Entities and the Repository Abstraction

@Entity/@Id/@GeneratedValue/@Column mapping'i, @Table/@Enumerated/nullable-unique kısıtları/argümansız constructor/entity vs DTO/equals()-hashCode() (kısaca), ve Repository → CrudRepository → PagingAndSortingRepository → JpaRepository hiyerarşisinin her katmanının ne kattığı -- bu projenin gerçek Topic/Category /TopicRepository sınıflarıyla. Spring Data JPA kategorisinin 2.'si.

Orta 30 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/entities-and-repositories

---

"JPA, Hibernate ve Spring Data JPA", zihinsel-model seviyesinde kaldı -- tek bir minimal `@Entity`, tek bir repository interface'i, ikisini de doğru yapan şeye dair gerçek bir ayrıntı yoktu. Bu ders bunu dolduruyor: bir sınıfın düzgün eşlenmiş bir JPA entity'si olması için tam olarak neye ihtiyacı olduğu, ve `Repository` → `CrudRepository` → `JpaRepository` hiyerarşisinin her katmanının gerçekte ne kattığı.

@Entity, @Table ve Primary Key'ler: @Id ve @GeneratedValue

Önceki dersin `@Entity` / `@Id` / `@GeneratedValue` üçlüsü başlangıç noktası -- bu projenin gerçek `Topic` entity'si, bu eşlemenin gerçekten tam bir versiyonunun neye benzediğini gösteriyor.

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;

// A fuller, teaching-annotated look at THIS PROJECT'S OWN Topic entity
// (see src/main/java/com/cdurgun/learning/domain/Topic.java) -- every
// annotation here maps directly onto a real column in the real "topic"
// table.
@Entity
// @Table names the table explicitly. Without it, Hibernate would derive a
// name from the class name itself -- being explicit avoids surprises when
// the class name and the table name diverge, or just for clarity.
@Table(name = "topic")
public class EntityMappingExample {

    @Id
```

```
    // GenerationType.IDENTITY delegates id generation to the database
    // itself (PostgreSQL's own auto-increment) -- the simplest strategy,
    // and the one this project uses everywhere. SEQUENCE and AUTO exist
    // for databases/scenarios that need more control over how ids are
    // generated; IDENTITY is enough for the vast majority of applications.
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // nullable = false, unique = true directly becomes a NOT NULL UNIQUE
    // constraint on the "slug" column -- these constraints are enforced by
    // the DATABASE, not just checked in Java, exactly like the Flyway
    // migration that created this table declares them.
    @Column(nullable = false, unique = true)
    private String slug;

    // A field with no @Column at all is still mapped -- Hibernate derives
    // a column name from the field name (estimatedMinutes -> a column
    // matching that name) unless told otherwise. @Column(name = "...") is
    // only needed when the column's real name doesn't match the field.
    @Column(name = "estimated_minutes")
    private Integer estimatedMinutes;

    // A relationship field -- @ManyToOne(fetch = FetchType.LAZY) tells
    // Hibernate that many Topics point to one Category. Treat this as just
    // another mapped field for now; exactly what "fetch = LAZY" means and
    // how relationships like this actually behave is the subject of
    // "Relationships, Fetching, and the N+1 Problem," later in this
    // category.
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id", nullable = false)
    private CategoryStandIn category;

