
OXiane
INSTITUT
make it clever

25/04/2023
Version 4

SPRING

JÉRÉMY PERROUAULT

SPRING JDBC

SPRING & JDBC

PRÉSENTATION SPRING JDBC

- Template pour accéder à la base de données
  - *C CREATE*
  - *R READ*
  - *U UPDATE*
  - *D DELETE*
- Effectuer des requêtes **SQL**
- Mapper le résultat à un graphe d'objets

PRÉSENTATION SPRING JDBC

- Dépendances
  - **spring-jdbc**
    - Brique **JDBC**
  - **commons-dbcp2** OU **HikariCP**
    - L'un des deux pour configurer un gestionnaire de pool de connexions

CONFIGURATION

CONFIGURATION DE SPRING JPA

CONFIGURATION

- Utilisation d'un bean `DataSource`

- Création de la `DataSource` gérée par Spring
  - Avec le pilote à utiliser
  - L'URL de connexion à la base de données
  - Les identifiants de connexion
  - Le maximum de connexions simultanées actives (si utilisation d'un Pool de connexions)

CONFIGURATION

```
@Bean
public DataSource dataSource() {
  BasicDataSource dataSource = new BasicDataSource();

  dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
  dataSource.setUrl("jdbc:mysql://localhost:3306/db");
  dataSource.setUsername("root");
  dataSource.setPassword("");
  dataSource.setMaxTotal(10);

  return dataSource;
}
```

CONFIGURATION

```
@Bean
public DataSource dataSource() {
  SimpleDriverDataSource dataSource = new SimpleDriverDataSource();

  dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
  dataSource.setUrl("jdbc:mysql://localhost:3306/db");
  dataSource.setUsername("root");
  dataSource.setPassword("");

  return dataSource;
}
```

CONFIGURATION

- Annoter la classe de configuration de `@EnableTransactionManagement`
  - Pour activer l'annotation `@Transactional`

```
@Bean
public JpaTransactionManager transactionManager(EntityManagerFactory emf) {
  JpaTransactionManager transactionManager = new JpaTransactionManager();

  transactionManager.setEntityManagerFactory(emf);
  return transactionManager;
}
```

UTILISATION

- Déclaration du *bean* `Repository`
  - `@Repository` (ne pas oublier de scanner les packages dans la configuration **SPRING** !)
- Récupération de `DataSource` (injection de la dépendance)
  - `@Autowired`

UTILISATION

```
@Repository
public class ProduitRepositoryJdbc implements IProduitRepository {
  @Autowired
  private DataSource dataSource;

  public List<Produit> findAll() {
    //...
  }

  public Produit save(Produit entity) {
    //...
  }

  //...
}
```

EXERCICE

- Implémenter **SPRING JDBC**
  - Et sa configuration
- Créer les interfaces `Repository<T>` et `ProduitRepository`
  - `count()` `long`
  - `findAll()` `List<T>`
  - `findById(int id)` `Optional<T>`
  - `save(T entity)` `T`
  - `delete(T entity)` `void`
  - `deleteById(int id)` `void`
- Créer la classe `ProduitRepositoryJdbc`

UTILISATION

- Déclaration du *bean* `Repository`
  - `@Repository` (ne pas oublier de scanner les packages dans la configuration **SPRING** !)
- Configuration et récupération de la `DataSource` (injection de la dépendance)
  - `@Autowired`

REQUÊTAGE

- Pour effectuer une requête, utilisation des classes
  - `JdbcTemplate` Pour la partie requête
  - `NamedParameterJdbcTemplate` Pour la partie requête nommée
  - `RowMapper<T>` Pour la partie mapping des informations **SQL** / graphe d'objets
  - `MapSqlParameterSource` Pour affecter des paramètres nommés à une requête

REQUÊTAGE

```
public class ProduitRowMapper implements RowMapper<Produit> {
  public Produit mapRow(ResultSet resultSet, int index) {
    Produit myProduit = new Produit();

    myProduit.setId(resultSet.getInt("PRO_ID"));
    myProduit.setLibelle(resultSet.getString("PRO_LIBELLE"));

    return myProduit;
  }
}
```

REQUÊTAGE

```
public List<Produit> findAll() {
  String myQuery = "SELECT * FROM produit";

  return jdbcTemplate.query(myQuery, new ProduitRowMapper());
}
```

