de Spring à Spring Boot

la persistance des données avec Spring Data

sommaire

la persistance

la persistance

la guerre des mondes

La persistance, c'est le mécanisme responsable de la sauvegarde et de la restauration des données dans un espace non volatile (fichiers texte, XML, bases de données...).

Des technologies très différentes existent dans un projet :

- **les langages de programmation objet** :
  - très répandus car ils ont fait leurs preuves
  - très outillés
- **les bases de données relationnelles** :
  - les + utilisées et conviennent bien aux données structurées
  - beaucoup de recul sur ces technologies
  - puissant langage de requêtes (SQL)
  - mais les énormes volumétries sont problématiques
  - mais nécessite une scalabilité verticale en cas de ressources insuffisantes
- **les bases de données No SQL (Not Only SQL)** :
  - en forte expansion
  - meilleure gestion des fortes volumétries
  - scalabilité horizontale + haute disponibilité
  - plutôt utilisées pour des données faiblement structurées, non relationnelles
  - chacune spécialisée dans un domaine (clé-valeur, document...).

la persistance

la guerre des mondes

Comment faire communiquer ces mondes ?

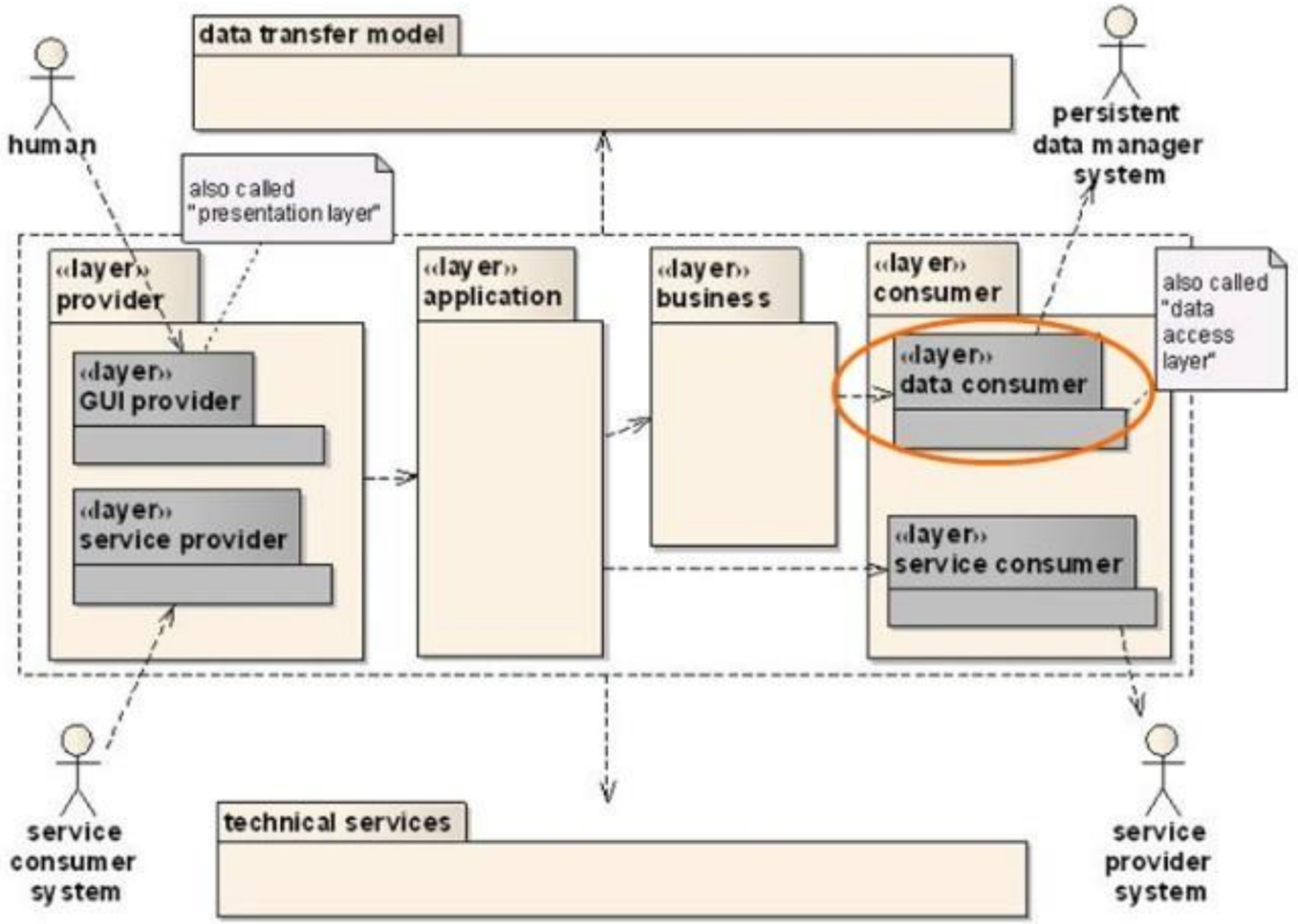
– **codage à la main des requêtes SQL** :
  - implémentations différentes d'une base à l'autre (ex : code SQL différent)
  - code difficile à tester

– **mapping objet/relationnel (ORM : Object Relational Mapping)** :
  - mise en relation des objets avec des tables
  - simplification de l'écriture de la couche d'accès aux données
  - abstraction des requêtes
  - utilisation de frameworks dédiés.

la couche d'accès aux données

la couche d'accès aux données

L'implémentation de la persistance doit être isolée dans la couche **d'accès aux données**.



la couche d'accès aux données

Elle s'appuie sur le **design pattern DAO (Data Access Object)** pour rendre la couche métier indépendante de la source de données sous-jacente (SQL, mapping O/R...) :

– exposition d'une interface indépendante de la technologie utilisée
– implémentation de la logique de persistance dans une technologie donnée
– échange de DTO (Data Transfer Object, ie. les concepts métiers) représentant le modèle objet persistant.