    protected EntityMappingExample() {
    }

    static class CategoryStandIn {
    }
}
```

`@Table(name = "topic")`, Hibernate'in sınıf adından bir tane çıkarmasına izin vermek yerine, tabloyu açıkça adlandırır -- adlar zaten eşleşecek olsa bile bunu yapmakta fayda var, çünkü eşlemeyi örtük değil, okunduğunda açık hâle getirir. `@GeneratedValue(strategy = GenerationType.IDENTITY)`, id üretimini PostgreSQL'in kendi auto-increment'ine devreder -- birkaç üretim stratejisinin en basiti, ve bu projenin her yerde kullandığı. `SEQUENCE` ve `AUTO`, daha ince kontrol gerektiren senaryolar için vardır, ama `IDENTITY` uygulamaların büyük çoğunluğunu kapsar. `category` alanına da dikkat et -- bu bir ilişki, `@ManyToOne` ile eşlenmiş. Şimdilik onu yalnızca başka bir eşlenmiş alan gibi ele al; `fetch = FetchType.LAZY`'nin gerçekte ne yaptığı, ve bunun gibi ilişkilerin gerçekte nasıl davrandığı, bu kategoride ilerideki "İlişkiler, Fetching ve N+1 Problemi"nin konusu.

Alanları Eşlemek: @Column, Nullable ve Unique Kısıtları

Bir alanın eşlenmesi için hiçbir annotation'a ihtiyacı yoktur -- Hibernate, alan adından otomatik olarak bir sütun adı çıkarır. `@Column`, daha fazlasını söylemesi gereken durumlar içindir.

`slug` üzerindeki `@Column(nullable = false, unique = true)`, yalnızca Java'da bir yerde kontrol edilen değil, VERİTABANININ kendisi tarafından zorlanan gerçek bir `NOT NULL UNIQUE` kısıtı hâline gelir -- bu projenin kendi Flyway migration'larının o sütun için bildirdiğiyle birebir eşleşerek. `@Column(name = "estimated_minutes")`, yalnızca gerçek sütun adı alan adıyla (burada `estimatedMinutes`) eşleşmediğinde gereklidir -- `slug`'da olduğu gibi zaten eşleştiklerinde, `name`e gerek yoktur.

@Enumerated ile Enum Alanları

Bir enum alanının açıkça verilmesi gereken bir kararı vardır, yoksa gerçekten tehlikeli bir varsayılan devreye girer.

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.Id;

public class EnumeratedFieldExample {

    enum Difficulty {
        BEGINNER, INTERMEDIATE, ADVANCED
    }

    @Entity
    static class Topic {
        @Id
        private Long id;

        // EnumType.STRING stores the enum's NAME ("INTERMEDIATE") in the
        // column -- readable in the database, and safe to reorder the
        // enum's constants later without corrupting existing rows. This
        // is what this project's own Topic entity actually uses.
        @Enumerated(EnumType.STRING)
        @Column(nullable = false)
        private Difficulty difficulty;

        // The alternative, EnumType.ORDINAL (or omitting @Enumerated
        // entirely, which defaults to it), stores the enum's POSITION
        // (0, 1, 2, ...) instead. It looks harmless until someone inserts
        // a new constant in the middle of the enum later -- every existing
        // row's stored number now points at a DIFFERENT constant than the
        // one it was saved with, silently. Avoid ORDINAL in real code.
    }

    public static void main(String[] args) {
        Topic topic = new Topic();
        topic.difficulty = Difficulty.INTERMEDIATE;

        System.out.println(topic.difficulty); // stored as the string "INTERMEDIATE"
    }
}
```

`@Enumerated(EnumType.STRING)`, enum sabitinin ADINI ("INTERMEDIATE") sütunda saklar -- veritabanında doğrudan okunabilir, ve enum'un sabitlerini daha sonra yeniden sıralamak güvenlidir. Alternatif, `EnumType.ORDINAL` (`@Enumerated`'i basitçe atlayarak elde ettiğin şey), sabitin POZİSYONUNU düz bir tamsayı olarak saklar -- biri enum'un ortasına yeni bir sabit ekleyene kadar zararsızdır, o noktada var olan her satırın sakladığı sayı, gerçekte kaydedildiği sabitten SESSİZCE farklı bir sabiti işaret eder. Bu projenin kendi `Topic.difficulty` alanı `STRING` kullanıyor, ve bu kendi kodunda da varsayılan seçim olmalı.

İyi Biçimlendirilmiş Bir Entity İçin Birkaç Kural Daha

Bir sınıfı gerçekten iyi biçimlendirilmiş bir JPA entity'si yapan şeyi tamamlayan bir avuç küçük kural var -- her biri kendi tam bölümüne ihtiyaç duymayacak kadar dar.

