

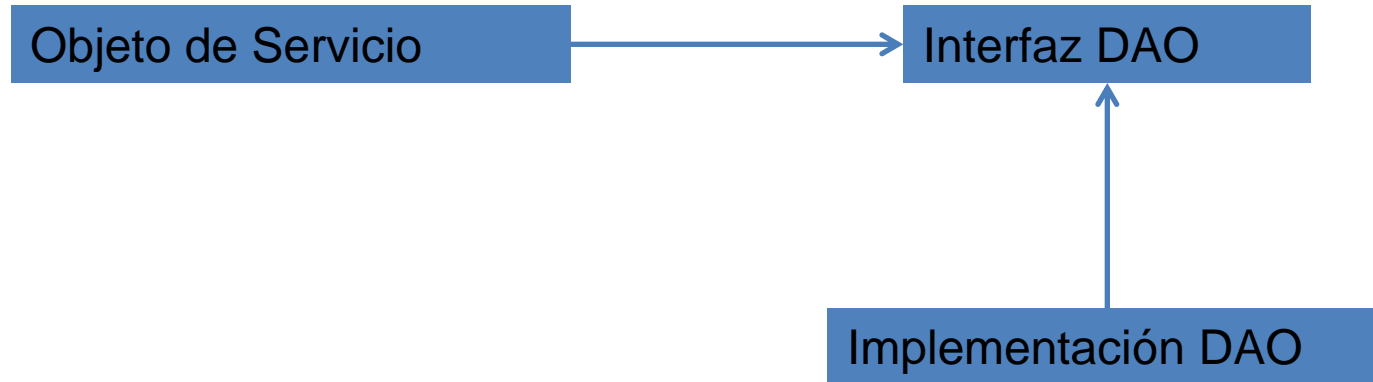
Spring JDBC

Antonio Espín Herranz

Introducción

- Spring permite la **integración con diversas tecnologías** de acceso a datos, como son:
 - JDBC puro.
 - Hibernate.
 - Ibatis.
 - JPA.
- **Spring elimina todo el código de acceso a datos mediante plantillas.**

Acceso a datos en Spring



- Los objetos de Servicio no gestionan su propio acceso a los datos, este se delega a los DAO.
- La interface DAO los mantiene acoplados de forma débil con el objeto de servicio.
- El enfoque de persistencia elegido queda oculto con el DAO y se exponen en el interface sólo los métodos relevantes.

Jerarquía de Excepciones

- **Spring** proporciona su **propia jerarquía de acceso a datos independiente de la plataforma** de persistencia elegida.
- En JDBC puro todos los errores se agrupan en `SQLException` y estamos obligados a capturarlos.
- **En el caso de Spring el desarrollador tiene la posibilidad de capturar las excepciones pero no obliga.**

Jerarquía de Excepciones

La clase principal de la jerarquía es:
DataAccessException

- CannotAcquireLockException
- CannotSerializeTransactionException
- CleanupFailureDataAccessException
- ConcurrencyFailureException
- DataAccessException
 - DataAccessResourceFailureException
 - DataIntegrityViolationException
 - DataRetrievalFailureException
 - DeadlockLoserDataAccessException
 - EmptyResultDataAccessException
 - IncorrectResultSizeDataAccessException
 - IncorrectUpdateSemanticsDataAccessException
 - InvalidDataAccessApiUsageException
 - InvalidDataAccessResourceUsageException
 - OptimisticLockingFailureException
 - PermissionDeniedDataAccessException
 - PessimisticLockingFailureException
 - TypeMismatchDataAccessException
 - UncategorizedDataAccessException

Spring JDBC

- Cuando trabajamos con BD desde el API de JDBC, mas o menos los pasos que tenemos que dar son los siguientes:
 - Definir los parámetros de conexión.
 - Abrir la conexión.
 - Especificar el comando SQL que queremos ejecutar.
 - Elaborar y ejecutar la instrucción.
 - Configurar el bucle para iterar a través de los resultados (si procede)
 - *Hacer el trabajo para cada iteración*
 - Control de excepciones.
 - Manejar las transacciones.
 - Cerrar la conexión.
 - ***Spring nos descarga de estas tareas. Se encarga de la gestión de la conexión.***

Spring JDBC

Acción	Spring	Tú
Definir los parámetros de conexión.		X
Abrir la conexión.	X	
Especificar la sentencia SQL.		X
Declarar parámetros y proporcionar valores de los parámetros.		X
Elaborar y ejecutar la sentencia.	X	
Configurar el bucle para iterar a través de los resultados (si los hay).	X	
Hacer el trabajo para cada iteración.		X
Procesar cualquier excepción.	X	
Manejar las transacciones.	X	
<u>Cerrar connection, statement y resultset.</u>	X	

