al nafi
EDUCATION BENEFITS ALL

Al Razzaq Program Part II

Notes by Al Nafi

Red Hat OpenShift Development I: Introduction to Containers with Podman

Author:

Osama Anwer Qazi

Introduction and overview of containers

Containers represent a lightweight, portable means of packaging applications and their dependencies into a single executable unit. They provide a consistent runtime environment across development, testing, and production stages, ensuring that applications behave the same regardless of where they run.

- **Key characteristics of containers:**

  1. **Isolation:** Each container encapsulates an application along with its libraries, binaries, and configuration files, preventing interference with other containers on the same host.
  2. **Portability:** By bundling all dependencies, containers can run on any system with a compatible container runtime (e.g., Podman, Docker, CRI-O).
  3. **Efficiency:** Containers share the host kernel rather than requiring a full guest operating system, which makes them more resource-efficient than traditional virtual machines.
  4. **Reproducibility:** A container image guarantees that the same application version, with the same dependencies, is deployed consistently across environments.

- **How containers facilitate application development:**

  1. **Consistent Dev-to-Prod Workflow:** Developers build and test against the same container image that will run in production, reducing “it works on my machine” issues.
  2. **Rapid Iteration:** Containers can be started and stopped in seconds, allowing developers to spin up isolated environments for feature development or troubleshooting without waiting for VM provisioning.
  3. **Modularization:** By breaking a monolithic application into multiple containers (each handling a specific function), teams can develop, test, and deploy components independently.
  4. **Dependency Management:** All required libraries, runtime versions, and configurations are defined in the image, eliminating version drift and simplifying onboarding of new team members.

5. **Scalability:** Containers can be replicated horizontally to handle increased load, and orchestration platforms (e.g., Kubernetes, OpenShift) manage scaling automatically.

- **Continuity with preceding topics:**
  In prior sections (e.g., "Introduction to Linux namespaces and cgroups"), you learned how Linux primitives enable resource isolation. Containers build upon these primitives to provide a higher-level abstraction. Understanding the kernel features that underlie containers is essential before diving into container tools like Podman.

- **Foundation for upcoming topics:**
  These container fundamentals set the stage for Podman basics, image management, and orchestration. Once you grasp the "why" and "what" of containers, you can focus on practical tasks such as building, running, and orchestrating them with Podman, Compose, and Kubernetes/OpenShift.

Podman basics
Podman (Pod Manager) is a daemonless container engine compatible with the Open Container Initiative (OCI) specifications. Unlike Docker, Podman does not require a long-running background service; each command interacts directly with the container runtime.

- **Daemonless architecture:**
  - Podman launches containers and manages them without a central daemon. Commands execute as regular processes, improving security and simplifying integration in CI/CD pipelines.
  - Rootless mode: Developers can run Podman entirely as a non-root user, reducing attack surface and avoiding privileged daemons.

- **Core operations in Podman:**
  - **View available containers and images:**
    - `podman ps` (lists running containers)
    - `podman ps -a` (lists all containers, including stopped)
    - `podman images` (lists local images)
  - **Run a container:**

        - When you execute `podman run [options] <image>`, Podman pulls the image (if not already present), creates a container, and starts it.
        - Options include:
            - `-d` (detached/background),
            - `-p` (port mappings),
            - `--name` (assign a name),
            - `-v` (mount volumes).
    - **Manage container lifecycle:**
        - **Start/Stop/Restart:** `podman start <container>`, `podman stop <container>`, `podman restart <container>`.
        - **Inspect:** `podman inspect <container>` provides JSON metadata (network settings, mounts, environment variables).
        - **Exec:** `podman exec -it <container> /bin/bash` to get an interactive shell.
        - **Remove:** `podman rm <container> and podman rmi <image>` remove containers and images, respectively.
- **Pods in Podman:**
    - A **pod** is a group of one or more containers that share networking and can share storage. This concept mirrors Kubernetes pods.
    - Use `podman pod create` to define a pod, then run containers within it (e.g., an application container and its helper sidecar).
- **Advantages over other engines:**
    - **Rootless execution:** Enhances security by running without root privileges.
    - **Docker CLI compatibility:** Many Docker commands work unmodified with Podman (aliasing `docker` to `podman`).
    - **Kubernetes YAML support:** You can generate pod definitions from running containers (`podman generate kube <container>`).

