Formation Spring Framework

Accès aux données

Spring Framework & données

- Spring offre une intégration avec des outils de persistance "classique" de l'écosystème Java :
  - JDBC
  - Hibernate
  - JPA
  - Ibatis
  - EJB
  - …
- Spring s'adapte à chaque technologie et masque les détails technique au développeur.
- Spring Framework encapsule les exceptions spécifiques des outils de persistance.
- Les transactions sont configurables par XML ou annotations et déléguées automatiquement au gestionnaire de transactions.

Hiérarchie des exceptions

- L'objectif est de pouvoir s'abstraire des exceptions spécifiques aux implémentations
  - SQLException ou HibernateException.
- Spring fournit la hiérarchie « DataAccessException »
  - Type « Runtime » unchecked.
  - Encapsule les types d'exceptions quelque soit la technologie.
- Quelques exceptions
  - CleanupFailureDataAccessException
    - Une exception a été levée pendant la libération de ressources (exemple méthode close() sur une connexion)
  - DataRetrievalFailureException
    - Une erreur s'est produite lors d'une requête de sélection
  - DeadlockLoserDataAccessException
    - Problème d'accès concurrents, processus bloqué par un lock

Spring Jdbc

JDBC...

```
try {
          conn = this.datasource.getConnection();
          stmt = conn.prepareStatement("select ID, FNAME, LNAME from person where LNAME = ?");
          stmt.setString(1, lname);
          ResultSet rs = stmt.executeQuery();
          while (rs.next()) {
                    p = new Person();
                    p.setId(rs.getInt("ID"));
                    p.setFirstName(rs.getString("FNAME"))
                    p.setLastName(rs.getString("LNAME"));
          }
} catch (SQLException e) {
          LOGGER.error(e);
} finally {
          try {
                    if (stmt != null)
                              stmt.close();
          } catch (SQLException e) {
                    LOGGER.warn(e);
          }
          try {
                    if (conn != null)
                              conn.close();
          } catch (SQLException e) {
                    LOGGER.warn(e);
          }
}
```

Spring Jdbc - Qui fait quoi ?

- Spring Jdbc est une librairie fournie par Spring qui permet d'utiliser JDBC plus simplement.

Action	Spring	Vous
Définir les paramètres de connexion		X
Ouvrir une connexion	X	
Spécifier la requête SQL		X
Définir les paramètres et fournir les valeurs des paramètres de requête		X
Exécuter la requête	X	
Mettre en place la boucle pour itérer sur les résultats	X	
Traitement des résultats pour chaque itération		X
Récupérer les exceptions	X	
Gérer les transactions	X	
Fermer les ressources (Connection, Statement, Resultset)	X	

Namespace jdbc

- L'espace de nommage JDBC apparait avec la version 3 de Spring.
- Spring propose deux tags XML
  - <jdbc:embedded-database> - met à disposition d'une datasource.
  - <jdbc:initialize-database> - initialise une base de données.
- Adapté pour la réalisation de tests unitaires automatisés ou non
  - S'intègre avec les bases de données intégrées.
  - Supporte nativement les outils HSQL (par défaut), H2 et Derby.

```
<jdbc:embedded-database id="dataSource">
     <jdbc:script location="classpath:db/schema.sql" />
     <jdbc:script location="classpath:db/init-data.sql" />
</jdbc:embedded-database>

<jdbc:initialize-database data-source="dataSource">
     <jdbc:script location="classpath:db/init-db.sql" />
</jdbc:initialize-database>
```

Source de données

- Spring Jdbc supporte les bases de données embarquées :
  - H2
  - HSQL
  - DERBY

```
EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
EmbeddedDatabase db = builder
    .type(EmbeddedDatabaseType.H2)
    .script("schema.sql")
    .script("test-data.sql")
    .build();
```

Source de données

- La classe DriverManagerDataSource fourni par Spring permet de créer une source de données.

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql://localhost:3306/pizzeria?useSSL=false");
dataSource.setUsername("root");
dataSource.setPassword("");
```

JdbcTemplate.queryForObject

```
@Repository
public class PizzaDao {

	private JdbcTemplate jdbcTemplate;

	@Autowired
	public PizzaDao(DataSource datasource) {
		this.jdbcTemplate = new JdbcTemplate(datasource);
	}

	public Integer countPizzas() {
		String sql = "SELECT COUNT(*) FROM PIZZA";
		return this.jdbcTemplate.queryForObject(sql, Integer.class);
	}
}
```

JdbcTemplate.queryForObject

```
@Repository
public class PizzaDao {

	private JdbcTemplate jdbcTemplate;