Ejemplo JDBC

```
public List findByRoom(Integer roomId) {  
    List patients = new ArrayList();  
    PreparedStatement query = null;  
    String queryString = "SELECT ID, NAME, LASTNAME FROM T_PATIENT WHERE ROOMID=?";  
    try {  
        Connection connection = dataSource.getConnection();  
        query = connection.prepareStatement(queryString);  
        query.setInt(1, roomId);  
        ResultSet rs = query.executeQuery();  
        while (rs.next()) {  
            Integer id = rs.getInt("ID");  
            String name = rs.getString("NAME");  
            String lastName = rs.getString("LASTNAME");  
            ...  
        }  
    }  
}
```


Ejemplo JDBC (parte 2)

```
Patient patient = new Patient(name,lastName);
patient.setId(id);
patient.setRoomId(roomId);
patients.add(patient);
}
} catch (SQLException e) { e.printStackTrace();
} finally {
try {
if (query != null) {query.close(); }
} catch (SQLException e) {
e.printStackTrace();
}
}
return patients;
}
```

Uso de Plantillas

- El método de plantilla define el esqueleto de un proceso.
- Con independencia de la tecnología utilizada que utilizamos es necesario **llevar a cabo una serie de pasos:**
 - **Obtener conexión, limpiar recursos, etc.**
 - Estos representan **tareas comunes.**
 - **Lo que difiere son las consultas** que hacemos para recuperar, insertar, actualizar o eliminar registros.

Uso de Plantillas

- Spring divide las partes comunes de las específicas.
- En las **partes fijas** intervienen las **plantillas**.
- El **código personalizado** son las **retrollamadas**, en este caso recurre a **objetos DAO personalizados**.
- Dependiendo de la tecnología utilizada tendremos que **elegir la plantilla adecuada**:
 - JDBC → JdbcTemplate.
 - Hibernate → HibernateTemplate.
 - Jpa → JpaTemplate.

Clases de apoyo DAO

- Aparte de la plantillas (una específica para cada tecnología).
- También **incorpora clases de apoyo DAO** (con la misma filosofía, una para cada tecnología).
- En el caso de **JDBC** tenemos **JdbcDaoSupport**.
- Podemos utilizar esta clase como base para crear nuestros DAO personalizados y dispondremos de un método **getJdbcTemplate** que nos **proporciona un JdbcTemplate** para trabajar con él.

Resumiendo

1. **Antes necesitamos un origen de datos:**
DataSource
2. **Elegir la tecnología** a utilizar para la capa de persistencia: Jdbc, Hibernate, etc.
3. **Seleccionar la plantilla adecuada**, por ejemplo: **JdbcTemplate**.
4. **Seleccionar la clase de apoyo DAO:**
SimpleJdbcDaoSupport.
5. **Implementar el patrón DAO** para nuestro objeto de Servicio (interface + clase de implementación).

Configurar orígenes de datos

- Disponemos de 3 formas para definir un origen de datos:
 - Registrarlo en un servidor de aplicaciones y localizarlo **mediante JNDI**.
 - **Pool de conexiones**. Por ejemplo, el de Apache.
 - Mediante un **controlador JDBC**.

Mediante JNDI

- En este caso tendríamos que declarar el origen de datos en el fichero de configuración de Spring utilizando el espacio de nombres **jee**.
- Habría que añadir la declaración de este espacio de nombres en el fichero:

  jee - <http://www.springframework.org/schema/jee>

- Ejemplo:
 - `<jee:jndi-lookupid="datasource" jndi-name="/jdbc/MiDS" resource-ref=true />`
- El bean ya estaría disponible.

Pool de conexiones

- En caso de utilizar el pool de Apache:

```
<bean id="ds" class="org.apache.commons.dbcp.BasicDataSource">  
  <property name="url" value="jdbc:mysql://localhost:3306/ejemplos" />  
  <property name="username" value="root" />  
  <property name="password" value="antonio" />  
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />  
  <property name="initialSize" value="5" />  
  <property name="maxActive" value="10" />  
</bean>
```

Propiedades:

initialSize	Número de conexiones cuando se crea el grupo.
maxActive	máximo de conexiones activas. 0 sin límite.

