Managing pods in Podman using REST API

Contents

1. Pods overview

This section provides basic information on pods and their usage.

A pod in Podman is a single container or a group of containers with shared resources. Pods support multiple processes and share networking and storage resources to allow communication between containers and coordinate their termination. You can run separate containers inside a pod for a single application for the frontend, backend, and database. The pod also can provide an isolated environment not connected to an external network. Besides containers created and assigned by the user, Podman always creates an infrastructure container for the pod. It provides means to start and stop containers assigned to the pod.

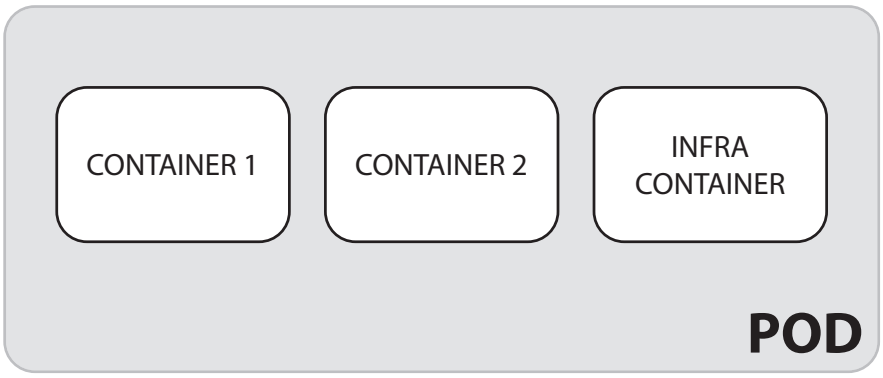

Figure 1: Structure of a pod with two containers created by a user and an infrastructure container

The pod concept in Podman is based on the smallest deployable software units you can create and manage in Kubernetes.

Additional resources:

- Overview of all actions available to manage pods

Read more:

- Podman overview
- Containers overview

1.1. Podman overview

This section provides basic information on the Pod Manager tool (Podman).

Podman is an open-source tool designed to create and manage pods and containers. It can be used as a drop-in replacement for docker but has unique features. Podman is:

- ***daemonless*** - it launches pods and containers as child processes rather than a program running in the background.
- ***rootless*** - you don't need root privileges to create and manage pods and containers.
- ***safe*** - while pods and containers don't have root privileges, they provide a barrier against external attackers.
- ***modular*** - it relies on other tools to build and manage container images.

1.2. Containers overview

This section provides basic information on container and containers images.

Containers provide an isolated environment for applications. A typical container consists of application binary, file system, environment settings, and needed libraries.

Containers are runtime instances of container images, which are executable files designed to be platform-independent and organized in a layered fashion. Images can be downloaded as ready-to-use executables from a registry, a service that stores container images. According to the user's needs, they can be used directly or modified. You also can create your images from scratch.

2. Podman REST API overview

This section provides basic information on the REST API implemented in Podman.

Application programming interface (API) provides a simple method of communication between products or services. Software developers mostly use it to integrate new applications into existing architecture.

Podman REST API is split into two layers:

- Compatible with Docker called `Compat API`
- Native API called `Libpod API` providing access to additional features not available in Docker, such as pods

New users should use native `Libpod API` when starting working with Podman. `Compat API` is provided for the current Docker users to help adopt Podman instead of Docker.

3. Creating pods using REST API

This procedure demonstrates creating a pod with two containers using native REST API on Fedora Linux. It is based on a typical example of creating a UI application with an associated database.

Prerequisites

- `podman` and `curl` packages are installed.

```
$ sudo dnf install podman curl
```

- Podman manually started as a service with user privileges.

```
$ podman system service -t 0 &
```

- Container images `wordpress` and `mariadb` are downloaded.

> **Note:** You can download container images using the CLI interface or REST API.

Procedure

1. Create a configuration file for the `mariadb` database container named `my-pod.conf`.

```
{
    ”portmappings”: [
    {
        ”container_port”: 80,
        ”host_port”: 8080
    }
    ],
    ”name”: ”my-pod”
}
```

> **Note:**
>
> - Configuration file contains only port mapping and a pod name.
> - There are more fields that can be filled. If not supplied, they receive default values.

2. Create an empty pod by sending `POST` request with `content-type:application/json` header to the `libpod/pods/create` endpoint, with `my-pod.conf` as a configuration file.

```
$ curl -XPOST --unix-socket /run/user/${UID}/podman/podman.sock \
    -H content-type:application/json \
    http://d/v3.0.0/libpod/pods/create -d @my-pod.conf
```

> **Note:**
>
> - `${UID}` is the bash variable that returns the current user ID
> - `/run/user/${UID}/podman/podman.sock` is the socket address of a podman service ran with the user privileges.

