Chapter 7. Multi-container Applications with Compose

Abstract

Goal	Run multi-container applications with Podman Compose.
Objectives	• Describe compose and its common use cases. • Configure a repeatable developer environment with Compose.
Sections	• Compose Overview and Use Cases (and Quiz) • Build Developer Environments with Compose (and Guided Exercise)
Lab	• Multi-container Applications with Compose

Compose Overview and Use Cases

Objectives

- Describe compose and its common use cases.

Orchestrate Containers with Podman Compose

In today's computing, many applications are developed as microservices. In the context of containers, developers manage each microservice as an independent container image. Because one application is composed of several containers, it might become difficult to manage the containers as a group.

For example, to develop a new microservice, developers commonly implement one of the following strategies:

- Create a *mock* for the services that your application requires, such as API endpoints that other microservices provide.
- Start the relevant parts of the application on a local machine.

It is ideal to run the full application on the local machine. When doing so is not possible, developers containerize mock services that approximate production environment.

Podman Compose is an open source tool that you can use to run *Compose files*. A Compose file is a YAML file that specifies the containers to manage, as well as the dependencies among them.

For example, consider the following Compose file:

```
services:
  orders: (1)
    image: quay.io/user/python-app (2)
    ports:
    - 3030:8080 (3)
    environment:
      ACCOUNTS_SERVICE: http://accounts (4)
```

(1) Declare the `orders` container.

(2) Use the `python-app` container image.

(3) Bind the `3030` port on your host machine to the `8080` port within the container.

(4) Pass the `ACCOUNTS_SERVICE` environment variable to the application.

Podman Compose complies with the *Compose Specification*, which defines a YAML-based schema to manage multi-container applications for container runtimes, such as Podman.

Although Compose files should work with any container runtime, some features might depend on specific implementations of the runtime. You might find Compose files called `docker-compose.yaml`. Docker introduced the `docker-compose.yaml` naming convention when it started the Compose project. However, Docker open sourced the Compose Specification, which also specifies the Compose file naming convention. According to the Compose Specification, the preferred name is `compose.yaml`.

Podman Compose became popular for the following uses:

Development environments
You can approximate a production environment on your local machine by providing applications, databases, or cache systems configuration in a single file.

Consequently, developers can automate complex multi-container deployments with one file.

Automated testing
You can define several test suites, along with their configurations and environment variables, within the same file. In addition, you might approximate an automated pipeline environment on a single machine.

Consequently, developers can execute their applications with multiple isolated database environments, and clients. Developers can also debug complex issues in their local environment, which increases development speed.

Using Podman Compose in a production environment is not recommended. Podman Compose does not support advanced features that you might need in a production environment, such as load balancing, distributing containers to multiple nodes, or managing containers on different nodes.

If you need a production container orchestration solution, then Kubernetes or Red Hat OpenShift is a better option. Red Hat OpenShift can run multi-container applications on different host machines, or *nodes*.

Podman Pods

Podman introduced an alternative to orchestrate containers on a single host with the usage of Podman pods.

Podman pods can use and produce Kubernetes YAML files. These Kubernetes YAML files are a declarative way to define and manage multiple containers and their resources.

The `podman generate kube` command generates a Kubernetes YAML file from existing Podman containers, pods, and volumes.
The `podman play kube` command reads the Kubernetes YAML file and then recreates the defined resources on a local machine. These commands help to ensure that a container that is developed with Podman can migrate and work in a Kubernetes ecosystem.

The Compose File

The Compose file is a YAML file that contains the following sections:

- `version` (deprecated): Specifies the Compose version used.
- `services`: Defines the containers used.
- `networks`: Defines the networks used by the containers.
- `volumes`: Specifies the volumes used by the containers.
- `configs`: Specifies the configurations used by the containers.
- `secrets`: Defines the secrets used by the containers.

The `secrets` and `configs` objects are mounted as a file in the containers.

WARNING

In YAML files, the number of spaces carries meaning.

For example, consider the following YAML snippet:

```
version: "3.9"
services:
backend: {}
```

The previous YAML snippet carries a different meaning from the following snippet:

```
version: "3.9"
services:
  backend: {}
```

The `backend` keyword is preceded by two spaces; therefore the `backend` object is an attribute of the `services` object.

The following Compose file defines the `backend` and `db` services. It overrides the default command for the backend image by specifying the `command` property. Finally, the Compose file configures the environment variables that are required to start the database container.

