ChatGPT

Interactive Docker Course Specifications

Overview

This specification defines a complete, interactive curriculum for learning **Docker** from first principles through to upper-intermediate topics. The goal is to build a self-paced, hands-on course that leverages GitHub Copilot's agent capabilities to generate interactive lessons, command-line tasks and small projects. Learners should move from understanding what containers are and how they differ from virtual machines to building, running and orchestrating multi-container applications.

Target audience

The course is aimed at developers, DevOps engineers and students with basic command-line experience who wish to:

- Understand container technology and how Docker implements it.
- Build, run and manage Docker containers in local and cloud environments.
- Persist data, configure networks and compose multi-service applications.
- Optimise images using best practices and multi-stage builds.
- Explore orchestration with Docker Swarm and discuss next steps toward Kubernetes.

Course structure

The course is organised into modules. Each module contains:

1. **Learning objectives** describing what the learner should achieve.
2. **Conceptual explanations** presented as narrative text, diagrams or short videos.
3. **Interactive exercises** that can be executed via an embedded terminal or code block; these tasks should instruct learners to run commands, write Dockerfiles or edit YAML files.
4. **Short assessments** (multiple choice or fill-in-the-blank) to reinforce comprehension.
5. **Hints and solutions** so learners can recover from mistakes without losing momentum.

GitHub Copilot agent should generate the scaffolding for each lesson, including markdown content, code snippets, tasks with clear instructions and validation logic. The agent should also produce template files (e.g. empty Dockerfiles or compose files) and pre-configure terminal environments when necessary.

Module specifications

Module 1 – Introduction to Containers and Virtualisation

Objectives

- Define what a container is and explain how it differs from a virtual machine.

- Recognise that containers share the host kernel but run in isolated namespaces and cgroups.
- Understand the benefits of containerisation for application consistency and portability.

Content & activities

1. **Narrative**: Explain the high-level difference between traditional virtual machines and containers, emphasising that containers share the host operating system kernel and are therefore lighter and faster to start. Include a short diagram comparing the two.
2. **Hands-on**: Guide the learner to run `docker version` and `docker info` to observe engine details and confirm the host's kernel. Use an interactive shell task to validate that the commands produce output.
3. **Quiz**: Present a short multiple-choice assessment asking which component containers share and why that matters for resource consumption.

Module 2 – Docker Architecture and CLI Fundamentals

Objectives

- Describe Docker's client–server architecture, including the roles of the **Docker daemon** (`dockerd`), **Docker client** and **Docker registry**.
- Use basic CLI commands (`docker images`, `docker ps`, `docker run`, `docker stop`, etc.) to manage containers.
- Understand that the client communicates with the daemon through a REST API and that registries store and distribute images.

Content & activities

1. **Diagram**: Visualise the architecture showing the client sending commands to the daemon, which interacts with the underlying kernel features (namespaces and cgroups) and the registry.
2. **Interactive exercise**: Provide tasks to list available images and running containers, then run an official image (e.g. `hello-world`). Verify successful execution by checking container status.
3. **Reflection**: Ask learners to explain how the client, daemon and registry work together when they run `docker run alpine echo "Hello"`.

Module 3 – Images and Image Creation

Objectives

- Differentiate between a **Docker image** and a **container** (an image is a read-only template; a container is a running instance of an image).
- Inspect images and understand layered file systems.
- Write a simple Dockerfile, build an image and tag it appropriately.
- Push and pull images from a registry.

Content & activities

1. **Narrative**: Describe how images are composed of layers and how tags allow versioning. Emphasise immutability of images.

2. **Exercise**: Provide a pre-populated project with an `app.py` script. Ask learners to write a Dockerfile that sets an appropriate base image (e.g. `python:3.10-slim`), copies the application, installs dependencies and defines a default command. Use an interactive terminal to build the image, run it and tag it.
3. **Registry interaction**: If network access is available, instruct learners to sign into Docker Hub or an internal registry, then push their image and pull it back down. Include fallback instructions if a network registry is not available.