- **Building on prior knowledge:**
  Having learned about container runtimes and OCI images, Podman serves as a tool to interface with those constructs. You don't need a separate daemon, which aligns with the DevOps principle of lightweight, composable tooling.

Container images
Container images are read-only templates comprising a filesystem snapshot, application binaries, configuration files, runtime libraries, and metadata (labels, environment variables, entrypoints). They serve as the blueprint for containers.

- **Image structure and layers:**
  - **Layers:** Each instruction in a containerfile (formerly Dockerfile) creates a new layer. Layers are stacked in a union filesystem, allowing for efficient reuse and caching.
    - **Base layer:** Often a minimal OS (e.g., Fedora, Ubuntu, Alpine).
    - **Intermediate layers:** Result from commands like `RUN apt-get install` or `COPY ./app /app`.
    - **Final layer:** Sets metadata—entrypoint, default command, environment variables.
  - **Copy-on-write:** When a container modifies a file, a new layer is created for that change rather than overwriting the image layers. Multiple containers can share layers without duplicating storage.
- **Container registries:**
  - **Public registries:**
    - **Docker Hub (hub.docker.com):** Largest public repository. Contains official, community, and certified images.
    - **Quay.io:** Popular among Red Hat and OpenShift users; offers automated builds and security scans.
    - **Red Hat Container Catalog:** Provides certified images optimized for OpenShift and supported by Red Hat.
  - **Private registries:**

        - **Self-hosted:** e.g., Harbor or a private instance of Docker Registry.
        - **Cloud-hosted:** Amazon ECR, Google Container Registry (GCR), Azure Container Registry (ACR).
    - **Authentication and access control:**
        - Registries may require credentials (username/password, service accounts, or token-based).
        - Podman can be configured to store credentials in `~/.docker/config.json` or via CLI flags.
- **Searching and pulling images:**
    - **Search:** `podman search <term>` queries configured registries. Filters can limit by stars, size, or whether it's an official image.
    - **Pull:** `podman pull registry.example.com/namespace/image:tag` downloads the specified image. If no tag is provided, Podman defaults to `:latest`.
- **Inspecting image metadata:**
    - `podman inspect <image>` outputs JSON with:
        - **Created date, size, architecture.**
        - **Labels:** Key/value metadata set in the containerfile (e.g., version, maintainer).
        - **Entrypoint and command:** Defines how the container starts.
- **Managing image versions (tags):**
    - Use **semantic versioning:** e.g., `v1.0.0`, `v1.0`, `latest`.
    - Whenever you update an application, build and push a new image with an incremented tag.
    - Avoid using `:latest` in production deployments to prevent unintended upgrades.
- **Continuity with custom images:**
  Understanding how to navigate and manage existing images is a prerequisite for

building custom images tailored to your applications (covered next).

Custom container images
Custom images enable you to encapsulate your application's runtime environment, dependencies, and configuration. Building images ensures reproducibility and forms the basis for deployment pipelines.

- **Containerfile fundamentals (theory only):**
  - **FROM instruction:** Specifies the base image (e.g., `FROM registry.access.redhat.com/ubi8/ubi:8.7`).
  - **RUN instruction:** Executes commands during build—typically package installation or environment setup.
  - **COPY / ADD instructions:** Copy files from the build context (local directory) into the image's filesystem.
  - **ENV instruction:** Defines environment variables (e.g., `ENV NODE_ENV=production`).
  - **WORKDIR instruction:** Sets the working directory for subsequent instructions.
  - **ENTRYPOINT / CMD instructions:** Define the default process that runs when a container starts.
  - **LABEL instruction:** Adds metadata—often used for automated scanning or documentation.
- **Build context and caching:**
  - **Context:** The directory (and subdirectories) sent to the container runtime when building. Only files within this directory can be referenced in a COPY/ADD.
  - **Layer caching:** Podman reuses unchanged layers from previous builds. If a RUN instruction's inputs haven't changed, that layer is fetched from cache, accelerating rebuilds.
- **Best practices for custom images:**
  - **Minimize layers:** Combine related commands into a single RUN to reduce the number of layers.