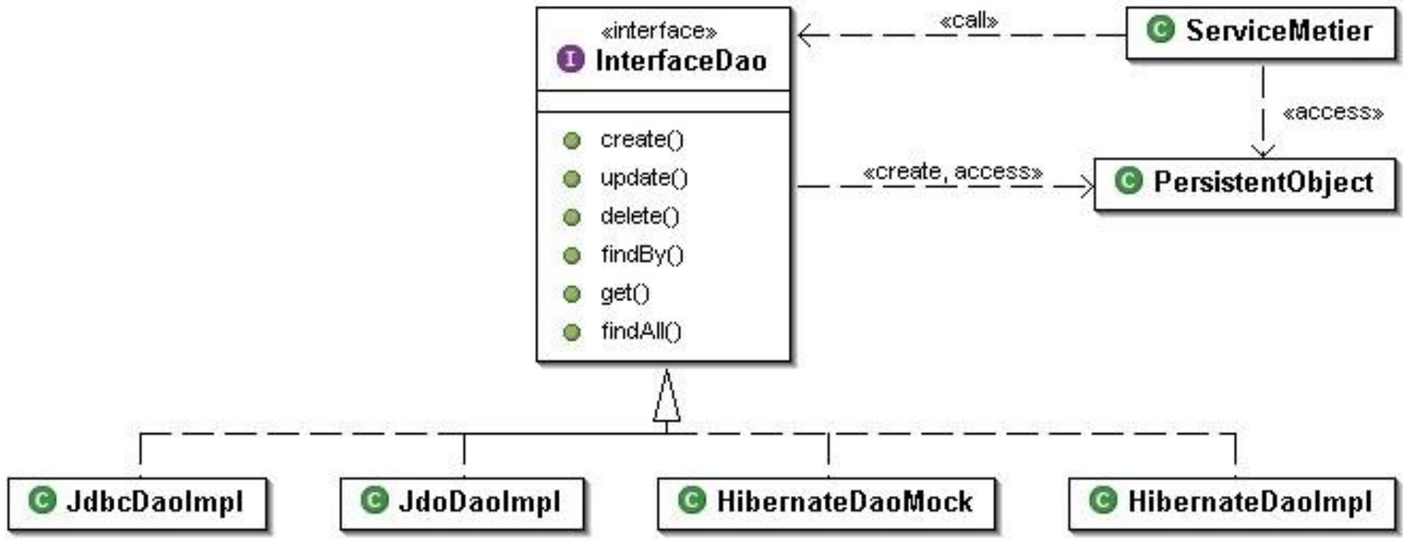
Le DAO propose généralement les méthodes **CRUD (Create/Read/Update/Delete)** suivantes :

– **get** : récupération d'un objet persistant grâce à son identifiant
– **create** : enregistrement d'un nouvel objet persistant
– **update** : mise à jour d'un objet persistant
– **delete** : suppression d'un objet persistant
– **findAll** : récupération de toutes les instances d'une classe persistante
– **findBy** : récupération des objets persistants selon des critères de sélection ou de tri.

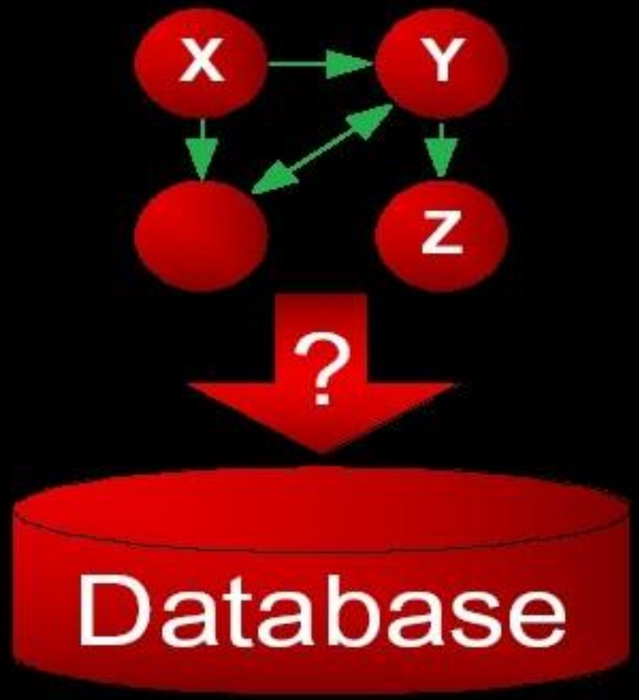
la couche d'accès aux données

L'interface du DAO expose les opérations disponibles pour chaque objet persistant.

L'implémentation permet de réaliser ces opérations dans plusieurs technologies, ou même de bouchonner à des fins de tests unitaires.



le mapping objet/relationnel



le mapping objet/relationnel

Il s'agit de définir les correspondances et règles de passage entre les objets et les tables, pour l'écriture, la suppression, la recherche...

> *Dans le cas général, 1 instance de classe = 1 enregistrement dans une table*

Il faut pour cela prendre en compte les différences entre les 2 modèles :

– l'organisation des données

> *objets/attributs <=> tables/colonnes*

– le typage des données

> *conversions, restrictions : String <=> varchar*

– l'unicité des objets

> *identifant (référence) en mémoire <=> clé primaire*

– relations entre objets/données

> *héritage, associations <=> tables de jointures, clés externes*

le mapping objet/relationnel

Les **apports** d'un framework de **mapping objet/relationnel** sont :

– code plus clair, moins verbeux
– pas de connaissance SQL nécessaire (mais utile quand même)
– code projet indépendant de la BDD sous-jacente
– migration facilitée d'une BDD à une autre
– gain de productivité
– fonctionnalités avancées configurables : cache, transactions, pagination...
– meilleures performances (en cas de bonne configuration)
– outillage : affichage des requêtes générées, reverse engineering...

Mais il y a quelques **désavantages** :

– nouveaux concepts à appréhender
– importants problèmes de performance en cas de mauvaise configuration
– peut s'avérer lourd à mettre en place pour de tous petits projets.

