JPA, Hibernate ve Spring Data JPA

Bir Java nesnesini bir veritabanına kalıcı hale getirme sorunundan başlayarak, ORM → JPA (spesifikasyon) → Hibernate (implementasyon) → Spring Data JPA (repository soyutlaması) → Spring Boot (auto-configuration) zihinsel modeli, bu projenin gerçek Topic/TopicRepository sınıflarıyla inşa ediliyor. Spring Data JPA kategorisinin 1.'si.

Orta 25 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/jpa-hibernate-and-spring-data-jpa

---

Bir Java sınıfı tanımlamayı, bir bağımlılığı inject etmeyi, ve onu bir REST endpoint'i üzerinden açığa çıkarmayı zaten biliyorsun. Henüz görmediğin şey, bir Java nesnesinin, her sınıfın her alanı için elle SQL yazmadan, gerçekte bir veritabanına NASIL girip çıktığı. Bu boşluk, bu kategorinin tamamının konusu -- ve tek bir repository interface'ine dokunmadan önce, bu ilk ders, birlikte gördüğün dört ismin altındaki zihinsel modeli inşa ediyor: JPA, Hibernate, Spring Data JPA, ve Spring Boot.

Sorun: Bir Java Nesnesi Bir Veritabanı Satırı Değildir

Mümkün olan en basit soruyla başla: bir Java nesnesini -- diyelim ki bir `id` ve bir `title`'ı olan bir `Topic`'i -- bir ilişkisel veritabanına gerçekte nasıl kalıcı hale getirirsin?

```
public class PlainJavaTopicExample {

    // A plain Java object -- nothing here says anything about a database.
    // The JVM only knows how to keep this alive in memory; it has no idea
    // "topic" should correspond to a table, or that "title" should
    // correspond to a column.
    static class Topic {
        private Long id;
        private String title;

        Topic(Long id, String title) {
            this.id = id;
            this.title = title;
        }
    }

    public static void main(String[] args) {
        Topic topic = new Topic(1L, "Introduction to Generics");

        // Without any help, "saving" this means hand-writing SQL yourself,
        // and hand-writing it again for every other entity in the
        // application:
        //
        //   String sql = "INSERT INTO topic (id, title) VALUES (?, ?)";
        //   preparedStatement.setLong(1, topic.getId());
        //   preparedStatement.setString(2, topic.getTitle());
        //   preparedStatement.executeUpdate();
        //
        // This is exactly the repetitive, error-prone gap ORM exists to close.
        System.out.println(topic.title);
    }
}
```

Bu `Topic` sınıfının hiçbir şeyi, bir `topic` tablosunda bir satır olarak sonlanması gerektiğini bilmiyor. Onu gerçekten kaydetmek için bir `PreparedStatement` yazman, her alanı elle bir sütuna eşlemen, ve uygulamadaki her entity için -- `Category`, `Course` ve geri kalan her şey -- aynı mekanik işi tekrarlaman gerekir. Onu geri okumak tam tersini gerektirir: bir `ResultSet` 'in her sütununu, yine elle, yine her sınıf için, bir alana geri eşlemek. Bunların hiçbiri tam olarak zor değil -- yalnızca tekrarlı, ince şekilde yanlış yapılması kolay, ve tekrar tekrar aynı şekildeki kod. Bu tekrar, sıradaki dört katmanın birer birer çözmek için var olduğu sorundur.

ORM Nedir?

Object-Relational Mapping (ORM), tam olarak bu çeviriyi otomatikleştirme genel fikridir -- bir aracın bir sınıfın yapısını okuyup onu saklamak ve geri getirmek için SQL üretmesine izin vermek, senin her sınıf için bu SQL'i elle yazman yerine. ORM'in kendisi Java'ya özgü bir şey, hatta tek bir araç bile değildir; bu bir kavramdır. Sırada tanıtılan JPA, bu fikrin Java'daki standartlaştırılmış ifadesidir.

JPA Nedir?

