registries to which `dch-photon` applies no security considerations. You cannot use this option when connecting to vSphere Integrated Containers Registry.
- `-local` : Do not bind the Docker API to external interfaces. Set this option to prevent the Docker API endpoint from binding to the external interface. Docker Engine only listens on `/var/run/docker.sock` .
- `-storage` : Sets the Docker storage driver that Docker Engine uses. By default, the storage driver is `overlay2` , which is the recommended driver when running Docker Engine as a container VM.
- `-tls` : Use TLS authentication for all connections. Implied by `-tlsverify` . This option enables secure communication with no verification of the remote end. To use custom certificates, copy them into the `/certs` folder in the `dch-photon` container VM. Certificates are generated automatically in `/certs` if you do not provide them.

  - Server certificate: `/certs/docker.crt`
  - Key for the server certificate: `/certs/docker.key`
- `-tlsverify` : Use TLS and authentication for all connections and verify the remote end. To use custom certificates, copy them into the `/certs` folder in the `dch-photon` container. Certificates are generated automatically in `/certs` if you do not provide them.

  - Server certificate: `/certs/docker.crt`
  - Key for the server certificate: `/certs/docker.key`
  - CA certificate: `/certs/ca.crt`
  - CA key: `/certs/ca-key.pem`
  - Client certificate: `/certs/docker-client.crt`
  - Client key: `/certs/docker-client.key`

- `vic-ip`: Set the IP address of the virtual container host for use in automatic certificate generation when running `dch-photon` containers behind a port mapping.

Using `dch-photon` with TLS Authentication

To configure the same certificate-based authentication for a `dch-photon` as you have for your VCH endpoint, you specify the `-tls` or `-tlsverify` option when you run the `dch-photon` the container VM. You then copy the appropriate certificates into the `dch-photon` container VM.

With Remote Verification

1. Create a `dch-photon` container without starting it.

   This example runs `dch-photon` behind a port mapping and specifies the `-tlsverify` option.

   ```
   docker create -p 12376:2376 --name dch-photon-tlsverify registry_address/default-project/dch-photon:1.13 -tlsverify
   ```

2. Copy the certificates into the `dch-photon` container.

   ```
   docker cp cert_folder/ca.pem dch-photon-tlsverify:/certs/ca.crt
   ```

   ```
   docker cp cert_folder/server-cert.pem dch-photon-tlsverify:/certs/docker.crt
   ```

   ```
   docker cp cert_folder/server-key.pem dch-photon-tlsverify:/certs/docker.key
   ```

3. Start the `dch-photon` container.

   ```
   docker start dch-photon-tlsverify
   ```

4. Connect to the `dch-photon` container.

   ```
   docker -H vch_adress:12376 --tlsverify info
   ```

Without Remote Verification

1. Create a `dch-photon` container without starting it.

   This example runs `dch-photon` behind a port mapping and specifies the `-tls` option.

   ```
   docker create -p 12376:2376 --name dch-photon-tls registry_address/default-project/dch-photon:1.13 -tls
   ```

2. Copy the certificates into the `dch-photon` container.

   ```
   docker cp cert_folder/server-cert.pem dch-photon-tls:/certs/docker.crt
   ```

   ```
   docker cp cert_folder/server-key.pem dch-photon-tls:/certs/docker.key
   ```

3. Start the `dch-photon` container.

   ```
   docker start dch-photon-tls
   ```

4. Connect to the `dch-photon` container.

   ```
   docker -H vch_adress:12376 --tls info
   ```

With Automatically Generated Certificates

To generate certificates automatically, specify either `-tls` or `-tlsverify`. If the `dch-photon` container runs behind a port mapping, specify the address of the VCH in the `-vic-ip` option. This address is used during certificate generation.

```
docker run -p 12376:2376 --name dinv-build -v mycerts:/certs vmware/dch-photon -tlsverify
-vic-ip vch_adress
```

You can then use `docker cp` to copy the automatically generated certificates to your local Docker client.

Expand the Root Disk on a `dch-photon` Docker Engine

Depending on how many images you are planning to build in a `dch-photon` Docker Engine instance, you might need a larger root disk than the default of 2GB.

To create a larger root disk, use the `docker volume create` command to create a disk of the desired size and then mount it to the `dch-photon` container VM by using the `-v` option.

Prerequisites

- You have access to a virtual container host (VCH) that the vSphere administrator configured so that it can connect to the registry to pull the `dch-photon` image. The VCH must also have a volume store named `default`. For information about how deploy a VCH that is suitable for use with `dch-photon`, see the Deploy a Virtual Container Host with a Volume Store and vSphere Integrated Containers Registry Access in *vSphere Integrated Containers for vSphere Administrators*.
- You have an instance of Docker Engine running on your local sytem.
- For simplicity, this example uses a VCH that was deployed with the `--no-tlsverify` option. If your VCH implements TLS verification of clients, you must import the VCH certificates into your Docker client and adapt the Docker commands accordingly. For information about how to connect a Docker client to a VCH that uses full TLS authentication, see Connecting to the VCH.

Procedure

1. Log in to vSphere Integrated Containers Registry from your VCH.

```
docker -H vch_address:2376 --tls login registry_address
```

2. Pull the `dch-photon` image into the image cache in your local Docker client.

```
docker  -H vch_address:2376 --tls pull registry_address/default-project/dch-photon:1.13
```

3. Create a volume of the desired size in your VCH.

```
docker -H vch_address:2376 --tls volume create --opt Capacity=30GB --name mydchdisk
```

4. Run the `dch-photon` container VM in the VCH, behind a port mapping.

```
docker -H vch_address:2376 --tls run --name DCH -d -v mydchdisk:/var/lib/docker -p
12375:2376 registry_address/default-project/dch-photon:1.13
```

5. Run `docker info` on the newly deployed docker host.

```
docker -H vch_address:12375 info
```

Configure `dch-photon` to Use Proxy Servers

If your environment uses proxies, you must configure `dch-photon` containers to use the proxy servers.

You can configure proxy servers on `dch-photon` containers either by setting environment variables at runtime, or by creating a custom `dch-photon` image that includes the same variables.

Set Environment Variables at Runtime

When you run the `dch-photon` container, use the `--env` option to add the proxy servers as environment variables. If you use this method, you must set the environment variables every time that you run `dch-photon`.

```
$ docker run
--detach
--env https_proxy=https://proxy.server.com:3128
--env http_proxy=http://proxy.server.com:3128
--publish 12376:2376
vmware/dch-photon:1.13
-tls
-vic-ip vch_adress
```

This command instantiates a `dch-photon` container with the following configuration:

- Uses `--detach` to run the container in the background.
- Sets HTTP and HTTPS proxy servers as environment variables.
- Exposes the Docker API running in the `dch-photon` container to port 12376 on the virtual container host (VCH) on which it is deployed.
- Uses the `dch-photon` options `-tls` and `-vic-ip` to use auto-generated certificates without client verification when connecting to the VCH.

