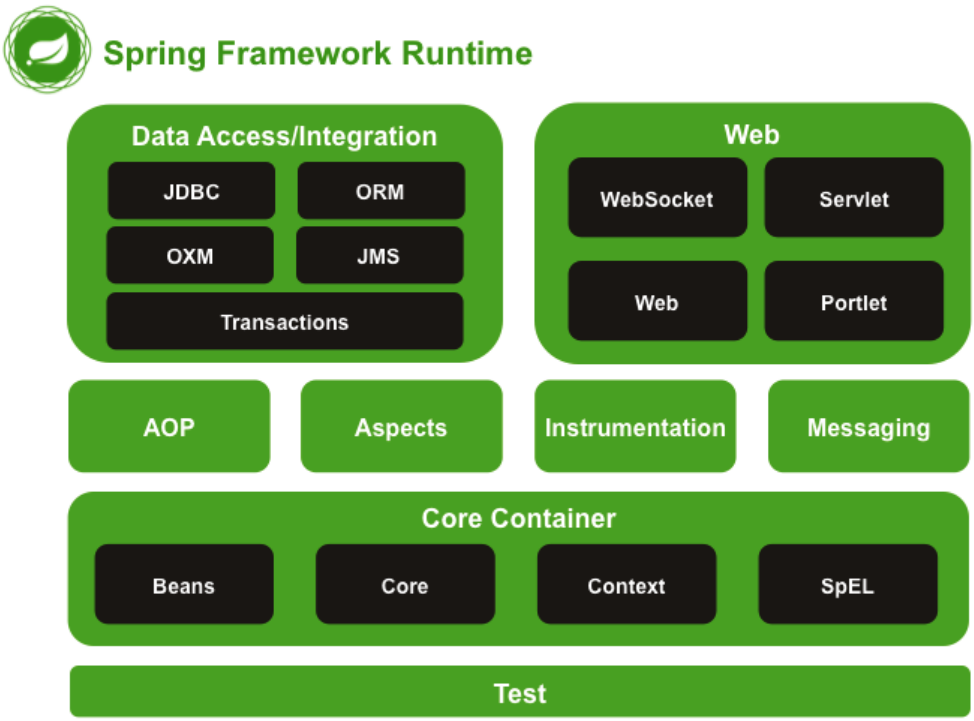
GUSTAVO CORONEL
DESARROLLA SOFTWARE

DESARROLLO WEB CON
SPRING BOOT



UNIDAD 07

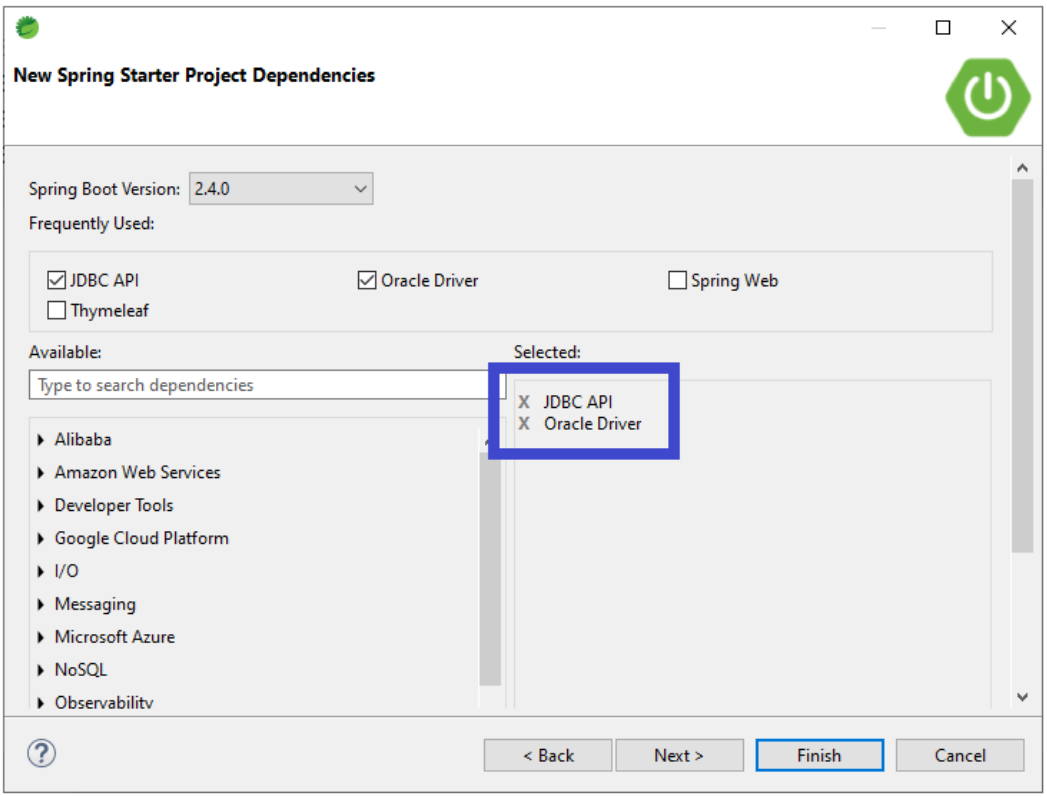
PARTE A - SPRING JDBC

Eric Gustavo Coronel Castillo
www.desarrollasoftware.com
gcoronelc@gmail.com
I N S T R U C T O R

CONTENIDO

CONFIGURACIÓN

Creación del proyecto



Dependencias

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
   <groupId>com.oracle.database.jdbc</groupId>
   <artifactId>ojdbc8</artifactId>
   <scope>runtime</scope>
</dependency>
```

Parámetros de Conexión

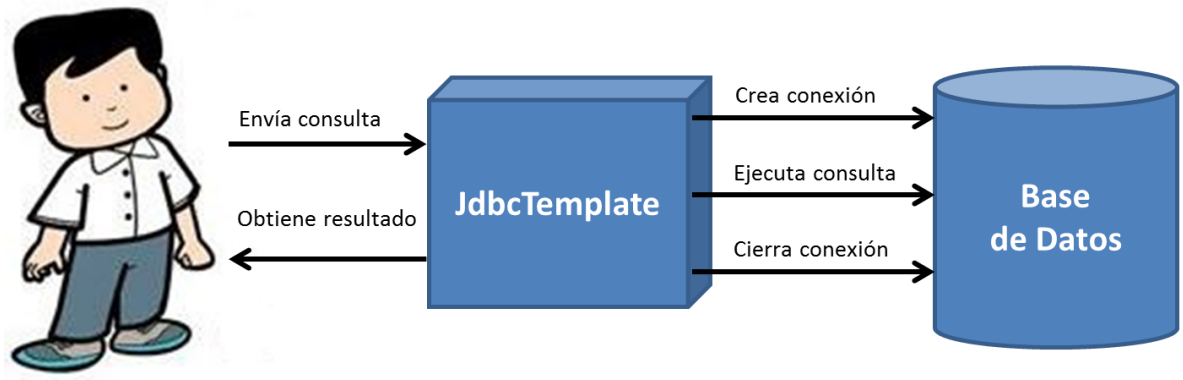
Estos parámetros se deben incluir en el archivo de propiedades (application.properties), a continuación tienes los parámetros con Oracle Database.

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@<servidor>:1521/<servicio>
spring.datasource.username=<usuario>
spring.datasource.password=<clave>
```

Este es un ejemplo:

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521/XE
spring.datasource.username=eureka
spring.datasource.password=admin
```

JDBCTEMPLATE



Vas a utilizar un objeto de tipo JdbcTemplate para ejecutar las operaciones contra la base de datos.

Este tipo de componente cuenta con todos los métodos para que puedas operar la base de datos, podrás ejecutar desde simples consultas hasta consultas complejas, sentencias insert, update y delete.

Para acceder a un objeto de este tipo debes inyectarlo en tu componente respectivo de la siguiente manera:

```
@Autowired
private JdbcTemplate jdbcTemplate;
```

A continuación, tienes un ejemplo sencillo de una consulta:

```
jdbcTemplate.query("SELECT * FROM cliente", (rs) -> {
   String paterno = rs.getString("vch_cliepaterno");
   String nombre = rs.getString("vch_clienombre");
   System.out.println(paterno + ", " + nombre);
});
```

CONSULTAS SIMPLES

Las consultas que deben retornar una fila generan una excepción de tipo `EmptyResultDataAccessException` en caso el resultado sea nulo, quiere decir cero filas de coincidencia.

Consultando un dato entero

