Deployment

How this category's localhost assumption, present in every lesson so far, actually gets deployed in the real world: small images with multi-stage Dockerfiles, orchestrating the whole system (eureka-server, config-server, api-gateway, order-service, inventory-service, Kafka) in one docker-compose file, environment-variable configuration in containers, why depends_on isn't enough without a health check, and a brief, honest look at Kubernetes.

Intermediate 26 min

Last updated: August 23, 2026
Read online: https://www.learnforgex.com/en/topics/deployment

---

Deployment

Every lesson in this Microservices category has assumed order-service, inventory-service, eureka-server, config-server, api-gateway, and Kafka are already running somewhere, reachable at `localhost`. That assumption has been doing real work quietly this whole time -- and it stops holding the moment any of these pieces needs to run somewhere that ISN'T the same machine. This closing lesson covers how the system this category built actually gets deployed.

What Does Deployment Mean for a Microservices System?

Deploying a single application usually means packaging it and running it somewhere. Deploying a microservices system means packaging and running MANY independently deployable pieces -- each with its own image, its own configuration, its own startup dependencies on the OTHERS -- and making sure they can all find each other once they're no longer all sharing the same `localhost`.

Why Does It Exist?

Every `localhost:8761`, `localhost:8888`, and `localhost:9092` this category's earlier lessons wrote assumed every service runs on the SAME machine, during local development. That assumption is exactly right for the examples this course builds -- and exactly wrong the moment any service moves to its own container, its own machine, or its own cloud instance. Deployment is the practice of making a system designed and tested this way actually run somewhere else, without rewriting its configuration by hand for every new environment.

History

Docker, released in 2013, popularized the container -- a package containing an application and everything it needs to run, isolated from whatever else is on the host machine, without the overhead of a full virtual machine. This solved a problem microservices make sharper than a single application ever did: many independently built pieces, each with potentially different dependency versions, all needing to run on the same infrastructure without interfering with each other. Docker Compose (packaged with Docker itself) extended this to ORCHESTRATING multiple containers together for local development -- exactly the scale of coordination this category's six pieces (order-service, inventory-service, eureka-server, config-server, api-gateway, and Kafka) now need.

Containerizing a Single Service: order-service's Dockerfile

A Dockerfile describes how to build an image -- a self-contained package including order-service's own jar and just enough of a Java runtime to run it.

```
# order-service's own Dockerfile -- lives at the root of order-service's own
# source tree, named plainly "Dockerfile" in a real project (this course's
# example naming convention adds a descriptive prefix, see "Containerizing a
# Single Service: order-service's Dockerfile"). Every OTHER service in this
# category (inventory-service, eureka-server, config-server, api-gateway) gets
# its own near-identical Dockerfile, changed only in which jar it copies.

# --- Build stage: has the JDK and Maven, but never ships in the final image ---
FROM eclipse-temurin:21-jdk AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN --mount=type=cache,target=/root/.m2 mvn -B package -DskipTests

# --- Runtime stage: only a JRE and the already-built jar ---
FROM eclipse-temurin:21-jre
WORKDIR /app
COPY --from=build /app/target/order-service.jar app.jar
EXPOSE 8081
ENTRYPOINT ["java", "-jar", "app.jar"]
```

A Multi-Stage Build: Keeping the Image Small

Notice `OrderServiceDockerfile.dockerfile` uses TWO stages: a `build` stage with the full JDK and Maven, and a separate final stage with only a JRE. The final image never includes Maven, the JDK's compiler, or order-service's own source code -- only the already-built jar and what's needed to RUN it. This keeps the shipped image meaningfully smaller and reduces its attack surface, without giving up anything the build itself needed.

