Framework Spring Boot

Framework Spring

- *Application open source (Libre)*
- *Plugins et prise en charge de la plupart des technologies Web*
- *De nombreuses API utilitaires*
- *Facile à utiliser*
- *Extrêmement populaire*
- *Prise en charge de Java, Groovy et Kotlin*
- *Bon pour le cloud et les microservices*

Spring Boot ?

- *Spring Boot est un **module** **de** Spring Framework.*

Cela nous permet de créer une application autonome avec des configurations minimales ou nulles.

- ***Il est préférable de l'utiliser si nous voulons développer une simple application basée sur Spring ou des services REST.***
- ***Spring Boot modules:***

Spring Boot fournit divers modules qui améliorent l'efficacité globale de l'application et contribuent au développement rapide du projet:

- 1) JWT Security
- 2) Dev tools
- 3) Auto Config
- 4) Testing of Application

Pourquoi Spring Boot ?

- *Applications Spring autonomes*
- *Serveur Web intégré, pas de guerre*
- *Accélérer le développement d'applications Spring*
- *Configurer automatiquement Spring dans la mesure du possible*
- *Pas de code généré*
- *Déploiement facile*
- *Prêt pour la production*

Technologies supportées

- **Core** : Spring Security, JTA, Spring Cache, Spring Session
- **Web** : Spring MVC, Websocket, Jersey, Mobile, HATEOS
- **Moteurs de templates** : Freemaker, Thymeleaf, Groovy, Mustache
- **Database/**
  - **SGBDR** : Spring Data JPA, JDBC, JOOQ
  - **NoSQL** : Redis, MongoDB, Elasticsearch, Cassandra
- La suite **Spring Cloud** : Eureka, Hystrix, Turbine, AWS, OAuth2
- **I/O** : Spring Batch et Integration, JavaMail, Camel, JMS, AMQP
- **Social** : Facebook, Linkedin, Twitter

Auto configuration

```
@SpringBootApplication
public class DemoApplication {
public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args); }
}
```

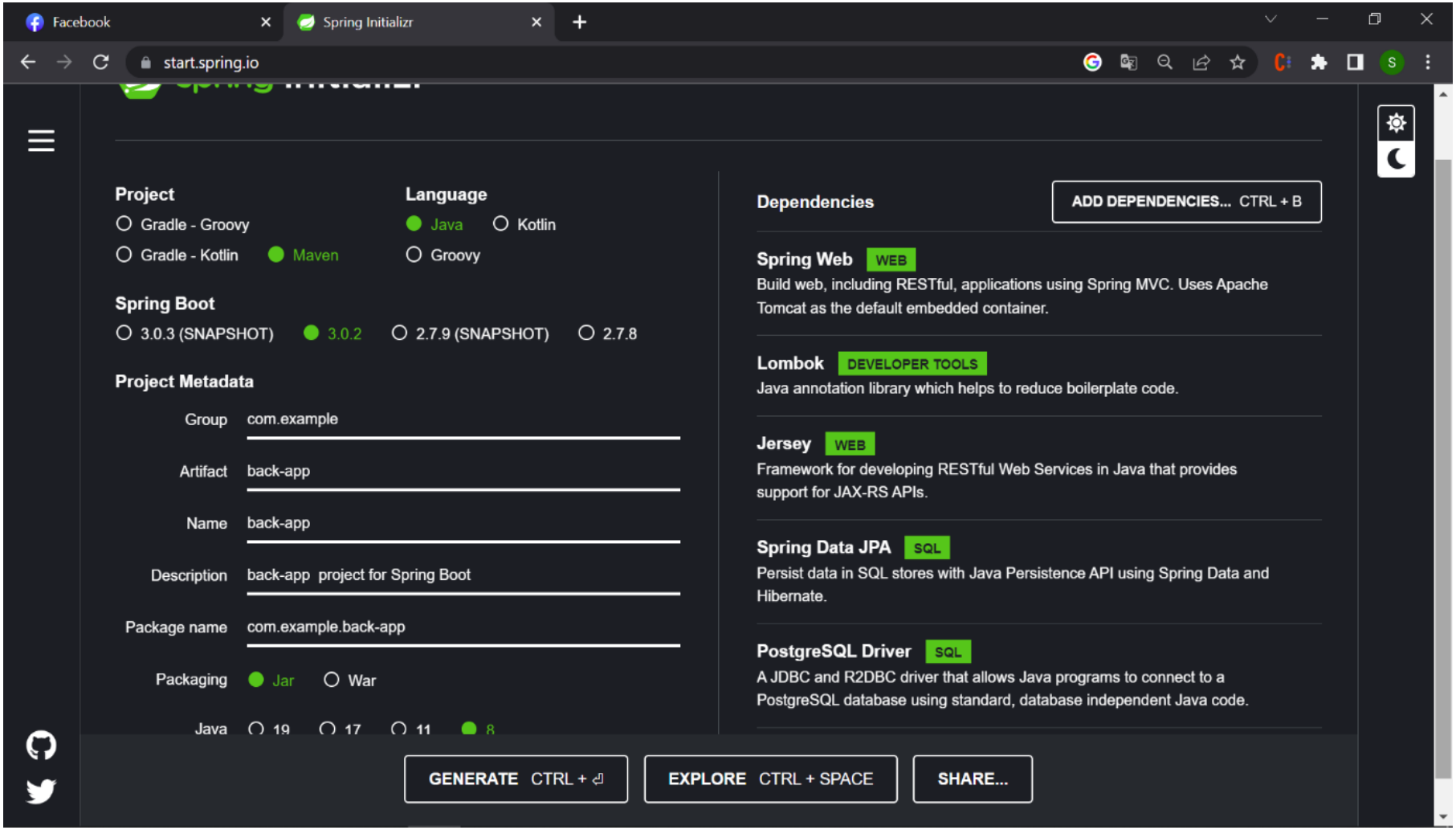
- L'annotation ***@SpringBootApplication*** déclenche la configuration automatique de l'infrastructure Spring
- Au démarrage de l'application, Spring Boot :
  - Scanne toutes classes de ***@Configuration***
    - Classes de configuration spécifiques à l'application
    - Classes de Spring Boot suffixées par ***AutoConfiguration***
  - Utilise les **JAR** présents dans le classpath pour prendre des décisions