	@Autowired
	public PizzaDao(DataSource datasource) {
		this.jdbcTemplate = new JdbcTemplate(datasource);
	}

	public Integer countPizzasByName(String name) {
		String sql = "SELECT COUNT(*) FROM PIZZA WHERE PIZZA_NAME=?";
		return this.jdbcTemplate.queryForObject(sql, Integer.class, name);
	}
}
```

RowMapper

```
@Repository
public class PizzaDao {

	…

	public Pizza findPizzaByName(String name) {
		String sql = "SELECT * FROM PIZZA WHERE PIZZA_NAME=?";
		return this.jdbcTemplate.queryForObject(sql, new PizzaMapper(), name);
	}
}

public class PizzaMapper implements RowMapper<Pizza> {

	public Pizza mapRow(ResultSet rs, int rowNum) throws SQLException {
		Pizza p = new Pizza();
		p.setId(rs.getInt("ID"));
		p.setPizzaName(rs.getString("PIZZA_NAME"));
		return p;
	}
}
```

RowMapper

```
@Repository
public class PizzaDao {

	…

	public List<Pizza> findAllPizza() {
		String sql = "SELECT * FROM PIZZA";
		return this.jdbcTemplate.query(sql, new PizzaMapper());
	}
}

public class PizzaMapper implements RowMapper<Pizza> {

	public Pizza mapRow(ResultSet rs, int rowNum) throws SQLException {
		Pizza p = new Pizza();
		p.setId(rs.getInt("ID"));
		p.setPizzaName(rs.getString("PIZZA_NAME"));
		return p;
	}
}
```

RowCallbackHandler

```
@Repository
public class PizzaDao {

	…

	public List<Pizza> findAllPizza() {
		String sql = "SELECT * FROM PIZZA";
		return this.jdbcTemplate.query(sql, new PizzaTxtExporter());
	}
}

public class PizzaTxtExporter implements RowCallBackHandler {

	public void processRow(ResultSet rs) throws SQLException {
		// code d'export
	}
}
```

ResultSetExtractor

```
@Repository
public class PizzaDao {

	…

	public List<Pizza> findAllPizza() {
		String sql = "SELECT * FROM PIZZA";
		return this.jdbcTemplate.query(sql, new PizzaExtractor());
	}
}

public class PizzaExtractor implements ResultSetExtractor<Pizza> {

	public Pizza extractData(ResultSet rs) throws SQLException {
		// code d'extraction
	}
}
```

Lequel utiliser ?

- L'interface « RowMapper »
  - Adapté lorsque chaque tuple est associé à un objet métier.
- L'interface « RowCallbackHandler »
  - Adapté lorsque aucun résultat ne doit être retourné *pour chaque tuple*.
- L'interface « ResultSetExtractor »
  - Adapté lorsqu'un ensemble de tuples correspond à un seul objet métier.

Insérer des données

```
@Repository
public class PizzaDao {

	…

	public void create(Pizza p) {
		String sql = "INSERT INTO PIZZA (ID,PIZZA_NAME) VALUES(?,?)";
		this.jdbcTemplate.update(sql, p.getId(), p.getPizzaName());
	}


	public void update(Pizza p) {
		String sql = "UPDATE PIZZA SET PIZZA_NAME = ? WHERE ID = ? ";
		this.jdbcTemplate.update(sql, p.getPizzaName(), p.getId());
	}

}
```

Gestion des transactions

Gestion des transactions

- Spring n'implémente pas de transaction manager mais permet de se connecter et de déléguer des appels aux gestionnaire transactionnels

- Plusieurs implémentations sont proposées
  - Transaction local
    - HibernateTransactionManager
    - JpaTransactionManager
    - DataSourceTransactionManager
  - Transaction managée
    - JtaTransactionManager
    - WebLogicJtaTransactionManager
    - WebSphereUowTransactionManager

Transaction Manager via XML

- Déclaration pour JDBC

```
<bean id="txManager"
     class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
     <property name="dataSource" ref="dataSource" />
</bean>
```

- Déclaration pour Hibernate

```
<bean id="txManager"
          class="org.springframework.orm.hibernate3.HibernateTransactionManager">
     <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

- Déclaration pour JTA

```
<tx:jta-transaction-manager/>
```

  - Spring détecte le serveur d'application et applique le manager adéquat (JtaTransactionManager, WebLogicJtaTransactionManager, WebSphereUowTransactionManager, …)

Transaction Manager via @

```
@EnableTransactionManagement
public class PizzaAppConfig {

    @Bean
    public PlatformTransactionManager txManager() {
        return new DataSourceTransactionManager(dataSource());
    }
```