Add Environment Variables to a Custom `dch-photon` Image

Build a new `dch-photon` image, for example named `dch-photon-proxy` based on the official one. To do this, you create a `Dockerfile` that includes proxy environment variables:

```
dockerfile
FROM vmware/dch-photon:1.13
ENV http_proxy http://proxy.server.com:8080
ENV https_proxy https://proxy.server.com:8080
```

If you use this method, you do not need to specify the environment variables each time you run containers from the custom `dch-photon-proxy` image.

Add a Custom Registry Certificate Authority to `dch-photon`

If your registry uses a custom Certificate Authority (CA), you can add the CA root and other certificates to trusted root of the `dch-photon` container.

You might need to do this if you have seen errors such as the following when attempting to log in to the registry:

```
Error response from daemon: Get https://exampleregistry:443/v2/: x509: certificate signed by unknown authority
```

Prerequisites

- You are using `dch-photon` as a container host in a CI or build/push setup.
- You used a custom CA to generate registry certificates.

Procedure

1. Obtain the root and any secondary certificate files, and copy them into `/etc/ssl/certs` on your working machine.
2. Build a new `dch-photon` image, for example named `dch-photon-ca`.

   To do this, you create a `Dockerfile` that extends the standard `dch-photon` image:

   ```
   dockerfile
   FROM vmware/dch-photon
   COPY certs/*.crt /etc/ssl/certs/
   RUN tdnf install -y openssl-c_rehash
   ADD docker-entrypoint.sh /docker-entrypoint.sh
   ```

   This image adds the following to `dch-photon`:

   - Copies the root and any secondary certificates into `/etc/ssl/certs` in the `dch-photon` container.
   - Installs `openssl-c_rehash`. You need to rehash the CAs so that programs such as OpenSSL can find newly added CAs.
   - Add in a script named `docker-entrypoint.sh` to run when you run containers from this image. This is optional.
3. Create the `docker-entrypoint.sh` script.

   This script injects the certificates into `dch-photon` and starts it.

   ```
   sh
   echo "Injecting CA certs"
   openssl x509 -in /etc/ssl/certs/root.pem -text >> /etc/pki/tls/certs/ca-bundle.crt
   openssl x509 -in /etc/ssl/certs/root-secondary.pem -text >> /etc/pki/tls/certs/ca-bundle.crt
   echo "Rehashing new certificates"
   ```

```
c_rehash
echo "Starting DinV"
exec /dinv -tls
```

Result

You can log in to the Docker registry that uses the custom CA from containers that you run from the `dch-photon-ca` image.

Using Volumes with vSphere Integrated Containers

vSphere Integrated Containers supports the use of container volumes. You can create container volumes either in volume stores on vSphere datastores or in NFS share points that you designate as volume stores. The vSphere datastore or NFS share point houses the volume store and containers build volumes in that volume store.

IMPORTANT: To use container volume capabilities with vSphere Integrated Containers, the vSphere administrator must configure one or more volume stores on the virtual container host (VCH). When the vSphere administrator creates a VCH, they can specify a vSphere datastore or NFS share point to use to store container volumes. For information about how to create VCHs with volume stores, see Specify Volume Stores. For information about how to add volume stores to existing VCHs, see Add Volume Stores.

- Obtain the List of Available Volume Stores
- Obtain the List of Available Volumes
- Create a Volume in a Volume Store
- Creating Volumes from Images
- Create a Container with a New Anonymous or Named Volume
    - Create a Container with a New Anonymous Volume
    - Create a Container with a Named Volume
- Mount Existing vSphere-Backed Volumes on Containers
- Sharing NFS-Backed Volumes Between Containers
- Obtain Information About a Volume
- Delete a Named Volume from a Volume Store
- Delete a Container and the Anonymous Volumes Attached to It
- Run a Container and Delete the Anonymous Volumes Attached to it when it Stops

For simplicity, the examples in this topic assume that the VCHs implement TLS authentication with self-signed server certificates, with no client verification.

Obtain the List of Available Volume Stores

To obtain the list of volume stores that are available on a VCH, run `docker info`.

```
docker -H virtual_container_host_address:2376 --tls info
```

The list of available volume stores for this VCH appears in the `docker info` output under `VolumeStores`.

```
[...]
Storage Driver: vSphere Integrated Containers Backend Engine
VolumeStores: volume_store_1 volume_store_2 ... volume_store_n
vSphere Integrated Containers Backend Engine: RUNNING
[...]
```

Obtain the List of Available Volumes

To obtain a list of volumes that are available on a VCH, run `docker volume ls`.

```
docker -H virtual_container_host_address:2376 --tls volume ls

DRIVER          VOLUME NAME
vsphere         volume_1
vsphere         volume_2
[...]           [...]
vsphere         volume_n
```

Create a Volume in a Volume Store

When you use the `docker volume create` command to create a volume, you can optionally provide a name for the volume by specifying the `--name` option. If you do not specify `--name`, `docker volume create` assigns a random UUID to the volume.

- If the vSphere administrator created the VCH with one or more volume stores, but none of the volume stores are named `default`, you must specify the name of an existing volume store in the `--opt VolumeStore` option. If you do not specify `--opt VolumeStore`, `docker volume create` searches for a volume store named `default`, and returns an error if no such volume store exists.

  ```
  docker -H virtual_container_host_address:2376 --tls volume create
  --opt VolumeStore=volume_store_label
  --name volume_name
  ```

- If the vSphere administrator created the VCH with a volume store named `default`, you do not need to specify `--opt VolumeStore` in the `docker volume create` command. If you do not specify a volume store name, the `docker volume create` command automatically uses the `default` volume store if it exists.

  ```
  docker -H virtual_container_host_address:2376 --tls volume create
  --name volume_name
  ```

- You can optionally set the capacity of a volume by specifying the `--opt Capacity` option when you run `docker volume create`. If you do not specify the `--opt Capacity` option, the volume is created with the default capacity of 1024MB.

  If you do not specify a unit for the capacity, the default unit will be in Megabytes.

  ```
  docker -H virtual_container_host_address:2376 --tls volume create
  --opt VolumeStore=volume_store_label
  --opt Capacity=2048
  --name volume_name
  ```

- To create a volume with a capacity in megabytes, gigabytes, or terabytes, include `MB`, `GB`, or `TB` in the value that you pass to `--opt Capacity`. The unit is case insensitive.

  ```
  docker -H virtual_container_host_address:2376 --tls volume create
  --opt VolumeStore=volume_store_label
  --opt Capacity=10GB
  --name volume_name
  ```

- vSphere Integrated Containers Engine currently only supports `ext4` file systems for volumes.

After you create a volume by using `docker volume create`, you can mount that volume in a container by running either of the following commands:

```
docker -H virtual_container_host_address:2376 --tls
create -v volume_name:/folder busybox
```

```
docker -H virtual_container_host_address:2376 --tls
run -v volume_name:/folder busybox
```

In the examples above, Docker mounts the volume `volume_name` to `/folder` in the container.

NOTE: When using a vSphere Integrated Containers Engine VCH as your Docker endpoint, the storage driver is always the vSphere Integrated Containers Engine Backend Engine. If you specify the `docker volume create --driver` option an error stating that a bad driver has been selected will occur.