Module 4 – Managing Containers

Objectives

- Run containers with various options (detached mode, port mapping, environment variables, resource limits).
- Attach to a running container, view logs and execute commands inside with `docker exec`.
- Start, stop, restart and remove containers, and examine exit codes.
- Understand the lifecycle of a container.

Content & activities

1. **Scenarios**: Ask learners to run a simple web server image with port mapping and environment variables. Provide tasks to inspect logs, open a shell inside the container and adjust resource limits.
2. **Lifecycle practice**: Create tasks where containers are started and stopped repeatedly; emphasise the difference between removing a container (`docker rm`) and removing its underlying image.
3. **Quiz**: Questions on what happens when a container process exits, how to retrieve exit codes and how to view logs.

Module 5 – Persistent Storage with Volumes

Objectives

- Explain the difference between volumes and bind mounts and why volumes are the preferred mechanism for persisting data.
- Create, list and remove volumes.
- Mount volumes into containers and share them across multiple containers.
- Understand that volumes exist outside the container lifecycle and survive container deletion.

Content & activities

1. **Narrative**: Discuss how volumes are managed by Docker, provide better performance than writing to the container file system and are safe to share between containers.
2. **Exercise**: Guide learners to create a volume (`docker volume create mydata`), mount it into a container running a database or file server, then inspect the volume's contents on the host. Show that removing the container leaves the volume intact.
3. **Multiple-container scenario**: Use two containers (e.g. an application and a backup utility) that both mount the same volume to illustrate data sharing.

Module 6 – Networking and Connectivity

Objectives

- Understand Docker network drivers (bridge, host, overlay, ipvlan, macvlan, none) and choose appropriate ones for a given scenario.
- Create custom bridge networks to isolate application components.
- Connect and disconnect containers from networks and perform port mappings.
- Explore multi-host networking concepts (overlay) in theory or via local demos.

Content & activities

1. **Narrative**: Summarise built-in drivers and their typical use cases. Highlight that bridge is the default for single-host containers, host removes isolation, overlay enables multi-host communication (e.g. in Swarm), macvlan gives each container its own MAC address (useful for legacy apps) and ipvlan controls IPv4/IPv6 assignments.
2. **Exercise**: Instruct learners to create a user-defined bridge network (`docker network create mynet`), attach two containers and verify they can communicate via container names. Then map ports to the host and confirm accessibility.
3. **Challenge**: Discuss overlay networks conceptually and provide optional tasks for learners with multiple Docker hosts to practise connecting across hosts.

Module 7 – Multi-Container Applications with Docker Compose

Objectives

- Explain what Docker Compose is and why it simplifies managing multi-service stacks.
- Write a `docker-compose.yml` file to define services, networks and volumes.
- Use `docker compose up`, `down`, `logs`, `ps` and `scale` to manage the stack.
- Manage environment variables and secret configuration within Compose files.

Content & activities

1. **Narrative**: Describe the benefits of Compose for local development, test and CI pipelines, emphasising the declarative nature of YAML configuration and reproducibility.
2. **Project**: Provide an example stack (e.g. a Flask web app talking to a Postgres database). Ask learners to write a compose file that builds the web image, pulls the database image, sets up a network and volume, and defines dependencies (`depends_on`).
3. **Scaling**: Show how to scale services (`docker compose up --scale web=3`) and observe how port mapping and load balancing work.
4. **Cleanup**: Use `docker compose down --volumes` to remove containers and persistent volumes when appropriate.

Module 8 – Writing Dockerfiles and Best Practices

Objectives

- Understand the differences between `RUN`, `CMD` and `ENTRYPOINT` instructions and when to use each.
- Apply multi-stage builds to reduce image size and separate build and runtime dependencies.
- Choose appropriate base images and keep images small and secure.
- Use `.dockerignore` to exclude unnecessary files and follow the principle of one process per container.
- Rebuild images regularly and implement minimal privileges and secrets management.