NOTE

This section uses the terms *service* and *container* interchangeably.

```
services:
  backend:
    image: quay.io/example/backend
    ports:
      - "8081:8080"
    command: sh -c "COMMAND"
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_ADMIN_PASSWORD: redhat
```

Start and Stop Containers with Podman Compose

You can execute a Compose file by using the `podman-compose up` command, which creates the defined objects, such as volumes or networks, and starts the defined containers. If the compose manifest contains no network definition, then podman-compose creates a default network to ease communication for the services that the manifest defines.

```
[user@host ~]$ podman-compose up
['podman', '--version', '']
using podman version: 4.2.0
** excluding:  set()
['podman', 'network', 'exists', 'user_default'] 1
['podman', 'network', 'create', '--label', 'io.podman.compose.project=user', '--label', 'com.docker.compose.project=user', 'u
ser_default'] 2
...output omitted...
```

1 Check whether the `user_default` network exists.

2 Create the `user_default` Podman network.

You can list objects that are defined in the Compose file by using the `podman` command, such as `podman network ls` to list Podman networks.

```
[user@host ~]$ podman network ls
NETWORK ID    NAME            VERSION     PLUGINS
cd32c76f42f9  user_default    0.4.0       bridge,portmap,firewall,tuning,dnsname
```

Podman generates predictable names for objects that are not explicitly named. In the preceding example, Podman creates a default network for the application.

The default network naming convention uses the current directory name and the `_default` suffix. The preceding example contains the network `user_default` because it shows a Compose file executed in the `/home/user` home directory.

The default container naming convention uses the *`DIRECTORY_SERVICE_NUMBER`* format. Because the preceding example does not configure the container name, it creates the `user_db_1` container.

You can override the default naming conventions by explicitly configuring object names.

The following list shows common `podman-compose up` command options in their short and long versions:

- `-d`, `--detach`: Start containers in the background.
- `--force-recreate`: Re-create containers on start.
- `-V`, `--renew-anon-volumes`: Re-create anonymous volumes.
- `--remove-orphans`: Remove containers that do not correspond to services that are defined in the current Compose file.

The `podman-compose stop` command stops running containers that are defined as services. Execute `podman-compose down` to stop and remove containers that are defined as services.

```
[user@host ~]$ podman-compose down
['podman', '--version', '']
using podman version: 4.2.0
** excluding:  set()
podman stop -t 10 compose_db_1
compose_db_1
exit code: 0
podman rm compose_db_1
aab3...b412
exit code: 0
...output omitted...
```

Podman preserves objects other than containers, such as networks and volumes, so that containers do not lose their local state when restarted.

Podman also provides a `podman compose` wrapper around the `podman-compose` command.

Networking

Use the `networks` keyword at the same indentation level as the `services` keyword to create and use Podman networks. If the `networks` keyword is not defined in the Podman Compose file, and then Podman Compose creates a default DNS-enabled network.

Consider the following Compose file that declares three containers: a front-end application, a back-end application, and a database.

```
services:
  frontend:
    image: quay.io/example/frontend
    networks: (1)
      - app-net
    ports:
      - "8082:8080"
  backend:
    image: quay.io/example/backend
    networks: (2)
      - app-net
      - db-net
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_ADMIN_PASSWORD: redhat
    networks: (3)
      - db-net

networks: (4)
  app-net: {}
  db-net: {}
```

(1) The `frontend` service is part of the `app-net` network.

(2) The `backend` service is part of the `app-net` and `db-net` networks.

(3) The `db` service is part of the `db-net`.

(4) Definition of the networks.

> **NOTE**
>
> The open and closed curly braces (`{}`) mark an empty object. For example, the `app-net: {}` and `app-net:` definitions are identical. Both definitions signal the creation of the `app-net` network that uses the default configuration.

Services can interact with any non-isolated container, but only can resolve names if they share at least one network. Consequently, the `frontend` service cannot communicate directly with the `db` service. Isolating network traffic provides security advantages, because you can prevent specific containers from accessing protected data.

Volumes

You can declare volumes by using the `volumes` keyword. Mount the volumes in containers by using the *`VOLUME_NAME`*`:`*`CONTAINER_DIRECTORY`*`:`*`OPTIONS`* syntax.

```
services:
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_ADMIN_PASSWORD: redhat
    ports:
      - "5432:5432"
    volumes: (1)
      - db-vol:/var/lib/postgresql/data

volumes: (2)
  db-vol: {}
```

(1) The `db` service maps the `db-vol` volume to the `/var/lib/postgresql/data` directory within the container.

(2) Declaration of volumes.

If you created a volume that is not managed by Podman Compose, for example, because you used the `podman volume create` command, then you can specify it in the volumes definition.

```
services:
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_ADMIN_PASSWORD: redhat
    ports:
      - "5432:5432"
    volumes:
      - my-volume:/var/lib/postgresql/data

volumes:
  my-volume:
    external: true
```

You can also define bind mounts by providing a relative or absolute path on your host machine. In the following example, Podman mounts the `./local/redhat` directory on the host machine as the `/var/lib/postgresql/data` directory in the container.

```
services:
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_ADMIN_PASSWORD: redhat
    ports:
      - "5432:5432"
    volumes:
      - ./local/redhat:/var/lib/postgresql/data:Z
```

REFERENCES

GitHub - Podman Compose

GitHub - Compose Spec

Podman Compose Vs Docker Compose