Spring STS

Spring Tool Suite (sts)

Download https://spring.io/tools/sts/all

Spring Boot


Facebook
Spring Initializr
start.spring.io
Project
Gradle - Groovy
Gradle - Kotlin
Maven
Language
Java
Kotlin
Groovy
Spring Boot
3.0.3 (SNAPSHOT)
3.0.2
2.7.9 (SNAPSHOT)
2.7.8
Project Metadata
Group
com.example
Artifact
back-app
Name
back-app
Description
back-app project for Spring Boot
Package name
com.example.back-app
Packaging
Jar
War
Java
19
17
11
8
Dependencies
ADD DEPENDENCIES... CTRL + B
Spring Web
WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
Lombok
DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.
Jersey
WEB
Framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs.
Spring Data JPA
SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
PostgreSQL Driver
SQL
A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.
GENERATE CTRL + ⏎
EXPLORE CTRL + SPACE
SHARE...

Maven

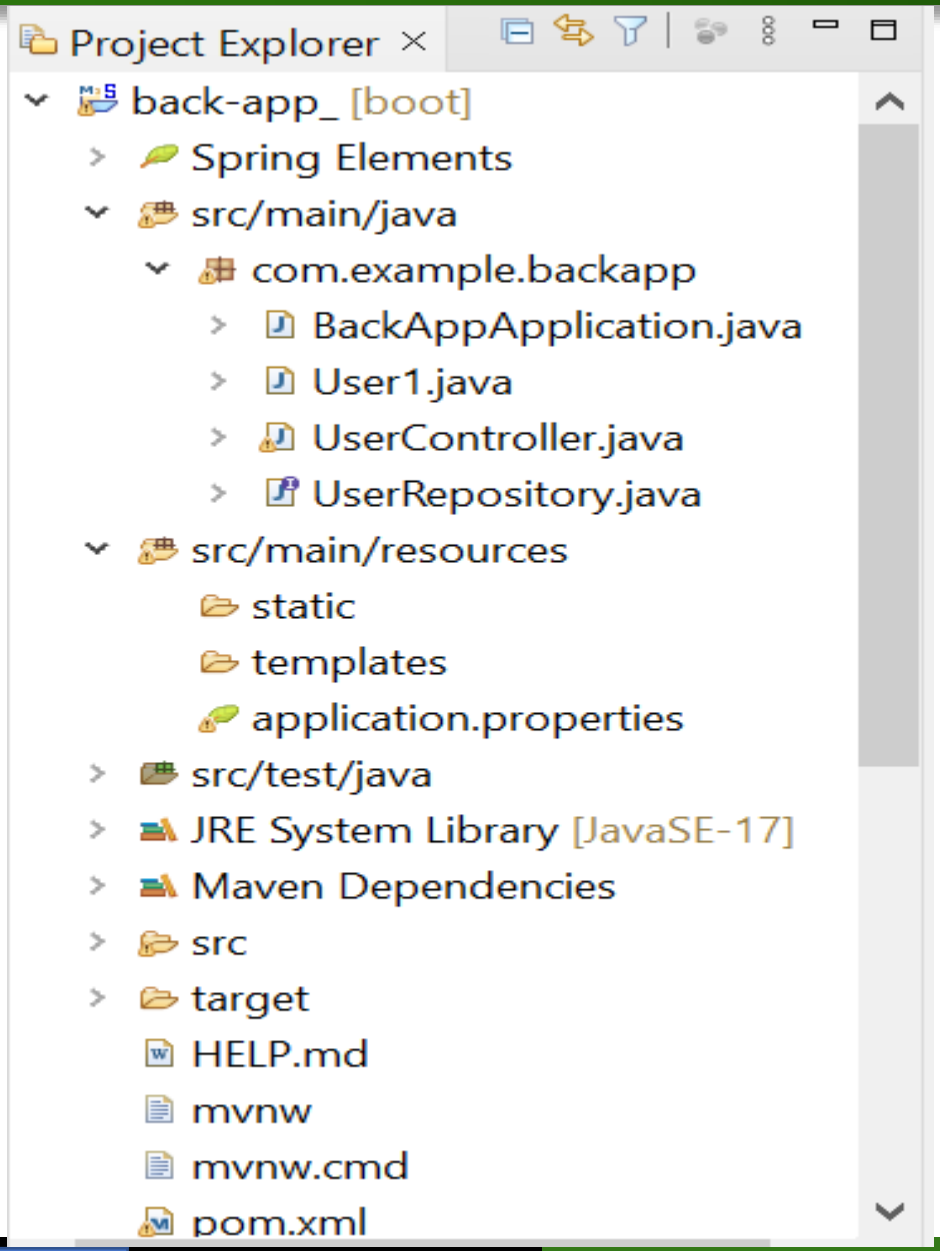
- Projet défini dans une fichier POM pom.xml
- Arborescence de projet type
- Définition des dépendances entre projets
  - <dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymock</artifactId>
    <version>3.1</version>
    <scope>test</scope>
    </dependency>
- Commandes de base :
  - mvn clean, compil, package, install, tomcat:deploy
- Utilisable depuis Eclipse

- Suite sur le support de cours "introMaven" de C. Dumoulin

Etude de cas : Backend spring Boot

Gestion des utilisateurs (suite)

Structure du Projet (1)



Structure du Projet (2)

– ***User 1*** *:classe de modèle de données des* ***utilisateurs*** *correspond aux utilisateurs d'entité et de table.*

– ***UserRepository*** *: est une interface qui étend* ***JpaRepository*** *pour les méthodes* ***CRUD*** *et les méthodes de recherche personnalisées. Il sera câblé automatiquement dans* ***UserController****.*

– ***UserController*** *est un* ***RestController*** *qui a des méthodes de mappage de requêtes pour les requêtes* ***RESTful*** *telles que :* ***getAllUsers, createUser, updateUser, deleteUser, findById…***

– ***application.properties :*** *Configuration pour Spring* ***Datasource****,* ***JPA*** *&* ***Hibernate****.*

– ***pom.xml*** *contient des dépendances pour Spring Boot et la base de données H2.*