REQUÊTAGE

```
public Produit findById(int id) {
  String myQuery = "SELECT * FROM produit WHERE PRO_ID = :id";
  NamedParameterJdbcTemplate myJdbcTemplate =
    new NamedParameterJdbcTemplate(dataSource);
  MapSqlParameterSource myQueryParams = new MapSqlParameterSource();

  myQueryParams.addValue("id", id);

  return myJdbcTemplate
    .queryForObject(myQuery, myQueryParams, new ProduitRowMapper());
}
```

REQUÊTAGE

```
public Produit findById(int id) {
  String myQuery = "SELECT * FROM produit WHERE PRO_ID = ?";

  return jdbcTemplate
    .queryForObject(myQuery, new ProduitRowMapper(), id);
}
```

EXERCICE

- Configurer `JdbcTemplate` en tant que composant **SPRING**
  - Modifier la `Repositoy` en conséquence
- Modifier le programme `Application`
  - Lister les produits (afficher l'id, le libellé et le prix)
- Compléter l'implémentation de la `Repository`
  - `findAll`
  - `findById`

REQUÊTAGE

- Pour effectuer une requête, utilisation des classes
  - Si les paramètres nommés s'appellent comme les attributs
  - `NamedParameterJdbcTemplate` Pour la partie requête
  - `BeanPropertySqlParameterSource` Pour affecter des paramètres nommés à une requête
  - Sinon
  - `NamedParameterJdbcTemplate` Pour la partie requête
  - `MapSqlParameterSource` Pour affecter des paramètres nommés à une requête

REQUÊTAGE

- En utilisant `JdbcTemplate`, les requêtes peuvent avoir des paramètres non nommés
  - On spécifie la valeur de ces paramètres, dans l'ordre, dans la méthode *update*

```
public Produit save(Produit entity) {
  String myQuery = "INSERT INTO produit (PRO_LIBELLE, PRO_PRIX) VALUES (?, ?)";
  jdbcTemplate.update(myQuery, entity.getLibelle(), entity.getPrix());

  return entity;
}
```

REQUÊTAGE

- En utilisant `NamedParameterJdbcTemplate`, les requêtes peuvent avoir des paramètres nommés
  - On spécifie la valeur de ces paramètres dans un objet de type `MapSqlParameterSource`

REQUÊTAGE

```
public Produit save(Produit entity) {
  String myQuery = "INSERT INTO produit (PRO_LIBELLE, PRO_PRIX)
                    VALUES (:libelle, :prix)";
  NamedParameterJdbcTemplate myJdbcTemplate =
    new NamedParameterJdbcTemplate(dataSource);
  MapSqlParameterSource myParams = new MapSqlParameterSource();

  myParams.addValue("libelle", entity.getLibelle(), Types.VARCHAR);
  myParams.addValue("prix", entity.getPrix(), Types.DECIMAL);

  myJdbcTemplate.update(myQuery, myParams);

  return entity;
}
```

REQUÊTAGE

- En utilisant `NamedParameterJdbcTemplate`, les requêtes peuvent avoir des paramètres nommés
  - **SI** on utilise tous les attributs d'une classe, on peut utiliser `BeanPropertySqlParameterSource`
    - Il génère un `MapSqlParameterSource` avec tous les paramètres et valeur de la classe de l'entité

REQUÊTAGE

```
public Produit save(Produit entity) {
  String myQuery = "INSERT INTO produit (PRO_LIBELLE, PRO_PRIX)
                    VALUES (:libelle, :prix)";

  NamedParameterJdbcTemplate myJdbcTemplate =
    new NamedParameterJdbcTemplate(dataSource);
  BeanPropertySqlParameterSource myParams =
    new BeanPropertySqlParameterSource(entity);

  myJdbcTemplate.update(myQuery, myParams);

  return entity;
}
```

REQUÊTAGE

- Il est possible d'utiliser simplement la méthode *`update`* de `JdbcTemplate`
  - Mais également `SimpleJdbcInsert`