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

import java.util.Objects;

@Entity
public class WellFormedEntityRulesExample {

    @Id
    @GeneratedValue
    private Long id;

    private String slug;

    // A no-args constructor is REQUIRED -- Hibernate creates entity
    // instances via reflection, before any field is populated, so there
    // must be a constructor it can call with nothing. "protected" (rather
    // than "public") is a common convention: it keeps the constructor
    // available to Hibernate and to code in the same package, while
    // discouraging application code elsewhere from calling it directly
    // and getting a half-built entity.
    protected WellFormedEntityRulesExample() {
    }

    public WellFormedEntityRulesExample(String slug) {
        this.slug = slug;
    }

    // equals()/hashCode() based on the ID is the common, safe choice for
    // an entity -- but only once the entity actually HAS an id. Two
    // brand-new, unsaved instances both have id == null, so comparing by
    // id alone would make every new, unsaved entity "equal" to every
    // other one -- this implementation deliberately treats two entities
    // as equal ONLY when they both have a real, matching id.
    @Override
    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        if (!(other instanceof WellFormedEntityRulesExample that)) {
            return false;
        }
        return id != null && id.equals(that.id);
    }

    @Override
    public int hashCode() {
        // A FIXED value, not Objects.hash(id) -- an entity's hashCode
        // must never change after it's been placed in a HashSet/HashMap,
        // but its id DOES change (from null to a real value) the moment
        // it's saved. Using a constant keeps hashCode stable across that
        // transition; equals() above still does the real comparison work.
        return Objects.hashCode(getClass());
    }
}
```

Argümansız bir constructor gereklidir, önceki dersin zaten belirttiği gibi -- Hibernate, entity instance'larını, hiçbir alan doldurulmadan önce, reflection ile inşa eder, bu yüzden hiçbir argümanla çağrılabilecek bir constructor'a ihtiyacı vardır; `protected` (`public` yerine), onu Hibernate'e açık tutarken uygulama kodunun onu doğrudan çağırmasını caydıran yaygın bir konvansiyondur. (Bu aynı zamanda, "Record"un işlediği gibi, bir `record`'un asla bir JPA entity'si OLAMAMASININ tam nedenidir -- ne argümansız bir constructor'ı ne de sonradan doldurulacak mutable alanları vardır.)

`equals()` / `hashCode()`, entity'lere özgü bir dikkat gerektirir. `equals()`'ı id'ye dayandırmak açıkça doğru görünür, ama iki yepyeni, kaydedilmemiş entity'nin ikisi de `id == null`'dır -- yalnızca id'ye göre karşılaştırmak, her yeni instance'ı diğer her yeni instance'a "eşit" yapardı, bu yüzden bu implementasyon iki entity'yi yalnızca ikisi de gerçek, eşleşen bir id'ye sahip olduğunda eşit sayar. `hashCode()`'un (id'yi hash'lemek yerine) SABİT bir değer döndürmesi daha ince bir nedenle önemlidir: bir entity'nin hashCode'u, bir `HashSet` / `HashMap`'e yerleştirildikten sonra asla değişmemelidir, ama id'si -- `null`'dan gerçek bir değere -- tam olarak kaydedildiği anda değişir, bu yüzden id'yi hash'lemek bu sözleşmeyi tam da önemli olduğu anda bozardı.

Entity vs. DTO

Bir `@Entity` ve bir DTO farklı sorunları çözer, ve bunları karıştırmak bir REST API'si olan bir Spring Boot uygulamasında gerçek sorunlara yol açar.

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@Entity
class Topic {
    @Id
    @GeneratedValue
    private Long id;
    private String slug;
    private Integer estimatedMinutes;
    // A real entity also carries a lazy @ManyToOne Category, JPA-internal
    // proxy state, and more -- none of which a client asking for "a topic"
    // over HTTP should ever see or need to know about.

    Long getId() { return id; }
    String getSlug() { return slug; }
    Integer getEstimatedMinutes() { return estimatedMinutes; }
}

// A DTO -- as covered in "Record" -- is a SEPARATE, deliberately narrow
// type exposing only what a specific API response actually needs.
record TopicResponse(Long id, String slug, Integer estimatedMinutes) {
    static TopicResponse from(Topic topic) {
        return new TopicResponse(topic.getId(), topic.getSlug(), topic.
getEstimatedMinutes());
    }
}

// Returning the @Entity directly from a controller couples the API's
// public shape to the database mapping (renaming a column would silently
// change the JSON response) and risks serializing a lazy field that
// throws outside a transaction -- covered in "Transaction Management."
// Returning a DTO avoids both: the response shape is controlled
// explicitly, independent of how the entity is mapped or fetched.
@RestController
class TopicController {

    @GetMapping("/topics/{id}")
    public TopicResponse getTopic(@PathVariable Long id) {
        Topic topic = new Topic(); // a real controller would load this from a repository
        return TopicResponse.from(topic);
    }
}
```

