Pratik Proje: Docker ile Spring Boot + PostgreSQL

Bu kursun her dersinin işlediği her parçayı -- multi-stage Dockerfile, non-root kullanıcı, HEALTHCHECK, Docker Compose, named volume, otomatik networking -- küçük, tam bir Spring Boot + PostgreSQL task tracker uygulamasında bir araya getiren, kursun kapanış dersi.

İleri 45 dk

Son güncelleme: 31 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/docker-practical-project

---

Bu kurstaki her ders bir parça inşa etti: image'lar ve container'lar, CLI, Dockerfile'lar, gerçek bir Spring Boot JAR'ını dockerize etmek, networking, volume'lar, Compose, ve production sertleştirmesi. Bu son ders hiçbir yeni şey öğretmiyor -- tek seferde kafanda tutabileceğin kadar küçük, tam, bağımsız bir Spring Boot + PostgreSQL uygulaması, o parçaların hepsini baştan sona bir arada çalışır hale getiriyor. Önceki derslerin bu platformun kendi, çok daha büyük `learning-platform` uygulamasını containerize eden Dockerfile'larının ve Compose dosyalarının aksine, bu bir görev takipçisi (task tracker) -- bilinçli olarak minimal, bu kursun daha önce hiç işlemediği hiçbir şeyi kullanmayan, gerçekten inşa edilip çalıştırılması amaçlanan bir proje.

Ne İnşa Ediyoruz

Bir **task tracker**: PostgreSQL destekli bir REST API, tam olarak iki işlemle -- her task'ı listelemek, ve yeni bir tane oluşturmak. Kimlik doğrulama, sayfalama, ya da task düzenleme hakkında hiçbir şey dahil değil; bu platformun kendi örnek yazım ilkesine göre, böyle bir egzersiz "kavramı anlaşılır kılacak en az kod"u kullanmalı, ve buradaki kavram containerization, uygulama tasarımı değil.

```
GET  /tasks   -> her task'ı listele
POST /tasks   -> yeni bir task oluştur
```

Uygulamanın Kendisi

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

// A complete, minimal Spring Boot + PostgreSQL application -- everything
// this lesson's Dockerfile and docker-compose.yml actually run. In a real
// project, Task/TaskRepository/TaskController would each live in their own
// file; they're combined here into one compilable teaching example, the
// same convention this course's other multi-class examples already use.
@SpringBootApplication
public class TaskTrackerApplication {

    public static void main(String[] args) {
        SpringApplication.run(TaskTrackerApplication.class, args);
    }

    @Entity
    static class Task {
        @Id
        @GeneratedValue
        private Long id;
        private String title;
        private boolean done;

        public Long getId() {
            return id;
        }

        public String getTitle() {
            return title;
        }

        public void setTitle(String title) {
            this.title = title;
        }

        public boolean isDone() {
            return done;
        }

        public void setDone(boolean done) {
            this.done = done;
        }
    }

    interface TaskRepository extends JpaRepository<Task, Long> {
    }

    @RestController
    static class TaskController {

        private final TaskRepository taskRepository;
```

```
        TaskController(TaskRepository taskRepository) {
            this.taskRepository = taskRepository;
        }

        @GetMapping("/tasks")
        List<Task> listTasks() {
            return taskRepository.findAll();
        }

