Session 3 : Spring Boot

sourcemind

What Is Spring Boot ?

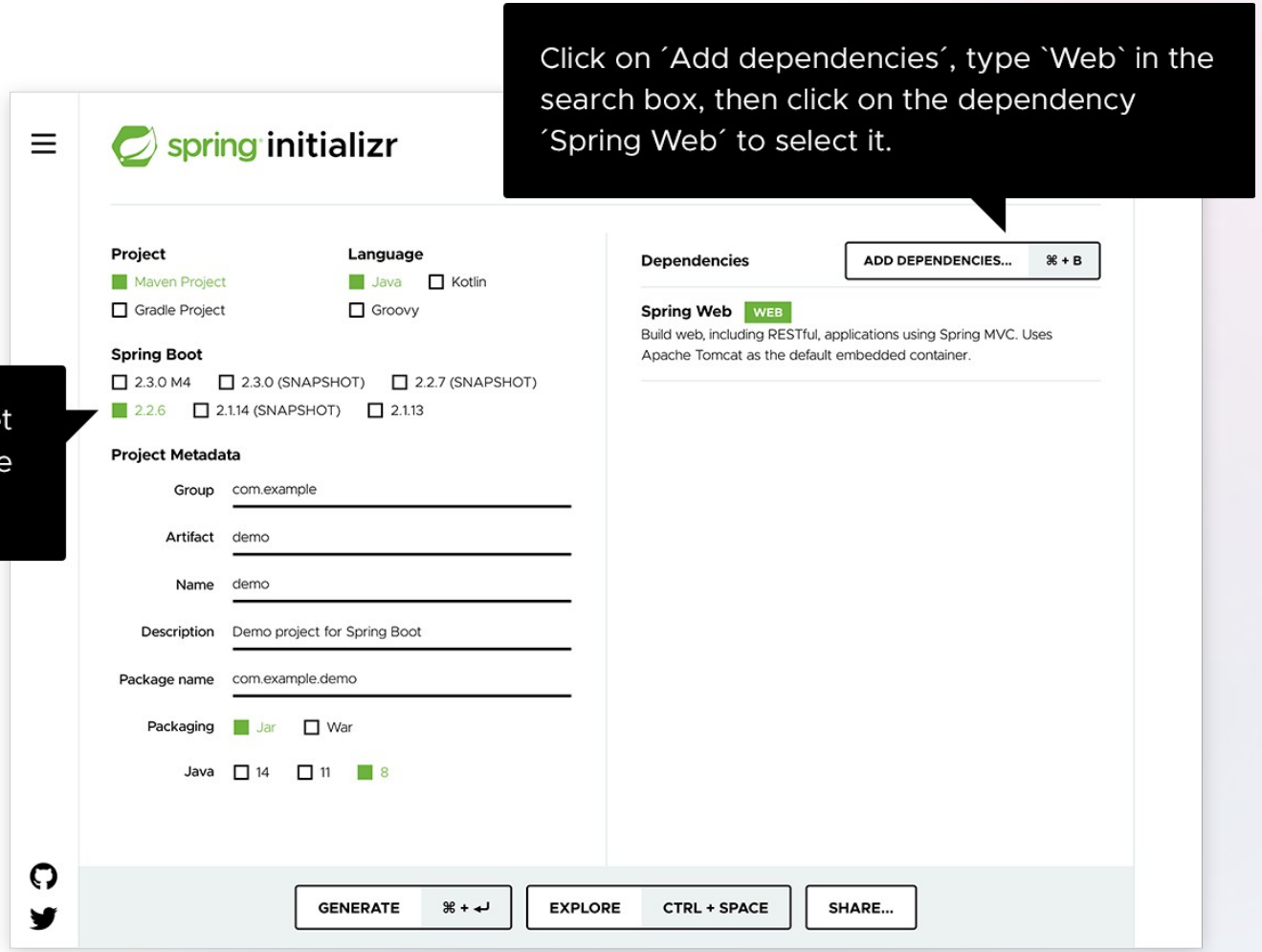
- Spring Boot is basically an extension of the Spring framework, which eliminates the boilerplate configurations required for setting up a Spring application.
- Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".
- it takes an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need minimal Spring configuration.