- **Order instructions for effective caching:** Place instructions least likely to change (e.g., installing OS packages) before copying application code, so code changes don't invalidate prior layers.
- **Use slim base images:** Start from minimal distributions (e.g., UBI Minimized or Alpine) to reduce image size and attack surface.
- **Avoid storing secrets:** Do not bake credentials or private keys into the image. Use environment variables or secret-injection mechanisms at runtime.
- **Include health checks and metadata:** Labels such as `org.opencontainers.image.revision` and `LABEL maintainer="team@example.com"` assist with traceability.

- **Examples of layering strategy (theory):**
  - **Layer 1 (Base):** `FROM registry.redhat.io/ubi8/ubi:latest`
  - **Layer 2 (System packages):** `RUN yum install -y python3 gcc && yum clean all`
  - **Layer 3 (Application dependencies):** `COPY requirements.txt /app/` → `RUN pip3 install -r /app/requirements.txt`
  - **Layer 4 (Application code):** `COPY . /app/`
  - **Layer 5 (Metadata & start command):** `WORKDIR /app` → `CMD ["python3", "app.py"]`
- **Continuous integration considerations:**
  - Automate image builds with CI pipelines (e.g., Jenkins, GitLab CI, GitHub Actions).
  - Configure the pipeline to automatically tag images based on Git commits or semantic version files.
  - Integrate security scanners (e.g., Clair, Trivy) to identify vulnerabilities in base layers and application dependencies before pushing to a registry.
- **Dependencies and upcoming topics:**
  Once a custom image is built, it may need persistent storage (covered in "Persisting data") or be integrated into multi-container setups (covered in "Multi-container

applications with compose”).

Persisting data
By default, container filesystems are ephemeral: when a container is removed, its data is lost. For stateful services (e.g., databases), you must configure persistent storage so that data survives container restarts or recreations.

- **Types of persistence:**
  - **Bind mounts:** Map a directory on the host to a directory inside the container (e.g., `-v /host/db:/var/lib/mysql`). Changes in the container immediately reflect on the host.
  - **Named volumes:** Managed by the container engine (e.g., `podman volume create dbdata` then `-v dbdata:/var/lib/postgresql/data`). Volumes live under a host-managed directory (often `/var/lib/containers/storage/volumes`).
  - **Storage classes in Kubernetes/OpenShift:** Abstract underlying storage (e.g., AWS EBS, OpenShift’s persistent volume claims) so that pods can claim persistent volumes.
- **Running database containers with persistence:**
  - **Example (MySQL):**
    - Create a volume: `podman volume create mysql-data`

Run container:

```
podman run -d \
 --name my-mysql \
 -e MYSQL_ROOT_PASSWORD=mysecret \
 -v mysql-data:/var/lib/mysql \
 registry.redhat.io/rhel8/mysql-80:latest
```

    - Data written to `/var/lib/mysql` inside the container persists in the `mysql-data` volume, surviving container removal.
  - **Example (PostgreSQL):**

Bind mount host directory:

```
 mkdir -p /opt/pgdata
podman run -d \
  --name my-postgres \
  -e POSTGRES_PASSWORD=mysecret \
  -v /opt/pgdata:/var/lib/postgresql/data \
  registry.redhat.io/rhel8/postgresql-13:latest
```

- ■ Host directory `/opt/pgdata` retains data beyond container lifecycle.

- **Consistency and backup strategies:**
  - **Database dumps:** Schedule regular logical backups (e.g., `mysqldump`, `pg_dump`) to a mounted volume or external storage.
  - **Snapshot-based backups:** If using a cloud provider, create volume snapshots at intervals to safeguard against data corruption.
  - **Replication:** Consider setting up a multi-node database cluster (e.g., MySQL master-slave) so that if one container fails, another can serve data.
- **Implications for scaling:**
  - When scaling database containers across multiple hosts, ensure that storage is accessible to all nodes (e.g., via NFS, Ceph, GlusterFS, or a cloud storage service).
  - In Kubernetes/OpenShift, use shared persistent volume claims or a distributed file system (e.g., OpenShift Container Storage) to allow multiple pods to access the same volumes if necessary.
- **Security considerations:**
  - **File permissions:** Ensure the host directories or volumes are owned by the database user inside the container (e.g., `chown 27:27 /opt/pgdata` for PostgreSQL's UID/GID).
  - **Encrypt data at rest:** Use filesystem encryption (e.g., LUKS) or rely on cloud-provided encrypted volumes.
  - **Network policies:** For Kubernetes/OpenShift, restrict database access to only the application pods via network policies or OpenShift SecurityContextConstraints.