Configuration du dépendance fichier pom.xml

*back-app_/pom.xml

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
            <version>3.0.2</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jersey</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
```

Configuration : Datasource, JPA, Hibernate (1)

fichier application.properties

application.properties

```
1 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
2 spring.jpa.hibernate.ddl-auto= update
3 spring.jpa.hibernate.show-sql=true
4 spring.datasource.url=jdbc:postgresql://localhost:5432/user
5 spring.datasource.username=postgres
6 spring.datasource.password=1234
```

Configuration : Datasource, JPA, Hibernate (2)

- **spring.datasource.url** : URL de la base de données.
- **spring.datasource.username** et **spring.datasource.password** sont identiques à celles de votre installation de base de données.

- Spring Boot utilise Hibernate pour l'implémentation de JPA, nous configurons **Dialect** pour la base de données

- **spring.jpa.hibernate.ddl-auto : (create, update ou validate)**
  - Est utilisé pour l'initialisation de la base de données (**create**).
  - Nous définissons la valeur « **update »**pour mettre à jour la base de données correspondant au modèle de données défini.
  - Pour la production, cette propriété doit être validée (**valide**).

Modèle de données (1)

User1.java

```java
@Entity
@Table(name = "user1")
public class User1 {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long numUser;

    @Column(name = "user_name")
    private String userName;

    @Column(name = "nom")
    private String nom;

    @Column(name = "prnom")
    private String prnom;

    @Column(name = "date_de_naissance")
    private Date dateDeNaissance;

    @Column(name = "lieu_de_naissance")
    private String lieuDeNaissance;

    @Column(name = "mot_passe")
    private String motPasse;

    @Column(name = "role")
    private int role;
```

Modèle de données (2)

- **Les annotation suivantes:**

@Entity: indique que la classe est une classe Java persistante.

@Table : fournit la table qui mappe cette entité**.**

@Id : concerne la clé primaire**.**

@GeneratedValue: est utilisée pour définir la stratégie de génération de la clé primaire**.**

GenerationType.AUTO : signifie champ d'incrémentation automatique**.**

@Column est utilisée pour définir la colonne dans la base de données qui mappe le champ annoté.

Repository Interface

UserRepository.java

```java
package com.example.backapp;

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

@Repository
public interface UserRepository extends JpaRepository<User1, Long> {
}
```

- *Permet l'interaction avec la Base de données*

nous pouvons utiliser les méthodes de **JpaRepository : save(), findOne(), findById(), findAll(), count(), delete(), deleteById()…** sans implémenter ces méthodes**.**

Nous pouvons définir également des méthodes de recherche personnalisées :– **findByName(string nom) : renvoie tous les utilisateurs dont la valeur = « nom».**

– findByNameContaining(string x) : renvoie tous les users dont le nom contient l'entrée x.

Spring Rest APIs Controller (1)

- *Le contrôleur fournit des API pour **créer**, **récupérer**, **mettre à jour**, **supprimer** et **trouver** des utilisateurs.*

– ***@CrossOrigin :*** *sert à configurer les origines autorisées.*

– ***@RestController :****est utilisée pour définir un contrôleur et pour indiquer que la valeur de retour des méthodes doit être liée au corps de la réponse Web.*

– ***@RequestMapping("/employees")*** *déclare que toutes les URL des API dans le contrôleur commenceront par : /employees.*

– *Nous utilisons **@Autowired** pour injecter le bean **UserRepository** dans la variable locale.*

Spring Rest APIs Controller (2)

UserController.java