Content & activities

1. **Narrative**: Summarise best practices, including combining commands in a single `RUN` layer, setting a clear default command with `CMD`, and enforcing a fixed executable with `ENTRYPOINT`. Explain why multi-stage builds lead to smaller, more secure production images, and emphasise using official or verified base images.
2. **Exercise**: Provide a Dockerfile with redundant layers and insecure patterns; ask learners to refactor it using multi-stage builds, `.dockerignore` and proper commands. Validate that the resulting image is smaller and functions correctly.
3. **Quiz**: Ask conceptual questions about the implications of placing `CMD` vs `ENTRYPOINT` at the end of a Dockerfile and the effect of a `.dockerignore` file.

Module 9 – Orchestration with Docker Swarm

Objectives

- Introduce container orchestration and position Docker Swarm as a simple built-in orchestrator.
- Define Swarm clusters, nodes (manager vs worker), services and tasks.
- Understand concepts of desired state, high availability via Raft consensus and built-in load balancing.
- Deploy and manage services on a single-node or multi-node Swarm cluster.

Content & activities

1. **Narrative**: Explain that a Swarm is a collection of nodes acting as a single system, with managers making scheduling decisions and workers executing tasks. Highlight that at least three managers are recommended for high availability and that services can be replicated or global.
2. **Lab**: Provide instructions to initialise a Swarm on the learner's machine (`docker swarm init`), create a replicated service (e.g. `nginx`), scale it, and observe load balancing via published ports. If learners have multiple nodes, include steps to add workers and managers.
3. **Discussion**: Summarise differences between Swarm and Kubernetes, emphasising simplicity vs extensive features, and encourage learners to explore Kubernetes as a next step.

Module 10 – Security Basics and Best Practices

Objectives

- Recognise that containers do not inherently provide strong security isolation and rely on kernel features such as namespaces, cgroups and capabilities.
- Apply minimal privilege principles (running as non-root, dropping unnecessary capabilities).
- Use trusted base images and vulnerability scanning tools.
- Secure secrets and environment variables.
- Harden network configurations and avoid exposing unnecessary ports.

Content & activities

1. **Narrative**: Provide a high-level overview of container security. Explain that although containers use isolation mechanisms, they still share the kernel, so kernel vulnerabilities affect all containers. Encourage using small base images, scanning for vulnerabilities and following least-privilege principles.
2. **Exercise**: Ask learners to modify a Dockerfile so that the application runs as a non-root user and to use environment variables rather than hard-coded secrets. Provide a scanning step using a free tool (if available) to detect outdated dependencies.
3. **Checklist**: Offer a security checklist for learners to reference in future projects.

Module 11 – Final Project and Next Steps

Objectives

- Consolidate all knowledge by building an end-to-end application using Docker.
- Design, build and orchestrate a multi-service stack with volumes, networks and optimised Dockerfiles.
- Demonstrate the ability to debug, scale and update services.
- Reflect on further areas for exploration (e.g. Kubernetes, advanced networking, CI/CD integration).

Project outline

1. **Scenario**: Present a simple microservice application (for example, a task manager API with a frontend and a backing database). Provide starter code and require learners to:
2. Write multi-stage Dockerfiles for each component.
3. Create a compose file defining services, networks and volumes.
4. Build and run the stack locally, then scale specific services.
5. Persist data using volumes and demonstrate backup and restore.
6. Perform a rolling update with Swarm (or simulate an update in Compose) without downtime.
7. **Validation**: Use GitHub Copilot agent to check that the application builds successfully, services can communicate and tasks can be created and retrieved.
8. **Reflection**: Ask learners to document challenges and next steps, such as exploring Kubernetes, Helm or container security tools.

Expectations for GitHub Copilot Agent

GitHub Copilot agent should interpret this specification to generate the course content. At a minimum, the agent must:

- Create individual markdown files or structured modules corresponding to each section above.
- Embed code blocks and shell tasks that learners can run. Provide automated checks to validate command output and file contents.
- Generate diagrams (using ASCII art or embedded images) to illustrate concepts like architecture, networking and multi-stage builds.
- Produce quizzes and provide answer keys with explanations.
- Supply starter project files and solutions for exercises, including Dockerfiles, compose files and source code where appropriate.

The resulting course should be navigable as a GitHub repository with a clear README and module index. The repository should be ready for learners to clone and begin the course immediately.