`Topic`'i doğrudan `TopicController`'dan döndürmek, API'nin genel JSON şeklini veritabanı eşlemesinin kendisine bağlardı -- bir sütunu yeniden adlandırmak, hiç kimse controller'a dokunmadan yanıtı değiştirir -- ve bir transaction dışında lazy bir alanı serialize etmeye çalışma riski taşır (tam olarak "Transaction Management"in işlediği `LazyInitializationException` senaryosu). "Record"da işlendiği gibi bir `record` olan

`TopicResponse`, `Topic`'in kendisinin nasıl eşlendiğinden ya da getirildiğinden tamamen bağımsız, yalnızca bu tek yanıtın gerçekten ihtiyaç duyduğunu açığa çıkaran küçük, ayrı bir türdür.

Repository Hiyerarşisi: Repository, CrudRepository ve JpaRepository

Bu projedeki her repository interface'i `JpaRepository`'yi genişletir -- ama `JpaRepository`'nin kendisi kısa bir zincirin son halkasıdır, ve her halkanın ne kattığını bilmek, o "bedava" metotların hepsinin gerçekte nereden geldiğini açıklar.

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

// This project's own CategoryRepository (see src/main/java/com/cdurgun/
// learning/repository/CategoryRepository.java) -- one line, three
// inherited tiers of behavior underneath it.
interface CategoryRepositoryExample extends JpaRepository<CategoryExample, Long> {

    // --- From Repository (the root, marker interface) ---
    // Contributes no methods at all -- it exists purely so Spring Data can
    // recognize "this interface is a repository" and generate a bean for it.

    // --- From CrudRepository ---
    // save(category), findById(id), findAll(), count(), existsById(id),
    // deleteById(id), delete(category), deleteAll() -- the basic create/
    // read/update/delete operations every repository needs, written ONCE
    // in Spring Data JPA's own code, inherited here for free.

    // --- From PagingAndSortingRepository (which JpaRepository also extends) ---
    // findAll(Sort sort), findAll(Pageable pageable) -- sorted and paged
    // reads, without a single line of query code. "Pagination, Sorting,
    // and Projections," later in this category, covers using these for real.

    // --- From JpaRepository itself (the most specific tier) ---
    // flush() (forces pending changes to the database immediately),
    // saveAndFlush(category), deleteAllInBatch() (a single bulk DELETE
    // instead of one per row) -- JPA-specific operations the more generic
    // tiers above don't know about.
}

