Deployment

Bu kategorinin şimdiye kadar her dersinin localhost varsayımının gerçek dünyada nasıl deploy edildiği: multi-stage Dockerfile ile küçük image'lar, tüm sistemi (eureka-server, config-server, api-gateway, order-service, inventory-service, Kafka) tek bir docker-compose dosyasında orkestre etmek, container'larda ortam değişkeni ile yapılandırma, depends_on'un health check olmadan neden yetmediği, ve Kubernetes'e kısa/dürüst bir bakış.

Orta 26 dk

Son güncelleme: 6 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/deployment

---

Deployment

Bu Mikroservisler kategorisindeki her ders, order-service, inventory-service, eureka-server, config-server, api-gateway ve Kafka'nın zaten bir yerde çalıştığını, `localhost`'ta ulaşılabilir olduğunu varsaydı. Bu varsayım tüm bu süre boyunca sessizce gerçek bir iş yaptı -- ve bu parçalardan herhangi biri AYNI makine OLMAYAN bir yerde çalışması gerektiği anda geçerliliğini yitiriyor. Bu kapanış dersi, bu kategorinin inşa ettiği sistemin gerçekte nasıl deploy edildiğini kapsıyor.

Bir Mikroservis Sistemi İçin Deployment Ne Demek?

Tek bir uygulamayı deploy etmek genellikle onu paketleyip bir yerde çalıştırmak demektir. Bir mikroservis sistemini deploy etmek, BİRÇOK bağımsız dağıtılabilir parçayı paketlemek ve çalıştırmak demektir -- her birinin kendi image'ı, kendi yapılandırması, DİĞERLERİNE olan kendi başlangıç bağımlılıkları var -- ve artık hepsi aynı `localhost`'u paylaşmadığında birbirlerini bulabilmelerini sağlamak.

Neden Var?

Bu kategorinin önceki derslerinin yazdığı her `localhost:8761`, `localhost:8888` ve `localhost:9092`, her servisin, yerel geliştirme sırasında, AYNI makinede çalıştığını varsaydı. Bu varsayım bu kursun inşa ettiği örnekler için tam olarak doğru -- ve herhangi bir servis kendi container'ına, kendi makinesine ya da kendi cloud instance'ına taşındığı anda tam olarak

yanlış. Deployment, bu şekilde tasarlanmış ve test edilmiş bir sistemin, her yeni ortam için yapılandırmasını elle yeniden yazmadan, gerçekten başka bir yerde çalışmasını sağlama pratiğidir.

Tarihçe

2013'te yayınlanan Docker, container'ı popülerleştirdi -- bir uygulamayı ve çalışması için ihtiyaç duyduğu HER ŞEYİ içeren, host makinedeki başka her şeyden izole, tam bir sanal makinenin ek yükü olmadan paketlenmiş bir paket. Bu, mikroservislerin tek bir uygulamanın hiçbir zaman yapmadığı kadar keskinleştirdiği bir sorunu çözdü: her biri potansiyel olarak farklı bağımlılık sürümlerine sahip birçok bağımsız inşa edilmiş parça, hepsinin birbirine karışmadan aynı altyapıda çalışması gerekiyor. Docker Compose (Docker'ın kendisiyle birlikte paketlenmiş), bunu yerel geliştirme için birden fazla container'ı BİRLİKTE ORKESTRE etmeye genişletti -- bu kategorinin altı parçasının (order-service, inventory-service, eureka-server, config-server, api-gateway, ve Kafka) şimdi tam olarak ihtiyaç duyduğu koordinasyon ölçeği.

Tek Bir Servisi Container'a Almak: order-service'in Dockerfile'ı

Bir Dockerfile, bir image'ın nasıl inşa edileceğini tarif eder -- order-service'in kendi jar'ını ve onu çalıştırmaya yetecek kadar bir Java runtime'ını içeren, kendi kendine yeterli bir paket.

```
# 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"]
```

Multi-Stage Bir Build: Image'ı Küçük Tutmak

`OrderServiceDockerfile.dockerfile`'ın İKİ aşama kullandığına dikkat edin: tam JDK ve Maven'a sahip bir `build` aşaması, ve yalnızca bir JRE'ye sahip ayrı bir son aşama. Son image hiçbir zaman Maven'ı, JDK'nın derleyicisini, ya da order-service'in kendi kaynak kodunu

içermez -- yalnızca zaten inşa edilmiş jar'ı ve onu ÇALIŞTIRMAK için gerekeni. Bu, gönderilen image'ı anlamlı ölçüde küçük tutar ve saldırı yüzeyini azaltır, build'in kendisinin ihtiyaç duyduğu hiçbir şeyden vazgeçmeden.