JPA (Jakarta Persistence API), Java'da object-relational mapping'in NASIL yapılacağını tarif eden bir interface ve annotation kümesidir (`@Entity` , `@Id` , `EntityManager` ve daha fazlası) -- bunu gerçekte YAPAN kodu sağlamadan, bir SPESİFİKASYONDUR. Bunu net olarak belirtmekte fayda var, çünkü yeni başlayanların sık yanlış anladığı bir soruyu cevaplıyor: JPA tek başına çalıştırabileceğin bir kütüphane DEĞİLDİR -- bir spesifikasyonun kendi başına hiçbir çalışma zamanı davranışı yoktur. Yalnızca bir sözleşme tanımlar; başka bir şeyin bu sözleşmeyi uygulaması gerekir.

Hibernate Nedir?

Hibernate, tam olarak o başka şeydir -- JPA spesifikasyonunun somut bir UYGULAMASI. Kodun `@Entity` kullandığında ya da JPA'nın tanımladığı bir metodu çağırdığında, altında gerçekte annotation'ı okuyan, SQL'i üreten, JDBC driver'ıyla konuşan, ve PostgreSQL'e bir satır koyan Hibernate'tir. Başka JPA implementasyonları da var (EclipseLink gibi), ama Hibernate, bu proje dahil gerçek Spring Boot uygulamalarında açık farkla en yaygın seçimdir.

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;

// The SAME class as PlainJavaTopicExample's Topic, with three annotations
// added. Nothing about the class's own Java behavior changed -- these
// annotations are metadata, read by JPA's implementation (Hibernate) to
// generate the INSERT/UPDATE/SELECT SQL that PlainJavaTopicExample had to
// write by hand.
@Entity
public class MinimalEntityExample {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    // A no-argument constructor is required -- JPA implementations create
    // entity instances via reflection before populating their fields. The
    // full reasoning behind this (and everything else about mapping an
    // entity correctly) is covered in "Entities and the Repository
    // Abstraction," the next lesson in this series.
    protected MinimalEntityExample() {
    }

    public MinimalEntityExample(String title) {
        this.title = title;
    }
}
```

Sınıf, `PlainJavaTopicExample`'ın `Topic`'inden değişmedi -- yalnızca üç annotation eklendi. `@Entity`, JPA'ya (ve altında, Hibernate'e) bu sınıfın bir tabloya eşlendiğini söyler; `@Id`, hangi alanın primary key olduğunu işaretler; `@GeneratedValue`, veritabanının o key'in değerini üretmesi gerektiğini söyler. Önceki örnekteki SQL'in hiçbiri artık yazılmak zorunda değil -- Hibernate onu bu annotation'lardan üretir. Ayrıca `MinimalEntityExample`'ın gerektirdiği argümansız constructor'a dikkat et: Hibernate, entity instance'larını alanlarını doldurmadan önce reflection ile inşa eder, ki bu tam olarak "Record"un, argümansız bir constructor'ı ve mutable alanları olmayan bir `record`'un JPA entity'si olarak hiç KULLANILAMAYACAĞINI belirtmesinin nedenidir.

> Argümansız bir constructor, bir sınıfın geçerli bir JPA entity'si olması için karşılaması gereken birkaç gereksinimden biridir -- bu kategorideki sıradaki ders "Entities and the Repository Abstraction", geri kalanını (`@Table`, `@Enumerated`, `nullable` / `unique` kısıtları ve daha fazlası) tam olarak işliyor.

```
Java nesnesi
     ↓
    JPA
     ↓
 Hibernate
     ↓
    SQL
     ↓
PostgreSQL
```

Spring Data JPA Nedir?

JPA artı Hibernate tek başına orijinal sorunu zaten çözer -- ama onları doğrudan kullanmak hâlâ, her entity için, neredeyse birebir tekrarlanan aynı bir avuç metotla (`save`, `findById`, `findAll`, `delete`) elle bir `EntityManager`-tabanlı sınıf yazmak anlamına gelir. Spring Data JPA, tam olarak bu kalan tekrarı ortadan kaldıran, JPA'nın üzerine inşa edilmiş bir REPOSITORY SOYUTLAMASIDIR: bir interface bildirirsin, ve Spring Data JPA, uygulama başlangıcında senin için çalışan bir implementasyon üretir.