- **Dependencies on multi-container setups:**
 Persistent storage is essential for stateful services in multi-container stacks (e.g., an application container connecting to a database container). When orchestrating multiple containers (covered later), define volumes at the compose or deployment level.

Troubleshooting containers
 Even though containers promote consistency, issues can still arise—misconfigurations, crashes, dependency mismatches, or runtime errors. Systematic troubleshooting involves examining logs, inspecting container state, and configuring remote debugging when necessary.

- **Analyzing container logs:**
  - **Podman logs:**
    - `podman logs <container>` streams stdout and stderr.
    - Options:
      - `--tail <n>` to view only the last n lines.
      - `-f` to follow logs in real time.
  - **Log persistence options:**
    - By default, Podman uses a JSON-file logger for containers. These files reside under `/var/lib/containers/storage/overlay-containers/<id>/userdata/oci-logs/`.
    - For production, redirect logs to centralized logging systems (e.g., Fluentd, Elasticsearch-Logstash-Kibana (ELK) stack) by configuring a logging driver or using `podman logs` in scripts.
- **Inspecting container state and resources:**
  - **podman inspect <container/image>:** Provides detailed JSON data about configurations, network settings, mounted volumes, environment variables, and resource constraints.
  - **Resource usage:** Use `podman stats` to monitor CPU, memory, and I/O usage for running containers. High memory or CPU usage may indicate resource exhaustion or runaway processes.

- **Common issues and theoretical resolutions:**
  - **Container fails to start:**
    - Check entrypoint or CMD syntax.
    - Inspect environment variables for missing or incorrect values.
    - Verify that required volumes are mounted and accessible.
  - **Application errors inside container:**
    - Use `podman exec -it <container> /bin/bash` to inspect file paths, configuration files, and logs within the container.
    - Confirm that necessary dependencies are present in the built image.
  - **Networking problems:**
    - Confirm that the container's ports are published (`-p host_port:container_port`).
    - Inspect network mode (`podman network ls` and `podman network inspect`).
    - In Podman rootless mode, additional firewall rules (e.g., rootless networking via slirp4netns) might need adjustment.
- **Configuring a remote debugger (theory):**
  To debug an application inside a container without rebuilding an image each time:
  - **Expose debugging port:**
    - Build your container with the debugger (e.g., `debugpy` for Python, `gdbserver` for C/C++).
    - Ensure the container's debugging port is published (e.g., `-p 5678:5678`).
  - **Mount source code:**
    - Instead of bundling source code in the image, mount the local source directory into the container (e.g., `-v $(pwd)/src:/app/src`). This allows stepping through code changes without rebuilding the image.

- **Attach debugger from host IDE:**
    - In your IDE (e.g., VS Code, PyCharm), configure a remote debugging session pointing to the container's IP (often `localhost` with mapped ports).
    - Set breakpoints and step through code as it runs in the container.

- **Integration with CI/CD and monitoring:**
    - Automate log aggregation by shipping container logs to a logging backend. In Kubernetes/OpenShift, use Fluent Bit or Drools to centralize logs.
    - Configure health-check probes (e.g., HTTP or command probes) so orchestration platforms can automatically restart or replace unhealthy containers.

- **Relation to future orchestration topics:**
  In multi-container or orchestrated environments (e.g., Kubernetes/OpenShift), centralized logging, monitoring, and debugging become more critical. Understanding how to troubleshoot individual containers lays the groundwork for diagnosing complex system-wide issues.