```
14 @CrossOrigin(origins = "http://localhost:4200")
15 @RestController
16 @RequestMapping("employees")
17 public class UserController {
18
19     @Autowired
20         private UserRepository UserRepository;
21
22         @GetMapping
23         public ResponseEntity<List<User1>> getAllUsers() {
24             List<User1> returnUsers = UserRepository.findAll();
25             return new ResponseEntity<>(returnUsers, HttpStatus.OK);
26         }
27
28         @GetMapping("{id}")
29         public ResponseEntity<User1> getBlogPostById(@PathVariable Long id) {
30             if (UserRepository.existsById(id)) {
31                 User1 user1 = UserRepository.findById(id).get();
32                 return new ResponseEntity<>(user1, HttpStatus.OK);
33             }
34
35             return new ResponseEntity<>(HttpStatus.BAD_REQUEST);   }
36
37         @PostMapping
38         public ResponseEntity<User1> createUser(@RequestBody User1 user1) {
39             System.out.println("Je suis dans la procedure CreateUser! ");
40             UserRepository.save(user1);
41             return new ResponseEntity<>(HttpStatus.CREATED);
42         }
```

Persistance des données:

JPA/Hibernate
Spring Data

Framework Hibernate

1 Introduction:

2 Jakarta Persistence API

2 Object-Relational Mapping (ORM)

3 Entity Bean

4 Persistance des champs

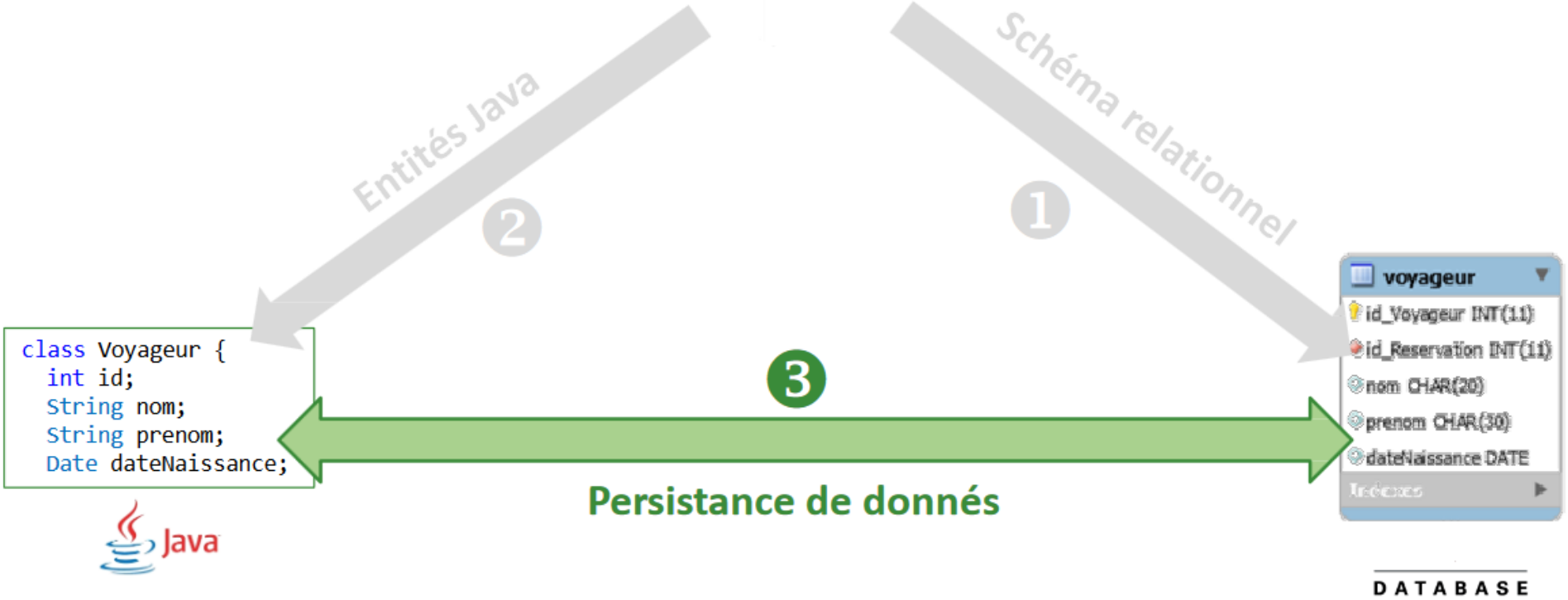
5 Hibernate DDL