Origen basado en el controlador JDBC

- El DataSource está representado por estas 4 propiedades.
 - **url**: Representa la cadena de conexión a la Bd.
 - jdbc:mysql://servidor:puerto/nombre_base_datos
 - El puerto por defecto: 3306, y el servidor: localhost / 127.0.0.1 si es nuestra propia máquina.
 - **driverClassName**: Depende de la base de datos que estemos utilizando.
 - Para MySQL: com.mysql.jdbc.Driver, el jar debe estar disponible en nuestro proyecto.
 - **userName, password**: Para conectar con la BD.

Definir el DataSource - Ejemplo

- El DataSource lo vamos a definir dentro del contexto de la aplicación (applicationContext.xml).

```
<bean id="ds"  
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
    <property name="url" value="jdbc:mysql://localhost:3306/ejemplos"></property>  
    <property name="username" value="root"></property>  
    <property name="password" value="antonio"></property>  
    <property name="driverClassName" value="com.mysql.jdbc.Driver">  
    </property>  
</bean>
```

Clases a utilizar – controlador JDBC

- Dentro de Spring disponemos de dos clases para definir el controlador JDBC.
 - **DriverManagerDataSource:**
 - **Devuelve una conexión cada vez** que se solicita una conexión.
 - **SingleConnectionDataSource:**
 - **Devuelve la misma conexión** cada vez que se solicita una.
- Este método **sólo se debería utilizar para aplicaciones pequeñas y para desarrollo** (no en producción).

Paquetes principales

- Dentro de Spring JDBC tenemos los siguientes paquetes:
 - **org.springframework.jdbc.core**
 - Nos permite acceso a base de datos de una forma cómoda.
 - La clase principal de este paquete es **JdbcTemplate**.
 - **org.springframework.jdbc.datasource**
 - Gestión y configuración de **datasources**. Los especificaremos dentro del fichero ApplicationContext.xml.
 - **org.springframework.jdbc.object**
 - Contiene clases que representan las consultas, actualizaciones y procedimientos almacenados.
 - **org.springframework.jdbc.support**
 - Se encuentra la clase SQLException. Todas las excepciones de BD se convierten a las definidas dentro del paquete: org.springframework.dao.

Trabajar con plantillas JDBC

- Disponemos de la clase:
 - **JdbcTemplate:**
 - Viene preparada para ejecutar consultas contra la BD.
 - Según queramos recuperar la cuenta, un campo o incluso mapear un objeto.
 - Nos permite recuperar colecciones de objetos de una forma sencilla.
 - Y ejecutar consultas que modifiquen los datos de las tablas.

JdbcTemplate

- Simplifica el uso de JDBC, ya que **controla** la creación y liberación de **recursos**.
- Se encarga de abrir y cerrar las conexiones.
- Trabaja con consultas SQL parametrizadas: **PreparedStatement**.
- Se puede definir dentro del contexto:

```
<bean class="org.springframework.jdbc.core.jdbctemplate" id="jdbcTemplate">  
    <constructor-arg ref="dataSource" />  
</bean>
```

JdbcTemplate - Ejemplos

- Suponemos que estamos en una clase que tiene definido un atributo del tipo:
 - `JdbcTemplate jdbcTemplate;`
 - El constructor de `JdbcTemplate` que utilizaremos será: `JdbcTemplate(dataSource dataSource);`
- Consultar el número de filas de una tabla:
`int rowCount = this.jdbcTemplate.queryForInt("select count(0) from tabla");`

JdbcTemplate - Ejemplos

- Podemos contar pasando un parámetro para hacer la consulta, en este caso estará representado por un array de objetos:
- ```
int numeroActores=
this.jdbcTemplate.queryForInt("select
count(0) from t_actors where first_name = ?",
new Object[]{"Joe"});
```
- Los parámetros se mandan como un array de objetos y se sustituyen de izq a der por las ?.



# JdbcTemplate - Ejemplos

- Podemos hacer la consulta para recuperar directamente un String.
- `String surname = (String) this.jdbcTemplate.queryForObject("select surname from t_actor where id = ?", new Object[]{new Long(1212)}, String.class);`
- En este caso estamos pasando los parámetros como un array de Object y el tipo devuelto hay que indicarlo mediante la clase (API de Reflexion).