Creating Volumes from Images

Some images, for example, `mongo` or `redis:alpine`, contain volume bind information in their metadata. vSphere Integrated Containers Engine creates such volumes with the default parameters and treats them as anonymous volumes. vSphere Integrated Containers Engine treats all volume mount paths as unique, in the same way that Docker does. This should be kept in mind if you attempt to bind other volumes to the same location as anonymous or image volumes. A specified volume always takes priority over an anonymous volume.

If you require an image volume with a different volume capacity to the default, create a named volume with the required capacity. You can mount that named volume to the location that the image metadata specifies. You can find the location by running `docker inspect image_name` and consulting the `Volumes` section of the output. The resulting container has the required storage capacity and the endpoint.

Create a Container with a New Anonymous or Named Volume

If you intend to create named or anonymous volumes by using `docker create -v` when creating containers, a volume store named `default` must exist in the VCH.

NOTES:

- vSphere Integrated Containers Engine does not support mounting vSphere datastore folders as data volumes. A command such as `docker create -v /folder_name:/folder_name busybox` is not supported if the volume store is a vSphere datastore.
- If you use `docker create -v` to create containers and mount new volumes on them, vSphere Integrated Containers Engine only supports the `-r` and `-rw` options.
- Anonymous volumes are only recommended for development rather than production environments. A valid use case for anonymous volumes is the creation of ephemeral Docker build hosts for a CI pipeline.

Create a Container with a New Anonymous Volume

To create an anonymous volume, you include the path to the destination at which you want to mount the anonymous volume in the `docker create -v` command. Docker creates the anonymous volume in the `default` volume store, if it exists. The VCH mounts the anonymous volume on the container.

The `docker create -v` example below performs the following actions:

- Creates a busybox container that uses an anonymous volume in the `default` volume store.
- Mounts the volume to `/volumes` in the container.

```
docker -H virtual_container_host_address:2376 --tls
create -v /volumes busybox
```

Create a Container with a Named Volume

To create a container with a new named volume, you specify a volume name in the `docker create -v` command. When you create containers that with named volumes, the VCH checks whether the volume exists in the volume store, and if it does not, creates it. The VCH mounts the existing or new volume on the container.

The `docker create -v` example below performs the following actions:

- Creates a busybox container
- Creates volume named `volume_1` in the `default` volume store.
- Mounts the volume to the `/volumes` folder in the container.

```
docker -H virtual_container_host_address:2376 --tls
create -v volume_1:/volumes busybox
```

Mount Existing vSphere-Backed Volumes on Containers

If your volume store is in a vSphere datastore, mounting existing volumes on containers is subject to the following limitations:

- vSphere Integrated Containers currently supports mounting a volume that is backed by vSphere on only one container at a time.
- Docker does not support unmounting a volume from a container, whether that container is running or not. When you mount a volume on a container by using `docker create -v`, that volume remains mounted on the container until you remove the container. When you have removed the container you can mount the volume onto a new container.
- If you intend to create and mount a volume on one container, remove that container, and then mount the same volume on another container, use a named volume. It is possible to mount an anonymous volume on one container, remove that container, and then mount the anonymous volume on another container, but it is not recommended to do so.

The `docker create -v` example below performs the following operations:

- Creates a container named `container1` from the `busybox` image.
- Mounts the named volume `volume1` to the `myData` folder on that container, starts the container, and attaches to it.
- After performing operations in `volume1:/myData`, stops and removes `container1`.
- Creates a container named `container2` from the Ubuntu image.
- Mounts `volume1` to the `myData` folder on `container2`.

```
docker -H virtual_container_host_address:2376 --tls
create --name container1 -v volume1:/myData busybox
docker start container1
docker attach container1

[Perform container operations and detach]

docker stop container1
docker rm container1
docker create -it --name container2 -v volume1:/myData ubuntu
docker start container2
docker attach container2

[Perform container operations with the same volume that was
previously mounted to container1]
```

Sharing NFS-Backed Volumes Between Containers

If your volume store is in an NFS share point, sharing volumes between containers is not subject to any limitations. In vSphere Integrated Containers, the `local` driver is the vSphere Integrated Containers Docker personality. Consequently, the way to create NFS volumes with vSphere Integrated Containers is slightly different to how you do it with regular Docker. All that you need to do to create an NFS volume for a container is provide the name of the appropriate volume store in the `docker volume create` command.

```
docker volume create --opt volumestore=nfs_volumestore_name
```

NOTE: vSphere Integrated Containers mounts NFS volumes as `root`. Consequently, if containers are to run as non-root users, the volume store must be configured with the correct permissions so that the non-root users can access it. For information about how to configure NFS volume stores for non-root users, see About NFS Volume Stores and Permissions in *vSphere Integrated Containers for vSphere Administrators*.

Obtain Information About a Volume

To get information about a volume, run `docker volume inspect` and specify the name of the volume.

```
docker -H virtual_container_host_address:2376 --tls
volume inspect volume_name
```

Delete a Named Volume from a Volume Store

To delete a volume, run `docker volume rm` and specify the name of the volume to delete.

```
docker -H virtual_container_host_address:2376 --tls
volume rm volume_name
```

Delete a Container and the Anonymous Volumes Attached to It

To remove a container and anonymous volumes joined to that container, run `docker rm -v`. If an anonymous volume is in use by another container, it is not removed.

```
$ docker rm -v container1
```

Run a Container and Delete the Anonymous Volumes Attached to it when it Stops

To run a container that creates anonymous volumes and then removes those volumes at the end of its run, run `docker run --rm`.

```
$ docker run --rm container1
```

Container Networking with vSphere Integrated Containers Engine

The following sections present examples of how to perform container networking operations when using vSphere Integrated Containers Engine as your Docker endpoint.

- Publish a Container Port
- Add Containers to a New Bridge Network
- Bridged Containers with an Exposed Port
- Deploy Containers on Multiple Bridge Networks
- Deploy Containers That Combine Bridge Networks with a Container Network
- Deploy a Container with a Static IP Address

To perform certain networking operations on containers, your Docker environment and your virtual container hosts (VCHs) must be configured in a specific way.

- For information about the default Docker networks, see https://docs.docker.com/engine/userguide/networking/.
- For information about the networking options with which vSphere administrators can deploy VCHs and examples, see Virtual Container Host Networks in *vSphere Integrated Containers for vSphere Administrators*.

NOTE: The default level of trust on VCH container networks is `published`. As a consequence, if the vSphere administrator did not configure `--container-network-firewall` on the VCH, you must specify `-p 80` in `docker run` and `docker create` commands to publish port 80 on a container. Alternatively, the vSphere administrator can configure the VCH to set `--container-network-firewall` to a different level.

Publish a Container Port

Connect a container to an external mapped port on the public network of the VCH:

```
$ docker run -p 8080:80 --name test1 my_container my_app
```

Result: You can access Port 80 on `test1` from the public network interface on the VCH at port 8080.

Add Containers to a New Bridge Network

Create a new non-default bridge network and set up two containers on the network. Verify that the containers can locate and communicate with each other:

```
$ docker network create -d bridge my-bridge-network
$ docker network ls
...
NETWORK ID          NAME                DRIVER
615d565d498c        my-bridge-network   bridge
...
$ docker run -d --net=my-bridge-network \
                --name=server my_server_image server_app
$ docker run -it --name=client --net=my-bridge-network busybox
```

```
/ # ping server
PING server (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.073 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.092 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.088 ms
```

Result: The `server` and `client` containers can ping each other by name.