Features

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' dependencies to simplify your build configuration
- Automatically configure Spring and 3rd party libraries whenever possible
- Provide production-ready features such as metrics, health checks, and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

Spring Initializr

- http://start.spring.io/



sourcemind

Spring Initializr

- Hello endpoint

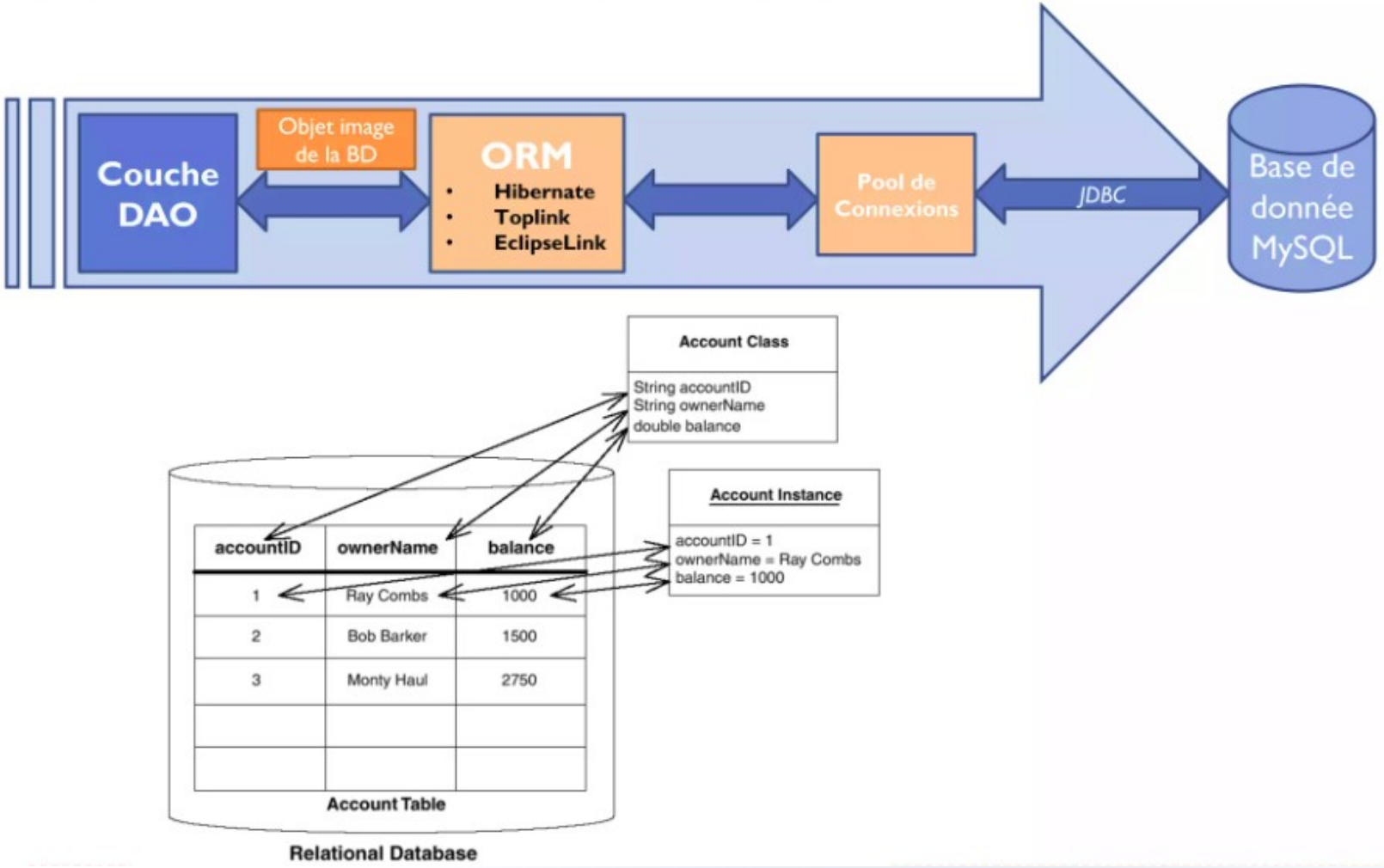
```
package com.example.demo;
 import org.springframework.boot.SpringApplication;
 import org.springframework.boot.autoconfigure.SpringBootApplication;
 import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
 import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication {
	public static void main(String[] args) {
		SpringApplication.run(DemoApplication.class, args);
	}
	@GetMapping("/hello")
	public String hello(@RequestParam(value = "name", defaultValue = "World") String name) {
		return String.format("Hello %s!", name);
	 }
	}
```

