

# Bir Spring Boot Uygulamasını Docker'a Taşımak

Bir Spring Boot JAR'ı için JRE vs JDK base image seçimi, .dockerignore, multi-stage build'ler, ve container-farkında JVM bellek boyutlandırması -- bu projenin kendi gerçek learning-platform uygulaması üzerinden.

Orta 35 dk

Oluşturulma tarihi: 5 Ekim 2026

Çevrimiçi oku: <https://www.learnforgex.com/tr/topics/dockerizing-a-spring-boot-application>

"Docker İmajları ve Dockerfile", Java devreye girmeden önce bilinçli olarak bir Dockerfile inşa etti ve çalıştırdı -- FROM , WORKDIR , COPY , RUN , EXPOSE , CMD / ENTRYPOINT , ve docker build artık zaten tanıdık. Buradaki hiçbir talimat değişmiyor -- bu ders, tam olarak aynı seti bu projenin kendi learning-platform Spring Boot uygulamasına (kendi pom.xml 'ine göre com.cdurgun:learning-platform:0.1.0-SNAPSHOT ) uyguluyor, ve özellikle gerçek bir Java uygulaması için önemli olan üç şeyi ekliyor: doğru base image'ı seçmek, .dockerignore , ve multi-stage build'ler.

## Bir JAR'dan Bir Container Image'a

mvn package , Docker hiç devreye girmeden önce zaten tek, kendi kendine yeten, çalıştırılabilir bir artifact üretir -- target/learning-platform-0.1.0-SNAPSHOT.jar . O JAR, bu projenin ihtiyaç duyduğu her bağımlılığı zaten paketler (Spring Boot'un kendi yeniden paketlemesi, spring-boot-starter-parent aracılığıyla, onu en baştan kendi kendine yeterli kılan şeydir); burada bu konuda hiçbir şey değişmiyor. Bir Dockerfile'ın eklediği şey, o zaten inşa edilmiş JAR'ı uyumlu bir Java runtime'ıyla birlikte tek bir image'a paketlemenin bir yolu, böylece onu çalıştırmak, belirli bir makinede o anda hangi JDK'nın kurulu olduğuna artık bağlı olmuyor.

## Bir Java Base Image Seçmek

Bir Java base image'ı, ayırt edilmeye değer iki şekilde gelir: derleyiciyi ve Java kodunu inşa etmek için gereken her şeyi içeren tam bir JDK image'ı, ve zaten derlenmiş bir JAR'ı çalıştırmak için yeterli olanı içeren yalnızca-JRE bir image'ı. target/learning-platform-0.1.0-SNAPSHOT.jar , Dockerfile hiç çalışmadan önce zaten tamamen inşa edilmiş olduğundan, son image'ın onu yalnızca çalıştırması yeterli -- bir JRE image'ı yeterlidir, ve tam olarak bu nedenle bir JDK image'ından bilinçli olarak daha küçüktür.

```
FROM eclipse-temurin:21-jre
```

`eclipse-temurin`, Eclipse Foundation'ın OpenJDK'nın kendi dağıtımıdır, resmi, versiyonlanmış Docker image'ları olarak yayınlanır -- `21-jre`, bu projenin `pom.xml` 'inden kendi `<java.version>21</java.version>` 'iyle eşleşiyor ("Docker İmajları ve Dockerfile"daki "Best Practices" e bkz., küçük, amaca özel bir base image'ı tercih etmek üzerine -- aynı akıl yürütme burada da geçerli, JAR zaten inşa edildikten sonra JDK yerine JRE).

## Bir Spring Boot JAR'ı İçin İlk Dockerfile

Base image'ı "Docker İmajları ve Dockerfile"ın zaten işlediği talimatlarla bir araya getirmek, tam, çalışan bir Dockerfile üretir:

```
FROM eclipse-temurin:21-jre

WORKDIR /app

COPY target/learning-platform-0.1.0-SNAPSHOT.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "app.jar"]
```

`COPY target/learning-platform-0.1.0-SNAPSHOT.jar app.jar`, zaten inşa edilmiş JAR'ı sabit, basit bir isim altında image'a getirir, ve `ENTRYPOINT ["java", "-jar", "app.jar"]`, "CMD vs ENTRYPOINT"ın önerdiği tam olarak sabit-komut desenidir -- bu container'ın tek işi her zaman bu tek JAR'ı çalıştırmaktır. Bu çalışır, ama adlandırılmaya değer gerçek bir ön koşulu var: `mvn package` 'in *host makinede*, `docker build` 'den önce çalıştırılmış olması gerekiyor, böylece JAR `COPY` 'nin bulması için zaten var olsun. Bu dersteki ilerideki "Multi-Stage Build'ler", bu ön koşulu tamamen kaldırıyor.

