

2.11 Activité Pratique: Intégration de Spring et Hibernate

L'intégration d'Hibernate avec Spring permet de simplifier la gestion des transactions et l'injection de dépendances, ce qui facilite la structuration d'un projet modulaire. Par rapport à une approche utilisant uniquement Hibernate, Spring offre une abstraction qui réduit le code lié à la gestion des transactions et à la configuration. Ce TP montre comment configurer un projet Maven avec Spring et Hibernate pour réaliser une gestion efficace des entités.

Dans ce travail pratique, il s'agit de créer un projet Maven qui intègre Spring et Hibernate pour gérer des entités via une base de données MySQL. L'objectif est de démontrer comment ces frameworks peuvent être utilisés ensemble pour simplifier la gestion des transactions, l'injection de dépendances et le mapping objet-relationnel (ORM). Le projet inclura une configuration de base de données, la création d'une entité simple, et la gestion des opérations CRUD (Create, Read, Update, Delete) via un DAO. La structure finale de ce projet est présenté dans la figure 3.1.

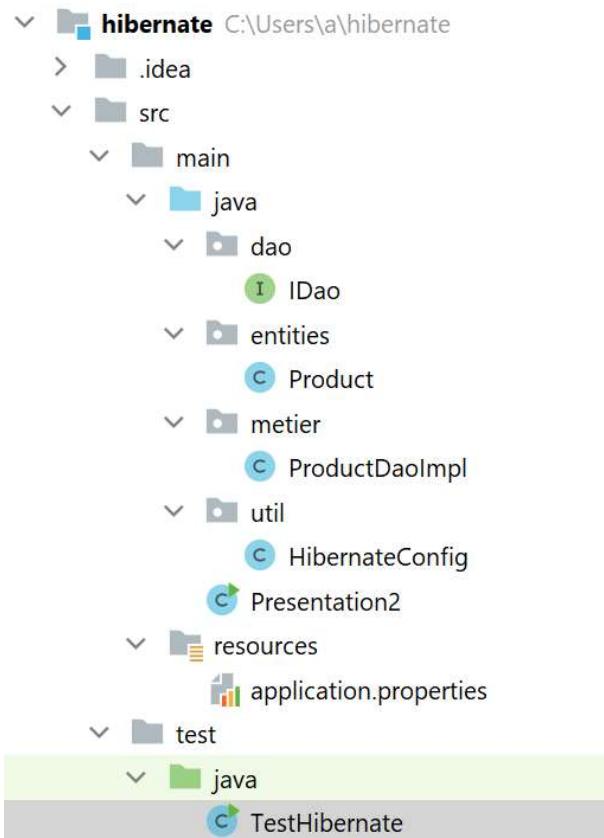


Figure 2.4: Structure de projet.

Objectifs pédagogiques

- Comprendre les bases de la gestion des dépendances avec Maven.
- Apprendre à configurer Spring pour gérer l'injection de dépendances et les transactions.
- Intégrer Hibernate pour la gestion des entités en base de données.
- Apprendre à créer un DAO pour gérer les opérations CRUD.
- Configurer une base de données MySQL pour interagir avec les entités du projet.

Étapes du TP

1. Initialisation du projet Maven

La première étape consiste à créer un projet Maven et à ajouter les dépendances nécessaires dans le fichier pom.xml. Ces dépendances incluent Spring (pour l'injection de dépendances et la gestion des transactions), Hibernate (pour l'ORM), et MySQL (pour la base de données).

Dépendances Maven

Listing 2.19: Dépendances Maven dans pom.xml

```

<dependencies>
    <!-- Spring Context pour l'injection de dépendances -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.22</version>
    </dependency>

    <!-- Spring ORM pour l'intégration avec Hibernate -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>5.3.22</version>
    </dependency>

    <!-- Hibernate Core pour la gestion ORM -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.12.Final</version>
    </dependency>

    <!-- MySQL Connector pour la connexion à la base de données -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.29</version>
    </dependency>

    <!-- Spring Transaction pour la gestion des transactions -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>5.3.22</version>
    </dependency>

    <!-- Validation API pour les annotations JPA -->
    <dependency>
        <groupId>javax.validation</groupId>
        <artifactId>validation-api</artifactId>
        <version>2.0.1.Final</version>
    </dependency>
</dependencies>

```

Explication :

- spring-context permet d'utiliser Spring pour l'injection de dépendances.
- spring-orm intègre Hibernate avec Spring pour la gestion ORM.
- hibernate-core est utilisé pour le mapping objet-relationnel.
- mysql-connector-java est utilisé pour connecter le projet à une base de données MySQL.
- spring-tx gère les transactions de manière transparente.

