Spring
Java Persistence API

Rubén Gómez García

Version 1.0.0 2019-03-26

Contenidos

Capítulo 1. Spring Jdbc

Framework que pretende dar soporte avanzado a la especificación JDBC de Java.

Se basa en la utilización de un Bean de tipo **JdbcTemplate**, que permite abstraer de la complejidad del Api de JDBC, ya que gestiona la conexión y el procesamiento de los **ResultSet**, para esto último ayudado de una implementación de **RowMapper<T>**.

Spring tambien ofrece una clase de soporte **JdbcDaoSupport** que embebe el uso de **JdbcTemplate**.

Para emplearlo, habra que definir una clase que extienda de **JdbcDaoSupport**

```
public class JdbcTemplateClienteDao extend JdbcDaoSupport implements ClienteDao {}
```

Una vez definida la clase, desde los métodos se tiene acceso a una instancia de **JdbcTemplate** con **getJdbcTemplate()**.

Y es esta clase **JdbcTemplate**, la que proporciona las funcionalidades para realizar las consultas, a traves de

- query()

```
List<CLiente> clientes = getJdbcTmplate().query("SELECT * FROM CLIENTE", new
ClienteRowMapper());
```

- queryForObject()
- update()

```
Map<String, Object> param = new HashMap<String, Object>();
        param.put("nombre", cliente.getNombre());
        param.put("direccion", cliente.getDireccion());
        param.put("telefono", cliente.getTelefono());
getTemplate().update("insert into clientes.cliente (nombre,direccion,telefono) values
(:nombre,:direccion,:telefono)", param);
```

Los métodos de la plantilla aceptarán de forma generica un String que represente la consulta a ejecutar, que se puede parametrizar con la sintaxis **:<nombre de parametro>** y un mapa, donde las claves son los nombres de los parametros de la consulta.

Siendo **RowMapper** una clase de utilleria que abstrae la extracción de los datos del **ResultSet**

```
public class RowMapperClienteImpl implements RowMapper<Cliente> {
    public Cliente mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Cliente(rs.getInt("id"),
                rs.getString("nombre"),
                rs.getString("direccion"),
                rs.getString("telefono")
                );
    }
}
```

Capítulo 2. Spring ORM

Framework que permite la integración de otros framework Orm como Hibernate, Jpa o MyIbatis.

De forma analoga a lo que sucede con JDBC, se proporcionan plantillas que encapsulan el manejo del API del ORM, facilitando la creación de la capa de persistencia, así se dispone de

- HibernateTemplate
- JpaTemplate

Estas plantillas dependerán de objetos del API correspondiente como son **SessionFactory** o **EntityManagerFactory**

2.1. Hibernate

Spring proporciona las siguientes implementaciones para construir los objetos **SessionFactory**.

- AnnotationSessionFactoryBean: Permite interpretar las anotaciones **@Entity**

```
<bean id="sessionFactory" class=
"org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="packagesToScan">
        <list>
            <value>com.curso.spring</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <propkey="dialect">org.hibernate.dialect.DerbyDialect</prop>
        </props>
    </property>
</bean>
```

En vez del **packageToScan**, se pueden indicar las **annotatedClases**

```
<bean id="sessionFactory" class=
"org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="annotatedClasses">
        <list>
            <value>com.entidades.Persona</value>
            <value>com.entidades.Factura</value>
        </list>
    </property>
    <propertyname="hibernateProperties">
        <props>
            <propkey="dialect">org.hibernate.dialect.DerbyDialect</prop>
        </props>
    </property>
</bean>
```

- LocalSessionFactoryBean

```
<bean id="sessionFactory" class=
"org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
        <list>
            <value>Persona.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="dialect">org.hibernate.dialect.DerbyDialect</prop>
        </props>
    </property>
</bean>
```

Con **JavaConfig** la creación del Bean seria similar

```
@Bean
@Autowired
public LocalSessionFactoryBean sessionFactory(DataSource ds) {
    LocalSessionFactoryBean lsfb = new LocalSessionFactoryBean();
    //Entidades
    lsfb.setPackagesToScan("com.atsistemas.persistencia.core.entidades");
    //Origen de datos
    lsfb.setDataSource(ds);
    //Otras propiedades
    Properties hibernateProperties = lsfb.getHibernateProperties();

    hibernateProperties.setProperty("hibernate.dialect",
"org.hibernate.dialect.MySQL57Dialect");
    hibernateProperties.setProperty("hibernate.show_sql", "true");
    hibernateProperties.setProperty("hibernate.format_sql", "true");
    hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "create");

    return lsfb;
}
```

A mayores del **Bean** de tipo **SessionFactory**, se precisa tambien de uno de tipo **TransactionManager**, que gestione la apertura y cierre de las transacciones, la declaracion de este Bean y su configuracion, esta explicado mas adelante en la seccion de **Transacciones**

2.2. Jpa

Spring proporciona las siguientes implementaciones para construir los objetos **EntityManager**.