sourcemind

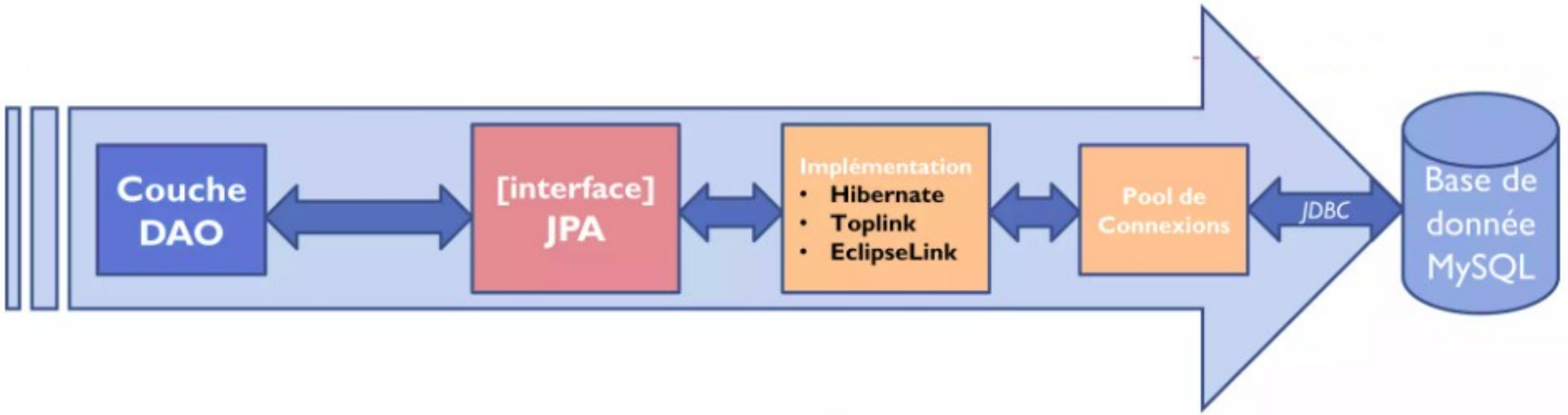
ORM : Object Relational Mapping

- Ease database access and manipulation



sourcemind

JPA : Java Persistence API



La spécification JPA est un ensemble d'interface du package javax.persistence

Exemple :

javax.persistence.Entity;
javax.persistence.EntityManagerFactory;
javax.persistence.EntityManager;
javax.persistence.EntityTransaction;

[EntityManager]
void **persist**(Entity entité)

JPA : Java Persistence API

- Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications.
  - ✓The Java Persistence API
  - ✓The query language
  - ✓The Java Persistence Criteria API
  - ✓Object/relational mapping metadata