- validation-api permet de valider les entités JPA avec des annotations.

2. Configuration de la base de données

La configuration de la base de données se fait à l'aide d'un fichier `application.properties` situé dans le répertoire `src/main/resources`. Ce fichier contient les informations de connexion à MySQL et les propriétés Hibernate.

Listing 2.20: Fichier application.properties

```
# Configuration de la base de données
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/base?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=

# Propriétés Hibernate
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Explication :

- Le pilote MySQL est spécifié par `spring.datasource.driver-class-name`.
- L'URL de connexion à la base de données est spécifiée par `spring.datasource.url`.
- `hibernate.dialect` permet à Hibernate de générer des requêtes SQL compatibles avec MySQL.
- `hibernate.ddl-auto=update` permet de créer automatiquement les tables si elles n'existent pas.
- `show-sql=true` affiche les requêtes SQL dans la console pour un débogage plus facile.

3. Création de l'entité Product

Une entité représente un objet stocké dans une base de données. Ici, l'entité `Product` représente un produit avec trois attributs : `id`, `name`, et `price`.

Listing 2.21: Classe Product

```
package entities;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private double price;

    public Product() {}
```

```
// Getters et Setters
}
```

Explication :

- `@Entity` indique que la classe `Product` est une entité JPA.
- `@Id` et `@GeneratedValue` définissent l'attribut `id` comme clé primaire auto-incrémentée.

4. Création de l'interface DAO

Le DAO (*Data Access Object*) est un patron de conception qui permet de gérer l'accès aux données d'une manière abstraite et modulaire. Ici, l'interface `IDao` définit les opérations CRUD pour gérer l'entité `Product`.

Listing 2.22: Interface IDao

```
package dao;

import java.util.List;

public interface IDao<T> {
    boolean create(T o);
    boolean delete(T o);
    boolean update(T o);
    T findById(int id);
    List<T> findAll();
}
```

Explication : Cette interface est générique, elle peut être utilisée pour n'importe quelle entité, pas seulement `Product`. Les méthodes `create`, `delete`, `update`, `findById`, et `findAll` permettent de gérer les entités en base de données.

5. Implémentation du DAO `ProductDaoImpl`

L'implémentation de `ProductDaoImpl` utilise Hibernate et Spring pour effectuer les opérations CRUD sur l'entité `Product`.

Listing 2.23: Classe ProductDaoImpl

```
package metier;

import dao.IDao;
import entities.Product;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Repository
public class ProductDaoImpl implements IDao<Product> {
```

```

    @Autowired
    private SessionFactory sessionFactory;

    @Override
    @Transactional
    public boolean create(Product product) {
        Session session = sessionFactory.getCurrentSession();
        session.save(product);
        return true;
    }

    // Méthodes delete, update, findById, findAll
    // A compléter
}

```

Explication :

- `@Transactional` permet à Spring de gérer les transactions automatiquement.
- `sessionFactory.getCurrentSession()` retourne la session Hibernate actuelle pour exécuter les opérations sur les entités.

6. Configuration de Spring et Hibernate

La classe `HibernateConfig` est essentielle pour configurer l'intégration de Hibernate avec Spring dans un projet Maven. Cette classe utilise des annotations Spring pour définir les beans qui gèrent la base de données et les transactions, ainsi que pour charger les paramètres de configuration depuis un fichier de propriétés.

Voici le code de `HibernateConfig` avec une explication détaillée de chaque composant :

Listing 2.24: Classe `HibernateConfig`

```

package util;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.orm.hibernate5.HibernateTransactionManager;
import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import javax.sql.DataSource;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration
@ComponentScan(basePackages = {"dao", "entities", "metier"})
@EnableTransactionManagement
@PropertySource("classpath:application.properties")
public class HibernateConfig {

    @Value("${spring.datasource.driver-class-name}")
    private String driverClassName;

    @Value("${spring.datasource.url}")
    private String url;
}

```

```

@Value("${spring.datasource.username}")
private String username;

@Value("${spring.datasource.password}")
private String password;

@Bean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName(driverClassName);
    dataSource.setUrl(url);
    dataSource.setUsername(username);
    dataSource.setPassword(password);
    return dataSource;
}