Note: Containers created on the default bridge network don't get name resolution by default in the way described above. This is consistent with docker bridge network behavior.

Bridged Containers with an Exposed Port

Connect two containers on a bridge network and set up one of the containers to publish a port via the VCH. Assume that `server_app` binds to port 5000.

```
$ docker network create -d bridge my-bridge-network
$ docker network ls
...
NETWORK ID          NAME                DRIVER
615d565d498c        my-bridge-network   bridge
...
$ docker run -d -p 5000:5000 --net=my-bridge-network \
                --name=server my_server_image server_app
$ docker run -it --name=client --net=my-bridge-network busybox
/ # ping -c 3 server
PING server (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.073 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.092 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.088 ms
/ # telnet server 5000
GET /

Hello world!Connection closed by foreign host
$ telnet vch_public_interface 5000
Trying 192.168.218.137...
Connected to 192.168.218.137.
Escape character is '^]'.
GET /

Hello world!Connection closed by foreign host.
```

Result: The `server` and `client` containers can ping each other by name. You can connect to `server` on port 5000 from the `client` container and to port 5000 on the VCH public network.

Deploy Containers on Multiple Bridge Networks

You can use multiple bridge networks to isolate certain types of application network traffic. An example may be containers in a data tier communicating on one network and containers on a web tier communicating on another. In order for this to work, at least one of the containers needs to be on both networks.

Docker syntax does not allow for the use of multiple `--net` arguments for `docker run` or `docker create`, so to connect a container to multiple networks, you need to use:

`docker network connect [network-id] [container-id]`

Note: With VIC containers, networks can only be added to a container when it's in its created state. They can't be added while the container is running.

Create two bridge networks, one for data traffic and one for web traffic

```
docker network create --internal bridge-db
docker network create bridge-web
```

Create and run the data container(s)

```
docker run -d --name db --net bridge-db myrepo/mydatabase
```

Create and run the web container(s) and make sure one is on both networks. Expose the web front end on port 8080 of the VCH.

```
docker create -d --name model --net bridge-db myrepo/web-model
docker network connect bridge-web web-model
docker start model
docker run -d -p 8080:80 --name view --net bridge-web myrepo/web-view
```

Result:

- `db` and `web-view` cannot communicate with each other
- `web-model` can communicate with both `db` and `web-view`
- `web-view` exposes a service on port 8080 of the VCH

Note: A container on multliple bridge networks will not get a distinct network interface for each network, rather it will get multiple IP addresses on the same interface. Use `ip addr` to see the IP addresses.

Deploy Containers That Combine Bridge Networks with a Container Network

A "container" network is a vSphere port group that a container can be connected to directly and which allows the container to have an external identity on that network. This can be combined with one or more private bridge networks for intra-container traffic.

NOTE: Multiple bridge networks are backed by the same port group as the default bridge, segregated via IP address management. Container networks are strongly isolated from all other networks.

A container network is specified when the VCH is installed using `vic-machine --container-network [existing-port-group]` and should be visible when you run `docker network ls` from a Docker client.

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
```

```
baf6919f5721        ExternalNetwork     external
fc41d9a86514        bridge              bridge
```

The three main advantages of using a container network over exposing a port on the VCH are that:

1) The container can get its own external IP address. 2) The container is not dependent on the VCH control plane being up for network connectivity. This allows the VCH to be powered down or upgraded with zero impact on the network connectivity of the deployed container. 3) This avoids the use of NAT, which will benefit throughput performance

Let's take the above example with the web and data tiers and show how it could be achieved using a container network.

Create one private bridge network for data traffic

```
docker network create --internal bridge-db
```

Create and run the data container(s)

```
docker run -d --name db --net bridge-db myrepo/mydatabase
```

Create and run the web container(s) and make sure one is on both networks. In this example, we only want the web-view container to have an identity on the ExternalNetwork, so the web-model container is only in the data network.

```
docker run -d --name model --net bridge-db myrepo/web-model
docker create -d -p 80 --name view --net bridge-db myrepo/web-view
docker network connect ExternalNetwork view
docker start view
```

Result:

- All the containers can communicate with each other.
- `db` and `web-model` cannot communicate externally
- `web-view` has its own external IP address and its service is available on port 80 of that IP address

Note: Given that a container network manifests as a vNIC on the container VM, it has its own distinct network interface in the container.

Deploy a Container with a Static IP Address

Deploy a container that has a static IP address on the container network. For you to be able to deploy containers with static IP addresses, the vSphere administrator must have specified the `--container-network-ip-range` option when they deployed the VCH. The IP address that you specify in `docker network connect --ip` must be within the specified range. If you do not specify `--ip`, the VCH assigns an IP address from the range that the vSphere administrator specified in `--container-network-ip-range`.

```
$ docker network connect --ip ip_address container-net container1
```

Result: The container `container1` runs with the specified IP address on the `container-net` network.

Creating Containerized Applications with vSphere Integrated Containers Engine

The topics in this section provides guidelines for container developers who want to use vSphere Integrated Containers Engine to develop and deploy a containerized application.

vSphere Integrated Containers is designed to help you get the best out of your vSphere infrastucture by adding a container consumption model to it. That means that you can consume vSphere networks, storage and compute in a way that's familiar, autonomous, scriptable, opinionated and portable. There are significant benefits to this approach and also limits to what you can do.

This section will help you to understand the considerations, benefits and limits to putting containers into production with vSphere Integrated Containers Engine. It includes plenty of examples of common deployment scenarios, including using Docker Compose.

- How to get the best out of vSphere Integrated Containers when putting containerized applications into production
- Example of deploying a single container VM into production with vSphere Integrated Containers engine
- Example of deploying multiple container VMs into production using Docker Compose

Putting Applications into Production with vSphere Integrated Containers Engine

vSphere Integrated Containers engine is designed to be a docker API compatible production endpoint for containerized workloads. As such, the design focus is on provisioning containerized applications with optimal isolation, security, data persistence, throughput performance and to take advantage of vSphere capabilities.

vSphere Integrated Containers engine is designed to make existing features of vSphere easy to consume and exploit by providing compatibilty with the Docker image format and Docker client. Inevitably that means that there are some differences between a regular Docker host and a virtual container host (VCH), and between a Linux container and a container VM. Some of those differences are intentional design constraints, such as there being no such thing as a "privileged" container in vSphere Integrated Containers. Some are because of a lack of functional completeness, while others are outside of the existing scope of the product, such as native support for `docker build`.

There are other sections that discuss these topics in more depth, but this section is intented to help you to understand how to maximize business value by understanding how the capabilities of the product map to production requirements.

Building Images for production

While official images on sites like Docker Hub are useful for showing how an application might be containerized, these images are rarely suitable to put into production as is. Exploring how to customize images is outside of the scope of this document, but important considerations include:

Anonymous volumes

You can specify a volume in a container image using the `VOLUME` keyword. However, this does not allow you to specify any characteristics about the volumes and is only recommended for development rather than production environments. A VCH can have mutliple volume stores and a volume is a disk, so being able to specify an appropriate volume store and the size of the disk is an important consideration.

Note also that a volume in vSphere Integrated Containers will have a `/lost+found` folder in it due to the ext4 filesystem and if your application needs an empty folder, you should specify a sub directory in the volume. Eg.

