__/__/____

Spring Boot + JPA & Hibernate

Spring & spring boot :

Spring → framework for building Java Appl^n
- like toolbox that provides ready-made tools for common problems like manging objects, transactions & logging etc.

IoC :

Inversion of control

Eg : Car car = new Car();
Engine engine = new Engine();
car.setEgine(Engine)

This Explains creating an objects for a class.

But in Spring Ioc, we can annotate (or) describe the ~~class~~ class & Spring will create & wired objects for that class.

@Component
class Engine {
{

@Component
class Car {
Private final Engine Engine;
Public Car(Engine Engine) {
this.engine = Engine;
}
{

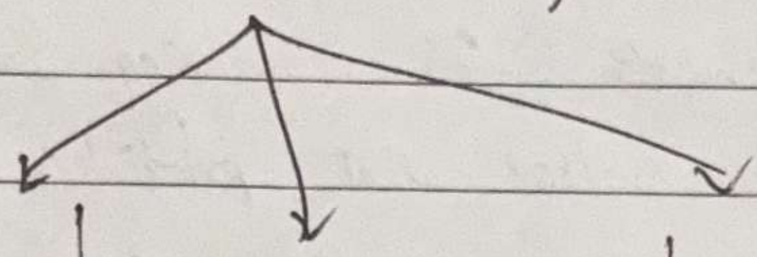
Creating & wiring of objects for a class which are annoted are handled by spring IoC

__/__/____

Dependency Injection (DI):

Providing dependencies instead of creating them inside a class.

Spring uses IoC to inject dependency.



Constructor Injection	Setter injection	field injection.

• injecting dependency into a class via its constructor
• no need @Autowired uses final fields
} same example on previous page
	Injected through setter methods
	Annotate with setter with @Autowired.
	can be used with optional via @Autowired(require = false).
	Eg: Class Car {
	Private Engine engine
	@Autowired
	Public void setEngine (Engine engine){
	this.Engine = engine;
	}
		Injected directly into fields.
		Class Car {
		@Autowired
		Private Engine Engine;
		}

__/__/____

Beans

bean → Java object managed by spring

bean
↓
Created → wired → used → destroyed } all this handle by spring

* bean can be declared by
@ component, @ service, @ Repository, @ Controller.

Application context : container that manages all beans.

* box that contains / stores all objects.

Spring boot :

spring + Auto configuration + Embedded Server.

Best practice in spring boot : MVC

Model — view — controller.

↓ ↘ ↘

Data & business logic → entity, repository & service layer.

User's request & response (JSP/JSON/React - frontend)

Middleman.
→ Takes request → call service returns response.

__/__/____

@RestController
@RequestMapping("/users")

Pubic Class userControllers { → controllers
    @Autowired.
    Privde final UserService userService; → service layer with repository.

@GetMapping("/{id}")
    pubic User getUserId(@PathVariable Long id) {.
        return userService.findUsersById(id);
    {

{

JPA & Hibernate :

JPA → Java Persistance API, not a framework, not a tool.

- its just specifiation
  ↓ set of rules & interfaces.

- it defines How Java object should be mapped to RDBMS

- JPA + Implimentation of JPA like Hibernate, Eclipselink, openJPA.

__/__/____

Hibernate : implementation of JPA specification.

*Entity Mapping → Maps Java object → DB tables.

*HQL (Hibernate query lang) → OO query lang

↓

Just like SQL but works with object

* Auto table creation * caching * lazy loading etc.

ORM → object Relational Mapping

reduce gap b/w Java object & Database Tables.

Eg;

@ Entity

@ Table (name = "users")

Public class User {

@ Id

Private Long id;

Private String name;

}

User u = new User(1L, 'example');

EntityManager.persist(u);

↓

Hibernate translate this to SQL like

Insert into users (id, name) values (1, 'Example');

__/__/____

JPA & JDBC;

using JDBC, we need to write SQL manually,
map results to Java objects manually.

But using JPA + Hibernate it will handle it Everything inside the also JDBC will work but no manual work needed.

Entity & Mapping

1. Java Class → JPA Entity (a table in DB)
   ↓
   we can do this by using (@) Entity annotation.
   (class name → Table name)

2. @Table
   customize Table name & schema.

3. @Id
   → Primary key column

4. @generatedValue
   → auto generate Primary key values

   ↳ Auto : Hibernate decides
   IDENTITY : auto-increment
   SEQUENCE : DB sequence
   TABLE : Uses a table to simulate sequence

__/__/____

5. @column → maps to specific DB column.

@Entity
@Table (name = 'users', schema = "app-data")
public class User {

@Id
@GeneratedValue
Private long Id;

@Column (name - "user-name")
Private string name;

}

Relationships ;

- 1-1 (one to one);
  1 Entity → maps to → one Entity.