Sistemin Tamamını Orkestre Etmek: docker-compose

Tek bir dosya, bu kategorinin inşa ettiği her parçayı, nasıl inşa edildiklerini, ve birbirlerine nasıl bağımlı olduklarını tarif eder.

```
# 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
```

> Bu, kursun dokuz mikroservis dersinin tek bir yerde birlikte tarif edildiği GERÇEKTEN İLK an -- `microservices-fundamentals`'tan `security`'ye kadar her şey tam olarak bu dosyaya doğru inşa edildi.

Container'larda Yapılandırma: application.yml Yerine Ortam Değişkenleri

Bu kategorinin önceki derslerinin bir `application.yml`'e sabit kodladığı her `localhost` -- Eureka'nın `defaultZone`'u, Config Server'ın `spring.config.import`'u, Kafka'nın `bootstrap-servers`'ı -- bir servis KENDİ container'ında çalıştığı anda kırılır, çünkü "localhost" o zaman o container'ı ifade eder, `eureka-server`'ınkini değil.

```
# 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.
```

> Bu, Configuration Management dersinin `ORDERS_DB_PASSWORD` gibi sırlar için zaten kullandığı AYNI ortam-değişkeni-override deseni -- burada servis ADRESLERİNE uygulanıyor. Yeni bir şey tanıtılmıyor; container deployment yalnızca bunun BU özel kullanımını opsiyonel olmaktan gerekli olmaya çeviriyor.

Başlangıç Sırası: depends_on Neden Yetmiyor

`DockerComposeConfig.yml`'in `depends_on: condition: service_healthy`'si, bir servisin `/actuator/health` endpoint'inin gerçekten geçmesini bekler, yalnızca container'ının başlamış olmasını değil -- eureka-server'ın süreci başlaması, kayıtları kabul etmeye HAZIR olduğu anlamına gelmez.

> `kafka`'nın `service_healthy` DEĞİL, `condition: service_started` kullandığına dikkat edin -- burada kullanılan sade Kafka image'ının yerleşik bir health check'i yok. `service_started` yalnızca container sürecinin çalışmaya BAŞLADIĞINI onaylar, Kafka'nın GERÇEKTEN bağlantı kabul etmeye hazır olduğunu DEĞİL -- order-service ve inventory-service'in kendi retry davranışı (Resilience4j dersine bakınız), "Kafka'nın container'ı başladı" ile "Kafka hazır" arasındaki boşluğu gerçekte emen şey, bu bağımlılık tanımı değil.

Yerelin Ötesi: Kubernetes'e Kısa, Dürüst Bir Bakış

Docker Compose gerçekten YEREL bir geliştirme ve tek-makine aracı -- bir servisi birden fazla makine boyunca çalıştırmaz, çökmüş bir container'ı farklı bir host'a yeniden başlatmaz, ve Kubernetes'in sahip olduğu gibi kendi yerleşik servis keşfine sahip değildir (Servis Keşfi ve Eureka dersinin, Eureka'nın tam da bu nedenle genellikle Kubernetes ÜZERİNDE gerekmediğine dair dürüst notuna bakınız). Kubernetes, bu kategorinin örneklerinin hiçbir zaman gerçekten ulaşmadığı bir ölçekteki sorunları çözer.

```
# 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/
```

> Bu, yalnızca ŞEKLİ tanıdık gelsin diye gösteriliyor -- bir Kubernetes cluster'ını gerçekten kurmak ve işletmek, bu dersin inşa etmeye çalışacağı kadar kapsam dışına çıkacak kadar başlı başına büyük bir konu. `OrderServiceDockerfile.dockerfile`'ın ürettiği container image'ı, Kubernetes'in çalıştıracağı AYNI image -- Kubernetes kaç örneğin çalıştığını ve nasıl orkestre edildiğini değiştirir, image'ın kendisinin nasıl inşa edildiğini değil.