- LocalContainerEntityManagerFactoryBean: Los Bean de entidad son gestionados por el contenedor, por lo que no es necesario definir el fichero **META-INF/persistence.xml** y en cambio si será necesario definir un **VendorAdapter** de entre los que proporciona Spring, dependiendo de la implementacion de JPA a emplear.
- EclipseLinkJpaVendorAdapter.
- HibernateJpaVendorAdapter.
- OpenJpaVendorAdapter.
- TopLinkJpaVendorAdapter.

```
<bean id="jpaVendorAdapter" class=
"org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <property name="database" value="Derby"/>
    <property name="showSql" value="true"/>
    <property name="generateDdl" value="false"/>
    <property name="databasePlatform" value="org.hibernate.dialect.DerbyDialect"/>
</bean>
```

Ademas de la definicion del Bean de Spring

```
<bean id="emf" class=
"org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource"ref="dataSource"/>
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter"/>
    <property name="packagesToScan" value="com.cursospring.modelo.entidad"/>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <!--<prop key="hibernate.current_session_context_class">thread</prop>--
><!-- "jta", "thread" y "managed" -->
            <prop key="hibernate.hbm2ddl.auto">create</prop><!-- create, validate,
update -->
            <prop key="hibernate.default_schema">CLIENTES</prop>
        </props>
    </property>
</bean>
```

- LocalEntityManagerFactoryBean: Los Bean de entidad son gestionados por la aplicación, por lo que hay que definir el fichero **META-INF/persistence.xml**

```
<persistence xmlns=http://java.sun.com/xml/ns/persistence version="1.0">
    <persistence-unitname="miPersistenceUnit">
        <class>com.entidades.Persona</class>
        <properties></properties>
    </persistence-unit>
</persistence>
```

Y se define el Bean de Spring

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

2.3. Gestion de Excepciones

Se proporciona un API de excepciones mas extendido que el que proporciona JDBC, con su **SQLException**.

Hibernate, por ejemplo, también ofrece un conjunto de excepciones ampliado sobre JDBC, el problema, es que son solo útiles para este framework, si queremos independencia con el framework de persistencia, podemos emplear el API de excepciones de Spring.

Dado que muchas de las excepciones no son recuperables, Spring opta por ofrecer una jerarquía de excepciones **unchecked** (RuntimeException), es decir, ¿sino no se puede arreglar el problema

porque esa necesidad añadir try o throws?

Para que Spring capture las excepciones lanzadas por hibernate y las convierta en excecpiones de Spring, hay que definir un nuevo bean el en contexto de Spring.

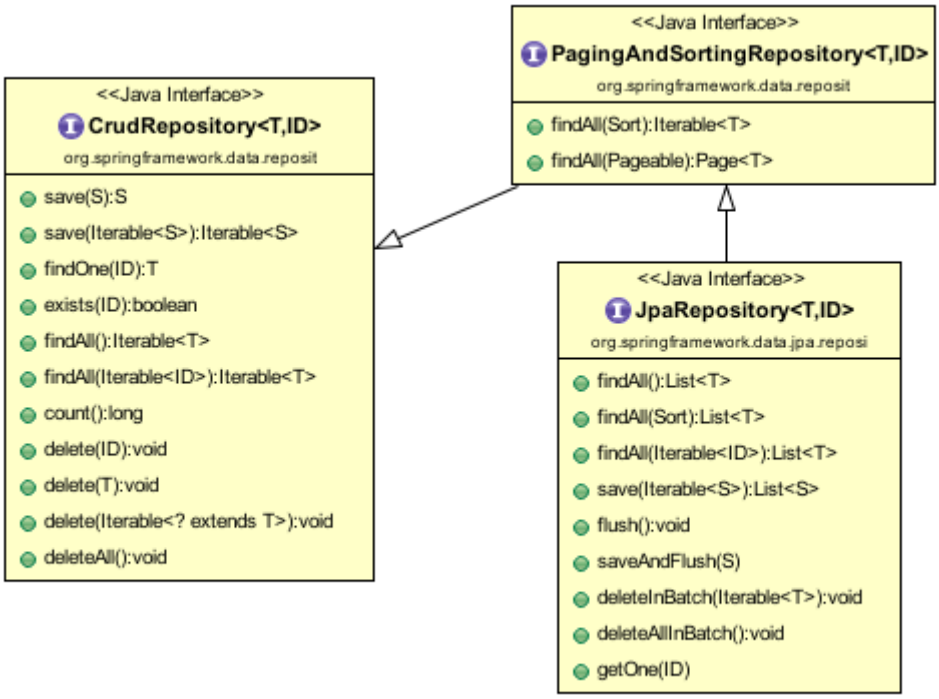
```
<bean class=
"org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>
```