@Transactional

- Sur une classe

```
@Transactional
public class PizzaService {
```

- Sur une méthode

```
@Transactional(timeout=60)
public void create(Pizza pizza) {
```

Gestion des exceptions

- @Transactionnal supporte la gestion des exceptions

```
@Transactional(rollbackFor=Throwable.class,noRollbackFor=PizzaAccessExcept
ion.class)
public void create(Pizza pizza) {
```

Propagation des transactions

Attribut	Si une transaction existe	Si pas de transaction
REQUIRED	Utiliser la transaction existante	Démarrer une nouvelle transaction
REQUIRES_NEW	Suspendre la transaction en cours. Démarrer une nouvelle transaction.	Démarrer une nouvelle transaction.
MANDATORY	Utiliser la transaction existante	Lancer une exception
NEVER	Lancer une exception	Traiter sans transaction
NOT_SUPPORTED	Suspendre la transaction en cours.	Traiter sans transaction
SUPPORTS	Utiliser la transaction existante	Traiter sans transaction
NESTED	Crée une transaction imbriquée qui peut être indépendante	Démarrer une nouvelle transaction

Spring ORM

Spring ORM

- Spring ORM offre un support de :
    - Hibernate
    - Java Data Objects (JDO)
    - Java Persistence API (JPA)

- Ce cours est centré sur le support de JPA.

- Dépendance Maven

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
</dependency>
```

Spring ORM - JPA

3 options de configuration JPA :

- **LocalEntityManagerFactoryBean**
  - Utilisé pour une application stand-alone ou pour des tests d'intégration

- Obtenir une instance d'EntityManagerFactory **depuis JNDI** (Java Naming and Directory Interface).
  - Utilisé pour une application déployée dans un serveur Java EE Full Profile

- **LocalContainerEntityManagerFactoryBean**
  - Utilisé pour des conteneurs légers

LocalEntityManagerFactoryBean

- Exemple de configuration

```
<beans>
    <bean id="emf"
        class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="pizzeria-pu" />
    </bean>
</beans>
```

- Permet d'obtenir une instance d'EntityManagerFactory.
  - Utilise en interne l'autodétection JPA pour Java SE (Persistence.createEntityManagerFactory).

EMF depuis JNDI

- Exemple de configuration

```
<beans>
    <jee:jndi-lookup id="myEmf" jndi-name="jdbc/pizzeria-pu" />
</beans>
```

LocalContainerEntityManagerFactoryBean

- Exemple de configuration

```
<bean id="emf"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="maDataSource" />
</bean>
```

@PersistenceUnit & @PersistenceContext

- Spring peut interpréter les annotations @PersistenceUnit et @PersistenceContext si le bean **PersistenceAnnotationBeanPostProcessor** est activé.
    - Bean automatiquement activé si configuration par annotation.

- Exemple d'activation

```
<bean
class="org.springframework.orm.jpa.support.Persistence
AnnotationBeanPostProcessor" />
```

Spring Data JPA

Spring Data JPA

- Spring Data JPA est un sous projet de Spring Data qui facilite l'utilisation de JPA :
  - Configuration aisée via Java Config
  - Support du mapping JPA XML ou via annotations
  - Support de la pagination
  - Support de requêtes dynamiques
  - …
- Objectif → Réduire le code redondant.
- Dépendance Maven :

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

Configurer Spring pour JPA

```
@EnableJpaRepositories(“fr.pizzeria.repos”)
public class PizzaAppConfig {

      @Bean
        public PlatformTransactionManager transactionManager() {
          JpaTransactionManager txManager = new JpaTransactionManager();
          txManager.setEntityManagerFactory(entityManagerFactory());
          return txManager;
        }

      @Bean
      public EntityManagerFactory entityManagerFactory() {

            HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
            vendorAdapter.setGenerateDdl(true);

            LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
            factory.setJpaVendorAdapter(vendorAdapter);
            factory.setPackagesToScan("fr.pizzeria.spring.domain");
            factory.setDataSource(dataSource());
            factory.afterPropertiesSet();

            return factory.getObject();
      }
```

Activer Spring Data JPA et indiquer les packages où se trouvent les interfaces

Un bean qui s’appelle transactionManager doit exister

Un bean qui s’appelle entityManagerFactory doit exister

Interface Repository

- Interface repository pour JPA

```
public interface PersonRepository extends JpaRepository<Person, Long> {

	List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

	List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);

	List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

	List<Person> findByLastnameIgnoreCase(String lastname);

	List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

	List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
}
```

TP – Spring Data JPA

- Utiliser Spring Data JPA