Software Engineering II: **(Docker/Podman) Compose**

Philipp Fruck

Podman Network Setup

How does rootless networking work?

- Podman 5.X: `pasta` as default network driver
  - Behaves like application is running on the host
  - Host IP, correct source IP, but still isolated
    - Port forwarding is required compared to host networking
- Previous default: `bridge` driver
  - Still default when connecting containers via shared network
  - NAT is used -> Source IP is always the NAT IP
  - IPv6: Interesting... Sometimes broken

Q: How can two pods access each other over the network?

- Pod: Share network namespace, reach each other via localhost
- Bridge: Connect containers to bridged network, both gain a NATed IP

Podman: Connecting Services

The following script spawn a Postgres client and server and connects them through a custom network

```
#!/bin/sh
set -x
net=test; client=pgclient; server=pgserver
pg_password='p4$$w0rd'; pg_version=18-alpine
# Create network and launch detached server
podman network create ${net}
podman run --rm --name ${server} --network ${net} -e POSTGRES_PASSWORD=${pg_password} \
  -d postgres:${pg_version}
# Wait some time to avoid race condition
sleep 3
# Run the client (note different password variable)
podman run --rm --name ${client} --network ${net} -e PGPASSWORD=${pg_password} \
  --entrypoint psql postgres:${pg_version} -h ${server} -U postgres -c "SELECT version();"
# clean up
podman rm -f ${client} ${server} && podman network rm -f ${net}
```

[finished]

Podman: Connecting Services

```
+ net=test
+ client=pgclient
+ server=pgserver
+ pg_password='p4$$w0rd'
+ pg_version=18-alpine
+ podman network create test
test
+ podman run --rm --name pgserver --network test -e 'POSTGRES_PASSWORD=p4$$w0rd' -d
postgres:18-alpine
18bc73b71cf1ce159240af477181a67e9e2d7d39c664b57ea6deb87798e10b4b
+ sleep 3
+ podman run --rm --name pgclient --network test -e 'PGPASSWORD=p4$$w0rd' --entrypoint psql
postgres:18-alpine -h pgserver -U postgres -c 'SELECT version();'
                                          version
-------------------------------------------------------------------------------------------
 PostgreSQL 18.2 on x86_64-pc-linux-musl, compiled by gcc (Alpine 15.2.0) 15.2.0, 64-bit
(1 row)

+ podman rm -f pgclient pgserver
+ podman network rm -f test
test
```

Podman: Connecting Services

Shell Script Approach

Do you like the manual setup?

- Easy setup of two Postgres instances
  - Can communicate over network
  - Includes DNS resolution -> Nice!

But:

- Verbose syntax
- Setup is error-prone
  - Lots of repeated custom logic
- What if one of the commands fails?

Goal

- Meta file containing setup instructions
  - More declarative than shell script
  - Better error handling
  - Cross platform
- Lifecycle handling?
  - Autostart?
  - Restart after crash?
  - Automatic updates??

Declarative Definition

We can use `docker-compose` and Quadlets for declarative definition

This can be seen as a subset of Infrastructure as Code

	docker-compose	Quadlet
Requirements	Docker/Podman + Compose Script	Podman + systemd
Lifecycle Manager	Docker Daemon	systemd
Recommended for	Development (project-level)	Servers (host-level)
File format	Custom (compose)	systemd / Kubernetes

Docker vs Podman Compose

There are two major implementation of the **compose spec** (https://compose-spec.io)

Docker Compose

- Reference implementation (Docker 1st party)
  - Always supports full feature set
- Utilizes Docker socket ("REST" API)
- Allows parallel operation (image pulls etc.)

- v1: Standalone Script `docker-compose`
- v2: Plugin for Docker `docker compose`

Podman Compose

- Community implementation for Podman
- Translates compose file into `podman` CLI commands
- Sequential operation (last time I checked)

- Standalone Script `podman-compose`

Docker Compose + Podman

We want to use Docker Compose with Podman

- v2 can still be used as standalone script
  - Many distros only ship v1.X though...
  - Ensure your distro ships `docker-compose` v2.X or install directly from GitHub Releases
    - `docker-compose --version`
- We need to enable the Podman Socket for Docker API support
  - `systemctl --user enable --now podman.socket`
- We need to tell `docker-compose` the path of our API socket!
  - `export DOCKER_HOST=unix://${XDG_RUNTIME_DIR}/podman/podman.sock`
  - Add this to your `~/.bashrc` or similar

All slides in this lecture assume this `docker-compose` + `podman` setup. If you use another setup, you need to replace the `docker-compose` commands (officially legacy) with `docker compose` or `podman-compose`.

