Production İçin Docker (Java Uygulamaları)

Layer caching ile daha hızlı rebuild'ler, root olmayan bir kullanıcı, Dockerfile HEALTHCHECK, Compose'un depends_on: condition: service_healthy'siyle gerçek bağımlılık hazırlığını beklemek, ve production konfigürasyonu/sırlar.

İleri 40 dk

Son güncelleme: 31 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/production-docker-for-java-applications

---

Bu kategoride şimdiye kadar olan her şey, gerçekten çalışan bir Dockerfile ve bir `docker-compose.yml` üretti -- ama "bir geliştiricinin kendi makinesinde çalışmak" ile "gerçekten deploy edilmeye hazır olmak" aynı çıta değil. "Docker Compose", bilinçli olarak bir boşluk açık bıraktı: `depends_on`'un, içindeki servisin gerçekten *hazır* olmasını değil, bir container'ın *başlamasını* beklemesi. Bu ders o boşluğu kapatıyor, ve bir geliştirme-seviyesi Docker kurulumunu bir production-seviyesi olandan özellikle ayıran bir avuç şeyi ekliyor: daha küçük bir image, daha hızlı rebuild'ler, root olmayan bir kullanıcı, gerçek health check'ler, ve konfigürasyon ile sırların gerçekte nerede yaşaması gerektiği.

Daha Küçük Image'lar, Yeniden Ziyaret Edilmiş

"Bir Java Base Image Seçmek", tam olarak bu nedenle tam bir JDK image'ı yerine `eclipse-temurin:21-jre`'yi zaten seçmişti -- bir base image'daki her gereksiz byte, bir saldırganın (ya da yalnızca yavaş bir deploy pipeline'ının) ilgilenmesi gerekmeyen boyuttur. `eclipse-temurin`, bu kursta şimdiye kadar kullanılan Ubuntu-tabanlı varsayılan tag'den bile daha küçük, Alpine-tabanlı varyantlar da (`21-jre-alpine`) yayınlar -- var olduğunu bilmeye değer, ama bu ders özellikle Ubuntu-tabanlı tag'e bağlı kalıyor, çünkü aşağıda `curl` için kullanılan `apt-get`'i, akıl yürütülecek ekstra, ayrı bir paket yöneticisi eklemeden kullanılabilir kılıyor.

Daha Hızlı Rebuild'ler İçin Talimatları Sıralamak (Layer Caching)

"Multi-Stage Build'ler", `mvn package`'i Docker'ın kendisinin içinde çalıştıran `builder` aşamasını zaten tanıttı -- ama o aşamanın *içindeki* talimatların sırası, henüz işlenmemiş bir nedenle önemli: Docker her katmanı önbelleğe alır, ve o katmanın bağlı olduğu hiçbir şey değişmediği sürece, onu yeniden çalıştırmak yerine önbellekteki katmanı yeniden kullanır.

```
COPY pom.xml .
RUN mvn -B dependency:go-offline

COPY src ./src
RUN mvn -B package -DskipTests
```

`mvn dependency:go-offline`, `pom.xml`'in bildirdiği her bağımlılığı, `src`'e hiç dokunmadan indirir -- `pom.xml`'i `src`'ten *önce* kopyalamak, ve o indirmeyi kendi `RUN` adımında çalıştırmak, Docker'ın bağımlılıkları yalnızca `pom.xml`'in kendisi gerçekten değiştiğinde yeniden indirmesi anlamına gelir. Tek bir Java dosyasını düzenleyip yeniden inşa etmek, artık bu projenin tüm bağımlılık ağacını yeniden indirmez -- yalnızca `COPY src ./src` katmanı ve sonrasındaki her şey yeniden çalışır, çünkü Docker'ın önbelleği, o katmanın bağlı olduğu herhangi bir şey değiştiği anda o katmanı, ve ondan sonraki her katmanı geçersiz kılar.

Root Olmayan Bir Kullanıcı Olarak Çalışmak

Varsayılan olarak, bir Dockerfile aksini söylemediği sürece, bir container içindeki bir süreç **root** olarak çalışır -- container'ın kendi dosya sistemiyle tam olarak aynı root, ki bu bir Spring Boot uygulamasının işini yapmak için gerçekte ihtiyaç duyduğundan daha fazla ayrıcalıktır.

```
RUN groupadd -r appuser && useradd -r -g appuser appuser

WORKDIR /app

COPY --from=builder --chown=appuser:appuser /build/target/learning-platform-0.1.0-SNAPSHOT.
jar app.jar

USER appuser
```