        @PostMapping("/tasks")
        Task createTask(@RequestBody Task task) {
            return taskRepository.save(task);
        }
    }
}
```

Gerçek, çok-dosyalı bir projede, `Task`, `TaskRepository`, ve `TaskController`, sıradan Spring Boot konvansiyonunu takip ederek her biri kendi dosyasında yaşardı -- burada tek, kendi kendine yeten bir öğretim örneği olarak tek dosyada birleştirildiler, bu kursun diğer çok-sınıflı örneklerinin zaten takip ettiği aynı konvansiyon. Açıkça adlandırılmaya değer üç şey var: `@Entity`, `Task`'ı, Spring Data JPA kursundaki "Entities and the Repository Abstraction"ın zaten işlediği tam olarak aynı şekilde bir veritabanı tablosuna eşler; `TaskRepository extends JpaRepository<Task, Long>`, hiçbir implementasyon yazılmadan `findAll()` ve `save()`'i bedava alır; ve aşağıdaki Compose dosyasında ayarlanan `SPRING_JPA_HIBERNATE_DDL_AUTO=update`, bilinçli olarak yalnızca-demo bir kısayoldur -- bu platformun kendi gerçek uygulaması, bunun yerine `ddl-auto: validate` ile Flyway migration'ları kullanır (Spring Data JPA kursundaki "JPA, Hibernate ve Spring Data JPA"ya bkz.), ki gerçek bir projenin başvurması gereken şey budur.

Dockerfile'ı Yazmak

Buradaki her şey "Bir Spring Boot Uygulamasını Docker'a Taşımak" ve "Production İçin Docker (Java Uygulamaları)"ın doğrudan bir uygulaması -- hiçbir yeni şey yok, yalnızca bir araya getirilmiş:

```
# 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/task-tracker-0.1.0.jar app.jar

USER appuser

EXPOSE 8080

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

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

Kendi katmanında önbelleğe alınmış bağımlılık indirmeleriyle ("Daha Hızlı Rebuild'ler İçin Talimatları Sıralamak (Layer Caching)") bir multi-stage build ("Multi-Stage Build'ler"), root olmayan bir `appuser` ("Root Olmayan Bir Kullanıcı Olarak Çalışmak"), ve bu uygulamanın kendi gerçek `/tasks` endpoint'ine karşı bir `HEALTHCHECK` ("Health Check'ler: Bir Dockerfile'da `HEALTHCHECK`") -- son iki dersin boyunca kullanılan aynı desen, `learning-platform`'un kendisi yerine bu küçük uygulamaya yöneltilmiş.

`docker-compose.yml`'i Yazmak

```
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - task-tracker-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
      SPRING_JPA_HIBERNATE_DDL_AUTO: update
    ports:
      - "8080:8080"

volumes:
  task-tracker-db-data:
```

`db` ve `app`, "Docker Compose"un zaten inşa ettiği aynı iki-servis şekli -- PostgreSQL'in verisi için bir named volume ("Docker Volumes"), `app`'in hiçbir manuel `docker network create` olmadan PostgreSQL'e `db:5432`'de ulaşmasını sağlayan otomatik isim-tabanlı networking ("İsimle Container'dan Container'a İletişim"), ve `app`'in `db`'nin yalnızca başlamasını değil gerçekten hazır olmasını beklemesini sağlayan `depends_on: condition: service_healthy` ("`depends_on`'un Gerçekten Beklemesini Sağlamak: Compose Sağlık Koşulları").

Baştan Sona Çalıştırmak

```
docker compose up -d
docker compose ps
```

`docker compose ps` her iki servisin de `healthy` olduğunu gösterdikten sonra, API, bu kurs boyunca `curl`'ün bu platformun kendi anasayfasına ulaştığı tam olarak aynı şekilde erişilebilir:

```
curl -X POST http://localhost:8080/tasks -H "Content-Type: application/json" -d '{"title":
"Finish the Docker course", "done": false}'
curl http://localhost:8080/tasks
```

Oluşturulan bir task, `GET /tasks`'tan hemen geri döner -- `app`'in bu Compose dosyasının otomatik olarak kurduğu ağ üzerinden `db`'ye gerçekten ulaştığının doğrulaması.