```
Repository
     ↓
Spring Data JPA
     ↓
    JPA
     ↓
 Hibernate
     ↓
PostgreSQL
```

Bu, bu projenin kendi kodunun gerçekte kullandığı katmandır.

```
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

// This is a simplified, teaching-focused look at THIS PROJECT'S OWN
// TopicRepository (see src/main/java/com/cdurgun/learning/repository/
// TopicRepository.java) -- extending JpaRepository<Topic, Long> is the
// entire implementation. No class implements this interface by hand;
// Spring Data JPA generates a working implementation at runtime.
interface TopicRepositoryExample extends JpaRepository<TopicExample, Long> {

    // save(topic), findById(id), findAll(), deleteById(id), and more all
    // come from JpaRepository for free -- none of them are written here.

    // findBySlug(...) is NOT free the same way -- Spring Data JPA reads
    // this method's NAME and derives a "WHERE slug = ?" query from it.
    // Exactly how that derivation works is the subject of "Query Methods
    // and JPQL with @Query," the lesson after next -- for now, notice only
    // that no query, no SQL, no JPQL is written here, and yet this method
    // works.
    Optional<TopicExample> findBySlug(String slug);
}

class TopicExample {
    private Long id;
    private String slug;
}
```

`TopicRepositoryExample` (doğrudan bu projenin gerçek `TopicRepository`'si baz alınarak modellendi), bir interface'ten başka hiçbir şey bildirmiyor, ama `save(...)`, `findById(...)`, `findAll()` ve `deleteById(...)`'nin hepsi çalışıyor -- bunlar `JpaRepository`'den bedavaya gelir. `findBySlug(...)` aynı şekilde miras alınmıyor; Spring Data JPA bu METOT ADINI okur ve arkasındaki sorguyu ondan çıkarır -- tam olarak nasıl olduğu, bundan iki ders sonraki "Query Methods and JPQL with @Query"nin konusu.

> Spring Data JPA, JPA'yı ya da Hibernate'i DEĞİŞTİRMEZ -- ikisinin de üzerine oturur. Spring Data JPA'nın çalıştırdığı her sorgu hâlâ JPA'nın `EntityManager`'ından geçer ve altında hâlâ Hibernate tarafından SQL'e dönüştürülür. Spring Data JPA'nın tüm işi, mekanizmanın kendisini değiştirmek değil, bunun etrafındaki tekrarlı boilerplate'i kaldırmaktır.

Dört Katman Nasıl Bir Araya Gelir

Yukarıdaki iki diyagramı tek bir resimde üst üste koymak, tüm ilişkiyi somutlaştırır:

```
Repository             (bir interface yazarsın)
     ↓
Spring Data JPA         (çalışan bir implementasyon üretir)
     ↓
    JPA                 (spesifikasyon: @Entity, EntityManager, ...)
     ↓
 Hibernate               (implementasyon: JPA çağrılarını SQL'e çevirir)
     ↓
    SQL
     ↓
PostgreSQL
```

Her katman, altındaki katmanın çözümsüz bıraktığı sorunu çözer: JPA, Java'da object-relational mapping'in NASIL GÖRÜNMESİ gerektiğini standartlaştırır; Hibernate bunu gerçekten YAPAR; Spring Data JPA, JPA/Hibernate'i doğrudan kullanmanın tekrarlı boilerplate'ini, bir seferde bir repository interface'i, ortadan kaldırır.

Spring Boot Nereye Oturuyor

Spring Boot'un buradaki rolü göründüğünden daha dar: yukarıdaki diyagrama başka bir katman eklemez bile -- yalnızca var olanları otomatik olarak birbirine bağlar. `spring-boot-starter-data-jpa`'yı classpath'e eklemek, Spring Boot'un bir `DataSource`, bir `EntityManagerFactory`, JPA provider'ı olarak Hibernate'i, ve repository interface'lerini gerçek, çalışan bean'lere dönüştüren altyapıyı -- hepsini tek bir satır manuel yapılandırma olmadan -- otomatik olarak yapılandırmasına yeter. Bu auto-configuration mekanizmasının (`@ConditionalOnClass`, auto-configuration sınıfları vb.) tam olarak nasıl çalıştığı zaten "Spring Boot Auto-Configuration ve Properties"te işlendi -- bu ders yalnızca sonucuna ihtiyaç duyuyor: kodun ile PostgreSQL arasındaki tesisat var, çünkü Spring Boot onu bir araya getirdi, sen yazdığın için değil.

Bu Projeden Küçük Bir Örnek

Bir araya getirildiğinde, bu uygulama başladığında ve daha sonra bir konuyu slug'ına göre okuduğunda gerçekte ne olduğu -- uydurma bir örnek değil, bu kod tabanındaki gerçek `Topic`, `Category`, `Course` ve `TopicRepository` sınıfları kullanılarak:

```
Topic.java (@Entity)
     ↓
Hibernate annotation'larını okur, onu "topic" tablosuna eşler
     ↓
TopicRepository extends JpaRepository<Topic, Long>
     ↓
Spring Data JPA başlangıçta gerçek bir implementasyon üretir
     ↓
topicRepository.findBySlug("records")
     ↓
Spring Data JPA metot adından bir sorgu çıkarır
     ↓
JPA'nın EntityManager'ı onu çalıştırır
     ↓
Hibernate onu SQL'e çevirir
     ↓
PostgreSQL bir satır döndürür
     ↓
Hibernate o satırı gerçek bir Topic nesnesine geri eşler
```

Bu uygulamanın kendi kaynak kodunda hiçbir yerde bunun için elle yazılmış bir `SELECT * FROM topic WHERE slug = ?` yoktur -- yukarıdaki her adım, bu dersin az önce tanıttığı dört katman sayesinde gerçekleşir, birinin o sorguyu elle yazması sayesinde değil.

Yaygın Yanlış Anlamalar

Dört ismin birbirine karışması kolay olduğu için doğrudan belirtmekte fayda olan birkaç ayrım var: **JPA, kurup çalıştırdığın bir kütüphane değildir** -- kendi başına hiçbir davranışı olmayan bir spesifikasyondur; bir şeyin onu uygulaması gerekir. **Hibernate ve JPA aynı şey değildir** -- Hibernate, JPA spesifikasyonunun bir (en yaygın) implementasyonudur, onun eş anlamlısı değil. **Spring Data JPA, Hibernate'in yerini almaz** -- hâlâ JPA'dan geçen ve altında hâlâ Hibernate tarafından SQL olarak çalıştırılan repository implementasyonları üretir; Spring Data JPA'da ikisini de atlayan hiçbir şey yoktur.

Sırada Ne Var

Bu ders bilinçli olarak zihinsel-model seviyesinde kaldı -- tek bir minimal `@Entity` ve tek bir repository interface'i, ilişki yok, custom sorgu yok, bir sınıfı geçerli bir entity yapan şeye derin bir dalış yok. Bu kategoride sıradaki "Entities and the Repository Abstraction", tam olarak orada devam ediyor: `@Id`, `@GeneratedValue`, `@Table`, `@Enumerated` ve bir avuç başka mapping annotation'ının gerçekte ne gerektirdiği, `Repository` → `CrudRepository` → `JpaRepository` interface hiyerarşisinin her birinin ne eklediği, ve bir sınıfı baştan iyi biçimlendirilmiş bir entity yapan şey.

Best Practices

- Spring Data JPA'da bir şey "sihir" gibi hissettirdiğinde dört katmanlı resmi aklında tut -- bir repository metodunun davranışı her zaman altındaki JPA ve Hibernate'e geri iz sürülebilir, Spring Data JPA'nın kendi başına icat ettiği bir şeye değil.
- JPA ve Hibernate'in altında hâlâ gerçek işi yaptığını anlamadan Spring Data JPA'nın repository soyutlamasına başvurma -- bu anlayış, sonraki, daha derin konuların (sorgular, fetching, persistence context) anlamlı olmasını sağlayan şeydir.
- Bir davranışın hangi katmana ait olduğundan emin olmadığında sor: bu, mapping'in NASIL GÖRÜNMESİ gerektiğiyle mi (JPA), gerçekte NASIL ÇALIŞTIRILDIĞIYLA mı (Hibernate), yoksa onları kullanmanın etrafındaki BOILERPLATE'İ KALDIRMAKLA mı (Spring Data JPA) ilgili?

Yaygın Hatalar

- "JPA" ve "Hibernate"i, bir spesifikasyon ve onun bir implementasyonu yerine, aynı şeyin birbirinin yerine geçebilir kelimeleri gibi ele almak.
- Spring Data JPA'nın JPA'dan tamamen ayrı bir teknoloji olduğunu varsaymak, onun üzerine doğrudan inşa edilmiş ince bir soyutlama katmanı olduğu yerine.
- Spring Boot'un altında egzotik bir şey yaptığını beklemek -- buradaki rolü, aksi hâlde elle kurabileceğin aynı JPA/Hibernate/Spring Data JPA parçalarını otomatik olarak yapılandırmaktan, başka bir şey değil.
- Repository metot adlarını, bunların altta yatan bir JPA/Hibernate mekanizmasından üretildiğini, mekanizmanın kendisi olmadığını anlamadan ezberlemeye doğrudan atlamak.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Düz bir Java nesnesinin bir veritabanı satırı hâline gelmesinin yerleşik bir yolu yoktur -- ORM, bu çeviriyi otomatikleştirme genel fikridir.
- JPA, Java'da object-relational mapping'in nasıl çalışması gerektiğini tarif eden bir spesifikasyondur; bir sözleşme tanımlar, bir çalışma zamanı değil.
- Hibernate, JPA'nın somut bir implementasyonudur -- `@Entity` ile işaretlenmiş sınıfları ve JPA çağrılarını gerçekte SQL'e çeviren şey odur.
- Spring Data JPA, JPA'nın üzerine inşa edilmiş bir repository soyutlamasıdır: bir interface bildirirsin, ve o çalışan bir implementasyon üretir, JPA/Hibernate'i doğrudan kullanmanın boilerplate'ini kaldırır.
- Spring Boot yeni bir katman eklemez -- diğer üçünü birbirine bağlayan `DataSource`, `EntityManagerFactory`, JPA provider'ı, ve repository altyapısını otomatik olarak yapılandırır.

Cheat Sheet