Compose: Simple Example

Compose Spec

- Simple YAML file (Minimal Syntax)
- All container under `services`
  - `web`: Implicit container name
- Declarative definition of CLI parameters

```yaml
services:
  web:
    image: docker.io/nginxinc/nginx-unprivileged
    ports:
      - 8080:8080/tcp
    volumes: # serve generated lecture slides
      - ../../_site:/usr/share/nginx/html:ro,Z
```

- Launching: `docker-compose -f examples/compose/compose.simple.yml up`
  - Relative paths are read from compose file
  - `compose.yml` and `docker-compose.yml` (and `.yaml`) are automatically detected --> no `-f`

Compose: Subcommands

Command	Action
`config`	Parse and show the final config
`pull`	Pull service images
`build`	Build or rebuild services if required
`create`	Creates the containers & networks
`start`	Start services (must be created first)
`restart`	Restart services
`up`	Shortcut for create and `start`
`stop`	Shutdown all services (keep containers)
`down`	Stop and remove containers & networks
`logs`	View output from containers
`exec`	Execute command in running container
`run`	Execute command in a new container
`watch`	Watch service, refresh when files change

Subcommands can be executed for all containers (default) or for a specific container

- `docker-compose [up/down/logs] myservice`
  - Use `up -d` to detach the logs
- Use `logs -f` to stream updated logs
- Use `up --build` for a `build` shortcut
- Use `run --rm` to prevent zombies

Compose: More Syntax

```yaml
services:
  app:
    build:
      context: .
      dockerfile: Containerfile
    container_name: demo-app
    command: ["--http", ":8080"]
    entrypoint: ["/bin/myapp"]
    user: "1000:1000"
    working_dir: /app
    env_file: .env
    environment:
      LECTURE: "SE2"
    ports:
      - 127.0.0.1:8080:8080/tcp
```

We can specify any container option in the compose file

- Build the service `app` from local `Containerfile` in directory `.`
- When using `Dockerfile` in same folder, `build .` is sufficient
  - Remember: No automatic rebuilds!

▒▒ Naming

- COMPOSE_PROJECT_NAME: Parent folder of the compose file, e.g. `compose`
  - Can be overridden (environment or top-level `name:`)
- `container_name` sets explicit name `demo-app` instead of `compose-app-1`

Compose: Variables

```yaml
services:
  app:
    image: alpine:${TAG:-latest}
    env_file: .env
    environment:
      OTHERPASSWORD: ${MYPASSWORD}
    command:
      - sh
      - -c
      - env | grep PASSWORD
```

Variables

- Compose reads current environment and parses `.env` file relative to compose file
- Variable must be passed to containers explicitly

Substitution

Syntax	Evaluates To
`${VAR}`	Value of `VAR`
`${VAR:-default}`	Value of `VAR`, otherwise `default`
`${VAR:?error}`	Require VAR or throw error

Compose: Variables

```yaml
services:
  app:
    image: alpine:${TAG:-latest}
    env_file: .env
    environment:
      OTHERPASSWORD: ${MYPASSWORD}
    command:
      - sh
      - -c
      - env | grep PASSWORD
```

```
# ../examples/compose/.env
MYPASSWORD=s3cr3t
```

```
file=../examples/compose/compose.env.yml
docker-compose -f $file config | grep -A2
environment
```

———————————— [finished] ————————————

```
    environment:
      MYPASSWORD: s3cr3t
      OTHERPASSWORD: s3cr3t
```

Compose: Network

```yaml
services:
  proxy:
    image:
nginxinc/nginx-unprivileged:alpine
    networks: ["web"]
  backend:
    image: python:alpine
    networks: ["web", "db"]
  db:
    image: postgres:alpine
    networks: ["db"]
networks:
  db:
  web:
    enable_ipv6: true
    ipam:
      config:
        - subnet: 2001:db8::/64
```

Config

- Compose creates a default project network
  - Each container has implicit default network
- Custom networks can be created under `networks:`
  - Simple name is sufficient
  - Advanced options like subnet and network driver can be specified
- Bridged networks support DNS for container names
  - E.g. `backend` can reach `db` via `db` hostname
  - Ensure Aardvark DNS plugin is installed!

Exercise

Remember the script from the beginning? Your task is to reproduce this using a compose file!