Este bean, se encarga de capturar todas las excepciones lanzadas por clases anotadas con @repository y realizar la conversión.

Capítulo 3. Spring Data Jpa

Framework que extiende las funcionalidades de Spring ORM, permitiendo definir **Repositorios** de forma mas sencilla, sin repetir código, dado que ofrece numerosos métodos ya implementados y la posibilidad de crear nuevos tanto de consulta como de actualización de forma sencilla.

El framework, se basa en la definición de interfaces que extiendan la siguiente jerarquia, concretando el tipo entidad y el tipo de la clave primaria.



De forma paralela a las interfaces Jpa, existen para Mongo y Redis.

Por lo que la creación de un repositorio con Spring Data Jpa, se basará en la implementación de la interface **JpaRepository**

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {}
```

El convenio indica que el nombre de la interface, será <**entidad**>**Repository**.

La magia de Spring Data, es que no es necesario definir las implementaciones, estas se crean en tiempo de ejecución según se defina el Repositorio.

Para que esto suceda, se tendrá que añadir a la configuración con JavaConfig

```
@EnableJpaRepositories(basePackages="com.cursospring.persistencia")
```

o para configuraciones con XML

```
<jpa:repositories base-package="com.cursospring.persistencia" />
```

Además Spring Data Jpa, exige la definición de un bean de tipo **AbstractEntityManagerFactoryBean** que se llame **entityManagerFactory**.

```
<bean id="entityManagerFactory" class=
"org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="ds" />
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
    <property name="packagesToScan" value="com.curso.modelo.entidad"/>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">validate</prop>
            <prop key="hibernate.default_schema">CLIENTES</prop>
        </props>
    </property>
</bean>
```

Y otro de tipo **TransactionManager** que se llame **transactionManager**

```
<bean id="transactionManager" class="
org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Ya que son dependencias de la implementación autogenerada que proporciona Spring Data.

3.1. Querys Personalizadas

Se pueden extender los métodos que ofrezca el Repositorio, siguiendo las siguientes reglas

- Prefijo del nombre del método **findBy** para busquedas y **countBy** para conteo de coincidencias.
- Seguido del nombre de los campos de busqueda concatenados por los operadores correspondientes: And, Or, Between, … Todos los operadores aquí

```
List<Person> findByLastname(String lastname);

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);
List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
```

Tambien se pueden definir consultas personalizadas con JPQL

```
@Query("from Country c where lower(c.name) like lower(?1)")
List<Country> getByNameWithQuery(String name);

@Query("from Country c where lower(c.name) like lower(?1)")
Country findByNameWithQuery(@Param("name") String name);

@Query("select case when (count(c) > 0)  then true else false end from Country c where
c.name = ?1)")
boolean exists(String name);
```

3.2. Paginación y Ordenación

De forma analoga a la definición

A las consultas se puede añadir un último parametro de tipo **Pageable** o **Sort**, que permite definir el criterio de paginación o de ordenación.

```
Country findByNameWithQuery(Integer population, Sort sort);

@Query("from Country c where lower(c.name) like lower(?1)")
Page<Country> findByNameWithQuery(String name, Pageable page);
```

Definiendose el criterio a aplicar en tiempo de ejecución

```
countryRepository.findByNameWithQuery("%i%", new Sort( new Sort.Order(Sort.Direction
.ASC,"name")));

Page<Country> page = countryRepository.findByNameWithQuery("%i%", new PageRequest(0,
3, new Sort( new Sort.Order(Sort.Direction.ASC,"name"))));
```

3.3. Inserción / Actualización

Se pueden definir con JPQL, siempre que se cumpla

- El método debe estar anotado con @Modifying si no Spring Data JPA interpretará que se trata de una select y la ejecutará como tal.
- Se devolverá o void o un entero (int\Integer) que contendrá el número de objetos modificados o eliminados.
- El método deberá ser transaccional o bien ser invocado desde otro que sí lo sea.

```
@Transactional
@Modifying
@Query("UPDATE Country set creation = (?1)"
int updateCreation(Calendar creation);
```

```
@Transactional
int deleteByName(String name);
```