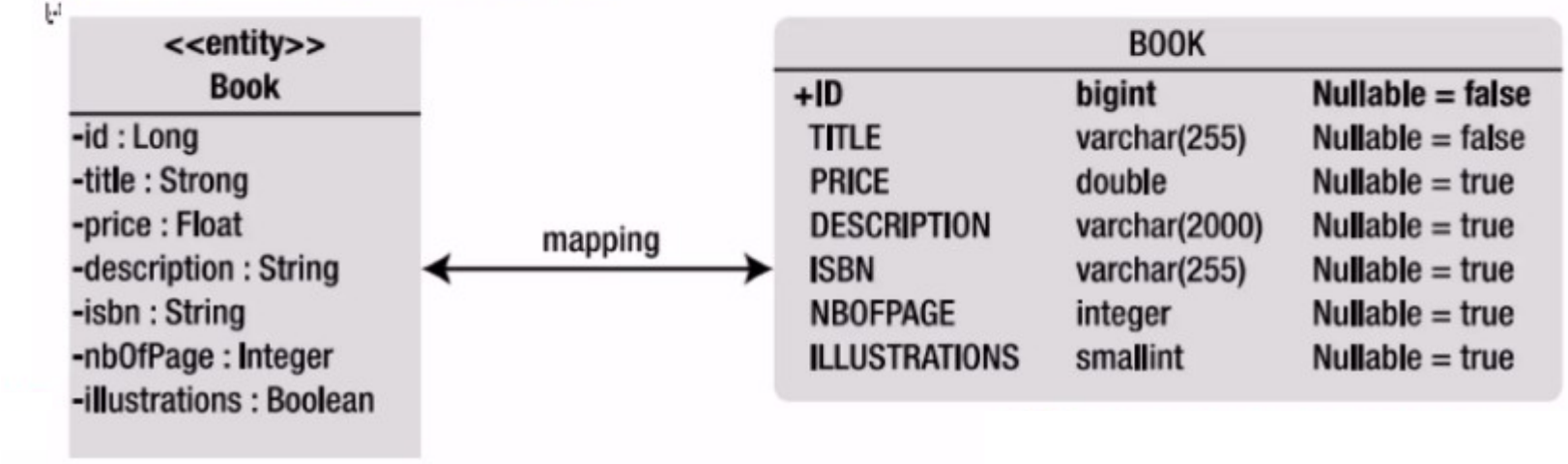
JPA : Entities

- An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table.

JPA : Entities

```java
@Entity
@Table(name = "BOOK")
public class Book {
    @Id
    @GeneratedValue
    private Long id;
    @Column(name = "TITLE",nullable = false)
    private String title;
    private Float price;
    @Basic(fetch = FetchType.LAZY)
    @Column(length = 2000)
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations

  //Les Getters et les Setters
```



JPA : Requirements for Entity Classes

- An entity class must follow these requirements.
  - ✓ The class must be annotated with the **javax.persistence.Entity** annotation.
  - ✓ The class must have a **public or protected, no-argument constructor**. The class may have other constructors.
  - ✓ The class must **not be declared final**. No methods or persistent instance variables must be declared final.
  - ✓ If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the Serializable interface.
  - ✓ Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
  - ✓ Persistent instance variables must be declared **private, protected, or package-private** and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods.

JPA : Persistent Fields and Properties in Entity Classes

- The fields or properties must be of the following Java language types:
  - ✓ Java primitive types
  - ✓ java.lang.String
  - ✓ Enumerated types
  - ✓ Other entities and/or collections of entities
  - ✓ Embeddable classes

JPA : Persistent Fields and Properties in Entity Classes

- The fields or properties must be of the following Java language types:
  - ✓ Java primitive types
  - ✓ java.lang.String
  - ✓ Enumerated types
  - ✓ Other entities and/or collections of entities
  - ✓ Embeddable classes
  - ✓ Other serializable types, including:
  - ✓ Wrappers of Java primitive types, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.TimeStamp, User-defined serializable types, byte[] , Byte[], char[], Character[]

JPA : Persistence Context

A **persistence context** is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed by an **EntityManager**.

It is the **first-level cache** where all the entities are fetched from the database or saved to the database.

The **EntityManager API** is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities

https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html

JPA : EntityManager methods

persist - Make an instance managed and persistent.

merge - Merge the state of the given entity into the current persistence context.

remove - Remove the entity instance.

find - Find by primary key. Search for an entity of the specified class and primary key.

refresh – it refreshes the state of the instance from the database

clear - Clear the persistence context, causing all managed entities to become detached.