```
Java nesnesi
     ↓
    JPA        (spesifikasyon: @Entity, @Id, EntityManager, ...)
     ↓
 Hibernate      (implementasyon: gerçek SQL'i üretir)
     ↓
    SQL
     ↓
PostgreSQL

Repository
     ↓
Spring Data JPA (interface'inin çalışan bir implementasyonunu üretir)
     ↓
    JPA
     ↓
 Hibernate
     ↓
PostgreSQL
```

```
@Entity
public class Topic {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
}

interface TopicRepository extends JpaRepository<Topic, Long> {
    Optional<Topic> findBySlug(String slug); // burada hiç SQL yazılmadı
}
```

Terimler Sözlüğü

- **ORM (Object-Relational Mapping)**: Java nesneleri ile ilişkisel veritabanı satırları /tabloları arasında otomatik çeviri yapma genel fikri.
- **JPA (Jakarta Persistence API)**: Java'da object-relational mapping'in nasıl çalışması gerektiğini tanımlayan bir spesifikasyon -- bir çalışma zamanı implementasyonu değil, bir sözleşme.
- **Hibernate**: JPA spesifikasyonunun somut bir implementasyonu; Spring Boot uygulamalarında kullanılan en yaygın olanı.
- **Spring Data JPA**: bildirilen bir interface'ten çalışan bir implementasyon üreten, JPA'nın üzerine inşa edilmiş bir repository soyutlaması.
- **JPA provider**: bir uygulamanın çalışma zamanında kullandığı belirli JPA implementasyonu -- bu projede Hibernate.