```
String sql = "select count(*) from cuenta";
int rowCount = jdbcTemplate.queryForObject(sql,Integer.class);
System.out.println("Cantidad de cuentas: " + rowCount);
```

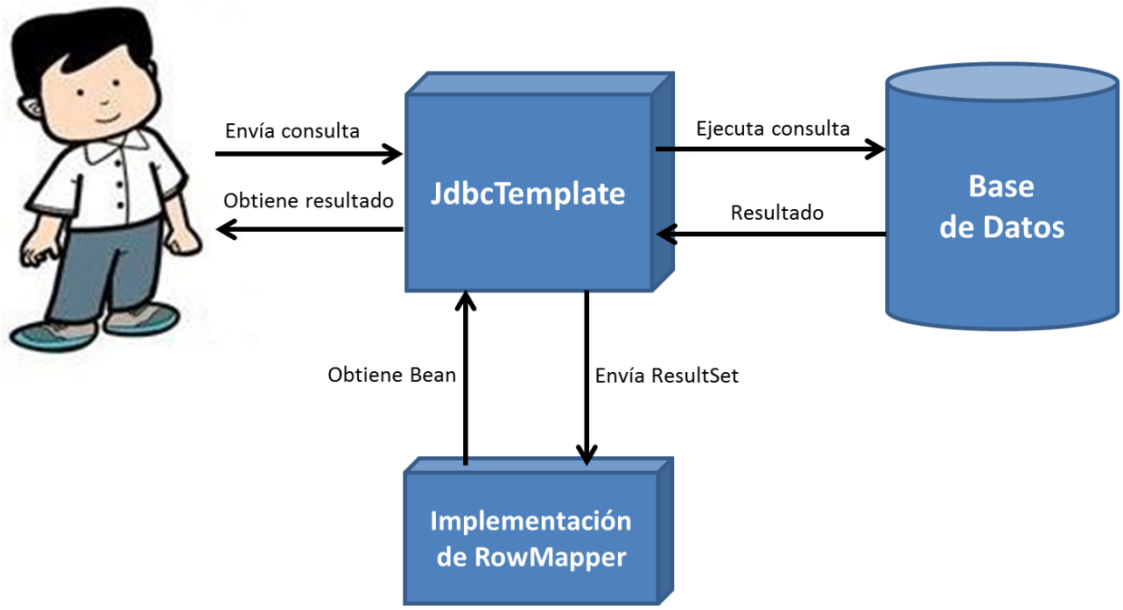
Consultando un dato double

```
String sql = "select sum(dec_cuensaldo) from cuenta";
Double saldo = jdbcTemplate.queryForObject(sql, Double.class);
System.out.println("Saldo total: " + saldo);
```

Consultando un String

```
String sql = "select vch_clienombre from cliente where chr_cliecodigo = ?";
String codigo = "00001";
String nombre = jdbcTemplate.queryForObject(sql, String.class, codigo);
System.out.println("Nombre: " + nombre);
```

USANDO ROWMAPPER



Para este caso debes utilizar una implementación de la interface `RowMapper` para pasar la fila actual de `ResultSet` a un bean.

Definición del bean

```
public class ClienteBean {

   private String codigo;
   private String paterno;
   private String materno;
   private String nombre;
   private String dni;
   private String ciudad;
   private String direccion;
   private String telefono;
   private String email;

   public ClienteBean () {
    }

   // Aquí debes crear los métodos getter y setter

}
```

Implementación de RowMapper

```
public class ClienteMapper implements RowMapper<ClienteBean> {

   @Override
   public ClienteBean mapRow(ResultSet rs, int rowNum) throws SQLException {
      ClienteBean bean = new ClienteBean();
      bean.setCodigo(rs.getString("chr_cliecodigo"));
      bean.setPaterno(rs.getString("vch_cliepaterno"));
      bean.setMaterno(rs.getString("vch_cliematerno"));
      bean.setNombre(rs.getString("vch_clienombre"));
      bean.setDni(rs.getString("chr_cliedni"));
      bean.setCiudad(rs.getString("vch_clieciudad"));
      bean.setDireccion(rs.getString("vch_cliedireccion"));
      bean.setTelefono(rs.getString("vch_clietelefono"));
      bean.setEmail(rs.getString("vch_clieemail"));
      return bean;
   }

}
```

Consultando una fila

```
String sql = "select chr_cliecodigo, vch_cliepaterno, vch_cliematerno, "
     + "vch_clienombre, chr_cliedni, vch_clieciudad, vch_cliedireccion, "
     + "vch_clietelefono, vch_clieemail "
     + "from cliente where chr_cliecodigo = ?";
String codigo = "00001";
ClienteBean bean = jdbcTemplate.queryForObject(sql,
   new ClienteMapper(), codigo );
System.out.println("Nombre: " + bean.getNombre());
System.out.println("Paterno: " + bean.getPaterno());
System.out.println("Materno: " + bean.getMaterno());
```

Debes considerar que si no encuentra ninguna coincidencia genera una excepción de tipo `EmptyResultDataAccessException.`

Consultando varias filas

Para este ejemplo se debes crear la clase `EmpleadoBean` y su respectiva clase de mapeo EmpleadoRowMapper.

El resultado se obtiene en una lista de tipo: `List<EmpleadoBean>`.