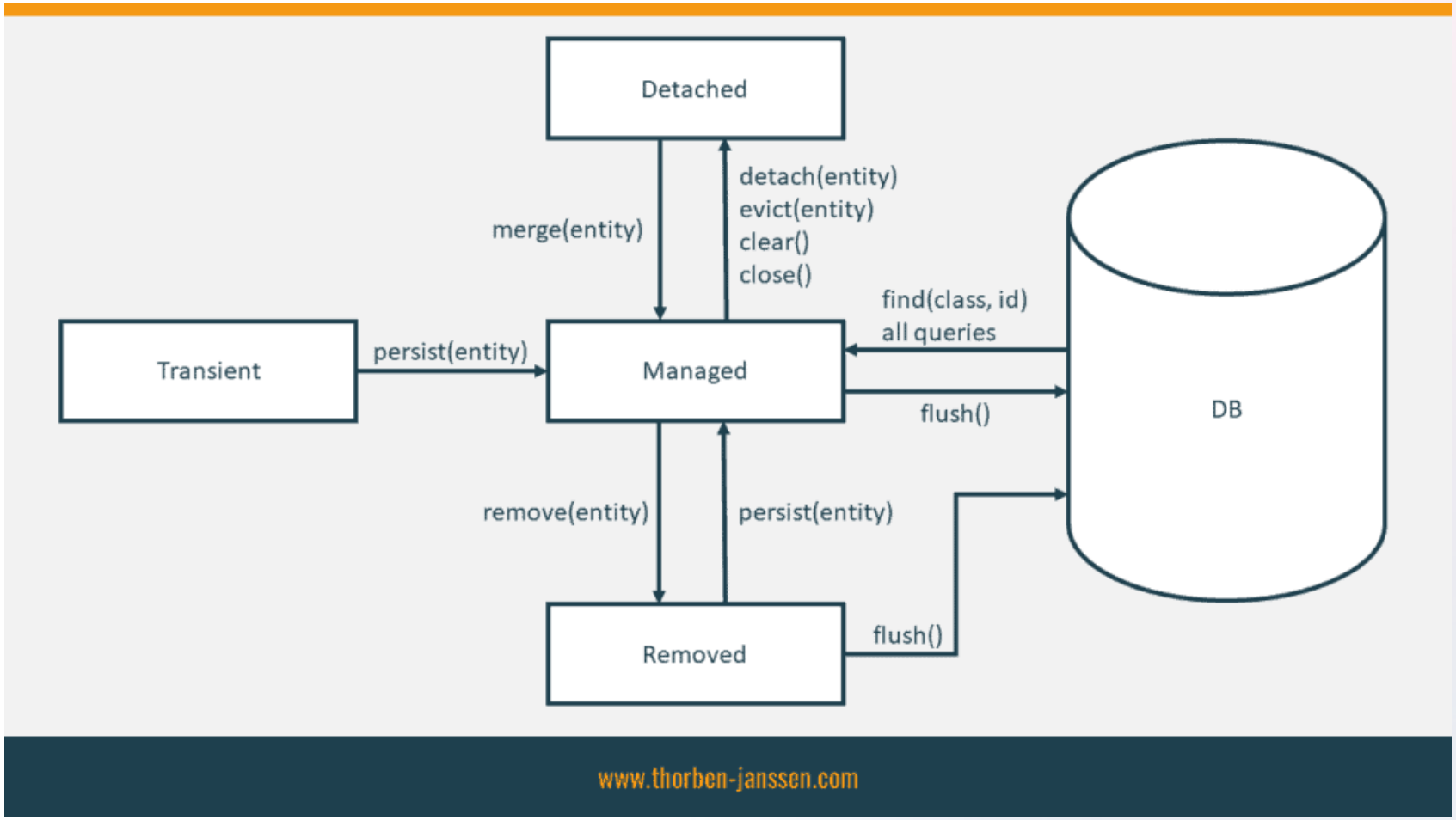
flush – Synchronizes the persistence context with the database.

contains – it checks if the managed entity belongs to the current persistence context.

createQuery - Create an instance of Query for executing a JPQL.

CreateNamedQuery, createNativeQuery, createNamedStoredProcedureQuery, createStoredProcedureQuery, etc. (‘
https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html)

JPA : Entity Lifecycle



https://thorben-janssen.com/entity-lifecycle-model/

JPA : EntityManagerFactory

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
```

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
3.   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
4.      <!--  provider -->
5.      <provider>org.hibernate.ejb.HibernatePersistence</provider>
6.      <properties>
7.         <!-- Classes persistantes -->
8.         <property name="hibernate.archive.autodetection" value="class, hbm" />
9.         <!-- logs SQL
10.           <property name="hibernate.show_sql" value="true"/>
11.           <property name="hibernate.format_sql" value="true"/>
12.           <property name="use_sql_comments" value="true"/>
13.        -->
14.        <!-- connexion JDBC -->
15.        <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
16.        <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/jpa" />
17.        <property name="hibernate.connection.username" value="jpa" />
18.        <property name="hibernate.connection.password" value="jpa" />
19.        <!--  création automatique du schéma -->
20.        <property name="hibernate.hbm2ddl.auto" value="create" />
21.        <!-- Dialecte -->
22.        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
23.        <!--  propriétés DataSource c3p0 -->
24.        <property name="hibernate.c3p0.min_size" value="5" />
25.        <property name="hibernate.c3p0.max_size" value="20" />
26.        <property name="hibernate.c3p0.timeout" value="300" />
27.        <property name="hibernate.c3p0.max_statements" value="50" />
28.        <property name="hibernate.c3p0.idle_test_period" value="3000" />
29.     </properties>
30.  </persistence-unit>
31. </persistence>
```

jpa is the unit name in the **METAINF/persistence.xml**

JPA : EntityManager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");