Hibernate est le framework de mapping objet/relationnel recommandé.

le mapping objet/relationnel

Voici les principales fonctions avancées apportées par l'ORM :

– gestion de **caches** d'objets persistants :
  - permet de minimiser le nombre de requêtes vers la source de données
– gestion de la **concurrence d'accès** :
  - verrouillage optimiste ou pessimiste des données
– optimisation du chargement des objets persistants :
  - à la demande (**lazy loading**) pour ne pas charger d'objets inutiles
  - du graphe complet (**eager loading**) d'objets persistants pour éviter l'effet **N+1 Select.**

> ***Le choix entre lazy et eager loading est très dépendant du cas d'utilisation.*** *C'est un subtil mélange entre :*
>
> *– le nombre de requêtes*
> *– la quantité d'informations ramenées par requête*
> *– l'utilisation qui est faite de ces informations.*

JPA

JPA

JPA (Java Persistence API) est un standard de la plateforme JEE qui définit :

– des règles de correspondance entre objets Java et tables de BDD relationnelles
– les interfaces (l'API) à respecter.

Les objets principalement manipulés sont les **Entity** : classes persistantes (POJO) avec des annotations JPA définissant les correspondances entre objets/attributs et tables/champs.

Tous les DAO manipulant chacun des objets persistents ont une grande partie de leur code qui est commune :

– se connecter à la BDD
– ouvrir une transaction
– écrire une requête sur cet objet persistent
– récupérer le résultat
– créer ou mettre à jour le bean Java
– commiter/rollbacker la transaction
– ...

JPA

Les frameworks de mapping O/R (Hibernate...) :

– implémentent les interfaces de l'API JPA
– ajoutent parfois des fonctionnalités propres.

Avantages :

– le code commun aux DAO est fourni (la couche d'abstraction)
– en cas de changement de framework (pour des raisons de performances, de roadmap...) :
  - changer simplement la librairie utilisée (**pom.xml** Maven) par une autre respectant le standard JPA
  - le code applicatif s'appuyant sur JPA reste inchangé.

JPA

paramétrage avec Spring Boot

Par défaut, Spring Boot offre un paramétrage de la persistance avec une configuration respectant les bonnes pratiques en vigueur.

Pour affiner ce paramétrage, on peut ajouter des informations dans le fichier **application.properties** (ou **application.yml**) :

```
spring.jpa.show-sql: true

# PostgreSQL parameters
spring.datasource.url = jdbc:postgresql://localhost:port/...
spring.datasource.username = <myUsername>
spring.datasource.password = <myPwd>
spring.datasource.driver-class-name = org.postgresql.Driver
```

D'autres **paramètres** sont disponibles.

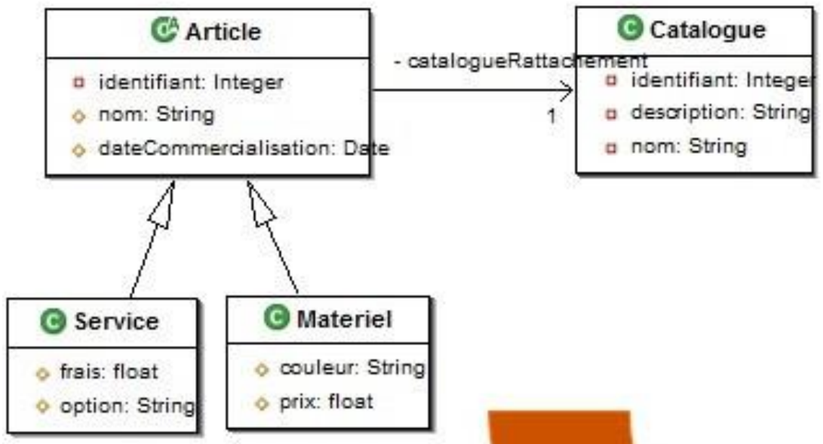
> *Note : en dehors d'une application Spring Boot, il faut configurer la persistence dans un fichier **persistence.xml**, y compris les préconfigurations qui sont offertes de base par Spring Boot (voir en annexes).*