```
docker run -v mydisk:/mountpoint -e DATA_DIR=/mountpoint/data myimage
```

Exposing network ports

You can expose network ports in a Dockerfile using EXPOSE and leave it up to the container engine to define port mappings using `docker run -P`. There are a few considerations with this.

If you want to expose your container to other containers on a bridge network, you don't need to use EXPOSE. Your container will be resolvable by name.

If you want your container to be externally accessible, vSphere Integrated Containers Engine gives you the option to use an external container network rather than port mapping. This is more robust and more performant because it doesn't depend on the container engine being available for a network connection and it doesn't rely on NAT networking. Your container gets its own IP address on that container network. Exposing your container on a container network cannot be specified in a Dockerfile.

If you want to use a port mapping on the VCH endpoint VM, it's rarely the case that you want the container engine to pick a random port and again, that's not something that can be specified in the Dockerfile. Better to use `docker run -p <external>:<internal>` at deployment.

Environment variables

Environment variables are a very useful way of setting both static and dynamic configuration. Use of Environment variables in a Dockerfile should be considered static configuration as they will be the same on every deployment. Setting them on the command-line allows for dynamic configuration and over-riding of static settings.

Ephemeral and Persistent State

The question of where a container stores its state is an important one. A container has an ephemeral filesystem and multiple optional persistent volume mounts. Any writes to any part of the filesystem that is not a mounted volume is stored only until the container is deleted.

When a regular Linux container is deployed into a VM, there are typically two types of filesystem in the guest OS. An overlay filesystem manages the image data and stores ephemeral state. A volume will typically be another part of the guest filesystem mounted into the container. As such it is also possible for Linux containers to have shared read/write access to the same filesystem on the container host. This is useful in development, but potentially problematic in production as it forces containers to be tied to each other and to a specific container host. That may well be by design in the case where multiple containers form a single service and a single unit of scale. What's important however is to consider the scope, persistence and isolation of data when deploying containerized applications.

Take a database container as an example. Its data almost certainly needs to be backed up, live beyond the lifecycle of the container and not be mixed up with any other kind of data. The problem of peristing such state onto a container host filesystem is that it's mixed in with other state and cannot easily be backed up, unless the host itself has a disk mounted specifically for that purpose. There are volume drivers that can be used with Docker Engine for this purpose. Eg. VMware Docker Volume Service

When you deploy a container to a VCH, ephemeral state is written to a delta disk (an ephemeral layer on top of the image layers) and volumes are independently mounted disks which can only be mounted to one container at a time. When creating a volume, you can specify the size of the disk and the volume store it gets deployed to. If you select a volume store backed by a shared datastore, that volume will be available to any container anywhere in the vSphere cluster. This is particularly useful when it comes to the live migration of stateful containers. The vSphere administrator will be responsible for backup policy associated with the datastore.

As such, vSphere Integrated Containers makes it easy to store persistent data to disks that are independent of VMs, can be written to shared datastores and can participate in the same backup and security policies as regular VMs.

Note that an anonymous volume declared in a Dockerfile will manifest as a mounted disk of a default size (1GB) to a default datastore. This is almost always going to be the wrong option in production for the reasons stated above.

You can use NFS to mount shared read-write volumes to container VMs.

Container Isolation

A container deployed to a VCH is strongly isolated by design. Strongly isolated means:

- The container gets its own Linux kernel which is not used for any other purpose
- The container gets its own filesystem and buffer cache which is not used for any other purpose
- The container cannot get access to the container control plane or get information about any other containers
- Privilege escalation or container breakouts in the conventional sense are not possible
- The container operates independent of its control plane (assuming port mapping is not being used)
- The container can take advantage of vSphere High Availability and vMotion

Network isolation is handled in a similar way to Docker, except that containers can be connected directly to vSphere port groups (see container networks). Storage isolation is discussed above.

This kind of strong isolation is best suited to a container workload that is a long-running service. If the service fails, it should have no impact on any other services. Examples of a long-running service are a database, web server, key-value store etc.

Containers are very flexible abstractions however and not every container is designed to be a single service. In fact, some containers are designed to be combined to form a single service and a single unit of scale. This notion is sometimes described as a Pod. In such a circumstance, it may be beneficial to run these as Linux containers in a single VM. vSphere Integrated Containers Engine provides built-in support for this model of provisioning Linux container hosts as vSphere Integrated Containers container VMs since 1.2.

What's important is to consider the policy needs of your application in terms of isolation. Strong isolation is a very important consideration in deploying robust applications into production and vSphere Integrated Containers makes it easy to turn that policy into plumbing.

Building and Deploying Single Containers to a Virtual Container Host

This section assumes that you already have a virtual container host (VCH) installed and that you are accessing it using TLS authentication.

For simplicity, pre-built Docker images are demonstrated to illustrate principles of operation. It is assumed that in reality you will have your own Docker images built.

This section will illustrate a number of useful capabilities such as pre-poluating data volumes, creating custom images and running daemon processes.

Deploying a Database - Postgres 9.6

All databases will have common requirements. A database should almost always be strongly isolated and long-running, so is a perfect candidate for a container VM. Steps to consider include:

1. Choose a volume store for your database state
2. Choose a size for your persistent volume
3. Choose a network for your container. Does it need to be exposed externally or privately to other containers?
4. How many CPUs and how much memory do you want for your database?

Note that the Dockerfile uses VOLUME and EXPOSE to illustrate that it needs to store persistent state and that you should be able to reach it on a particular port. As discussed here, anonymous volumes and random port mappings are fine for a sandbox, but not for production.

In this example, we create a 10GB volume disk on a backed up shared datastore. We'll use a private network to access the database, assuming that another container will need to access it privately. We use environment variables to set the data directory and password. We give the container a name so that it can be resolved using that name on the private network. Finally, we choose 2 vCPUs and 4GB of RAM.

```
docker network create datanet
docker volume create --opt Capacity=10G --opt VolumeStore=shared-backedup pgdata
docker run --name db -d -v pgdata:/var/lib/postgresql/data -e POSTGRES_PASSWORD=y7u8i9o0p --cpus 2 -m 4g --net datanet po
stgres:9.6
```

Once the container has started, you can use `docker ps` to make sure it's running. You can use `docker logs db` to see the logs. You can use `docker exec -it db /bin/bash` to get a shell into the container.

Now let's check that it's visible on the private network and it's running correctly. We can do this using a VIC container running on the same private network:

```
docker run --rm -it --net datanet postgres:9.6 /bin/bash
   $ ping db
     PING db (172.18.0.2): 56 data bytes
     64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.856 ms
     ...
   $ pg_isready -h db
     db:5432 - accepting connections
```

If we stop and delete the container, the data volume will persist. It will even persist beyond the lifespan of the VCH unless `vic-machine delete --force` is used.

Deploying an Application Server - Tomcat 9 with JRE 8

Looking at the Dockerfile here, there are no anonymous volumes specified. However, we need to consider how to get our application deployed and we may want to set some JVM configuration.

Let's start by deploying Tomcat on an external container network to make sure it works

```
docker run --name web -d -p 8080 -e JAVA_OPTS="-Dkey=value" --net ExternalNetwork tomcat:9
docker logs web
docker inspect web | grep IPAddress
curl <external-ip>:8080
```

Hopefully an index.html showing Tomcat server running is shown. Of course you can also test this using a browser. Note that you can pass JRE options in as an environment variable as per the example above.

Note also that a container VM already has an instance of the haveged service running to provide sufficient entropy for faster startup (see https://wiki.apache.org/tomcat/HowTo/FasterStartUp).

Next step is to consider how to get a webapp onto the application server. There are static and dynamic approaches to this problem.

Pre-populate a Volume

You can use a container to pre-populate a volume with a web application that you then bind when you run the application server. This is a late-binding dynamic approach that has the advantage that the container image remains general-purpose. The downside is that it requires an extra step to populate the volume.