EntityManager  em  =  emf.createEntityManager() ;
```

JPA : Insertion operation

Version Java SE

```
public class Main {

 public static void main(String[] args) {

  // On crée une instance de livre
  Book book = new Book();
  book.setTitle("MUGC: JPA\MYSQL");
  book.setPrice(12.5F);
  book.setDescription("Science fiction");
        …
 // On récupère un pointeur sur l'entity manager
 // Remarque : dans une appli web, pas besoin de
    faire tout cela !
 EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("jpa");
 EntityManager em = emf.createEntityManager();

 // On rend l'objet « persistant » dans la base (on
    l'insère)
 EntityTransaction tx = em.getTransaction();
 tx.begin();
  em.persist(book);
 tx.commit();

 em.close();
 emf.close();

 }
```

Version Java EE

```
@Stateless
public class BookBean {

    @PersistenceContext(unitName = "jpa")
    private EntityManager em;

    public void createBook() {
          Book book = new Book();
        book.setTitle("MUGC: JPA\MYSQL");
        book.setPrice(12.5F);
        book.setDescription("Science fiction");
        book.setIsbn("1-84023-742-2");
        book.setNbOfPage(354);
        book.setIllustrations(false);

        em.persist(book);

        // Récupère le livre dans la BD par sa clé
     primaire
        book = em.find(Book.class, 1234L);

        System.out.println(book);
    }
}
```

sourcemind

Spring data jpa

- Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This module deals with enhanced support for JPA based data access layers. It makes it easier to build Spring-powered applications that use data access technologies.

https://spring.io/projects/spring-data-jpa

Repository definitions using module-specific interfaces

```
interface MyRepository extends JpaRepository<User, Long> { }

@NoRepositoryBean
interface MyBaseRepository<T, ID> extends JpaRepository<T, ID> {  }


interface UserRepository extends MyBaseRepository<User, Long> {  }
```

- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

Hibernate

Hibernate is an **Object/Relational Mapping** (ORM) solution for programs written in Java and other JVM languages.

While a strong background in SQL is not required to use Hibernate, a basic understanding of its concepts is useful.

https://docs.jboss.org/hibernate/orm/6.3/quickstart/html_single/

JPA relations

- Types : **one-to-one, one-to-many, many-to-one, and many-to-many**
  - ✓ **One-to-one**: Each entity instance is related to a single instance of another entity.
  - ✓ **One-to-many**: An entity instance can be related to multiple instances of the other entities.
  - ✓ **Many-to-one**: Multiple instances of an entity can be related to a single instance of the other entity. This multiplicity is the opposite of a one-to-many relationship.
  - ✓ **Many-to-many**: The entity instances can be related to multiple instances of each other.

JPA relations : Cascade Operations for Entities

Cascade Operation	Description
ALL	All cascade operations will be applied to the parent entity's related entity. `All` is equivalent to specifying `cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}`
DETACH	If the parent entity is detached from the persistence context, the related entity will also be detached.
MERGE	If the parent entity is merged into the persistence context, the related entity will also be merged.
PERSIST	If the parent entity is persisted into the persistence context, the related entity will also be persisted.
REFRESH	If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.
REMOVE	If the parent entity is removed from the current persistence context, the related entity will also be removed.

Ex : @OneToMany(**cascade=REMOVE**, mappedBy="customer")
public Set<Order> getOrders() { return orders; }

sourcemind

JPA relations : one-to-one

```
@Entity
@Table(name = "jpa03_hb_personne")
 public class Personne {
      ...

     @OneToOne(cascade = CascadeType.ALL, fetch=FetchType.LAZY)
      /*@OneToOne(cascade = {CascadeType.MERGE,        CascadeType.PERSIST,
       CascadeType.REFRESH, CascadeType.REMOVE}, fetch=FetchType.LAZY) */
     @JoinColumn(name = "adresse_id", unique = true, nullable = false)
      private Adresse adresse;
      ...

}
```

```
@Entity
@Table(name = "jpa03_hb_adresse")
public class Adresse{
     ...
     /*n'est pas obligatoire*/
    @OneToOne(mappedBy = "adresse", fetch=FetchType.EAGER)
     private Personne personne;
      ...