3.4. Procedimientos almacenados

Se han de declarar en la clase **@Entity** con las anotaciones

- **@NamedStoredProcedureQueries**
- **@NamedStoredProcedureQuery**
- **@StoredProcedureParameter**

```
@Entity
@Table(name = "MYTABLE")
@NamedStoredProcedureQueries({
    @NamedStoredProcedureQuery(
        name = "in_only_test",
        procedureName = "test_pkg.in_only_test",
        parameters = {
            @StoredProcedureParameter(
                mode = ParameterMode.IN,
                name = "inParam1",
                type = String.class)
        }
    ), @NamedStoredProcedureQuery(
        name = "in_and_out_test",
        procedureName = "test_pkg.in_and_out_test",
        parameters = {
            @StoredProcedureParameter(
                mode = ParameterMode.IN,
                name = "inParam1",
                type = String.class),
            @StoredProcedureParameter(
                mode = ParameterMode.OUT,
                name = "outParam1",
                type = String.class)
        })
})
public class MyTable implements Serializable { }
```

Y para emplearlas en los repositorios de **JPA Data**, se han de utilizar las siguientes anotaciones

- **@Procedure**
- **@Param**

```
public interface MyTableRepository extends CrudRepository<MyTable, Long> {

    @Procedure(name = "in_only_test")
    void inOnlyTest(@Param("inParam1") String inParam1);

    @Procedure(name = "in_and_out_test")
    String inAndOutTest(@Param("inParam1") String inParam1);
}
```

3.5. Spring Boot

Si se emplea con Spring Boot, unicamente con añadir el starter **spring-boot-starter-data-jpa**

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

Y el driver de la base de datos, por ejemplo en este caso para una base de datos H2 embebida

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

Se tendrán configurados los objetos de JPA necesarios en el contexto de Spring y se buscarán las interfaces que hereden de **Repository** en los subpaquetes del paquete base de la aplicacion Spring Boot.

Simplemente con añadir el driver de las bases de datos embebidas (H2, Derby y HSQL), por defecto se define una cadena de conexion a una base de datos del tipo correspondiente en memoria, por ejemplo para H2, los datos de conexion son los siguientes.

- **DriverClass**: org.h2.Driver
- **JDBC URL**: jdbc:h2:mem:testdb
- **UserName**: sa
- **Password**: <blank>

Se puede emplear una pequeña aplicación web que se incluye con el driver de H2 para acceder a esta base de datos y realizar pequeñas tareas de administracion.

Al ser una consola Web, se ha de añadir tambien el starter web

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

Una vez añadido, se ha de registrar el Servlet **org.h2.server.web.WebServlet** que se proporciona en dicho jar, que es el que da acceso a la aplicacion de gestion de H2.

Una alternativa para registrar el servlet es a través del **ServletRegistrationBean** *de Spring*

```
@Configuration
public class H2Configuration {
    @Bean
    ServletRegistrationBean h2servletRegistration() {
        ServletRegistrationBean registrationBean = new ServletRegistrationBean(new
WebServlet(), "/console/*");
        return registrationBean;
    }
}
```

Estando accesible la herramienta en la ruta **http://localhost:8080/console**.

3.6. Query DSL

API que permite unificar la creación de consultas en java para consumir distintos almacenes de datos, como: JPA, SQL, JDO, Lucene, MongoDB

Para poder emplear este API, se han de añadir las siguientes dependencias Maven

```
<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-apt</artifactId>
    <version>4.1.4</version>
</dependency>

<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-jpa</artifactId>
    <version>4.1.4</version>
</dependency>
```

En Spring Boot, ya se contempla esta libreria, por lo que unicamente habrá que añadir

```
<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-apt</artifactId>
</dependency>

<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-jpa</artifactId>
</dependency>
```

Y el siguiente plugin que permite generar el modelo de tipos empleado en la generación de consultas con **QueryDSL**, que son clases denominadas como las clases **Entity** con prefijo **Q**, y que

son generadas a partir de las clases **@Entity**.

```
<plugin>
    <groupId>com.mysema.maven</groupId>
    <artifactId>apt-maven-plugin</artifactId>
    <version>1.1.3</version>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>process</goal>
            </goals>
            <configuration>
                <outputDirectory>target/generated-sources/java</outputDirectory>
                <processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Una vez definido el plugin y definidas las **Entity**, se ha de lanzar el comando

```
> mvn compile
```

Para que se generen las clases **Q**.

Este modelo de clases, proporciona un objeto estatico por cada clase que permite definir las consultas.