Expected output: JSON structure with pod ID.

3. Create a configuration for the `mariadb` database container named `mariadb.conf`.

```
{
    ”image” : ”wordpress”,
    ”env”: {
        ”MYSQL_ROOT_PASSWORD”: ”w0rdpr3ss”,
        ”MYSQL_DATABASE”: ”wp”,
        ”MYSQL_USER” : ”wordpress”,
        ”MYSQL_PASSWORD” : ”w0rdpr3ss”
    },
    ”restart_policy”: ”always”,
    ”pod”: ”my-pod”,
    ”name”: ”mariadb”
}
```

> Note:
>
> - `env` section contains environment variables that will be passed to the container.
> - In the case of a failure, the container will be restarted due to `restart_policy` set to `always`.

4. Create a container named `mariadb` in the existing `my-pod` pod by sending `POST` request to the `libpod/containers/create` endpoint, with `mariadb.conf` as a configuration file.

```
$ curl -XPOST --unix-socket /run/user/${UID}/podman/podman.sock \
    -H content-type:application/json \
    http://d/v3.0.0/libpod/containers/create -d @mariadb.conf
```

Expected output: JSON structure with container ID and warnings list.

5. Create a configuration file for the `wordpress` container named `wordpress.conf`.

```
{
    ”image” : ”wordpress”,
    ”env”: {
        ”WORDPRESS_DB_NAME”: ”wp”,
        ”WORDPRESS_DB_USER”: ”wordpress”,
        ”WORDPRESS_DB_PASSWORD” : ”w0rdpr3ss”,
        ”WORDPRESS_DB_HOST” : ”127.0.0.1”
    },
    ”pod”: ”my-pod”,
    ”name”: ”wordpress”
}
```

6. Create a container named `wordpress` in the existing `my-pod` pod by sending `POST` request to the `libpod/containers/create` endpoint, with `wordpress.conf` as a configuration file.

```
$ curl -XPOST --unix-socket /run/user/${UID}/podman/podman.sock \
    -H content-type:application/json \
    http://d/v3.0.0/libpod/containers/create -d @wordpress.conf
```

Expected output: JSON structure with container ID and warnings list.

7. Start `my-pod` pod by sending `POST` request to the `libpod/pods/my-pod/start` endpoint.

```
$ curl -XPOST --unix-socket /run/user/${UID}/podman/podman.sock \
    -H content-type:application/json \
    http://d/v3.0.0/libpod/pods/my-pod/start
```

Expected output: JSON structure with pod ID and a potential error field.

