Containers

CSC 510

Richard Kelley

Why Containers Exist

- **Local installs create fragile environments**
  - Software often depends on specific OS versions, libraries, runtimes, and system tools
  - Small differences between machines lead to configuration drift and setup failures
  - Onboarding new developers or students becomes slow and error-prone
- **Virtual machines solve isolation, but at high cost**
  - VMs bundle an entire guest OS, increasing startup time and resource usage
  - Running many VMs in parallel is expensive and operationally heavy
  - VM images are large and slow to distribute
- **Containers provide lightweight, process-level isolation**
  - Containers package applications with their dependencies, but share the host kernel
  - Startup is fast, resource overhead is low, and scaling is easier
  - The same container can run consistently across laptops, servers, and cloud systems

Containers in Modern Development

- **Containers as a standard development artifact**
  - Applications are defined declaratively and versioned alongside source code
  - Developers build and run the same container locally that will run in production
  - Reduces the gap between development, testing, and deployment environments
- **Role in modern cloud workflows**
  - Containers are the unit of deployment in most cloud-native systems
  - They integrate naturally with CI/CD pipelines and automated testing
  - Cloud platforms assume containerized workloads by default
- **Connection to DevOps and reproducibility**
  - Infrastructure and environment setup become code, not manual steps
  - Builds are deterministic and repeatable across time and teams
  - Tools like **Docker** make environments portable, auditable, and easier to reason about

Containers, Virtual Machines, and Images

- **Containers vs virtual machines**
  - Virtual machines virtualize *hardware* and include a full guest operating system
  - Containers virtualize *processes* and share the host operating system kernel
  - Containers are lighter-weight, start faster, and use fewer resources
  - VMs provide stronger isolation boundaries; containers prioritize efficiency and portability
- **Abstraction level trade-offs**
  - VMs are closer to physical machines and behave like independent computers
  - Containers are closer to applications and behave like isolated processes
  - Containers assume a compatible host OS, which simplifies deployment but constrains portability
- **Images vs containers**
  - An *image* is a static, immutable definition of an environment and application
  - A *container* is a running instance created from an image
  - Multiple containers can be launched from the same image simultaneously

The Container Lifecycle

- **Build**
  - An image is built from a declarative specification (typically a Dockerfile)
  - Each build step produces a layered, cached filesystem snapshot
  - Images are versioned and can be stored or shared via registries
- **Run**
  - Running an image creates a container with its own process space and filesystem view
  - Configuration such as ports, environment variables, and volumes is applied at runtime
  - Containers can run interactively or in the background
- **Stop and remove**
  - Stopping a container halts the running process but preserves its state
  - Removing a container deletes the runtime instance, not the underlying image
  - Images persist independently and can be reused to create new containers
- **Mental model**
  - Images are reusable artifacts; containers are disposable execution units
  - This separation enables repeatable deployments and easy cleanup
  - Tools like [Docker] formalize this lifecycle into a simple, consistent workflow

Components You Interact With

- **Docker Engine**
  - The core runtime responsible for building images and running containers
  - Manages container lifecycle, networking, and storage on the host system
  - Runs as a background service on machines that support containers
- **Docker CLI**
  - The primary user interface for interacting with Docker
  - Commands like build, run, ps, and stop translate user intent into API calls
  - The CLI itself does not run containers; it sends requests to the Docker Engine
- **Separation of concerns**
  - The CLI can run locally or remotely
  - The Engine performs all privileged operations
  - This separation enables automation, scripting, and remote management

Docker Desktop and the Client–Daemon Model

- **Docker Desktop on macOS and Windows**
  - Provides a complete Docker environment on systems without native container support
  - Runs a lightweight Linux virtual machine behind the scenes
  - Bundles Docker Engine, CLI, networking, and filesystem integration
  - Handles OS-specific details so users can focus on containers
- **Client–daemon model**
  - The Docker CLI acts as a client
  - The Docker Engine runs as a long-lived daemon process
  - Communication occurs over a local or remote API
- **Why this model matters**
  - Multiple tools can talk to the same Docker Engine
  - Permissions and security are centralized in the daemon
  - This architecture underpins remote Docker usage and CI systems
- **Big picture**
  - **Docker** is a platform, not just a command-line tool
  - Desktop environments simplify onboarding
  - The underlying architecture scales from a laptop to cloud servers