@Bean
public LocalSessionFactoryBean sessionFactory() {
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
    sessionFactory.setDataSource(dataSource());
    sessionFactory.setPackagesToScan("entities");
    return sessionFactory;
}

@Bean
public HibernateTransactionManager transactionManager() {
    HibernateTransactionManager txManager = new HibernateTransactionManager();
    txManager.setSessionFactory(sessionFactory().getObjet());
    return txManager;
}
}

```

a. @Configuration

L'annotation `@Configuration` indique à Spring que cette classe contient des définitions de beans à gérer dans le contexte de l'application. Cela permet à Spring de scanner et d'instancier les beans définis dans cette classe lors du démarrage.

b. @ComponentScan

L'annotation `@ComponentScan(basePackages = "dao", "entities", "metier")` permet à Spring de scanner automatiquement les packages spécifiés (dao, entities, metier) à la recherche de composants annotés comme `@Repository`, `@Service`, ou `@Component`. Cela permet à Spring de gérer automatiquement les classes dans ces packages comme des beans.

c. @EnableTransactionManagement

L'annotation `@EnableTransactionManagement` permet d'activer la gestion des transactions dans Spring. Avec cette configuration, il est possible d'utiliser l'annotation `@Transactional` pour marquer les méthodes qui nécessitent une gestion transactionnelle (par exemple, pour les opérations CRUD).

4. @PropertySource

L'annotation `@PropertySource("classpath:application.properties")` indique à Spring de charger les propriétés de configuration depuis un fichier externe situé dans `src/main/resources/application.properties`. Les propriétés de ce fichier sont ensuite injectées dans les beans grâce à l'annotation `@Value`.

d. Méthode dataSource()

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName(driverClassName);
    dataSource.setUrl(url);
    dataSource.setUsername(username);
    dataSource.setPassword(password);
    return dataSource;
}
```

Cette méthode définit un bean DataSource, qui fournit la configuration de connexion à la base de données MySQL.

- DriverManagerDataSource est une implémentation de DataSource qui permet de spécifier le pilote, l'URL, le nom d'utilisateur et le mot de passe pour se connecter à la base de données.
- Les propriétés driverClassName, url, username, et password sont injectées à partir du fichier application.properties via l'annotation @Value.
- Ce bean est essentiel pour fournir la connexion à la base de données à Hibernate.

e. Méthode sessionFactory()

```
@Bean
public LocalSessionFactoryBean sessionFactory() {
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
    sessionFactory.setDataSource(dataSource());
    sessionFactory.setPackagesToScan("entities");
    return sessionFactory;
}
```

- LocalSessionFactoryBean est un bean Spring qui configure Hibernate pour créer des sessions permettant d'interagir avec la base de données.
- La méthode sessionFactory.setDataSource() associe la source de données (définie dans dataSource()) à Hibernate.
- sessionFactory.setPackagesToScan("entities") indique à Hibernate où scanner les classes annotées @Entity, ici dans le package entities.
- Ce bean permet à Hibernate d'initialiser la gestion des entités et des sessions pour effectuer des opérations CRUD.

f. Méthode transactionManager()

```
@Bean
public HibernateTransactionManager transactionManager() {
    HibernateTransactionManager txManager = new HibernateTransactionManager();
    txManager.setSessionFactory(sessionFactory().getObjectType());
    return txManager;
}
```

- HibernateTransactionManager est le gestionnaire de transactions pour Hibernate. Il permet de coordonner les transactions lors de l'exécution des opérations en base de données.
- txManager.setSessionFactory(sessionFactory().getObjectType()) lie le gestionnaire de transactions à la SessionFactory pour gérer les sessions et les transactions.
- Ce bean garantit que toutes les opérations marquées @Transactional seront exécutées dans une transaction gérée par Hibernate.

Test de la configuration

La classe TestHibernate est utilisée pour vérifier que la configuration de Spring et Hibernate est correcte. Elle permet de tester l'initialisation de la SessionFactory et du HibernateTransactionManager afin de s'assurer que tout fonctionne correctement avant d'exécuter d'autres opérations sur les entités.

