Chapter 6. Troubleshooting Containers

Abstract

Goal	Analyze container logs and configure a remote debugger.
Objectives	• Read container logs and troubleshoot common container problems. • Configure a remote debugger during application development.
Sections	• Container Logging and Troubleshooting (and Guided Exercise) • Remote Debugging Containers (and Guided Exercise)
Lab	• Troubleshooting Containers

Container Logging and Troubleshooting

Objectives

- Read container logs and troubleshoot common container problems.

Troubleshoot Container Startup

A container might not be able to start successfully for different reasons, for example, due to missing configuration or a file access issue.

Depending on the containerized application, the process executed by the container can either exit with an error status or keep running in an inconsistent state. You can list running and stopped containers by using the `podman ps -a` command.

```
[user@host ~]$ podman ps -a
CONTAINER ID   IMAGE    COMMAND  CREATED    STATUS      PORTS   NAMES
b72652504844   ...              28 sec...  Up...               ok_service
867f6f559629   ...      /bin...  21 sec...  Exited...   0...    failing_service
```

If the container is in the `Exited` status, then the problem might be in the start-up process. Many applications output error information when they encounter an issue during startup. To access this information, you can use the `podman logs` command.

```
[user@host ~]$ podman logs CONTAINER
...output omitted...
```

If you want to follow the logs in real time, then you can add the `-f` option to the preceding command.

Troubleshoot Container Networking

Common container networking issues include the following situations:

- Incorrect port mapping.
- No network access between containers.
- Hostname resolution problems (DNS).

Port Mapping Issues

Developers must distinguish between the following port configuration:

- Ports that the application inside the container listens on.
- Podman port mapping configuration, which maps ports on the host to the application ports inside the container.

If the port that you assign to the container in the port mapping configuration does not match the application port, then the containerized application fails to communicate with the host.

You can use the `podman port CONTAINER` command to list the current container port mapping.

```
[user@host ~]$ podman port CONTAINER
8000/tcp -> 0.0.0.0:8080
```

In the preceding example, a container exposes the `8000` container port to the host machine on the `8080` host port at all the host network interfaces.

Applications might listen on a different port than you expect. Because Podman commands do not interact with the application, you cannot use Podman commands to verify the port that the application uses.

To verify the application ports in use, list the open network ports in the running container. Use Linux commands such as the socket statistics (`ss`) command to list open ports. A socket is the combination of a port and an IP address. The `ss` command lists the open sockets in a system. You can provide the `ss` command with options to filter and produce the desired output:

- `-p`: display the process using the socket
- `-a`: display listening and established connections
- `-n`: display numeric ports instead of mapped service names
- `-t`: display TCP sockets

```
[user@host ~]$ podman exec -it CONTAINER ss -pant
Netid State  ... Local Address:Port   Peer Address:Port   Process
tcp   LISTEN ...       0.0.0.0:9091        0.0.0.0:*      users:("python",pid=...)
...output omitted...
```

Use the output of the `ss` command to verify that your port mapping matches the application ports in use. You can use values such as the local address and port, the peer address and port, and the process. In the preceding example, a Python service is listening on port 9091. Therefore, to expose this service to the host machine, make sure that a port mapping exists for this port.

To reduce the container attack surface, containers usually lack many commands. You can run the host system commands within the container *network namespace* by using the `nsenter` command. A container namespace is how the Linux kernel isolates a system resource view from the container perspective. For example, a container network namespace shows the system's networking stack to the container as if the container was the only process using the stack. There are other types of namespaces, but this material is out of scope for the course.

You can target a container namespace by using the `nsenter -n -t` command with the container process ID (PID).