`groupadd` / `useradd`, image içinde sıradan, ayrıcalıksız bir Linux kullanıcısı oluşturur (burada mevcuttur çünkü Ubuntu-tabanlı `eclipse-temurin:21-jre` tag'i, "Daha Küçük Image'lar, Yeniden Ziyaret Edilmiş"e göre onlara sahiptir); `COPY --chown`, o yeni kullanıcıya okuması gereken JAR'ın sahipliğini verir; ve `USER appuser`, kendisinden sonraki her talimatı -- son `ENTRYPOINT`'in `java` süreci dahil -- root yerine o kullanıcı olarak çalışacak şekilde değiştirir. Bu, katmanlı savunmadır (defense in depth): her olası açığı önlemez, ama çalışan Java sürecinin ele geçirilmesinin container içinde otomatik olarak root'u vermeyeceği anlamına gelir.

Health Check'ler: Bir Dockerfile'da `HEALTHCHECK`

`HEALTHCHECK`, Docker'a çalışan bir container'a gerçekten "çalışıyor musun?" diye nasıl sorulacağını söyler -- periyodik olarak container içinde bir komut çalıştırıp başarılı olup olmadığını takip ederek.

```
HEALTHCHECK --interval=30s --timeout=3s --start-period=10s \
  CMD curl -f http://localhost:8080/en || exit 1
```

> "Bu Spring Boot uygulaması sağlıklı mı?" sorusuna cevap vermenin standart, amaca özel yolu Spring Boot Actuator'ın `/actuator/health` endpoint'idir -- ama bu proje şu anda `spring-boot-starter-actuator`'a bağımlı değil. Yukarıdaki `HEALTHCHECK`, bu projenin kendi gerçek anasayfa rotasını (`/en`, bu kurs boyunca çalışan bir container'ı doğrulamak için kullanılan aynı rota) pragmatik bir yerine geçen olarak curl'lüyor: başarılı bir yanıt, özel bir health endpoint'i olmasa bile, en azından uygulamanın başladığının ve istekleri sunduğunun kanıtıdır.

`docker ps`, bir `HEALTHCHECK` yapılandırıldığında bir container'ın sağlık durumunu (`healthy` / `unhealthy` / `starting`) gösterir -- yalnızca "süreç çökmedi"nin kendi başına sağlamadığı görünür geri bildirim.

depends_on'un Gerçekten Beklemesini Sağlamak: Compose Sağlık Koşulları

"`depends_on` — Başlangıç Sırası", düz `depends_on`'un yalnızca bir container'ın *başlamasını* beklediği konusunda uyarmıştı. Compose'un uzun-form `depends_on`'u, tam olarak bu boşluğu, yalnızca sürecinin başlamasını beklemek yerine bir servisin gerçek `healthcheck` sonucunu bekleyerek kapatır:

```
db:
  image: postgres:16
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres"]
    interval: 5s
    timeout: 3s
    retries: 5

app:
  build: .
  depends_on:
    db:
      condition: service_healthy
```

`pg_isready`, resmi `postgres` image'ının içinde zaten kurulu, özellikle "PostgreSQL gerçekten bağlantı kabul etmeye hazır mı?" sorusuna cevap vermek için inşa edilmiş gerçek bir araçtır -- bu ders için yazılmış bir script değil. `condition: service_healthy`, Compose'a `app`'i, `db`'nin `healthcheck`'i başarı bildirene kadar başlatmamasını söyler -- "Docker Compose"un bilinçli olarak açık bıraktığı tam boşluğu kapatarak.

Production İçin Ortam Değişkeni Tabanlı Konfigürasyon

Bu projenin kendi `application-prod.yml`'i, gerçekten hassas her değeri -- `DB_URL`, `DB_USERNAME`, `DB_PASSWORD` -- sabit kodlamak yerine zaten ortam değişkenlerinden okuyor (`${DB_URL}` ve gerisi), kendi yorumu akıl yürütmeyi doğrudan belirtiyor: *"Üretimde tüm gizli bilgiler ortam değişkenlerinden okunur, repoya asla yazılmaz."* Bu projenin Dockerfile'ından inşa edilen bir container, aynı sıradan Spring Boot mekanizmasıyla o aynı profille etkinleştirilir:

```
docker run -e SPRING_PROFILES_ACTIVE=prod -e DB_URL=... -e DB_USERNAME=... -e
DB_PASSWORD=... learning-platform:1.0
```

Burada Docker'a özgü hiçbir şey yok -- `-e`, `application-prod.yml`'in zaten var olmasını beklediği ortam değişkenlerini bir container'ın almasının yalnızca bir yolu, "Bir Spring Boot Uygulamasını Docker'a Taşımak"ın `SPRING_DATASOURCE_URL` için zaten kullandığı aynı gevşek-bağlama mekanizması.