Ey ; 1 User has 1 Profile

@ Entity
@ Table
Public class - User {
@ Id.
Private long id; → Primary key
@ OneToOne (cascade = cascadeType. ALL)
@ JoinColumn (name = "Profile - id") → Foreign key
Pride Profile profile
}

__/__/____

one to many & many to one

Eg; one dept → many Employees

one to many

@Entity
Public Class Department {

@Id @GeneratedValue
Private Long id;

@OneToMany (mappedBy = "department")
Private List<Employee> Employees = new ArrayList<>();

}

many to one

@Entity
pulic class Employee {

@Id
Private Long id;

@ManyToOne
@JoinColumn (name = 'dept_id") → Forige key
private Department department;

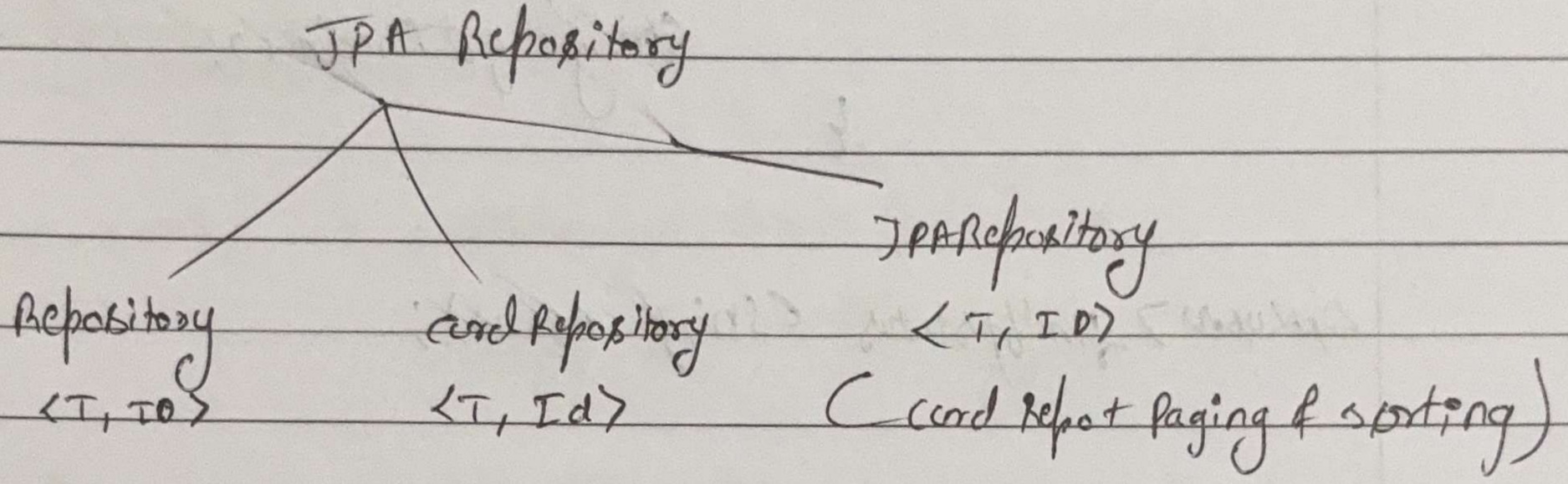
}

__/__/____

Repository layer

Spring Data JPA provides Interface called Repository

it is just an interfaces it will includes some basic methods to save, insert, update, delete, find, findAll to the respective tables.



method name → Hibernate → Query

Eg; findByName (string name) → method
↓ Hibernate
select * from User where name = ?;

Custom queries

we can write our own query with

@Query

Eg; @Query("select u from User where u.status = :status")
List<User> getByStatus(String status)

JPQL

JPQL works on entities of fields not table names

__/__/____

projection

- Some time we don't want entire table content
  At that we can restrict the response. Just like defining return type.

→ interface ; public ~~[redacted]~~ interface UserProjection {

```
String getName();
String getEmail();
}
```

List<User> findByStatus(String status);

→ class DTO

```
Public Class UserDTO {
    private String name;
    Private String Email;
    {
```

@Query("select new .com. userDTO(u.name, u.email) from User u)

__/__/____

Entity Manager: (EM)

It is a core interface of JPA

↓.

- Manages the peristence of entities (life cycle).
- Bridge b/w App'n & DB.

Methods in EM:

→ persist → entityManager.Persist(user) →
= nothing but insert query.

→ merge → update to entity

→ remove → Deletes the entity

→ find(class <T> EntityClass, object P.K)

Query with EM:

↳ JPQL

↳ Native Query

JPQL: operates on entities & fields

Native Query raw SQL Directly on DB.

__/__/____

Consider Table name = users & Entity name as User
And fields as name, Email & Columns as User-name &
user-email.

JPQL	native query
List<User> users = EntityManager. createQuery ("SELECT u from User u") .getResultList();	List<User> users = EntityManager. createNativeQuery ("select * from users") .getResultList();