Entities and the Repository Abstraction

@Entity/@Id/@GeneratedValue/@Column mapping, plus @Table/@Enumerated/nullable-unique constraints/the no-args-constructor requirement/entity vs. DTO/equals()-hashCode() (concisely), and what each tier of the Repository → CrudRepository → PagingAndSortingRepository → JpaRepository hierarchy actually contributes -- using this project's own Topic/Category/TopicRepository classes. The 2nd lesson in the Spring Data JPA category.

Intermediate 30 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/entities-and-repositories

---

"JPA, Hibernate, and Spring Data JPA" stayed at the mental-model level — one minimal `@Entity`, one repository interface, no real detail on what makes either of them correct. This lesson fills that in: exactly what a class needs to be a properly mapped JPA entity, and exactly what each tier of the `Repository` → `CrudRepository` → `JpaRepository` hierarchy actually contributes.

@Entity, @Table, and Primary Keys: @Id and @GeneratedValue

The previous lesson's `@Entity` / `@Id` / `@GeneratedValue` trio is the starting point — this project's real `Topic` entity shows what a genuinely complete version of that mapping looks like.

```
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")` names the table explicitly, rather than letting Hibernate derive one from the class name — worth doing even when the names would match, since it makes the mapping obvious to read rather than implicit. `@GeneratedValue(strategy = GenerationType.IDENTITY)` delegates id generation to PostgreSQL's own auto-increment — the simplest of a few generation strategies, and the one this project uses everywhere; `SEQUENCE` and `AUTO` exist for scenarios needing finer control, but `IDENTITY` covers the vast majority of applications. Notice the `category` field, too — it's a relationship, mapped with `@ManyToOne`. Treat it as just another mapped field for now; what `fetch = FetchType.LAZY` actually does, and how relationships like this really behave, is the subject of "Relationships, Fetching, and the N+1 Problem," later in this category.

Mapping Fields: @Column, Nullable and Unique Constraints

A field needs no annotation at all to be mapped — Hibernate derives a column name from the field name automatically. `@Column` is for the cases that need to say more.

`@Column(nullable = false, unique = true)` on `slug` becomes a real `NOT NULL UNIQUE` constraint, enforced by the DATABASE itself, not just checked somewhere in Java — exactly matching what this project's own Flyway migrations declare for that column. `@Column(name = "estimated_minutes")` is only needed when the real column name doesn't match the field name (`estimatedMinutes` here) — when they already match, as with `slug`, no `name` is necessary.

Enum Fields with @Enumerated

An enum field needs one specific decision made explicitly, or a genuinely dangerous default kicks in.

```
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)` stores the enum constant's NAME ("INTERMEDIATE") in the column — readable directly in the database, and safe to reorder the enum's constants later. The alternative, `EnumType.ORDINAL` (what you get by simply omitting `@Enumerated`), stores the constant's POSITION as a plain integer instead — harmless until someone inserts a new constant in the middle of the enum, at which point every existing row's stored number silently points at a different constant than the one it was actually saved with. This project's own `Topic.difficulty` field uses `STRING`, and that should be the default choice in your own code too.

A Few More Rules for a Well-Formed Entity

A handful of smaller rules round out what makes a class a genuinely well-formed JPA entity — each narrow enough not to need its own full section.

```
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());
    }
}
```

A no-args constructor is required, as the previous lesson already noted — Hibernate builds entity instances via reflection, before any field is populated, so it needs a constructor callable with nothing; `protected` (rather than `public`) is a common convention that keeps it available to Hibernate while discouraging application code from calling it directly. (This is also exactly why, as "Record" covers, a `record` can never be a JPA entity — it has no no-args constructor and no mutable fields to populate afterward.)

`equals()` / `hashCode()` need care specific to entities. Basing `equals()` on the id looks obviously correct, but two brand-new, unsaved entities both have `id == null` — comparing by id alone would make every new instance "equal" to every other one, so this only counts two entities as equal once they both have a real, matching id. `hashCode()` returning a FIXED value (rather than hashing the id) matters for a subtler reason: an entity's hashCode must never change after it's placed in a `HashSet` / `HashMap`, but its id changes — from `null` to a real value — the moment it's saved, so hashing the id would break that contract right when it matters most.

Entity vs. DTO

An `@Entity` and a DTO solve different problems, and conflating them causes real issues in a Spring Boot application with a REST API.

```
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);
    }
}
```

Returning `Topic` directly from `TopicController` would couple the API's public JSON shape to the database mapping itself — renaming a column changes the response without anyone touching the controller — and risks trying to serialize a lazy field outside a transaction (exactly the `LazyInitializationException` scenario "Transaction Management" covers). `TopicResponse`, a `record` as covered in "Record," is a small, separate type exposing only what this one response actually needs, entirely independent of how `Topic` itself is mapped or fetched.