6 Associations

7 Langage JPQL

Introduction : persistance de données?

- **Mapping Objet/Relationnelle**



Jakarta Persistence API (1)

- L'API Java Persistence (**JPA**) est une approche possible de l'**ORM (Object-Relational Mapping)**.

- **JPA** est une abstraction de la couche **JDBC,** les classes et les annotations **JPA** sont dans le package **javax.persistance (jakarta.persistence).**

- **JPA 1.0** a été introduite dans la spécification **JAVA EE 5,** la spécification **Jakarta EE 9 (eclipse)** supporte la version **JPA 3.0**

Jakarta Persistence API (2)

- **Composants:**

  – **ORM:** mécanisme de mapping **relationnel objet**
  – **L'API Entity Manager** qui gère les opérations **CRUD**
  – **JPQL :** langage de **requête orienté objet.**
  – **JTA (Jakarta Transaction API)** : Mécanisme de gestion des verrouillages et des transactions dans un environnement concurren

- Les **implémentations** populaires de JPA sont :
  - **Hibernate,**
  - **EclipseLink**
  - **Apache OpenJPA.**

Object-Relational Mapping

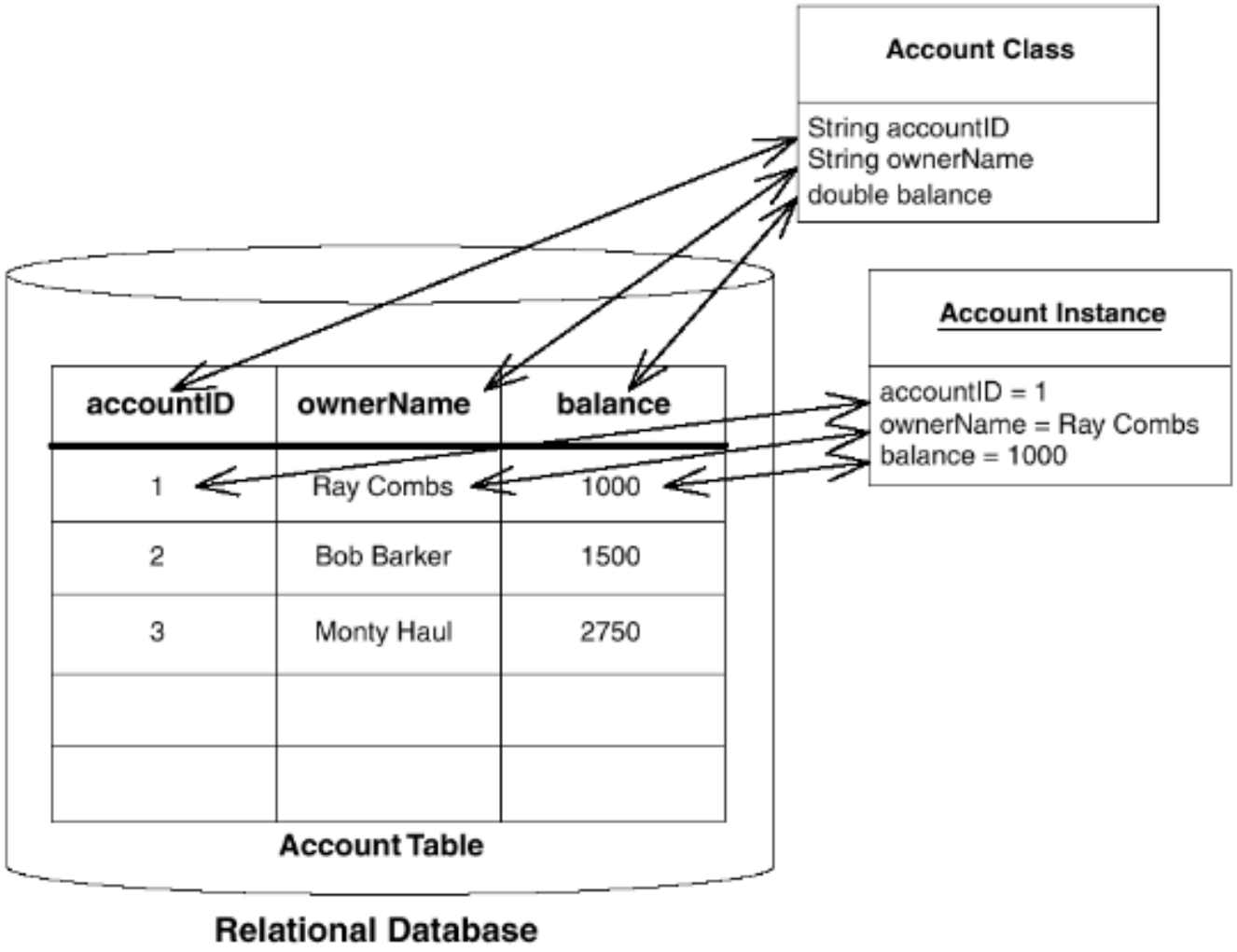
- Le processus de mappage des **objets Java** aux **tables** de base de données et vice versa est appelé « **Object-relationnal Mapping**" (**ORM**).

- On stocke l'**état** d'un **objet** dans une base de donnée.
  - Exemple :

    la *classe* **Personne** possède deux attributs `nom` et `prenom`, on associe cette classe à une **table** qui possède deux **colonnes** : **`nom`** et **`prenom`**.