```
String sql = "select chr_cliecodigo, vch_cliepaterno, vch_cliematerno, "
     + "vch_clienombre, chr_cliedni, vch_clieciudad, vch_cliedireccion, "
     + "vch_clietelefono, vch_clieemail "
     + "from cliente where vch_cliepaterno like ?";
String paterno = "C%";
List<ClienteBean> lista = jdbcTemplate.query(sql,
   new ClienteMapper(), paterno );
for(ClienteBean bean: lista){
   System.out.println(bean.getCodigo() + " " + bean.getNombre() +
        " " + bean.getPaterno() + " " + bean.getMaterno());
}
```

USANDO MAP

También puedes obtener el resultado de una consulta en objetos de tipo `Map`.

Es muy apropiado cuando la consulta se realiza a varias tablas.

Consultando una fila

```
String sql = "select chr_cuencodigo cuenta,  chr_monecodigo moneda, "
     + "dec_cuensaldo saldo "
     + "from cuenta where chr_cuencodigo = ?";
String cuenta = "00100001";
Map<String, Object> rec = jdbcTemplate.queryForMap(sql, cuenta);
for (String key : rec.keySet()) {
   System.out.println(key + ": " + rec.get(key));
}
```

En este caso, las claves del objeto `Map` son los alias de las columnas.

Debes considerar que si no encuentra ninguna coincidencia genera una excepción de tipo `EmptyResultDataAccessException.`

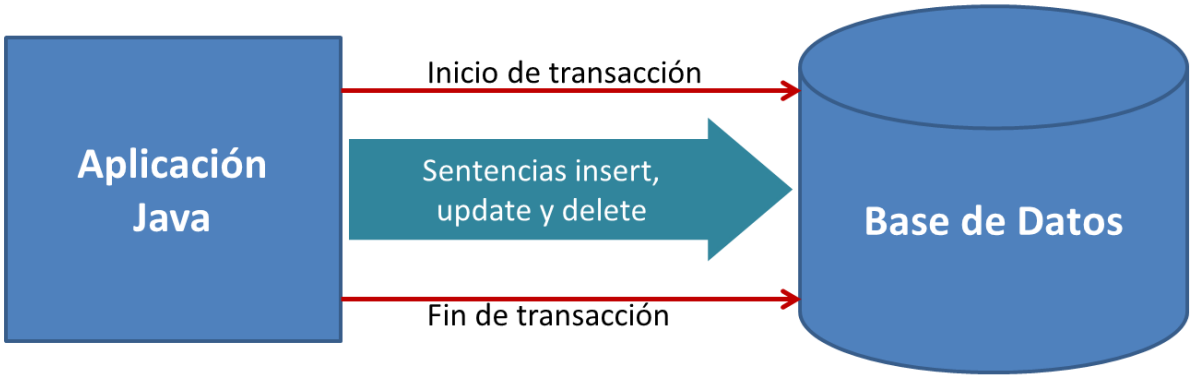
Consultar varias filas

El resultado se obtiene en una lista del siguiente tipo: `List<Map<String,Object>>`.