JPA

exemple de mapping O/R

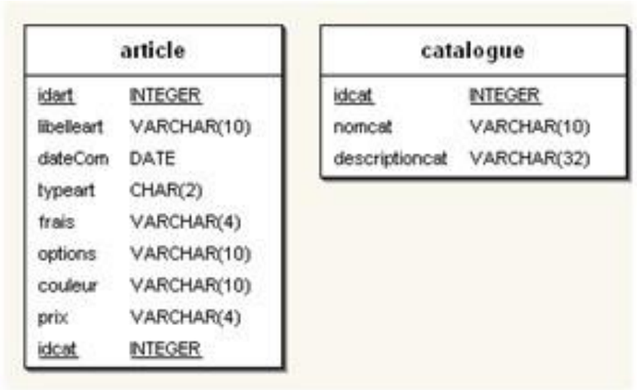
modèle objet

4 classes avec un héritage
et une association



modèle relationnel

2 tables dont une avec une clé étrangère



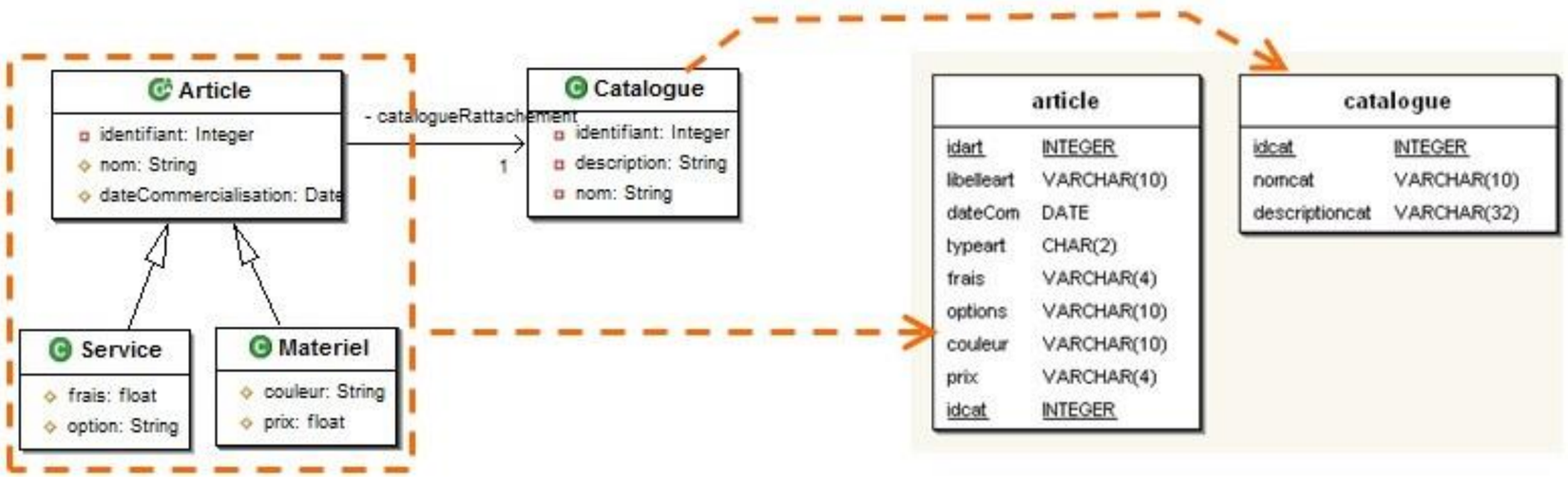
JPA

1/ faire correspondre les classes et les tables

– 1 classe <=> 1 table dans les cas classiques, mais ce n'est pas systématique, par exemple pour représenter un **héritage.**

En base de données, l'héritage peut être représenté de différentes façons :

– 1 table pour la hiérarchie complète, avec un champ discriminant et des colonnes vides
– 1 table par classe concrète avec union : la clé *idart* est partagée par les 2 tables *Service* et *Materiel*
– 1 table pour chaque classe de la hiérarchie si beaucoup d'attributs (et peu en commun).



JPA

1/ faire correspondre les classes et les tables

Avec JPA, grâce aux annotations :

- **@Entity** : indique qu'il s'agit d'une entité (classe) persistante
- **@Table** (optionnel) : détermine le nom de la table s'il n'est pas le même que celui de la classe.

```
public class Catalogue implements Serializable {

  private int identifiant;
  private String nom;
  private String description;
  private List<Article> articles;

  // getters and setters
  ...
}
```

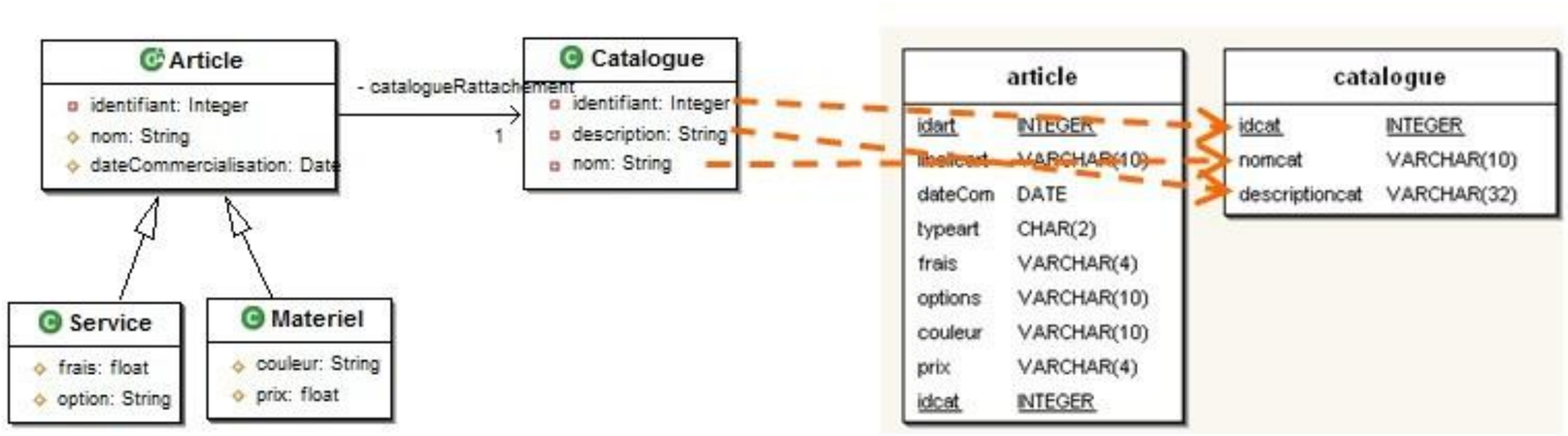
JPA

2/ faire correspondre les attributs simples et les colonnes

Les attributs simples sont ceux qui ne référencent pas d'autres objets persistants.

Pour chacun d'eux, des conversions seront appliquées entre le type Java et le type SQL.

Pour le cas particulier de la clé primaire, il faut indiquer comment elle est générée.



JPA

2/ faire correspondre les attributs simples et les colonnes

Avec JPA, grâce aux annotations :

- **@Id** : indique que l'attribut joue le rôle de clé primaire (simple). Selon la spécification JPA, chaque entité doit avoir un identifiant unique
- **@GeneratedValue** : indique la stratégie de génération de la clé primaire (séquence Oracle...)
- **@Column** (optionnel) : détermine le nom de la colonne s'il n'est pas le même que l'attribut de la classe.

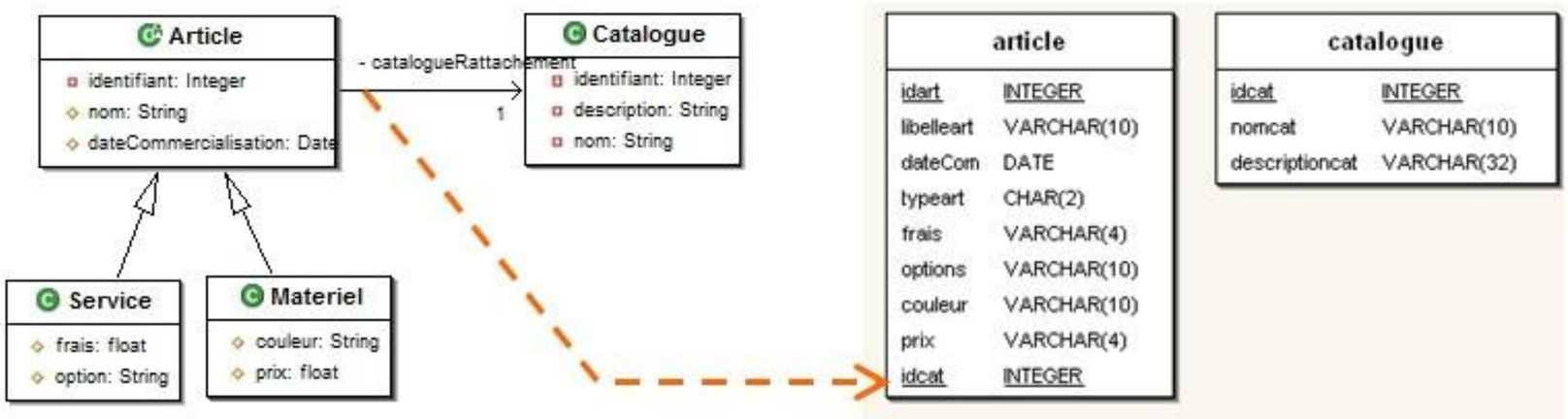
```
@Entity
public class Catalogue implements Serializable {




    private int identifiant;


    private String nom;
  ...
}
```