What an Image Is

- **What a Docker image is**
  - A Docker image is an immutable, read-only template used to create containers
  - It defines the filesystem contents, runtime environment, and default behavior
  - Images are not running programs; they are packaged environments
- **What an image contains**
  - An operating-system userland (but not a kernel)
  - Application code, libraries, language runtimes, and system tools
  - Metadata describing how a container should start
- **Key mental model**
  - Images are artifacts you build, version, and share
  - Containers are temporary runtime instances created from images
  - This separation enables consistency across machines and environments

Base Images, Layering, and Trust

- **Base images**
  - A base image is the starting point for building another image
  - Examples include minimal Linux environments or language runtimes
  - Choosing a base image affects size, security surface, and compatibility
- **Layered filesystem model**
  - Images are built as a stack of layers, one per build step
  - Layers are cached and reused across images when possible
  - This makes builds faster and image distribution more efficient
- **Public images and trust**
  - Public images are commonly pulled from **Docker Hub**
  - Image sources vary in quality, maintenance, and security posture
  - Official and well-maintained images reduce risk but still require scrutiny
- **Practical implication**
  - Images are easy to reuse and share, but easy to misuse
  - Understanding provenance and update practices matters
  - **Docker** provides tooling, but responsibility for trust remains with the user

Dockerfiles: Purpose and Structure

- **Purpose of a Dockerfile**
  - A Dockerfile is a declarative recipe for building a Docker image
  - It specifies *what* the environment should contain, not *how* to manually create it
  - The file is version-controlled alongside application code
- **Why Dockerfiles matter**
  - They make environment setup explicit and reproducible
  - Anyone with the Dockerfile can rebuild the same image
  - They eliminate undocumented setup steps and “tribal knowledge”
- **Mental model**
  - A Dockerfile describes a sequence of filesystem and configuration changes
  - Each instruction produces a new image layer
  - The final image is the cumulative result of all steps

Dockerfiles: Instructions and Building Images

- **Common Dockerfile instructions**
  - FROM: selects the base image to build on top of
  - RUN: executes commands at build time to install software or modify the image
  - COPY / ADD: places files from the host into the image filesystem
  - CMD: specifies the default command when a container starts
  - ENTRYPOINT: defines the primary executable for the container
- **Build-time vs run-time**
  - FROM, RUN, and COPY affect the image itself
  - CMD and ENTRYPOINT define container startup behavior
  - Understanding this distinction prevents common configuration mistakes
- **Building an image**
  - The image is built by running a build command in a directory containing a Dockerfile
  - Docker executes instructions top-to-bottom, caching layers when possible
  - The result is a tagged image ready to be run or shared
- **Big picture**
  - Dockerfiles turn environments into code
  - They are the foundation for reproducible builds and CI pipelines
  - Tools like **Docker** standardize this workflow across machines and teams

Example

```
# Use Ubuntu 24.04 as the base image
FROM ubuntu:24.04

# Avoid interactive prompts during package install
ENV DEBIAN_FRONTEND=noninteractive

# Install system dependencies needed to install uv
RUN apt-get update && apt-get install -y \
    curl \
    ca-certificates \
    python3 \
    && rm -rf /var/lib/apt/lists/*

# Install uv
RUN curl -Ls https://astral.sh/uv/install.sh | sh

# Ensure uv is on PATH
ENV PATH="/root/.cargo/bin:${PATH}"

# Create a simple Python script
RUN echo 'print("Hello Docker")' > /hello.py

# Run the script by default
CMD ["python3", "/hello.py"]
```

Running the example

```
% docker build -t hello-docker .

% docker run hello-docker

% docker run -it hello-docker /bin/bash
```

Running Containers: Starting and Managing Execution

- **Running a container**
  - Running a container means starting a process inside an isolated environment
  - The container is created from an image and exists only while that process runs
  - Containers are intended to be easy to start, stop, and discard
- **Foreground vs detached mode**
  - Foreground mode runs the container attached to the terminal
  - Output is streamed directly, and stopping the terminal stops the container
  - Detached mode runs the container in the background as a service
  - The container continues running independently of the terminal session
- **Execution model**
  - A container typically runs a single main process
  - When that process exits, the container stops
  - This design encourages simple, composable services