The script is located in the `git` repo under `../examples/compose/manual.sh`

- Spawn two Postgres containers
  - `client` and `server`
  - `server` should keep running (daemon)
  - `client` should only execute the query and exit

Use a `.env` file like this to share variables between the two containers:

```
POSTGRES_VERSION=18-alpine
POSTGRES_PASSWORD=s3cr3t
```

Compose: Advanced Network

```yaml
services:
  proxy_host:
    image: nginxinc/nginx-unprivileged:alpine
    network_mode: host
  proxy_old: # Workaround for Podman 4.X
    image: nginxinc/nginx-unprivileged:alpine
    network_mode: "
slirp4netns:port_handler=slirp4netns"
  proxy_new: # Workaround for Podman 5.X
    image: nginxinc/nginx-unprivileged:alpine
    network_mode: "pasta"
```

As mentioned, source IP handling can be difficult with rootless networking

- Other network modes than `bridge` can be used
  - No support for DNS resolution!
- To get source IPs with bridge network:
  - Look into **socket activation** (https://systemd.io/DAEMON_SOCKET_ACTIVATION)

Compose: Sidecars

In K8s (or Pods in general) there is the "Sidecar" pattern:

- Shared network namespace between containers
- Can reach each other via localhost
- Problem: No native Docker support for Pods
  - We can still share the network
  - Ports must be exposed from main service
  - Funny issues when starting/deleting containers

```yaml
services:
  app:
    image: nginx:alpine
    ports:
      - 8001:80   # app port
      - 8080:8080 # sidecar port
  sidecar:
    image: nginxinc/nginx-unprivileged
    entrypoint:
      - sh
      - -c
      - sleep 1 && nginx -g "daemon off;"
    network_mode: "service:app"
```

Compose: Healthcheck & Dependencies

```yaml
services:
  app:
    image: nginx:alpine
    healthcheck:
      test: ["CMD", "curl","[::1]"]
      interval: 3s
      retries: 5
      start_period: 1s
      timeout: 10s
  dependend:
    image: nginx:alpine
    depends_on:
      app:
        condition: service_healthy
```

- We can define a startup order between services
  - `depends_on` with multiple conditions
  - `service_running`: Wait for service to start
  - `service_healthy`: Wait for health check
- Healthchecks...
  - can be specified on image level
  - can be overridden by compose
  - run inside the container
  - use `CMD` or `CMD-SHELL`
    - array or string notation

Compose: Watch

```yaml
services:
  app:
    build: ./watch
    entrypoint: ["sleep", "9000"]
    develop:
      watch:
        - action: rebuild
          path: ./watch/requirements.txt

```

Auto update builds

- For development, we can watch and reload project
- Can be done for volume mounts (hot-reload)
  - `action: sync`
- Also for full container builds
  - `action: build`
- Use `docker-compose up -w` to watch

Compose: Anchors

```yaml
x-common-env: &common-env
  TZ: UTC
  LOG_LEVEL: info

services:
  web:
    environment:
      <<: *common-env
      HTTP_PORT: 8080
    volumes: &app-volumes
      ./data:/data:ro,z
  worker:
    environment: *common-env
    volumes: *app-volumes
```

- YAML anchors can be used to reuse config
- Can be placed anywhere in the file
  - Use top-level `x-` attributes for config snippets (no invalid syntax errors)
- Advantage: Ensure consistent config
- Disadvantage: Harder to read

Compose: Profiles

```yaml
services:
  app:
    image: python:alpine
    profiles: ["app", "web"]
  web:
    image: nginx:alpine
    profiles: ["web"]
  db:
    image: postgres:18-alpine
    profiles: ["app", "db"]
```

Separating larger applications

- Profiles can be used to divide complex setups
- Each service can be added to multiple profiles
- Use `docker-compose --profile db up` to spawn db
  - Use multiple --profile flags for `web` + `app`
  - Or: Use `COMPOSE_PROFILES=web,app` environment

Compose: File based separation

Include

- The `include` directive can be used to import content from other compose files
  - Simple, declarative way if files get too large

```yaml
include:
  - ./compose.simple.yml
services:
  db:
    image: postgres:alpine
```

Multiple files via CLI

- We can also specify multiple files via CLI
- Useful if we use tooling around compose
  - E.g. conditionally include files
- `docker-compose -f file1 -f file2`
- Each file overrides contents of previous files
  - Objects (lists, dicts) get merged

```
file=../examples/compose/compose.include.yml
docker-compose -f $file config --services
```

```
──────────────────── [finished] ────────────────────
```

```
db
web
```

Thank you for your attention!

Don't forget the feedback