```
String sql = "select int_movinumero nromov, dtt_movifecha fecha, "
     + "chr_tipocodigo tipo, dec_moviimporte importe "
     + "from movimiento where chr_cuencodigo = ?";
String cuenta = "00100001";
List<Map<String,Object>> lista = jdbcTemplate.queryForList(sql, cuenta );
for(Map<String,Object> r: lista){
   System.out.println(r.get("nromov") + " " + r.get("fecha") +
      " " + r.get("tipo") + " " + r.get("importe"));
}
```

FUNDAMENTOS DE TRANSACCIONES

¿Qué es una Transacción?



Una transacción en un Sistema de Gestión de Bases de Datos (SGBD), es un conjunto de operaciones (instrucciones) que se ejecutan formando una unidad de trabajo, es decir, en forma indivisible o atómica.

Una unidad lógica de trabajo debe exhibir cuatro propiedades, conocidas como propiedades ACID (atomicidad, coherencia, aislamiento y durabilidad), para ser calificada como transacción.

- **Atomicidad**

  Una transacción debe ser una unidad atómica de trabajo, tanto si se realizan todas sus modificaciones en los datos, como si no se realiza ninguna de ellas.

- **Coherencia**

  Solo son ejecutadas aquellas transacciones que no tiene conflicto con las reglas y directrices de integridad de la base de datos.

  Cuando finaliza, una transacción debe dejar todos los datos en un estado coherente.

- **Aislamiento**

  Las modificaciones realizadas por una transacción se deben aislar de las modificaciones llevadas a cabo por otras transacciones simultáneas.

- **Durabilidad**

  Una vez concluida una transacción, sus efectos son permanentes en el sistema. Las modificaciones persisten aún en el caso de producirse un error del sistema.

Regularmente las transacciones comienzan con un `BEGIN WORK` y finalizan con un `COMMIT/COMMIT WORK` o `ROLLBACK/ROLLBACK WORK`.

Aislamiento de una Transacción

Más de un cliente puede acceder a una base de datos al mismo tiempo o superpuestos. A esto se le denomina: Concurrencia.

Frente a esta situación, surge el concepto de `“Aislamiento de una transacción”`.

Posibles situaciones de concurrencia:

- Dirty read - Lectura sucia o incorrecta
- Non-repeatable read - Lectura no repetida
- Phantom Read- Lectura fantasma

Para explicar estas situaciones utilizaremos:

- T1: Denominación de la primera transacción
- T2: Denominación de la segunda transacción

Dirty read - Lectura sucia o incorrecta

- La transacción T1 ejecuta un cambio en una fila.
- La transacción T2 lee la fila antes de que T1 ejecute el `“commit”`.
- Si T1 ejecuta un `“rollback”`, T2 habrá leído un dato que nunca existió.

Non-repeatable read - Lectura no repetida

- T1 lee una fila.
- T2 modifica o borra la misma y ejecuta `“commit”`
- Si T1 intenta volver a leer la fila verá que esta es distinta o no existe.

Phantom read - Lectura fantasma

- T1 realiza una búsqueda bajo un cierto criterio y recibe un número de filas.
- T2 ejecuta un cambio el cual va a alterar al criterio de búsqueda de T1.
- Si T1 vuelve a ejecutar la misma búsqueda bajo el mismo criterio, recibirá un número distinto de filas.

Niveles de Aislamiento

El nivel de aislamiento controla el nivel de bloqueo que ejecuta el SGBD sobre los datos.

Existen 4 niveles de aislamiento:

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

Read Uncommitted

Este es el menor nivel de aislamiento. En él se permiten las lecturas sucias, por lo que una transacción puede ver cambios aún no confirmados (`commit`) por otra transacción.

Read Committed

En este nivel de aislamiento, el SGBD que implemente el control de concurrencia basado en bloqueos mantiene los bloqueos de escritura hasta el final de la transacción, mientras que los bloqueos de lectura se cancelan tan pronto como acaba la operación de la instrucción `SELECT`, por lo que el efecto de las lecturas no repetibles puede ocurrir.

Repeatable Read

En este nivel de aislamiento, el SGBD que implemente el control de concurrencia basado en bloqueos mantiene los bloqueos de lectura y escritura hasta el final de la transacción. Sin embargo, las lecturas fantasmas pueden ocurrir.

Serializable

Este es el nivel de aislamiento más alto. Especifica que todas las transacciones ocurran de modo aislado, o dicho de otro modo, como si todas las transacciones se ejecutaran de modo serie, una tras otra. La sensación de ejecución simultánea de dos o más transacciones que perciben los usuarios sería una ilusión producida por el SGBD.

Si el SGBD hace una implementación basada en bloqueos, la serialización requiere que los bloques de lectura y escritura se liberen al final de la transacción. Del mismo modo deben realizarse bloqueos de rango sobre los datos seleccionados con `SELECT` usando WHERE- para evitar el efecto de las lecturas fantasma.

El siguiente cuadro resume los niveles de aislamiento:

NIVEL DE AISLAMIENTO	EFECTO EN LA LECTURA		
	DIRTY READ	NON-REPEATABLE READ	PHANTOM READ
READ UNCOMMITTED	POSIBLE	POSIBLE	POSIBLE
READ COMMITTED	NO POSIBLE	POSIBLE	POSIBLE
REPEATABLE READ	NO POSIBLE	NO POSIBLE	POSIBLE
SERIALIZABLE	NO POSIBLE	NO POSIBLE	NO POSIBLE

PROGRAMANDO TRANSACCIONES

Anotación @Transactional

Esta anotación se utiliza en aquellos métodos en que programas una transacción.

Configurando un Método

El siguiente ejemplo hace que el método sea transaccional.