Orchestrating the Whole System: docker-compose

One file describes every piece this category built, how they're built, and how they depend on each other.

```
# Real filename: docker-compose.yml, at the root of a workspace containing
# every service this category has built -- order-service, inventory-service,
# eureka-server (Service Discovery & Eureka), config-server (Configuration
# Management), api-gateway (API Gateway), and Kafka (Event-Driven Architecture
# & Kafka). One "docker compose up" starts the WHOLE system this course has
# been building, service by service, across nine lessons.

services:
  eureka-server:
    build: ./eureka-server
    ports:
      - "8761:8761"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8761/actuator/health"]
      interval: 5s
      retries: 10

  config-server:
    build: ./config-server
    ports:
      - "8888:8888"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8888/actuator/health"]
      interval: 5s
      retries: 10

  kafka:
    image: apache/kafka:3.9.0
    ports:
      - "9092:9092"

  api-gateway:
    build: ./api-gateway
    ports:
      - "8090:8090"
    depends_on:
      eureka-server:
        condition: service_healthy   # see "Startup Order: Why depends_on Isn't
                                      # Enough" -- waits for a PASSING health
                                      # check, not just a started container

  order-service:
    build: ./order-service
    depends_on:
      eureka-server:
        condition: service_healthy
      config-server:
        condition: service_healthy
      kafka:
        condition: service_started   # Kafka's own image has no built-in
```

```
                                    # healthcheck here -- see the warning in
                                    # "Startup Order" about what
                                    # service_started does NOT guarantee

  inventory-service:
    build: ./inventory-service
    depends_on:
      eureka-server:
        condition: service_healthy
      config-server:
        condition: service_healthy
      kafka:
        condition: service_started
```

> This is genuinely the FIRST time this course's nine microservices lessons are described together in one place -- everything from `microservices-fundamentals` through `security` built toward exactly this file.

Configuration in Containers: Environment Variables Over application.yml

Every `localhost` this category's earlier lessons hardcoded into an `application.yml` -- Eureka's `defaultZone`, Config Server's `spring.config.import`, Kafka's `bootstrap-servers` -- breaks once a service runs in its OWN container, because "localhost" then refers to that container, not to `eureka-server`'s.

```
# order-service's application.yml, as it needs to look when running inside
# Docker Compose (see DockerComposeConfig.yml) -- every "localhost" this
# category's earlier lessons used (Service Discovery & Eureka's
# eureka.client.service-url.defaultZone, Configuration Management's
# spring.config.import, Event-Driven Architecture & Kafka's
# spring.kafka.bootstrap-servers) breaks once order-service runs in its OWN
# container, because "localhost" then means order-service's OWN container,
# not eureka-server's.

spring:
  config:
    import: "configserver:${CONFIG_SERVER_URL:http://localhost:8888}"
  kafka:
    bootstrap-servers: ${KAFKA_BOOTSTRAP_SERVERS:localhost:9092}

eureka:
  client:
    service-url:
      defaultZone: ${EUREKA_URL:http://localhost:8761/eureka/}

# In DockerComposeConfig.yml, Compose's own internal DNS resolves a service's
# NAME (eureka-server, config-server, kafka) to its container's address --
# these three environment variables, set ONLY in the container environment,
# override the ${...:localhost...} defaults above with those service names
# (CONFIG_SERVER_URL=http://config-server:8888, and so on). Running order-
# service directly (not in a container) still works exactly as every earlier
# lesson in this category already showed, unchanged, since the defaults are
# still "localhost". This is the SAME environment-variable-override pattern
# the Configuration Management lesson already used for secrets (see its
# ${ORDERS_DB_PASSWORD} discussion) -- applied here to service ADDRESSES
# instead of a password.
```

> This is the SAME environment-variable-override pattern the Configuration Management lesson already used for secrets like `ORDERS_DB_PASSWORD` -- applied here to service ADDRESSES instead. Nothing new is being introduced; container deployment just makes THIS particular use of it necessary rather than optional.

Startup Order: Why depends_on Isn't Enough

`DockerComposeConfig.yml`'s `depends_on: condition: service_healthy` waits for a service's `/actuator/health` endpoint to actually pass, not just for its container to have started -- eureka-server's process starting doesn't mean it's ready to accept registrations yet.

> Notice `kafka` uses `condition: service_started`, not `service_healthy` -- the plain Kafka image used here has no built-in health check. `service_started` only confirms the container process began running, NOT that Kafka is actually ready to accept connections -- order-service and inventory-service's own retry behavior (see the Resilience4j lesson) is what actually absorbs the gap between "Kafka's container started" and "Kafka is ready," not this dependency declaration.