Veri Kalıcılığını Doğrulamak

Bu kurulumun doğru olduğunun gerçek kanıtı bir kez çalışması değil -- verinin, "Docker Volumes" ve "Docker Compose"un söylediği olaylardan tam olarak sağ çıkması, ve bilinçli olarak sağ çıkmaması gereken tek olaydan sağ çıkmaması:

```
#!/bin/sh
# Bringing the whole thing up, creating real data through the REST API,
# confirming it's actually stored in PostgreSQL, and confirming it survives
# a restart of just the app container.

docker compose up -d
# Output (trimmed):
# => Container task-tracker-db-1    Started
# => Container task-tracker-db-1    Healthy
# => Container task-tracker-app-1   Started

docker compose ps
# Output:
# NAME                    IMAGE               STATUS                    PORTS
# task-tracker-app-1      task-tracker-app    Up 10 seconds (healthy)   0.0.0.0:8080->8080
/tcp
# task-tracker-db-1       postgres:16         Up 15 seconds (healthy)

curl -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{"title": "Finish the Docker course", "done": false}'
# Output:
# {"id":1,"title":"Finish the Docker course","done":false}

curl http://localhost:8080/tasks
# Output:
# [{"id":1,"title":"Finish the Docker course","done":false}]

# Restart just the app container -- the database keeps running, untouched.
docker compose restart app

curl http://localhost:8080/tasks
# Output:
# [{"id":1,"title":"Finish the Docker course","done":false}]

# The task survived the app restart, because it was never stored in the app
# container to begin with -- it lives in PostgreSQL, in the named volume.

# The real test, per "Docker Volumes": tear down BOTH containers entirely,
# not just restart one, and confirm the data is still there afterward.
docker compose down
# Output (trimmed):
# => Container task-tracker-app-1   Removed
# => Container task-tracker-db-1    Removed
# => Network task-tracker_default   Removed

docker compose up -d
# Output (trimmed):
# => Container task-tracker-db-1    Started
# => Container task-tracker-app-1   Started

curl http://localhost:8080/tasks
# Output:
# [{"id":1,"title":"Finish the Docker course","done":false}]

# Still there -- docker compose down (without -v) never touched the named
# volume, exactly as "Docker Compose" described.
```

`docker compose down`'dan önce oluşturulan bir task, `docker compose up -d` her şeyi geri getirdikten sonra hâlâ orada -- çünkü `docker compose down`, `-v` olmadan, oluşturduğu named volume'a hiç dokunmaz, tam olarak "Docker Compose"un tarif ettiği gibi. Bunun yerine herhangi bir noktada `docker compose down -v` çalıştırmak, bu kurulumdaki verileri gerçekten silen tek komut olurdu -- farkı kendin görmek için bir kez, bilinçli olarak denemeye değer.

Bu Proje Neyi Gösteriyor

Buradaki her dosya tamamını okuyabilecek kadar küçük, ve içindeki her satır belirli bir önceki derse geri izlenebilir:

```
TaskTrackerApplication.java  -> gerçek bir Spring Data JPA entity + repository + controller
TaskTrackerDockerfile        -> Bir Spring Boot Uygulamasını Docker'a Taşımak, Production
İçin Docker
TaskTrackerCompose.yml       -> Docker Networking, Docker Volumes, Docker Compose
```

Daha büyük, gerçek dünyadaki bir Spring Boot uygulamasıyla ilgili hiçbir şey bu resmi türden değiştirmiyor -- daha büyük bir `pom.xml`, daha fazla entity, daha fazla endpoint, hatta bu platformun kendi `learning-platform`'unun kendisi bile, hepsi tam olarak aynı şekille containerize edilir: bir multi-stage Dockerfile, veritabanı için bir named volume, ve hepsini bir araya bağlayan bir `docker-compose.yml`.

Yaygın Hatalar

- `SPRING_JPA_HIBERNATE_DDL_AUTO=update`'i gerçek, production bir Spring Boot projesinin kullanması gereken bir şey olarak ele almak -- bu yalnızca bu küçük demo için bilinçli bir kısayol; gerçek bir proje bunun yerine Flyway migration'ları ve `ddl-auto: validate` kullanır (bkz. "Uygulamanın Kendisi").
- "Veri Kalıcılığını Doğrulamak"taki `docker compose down` / `docker compose up -d` döngüsünü atlayıp yalnızca `app` servisini yeniden başlatmak -- bu yalnızca app container'ının stateless olduğunu kanıtlar, volume'un kendisinin gerçek bir sökümden sağ çıktığını değil.
- Bu dersin Compose dosyasındaki `POSTGRES_PASSWORD` ve `SPRING_DATASOURCE_PASSWORD`'un illüstratif olduğunu unutmak -- "Docker Compose"un kendi notuyla eşleşiyor, gerçek bir deployment'ın bu tür değerleri ayrı, commit edilmemiş bir `.env` dosyasına ya da uygun bir secret mekanizmasına taşıdığı.