# JdbcTemplate - Ejemplos

- También podemos recuperar directamente un objeto cargado:

```
Actor actor = this.jdbcTemplate.queryForObject(
 "select first_name, surname from t_actor where id = ?",
 new Object[]{new Long(1212)}, // Los parámetros como array de Object
 new RowMapper<Actor>() { // Lo podemos crear mediante una clase anónima.
 // de la misma forma que los eventos en Swing.
 public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
 Actor actor = new Actor();
 actor.setFirstName(rs.getString("first_name"));
 actor.setSurname(rs.getString("surname"));
 return actor;
 }
 });
```

# JdbcTemplate - RowMapper

- Para mapear el objeto tenemos que utilizar el **interface RowMapper** e implementar el **método: mapRow** que recibe un **ResultSet**.
- Dentro de este método **creamos** un **objeto** y le **asignamos** sus **propiedades** mediante los métodos de **ResultSet** (getString, getInt, etc.)
  - Los métodos ya conocidos del API de JDBC.

# JDBCTemplate - Ejemplos

- Recuperar una lista de objetos:
  - Podemos representarla por **List** o **Collection**.

Collection actors =

```
this.jdbcTemplate.query("select first_name, surname from t_actor",
new RowMapper<Actor>() {
 public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
 Actor actor = new Actor();
 actor.setFirstName(rs.getString("first_name"));
 actor.setSurname(rs.getString("surname"));
 return actor;
 }
});
```

}); // Al igual que para recuperar un object, tenemos una select que devuelve N registros y nos devuelve una collection.

# JdbcTemplate - Ejemplos

- Implementando RowMapper en otra clase, dentro de la nuestra:

```
public Collection findAllActors() {
 return this.jdbcTemplate.query("select first_name, surname from t_actor",
 new ActorMapper());
}

private static final class ActorMapper implements RowMapper {
 public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
 Actor actor = new Actor();
 actor.setFirstName(rs.getString("first_name"));
 actor.setSurname(rs.getString("surname"));
 return actor;
 }
}
```

# JdbcTemplate - Update

- Representan las **consultas de acción** de una Base de datos: Insert, Delete, Update.
- Dentro de JdbcTemplate están representadas mediante el método **update**.

- **Insertar:**

```
this.jdbcTemplate.update("insert into t_actor (first_name, surname) values (?, ?)", new Object[] {"Leonor", "Watling"});
```

- **Actualizar:**

```
this.jdbcTemplate.update("update t_actor set weapon = ? where id = ?", new Object[] {"Banjo", new Long(5276)});
```

- **Eliminar:**

```
this.jdbcTemplate.update("delete from orders");
```

- *Los posibles parámetros como siempre se pasan como un array de Object.*

# JdbcTemplate– Otras operaciones

- Ejecución de **sentencias DML**: Método `execute`.  
`this.jdbcTemplate.execute(  
 "create table mytable (id integer, name varchar(100))");`
- Ejecución de **procedimientos almacenados**:  
`this.jdbcTemplate.update(  
 "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",  
 new Object[]{new Long(unionId)});`

# Resumen – Métodos JdbcTemplate

| JdbcTemplate                                                                         | Operación sobre la Base de datos                                 |
|--------------------------------------------------------------------------------------|------------------------------------------------------------------|
| Constructor: JdbcTemplate(DataSource)                                                | -                                                                |
| Contar filas: queryForInt(SQL)                                                       | Select count(*) from tabla                                       |
| Contar filas: queryForInt(SQL, Object[]);                                            | Select count(*) from tabla where nombre='parámetro'              |
| Actualizar la BD: update(SQL, Object[])                                              | Insert into, update, delete, call (a procedimientos almacenados) |
| Crear tablas, index: execute(SQL)                                                    | Create table ... (sentencias DML).                               |
| Capturar un registro de la base de datos<br>queryForObject(SQL, Object[], RowMapper) | Select * from tabla where id = ?                                 |
| Capturar varios registros de la Base de datos<br>query(SQL, RowMapper)               | Select * from tabla                                              |
| Capturar varios registros de la BD (cond.)<br>query(SQL, Object[], RowMapper)        | Select * from tabla where pais="España"                          |



# Buenas prácticas con JdbcTemplate

- Definir el patrón **DAO**.
  - Definimos el **bean (MyBean)**: Cliente, Pedido, Factura, etc.
  - Definir un **interface** con todas las operaciones que se puede realizar sobre el Bean. **IMyBeanDAO**.
    - **Operaciones CRUD: Create, Read, Update, Delete.**
      - `public boolean create(MyBean bean);`
      - `public boolean delete(long clave);`
      - `public boolean update(MyBean bean);`
      - `public MyBean read(long clave);`
      - `public List<MyBean>selectAll();`
  - Una clase **MyBeanDAO** que implementa las operaciones. Y tiene como att. Un **JdbcTemplate**.