```
docker volume create webapp
docker run --rm -v webapp:/data -w /data tomcat:9 curl -O https://tomcat.apache.org/tomcat-6.0-doc/appdev/sample/sample.w
ar
docker run --name web -d -p 8080 -v webapp:/usr/local/tomcat/webapps --net ExternalNetwork tomcat:9
curl <external-ip>:8080/sample/index.html
```

The volume is a disk of default size, in this case 1GB. The command to populate the volume mounts it at `/data` and then tells the container to use `/data` as the working directory. It then uses the fact that the Tomcat container has `curl` installed to download a sample web app as a WAR file to the volume. When the volume is mounted to `/usr/local/tomcat/webapps`, it replaces any existing webapps such as the welcome page and Tomcat runs just the sample app.

If you don't want the volume to completely replace the existing `/webapps` directory, you can modify the above example to extract the WAR file to the volume and then mount the volume as a subdirectory of webapps.

```
docker volume create webapp
docker run --rm -v webapp:/data -w /data tomcat:9 /bin/bash -c "curl -O https://tomcat.apache.org/tomcat-6.0-doc/appdev/s
ample/sample.war; unzip sample.war; rm sample.war"
docker run --name web -d -p 8080 -v webapp:/usr/local/tomcat/webapps/sample --net ExternalNetwork tomcat:9
curl <external-ip>:8080/sample/index.html
```

Note that running multiple commands on a container can be done using `/bin/bash -c`. There's a discussion below as to why this isn't necessarily ideal for a running service, but for chaining simple commands together, it works fine. Now, not only is your sample app available, but any other app baked into the image in `/usr/local/tomcat/webapps` is also available.

Build a custom image

Building a custom image allows you to copy the sample webapp into the container image filesystem and make some other improvements and upgrades while you're there. This then creates a single purpose container that runs the webapp(s) baked into it.

Note that vSphere Integrated Containers Engine does not have a native docker build capability. Containers should be built using Docker Engine and vSphere Integrated Containers Engine relies on the portability of the Docker image format to run them. In order to do this, the built image needs to be pushed to a registry that the VCH can access. This is one reason why such a registry is built into the vSphere Integrated Containers product.

Dockerfile:

```
FROM tomcat:9

ENV JAVA_OPTS "-Dkey=value"
COPY sample.war /usr/local/bin/webapps
```

In a VM running standard Docker Engine:

```
docker build -t <registry-address>/<project>/<image name> .
docker login <registry-address>
docker push <registry-address>/<project>/<image name>
```

From a Docker client attached to a VCH

```
docker run --name web -d -p 8080 --net ExternalNetwork <registry-address>/<project>/<image name>
```

Running Daemon Processes in a VIC container

Although a VIC container is a VM, it is a very opinionated VM in that it has the same constraints as a container. It doesn't have a conventional init system and its lifecycle is coupled to a single main process. There are a few ways of running daemon processes in a container - many of which are far from ideal.

For example, simply chaining commands in a Dockerfile `CMD` instruction techically works, but it compromises the signal handling and exit codes of the container. As a result, `docker stop` will almost certainly not work as intended. Let's imagine we want to run the `sshd` daemon in the background to grant users shell access into our web server, rather than giving them broader `docker exec` privileges.

```
FROM tomcat:9

RUN apt-get update;apt-get install -y openssh-server
COPY sample.war /usr/local/bin/webapps
CMD /usr/sbin/sshd && catalina.sh run
```

So this is not a recommended approach. Try running `docker stop` and it will timeout and eventually kill the container. This is not a problem exclusive to vSphere Integrated Containers Engine, this is a general problem with container images.

A much simpler approach is to run sshd using `docker exec` once the container is started:

```
docker run --name web -d -p 8080 -v webapp:/usr/local/tomcat/webapps --net ExternalNetwork <registry-address>/<image name
>
docker exec -d web /usr/sbin/sshd
```

Docker exec with the `-d` option runs a process as a daemon in the container. While this is arguably the neatest solution to the problem, it does require a subsequent call to the container after it's started. While it's relatively simple to script this, it doesn't work well in a scenario such as a Compose file.

So a third approach is to create a script that the container starts when it initializes that uses a trap handler to manage signals.

rc.local

```
#!/bin/bash

cleanup()
{
   kill $(pidof /docker-java-home/jre/bin/java)
}

trap cleanup EXIT

/usr/sbin/sshd
catalina.sh run
```

Dockerfile

```
FROM tomcat:9

RUN apt-get update;apt-get install -y openssh-server
COPY sample.war /usr/local/bin/webapps
CMD [ "/etc/rc.local" ]
COPY rc.local /etc/
```

Deploying a Development Environment

You can use VIC to run a development environment that can be used either interactively or as a means of running builds or test suites.

Let's look at some simple examples. Regardless of the approach, we'll need code mounted into the development environment. The simplest way to achieve this is using a volume. Let's download the VIC repository onto a volume.

```
docker volume create vic-build
docker run --rm -v vic-build:/build -w /build golang:1.8 git clone https://github.com/vmware/vic.git
```

Interactive

The source code tree lives on the persistent volume and can be re-used across invocations of the development environment. The command below will take you straight into a golang development environment shell.

```
docker run --rm -it -v vic-build:/go/src/github.com/vmware/ -w /go/src/github.com/vmware/vic golang:1.8
```

Running a Build

Let's build VIC using the volume created above. That's a simple matter of appropriately sizing the container VM and running `make`.