> ***Attention** : **nullable=false, unique=true** ne sont utiles que si on génère le schéma de BDD à partir du code. Au runtime, ces contraintes sur les champs sont vérifiées par la BDD, pas par l'application.*

JPA

3/ définir les attributs de relation

– indiquer la multiplicité (1-1, 1-N, N-N...)
– indiquer si la relation est uni-directionnelle ou bi-directionnelle suivant les parcours possibles dans les données
– définir si nécessaire une table de jointure
– lister la (les) clé(s) étrangère(s) représentant la cible des attributs de relation.



JPA

3/ définir les attributs de relation

Avec JPA, grâce aux annotations :

- **@OneToMany** (1-N)
- **@ManyToOne** (N-1)
- **@OneToOne** (1-1)
- **@ManyToMany** (N-N)

```
@Entity
public class Catalogue implements Serializable {
    ...
    @OneToMany(mappedBy="catalogue")    // navigation Catalogue -> collection d'Articles
    private List<Article> articles;
    ...
}

@Entity
public class Article implements Serializable {
    ...
    @ManyToOne                          // navigation Article -> Catalogue
    @JoinColumn(name="idcat")    // inutile si la clé étrangère de la table a le même nom que l'attribut
    protected Catalogue catalogue;
    ...
}
```

JPA

3/ définir les attributs de relation : paramétrage

Ces annotations peuvent être complétées de paramètres importants :

– **cascade** : propagation des opérations (**persist**, **delete**...) en cascade aux entités en relation (pas disponible pour **@ManyToOne**)
– **orphanRemoval** : une entité peut-elle exister sans père ? (disponible seulement pour **@OneToOne** et **@OneToMany**)
–**fetch** : chargement de la relation à la demande (*lazy loading*, choix par défaut pour les relations *XxxToMany*) ou immédiate (*eager loading*, choix par défaut pour les relations *XxsToOne*).

```
@Entity
public class Catalogue implements Serializable {
    ...
    @OneToMany(mappedBy="catalogue", fetch=FetchType.LAZY, cascade=CascadeType.ALL, orphanRemoval=true)
    private List<Article> articles;
    ...
}

@Entity
public class Article implements Serializable {
    ...
    @ManyToOne(fetch=FetchType.EAGER)
    @JoinColumn(name="idcat")
    protected Catalogue catalogue;
    ...
}
```

JPA

3/ définir les attributs de relation : conseils pour les relations bidirectionnelles

En création et/ou modification, Hibernate :

– se charge de la **cohérence des données dans la base de données**
– laisse à la charge de l'**application** le maintien de la **cohérence en mémoire** des objets Java.

Le problème peut survenir sur les **relations bidirectionnelles** notamment. Si la grappe d'objets en relation est créée/modifiée et persistée :

– en 1 fois, aucun problème de cohérence
– en plusieurs étapes, des incohérences peuvent apparaître côté Java.

Pour éviter cela :

– **créer des méthodes (addXxx(), removeXxx())** sur la classe persistante propriétaire de la relation
– elles se chargeront de maintenir la cohérence (niveau Java) de la relation dans les 2 sens.

JPA

3/ définir les attributs de relation : conseils pour les relations bidirectionnelles

*Exemple : relation bidirectionnelle entre un **Catalogue** et ses **Articles** :*

```
@Entity
public class Catalogue {

    @OneToMany(mappedBy="catalogue")
    private List<Article> articles = new ArrayList<Article>();

    public void addArticle(Article article) {
        articles.add(article);          // ajouter un Article à la liste ci-dessus
        article.setCatalogue(this);                 // ET renseigner à quel Catalogue appartient
    }   cet Article

    public void removeArticle(Article article) {
        article.setCatalogue(null);     // supprimer le lien Article -> Catalogue
        for (int i=0; i<articles.size(); i++) {
            if (articles.get(i).getId() == article.getId()) {
                articles.remove(i);     // supprimer l'Article
            }
        }
    }
    ...
}
```

JPA

synchronisation avec la BDD

La synchronisation en base de données s'effectue à l'initiative :

– du moteur de persistance avant l'exécution d'une requête pour assurer la cohérence des données
– du moteur de persistance au moment du *commit*
– de l'application via la méthode **flush()**.

Le paramétrage de la synchronisation est possible :

– **FlushModeType.AUTO** : comme indiqué ci-dessus
– **FlushModeType.COMMIT** : seulement lors du commit ou du flush.

JPA

concurrence d'accès

Verrouillage pessimiste :

–la donnée est verrouillée (mise à jour impossible) tant que celle-ci est utilisée (en lecture, en écriture...) par un client. Tous les autres utilisateurs ne peuvent que la lire.

Verrouillage optimiste :

– la donnée n'est verrouillée que lors de sa modification effective en base de données (très rapide)
– avant toute modification :
-le moteur de persistance compare les versions de la donnée entre l'entité persistante (objet Java) et l'enregistrement en base de données
- si leur version est égale : mise à jour effectuée et la version de la donnée est incrémentée des 2 côtés
-sinon : la donnée a été modifiée entre son accès (sa lecture) et sa tentative de modification. Une **OptimisticLockException** est automatiquement levée.