8. Optional: Inspect in your web browser `wordpress` application running on `http://localhost:8080/`
   Expected output:

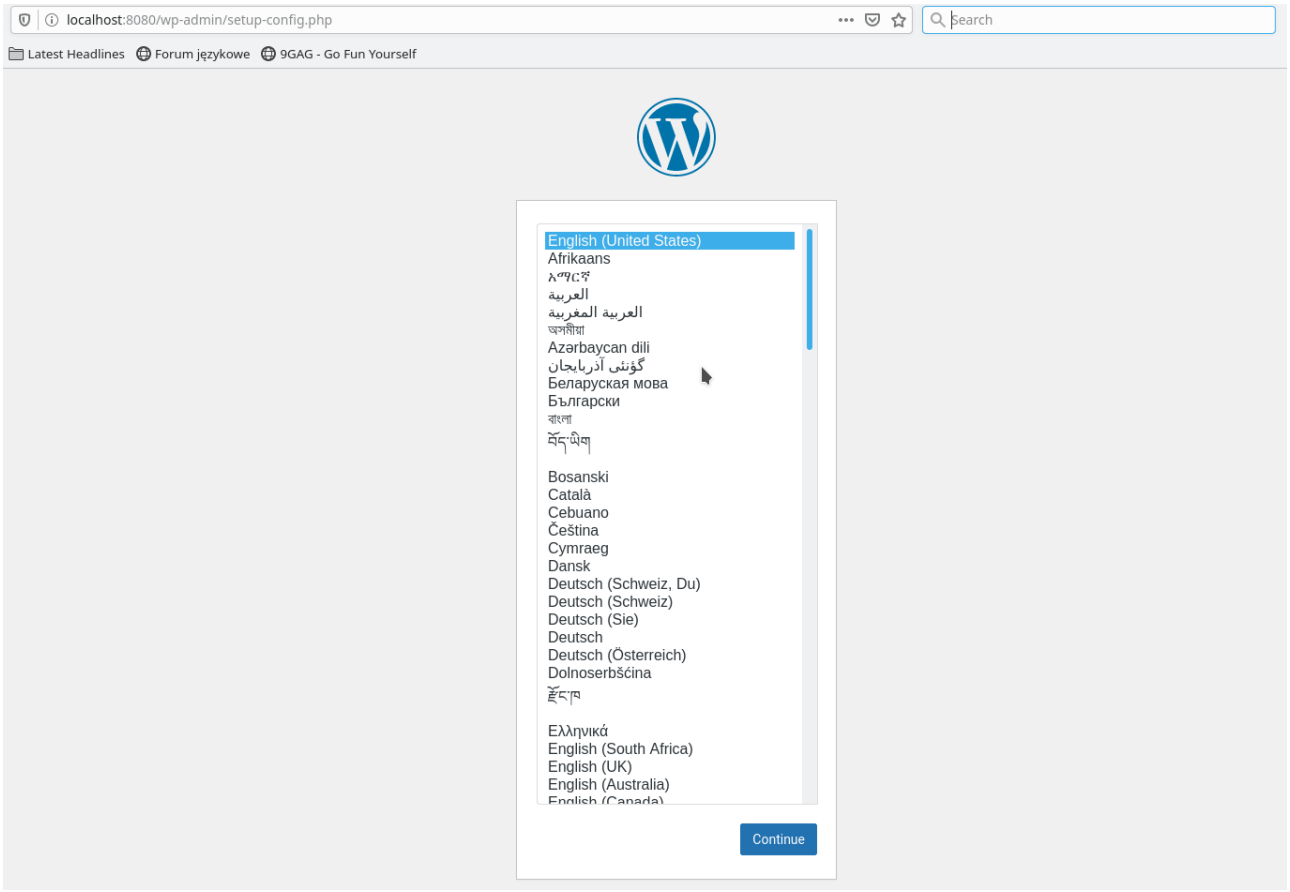

Figure 2: WordPress running inside a pod

Additional resources

- `podman-pod-create` man page
- Podman API documentation on creating and starting a pod
- Podman API documentation on creating a container

4. Getting pod information using REST API

This procedure demonstrates how to get pod information using native REST API on Fedora Linux.

Prerequisites

- `podman`, `curl`, and `jq` packages are installed.

```
$ sudo dnf install podman curl jq
```

- Podman manually started as a service with user privileges.

```
$ podman system service -t 0 &
```

- At least one pod has been created.

Procedure

1. List processes running inside `my-pod` pod by sending `GET` request to the `libpod/pods/my-pod/top` endpoint.

   ```
   $ curl --unix-socket /run/user/${UID}/podman/podman.sock \
   http://d/v3.0.0/libpod/pods/my-pod/top | jq
   ```

   Expected output: JSON structure with running command name and other important information on running processes like `PID`, `USER` or CPU usage.

2. Display information describing `my-pod` pod by sending `GET` request to the `libpod/pods/my-pod/json` endpoint.

   ```
   $ curl --unix-socket /run/user/${UID}/podman/podman.sock \
   http://d/v3.0.0/libpod/pods/my-pod/json |jq
   ```

   Expected output: JSON structure with all the information describing a pod, such as a name, creation timestamp, number of containers, state of containers, and more.

Additional resources

- `podman-pod-inspect` and `podman-pod-top` man pages
- Podman API documentation on listing pod processes and inspecting a pod

5. Stopping pods using REST API

This procedure demonstrates how to stop a pod using native REST API on Fedora Linux.

Prerequisites

- `podman` and `curl` packages are installed.

  ```
  $ sudo dnf install podman curl
  ```

- Podman manually started as a service with user privileges.

  ```
  $ podman system service -t 0 &
  ```

- At least one pod has been created.

Procedure

1. Stop `my-pod` pod using REST API by sending `POST` request to the `libpod/pods/my-pod/stop` endpoint.

```
$ curl -XPOST --unix-socket /run/user/${UID}/podman/podman.sock \
-H content-type:application/json \
http://d/v3.0.0/libpod/pods/my-pod/stop
```

Expected output: JSON structure with pod ID and a potential error field.

2. Optional: You can remove stopped `my-pod` pod using REST API if no longer needed.

```
$ curl -XDELETE --unix-socket /run/user/${UID}/podman/podman.sock \
-H content-type:application/json \
http://d/v3.0.0/libpod/pods/my-pod
```

Expected output: JSON structure with pod ID and a potential error field.

Additional resources

- `podman-pod-stop` and `podman-pod-rm` man pages
- Podman API documentation on stopping and removing a pod