- On décompose chaque **objet** en une suite de variables dont on ***stockera*** la valeur dans une ou plusieurs ***tables***.
- Permet des requêtes complexes.

Example compte bancaire



Qu'est-ce qu'un Entity Bean ? (1)

- Ce sont des **objets** qui savent **se mapper** dans une **base de donnée.**
- Ils utilisent un mécanisme de **persistance**
- Ils servent **à représenter** sous forme **d'objets** des **données** situées dans une **base de donnée**
- Le plus souvent **un objet = une** ou **plusieurs ligne(s)** dans **une** ou **plusieurs table(s)**

Qu'est-ce qu'un Entity Bean ? (2)

- Toutes les **classes d'entité** doivent :
  - définir une **clé primaire**,
  - avoir un **constructeur non arg** et/ou ne pas être autorisées à être **finales**.
  - Les **clés** peuvent être un **seul champ** ou une **combinaison** de champs.
- **JPA** permet de **générer** automatiquement la **clé primaire @Id** dans la base de données via l'annotation **@GeneratedValue**.
- Par défaut, le nom de la **table** correspond au nom de la **classe**. Vous pouvez changer cela avec l'ajout à l'annotation **@Table(name="NEWTABLENAME")**.
- **@Column**: pour personnaliser les attributs (par défaut le nom d'un champ dans la table est identique au nom de l'attribut)
- **@Transient** champ ne sera pas enregistré dans la **BD**

Exemple