## .dockerignore — Build Context'e Ne Gönderilmemeli

"Bir Image İnşa Etmek: `docker build`", `docker build` 'in sondaki `.` 'inin **build context** olduğunu -- Docker'ın `Dockerfile` 'ı ve her `COPY` 'nin referans verdiği dosyayı okuduğu dizin -- zaten kurmuştu. Onu sınırlayan hiçbir şey olmadan, o context proje klasöründeki her şeyi içerir -- `.git` 'in tam geçmişi, IDE konfigürasyonu, dokümantasyon -- inşa edilen image'ın gerçekte hiçbirine ihtiyacı yok. `Dockerfile` 'in hemen yanında oturan bir `.dockerignore` dosyası, tam olarak bir `.gitignore` 'un bir commit'ten dosyaları dışladığı gibi dışlar:

```
.git/  
.idea/  
*.iml  
.mvn/  
mvnw  
mvnw.cmd  
*.md  
docs/
```

target/ 'in bilinçli olarak henüz bu listede **olmadığına** dikkat et -- yukarıdaki tek-aşamalı Dockerfile, COPY 'nin bulması için target/learning-platform-0.1.0-SNAPSHOT.jar 'in build context'te var olmasına hâlâ ihtiyaç duyuyor. "Multi-Stage Build'ler" bunu değiştiriyor, ve değiştirdiğinde, target/ de .dockerignore 'a ait oluyor.

Daha küçük bir build context yalnızca disk kullanımıyla ilgili değil -- docker build her çalıştığında tüm context diskten (ve bazı Docker kurulumlarında, uzak bir daemon'a bir socket üzerinden) okunur, bu yüzden .git 'i ve diğer ilgisiz dizinleri dışlayan bir .dockerignore , her build'i yalnızca daha derli toplu değil, anlamlı ölçüde daha hızlı da tutar.

## Multi-Stage Build'ler

Bir **multi-stage build**, tek bir Dockerfile 'da birden fazla FROM kullanır, her FROM yeni bir aşama (stage) başlatır, ve sonraki bir aşama önceki bir aşamadan belirli dosyaları seçici olarak kopyalayabilir -- o önceki aşamanın ürettiği ya da içerdiği her şeyin geri kalanını atarak.

```
# Stage 1: build the JAR with Maven -- this stage's contents never reach the final image.  
FROM maven:3.9-eclipse-temurin-21 AS builder  
  
WORKDIR /build  
  
COPY pom.xml .  
COPY src ./src  
  
RUN mvn -B package -DskipTests  
  
# Stage 2: run it with just a JRE -- no Maven, no JDK, no source code.  
FROM eclipse-temurin:21-jre  
  
WORKDIR /app  
  
COPY --from=builder /build/target/learning-platform-0.1.0-SNAPSHOT.jar app.jar  
  
EXPOSE 8080  
  
ENTRYPOINT ["java", "-jar", "app.jar"]
```

FROM ... AS builder ile builder adı verilen ilk aşama, mvn package 'i container'ın kendisinin içinde çalıştırmak için bir Maven+JDK image'ı kullanır -- bu projenin pom.xml 'i ve src/ 'i içeri

girer, bir JAR dışarı çıkar, tamamen Docker'ın inşa sürecinin içinde, host makinede zaten kurulu bir Maven ya da JDK'ya hiç bağımlı olmadan. İkinci aşama, öncekiyle tam olarak aynı küçük `eclipse-temurin:21-jre` base'inden tamamen yeniden başlar, ve `COPY --from=builder /build/target/learning-platform-0.1.0-SNAPSHOT.jar app.jar`, ilk aşamaya geri uzanıp yalnızca gerçekten ihtiyaç duyduğu tek dosyayı alır. Maven, JDK, `pom.xml`, ve tam `src/` ağacı -- `builder` aşamasının içerdiği her şey -- son image'ın hiç parçası olmaz.

`COPY --from=builder` yalnızca önceki aşamaya bir isim verildiği için çalışır ( `AS builder` ) -- olmadan, sonraki bir aşamanın referans verecek hiçbir şeyi olmaz. İleride gerçekten yeni bir sebep için ikinci bir `FROM` eklemek ve `AS <isim>` 'i unutmak kolaydır, o noktada sonraki bir `COPY` 'deki `--from=builder` sessizce bozulur.