Best Practices

- **Her servisin Dockerfile'ı için multi-stage bir build kullan** -- "Multi-Stage Bir Build: Image'ı Küçük Tutmak" bölümüne bakınız -- desen order-service, inventory-service ve bu kategorideki her diğer serviste özdeş.
- **Ortamlar arasında değişen her şeyi ortam değişkenleriyle override et**, bu kategorinin önceki derslerinin yerel geliştirme için zaten kullandığı AYNI `localhost` varsayılanlarını koruyarak -- `OrderServiceContainerConfig.yml`'e bakınız.
- **Gerçek bir health check var olduğu her yerde health-check-tabanlı başlangıç sıralamasını (`condition: service_healthy`) kullan**, olmayan bağımlılıklar için bir servisin kendi retry/resilience davranışına (Resilience4j dersine bakınız) yaslan -- "Başlangıç Sırası"ndaki uyarıya bakınız.
- **Docker Compose'u bir yerel geliştirme ve gösterim aracı olarak ele al**, bir production deployment hedefi olarak değil -- "Yerelin Ötesi"ne bakınız.

Yaygın Hatalar

- **Multi-stage bir build olmadan bir Dockerfile göndermek.** Tek aşamalı bir build, tüm JDK'yı, Maven'ı, ve build cache'ini son image'ın içinde gönderir -- gerekenden çok daha büyük, ve daha büyük bir saldırı yüzeyi.
- **`depends_on`'un (bir health check koşulu olmadan) bir bağımlılığın gerçekten HAZIR olduğu anlamına geldiğini varsaymak**, yalnızca başlamış olduğunu değil. Çalışan bir container'ın mutlaka henüz trafik kabul ediyor olması gerekmez.
- **`localhost`'u bir Dockerfile'a ya da container image'ının kendisine sabit kodlamak**, container başlangıcında çözülen bir ortam değişkeni yerine -- alternatif için `OrderServiceContainerConfig.yml`'e bakınız.
- **Gerçekten ihtiyaç duymadan Kubernetes'e yönelmek.** Kubernetes'in var olma amacı olan ölçek sorunları (çoklu makine orkestrasyon, otomatik yeniden zamanlama), Docker Compose'un yerel geliştirme için zaten iyi çözdüğü sorunlarla AYNI değil -- "Yerelin Ötesi"ne bakınız.

Özet, Cheat Sheet ve Terimler Sözlüğü

Deployment, `localhost` varsayılanlarıyla tasarlanmış bir mikroservis sisteminin gerçekten başka bir yerde çalışmasını sağlar. Multi-stage bir Dockerfile, her servisin gönderilen image'ını küçük tutar; Docker Compose, bu kategorinin inşa ettiği her parçayı (order-service, inventory-service, eureka-server, config-server, api-gateway, Kafka) yerel geliştirme için birlikte orkestre eder, ortam değişkenleri `application.yml` varsayılanlarını ortam başına override eder, ve bir health check var olduğunda health-check-tabanlı `depends_on` koşulları başlangıç sırasını ele alır. Kubernetes farklı, daha büyük ölçekli bir sorun kümesini çözer -- burada inşa etmek gerçekten kapsam dışı, ama şekli tanıdık gelsin diye kısaca gösterildi.

Hızlı referans:

```
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
```

Terimler Sözlüğü

Container — Bir uygulamayı ve çalışması için ihtiyaç duyduğu her şeyi içeren, host makineden izole, kendi kendine yeterli bir paket.

Multi-Stage Build — Ayrı bir build aşaması (tam build araçlarıyla) ve daha ince bir son aşama (yalnızca çalışmak için gerekenle) kullanan bir Dockerfile deseni.

Docker Compose — Öncelikle yerel geliştirme için birden fazla container'ı birlikte orkestre etmeye yarayan bir araç.

Health Check — Bir container orkestratörünün bir servisin yalnızca başlamış değil, gerçekten hazır olup olmadığını belirlemek için kullandığı bir prob (`/actuator/health` gibi).

Kubernetes — Servisleri, Docker Compose'un hedeflediğinin ötesinde bir ölçekte, birden fazla makine boyunca çalıştırmak için bir container orkestrasyon platformu.