```
package spring.cours.jpa.model;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Utilisateur {
@Id
private int id;
private String nom;

public int getId() {
return id;
}
public void setId(int id) {
this.id = id;
}

public String getNom() {
return nom;
}

public void setNom(String nom) {
this.nom = nom;
}

}
```

Propriétés d'une colonne

- @Column permet de déterminer les propriétés d'une colonne dans la table associée.
- Principales propriétés de l'annotation column :

Propriété	Valeur par défaut
name	Nom de l'attribut
nullable	True
Length (String)	255
unique	False

```
@Entity
@Table(name="users")
public class Utilisateur {
@Id
private int id;
@Column(nullable = false, name = "username", length = 60)
private String nom;
@Column(unique = true)
private String email;
// get, set
}
```

Hibernate DDL : configuration

La propriété **spring.jpa.hibernate.ddl-auto** définit le mode de génération de la base de données par Hibernate, valeurs:

– **create**

– **create-drop**: à chaque démarrage la base de données est créée puis supprimée.

– **update** : mise du schéma de la base de données (si c'est possibe)

– v**alidate** : valeur par défaut (à utiliser en production).

Mapping des Associations

La spécification **JPA** supporte trois types d'association :

@One to one

@Many to one / @One to Many

@Many to Many

Les trois types d'association peuvent être **unidirectionnelles** (navigables dans un sens) ou **bidirectionnelles** (navigables dans les deux sens), la navigabilité impacte l'utilisation des classes dans le programme et non pas les tables générées dans la base de données.

Association: @ManyToOne

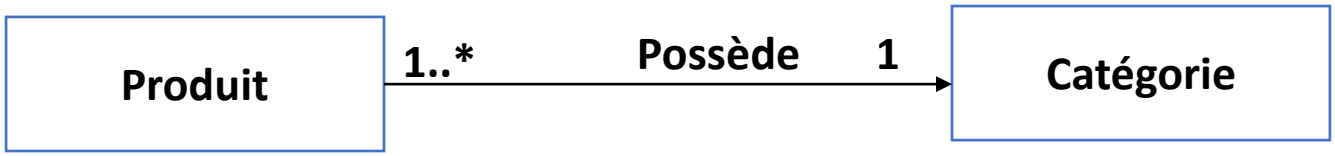
L'association **ManyToOne** est la plus utilisée dans la pratique, dans la classe **Produit** on ajoutera un attribut de type **Categorie**.

Categorie