```
Customer customer = QCustomer.customer;
LocalDate today = new LocalDate();
BooleanExpression customerHasBirthday = customer.birthday.eq(today);
BooleanExpression isLongTermCustomer = customer.createdAt.lt(today.minusYears(2));
```

Una vez generado el modelo de clases de **QueryDSL**, en proyecto **Data JPA**, se puede incluir la interface **QueryDslPredicateExecutor** a los repositorios.

```
public interface CustomerRepository extends JpaRepository<Customer, Long>,
QueryDslPredicateExecutor<Customer> {
}
```

Obteniendo un método **findAll** que permite ejecutar consultas de tipo **QueryDSL**.

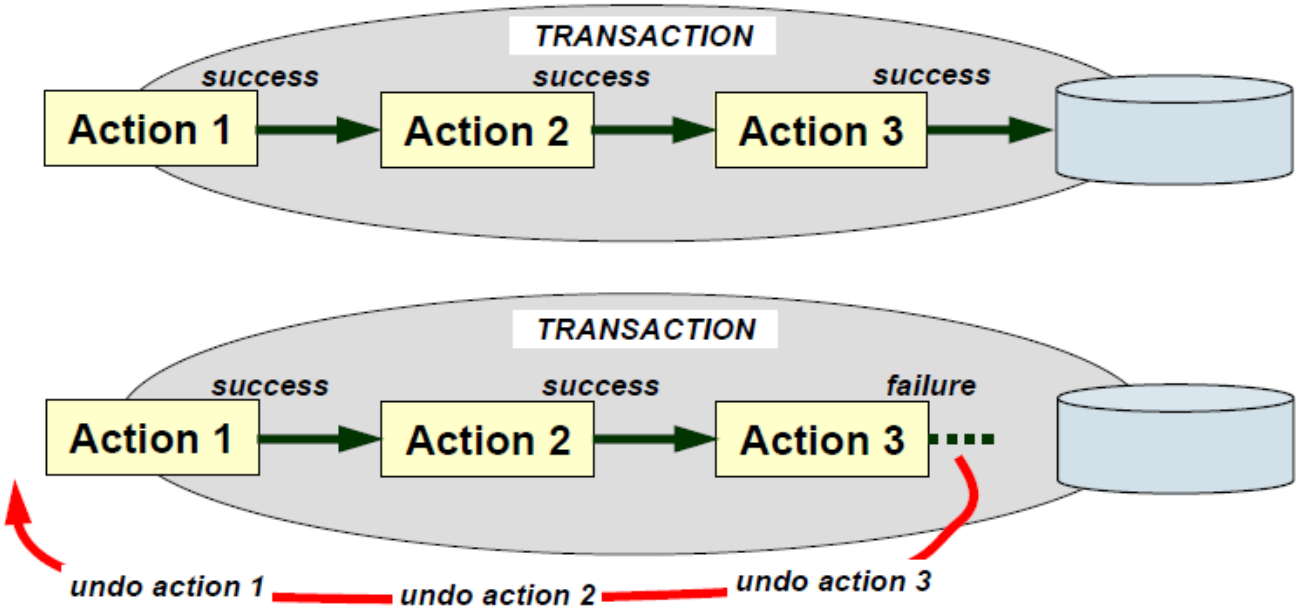
```
BooleanExpression customerHasBirthday = customer.birthday.eq(today);
BooleanExpression isLongTermCustomer = customer.createdAt.lt(today.minusYears(2));
customerRepository.findAll(customerHasBirthday.and(isLongTermCustomer));
```

Capítulo 4. Transacciones

Una transacción, es una operación de todo o nada, compuesta por una serie de suboperaciones, que se han de llevar a cabo como una unidad o no realizarse ninguna.

Características de las Transacciones (ACID)

- **Atómicas**: Todas las suboperaciones de la transacción se realizan como una unidad, si alguna no se puede llevar a cabo, no se realiza ninguna.
- **Coherentes**: Una vez finalizada la transacción, ya sea con éxito o no, el sistema queda en un estado coherente.
- **Independientes**: Deben de estar aisladas unas de otras, evitando lecturas y escrituras simultaneas.
- **Duraderas**: Los resultados de la transacción, deben ser persistidos, evitando su perdida por un fallo del sistema.



Spring es compatible con la gestión de transacciones de forma declarativa y programática.

La gestión programática, permite mayor precisión a la hora de establecer los limites de la transacción, ya que la unidad de operación de la transacción en este caso es la sentencia

La gestión declarativa, es menos precisa ya que la unidad de operación será el método, pero es más limpia ya que permite desacoplar un requisito de su comportamiento transaccional.

Para transacciones no distribuidas, cuyo ambito es un único recurso transaccional, por ejemplo la base de datos de clientes, Spring permite emplear como gestor de la transacción, el que incluye el proveedor de persistencia, Hibernate, OpenJpa, EclipseLink, …., sea este gestor JTA, o no.

Para transacciones distribuidas, se empleara una implementación de JTA.

La forma de adaptar la implementación especifica al contexto de Spring, será a través de objetos **PlatformTransactionManager** proporcionados por Spring

Spring proporciona entre otras las siguientes implementaciones

- CciLocalTransactionManager
- DataSourceTransactionManager
- HibernateTransactionManager
- JdoTransactionManager
- JpaTransactionManager
- JtaTransactionManager
- JmsTransactionManager
- WeblogicJtaTransactionManager, ...

4.1. Transacciones con JDBC

En el caso de JDBC puro, se empleará **DataSourceTransactionManager**, del cual se definirá un Bean

En XML

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

En JavaConfig

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

4.2. Transacciones con Hibernate

En el caso de Hibernate, se empleará **HibernateTransactionManager**, del cual se definirá un Bean

En el XML