class CategoryExample {
}
```

`Repository`, köktür -- hiçbir metot katmayan bir marker interface'i; tek işi, Spring Data'nın "bu bir repository" olduğunu tanımasına ve onun için bir bean üretmesine izin vermektir. `CrudRepository`, her repository'nin ihtiyaç duyduğu temel işlemleri ekler -- `save`, `findById`, `findAll`, `count`, `existsById`, `deleteById` ve daha fazlası -- Spring Data JPA'nın kendi içinde bir kez yazılmış, burada bedavaya miras alınan. `JpaRepository`'nin de genişlettiği `PagingAndSortingRepository`, `findAll(Sort)` ve `findAll(Pageable)` ekler -- kendi sorgu kodun olmadan sıralı ve sayfalı okumalar; bunları gerçekten kullanmak, bu kategoride ilerideki "Sayfalama, Sıralama ve Projeksiyonlar"ın konusu. `JpaRepository`'nin kendisi, daha genel

katmanların bilmediği JPA'ya özgü ekstralar ekler -- `flush()`, `saveAndFlush(...)`, `deleteAllInBatch()`.

Her Katman Gerçekte Sana Ne Verir

Sade bir dille: `interface CategoryRepository extends JpaRepository<Category, Long> {}`'ı -- boş bir gövdeyle -- bildirmek, hiçbirini senin yazmadığın, tamamen çalışan bir `save`, `findById`, `findAll`, `count`, `existsById`, `deleteById`, sıralı/sayfalı okumalar, ve JPA'ya özgü toplu işlemler zaten verir. Bu tam olarak önceki dersteki "Spring Data JPA çalışan bir implementasyon üretir" ifadesinin anlamıydı -- gerçek işi yapan bu üç miras alınan katmandır, her interface için özel olarak icat edilen bir şey değil.

Bir Araya Getirmek: Bu Projenin Topic'i ve Category'si

Bu projenin gerçek `Topic` ve `Category` entity'leri, ve onların repository'leri, bu dersin her parçasını birbirine bağlıyor: `Topic`, `@Entity`/`@Table` ile `topic` tablosuna eşlenir, `id`'si `@GeneratedValue(strategy = IDENTITY)` ile veritabanı tarafından üretilir, `slug`'ı `@Column` ile `NOT NULL UNIQUE`'dir, `difficulty`'si `@Enumerated` ile `STRING`-eşlenmiş bir enum'dur, ve `TopicRepository extends JpaRepository<Topic, Long>`, sıfır elle yazılmış CRUD koduyla ona çalışan kalıcılık verir. `Category`, kendi tablosu için aynı deseni birebir izler. Ne entity'nin ne de repository'nin hiçbiri, bu dersin az önce işlediğinin ötesinde hiçbir şey gerektirmedi.

Yaygın Yanlış Anlamalar

"Bir entity'nin yalnızca getter/setter'a ihtiyacı vardır." İyi biçimlendirilmiş bir entity'nin ayrıca argümansız bir constructor'a, ve (genelde) id-tabanlı `equals()`/`hashCode()`'a ihtiyacı vardır -- ikisini de atlamak, gecikmeli de olsa gerçek sorunlara yol açar. **"`@Column`, her alan için gereklidir."** Değildir -- Hibernate varsayılan olarak her alanı eşler; `@Column`, yalnızca varsayılanın ötesinde bir şey (farklı bir isim, bir kısıt) söylemek içindir. **"Bir repository interface'inin metotları bir şekilde interface başına, sıfırdan üretilir."** Üretilmez -- `save`/`findById`/`findAll` ve geri kalanı, Spring Data JPA'nın zaten bir kez implement ettiği küçük, sabit bir interface kümesinden (`CrudRepository`, `PagingAndSortingRepository`, `JpaRepository`) gelir; senin interface'in onları yalnızca miras alır.

Sırada Ne Var

Bu ders, tek, bağımsız bir entity'yi eşlemeyi ve bedavaya gelen repository metotlarını kullanmayı işledi -- hiçbir yerde custom bir sorgu yazılmadı. Bu kategoride sıradaki "Query Metotları ve JPQL ile @Query", tam olarak bunu işliyor: bir metodun adından bir sorgu çıkarmak (`TopicRepository`'nin gerçek `findBySlug(...)`'ı gibi, önceki derste yalnızca geçerken adı anılan), ve türetilmiş bir isim yetmediğinde `@Query` ile doğrudan JPQL yazmak.

Best Practices

- Her `@Enumerated` alanı için `EnumType.STRING` kullan -- `ORDINAL`'ın sessiz-bozulma riski, biraz daha küçük depolama alanını neredeyse hiçbir zaman haklı çıkarmaz.
- Her entity'ye argümansız bir constructor (kural olarak protected) ve sabit bir `hashCode()` ile id-tabanlı `equals()`/`hashCode()` ver -- ikisini de sonradan akla gelen bir şey değil, bir kontrol listesi olarak ele al.
- Bir REST API'sinden entity değil, DTO döndür -- yanıt şeklinin entity'nin kendi eşlemesiyle sürüklenmesine izin vermek yerine bilinçli olarak karar ver.
- Varsayılan seçim olarak (daha dar `CrudRepository` yerine) `JpaRepository`'ye başvur -- eklediği paging/sorting ve JPA'ya özgü ekstralar, nadiren vazgeçmek isteyeceğin şeylerdir.

Yaygın Hatalar

- `@Enumerated(EnumType.STRING)`'i atlayıp varsayılan olarak `ORDINAL`'ı almak, sonra enum'u daha sonra yeniden sıralayıp var olan veriyi sessizce bozmak.
- `equals()`/`hashCode()`'u, lazy bir ilişki dahil, bir entity'nin TÜM alanlarına dayandırarak yazmak -- bu, istenmeyen veritabanı erişimini tetikleyebilir, ya da o anda ne yüklü olduğuna bağlı olarak tutarsız sonuçlar üretebilir.
- Bir `@RestController` metodundan düz bir `@Entity` döndürmek, API'nin yanıt şeklini veritabanı şemasına bağlamak ve bir `LazyInitializationException` riski taşımak.
- Bir repository interface'inin bir yerde elle yazılmış bir implementasyona ihtiyacı olduğunu varsaymak -- yoktur; miras alınan üç katman zaten başlangıçta bir tane sağlar.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- `@Entity` + `@Table`, bir sınıfı bir tabloya eşler; `@Id` + `@GeneratedValue(strategy = IDENTITY)`, bu projenin standart primary-key eşlemesidir.
- `@Column(nullable = ..., unique = ...)`, gerçek veritabanı kısıtlarını zorlar; bir alanın varsayılan olarak eşlenmesi için hiçbir annotation'a ihtiyacı yoktur.
- `@Enumerated(EnumType.STRING)`, bir enum alanını eşlemenin güvenli yoludur -- `ORDINAL`, enum daha sonra yeniden sıralanırsa sessiz veri bozulması riski taşır.
- İyi biçimlendirilmiş bir entity'nin argümansız bir constructor'a ve (genelde) sabit bir `hashCode()` ile id-tabanlı `equals()`/`hashCode()`'a ihtiyacı vardır.
- Bir entity ve bir DTO farklı sorunları çözer -- bir REST API'sinden doğrudan entity değil, DTO döndür.
- `Repository` → `CrudRepository` → `PagingAndSortingRepository` → `JpaRepository`, gerçek bir kalıtım zinciridir -- her "bedava" repository metodu, bu zincirin belirli bir katmanına kadar iz sürülebilir.

Cheat Sheet