## JVM ve Container Kaynak Sınırları

Java 10'dan beri, JVM varsayılan olarak **container-farkındadır** -- `docker run -m` ile ("Docker Compose"ta, bir servisin kendi bellek sınırıyla) ayarlanmış bir bellek sınırı olan bir container içinde çalışırken, eski JVM'lerin yaptığı gibi host makinenin tam belleğini görmek yerine, doğrudan o cgroup tarafından dayatılan sınırı okur. Varsayılan olarak, JVM heap'ini o sınırın bir yüzdesi olarak boyutlandırır ( `-XX:MaxRAMPercentage`, varsayılan olarak 25.0) -- JVM'in kendisinin ihtiyaç duyduğu thread stack'leri, metaspace, ve diğer heap-dışı bellek için yer bırakarak.

```
docker run -m 512m learning-platform:0.1.0
```

512MB'lık bir sınırın %25'i, gerçek bir Spring Boot uygulaması için oldukça küçük bir heap'tir -- *tek işi* bu tek JVM'i çalıştırmak olan bir container için (bu kurstaki her örnekte doğru), bu yüzdeyi açıkça yükseltmek yaygındır:

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

Bu, JVM'in kendisini yine de container'a gerçekte verilen herhangi bir bellek sınırına göre boyutlandırmasına izin verir -- `-m` üzerinden, ya da bu kursta ileride "Docker Compose"un kendi bellek ayarları üzerinden -- container'ın bellek sınırı her değiştiğinde elle güncellenmesi gereken sabit bir `-Xmx` değeri sabit kodlamak yerine.

## Hepsini Bir Araya Getirmek

Yukarıdaki multi-stage Dockerfile'ı ve bu projenin "Docker CLI Temelleri"nden kendi PostgreSQL container'ını kullanan tam inşa-çalıştır döngüsü:

```
#!/bin/sh
# Building this project's own image with the multi-stage Dockerfile -- no
# local Maven or JDK install required, Docker builds the JAR internally --
# and running it against the PostgreSQL container from "Docker CLI Fundamentals".

docker build -t learning-platform:0.1.0 .
# Output (trimmed):
# [+] Building 42.7s (14/14) FINISHED
# => [builder 1/4] FROM docker.io/library/maven:3.9-eclipse-temurin-21
# => [builder 4/4] RUN mvn -B package -DskipTests
# => [stage-1 3/3] COPY --from=builder /build/target/learning-platform-0.1.0-SNAPSHOT.jar
app.jar
# => naming to docker.io/library/learning-platform:0.1.0

docker run -d --name learning-platform-app \
  -p 8080:8080 \
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://host.docker.internal:5432/postgres \
  -e SPRING_DATASOURCE_PASSWORD=secret \
  learning-platform:0.1.0
# Output:
# 9b8c7d6e5f4a3b2c1d0e9f8a7b6c5d4e3f2a1b0c9d8e7f6a5b4c3d2e1f0a9b8c

docker logs -f learning-platform-app
# Output (trimmed):
# Started LearningPlatformApplication in 3.1 seconds

curl http://localhost:8080/en
# Output (trimmed):
# <!doctype html>
# <html> ... the actual homepage this project renders ...

docker stop learning-platform-app
docker rm learning-platform-app
```

`SPRING_DATASOURCE_URL` ve `SPRING_DATASOURCE_PASSWORD`, sıradan Spring Boot ortam değişkeni geçersiz kılmalarıdır -- bu projede `application.yml` / `application-prod.yml` 'in zaten dayandığı aynı gevşek-bağlama (relaxed binding) mekanizması, yalnızca bir properties dosyası yerine `-e` üzerinden sağlanıyor. `host.docker.internal`, bir container'ın host makinenin kendi ağında çalışan bir servise ulaşmasını sağlar -- container'ların birbirine ağ üzerinden gerçekte nasıl ulaştığının tam hikayesi, `host.docker.internal` 'e daha temiz bir alternatif dahil, bu kurstaki bir sonraki kategorinin ilk dersi "Docker Networking".

## Yaygın Hatalar

- Henüz var olmayan bir JAR'ı kopyalamaya çalışmak -- bu dersteki ilk Dockerfile gibi tek-aşamalı bir Dockerfile, `mvn package` 'in host'ta zaten çalıştırılmış olmasını gerektirir; bu adımı unutmak sessiz bir no-op değil, bir `COPY` hatası üretir.
- Son, çalışan container için bir JRE image'ı yeterliyken tam bir JDK base image kullanmak -- çalışan uygulamanın hiç kullanmadığı bir derleyici ve inşa araçları için ekstra boyut ve saldırı yüzeyi.