> *La spécification JPA se positionne explicitement en faveur d'un **verrouillage optimiste** pour obtenir de meilleures performances.*

JPA

concurrence d'accès

Pour mettre en place un **verrouillage optimiste** :

– dans la table de la base de données : avoir une colonne **version**

–dans l'entité persistante : avoir un attribut **version** (donc mappé sur la colonne **version**) annoté **@Version**, uniquement manipulé par le moteur de persistance, qui se charge de la cohérence des données :

```
@Entity
public class Catalogue implements Serializable {
    ...


    ...
}
```

JPA

validation des données

Bean Validation (JSR 303) a pour objectif de standardiser la déclaration et la vérification des contraintes sur un modèle Java (avant l'appel de la BDD), via des annotations : **@NotNull, @Null, @NotEmpty, @Size, @Min, @Max, @Pattern...**

En cas d'échec de validation :

– une **ConstraintViolationException** est levée. La transaction est marquée *rollbackOnly* (ie : devant être annulée).
– il est possible de renseigner une clé d'un fichier **.properties** afin d'associer le message adéquat (et internationalisé) :

```
@NotEmpty(message = "NotEmpty.catalogue.nom")      // @NotEmpty = @NotNull + @NotBlank
private String nom;
```

Pour activer la validation automatique des contraintes sur les champs, il faut ajouter cette dépendance dans le fichier **pom.xml** :

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

JPA

gestion des caches

Il existe 3 types de caches :

– **cache de niveau 1** :
  - les entités utilisées sont mises en cache pour la durée d'une transaction
  - si elles sont modifiées plusieurs fois lors de la même transaction, un seul commit en base sera fait à la fin de la transaction
– **cache de niveau 2** :
  - partagé par toutes les transactions
  - à utiliser pour les données de référence qui n'évoluent pas (pas de synchronisation à faire)
  - annoter les classes éligibles avec **javax.persistence.@Cacheable(true)**
– **cache de requêtes** :
  - utile pour des requêtes très consommatrices/volumineuses et souvent utilisées
  - mise en cache de la requête (+ ses paramètres) et des **id** des entités retournées, mais :
  - un accès à la base sera fait pour accéder aux entités
  - à utiliser conjointement avec le cache de niveau 2, pour stocker les entités au-delà d'une transaction.

JPA

langages de requêtes

Il est possible d'écrire des requêtes en plusieurs langages :

– **JPQL (Java Persistence Query Language)** :

```
// récupérer le pays et sa capitale
Select country From Country country JOIN FETCH country.capital;
```

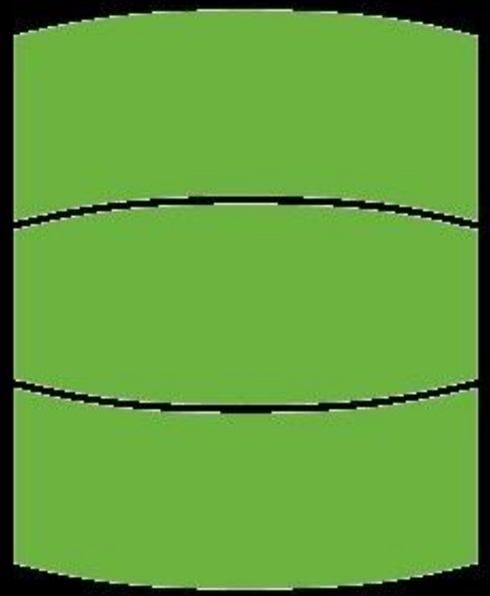
– **Query DSL (Domain Specific Language)** :

```
List<Market> markets = query.selectFrom(market)
  .where(market.name.eq("Entreprise"))
  .uniqueResult(market);
```

– **API Criteria** :