REQUÊTAGE

```
public Produit save(Produit entity) {
  SimpleJdbcInsert myJdbcInsert = new SimpleJdbcInsert(jdbcTemplate);
  MapSqlParameterSource myParams = new MapSqlParameterSource();

  myParams.addValue("PRO_LIBELLE", entity.getLibelle(), Types.VARCHAR);
  myParams.addValue("PRO_PRIX", entity.getPrix(), Types.DECIMAL);

  int id = myJdbcInsert
             .withTableName("produit")
             .usingGeneratedKeyColumns("PRO_ID")
             .executeAndReturnKey(myParams).intValue();

  entity.setId(id);

  return entity;
}
```

REQUÊTAGE

- En utilisant `SimpleJdbcInsert`
  - **SI** le nom des champs est le même que le nom des attributs, on peut utiliser `BeanPropertySqlParameterSource`
    - Il génère un `MapSqlParameterSource` avec tous les paramètres et valeur de la classe de l'entité

REQUÊTAGE

```
public Produit save(Produit entity) {
  SimpleJdbcInsert myJdbcInsert = new SimpleJdbcInsert(jdbcTemplate);
  BeanPropertySqlParameterSource myParams =
    new BeanPropertySqlParameterSource(entity);

  int id = myJdbcInsert
             .withTableName("produit")
             .usingGeneratedKeyColumns("id")
             .executeAndReturnKey(myParams).intValue();

  entity.setId(id);

  return entity;
}
```

EXERCICE

- Créer le programme **Generator**
  - Insérer quelques produits

- Compléter l'implémentation de la `Repository`
  - `save` (Insert et Update)
  - `delete`
  - `deleteById`

PROCÉDURES STOCKÉES

```
public Produit save(Produit entity) {
  String myQuery = "CALL insertProduit(?, ?)";
  jdbcTemplate.update(myQuery, entity.getLibelle(), entity.getPrix());

  return entity;
}
```

PROCÉDURES STOCKÉES

- Comme pour l'insert
  - On a la possibilité d'utiliser la méthode « update » le `JdbcTemplate`
  - Mais également `SimpleJdbcCall`

PROCÉDURES STOCKÉES

```
public Produit save(Produit entity) {
  SimpleJdbcCall myJdbcCall = new SimpleJdbcCall(jdbcTemplate)
                                  .withProcedureName("insertProduit");

  myJdbcCall.addDeclaredParameter(
    new SqlParameter("paramLibelle", Types.VARCHAR));
  myJdbcCall.addDeclaredParameter(
    new SqlParameter("paramPrix", Types.DECIMAL));


  // Retourne un Map<String, Object>
  myJdbcCall.execute(entity.getLibelle(), entity.getPrix());

  return entity;
}
```

PROCÉDURES STOCKÉES

- En utilisant `SimpleJdbcCall`
  - **SI** le nom des attributs est le même que le nom des paramètres, on peut utiliser `BeanPropertySqlParameterSource`
    - Il génère un `MapSqlParameterSource` avec tous les paramètres et valeur de la classe de l'entité

PROCÉDURES STOCKÉES

```
public Produit save(Produit entity) {
  SimpleJdbcCall myJdbcCall = new SimpleJdbcCall(jdbcTemplate)
                                  .withProcedureName("insertProduit");

  BeanPropertySqlParameterSource myParams =
    new BeanPropertySqlParameterSource(entity);


  // Retourne un Map<String, Object>
  myJdbcCall.execute(myParams);

  return entity;
}
```

EXERCICE

- Manipuler les procédures stockées
  - Appeler une procédure qui insère un produit

TRANSACTIONS

GÉRER LES TRANSACTIONS

CONFIGURATION

- Une transaction est **ACID**
  - **A**tomicité — Au complet ou pas du tout
  - **C**ohérence — Respect des contraintes
  - **I**solation — Pas de dépendance entre transactions
  - **D**urabilité — Persistance des informations

- *Sans transaction, chaque requête est indépendante*
- *Avec une transaction, toutes les requêtes doivent passer*

CONFIGURATION

- Utilisation d'un bean
  - **TransactionManager**
    - Utilise la référence de la **DataSource**

- Activation des annotations @**Transactional**
  - Utilise la référence du **TransactionManager**

CONFIGURATION

```java
@Bean
public PlatformTransactionManager transactionManager(DataSource dataSource) {
  return new DataSourceTransactionManager(dataSource);
}
```

- Annoter la classe de configuration de `@EnableTransactionManagement`
  - Pour activer l'annotation `@Transactional`