```
@Entity
@Table(name = "topic")
public class Topic {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String slug;

    @Enumerated(EnumType.STRING)
    private Difficulty difficulty;

    protected Topic() {} // gerekli, argümansız

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Topic t)) return false;
        return id != null && id.equals(t.id);
    }

    @Override
    public int hashCode() { return Objects.hashCode(getClass()); }
}

// Repository → CrudRepository → PagingAndSortingRepository → JpaRepository
interface TopicRepository extends JpaRepository<Topic, Long> {
    // save/findById/findAll/count/existsById/deleteById -- CrudRepository'den
    // findAll(Sort)/findAll(Pageable)                    -- PagingAndSortingRepository'den
    // flush()/saveAndFlush()/deleteAllInBatch()            -- JpaRepository'den
}
```

Terimler Sözlüğü

- **@Table**: bir `@Entity`'nin sınıf adından çıkarılan bir isme güvenmek yerine, eşlendiği tabloyu açıkça adlandırır.
- **GenerationType.IDENTITY**: id üretimini veritabanının kendi auto-increment'ine devreden bir id-üretim stratejisi.
- **EnumType.STRING vs. ORDINAL**: eşlenmiş bir sütunda bir enum'un adını (yeniden sıralaması güvenli) mı yoksa sayısal pozisyonunu (yeniden sıralaması güvensiz) mü saklamak.
- **DTO (Data Transfer Object)**: genelde bir `record`, veriyi belirli bir sınır (bir API yanıtı gibi) için, nasıl kalıcı hale getirildiğinden bağımsız olarak şekillendirmek için tasarlanmış bir tür.
- **Repository / CrudRepository / PagingAndSortingRepository / JpaRepository**: her Spring Data JPA repository'sinin arkasındaki kalıtım zinciri, her katman belirli bir miras alınan metot kümesi katıyor.