Best Practices

- Bu tam projeyi yalnızca okumak yerine, bir kez elle inşa et ve çalıştır -- buradaki her komut zaten önceki bir derste tek tek çalıştırılıp doğrulandı, ama hepsini birlikte baştan sona çalıştırmak, tam resmi gerçekten pekiştiren şey.

- İleride gerçek bir uygulamayı containerize ederken, varsayılan olarak bu aynı şekle başvur: önbelleğe alınmış bağımlılık indirmeleriyle bir multi-stage Dockerfile ve root olmayan bir kullanıcı, herhangi bir stateful şey için bir named volume, ve gerçek sağlık koşullarıyla bir Compose dosyası -- bu kurs böyle söylediği için değil, her parça yol boyunca işlenen gerçek, belirli bir problemi çözmek için seçildiği için.
- `docker compose down -v`'yi bilinçli, ayrı bir karar olarak ele almaya devam et, asla bir alışkanlık olarak değil -- bu dersin kendi demosunun düz `docker compose down`'ın veriyi dokunulmadan bırakmasına güvenmesi tam olarak bu nedenle.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bu dersin task tracker'ı, bu kursun zaten işlediğinin ötesinde hiçbir şey kullanmadan containerize edilmiş, küçük, tam bir Spring Boot + PostgreSQL uygulamasıdır.
- Dockerfile'ı, "Bir Spring Boot Uygulamasını Docker'a Taşımak" ve "Production İçin Docker (Java Uygulamaları)"ı doğrudan uygular: multi-stage build, önbelleğe alınmış bağımlılık katmanı, root olmayan kullanıcı, `HEALTHCHECK`.
- `docker-compose.yml`'i, "Docker Networking", "Docker Volumes", ve "Docker Compose"u doğrudan uygular: otomatik isim-tabanlı servis keşfi, PostgreSQL'in verisi için bir named volume, ve sağlık-tabanlı bir `depends_on`.
- API üzerinden oluşturulan veri, tam bir `docker compose down` / `docker compose up -d` döngüsünden sağ çıkar -- verinin gerçekte nerede yaşadığının named volume olduğunun, ikisi container değil, kanıtı.
- Aynı şekil -- multi-stage Dockerfile, named volume, sağlık koşullarıyla Compose -- değişmeden çok daha büyük gerçek bir uygulamaya ölçeklenir, bu platformun kendi `learning-platform`'u dahil.

Cheat Sheet

```
docker compose up -d              # her şeyi inşa et ve başlat
docker compose ps                 # her iki servisin de sağlıklı olduğunu doğrula
curl -X POST http://localhost:8080/tasks -H "Content-Type: application/json" -d '{"title":
"...", "done": false}'
curl http://localhost:8080/tasks  # task'ları listele
docker compose down               # container'ları sök; volume sağ kalır
docker compose up -d              # geri getir; veri hâlâ orada
```

Terimler Sözlüğü

- **Task tracker**: bu dersin kendi minimal, tam Spring Boot + PostgreSQL demo uygulaması -- iki endpoint, bir entity, bilinçli olarak daha fazlası yok.
- **Baştan sona doğrulama (end-to-end verification)**: bir sistemin gerçekten çalıştığını, onu gerçekten kullanarak -- gerçek API'si üzerinden gerçek veri oluşturarak, sonra o verinin sağ çıkması gereken belirli olaylardan sağ çıktığını kanıtlayarak -- doğrulamak.