```
docker run --rm -m 4g -v vic-build:/go/src/github.com/vmware/ -w /go/src/github.com/vmware/vic golang:1.8 make all
```

The output of the build also lives on the volume. You need to ensure that the volume is big enough. vSphere Integrated Containers Engine supports NFS volume mounts which could be a great alternative for the build source and output.

Building and Deploying Multi-Container Applications to a Virtual Container Host

Having examined some of the considerations around deploying single containers to a virtual container host (VCH), this section examples how to deploy applications that are comprised of multiple containers.

There are two approaches you can take to this. The most instinctive approach would be to create scripts that manage the lifecycle of volumes, networks and containers.

The second approach is to use a manifest-based orchestrator such as Docker Compose. Docker Compose is a proprietary orchestrator that drives the Docker API and ties other pieces of the Docker ecosystem together including Build and Swarm. Given that vSphere Integrated Containers Engine doesn't currently support either Build or Swarm, Compose compatibility is necessarily limited. However, Compose can still be a useful tool, provided those limitations are understood.

Scripting Multi-Container Applications

Let's start by looking at how you would script Wordpress running in one container and a MySQL database in another. We can then use some of the considerations and topics discussed and apply that to the Compose example later.

As with the single container examples, we need to consider:

1. What persistent state needs to be stored and where should it go?
2. How should the containers communicate with each other?
3. Does each container need to be strongly isolated?
4. How should each container be sized?

For this example, we're going to create two named volumes on different vSphere datastores. Database state is going to a persistent volume on a shared datastore that's backed up and encrypted. The Wordpress HTML state is going to a shared datastore that's less expensive.

We're going to create a private network for the database and expose the Wordpress container on a second network that exposes a port on the VCH endpoint.

The Wordpress application server and the database container don't necessarily have to be separate failure domains, but one of the advantages of vSphere Integrated Containers Engine is that it makes it easy to deploy them that more secure way, so that's the approach we're taking here.

The question of sizing is a simple matter of setting virtual CPUs and memory on each container.

If we were to create a shell script to stand this up, it might look like this:

```
#!/bin/bash

DB_PASSWORD=wordpress
DB_USER=wordpress

WEB_CTR_NAME=web
```

```
DB_CTR_NAME=db

# pull the images first
docker pull wordpress
docker pull mysql:5.7

# create a persistent volume for the database
docker volume create --opt Capacity=4G --opt VolumeStore=backed-up-encrypted db-data
docker volume create --opt Capacity=2G --opt VolumeStore=default html-data

# create a private network for the web container to talk to the database. This will fail if the network already exists.
docker network create --internal db-net
docker network create web-net

# start the database container - specify a subdirectory on the volume as the data dir
docker run -d --name $DB_CTR_NAME --net db-net -v db-data:/var/lib/mysql --cpus 1 -m 2g -e MYSQL_ROOT_PASSWORD=somewordpr
ess -e MYSQL_DATABASE=$DB_PASSWORD -e MYSQL_USER=$DB_USER -e MYSQL_PASSWORD=wordpress mysql:5.7

# start the web container - note it resolves the database container by name over db-net
docker create --name $WEB_CTR_NAME --net web-net -p 8080:80 -v html-data:/var/www/html --cpus 2 -m 4g -e WORDPRESS_DB_HOS
T=$DB_CTR_NAME:3306 -e WORDPRESS_DB_USER=$DB_USER -e WORDPRESS_DB_PASSWORD=$DB_PASSWORD wordpress

docker network connect db-net $WEB_CTR_NAME

docker start $WEB_CTR_NAME

# check that the containers are up and look at the IP address and port of the web container
docker ps | grep "$WEB_CTR_NAME\|$DB_CTR_NAME"
```

A second script to shut down the two containers and clean up everything might look like this:

```
#!/bin/bash

docker stop web db
docker rm web db

# uncomment to delete volume state
# docker volume rm db-data html-data

# uncomment to delete networks
# docker network rm db-net web-net
```

Blocking on Container Readiness

In the above example, the Wordpress container waits for about 10 seconds for the database to come up and be ready. What if it needs to wait longer than that? This is one of the ways `docker exec` can be useful. For example:

```
# wait until the database is up
while true; do
   docker exec -it db mysqladmin --user=$DB_USER --password=$DB_PASSWORD version > /dev/null 2>&1
   if [ $? -eq 0 ]; then
      break
   fi
   sleep 5
done
```

It's worth noting that the MySQL docker hub page states:

```
If there is no database initialized when the container starts, then a default database will be created.
While this is the expected behavior, this means that it will not accept incoming connections until such initialization co
mpletes.
This may cause issues when using automation tools, such as docker-compose, which start several containers simultaneously.
```

The user of `docker exec` is the quickest and simplest mechanism you can use to execute a binary in a running container and test its return code. A cleaner solution might be to add your own custom script to the database image that blocks until the database is ready and then call that using `docker exec`. This eliminates the need to call `docker exec` in a sleep loop.

If you want to modify the Wordpress image to add a database connection test, you would have to create a script that the container will evoke that runs the test before running the main process and deals correctly with signal handling. See here for a discussion on ways to achieve this.

Running Multi-Container Applications Using Docker Compose

Before we get into the topic of ***building*** applications for Docker Compose, let's look at an example of how we would run the equivalent of the above script using Docker Compose and vSphere Integrated Containers engine.

Docker Compose serializes a manifest in a YML file which the `docker-compose` binary turns into docker commands. The equivalent of the above script as a Compose file would be the following:

```
version: '2'

services:
   db:
     image: mysql:5.7
     command: --datadir=/var/lib/mysql/data
     volumes:
       - db-data:/var/lib/mysql
     networks:
       - db-net
     environment:
       MYSQL_ROOT_PASSWORD: somewordpress
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD: wordpress

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8080:80"
     volumes:
       - html-data:/var/www/html
     networks:
       - web-net
       - db-net
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
```

```
      WORDPRESS_DB_PASSWORD: wordpress

volumes:
   db-data:
      driver: "vsphere"
      driver_opts:
         Capacity: "4G"
         VolumeStore: "backed-up-encrypted"
   html-data:
      driver: "vsphere"
      driver_opts:
         Capacity: "2G"
         VolumeStore: "default"

networks:
   web-net:
   db-net:
      internal: true
```

Note that there is no way to run `exec` commands explicitly in a compose file, so any waits for dependent services to come up need to be built into the containers themselves.

How to Manage the Application Lifecycle with docker-compose and vSphere Integrated Containers Engine

Assuming you've downloaded an appropriate version of the docker-compose binary, you need to point docker-compose at a VCH endpoint. This is done either by setting `DOCKER_HOST=<endpoint-ip>:<port>` or using `docker-compose -H <endpoint-ip>:<port>`.

Dependencies between the compose file and vic-machine configuration

Given that the VCH lifecycle is handled by a vSphere administrator, there may be some named resources in the VCH that need to be referenced in the Compose file. For example, in the Compose file above are the names of two volume stores. There may other assumptions, such as the name of a container network for example. As a user, it's important to know how to get this information from your VCH so that you can configure your Compose file appropriately.

To view a list of networks that have been pre-configured by the vSphere admin, use `docker network ls` and look for ones marked `external`.

To view a list of volume stores that have been pre-configured by the vSphere admin, use `docker info | grep VolumeStores`.

TLS Authentication

Assuming you're using TLS authentication to the Docker endpoint, that is either done using environment variables or command-line options.

With environment variables, it's assumed that you've already set `DOCKER_TLS_VERIFY=1` and `DOCKER_CERT_PATH=<path to client certs>`. This is required in order to use the Docker client. For `docker-compose` you have to additionally set `COMPOSE_TLS_VERSION=TLSv1_2`. You can then run `docker-compose up -d` to start the application (assuming you've also set `DOCKER_HOST` to point to the VCH endpoint).

Using command-line arguments with Docker client is a little more clumsy as each key has to be specified independently and the same is true of `docker-compose`. Regardless, the only way to specify the TLS version is through the environment variable above `COMPOSE_TLS_VERSION=TLSv1_2`. You can then run `docker-compose -H <endpoint-ip>:2376 --tlsverify --tlscacert="<local-ca-path>/ca.pem" --tlscert="<local-ca-path>/cert.pem" --tlskey="<local-ca-path>/key.pem" compose up -d`

Lifecycle Commands

The docker-compose binary is well documented and it is outside of the scope of this document to go into detail on that. However, given the example given above, the following lifecycle commands work:

```
docker-compose pull                    # pull the required images
docker-compose up -d                   # start the application in the background
docker-compose logs                    # see the logs of the containers started
docker-compose images                  # list the images in use
docker-compose stop                    # cleanly stop the running containers, leave container state
docker-compose kill                    # force kill of the container processes
docker-compose start                   # restart the application
docker-compose down                    # stop the application and remove the resources, leaving persistent volumes and im
ages
docker-compose down --volumes --rmi    # stop the application and remove all resources including volumes and images
```

Building Multi-Container Applications Using Docker Compose

Given that vSphere Integrated Containers Engine does not have a native build capability, it does not interpret the `build` keyword in a compose file and `docker-compose build` will not work when `DOCKER_HOST` points to a VIC endpoint. vSphere Integrated Containers Engine relies upon the portability of the docker image format and it is expected that a regular Docker Engine will be used in a CI pipeline to build container images for test and deployment.

There are two ways to work around this. You can create separate Compose files for build and run, or you can use the same Compose file but just make sure to add a couple of arguments. We will explore both options here using another example of a Compose file that includes build instructions. In this case, the sample voting application found here.

Let's start by cloning the repository: `git clone git@github.com:dockersamples/example-voting-app.git` and we'll start by looking at `docker-compose-simple.yml`.

Using separate Compose files

You can strip a Compose file down to an absolute minimum if you want to use it just for building and pushing images. If you want to run the application on a VIC endpoint, you'll need to also push the built images to a docker registry visible to your VCH, so that they can be deployed. In order to do that, we need to add `image` directives to the Compose file.

```
$ more docker-compose-simple-build.yml
version: "2"

services:
  vote:
    build: ./vote
    image: <registry-address>/<project>/vote:0.1
```

```
  worker:
    build: ./worker
    image: <registry-address>/<project>/worker:0.1

  result:
    build: ./result
    image: <registry-address>/<project>/result:0.1

$ sudo docker-compose -f docker-compose-simple-build.yml build
$ sudo docker login <registry>
$ sudo docker-compose -f docker-compose-simple-build.yml push
```

Now that the application is built and pushed, you need to create a second Compose file for deployment that reflects the deployment considerations discussed earlier in terms of isolation, peristent volume state, networking etc. The Compose file provided in the repo is simply an example and you would typically expect to have to change it to suit your needs. Let's do that, but keep it as simple as possible to begin with.

Modifications from the original file are highlighted as comments

```
version: "2"      # vSphere Integrated Containers Engine supports Compose file version 2

services:
  vote:
    image: <registry-address>/<project>/vote:0.1    # Fully-qualified image name
    command: python app.py
    ports:                          # Local ./vote volume mount removed - use the app.py built-in
      - "5000:80"

  redis:
    image: redis:alpine
    ports: ["6379"]

  worker:
    image: <registry-address>/<project>/worker:0.1    # Fully-qualified image name

  db:
    image: postgres:9.4

  result:
    image: <registry-address>/<project>/result:0.1    # Fully-qualified image name
    command: nodemon --debug server.js
    ports:                          # Local ./results volume mount removed - use the server.js built-in
      - "5001:80"
      - "5858:5858"
```

Let's review the changes that were made to this Compose file.

- Fully qualified image name

In most real-world scenarios, container images will be pushed to a registry before they're deployed into production. That means that the registry and a project will be part of the image name. The only way it will run with just the container name is if it has been built locally.

- Removed local volume mappings

Local volume mounts are useful for development and testing as they allow source trees and data to be easily mapped into a container. In production however, making a container host stateful for the purpose of seeding the container with configuration or application data is only feasible if the container is guaranteed to be deployed to the stateful host. In general, best practice is to keep a container host as stateless as possible.

vSphere Integrated Containers Engine cannot map volumes from a local filesystem into a container because vSphere Integrated Containers Engine containers are strongly isolated and don't share a common filesystem. Despite this, it is still possible in VIC to add state to a container by pre-populating a volume with data and mounting it (TBD: link to "Pre-populate a Volume").

Combining into a single Compose file

If separate Compose files feels clunky, it's quite possible to build, push and run from the same Compose file. All we need to do is to merge them together and then make sure we tell docker-compose what we want. Here's an example of a merged file:

```
version: "2"

services:
  vote:
    build: ./vote
    image: <registry-address>/<project>/vote:0.1
    command: python app.py
    ports:
      - "5000:80"

  redis:
    image: redis:alpine
    ports: ["6379"]

  worker:
    build: ./worker
    image: <registry-address>/<project>/worker:0.1

  db:
    image: postgres:9.4

  result:
    build: ./result
    image: <registry-address>/<project>/result:0.1
    command: nodemon --debug server.js
    ports:
      - "5001:80"
      - "5858:5858"
```

Build and push work in just the same way as the previous example. The rest of the directives are ignored.

In order to deploy this to a VIC endpoint however, you need to first explicitly pull the images. Otherwise docker-compose will try to build them, even if you attempt to run with `--no-build`. Then you run the Compose file with `--no-build` to tell docker-compose to ignore the build directives.

```
$ sudo docker-compose -f docker-compose-simple-vic.yml build
$ sudo docker-compose -f docker-compose-simple-vic.yml push
$ docker-compose -f docker-compose-simple-vic.yml pull
$ docker-compose -f docker-compose-simple-vic.yml up --no-build -d
```

In the example above, the use of `sudo` creates a child shell that runs a local Docker Engine and bypasses the environment variables configured to make docker-compose talk to a VIC endpoint. In this way, it's possible to do a build, push, pull and run from the same shell using the same client.

A Summary on Compatibility

Given that VIC is designed to be an enterprise runtime and has unique isolation characteristics applied to the containers it deploys, a Docker Compose script downloaded from the web may not work without modification.

This is partly a question of functional completeness of vSphere Integrated Containers Engine docker API support and partly a question of its inherent design. There are some highly detailed technical sections in the documentation highlighting all of the capabilities vSphere Integrated Containers Engine currently supports, but here is a high-level summary of topics discussed in more detail above:

- vSphere Integrated Containers Engine supports version 2 of the Compose File format.
- vSphere Integrated Containers Engine has no native build support.
- VIC containers take time to boot and thus may exhibit timing related issues. Eg. You may need to set `COMPOSE_HTTP_TIMEOUT` to a higher value than the default.
- VIC containers have no notion of local read-write shared storage.

One of the main reasons this section takes you through all the considerations of putting a multi-container application into production with the Docker client prior to introducing Docker Compose is to help you understand how to configure Compose to work with the capabilities of VIC. Trying to work the opposite way around, by trying to configure VIC to work with capabilities of Compose may be trickier for the reasons stated.