Listing 2.25: Classe TestHibernate

```

import org.hibernate.SessionFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.orm.hibernate5.HibernateTransactionManager;
import util.HibernateConfig;

public class TestHibernate {
    public static void main(String[] args) {
        // Chargement du contexte Spring avec la configuration Hibernate
        ApplicationContext context = new
            AnnotationConfigApplicationContext(HibernateConfig.class);

        // Vérification de la configuration de SessionFactory
        SessionFactory sessionFactory = context.getBean(SessionFactory.class);
        if (sessionFactory != null) {
            System.out.println("SessionFactory is configured correctly.");
        } else {
            System.out.println("SessionFactory not found.");
        }

        // Vérification de la configuration de HibernateTransactionManager
        HibernateTransactionManager txManager =
            context.getBean(HibernateTransactionManager.class);
        if (txManager != null) {
            System.out.println("Transaction Manager is configured correctly.");
        } else {
            System.out.println("Transaction Manager not found.");
        }
    }
}

```

a. ApplicationContext et AnnotationConfigApplicationContext

La méthode main commence par initialiser un ApplicationContext, qui est le conteneur de Spring. Ici, AnnotationConfigApplicationContext est utilisé pour charger la configuration définie dans la classe HibernateConfig.

```

ApplicationContext context = new
    AnnotationConfigApplicationContext(HibernateConfig.class);

```

Explication : AnnotationConfigApplicationContext permet de charger un contexte Spring à partir d'une configuration basée sur des annotations, dans ce cas, la classe HibernateConfig. Cela initialise tous les beans déclarés dans cette configuration.

b. Vérification de SessionFactory

Ensuite, le code tente d'obtenir un bean de type SessionFactory depuis le conteneur Spring.

```

SessionFactory sessionFactory = context.getBean(SessionFactory.class);

```

```

if (sessionFactory != null) {
    System.out.println("SessionFactory is configured correctly.");
} else {
    System.out.println("SessionFactory not found.");
}

```

Explication : SessionFactory est un composant clé d’Hibernate qui permet de créer des sessions pour interagir avec la base de données. Ce test assure que le bean SessionFactory a été correctement configuré par Spring en appelant la méthode sessionFactory() dans HibernateConfig.

c. Vérification de HibernateTransactionManager

La deuxième vérification porte sur le HibernateTransactionManager, qui gère les transactions pour les opérations sur la base de données.

```

HibernateTransactionManager txManager =
    context.getBean(HibernateTransactionManager.class);
if (txManager != null) {
    System.out.println("Transaction Manager is configured correctly.");
} else {
    System.out.println("Transaction Manager not found.");
}

```

Explication : HibernateTransactionManager est responsable de la gestion des transactions dans le projet. Cette vérification assure que le gestionnaire de transactions a été correctement configuré pour gérer les transactions autour des méthodes marquées @Transactional.

d. Importance des vérifications

Ces vérifications permettent de s’assurer que le projet est correctement configuré avant de commencer à effectuer des opérations sur la base de données. Une mauvaise configuration de la SessionFactory ou du HibernateTransactionManager empêcherait le bon fonctionnement de l’application et pourrait entraîner des erreurs lors de l’exécution des opérations CRUD.



La classe TestHibernate est un outil simple mais important pour s’assurer que les composants clés de la configuration Spring-Hibernate, tels que SessionFactory et HibernateTransactionManager, sont correctement initialisés avant de procéder à toute interaction avec la base de données. Cela permet de vérifier la cohérence de la configuration et d’identifier d’éventuels problèmes avant de les rencontrer en production.

7. Test et utilisation des DAO

Enfin, la classe Presentation2 permet de tester l’ajout d’un produit à la base de données et de s’assurer que le DAO fonctionne correctement.

Listing 2.26: Classe Presentation2

```

import dao.IDao;
import entities.Product;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import util.HibernateConfig;

```

```
public class Presentation2 {  
    public static void main(String[] args) {  
        ApplicationContext context = new  
            AnnotationConfigApplicationContext(HibernateConfig.class);  
  
        IDao<Product> productDao = context.getBean(IDao.class);  
  
        Product product = new Product();  
        product.setName("Produit 1");  
        product.setPrice(100.0);  
  
        productDao.create(product);  
  
        System.out.println("Produit sauvegardé :" + product.getName());  
    }  
}
```

 Ce travail pratique montre comment intégrer Spring et Hibernate dans un projet Maven pour gérer des entités dans une base de données MySQL. Grâce à la gestion des transactions et à l'injection de dépendances fournies par Spring, il est possible de structurer efficacement le code et de réduire la complexité de la gestion des entités.