Temel Image Güvenlik Değerlendirmeleri

Bu kursta daha önce zaten kurulmuş birkaçı da olan bir avuç alışkanlık, anlamlı ölçüde daha güvenli image'lara toplanıyor:

- **Kesin tag'leri sabitle** (`postgres:16`, `eclipse-temurin:21-jre`, asla `:latest` değil) -- "Docker CLI Temelleri"ndeki "Best Practices" nedenini zaten işledi; bu, production için de aynı ölçüde geçerli.
- **Bir Dockerfile'da `ENV` ya da sabit kodlanmış bir değerle gerçek bir sırrı hiçbir zaman image'a pişirme** -- o şekilde ayarlanan her şey, image'ın kendi katmanlarının bir parçası olur, onu çekebilen ya da inceleyebilen herkes tarafından, o değer daha sonra "değiştirilse" bile, kalıcı olarak okunabilir. Sırlar, `docker run` / `docker compose up` zamanında sağlanan ortam değişkenlerine aittir (bkz. "Production İçin Ortam Değişkeni Tabanlı Konfigürasyon"), hiçbir zaman image'ın kendisine değil.
- **Base image'ı, ve üzerine kurulanı minimal tut** -- `RUN apt-get install`ın kurduğu her paket (`HEALTHCHECK` için yukarıdaki `curl`), alışkanlıkla değil, gerekçelendirilmesi gereken bir şeydir.
- **Root olmayan bir kullanıcı olarak çalış** (yukarıya bkz.) -- ele geçirilmiş bir sürecin container içinde yapabileceklerinde küçük, gerçek bir azalma.

Hepsini Bir Araya Getirmek

Bu proje için tam, production-odaklı Dockerfile ve Compose dosyası -- katmanlı önbelleğe alınmış bağımlılık indirmeleri, root olmayan bir kullanıcı, bir `HEALTHCHECK`, ve `depends_on`'un gerçekten beklemesini sağlayan bir Compose `healthcheck` koşulu:

```
# Stage 1: build the JAR, with dependency downloads cached in their own layer.
FROM maven:3.9-eclipse-temurin-21 AS builder

WORKDIR /build

COPY pom.xml .
RUN mvn -B dependency:go-offline

COPY src ./src
RUN mvn -B package -DskipTests

# Stage 2: a small, non-root, health-checked runtime image.
FROM eclipse-temurin:21-jre

RUN apt-get update \
    && apt-get install -y --no-install-recommends curl \
    && rm -rf /var/lib/apt/lists/*

RUN groupadd -r appuser && useradd -r -g appuser appuser

WORKDIR /app

COPY --from=builder --chown=appuser:appuser /build/target/learning-platform-0.1.0-SNAPSHOT.
jar app.jar

USER appuser

EXPOSE 8080

HEALTHCHECK --interval=30s --timeout=3s --start-period=10s \
  CMD curl -f http://localhost:8080/en || exit 1

ENTRYPOINT ["java", "-XX:MaxRAMPercentage=75.0", "-jar", "app.jar"]
```

```
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - db-data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 3s
      retries: 5

  app:
    build: .
    depends_on:
      db:
        condition: service_healthy
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://db:5432/postgres
      SPRING_DATASOURCE_PASSWORD: secret
    ports:
      - "8080:8080"

volumes:
  db-data:
```

Burada bu kursun zaten işlediği mekanizmalardan hiçbirinin yerine geçen bir şey yok -- "Bir Spring Boot Uygulamasını Docker'a Taşımak"tan aynı multi-stage build, "Docker Compose" tan aynı docker-compose.yml şekli, üzerine bu dersin tanıttığı bir avuç production'a özel ekleme katmanlanmış, her biri önceki bir dersin bilinçli olarak açık bıraktığı gerçek bir boşluğu ele alıyor.

Yaygın Hatalar

- Bir container'ı root olarak çalışır bırakmak çünkü "zaten çalışıyor" -- çalışır, ama bu bir Spring Boot sürecinin ihtiyaç duyduğundan daha fazla ayrıcalıktır, ve "Root Olmayan Bir Kullanıcı Olarak Çalışmak"ın tüm meselesi bundan kaçınmanın neredeyse hiçbir maliyeti olmamasıdır.
- Bir Dockerfile'a `ENV` ile gerçek bir şifre ya da API anahtarı yazıp "sonra kaldırılabilir" varsaymak -- kaldırılamaz; inşa edildiği anda o image'ın katmanlarına kalıcı olarak pişirilir (bkz. "Temel Image Güvenlik Değerlendirmeleri").
- Bir Dockerfile'a bir `HEALTHCHECK` eklemek ama onu Compose'un `depends_on: condition: service_healthy`'sinde hiç kullanmamak -- kontrol çalışır, ama gerçekte hiçbir şey onu beklemez, ki bu sessizce tüm meseleyi geçersiz kılar.
- Multi-stage bir Dockerfile'ı, `COPY src ./src`'in bağımlılık-indirme adımından önce olacağı şekilde yeniden sıralamak -- bu, "Daha Hızlı Rebuild'ler İçin Talimatları Sıralamak (Layer Caching)"ın tarif ettiği layer caching faydasını, hiçbir görünür hata üretmeden sessizce geri alır.