Multi-container applications with compose
Compose (originally Docker Compose, compatible with Podman Compose) provides a declarative way to define and manage multi-container applications by specifying services, networks, and volumes in a single YAML file (typically `docker-compose.yml` or `podman-compose.yml`).

- **Conceptual overview of Compose:**
    - **Services:** Represent each containerized component (e.g., web server, database, cache).
    - **Networks:** Define how services communicate—by default, Compose creates a dedicated network where service names act as hostnames.
    - **Volumes:** Declare persistent storage to be shared among services (e.g., a shared data directory or database volume).

- **Benefits of using Compose (theory):**

- **Single source of truth:** All container definitions, environment variables, port mappings, and volumes are described in one file.
- **Simplified lifecycle management:** Execute `podman-compose up -d` to start all services; `podman-compose down` tears them down, removing networks and optionally volumes.
- **Consistent environment for development and testing:** Teams can share the same compose file, ensuring identical service interconnections.
- **Isolation of environments:** Compose allows multiple distinct stacks to run on the same host without interfering (e.g., staging, testing, production).

- **Core Compose file elements (theory):**
  - **version:** Specifies the Compose file syntax version (e.g., `"3.8"`).
  - **services:**
    - **image / build:** Either pull an existing image or build from a Dockerfile.
    - **ports:** Host-to-container port mappings (e.g., `"8080:80"`).
    - **volumes:** Mount named volumes or host directories.
    - **environment:** Environment variables (e.g., `- “DB_PASSWORD=secret”`).
    - **depends_on:** Service dependency order (e.g., web depends_on db).
    - **networks:** Specify networks services attach to (default is a bridge network).
  - **volumes:** Define named volumes and driver options (e.g., `dbdata: { driver: local }`).
  - **networks:** Define custom networks (e.g., overlay networks for multi-host communication).
- **Example architecture (conceptual):**
  - **Service: webapp**
    - Build from custom image.

    - Expose port 80.
    - Depends on db.
  - **Service: db**
    - Use official PostgreSQL image.
    - Mount volume `dbdata` at `/var/lib/postgresql/data`.
    - Expose port 5432 (only internal).
  - **Service: redis**
    - Use official Redis image.
    - No persistent volume (appropriate for cache).
- **Networking in Compose:**
  - By default, Compose creates a single network named `<project>_default`.
  - Services can reach each other using service names as DNS: e.g., webapp connects to `postgres:5432`.
  - You can declare additional networks to segregate traffic (e.g., a front-end network and a back-end network).
- **Scaling and dependencies:**
  - Use `podman-compose up --scale webapp=3` to run multiple instances of the web service behind a shared network. Persistence for sessions can be managed via external cache (e.g., Redis).
  - The `depends_on` directive guarantees startup order but does not wait for a service to be "ready"—for databases, incorporate health-check logic or retry mechanisms in the application.
- **Transition to orchestration platforms:**
  While Compose is suitable for local development and small deployments, production environments typically require orchestration platforms (e.g., Kubernetes/OpenShift). In fact, Compose files can be translated into Kubernetes manifests (e.g., with tools like `kompose`)—providing a bridge to more sophisticated orchestration.

Container orchestration with Kubernetes and OpenShift
Orchestration platforms manage multiple containers across a cluster of hosts, handling deployment, scaling, networking, storage, and self-healing. Kubernetes (upstream) and OpenShift (Red Hat's enterprise distribution of Kubernetes) provide declarative APIs to automate container lifecycle at scale.

- **Key orchestration concepts:**
  - **Cluster:** A set of worker nodes (where containers run) and control-plane nodes (master components that schedule workloads and maintain cluster state).
  - **Pod:** The smallest deployable unit—a group of one or more containers sharing the same network namespace and storage volumes.
  - **Deployment:** Manages stateless workloads—ensures a specified number of pod replicas are running, supports rolling updates and rollbacks.
  - **StatefulSet:** Manages stateful workloads (e.g., databases) that require stable network identities and persistent storage.
  - **Service:** Defined as an abstraction that exposes a set of pods (selected by labels) via a stable IP and DNS name. Types include ClusterIP, NodePort, and LoadBalancer.
  - **Ingress / Route (OpenShift):** Provides external HTTP/HTTPS access to services, often with hostname-based routing and TLS termination.
  - **ConfigMap and Secret:** Store configuration data and sensitive information (e.g., API keys) separately from container images, injected into pods as environment variables or mounted files.
  - **PersistentVolume (PV) and PersistentVolumeClaim (PVC):** Abstract storage resources that pods can consume, decoupling storage provisioning from pod definitions.
- **Differences between Kubernetes and OpenShift:**
  - **Authentication and authorization:**
    - OpenShift integrates with Red Hat's SSO (Keycloak) and provides Role-Based Access Control (RBAC) out of the box.
    - Kubernetes requires additional configuration (e.g., integrating with an identity provider).