To get the container PID you can use the following `podman inspect` command:

```
[user@host ~]$ podman inspect CONTAINER --format '{{.State.Pid}}'
CONTAINER_PID
```

After getting the container PID, you can run the `nsenter` command. Run the command with elevated privileges by using `sudo`, as follows:

```
[user@host ~]$ sudo nsenter -n -t CONTAINER_PID ss -pant
Netid State  ... Local Address:Port   Peer Address:Port   Process
tcp   LISTEN ...       0.0.0.0:9091        0.0.0.0:*      ...
...output omitted...
```

> **NOTE**
>
> Containerized applications should listen on the 0.0.0.0 address, which refers to any network interface within the container. Using the 127.0.0.1 loopback interface isolates the application from communicating outside of the container.

Container Network Connectivity Issues

Developers commonly use Podman networks to isolate container network traffic from the host. If a container does not attach to a Podman network, then it cannot use podman networking for communicating with other containers.

You can use the `podman inspect` command to verify that every container is using a specific network.

```
[user@host ~]$ podman inspect CONTAINER --format='{{.NetworkSettings.Networks}}'
map[network_name:0xc000a825a0]
```

When containers communicate by using Podman networks, there is no port mapping involved.

Name Resolution Issues

Podman networks provide connectivity for container-to-container network traffic by using IP addresses. However, because IP addresses might change, developers use hostnames to address these containers in a predictable way. The Domain Name System (DNS) service translates container names to IP addresses for network communication. If the network in use does not have DNS enabled, then containers cannot resolve the container hostnames and cannot communicate.

To ensure that DNS is enabled for a Podman network use the `podman network inspect` command.

```
[user@host ~]$ podman network inspect NETWORK
...output omitted...
  "dns_enabled": true,
...output omitted...
```

Podman Events

When troubleshooting container issues, the initial step is to gather information. You typically start by examining the container logs. However, this approach assumes that the container was operational for sufficient time to produce logs. If the container had never started, there would have been no logs. In such a situation, you can use events to obtain more data. Events provide supplementary information in addition to the container logs.

Podman includes an events system that records activities. Podman monitors objects such as containers, images, pods, and volumes. Podman also tracks event types such as create, start, stop, pull, or remove. For instance, with events, you can follow when you create a container or pull an image. Monitor and view events in Podman by using the `podman events` command.

Podman requires a backend logging mechanism for recording events. By default, the logging mechanism used is journald.
The `events_logger` field in the `containers.conf` file controls this behavior. The available logging methods are file, journald, and none.

Use the `podman info` command to view the current value.

```
[user@host ~]$ podman info --format {{.Host.EventLogger}}
journald
```

To monitor events in Podman, use the `podman events` command. By default, the `podman events` command follows new events as they occur continuously. You can use the `--stream=false` option to force the command to exit after reading the last known event. The following example prints the image pull and container create events.

```
[user@host ~]$ podman events --stream=false
2024-01-16 13:51:46.074958359 -0500 EST system refresh
2024-01-16 ... -0500 EST image pull 699...47d registry.access.redhat.com/rhscl/httpd-24-rhel7 (1)
2024-01-16 ... -0500 EST container create fb6...ce4 (image=registry.access.redhat.com/rhscl/httpd-24-rhel7:latest (2)
...output omitted...
```

(1) The image pull event.

(2) The container create event.

The `--filter` or `-f` option enables you to filter events by object and event type multiple times.

```
[user@host ~]$ podman events --filter event=create --filter type=container --stream=false
2024-01-16 ... -0500 EST container create fb6...ce4 (image=registry.access.redhat.com/rhscl/httpd-24-rhel7:latest
...output omitted...
```

You can use the `--since` and `--until` options to view past events. These options enable you to specify a time range for the events you are interested in. The `--since` and `--until` option values can be RFC3339Nano time stamps or a Go duration string such as `10m` or `5h`.

```
[user@host ~]$ podman events --since 5m --stream=false
2024-01-28 ... -0500 EST image pull bc1...011 registry.access.redhat.com/ubi8/ubi
2024-01-28 ... -0500 EST container remove f9f...63e (image=registry.access.redhat.com/ubi8/ubi:latest
```

REFERENCES

Namespaces and Nsenter

podman-events(1) man page