Best Practices

- Dockerfile talimatlarını en-az-değişenden en-çok-değişene sırala -- önce bağımlılık manifestoları ve indirmeleri, en son uygulama kaynak kodu -- böylece sıradan kod değişiklikleri mümkün olan en az sayıda önbelleğe alınmış katmanı geçersiz kılar.
- Her production'a yönelik Dockerfile'a, o kullanıcının okuması gereken dosyalar doğru sahiplikle kopyalandıktan hemen sonra bir `USER` talimatı ekle.
- Her Compose `healthcheck`'i, ona gerçekten bağımlı olan her şeyde bir `depends_on: condition: service_healthy` ile eşleştir -- hiçbir şeyin beklemediği bir health check yalnızca görünürlük sağlar, doğruluk değil.
- Sızdırılması utandırıcı ya da tehlikeli olacak herhangi bir değeri (bir şifre, bir API anahtarı, özel bir URL), Dockerfile'ın kendisinde değil, çalışma zamanında sağlanan bir ortam değişkeninde bulunması gereken bir şey olarak ele al.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bağımlılık manifestolarını kopyalayıp bağımlılıkları uygulama kaynak kodunu kopyalamadan önce indirmek, Docker'ın layer cache'inin sıradan kod değişikliklerinde bağımlılıkları yeniden indirmeyi atlamasını sağlar.
- `COPY --chown` ile eşleştirilmiş bir `USER` talimatı, containerize edilmiş uygulamayı varsayılan olarak root yerine ayrıcalıksız bir kullanıcı olarak çalıştırır.
- Bir Dockerfile `HEALTHCHECK`'i, Docker'ın (ve `docker ps`'in) çalışan bir container'ın yalnızca süreci hâlâ hayatta mı değil, gerçekten çalışıyor mu olduğunu takip etmesini sağlar.
- Compose'un uzun-form `depends_on: condition: service_healthy`'si, bir servisin kendi `healthcheck`'iyle eşleştirildiğinde, tek başına `depends_on`'un açık bıraktığı "container başladı ama henüz hazır değil" boşluğunu gerçekten kapatan şeydir.
- Gerçek sırlar, `docker run` / `docker compose up` zamanında sağlanan ortam değişkenlerine aittir -- bir image'ın kendi katmanları etkin olarak kalıcıdır ve onu çekebilen herkes tarafından okunabilir.

Cheat Sheet

```
COPY pom.xml .
RUN mvn -B dependency:go-offline    # kaynak kod değişikliklerinden ayrı önbelleğe alınır
COPY src ./src
RUN mvn -B package -DskipTests

RUN groupadd -r appuser && useradd -r -g appuser appuser
COPY --from=builder --chown=appuser:appuser /build/target/*.jar app.jar
USER appuser

HEALTHCHECK --interval=30s --timeout=3s --start-period=10s \
  CMD curl -f http://localhost:8080/en || exit 1
```

```
depends_on:
  db:
    condition: service_healthy   # yalnızca başlamasını değil, db'nin healthcheck'ini bekler
```

Terimler Sözlüğü

- **Layer caching**: Docker'ın, o katmanın bağlı olduğu hiçbir şey değişmediği sürece, bir talimatı yeniden çalıştırmak yerine önceden inşa edilmiş bir katmanı yeniden kullanması.
- `USER`: sonraki her talimatı, ve son çalışan süreci, belirli bir (ideal olarak root olmayan) kullanıcıya çeviren bir Dockerfile talimatı.
- `HEALTHCHECK`: Docker'ın bir container'ın gerçekten çalışıp çalışmadığını -- yalnızca çalışıyor olup olmadığını değil -- belirlemek için periyodik olarak container içinde çalıştırdığı bir komutu tanımlayan bir Dockerfile talimatı.
- `condition: service_healthy`: yalnızca container'ının başlamasını beklemek yerine, bir bağımlılığın `healthcheck`'inin başarılı olmasını bekleyen bir Compose `depends_on` ayarı.