```
@Transactional
public void insertar(ClienteBean bean) {

// Aquí programas la transacción

}
```

Definir el nivel a Aislamiento

El grado de aislamiento que tiene esta transacción para otras transacciones. ¿Por ejemplo, puede esta transacción ver registros no comprometidos (uncommited) desde otras transacciones?

La propiedad `isolation` determina el nivel de aislamiento de la transacción:

```
@Transactional( isolation=Isolation.<NIVEL> )
```

Los valores que puede tomar son:

- **DEFAULT:** Utiliza el nivel establecido en el SGBD
- **READ_UNCOMMITED**: Este nivel, permite lecturas sobre registros que aún no se han confirmado, esto se conoce como lecturas sucias, puede provocar que tus consultas contengan registros inválidos
- **READ_COMMITED**: Este nivel, hace que la transacción sólo lea registros que ya estén confirmados.
- **REPEATABLE_READ**: Este nivel no permite lecturas sobre filas que no tengan cambios confirmados, no permite situaciones donde una transacción lee un registro,

una segunda transacción altera el registro, y la primera transacción vuelve a leer el registro, obteniendo así diferentes valores la segunda ocasión.

- **SERIALIZABLE**: Este nivel no permite lecturas sucias, lecturas repetibles y lecturas fantasma, la situación donde se hace una consulta, se obtiene una serie de registros, y una transacción inserta un nuevo registro donde se satisface la condición `WHERE` de la consulta, el nuevo registro sería el fantasma.

Aquí tenemos un ejemplo de cómo utilizar esta propiedad:

```
@Transactional(isolation=Isolation.SERIALIZABLE)
public void insertar(ClienteBean bean) {

   // Aquí programas la transacción

}
```

Propagación de Transacciones

Normalmente todo el código ejecutado dentro del alcance de una transacción será ejecutado en esa transacción. Existen varias opciones que especifican el comportamiento si un método es ejecutado cuando un contexto transaccional ya existe. Por ejemplo, si un contexto transaccional ya existe podemos continuar ejecutando el método en la transacción existente o podemos suspender la transacción y crear una nueva transacción. Spring ofrece todas las opciones de propagación que ofrece EJB.

Dichas opciones de propagación se establecen en la propiedad `propagation`:

```
@Transactional( propagation=Propagation.<OPCIÓN> )
```

Los valores que puede tomar `OPCIÓN` son:

- **MANDATORY**: Este atributo obliga a la transacción a ser ejecutada en un contexto transaccional, si es que no existe un contexto transaccional en la ejecución del método Spring genera una excepción de tipo `IllegalTransactionStateException`.
- **REQUIRED**: Si el método es invocado desde un contexto transaccional, entonces el método será invocado en el mismo contexto transaccional. Si el método no es invocado desde un contexto transaccional, entonces el método creará una nueva

transacción e intentará confirmar (`commit`) la transacción cuando el método termine su ejecución.

- **REQUIRES_NEW**: El método siempre creará una nueva transacción cuando sea invocado y confirmará (`commit`) la transacción cuando el método termine su ejecución. Si ya existe un contexto transaccional, entonces Spring suspenderá la transacción existente y creará otra transacción, cuando el método termine su ejecución comprometerá la transacción y reanudará la transacción suspendida.
- **NOT_SUPPORTED**: Si el método es ejecutado en un contexto transaccional, entonces este contexto no es propagado a la ejecución del método, por lo que spring suspenderá el contexto transaccional y lo reanudará cuando el método termine su ejecución.
- **SUPPORTS**: Si ya existe un contexto transaccional, entonces el método será invocado en el mismo contexto transaccional (igual que REQUIRED), si no existe un contexto transaccional entonces no se crea un contexto transaccional (igual que NOT_SUPPORTED)
- **NEVER**: Este atributo obliga que la ejecución del método no sea invocado desde un contexto transaccional, de lo contrario Spring genera una excepción.
- NESTED: Se ejecuta dentro de una transacción anidada si un contexto transaccional existe.

Controlando la Cancelación de la Transacción

Normalmente, cuando programas transacciones con Spring esperas que realice `rollback` cuando se produce una excepción, pero, lo concreto es que por defecto solo hace `rollback` cuando se trata de excepciones no controladas, de tipo `RunTimeException` o que herede de `RunTimeException`.

Para evitar esta situación debes configurar explícitamente que realice `rollback` cuando la transacción es de tipo `Exception`, para lo cual tienes las propiedades: `rollbackFor` y `rollbackForClassName`.

A continuación, tienes un ejemplo de `rollbackFor`:

```
@Transactional(propagation=Propagation.REQUIRED, rollbackFor=Exception.class)
public void insertar(ClienteBean bean) throws Exception {

   // Proceso

}
```

Aquí tienes otro ejemplo con `rollbackForClassName`:

```
@Transactional(propagation=Propagation.REQUIRED,
             rollbackForClassName={"Exception"})
public void insertar(ClienteBean bean) throws Exception {

// Proceso

}
```

EJEMPLO ILUSTRATIVO

A continuación, tienes un ejemplo completo que permite registrar nuevos clientes:

```
@Transactional(propagation=Propagation.REQUIRED, rollbackFor=Exception.class)
public void ejmTX(ClienteBean bean) {

   // Actualizar contador
   String sql = "update contador set int_contitem = int_contitem + 1
        where vch_conttabla = 'Cliente' ";
   int filas = jdbcTemplate.update(sql);
   if (filas == 0) {
      throw new RuntimeException("El contador de la tabla CLIENTE no existe.");
   }
   // Leer datos para la generación del codigo
   sql = "select int_contitem, int_contlongitud from contador
        where vch_conttabla = 'Cliente' ";
   Map<String, Object> map = jdbcTemplate.queryForMap(sql);
   int cont = Integer.parseInt(map.get("int_contitem").toString());
   int size = Integer.parseInt(map.get("int_contlongitud").toString());
   // Generar el nuevo código
   String formato = "%0" + size + "d";
   String codigo = String.format(formato, cont);
   // Insertar el nuevo cliente
   sql = "insert into cliente(chr_cliecodigo, vch_cliepaterno, "
        + "vch_cliematerno, vch_clienombre, chr_cliedni, "
        + "vch_clieciudad, vch_cliedireccion, vch_clietelefono,"
        + "vch_clieemail) values(?,?,?,?,?,?,?,?,?)";
   Object[] args = { codigo, bean.getPaterno(), bean.getMaterno(),
                     bean.getNombre(), bean.getDni(), bean.getCiudad(),
                     bean.getDireccion(), bean.getTelefono(),
                     bean.getEmail() };
   jdbcTemplate.update(sql, args);
   // Retornar el codigo
   bean.setCodigo(codigo);

}
```

PROCEDIMIENTOS ALMACENADOS

CASOS DE IMPLEMENTAR

Caso 1: Consultar Saldo de una Cuenta

Mediante el uso de un procedimiento se procederá a consultar el saldo a una cuenta, el objetivo es ilustrarte el uso de parámetros de entrada y salida.

A continuación, se tiene el script para crear el procedimiento:

```
create or replace procedure usp_egcc_saldo_cuenta
( p_cuenta varchar2, p_saldo out number )
is
begin

   select dec_cuensaldo into p_saldo
   from cuenta
   where chr_cuencodigo = p_cuenta;

end;
/
```

A continuación, se tiene una ejecución exitosa:

```
set serveroutput on

declare
   v_saldo number;
begin
   usp_egcc_saldo_cuenta('00100001',v_saldo);
   dbms_output.put_line('Saldo: ' || v_saldo);
end;
/
```

Caso 2: Registrar un Retiro

Enunciado

Para ilustrar el uso de procedimientos almacenados se implementará el proceso "Registrar Retiro", para lo cual se necesita previamente verificar si la cuenta tiene saldo suficiente.

Suponiendo que los movimientos tienen un costo, también se registrar el costo que involucra la operación.

Implementación

El script es el siguiente:

```
create or replace procedure usp_egcc_retiro
(p_cuenta varchar2, p_importe number, p_empleado varchar2, p_clave varchar2)
as
   v_msg varchar2(1000);
   v_saldo number(12,2);
   v_moneda char(2);
   v_cont number(5,0);
   v_estado varchar2(15);
   v_costoMov number(12,2);
   v_clave varchar2(10);
   v_excep1 Exception;
begin
   select
      dec_cuensaldo, chr_monecodigo, int_cuencontmov,
      vch_cuenestado, chr_cuenclave
   into v_saldo, v_moneda, v_cont, v_estado, v_clave
   from cuenta
   where chr_cuencodigo = p_cuenta
   for update;
   if v_estado != 'ACTIVO' then
      raise_application_error(-20001,'Cuenta no esta activa.');
   end if;
   if v_clave != p_clave then
      --raise_application_error(-20001,'Datos incorrectos.');
      raise v_excep1;
   end if;
   select dec_costimporte
      into v_costoMov
      from costomovimiento
```

```
      where chr_monecodigo = v_moneda;
   v_saldo := v_saldo - p_importe - v_costoMov;
   if v_saldo < 0.0 then
      raise_application_error(-20001,'Saldo insuficiente.');
   end if;
   -- Actualiza la cuenta
   update cuenta
      set dec_cuensaldo = v_saldo,
      int_cuencontmov = int_cuencontmov + 2
      where chr_cuencodigo = p_cuenta;
   -- Movimiento de retiro
   v_cont := v_cont + 1;
   insert into movimiento(chr_cuencodigo,int_movinumero,dtt_movifecha,
      chr_emplcodigo,chr_tipocodigo,dec_moviimporte,chr_cuenreferencia)
      values(p_cuenta,v_cont,sysdate,p_empleado,'004',p_importe,null);
   -- Novimiento Costo
   v_cont := v_cont + 1;
   insert into movimiento(chr_cuencodigo,int_movinumero,dtt_movifecha,
      chr_emplcodigo,chr_tipocodigo,dec_moviimporte,chr_cuenreferencia)
      values(p_cuenta,v_cont,sysdate,p_empleado,'010',v_costoMov,null);
   -- Confirmar la Tx
   commit;
exception
   when v_excep1 then
      rollback; -- cancelar transacciÃ³n
      raise_application_error(-20001,'Clave incorrecta.');
   when others then
      v_msg := sqlerrm; -- capturar mensaje de error
      rollback; -- cancelar transacciÃ³n
      raise_application_error(-20001,v_msg);
end;
/
```

Prueba

A continuación, tienes una ejecución exitosa:

```
call usp_egcc_retiro('00100001',200,'0001','123456');
/
```

EJECUCIÓN DIRECTA

Fundamentos

Puedes usar cualquiera de los métodos `query` de `JdbcTemplate` o el método `update` para ejecutar el procedimiento almacenado.

A continuación, tienes la sintaxis:

```
jdbcTemplate.update("call procedimiento (?, ?, ...)", parámetros);
```

Ejemplo

El siguiente método ejecuta el procedimiento "`usp_egcc_retiro`" para procesar el retiro de una cuenta:

```
public void retiro
(String cuenta, double importe, String clave, String codEmp) {

   Object[] args = {cuenta, importe, clave, codEmp};
   jdbcTemplate.update("call usp_egcc_retiro(?, ?, ?, ?)", args);

}
```

CLASE STOREPROCEDURE

Fundamentos

La clase `StoredProcedure` te brinda mayores opciones de control para ejecutar procedimientos almacenados.

Para implementarlo debes crear una clase que herede de `StoredProcedure`:

```
public class ProcedureSaldoCuenta extends StoredProcedure {

   . . .

}
```

Ejemplo

El siguiente ejemplo implementa la ejecución del procedimiento `usp_saldo_cuenta`:

```
import java.sql.Types;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.StoredProcedure;
import org.springframework.stereotype.Service;

@Service
public class SaldoCuenta extends StoredProcedure {

   private static final String PROC_NAME = "usp_egcc_saldo_cuenta";

   @Autowired
   public SaldoCuenta(JdbcTemplate jdbcTemplate) {
      super(jdbcTemplate, PROC_NAME);
      setFunction(false);
      declareParameter(new SqlParameter("p_cuenta", Types.VARCHAR));
      declareParameter(new SqlOutParameter("p_saldo", Types.DECIMAL));
      compile();
```

```
  }

  public double ejecutar(String cuenta) {
     Map<String, Object> rec = super.execute(cuenta);
     return Double.parseDouble(rec.get("p_saldo").toString());
  }

}
```

A través del método `ejecutar` le pasas la cuenta y obtienes el saldo o una excepción en caso que la cuenta no exista.

A continuación, tienes un ejemplo de la ejecución:

```
  double saldo = saldoCuenta.ejecutar("00100001");
  System.out.println("Saldo: " + saldo);
```