```
<bean id="transactionManager" class=
"org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

En JavaConfig

```
@Bean
public HibernateTransactionManager transactionManager(SessionFactory sessionFactory){
    return new HibernateTransactionManager(sessionFactory);
}
```

4.3. Transacciones con Jpa

En el caso de Jpa, se empleará **JpaTransactionManager**, del cual se definirá un Bean

En el XML

```
<bean id="transactionManager" class="
org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

En JavaConfig

```
@Bean
public JpaTransactionManager transactionManager(EntityManagerFactory
entityManagerFactory){
    return new JpaTransactionManager(entityManagerFactory);
}
```

4.4. Transacciones con Jta

En el caso de Jta, se empleará **JtaTransactionManager**, del cual se definirá un Bean

En el XML

```
<bean id="transactionManager" class=
"org.springframework.transaction.jta.JtaTransactionManager">
    <propertyname="transactionManagerName"  value="java:/TransactionManager"/>
</bean>
```

En JavaConfig

```
@Bean
public JtaTransactionManager transactionManager(){
    JtaTransactionManager transactionManager = new JtaTransactionManager();
    transactionManager.setUserTransactionName("java:comp/UserTransaction");
    return transactionManager;
}
```

4.5. Transacciones programaticas

Spring proporciona **TransactionTemplate**, una clase que permite manejar transacciones programáticas.

Esta clase necesitará cualquiera de las implementaciones de **TransactionManager**.

```
<bean id="transactionTemplate" class=
"org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="transactionManager" />
</bean>
```

Para emplearla, se deberá incluir como dependencia en aquellos Bean que precisen manejar el comportamiento transaccional, los **commit** y **rollback**

```
<bean id="MiServicio" class="com.servicio.MiServicioImpl">
    <property name="transactionTemplate" ref="transactionTemplate"/>
</bean>
```

La clase **TransactionTemplate** ofrece el método **execute(TransactionCallback)**, que permite definir el ambito en el que se realiza la transacción, todas las operaciones que se ejecuten dentro del **TransactionCallback** estarán dentro de la misma transacción.

Para trabajar con **TransactionCallback**, existen dos posibilidades dependiendo de si la transacción retorna un resultado o no, si retorna un resultado se ha de implementar la interace **TransactionCallback<T>** y si no retorna resultado se ha implementar la clase abstracta **TransactionCallbackWithoutResult**

```
txTemplate.execute(new TransactionCallbackWithoutResult() {
    @Override
    //Dentro de este método se ejecutan todas las operaciones que conformen la Tx
    protected void doInTransactionWithoutResult(TransactionStatus status) {
        em.persist(cliente);
    }
});
```

Si no se produce ningun error en la ejecucion de las sentencias que forman la transacción, se hará **commit** y si se produce algun error, se hará **rollback**.

4.6. Transacciones declarativas

Para configurar las transacciones se emplearán las etiquetas **@Transactional** aplicadas a métodos o clases.