}
```

sourcemind

JPA relations : one-to-many / many-to-one

```
@Entity
@Table(name="jpa05_hb_article")
 public class Article{
      ...
    // relation principale Article (many) -> Category (one) implémentée par une clé
    // étrangère (categorie_id) dans Article
    // 1 Article a nécessairement 1 Categorie (nullable=false)

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name = "categorie_id", nullable = false)
     private Categorie categorie;
     ...
}
```

```
@Entity
@Table(name="jpa05_hb_categorie")
public class Categorie{
    ...
   // relation inverse Categorie (one) -> Article (many) de la relation Article (many)– > Categorie (one)
   @OneToMany(mappedBy = "categorie", cascade = { CascadeType.ALL })
    private Set<Article> articles = new HashSet<Article>();
    ...

}
```

sourcemind

JPA relations : one-to-many / many-to-one

```
@Entity
@Table(name="jpa05_hb_article")
 public class Article{
      ...
    // relation principale Article (many) -> Category (one) implémentée par une clé
    // étrangère (categorie_id) dans Article
    // 1 Article a nécessairement 1 Categorie (nullable=false)

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name = "categorie_id", nullable = false)
     private Categorie categorie;
     ...
}
```

```
@Entity
@Table(name="jpa05_hb_categorie")
public class Categorie{
    ...
   // relation inverse Categorie (one) -> Article (many) de la relation Article (many)– > Categorie (one)
   @OneToMany(mappedBy = "categorie", cascade = { CascadeType.ALL })
    private Set<Article> articles = new HashSet<Article>();
    ...

}
```

sourcemind

JPA relations : many-to-many

```
@Entity
@Table(name = "jpa07_hb_personne")
 public class Personne {
      ...
    // relation Personne (many) -> Activite (many) via une table de jointure
      personne_activite
    @ManyToMany(cascade={CascadeType.PERSIST})
    @JoinTable(name="jpa07_hb_personne_activite",
                    joinColumns = @JoinColumn(name = "PERSONNE_ID"),
                    inverseJoinColumns = @JoinColumn(name =
"ACTIVITE_ID"))

      private Set<Activite> activites = new HashSet<Activite>();
       ...

}
```

```
@Entity
@Table(name = "jpa07_hb_activite")
public class Activite{
    ...

    // relation inverse Activite -> Personne
    @ManyToMany(mappedBy = "activites")
    private Set<Personne> personnes = new HashSet<Personne>();
...

}
```

sourcemind

JPQL : Java Persistence Query Language

- **JPQL** is a powerful query language that allows you to define database queries **based on your entity model**. Its structure and syntax are very similar to SQL
- JPQL uses the entity object model instead of database tables to define a query
- Hibernate, or any other JPA implementation, has to **transform the JPQL query into SQL**

https://thorben-janssen.com/jpql/

JPQL : The FROM clause

SELECT a FROM Author a

We reference the **Author entity** instead of the **author table** and assign the identification **variable a** to it. The identification variable is often called alias and is similar to a variable in your Java code. It is used in all other parts of the query to reference this entity.

JPQL : Inner Joins

SELECT a, b FROM Author a JOIN a.books b

The definition of the **Author entity** provides all information needed to join it to the **Book entity**, and you don’t have to provide an additional **ON statement**. In this example, JPA implementation uses the primary keys of the Author and Book entity to join them via the association table of the many-to-many association.

JPQL : Left Outer Joins / Right Outer Joins

```
SELECT a, b FROM Author a LEFT JOIN a.books b
```

```
SELECT a, b FROM Author a RIGHT JOIN a.books b
```

Inheritance mapping strategies

- Inheritance is one of the key concepts in Java, and it’s used in most domain models.
- You can choose between 4 strategies that map the inheritance structure of your domain model to different table structures. Each of these strategies has its advantages and disadvantages.
  - **Mapped Superclass**
  - **Table per Class**
  - **Single Table**
  - **Joined**

Inheritance mapping strategies

- **MappedSuperclass** – the parent classes, can’t be entities
- **Single Table** – The entities from different classes with a common ancestor are placed in a single table.
- **Joined Table** – Each class has its table, and querying a subclass entity requires joining the tables.
- **Table per Class** – All the properties of a class are in its table, so no join is required.

Mapped Superclass