```
@Entity

public class Categorie {

            @Id

            @GeneratedValue(strategy =
GenerationType.IDENTITY)

            private long catId;

            private String libelle;

…
```

Produit

```
@Entity

public class Produit {

@Id
@GeneratedValue(strategy=GenerationType
.IDENTITY)

private long reference;

private String designation;

private double prix;

@ManyToOne

private Categorie categorie;
```



Association: @OnetoMany

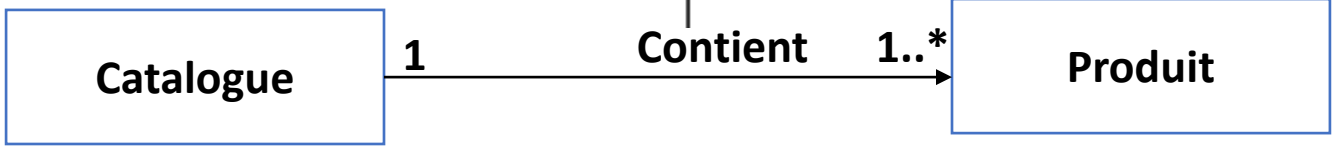
Navigabilité : Catalogue → Produit

pour généré une ***clé étrangère*** dans la table **Produit**, il faut ajouter l'annotation **JoinColumn** pour définir le nom de la clé étrangère dans la table produit :

Catalogue

```
@Entity
public class Catalogue {
            @Id
            @GeneratedValue(strategy =
GenerationType.IDENTITY)
            private long id;
            private String titre;
            @OneToMany
            private List<Produit>
produits=new ArrayList<>();
…}
```

```
@OneToMany
@JoinColumn(name="catalogue_id")
private List<Produit> produits=new ArrayList<>();
```



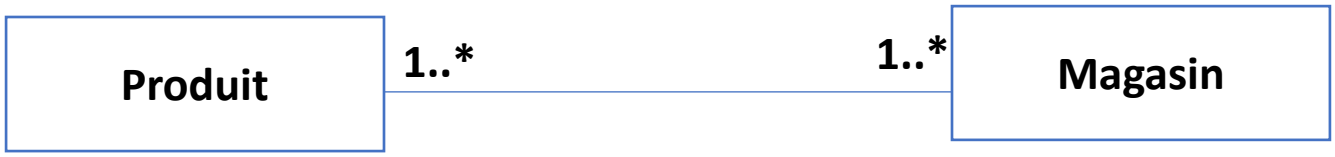
Association: @ManytoMany

- la navigabilité **Produit → Magasin**

@ManyToMany(mappedBy = "produits")

private List<Magasin> magasins=new ArrayList<>();

Ce qui permettra d'accéder aux magasins à partir d'un produit
p.getMagasins()



Langage JPQL

JPQL (Java Persistence Query Language) est un langage d'interrogation orienté objet similaire au langage SQL,
Structure d'une requête

Exemple de requêtes:

- SELECT li From Livre li
- SELECT li From Livre li Where li.titre='UML'
- SELECT li.titre, li.Editeur from livre li
- SELECT c.Adresse.pays from client c

Syntaxe d'une requête:

- SELECT
- FROM
- [WHERE]
- [ORDER BY]
- [GROUP BY]
- [HAVING]

Sélection selon une condition (JPA 2.0)

- SELECT CASE l.editeur WHEN 'Eyrolles"
- THEN l.prix * 0.5
- ELSE l.prix * 0.8
- END
- FROM Livre l

SQL	JPQL
SELECT * FROM produit WHERE prix > 100 ORDER BY designation;	**SELECT p** **FROM** Produit p **WHERE p.prix** > 100 **ORDER BY** p.designation