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Country> cq =  cb.createQuery(Country.class);
Root<Country> country = cq.from(Country.class);
cq.select(country);
TypedQuery<Country> tq = em.createQuery(cq);
List<Country> allCountries = tq.getResultList();
```

Spring Data



Spring Data

Spring Data est composé de nombreux modules spécialisés par technologie :

- Spring Data JPA : interactions avec les BDD relationnelles en s'appuyant sur **Hibernate**
- Spring Data REST
- Spring Data MongoDB
- Spring Data KeyValue
- ...

Spring Data offre une approche **Domain Driven Design (DDD)** de l'accès aux données, en s'appuyant sur le design pattern **Repository**.

Domain Driven Design (DDD) : approche partant d'une modélisation du domaine métier :

- définition de concepts métiers partagés par tous les acteurs du projet : développeurs, fonctionnels...
- précisions sur le comportement et les règles d'interactions entre ces concepts
- donner au code une conception orientée métier.

Pattern Repository : mapping entre les données du modèle métier et les données stockées, selon les technologies sous-jacentes (BDD relationnelles...).

Spring Data

repositories

Spring Data offre des repositories génériques et d'autres spécialisés en fonction de la source de données sous-jacente (JPA, MongoDB...).

Ainsi, Spring Data fournit une interface générique **Repository<T,ID>**, permettant de spécifier :

– le type d'entité (**T**) géré par ce repository
– le type de l'identifiant de cette entité (**ID**).

L'interface générique **CrudRepository<T,ID>** :

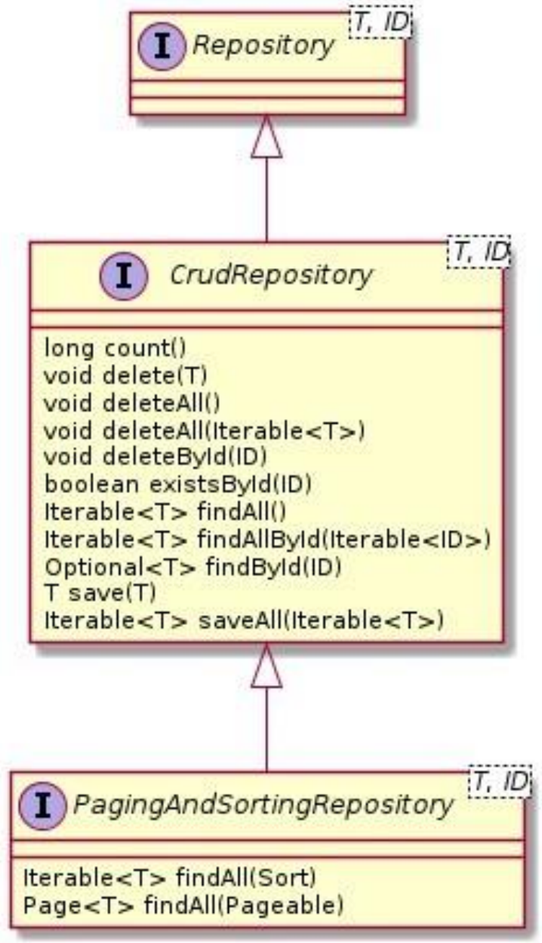
– étend **Repository<T,ID>**
– définit les signatures des méthodes CRUD (leur implémentation sera automatiquement générée par le framework).

L'interface générique **PagingAndSortingRepository<T,ID>** :

– étend **CrudRepository<T,ID>**
– ajoute des méthodes de pagination et de tri.

Spring Data

repositories



Spring Data

repositories

L'interface générique **QueryByExampleExecutor<T>** :

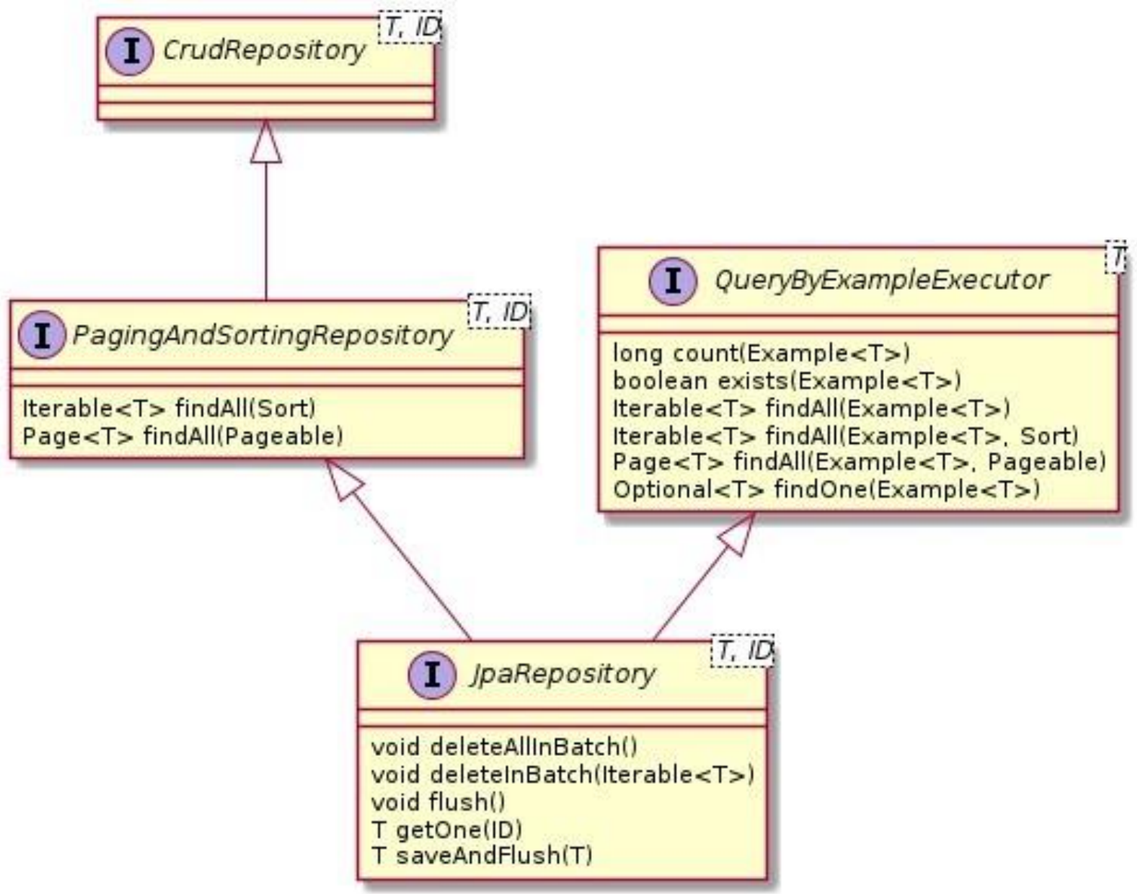
- ajoute des méthodes de recherche à partir d'un **Example<T>** passé en paramètre
- **Example<T>** est une instance de la classe d'entité qui sert de pattern de recherche.

L'interface **Spring Data JPA JpaRepository<T,ID>** :

- étend **PagingAndSortingRepository<T,ID>**
- étend **QueryByExampleExecutor<T>**
- ajoute des méthodes liées aux batchs ou de flush des données en BDD.

Spring Data

repositories



mise en oeuvre

mise en oeuvre

configuration Maven

Dans le fichier **pom.xml**, il faut d'abord avoir la dépendance vers le starter **spring-boot-starter-data-jpa** :

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

Cette dépendance permet l'utilisation des librairies liées au développement JPA (notamment Hibernate avec des propriétés configurées par défaut).

mise en oeuvre

utilisation d'une base de données mémoire

Le plus simple pour débuter un développement nécessitant l'usage d'une base de données, est d'utiliser une base de données mémoire. Cela évite d'installer une vraie base de données.

Elle sera chargée en mémoire au démarrage de l'application, et sera détruite à l'arrêt de celle-ci.

Par défaut, Spring Boot peut auto-configurer une base de données **H2**, **HSQL** ou **Derby**, du moment que la dépendance Maven est présente.

La dépendance Maven suivante permet d'utiliser la base mémoire **H2** :

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

mise en oeuvre

utilisation d'une base de données persistante

Par la suite (en intégration continue, en production...), il faut bien sûr s'appuyer sur une base de données persistante (MariaDB, MySQL, PostgreSQL, Oracle...).

Préciser dans le **pom.xml** quel type de base sera utilisée (ici, MariaDB) :

