Build Developer Environments with Compose

Objectives

- Configure a repeatable developer environment with Compose.

Podman Compose and Podman

Podman Compose translates Compose files into Podman CLI commands. Through translated commands, Podman Compose interacts directly with Podman and does not communicate with Podman's API socket. In contrast, Docker Compose communicates directly with the Docker daemon, which runs as root, by using its REST API.

By not interacting with the Podman's API socket, Podman Compose removes the need to run the Podman service that provides the API socket, saving resource consumption and providing a more native and lightweight solution for Podman users. Additionally, because it interacts with Podman directly, Podman Compose has better support for rootless containers than Docker Compose.

Before you use Podman Compose, install a compatible Podman version on your system. This course uses Podman 5.2.2.

You can confirm the Podman version with the `podman --version` command.

```
[user@host ~]$ podman --version
podman version 5.2.2
```

Apart from being available as a regular package in your distribution, Podman Compose is also available as a Python package. You can install it with the `pip` command, as follows:

```
[user@host ~]$ pip install podman-compose
```

You can also install the latest development version from the Podman Compose GitHub repository:

```
[user@host ~]$ pip install https://github.com/containers/podman-
compose/archive/devel.tar.gz
```

Podman 5 introduced the `podman compose` subcommand as a thin wrapper around an external compose provider such as docker-compose or podman-compose. This means that for example, you must have `podman-compose` installed for the subcommand to work.

Multi-container Developer Environments with

Compose

With Podman Compose, you can declaratively configure the services that you need to develop your application. You can place the required services into the Compose file inside your application repository so that developers can start these services by issuing a single `podman-compose up` command. Running your development environment dependencies like this isolates the developer from shared developer environments. Also, this predefined configuration saves developers from having to locally configure services such as databases or external dependencies.

For example, to develop a back-end application, you can use Podman Compose and a single Compose file to deploy a development environment that includes a PostgreSQL database container and a pgAdmin interface container to manage the database. You can mount data loading scripts for the database to provide developers with the required development data. You can specify the service dependencies by using the `depends_on` property followed by a list of services that need to start first.

```
services:
  database:
    image: "registry.redhat.io/rhel9/postgresql-13"
    container_name: "appdev-postgresql"
    volumes:
      - ./scripts/database/initial-data:/opt/app-root/src/postgresql-start:Z
...output omitted...
  database-admin:
    image: "registry.connect.redhat.com/crunchydata/crunchy-pgadmin4:ubi8-4.30-1"
    container_name: "appdev-pgadmin"
    depends_on:
     - database
...output omitted...
```

The previous example demonstrates that the `database-admin` container must start after the `database` container.

If your application depends on external services, then you can add a mock server to your Compose file. You can provide the mock server endpoints and fixed responses as configuration files in your repository. Then you can bind mount these files by using relative paths to your project directory within your Compose file.

```
services:
...output omitted...
  mock_service:
    image: "MOCK_SERVICE_IMAGE"
    volumes:
      - ./mocks/SERVICE/ENDPOINTS:TARGET_DIRECTORY:Z
      - ./mocks/SERVICE/FIXED_RESPONSES:TARGET_DIRECTORY:Z
...output omitted...
```