```
@Transactional
public void altaCliente(Cliente cliente) {
    clienteDAO.insertar(cliente);
}
```

Ojo que hay dos anotaciones una del estandar y otra de Spring, la de Spring es mas configurable, la del estandar precisa de otra anotacion **@TransactionAttribute** para configurar el comportamiento transaccional

Las cuales habrá que activar, para ello en XML

```
<tx:annotation-driven proxy-target-class="true" transaction-manager=
"transactionManager" />
```

Y en JavaConfig

```
@Configuration
@EnableTransactionManagement
public class Configuracion{}
```

4.7. Configuracion de Transacciones

La configuracion de las transacciones consiste en definir las **políticas transaccionales** de la transacción.

- **Propagation Behavior**. Limites de la transacción.
- **Isolation Level**. Nivel de aislamiento. Como la afectan otras transacciones.
- **Read Only**. En el caso de que sean de solo lectura para optimizar.
- **Timeout**. Máximo tiempo que es aceptable para la tx.
- **Normas de reversión**. Listado de excepciones que causan rollback y cuales no.

4.7.1. Propagation Behavior

Define los limites de la transacción, cuando empieza, cuando se pausa, cuando es obligatoria, ... puede ser:

- **PROPAGATION_MANDATORY**: La ejecución del método debe realizarse en una transacción. Si no la hay, se lanza una excepción.
- **PROPAGATION_NESTED**: La ejecución del método debe realizarse en una transacción anidada, no existe compatibilidad con todos los proveedores.
- **PROPAGATION_NEVER**: La ejecución del método NO debe realizarse en una transacción. Si hay una en curso se lanza una excepción.

- **PROPAGATION_NOT_SUPPORTED**: La ejecución del método NO debe realizarse en una transacción. Si hay una en curso esta se suspende.
- **PROPAGATION_REQUIRED**: La ejecución del método debe realizarse en una transacción. Si no existe una en curso, se genera una nueva.
- **PROPAGATION_REQUIRES_NEW**: La ejecución del método debe realizarse en una transacción propia, siempre se crea una transacción. Si hay una en curso se suspende.
- **PROPAGATION_SUPPORTS**: La ejecución del método no tiene porque realizarse en una transacción, pero si hay una en curso se ejecuta dentro de ella.

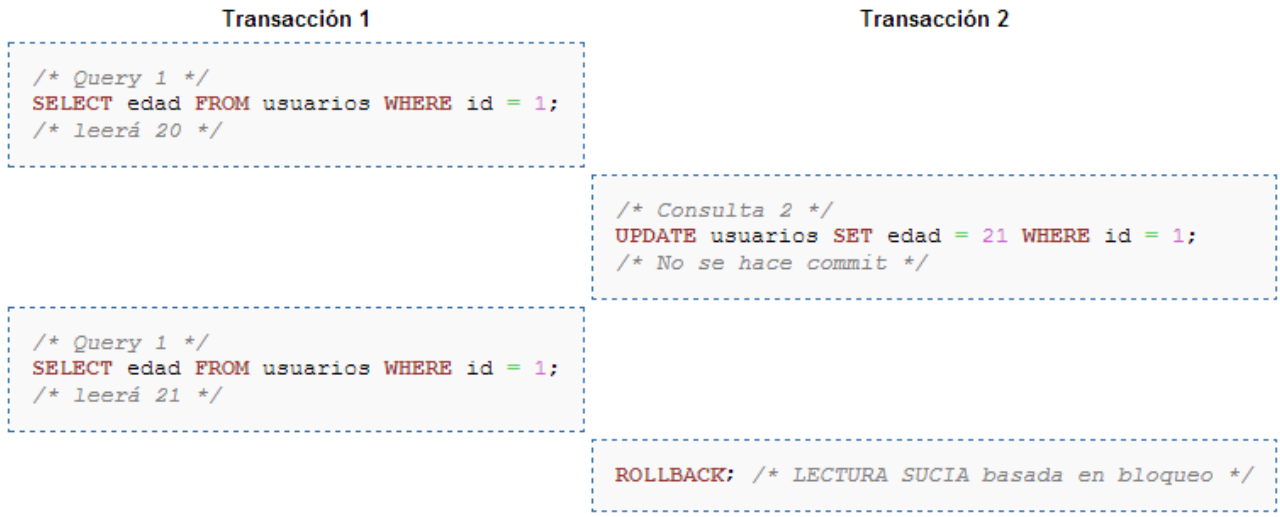
El valor por defecto es **PROPAGATION_REQUIRED**.

4.7.2. Isolation Level

Cuando varias transacciones, comparten datos, se pueden producir una serie de problemas.

- **Lectura de datos sucios**: Ocurre cuando se le permite a una transacción la lectura de una fila que ha sido modificada por otra transacción concurrente pero todavía no ha sido cometida.

Los datos leído por Tx1, han sido escritos por Tx2, pero todavía no son persistentes (commit).



- **Lectura no repetible**: Ocurre cuando en el curso de una transacción una fila se lee dos veces y los valores no coinciden.

Los datos obtenido por Tx1 al realizar la misma consulta no son iguales, dado que Tx2, los esta variando.

Transacción 1