Beyond Local: A Brief, Honest Look at Kubernetes

Docker Compose is genuinely a LOCAL development and single-machine tool -- it doesn't run a service across multiple machines, doesn't restart a crashed container onto a different host, and doesn't have its own built-in service discovery the way Kubernetes does (see the Service Discovery & Eureka lesson's honest note that Eureka usually isn't needed ON Kubernetes for exactly this reason). Kubernetes solves problems at a scale this category's examples never actually reach.

```
# A PREVIEW, not a working setup this lesson builds toward -- see "Beyond
# Local: A Brief, Honest Look at Kubernetes". This is what order-service's
# Docker image (built from OrderServiceDockerfile) would look like described
# as a Kubernetes Deployment instead of a Docker Compose service -- shown here
# only so the SHAPE looks familiar, not as something this lesson walks through
# setting up.

apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
spec:
  replicas: 3                          # Kubernetes runs THREE instances of
                                        # order-service by default here --
                                        # Docker Compose, by contrast, runs
                                        # exactly one of each service unless
                                        # told otherwise
  selector:
    matchLabels:
      app: order-service
  template:
    metadata:
      labels:
        app: order-service
    spec:
      containers:
        - name: order-service
          image: order-service:latest
          ports:
            - containerPort: 8081
          env:
            - name: EUREKA_URL
              value: http://eureka-server:8761/eureka/
```

> This is shown only so the SHAPE looks familiar -- actually setting up and operating a Kubernetes cluster is a large enough topic on its own to be genuinely out of scope for this lesson to build toward. The container image `OrderServiceDockerfile.dockerfile` produces is the SAME image Kubernetes would run -- Kubernetes changes how many instances run and how they're orchestrated, not how the image itself is built.

Best Practices

- **Use a multi-stage build for every service's Dockerfile** -- see "A Multi-Stage Build: Keeping the Image Small" -- the pattern is identical across order-service, inventory-service, and every other service in this category.
- **Override configuration with environment variables for anything that changes between environments**, keeping the same `localhost` defaults this category's earlier lessons already used for local development -- see `OrderServiceContainerConfig.yml`.
- **Use health-check-based startup ordering (`condition: service_healthy`) wherever a real health check exists**, and lean on a service's own retry/resilience behavior (see the Resilience4j lesson) for dependencies that don't provide one -- see the warning in "Startup Order".
- **Treat Docker Compose as a local development and demonstration tool**, not a production deployment target -- see "Beyond Local".

Common Mistakes

- **Shipping a Dockerfile without a multi-stage build.** A single-stage build ships the entire JDK, Maven, and the build cache inside the final image -- far larger than necessary, and a larger attack surface.
- **Assuming `depends_on` (without a health check condition) means a dependency is actually READY**, not just started. A container that's running isn't necessarily accepting traffic yet.
- **Hardcoding `localhost` into a Dockerfile or a container image itself**, instead of an environment variable resolved at container startup -- see `OrderServiceContainerConfig.yml` for the alternative.
- **Reaching for Kubernetes before actually needing what it solves.** The scale problems Kubernetes exists for (multi-machine orchestration, automatic rescheduling) aren't the same problems Docker Compose already solves well for local development -- see "Beyond Local".

Summary, Cheat Sheet, and Glossary

Deployment makes a microservices system designed with `localhost` defaults actually run somewhere else. A multi-stage Dockerfile keeps each service's shipped image small; Docker Compose orchestrates every piece this category built (order-service, inventory-service, eureka-server, config-server, api-gateway, Kafka) together for local development, with environment variables overriding `application.yml` defaults per environment, and health-check-based `depends_on` conditions handling startup order where a health check exists. Kubernetes solves a different, larger-scale set of problems -- genuinely out of scope to build here, but shown briefly so its shape looks familiar.

Quick reference:

```
FROM eclipse-temurin:21-jdk AS build
WORKDIR /app
COPY . .
RUN mvn -B package -DskipTests

FROM eclipse-temurin:21-jre
COPY --from=build /app/target/app.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

```
# docker-compose.yml
services:
  order-service:
    build: ./order-service
    depends_on:
      eureka-server:
        condition: service_healthy
```

Glossary

Container — A self-contained package including an application and everything it needs to run, isolated from the host machine.

Multi-Stage Build — A Dockerfile pattern that uses a separate build stage (with full build tools) and a slimmer final stage (with only what's needed to run).

Docker Compose — A tool for orchestrating multiple containers together, primarily for local development.

Health Check — A probe (like `/actuator/health`) a container orchestrator uses to determine whether a service is actually ready, not just started.

Kubernetes — A container orchestration platform for running services across multiple machines at a scale beyond what Docker Compose targets.