- **Security context constraints (SCC):**
  - OpenShift enforces stricter defaults (restricting privileged containers unless explicitly permitted).
  - Kubernetes uses PodSecurityPolicies (deprecated in newer versions) or Pod Security Admission to enforce policies.
- **Developer experience:**
  - OpenShift provides a Web Console for visual management of projects, builds, and deployments.
  - Kubernetes typically relies on `kubectl` or third-party dashboards.
- **Built-in CI/CD (OpenShift Pipelines):**
  - OpenShift includes Tekton-based pipelines, enabling a seamless CI/CD experience.
  - Kubernetes users often deploy separate pipeline tools (e.g., Jenkins, Argo CD).

- **Orchestrating containerized applications (theory):**
  - **Define a Deployment (or BuildConfig in OpenShift):**
    - Specify the container image, number of replicas, update strategy (rolling vs. recreate), and resource limits.
    - In OpenShift, a BuildConfig can automatically build images from source (Git), stream logs, and push images to the integrated registry.
  - **Expose the Deployment via a Service:**
    - Create a Service manifest (ClusterIP for internal, LoadBalancer or NodePort for external).
    - In OpenShift, create a Route to map an external hostname to the service.
  - **Configure storage for stateful workloads:**
    - Define a PVC that requests storage from a StorageClass (e.g., AWS EBS, GCE Persistent Disk, or OpenShift Container Storage).

        - For databases, use a StatefulSet to ensure stable network identifiers (`<pod-name>-0`, `<pod-name>-1`, etc.).
    - **Scale and update:**
        - Update the Deployment manifest with a new image tag or configuration change, then apply (`kubectl apply -f deployment.yaml`).
        - Kubernetes/OpenShift performs a rolling update—creating new pods, shifting traffic, and removing old pods gradually.
        - Horizontal Pod Autoscaler (HPA) can automatically adjust replica counts based on CPU/memory metrics or custom application metrics.
- **Networking and service discovery:**
    - **Kubernetes CNI plugins:** Calico, Flannel, or OpenShift's OVN-Kubernetes provide L3 networking between pods.
    - **DNS:** CoreDNS (in Kubernetes) and OpenShift's internal DNS resolve service names to cluster IPs, enabling seamless inter-pod communication.
- **Security and compliance in orchestration:**
    - **Least-privilege containers:** Use minimal security contexts, run as non-root where possible, and avoid privileged containers.
    - **Network policies (Kubernetes) / OpenShift NetworkPolicy:** Restrict traffic between pods—e.g., only the frontend can access the backend.
    - **Image certification:** In OpenShift, use Red Hat-certified images or scan images in the integrated registry with Clair.
    - **Audit logging:** Enable audit logs on the API server to track changes and detect unauthorized attempts.
- **Continuity with earlier topics:**
  The theory of orchestration builds upon:
    - Container fundamentals (namespaces, cgroups)
    - Podman usage (building and running individual containers)
    - Compose (managing multiple containers locally)
    - Persistence (defining volumes shared among pods)

- ○ Troubleshooting (monitoring logs and health within a large-scale cluster)

Conclusion

These theoretical notes cover the lifecycle of containerized application development—from understanding container fundamentals to orchestrating at scale—and lay the groundwork for infusing AI into every stage. By mastering these concepts, you'll be equipped to implement secure, scalable, and intelligent containerized platforms aligned with the Al Razzaq Part 2 series, and be prepared for upcoming hands-on exercises and advanced topics.