- `.dockerignore` 'u atlayıp `.git` 'in tam geçmişini (ve proje klasöründeki her şeyi) her tek build'de build context'e göndermek.
- Bir inşa aşamasını `AS builder` olarak adlandırıp sonra sonraki bir `COPY --from=...` 'de başka bir şey olarak referans vermek (ya da ismi tamamen unutmak) -- isim tam olarak eşleşmeli.
- JVM'in varsayılan container-farkında boyutlandırmasına (ya da açık bir `-XX:MaxRAMPercentage` 'a) güvenmek yerine sabit bir `-Xmx` değeri sabit kodlamak -- sabit bir değer, container'ın kendi bellek sınırı değiştiği anda sessizce gerçekte eşleşmemeye başlar.

## Best Practices

- Bir multi-stage build'in son aşaması için bir JRE base image'ı tercih et -- JAR'ı üreten JDK, Maven, ve kaynak kod, onu yalnızca çalıştıran image'da var olmak için hiçbir sebep taşımaz.
- Bir Dockerfile'ı her zaman bir `.dockerignore` ile eşleştir -- daha küçük build context, daha hızlı build'ler, ve `.git` geçmişini ya da yerel IDE konfigürasyonunu bir image'a kazara sızdırma riski yok.
- Yalnızca bu proje için değil, gerçek bir uygulama image'ı için varsayılan olarak bir multi-stage build'e başvuru -- "bu image'ı inşa etmek için Maven'in/bir JDK'nın zaten kurulu olması gerekiyor" gereksinimini tamamen kaldırır, ki bu image'ı bir geliştiricinin kendi makinesi olmayan bir makinede inşa eden herkes için önemlidir.
- JVM'in yerleşik container-farkındalığının varsayılan olarak boyutlandırmayı yapmasına izin ver, ve yalnızca container'ın tek amacı bu tek JVM'i çalıştırmak olduğunda ve varsayılan % 25 açıkça fazla tutucu olduğunda açık bir `-XX:MaxRAMPercentage` 'a başvuru.

## Özet, Cheat Sheet ve Terimler Sözlüğü

### Özet

- "Docker İmajları ve Dockerfile"daki aynı Dockerfile talimatları, gerçek bir Spring Boot JAR'ına değişmeden uygulanır -- yalnızca base image ve birkaç ayrıntı farklıdır.
- Bir JRE base image'ı ( `eclipse-temurin:21-jre` ), son, çalışan container için yeterlidir -- tam bir JDK yalnızca JAR'ı *inşa etmek* için gereklidir, onu çalıştırmak için değil.
- `.dockerignore` , tam olarak `.gitignore` 'un onları bir commit'ten dışladığı gibi dosyaları build context'ten dışlar -- daha küçük context, daha hızlı build'ler, kazara sızıntı yok.
- Bir multi-stage build, tek bir Dockerfile'da birden fazla `FROM` talimatı kullanır, JAR'ı önceki, adlandırılmış bir aşamada Maven+JDK ile inşa eder, ve `COPY --from=<aşama>` yalnızca bitmiş JAR'ı küçük, son bir JRE-tabanlı aşamaya kopyalar.
- JVM, Java 10'dan beri container-farkındadır -- container'ın gerçek bellek sınırını okur ve heap'ini onun bir yüzdesi olarak otomatik olarak boyutlandırır ( `-XX:MaxRAMPercentage` , varsayılan olarak 25.0), varsayılan olarak elle `-Xmx` ayarlaması gerekmez.

## Cheat Sheet

```
FROM maven:3.9-eclipse-temurin-21 AS builder    # inşa aşaması: Maven + JDK var
WORKDIR /build
COPY pom.xml .
COPY src ./src
RUN mvn -B package -DskipTests
```

```
FROM eclipse-temurin:21-jre                    # son aşama: yalnızca JRE
WORKDIR /app
COPY --from=builder /build/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

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

## Terimler Sözlüğü

- **JRE image:** zaten derlenmiş Java bytecode'unu *çalıştırmak* için gerekli yalnızca şeyi içeren bir base image -- derleyici yok.
- **JDK image:** kaynak koddan Java kodu *inşa etmek* için gereken Java derleyicisini ve tam araç zincirini ek olarak içeren bir base image.
- **.dockerignore :** Dockerfile 'ın yanında oturan, .gitignore 'un onları bir commit'ten dışladığı gibi yolları build context'ten dışlayan bir dosya.
- **Multi-stage build:** birden fazla FROM içeren bir Dockerfile , ki sonraki bir aşama önceki bir aşamanın içerdiği her şeyin geri kalanını atarken belirli dosyaları COPY --from=<önceki-aşama> ile kopyalayabilir.
- **Container-farkında JVM:** JVM'in (Java 10'dan beri) cgroup'lar üzerinden container'ın kendi bellek sınırını okuyup heap'ini host makinenin toplam belleği yerine onun bir yüzdesi olarak boyutlandırılan varsayılan davranışı.