```sql
/* Consulta 1 */
SELECT * FROM usuarios WHERE id = 1;
```

Transacción 2

```sql
/* Consulta 2 */
UPDATE usuarios SET edad = 21 WHERE id = 1;
COMMIT; /* en MVCC o READ COMMITTED basado en bloqueos */
```

```sql
/* Consulta 1 */
SELECT * FROM usuarios WHERE id = 1;
COMMIT; /* REPEATABLE READ basado en bloqueos */
```

- **Lectura fantasma**: Ocurre cuando, durante una transacción, se ejecutan dos consultas idénticas, y los resultados de la segunda son distintos de los de la primera. Es una caso particular de las lecturas no repetibles

Transacción 1

```sql
/* Consulta 1 */
SELECT * FROM usuarios
WHERE edad BETWEEN 10 AND 30;
```

Transacción 2

```sql
/* Consulta 2 */
INSERT INTO usuarios VALUES ( 3, 'Bob', 27 );
COMMIT;
```

```sql
/* Consulta 1 */
SELECT * FROM usuarios
WHERE edad BETWEEN 10 AND 30;
```

Para que esto no suceda, se puede recurrir a aislar los datos cuando estén siendo empleados por una transacción, bloqueándolos, pero esto produce una caída del rendimiento, por lo que hay que flexibilizar, permitiendo definir niveles de aislamiento.

Se definen los siguientes niveles de aislamiento

- **ISOLATION_DEFAULT**: Aplica el determinado por el almacén de datos al que accede.
- **ISOLATION_READ_UNCOMMITED**: Permite leer datos no consolidados.
- **ISOLATION_READ_COMMITED**: Permite leer datos de transacciones consolidadas. Evita lectura sucia.
- **ISOLATION_REPEATABLE_READ**: Varias lecturas del mismo campo, generan los mismos resultados excepto que la transacción lo modifique. Evita la lectura de datos sucios y la no repetible.
- **ISOLATION_SERIALIZABLE**: Aislamiento perfecto. Evita la lectura de datos sucios, la lectura no repetible y las lecturas fantasmas.

No todos los orígenes son compatibles con estos niveles.

El nivel por defecto es **ISOLATION_READ_COMMITED**.

Nivel de aislamiento	Lectura sucia	Lectura no repetibles	Lectura fantasma
Read Uncommitted	puede ocurrir	puede ocurrir	puede ocurrir
Read Committed	-	puede ocurrir	puede ocurrir
Repeatable Read	-	-	puede ocurrir
Serializable	-	-	-

4.7.3. Read Only

Si solo se van a realizar operaciones de solo lectura, Spring permite marcar esta opción, para llevar a cabo ciertas optimizaciones sobre la transacción.

Dado que esta optimización se aplica sobre una nueva transacción, tendrá sentido aplicarlo solo donde se inicien, esto es en transacciones cuya propagación sea:

- PROPAGATION_REQUIRED.
- PROPAGAION_REQUIRES_NEW.
- PROPAGATION_NESTED.

Si se trabaja con hibernate, esto implica que se empelara el modo de vaciado FLUSH_NEVER, que permite que Hibernate no sincronice con la base de datos hasta el final.

4.7.4. Timeout

Cuando se quiere limitar el tiempo durante el cual, la transacción puede tener bloqueados recursos del entorno de persistencia, será necesario especificar un tiempo máximo de espera, esto solo tendrá sentido donde se inicien, esto es en transacciones cuya propagación sea:

- PROPAGATION_REQUIRED.
- PROPAGATION_REQUIRES_NEW.
- PROPAGATION_NESTED.

4.7.5. Rollback

Conjunto de normas que definen cuando una excepción causa una reversión.

De forma predeterminada, solo se revierten las transacciones cuando hay excepciones en tiempo de ejecución (RuntimeException) y no con las comprobadas, al igual que con los EJB., aunque este comportamiento es configurable.