CONFIGURATION

- Déclaration de l'utilisation des transactions **Spring**
  - `@Transactional` sur la classe ou sur les méthodes concernées
  - Au niveau *`Service`* ou *`Repository`*

TRANSACTIONS

- Plusieurs options possibles avec `@Transactional` (**SPRING**)

Option	Description
`propagation`	Précise le mode de propagation de la transaction avec un énumérateur **Propagation**. La valeur par défaut est **`Propagation.`***`REQUIRED`*
`readonly`	Booléen qui indique au gestionnaire de transactions si celle-ci est en lecture seule (mises à jour impossibles) ou non
`isolation`	Précise le niveau d'isolation de la transaction avec un énumérateur **Isolation**. La valeur par défaut est **`Isolation.`***`DEFAULT`*
`rollback`	Nature de l'exception levée à partir de laquelle faire un `Rollback.` **`RuntimeException`** et ses dérivées par défaut.

PROPAGATION

Propagation	Description
REQUIRED	Utilise la transaction existante, ou en crée une nouvelle si aucune n'existe. Si `Rollback`, on rollback toutes les instructions, sur tous les niveaux de la transaction.
REQUIRED_NEW	Crée une nouvelle transaction (même si une existe déjà) . Si transaction déjà existante, elle est mise en pause. Si `Rollback`, on rollback uniquement les instructions à partir de cette méthode.
NESTED	Utilise une seule transaction : la transaction physique. Met en place les mécanismes de *savepoints*. **Valable uniquement dans le cas de JDBC.** Si `Rollback`, on rollback uniquement les instructions de la méthode en échec.

PROPAGATION

Propagation	Description
MANDATORY	Utilise une transaction qui **doit** exister au préalable. Si aucune transaction n'existe, une exception est levée. Si `Rollback`, on rollback toutes les instructions, sur tous les niveaux de la transaction.
NEVER	Doit s'exécuter en dehors de toute transaction. Si une transaction existe, une exception est levée. Pas de `Rollback` possible.
NOT_SUPPORTED	S'exécute en dehors de toute transaction. Si une transaction existe, elle est mise en pause. Pas de `Rollback` possible.
SUPPORTS	Utilise la transaction existante. Si aucune n'existe, elle n'utilisera pas de transaction. Si transaction et si `Rollback`, on rollback toutes les instructions, sur tous les niveaux de la transaction.

ISOLATION

Isolation	Description
READ_UNCOMMITED	Dirty reads, Non-repeatable reads, Phantom reads
READ_COMMITED	Non-repeatable reads, phantom reads
REPEATABLE_READ	Phantom reads
SERIALIZABLE	-
DEFAULT	*READ_COMMITED* && *REPEATABLE_READ*

ISOLATION

Isolation	Description
READ_UNCOMMITED	Permet la lecture d'informations non commités (*dirty reads*) : - Nouveau produit dans la transaction #1 - Rechargement de ce nouveau produit dans la transaction #2 Si *dirty reads*, la lecture du nouveau produit dans la transaction #2 fonctionnera (alors que le produit n'est pas encore en base de données, voire que la transaction #1 a rollbacké …)
READ_COMMITED	Prévient du phénomène *dirty reads* Ne permet pas la lecture d'informations non commitées Mais ne prévient pas des autres phénomènes

ISOLATION

Isolation	Description
REPEATABLE_READ	Prévient du phénomène *non-repeatable reads* Empêche une lecture d'information différente pour une même transaction : - Lecture d'un produit dans la transaction #1 - Modification du produit dans la transaction #2 - Lecture du produit dans la transaction #1 Si *non-repeatable reads*, les lectures liront des informations différentes
SERIALIZABLE	Prévient du phénomène *phantom reads* La transaction doit être terminée (attente) pour lire - Enregistrement d'un nouveau produit dans la transaction #1 - Lecture de la liste des produits dans la transaction #2 Si *phantom reads*, alors la lecture peut exclure le nouveau produit (pas sauvegardé à temps pour la lecture)

EXERCICE

- Démonstration @**`Transactional`** *isolation*
  - Avec Service Asynchrone

EXERCICE

- Terminer le **CRUD** Produit
  - Avec une gestion des transactions au niveau *Service*