Running Containers: Networking and Data Persistence

- **Port mapping**
  - Containers have their own network namespace and internal ports
  - Port mapping exposes a container's internal port to the host system
  - This allows services running inside containers to be accessed externally
- **Basic container networking**
  - Containers can communicate with the host and with each other
  - Networking defaults are designed to be simple for local development
  - More advanced networking is layered on top of these basics
- **Mounting volumes**
  - By default, container filesystems are ephemeral
  - Volumes allow data to persist beyond the lifetime of a container
  - Bind mounts enable live code edits from the host to appear inside the container
- **Why this matters**
  - Containers remain disposable while data persists safely
  - Development workflows can use live reload without rebuilding images
  - **Docker** provides these primitives to balance isolation with practicality

Finding and Understanding Images

- **Pulling images from registries**
  - Container images are typically stored in remote registries
  - Pulling an image downloads its layers to the local machine
  - Images are identified by names and optional version tags
- **Common registries**
  - Public images are most often pulled from **Docker Hub**
  - Organizations may run private registries for internal use
  - Registries act as distribution points, not execution environments
- **Why registries matter**
  - Images can be shared consistently across teams and systems
  - Versioned images enable reproducible deployments
  - Registries decouple image creation from image execution

Working with Existing Images: Inspection and Execution

- **Inspecting images**
  - Images can be examined to understand their layers, size, and metadata
  - Inspection reveals exposed ports, default commands, and environment variables
  - This helps users treat images as transparent artifacts rather than black boxes
- **Inspecting containers**
  - Running containers expose runtime state such as network settings and mounts
  - Inspection helps diagnose configuration and connectivity issues
  - Logs provide visibility into application behavior
- **Running prebuilt images**
  - Many common services are available as ready-to-run images
  - Web servers, databases, and development tools can be launched without installation
  - Configuration is typically provided through environment variables or mounts
- **Practical takeaway**
  - Using prebuilt images accelerates experimentation and learning
  - Understanding inspection tools prevents misuse and confusion
  - **Docker** enables this workflow without requiring custom image builds

Development Workflow: Docker in Day-to-Day Development

- **Using Docker for local development**
  - Docker allows developers to run applications in environments that closely match production
  - Language runtimes, system libraries, and tools are defined once and reused everywhere
  - Local machines become hosts rather than snowflake environments
- **Containers as development environments**
  - Editors and tools interact with services running inside containers
  - Multiple projects with conflicting dependencies can coexist on the same machine
  - Setup is reduced to starting containers instead of manual installation
- **Workflow implications**
  - New developers or students can start working with minimal setup
  - Environment differences stop being a primary source of bugs
  - Development becomes more predictable and repeatable

Development Workflow: Reproducibility and Iteration

- **Eliminating “it works on my machine”**
  - Applications are tested and run in the same containerized environment everywhere
  - Configuration is explicit and version-controlled
  - Failures are easier to reproduce and debug across machines
- **Rebuilding images vs reusing containers**
  - Images are rebuilt when the environment or dependencies change
  - Containers are reused or restarted during normal development cycles
  - Containers are treated as disposable; images are the durable artifacts
- **Efficient iteration**
  - Code changes can be mounted into running containers without rebuilding images
  - Rebuilds are reserved for dependency or configuration changes
  - This balance enables fast feedback while preserving reproducibility
- **Big picture**
  - Docker shifts development from machine-centric to artifact-centric workflows
  - Teams reason about environments as code, not setup instructions
  - **Docker** provides the tooling that makes this practical at scale

Docker Compose: Why and What

- **Why multiple containers are needed**
  - Real applications are composed of multiple services, not a single process
  - Common components include a web server, application backend, and database
  - Separating services into containers preserves modularity and isolation
- **Single responsibility principle**
  - Each container is designed to run one main service
  - Services can be updated, restarted, or replaced independently
  - This mirrors how applications are structured in production environments
- **What Docker Compose provides**
  - Docker Compose defines multi-container applications declaratively
  - Services, networks, and volumes are described in a single configuration file
  - A single command can start or stop the entire application stack

Docker Compose: Scope and Appropriateness

- **Defining multi-container applications**
  - Each service specifies its image, configuration, and dependencies
  - Networking between services is handled automatically
  - Environment variables and volumes are centralized and explicit
- **When Docker Compose is appropriate**
  - Local development and testing of multi-service applications
  - Teaching environments where reproducibility matters
  - Small-scale deployments and demos
- **When Compose is overkill**
  - Single-container applications
  - Very simple scripts or short-lived experiments
  - Large-scale production systems that require orchestration
- **Context**
  - Docker Compose focuses on developer convenience and clarity
  - It intentionally avoids complex scheduling and scaling logic
  - **Docker** positions Compose as a bridge between single containers and full orchestration systems