```java
@MappedSuperclass
public abstract class Publication {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = “id”, updatable = false, nullable = false)
    protected Long id;

    @Column
    protected String title;

    @Version
    @Column(name = “version”)
    private int version;

    @Column
    @Temporal(TemporalType.DATE)
    private Date publishingDate;

    …
}
```

```java
@Entity(name = “Book”)
public class Book extends Publication {

    @Column
    private int pages;

    …
}
```

```java
@Entity(name = “BlogPost”)
public class BlogPost extends Publication {

    @Column
    private String url;

    …
}
```

Table per Class

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Publication {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = “id”, updatable = false, nullable = false)
    protected Long id;

    @Column
    protected String title;

    @Version
    @Column(name = “version”)
    private int version;

    @ManyToMany
    @JoinTable(name = “PublicationAuthor”, joinColumns = { @JoinColumn(name =
“publicationId”, referencedColumnName = “id”) }, inverseJoinColumns =
{ @JoinColumn(name = “authorId”, referencedColumnName = “id”) })
    private Set authors = new HashSet();

    @Column
    @Temporal(TemporalType.DATE)
    private Date publishingDate;

    …
}
```

```
@Entity(name = “Book”)
public class Book extends Publication {

    @Column
    private int pages;

    …
}
```

```
@Entity(name = “BlogPost”)
public class BlogPost extends Publication {

    @Column
    private String url;

    …
}
```

sourcemind

Single Table

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = “Publication_Type”)
public abstract class Publication {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = “id”, updatable = false, nullable = false)
    protected Long id;

    @Column
    protected String title;

    @Version
    @Column(name = “version”)
    private int version;

    @ManyToMany
    @JoinTable(name = “PublicationAuthor”, joinColumns = { @JoinColumn(name =
“publicationId”, referencedColumnName = “id”) }, inverseJoinColumns =
{ @JoinColumn(name = “authorId”, referencedColumnName = “id”) })
    private Set authors = new HashSet();

    @Column
    @Temporal(TemporalType.DATE)
    private Date publishingDate;

    …
}
```

```
@Entity(name = “Book”)
@DiscriminatorValue(“Book”)
public class Book extends Publication {

    @Column
    private int pages;

    …
}
```

```
@Entity(name = “BlogPost”)
@DiscriminatorValue(“Blog”)
public class BlogPost extends Publication {

    @Column
    private String url;

    …
}
```

Joined

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Publication {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = “id”, updatable = false, nullable = false)
    protected Long id;

    @Column
    protected String title;

    @Version
    @Column(name = “version”)
    private int version;

    @ManyToMany
    @JoinTable(name = “PublicationAuthor”, joinColumns = { @JoinColumn(name
= “publicationId”, referencedColumnName = “id”) }, inverseJoinColumns =
{ @JoinColumn(name = “authorId”, referencedColumnName = “id”) })
    private Set authors = new HashSet();

    @Column
    @Temporal(TemporalType.DATE)
    private Date publishingDate;

    …
}
```

```
@Entity(name = “Book”)
public class Book extends Publication {

    @Column
    private int pages;

    …
}
```

```
@Entity(name = “BlogPost”)
public class BlogPost extends Publication {

    @Column
    private String url;

    …
}
```

sourcemind

Choosing a Strategy

- If you require the best performance and need to use polymorphic queries and relationships, you should choose the single table strategy. But be aware, that you can’t use not null constraints on subclass attributes which increase the risk of data inconsistencies.
- If data consistency is more important than performance and you need polymorphic queries and relationships, the joined strategy is probably your best option.
- If you don’t need polymorphic queries or relationships, the table per class strategy is most likely the best fit. It allows you to use constraints to ensure data consistency and provides an option of polymorphic queries.

Data sources

- The DataSource works as a factory for providing database connections. It is an alternative to the DriverManager facility. A datasource uses a URL along with username/password credentials to establish the database connection.

- In Java, a datasource implements the javax.sql.DataSource interface. This datasource will typically be registered with the JNDI service and can be discovered using its JNDI name.

Configure Spring Data JPA

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>${postgresql.version}</version>
</dependency>
```

```
spring.datasource.url=jdbc:postgresql://localhost:5432/test
spring.datasource.username=postgres
spring.datasource.password=postgres
```

End

sourcemind