```
<dependency>
    <groupId>org.mariadb.jdbc</groupId>
    <artifactId>mariadb-java-client</artifactId>
    <scope>runtime</scope>
</dependency>
```

mise en oeuvre

utilisation d'une base de données persistante

Ensuite, en configurant (au moins) l'URL de la base de données dans le fichier **application.properties/yml**, Spring Boot considère qu'il ne doit pas utiliser une base de données mémoire, mais la base spécifiée :

```
spring.datasource.url = jdbc:mysql://${TODO_BACK_DB_SERVICE_HOST}:3306/${MY_DATABASE}?autoReconnect=true&useSSL=false
spring.datasource.userName = ${MY_USER}
spring.datasource.password = ${MY_PASSWORD}

spring.datasource.driver-class-name = org.mariadb.jdbc.Driver
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MariaDB10Dialect
```

En général, le *driver* à utiliser est facultatif, car Spring Boot saura le déduire à partir de l'URL.

mise en oeuvre

création d'un Repository standard

Pour accéder aux données, il suffit de créer **une interface** annotée **@Repository**, qui étend l'un des précédents repositories (**JpaRepository<T,ID>**...).

Ainsi, on peut :

– bénéficier automatiquement des méthodes héritées (ex : CRUD)
– ajouter de nouvelles signatures de méthodes respectant un nommage précis :

> ***findByXxx****... (**Xxx** étant le nom d'un attribut de la classe persistante)*

– utiliser certains **mots-clés** dans ces noms de méthodes :

> ***And, Or, Like, Containing, StartingWith, Exists, OrderBy, GreaterThan, Between, After****...*

– ajouter des **query methods** annotées **@Query**, en précisant la requête JPQL à exécuter :

Remarque : l'utilisation d'**Example<T>** est une autre façon d'accéder aux données sans devoir ajouter de nouvelles méthodes. Mais il faut quand même créer son repository.

mise en oeuvre

création d'un Repository standard

*Exemple d'un **Repository** de gestion de classe persistante **Person** :*

```
@Repository
public interface PersonRepository extends JpaRepository<Person, Long>
{
  // JpaRepository<Person, Long> fournit les méthodes CRUD.
  // Les méthodes suivantes n'ont pas besoin d'implémentation.

  // SELECT p FROM Person p where p.lastname='lastname'
  List<Person> findByLastname(String lastname);

  // SELECT p FROM Person p where p.lastname='lastname' order by...
  List<Person> findByLastname(String lastname, Sort sort);

  // SELECT p FROM Person p where p.lastname='lastname' and p.firstname='firstname'
  List<Person> findByLastnameAndFirstname(String lastname, String firstname);

  // SELECT p FROM Person p where p.lastname LIKE 'lastname'
  List<Person> findByLastnameContaining(String lastname);

  ...
```

mise en oeuvre

création d'un Repository standard

```
  ...

  // SELECT p From Person p where p.address.zipcode='zipcode'
  List<Person> findByAddress_ZipCode(String zipcode);

  // query method : définition du JPQL à exécuter par Spring Data
  @Query("SELECT COUNT(p) FROM Person p where p.lastname LIKE CONCAT('%',CONCAT(?1,'%'))")
  long countPersonWithLastnameContaining(String lastname);

  // query method avec des paramètres nommés
  @Query("SELECT p FROM Person p where p.lastname = :lastname)")
  List<Person> findByLastname(@Param("lastname") String lastname);

}
```

mise en oeuvre

création d'un Repository spécifique

Certains cas ne sont pas prévus par Spring Data (recherche multi-critères...), et les techniques vues précédemment ne suffisent pas.

Il est toujours possible de définir ses propres méthodes spécifiques, qu'il faut implémenter.

Pour que le tout s'articule bien dans Spring Data, il faut alors créer :

– une **interface spécifique** qui liste les signatures customisées
– un **Repository standard** (avec ou sans méthodes auto-implémentées) qui étend :
    - un repository standard Spring Data (**JpaRepository<T,ID>**...)
    - l'interface spécifique
– une **classe d'implémentation** pour l'interface spécifique :
    - annotée **@Repository**
    - qui a pour nom : **<nomInterfaceRepositoryStandard>Impl.java**.

mise en oeuvre

création d'un Repository spécifique

```
// interface spécifique
public interface PersonRepositoryCustom {
  // signatures de méthodes spécifiques
  ...
}
```

```
// repository standard
public interface PersonRepository extends JpaRepository<Person, Long>, PersonRepositoryCustom {

  // méthodes respectant les patterns attendus, ou query methods...
  ...

}
```

```
// classe d'implémentation nommée <repositoryStandard>Impl
@Repository
public class PersonRepositoryImpl implements PersonRepositoryCustom {
  // implementation des méthodes spécifiques
  ...
}
```

mise en oeuvre

utilisation des Example<T>

Un problème peut survenir avec la méthode précédente, lorsque l'entité persistante en question contient plusieurs attributs, et qu'on aimerait pouvoir faire des recherches sur un ou plusieurs attibuts combinés.

Le nombre de signatures à ajouter peut vite devenir grand et fastidieux à énumérer.

Une alternative puissante est de passer par les **Example<T>** (grâce à l'interface **QueryByExampleExecutor<T>**).

Dans la couche métier, il faut :

– créer/récupérer une instance de l'entité persistante, dont certains attributs sont renseignés (critères de recherche)
– ajouter des règles (**ExampleMatcher**) sur la recherche (containing, starting...)
– appeler l'une des méthodes de notre repository qui accepte un **Example<T>** en paramètre.

mise en oeuvre

utilisation des Example<T>

> *Exemple : imaginons qu'un utilisateur, via un formulaire de recherche, a saisi les critères de recherche suivants pour une **Person** :*

```
// Person(id, firstName, lastName, age, address)
Person person = new Person(null, "Mickael", null, 18, null);
```

> *Voici coment utiliser cet exemple de **Person** :*

```
public List<Person> findAllPersonsByExample(Person person) {

    // création des règles de recherche
    ExampleMatcher matcher = ExampleMatcher.matching()
          .withIgnorePaths("id", "lastname", "address")
          .withStringMatcher(ExampleMatcher.StringMatcher.CONTAINING);

    // création de l'Example avec les critères + les règles
    Example<Person> exampleOfPerson = Example.of(person, matcher);

    return repoPerson.findAll(exampleOfPerson);
}
```