The Repository Hierarchy: Repository, CrudRepository, and JpaRepository

Every repository interface in this project extends `JpaRepository` — but `JpaRepository` itself is the last link in a short chain, and knowing what each link contributes explains where all those "free" methods actually come from.

```
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` is the root — a marker interface contributing no methods at all; its only job is letting Spring Data recognize "this is a repository" and generate a bean for it. `CrudRepository` adds the basic operations every repository needs — `save`, `findById`, `findAll`, `count`, `existsById`, `deleteById`, and more — written once inside Spring Data JPA itself, inherited here for free. `PagingAndSortingRepository` (which `JpaRepository` also extends) adds `findAll(Sort)` and `findAll(Pageable)` — sorted and paged reads, with no query code of your own; using these for real is the subject of "Pagination, Sorting, and Projections," later in this category. `JpaRepository` itself adds JPA-specific extras the more generic tiers don't know about — `flush()`, `saveAndFlush(...)`, `deleteAllInBatch()`.

What Each Tier Actually Gives You

Put plainly: declaring `interface CategoryRepository extends JpaRepository<Category, Long> {}` — with an empty body — already gives you a fully working `save`, `findById`, `findAll`, `count`, `existsById`, `deleteById`, sorted/paged reads, and JPA-specific batch operations, none of which you wrote. This is exactly what "Spring Data JPA generates a working implementation" meant in the previous lesson — it's these three inherited tiers doing the actual work, not something invented specially for each interface.

Putting It Together: This Project's Topic and Category

This project's real `Topic` and `Category` entities, and their repositories, tie every piece of this lesson together: `Topic` maps to the `topic` table via `@Entity` / `@Table`, its `id` is database-generated via `@GeneratedValue(strategy = IDENTITY)`, its `slug` is `NOT NULL UNIQUE` via `@Column`, its `difficulty` is a `STRING`-mapped enum via `@Enumerated`, and `TopicRepository extends JpaRepository<Topic, Long>` gives it working persistence with zero hand-written CRUD code. `Category` follows the identical pattern for its own table. Nothing about either entity or either repository required anything beyond what this lesson just covered.

Common Misconceptions

"An entity needs getters/setters and that's it." A well-formed entity also needs a no-args constructor, and (usually) id-based `equals()` / `hashCode()` — skipping either causes real, if delayed, problems. **"`@Column` is required for every field."** It isn't — Hibernate maps every field by default; `@Column` is only for saying something beyond the default (a different name, a constraint). **"A repository interface's methods are somehow generated per-interface, from scratch."** They're not — `save` / `findById` / `findAll` and the rest come from a small, fixed set of interfaces (`CrudRepository`, `PagingAndSortingRepository`, `JpaRepository`) that Spring Data JPA already implements once; your interface just inherits them.

What Comes Next

This lesson covered mapping a single, standalone entity and using the repository methods that come for free — no custom queries were written anywhere. "Query Methods and JPQL with @Query," next in this category, covers exactly that: deriving a query from a method's name (like `TopicRepository`'s real `findBySlug(...)`, only named in passing back in the previous lesson), and writing JPQL directly with `@Query` when a derived name isn't enough.

Best Practices

- Use `EnumType.STRING` for every `@Enumerated` field — `ORDINAL`'s silent-corruption risk almost never justifies its slightly smaller storage footprint.

- Give every entity a no-args constructor (protected, by convention) and id-based `equals()` / `hashCode()` with a fixed `hashCode()` — treat both as a checklist, not an afterthought.
- Return DTOs, not entities, from a REST API — decide the response shape deliberately instead of letting it drift with the entity's own mapping.
- Reach for `JpaRepository` (not the narrower `CrudRepository`) as the default choice — the paging/sorting and JPA-specific extras it adds are rarely something you'd want to give up.

Common Mistakes

- Omitting `@Enumerated(EnumType.STRING)` and getting `ORDINAL` by default, then reordering the enum later and silently corrupting existing data.
- Writing `equals()` / `hashCode()` based on ALL of an entity's fields, including a lazy relationship — this can trigger unwanted database access, or produce inconsistent results depending on what's currently loaded.
- Returning a plain `@Entity` from a `@RestController` method, coupling the API's response shape to the database schema and risking a `LazyInitializationException`.
- Assuming a repository interface needs a manually written implementation somewhere — it doesn't; the three inherited tiers already provide one at startup.

Summary, Cheat Sheet, and Glossary

Summary

- `@Entity` + `@Table` map a class to a table; `@Id` + `@GeneratedValue(strategy = IDENTITY)` is this project's standard primary-key mapping.
- `@Column(nullable = ..., unique = ...)` enforces real database constraints; a field needs no annotation at all to be mapped by default.
- `@Enumerated(EnumType.STRING)` is the safe way to map an enum field — `ORDINAL` risks silent data corruption if the enum is ever reordered.
- A well-formed entity needs a no-args constructor and (usually) id-based `equals()` / `hashCode()` with a fixed `hashCode()`.
- An entity and a DTO solve different problems — return DTOs from a REST API, not entities directly.
- `Repository` → `CrudRepository` → `PagingAndSortingRepository` → `JpaRepository` is a real inheritance chain — every "free" repository method traces back to one specific tier of it.

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() {} // required, no-args

    @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 -- from CrudRepository
    // findAll(Sort)/findAll(Pageable)                    -- from PagingAndSortingRepository
    // flush()/saveAndFlush()/deleteAllInBatch()          -- from JpaRepository
}
```

Glossary

- **@Table**: names the table an `@Entity` maps to, explicitly rather than relying on a name derived from the class.
- **GenerationType.IDENTITY**: an id-generation strategy that delegates to the database's own auto-increment.
- **EnumType.STRING vs. ORDINAL**: storing an enum's name (safe to reorder) vs. its numeric position (unsafe to reorder) in a mapped column.
- **DTO (Data Transfer Object)**: a type — often a `record` — designed to shape data for a specific boundary (like an API response), independent of how it's persisted.
- **Repository / CrudRepository / PagingAndSortingRepository / JpaRepository**: the inheritance chain behind every Spring Data JPA repository, each tier contributing a specific set of inherited methods.