# Ejemplo

```
public class MyBean {
 private String nombre;
 ... // Las propiedades del Bean, con los métodos set / get / toString().
}

public interface IMyBeanDAO {
 public boolean create(MyBean bean);
 public boolean delete(long clave);
 public boolean update(MyBean bean);
 public MyBean read(long clave);
 public List<MyBean>selectAll();
}

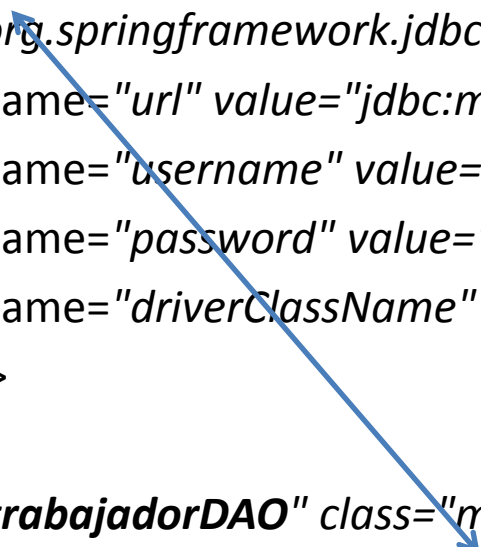
public class MyBeanDAO implements IMyBeanDAO {
 private JdbcTemplate jdbcTemplate;
 // Las operaciones del interface ...
}
```

# Buenas prácticas con JdbcTemplate

- El DAO también se registra dentro del **ApplicationContext.xml**

```
<bean id="ds"
 class="org.springframework.jdbc.datasource.DriverManagerDataSource">
 <property name="url" value="jdbc:mysql://localhost:3306/ejemplos"></property>
 <property name="username" value="root"></property>
 <property name="password" value="antonio"></property>
 <property name="driverClassName" value="com.mysql.jdbc.Driver">
 </property>
</bean>

<bean id="trabajadorDAO" class="modelo.dao.TrabajadorDAO">
 <property name="dataSource" ref="ds"></property>
</bean>
```



# Buenas prácticas con JDBCTemplate

- Cuando queremos trabajar con el DAO **recuperamos la instancia de la Factoría:**
- Lo podemos recuperar de: **ApplicationContext.**  
ApplicationContext factory = **new**  
**ClassPathXmlApplicationContext("applicationContext.xml");**  
TrabajadorDAO trabajadorDAO = (TrabajadorDAO)  
factory.getBean("trabajadorDAO");
- Indicamos el identificador declarado en el fichero de configuración.

# Uso de clases DAOsupport

- En este caso nos podemos evitar definir el objeto `JdbcTemplate` dentro de la clase del DAO.
- Para ello extendemos la clase correspondiente de **`JdbcDaoSupport`**.
- En nuestro caso será: **`JdbcDaoSupport`**.
- Dispone de un método:
  - **`getJdbcTemplate()` / `setJdbcTemplate(JdbcTemplate)`**
  - Y ya no hace falta definirlo en el contexto de Spring.

# Implementación patrón DAO con las clases de Apoyo

```
public interface SpitterDao {
 void addSpitter(Spitter spitter);
 void saveSpitter(Spitter spitter);
 Spitter getSpitterById(long id);
 List<Spittle> getRecentSpittle();
 void saveSpittle(Spittle spittle);
 List<Spittle> getSpittlesForSpitter(Spitter spitter);
 Spitter getSpitterByUsername(String username);
 void deleteSpittle(long id);
 Spittle getSpittleById(long id);
 List<Spitter> findAllSpitters();
}
```

# Implementación patrón DAO con las clases de Apoyo

```
public class SimpleJdbcSupportSpitterDao extends JdbcDaoSupport implements SpitterDao {
```

```
 public Spitter getSpitterById(long id) {
```

```
 return getJdbcTemplate().queryForObject(
```

```
 SQL_SELECT_SPITTER_BY_ID,
```

```
 new ParameterizedRowMapper<Spitter>() {
```

```
 public Spitter mapRow(ResultSet rs, int rowNum) throws SQLException {
```

```
 Spitter spitter = new Spitter();
```

```
 spitter.setId(rs.getLong(1));
```

```
 spitter.setUsername(rs.getString(2));
```

```
 spitter.setPassword(rs.getString(3));
```

```
 spitter.setFullName(rs.getString(4));
```

```
 spitter.setEmail(rs.getString(5));
```

```
 return spitter;
```

```
 }
```

```
 }, id);
```

```
 }
```

```
}
```

```
 public void addSpitter(Spitter spitter) {
 getJdbcTemplate().update(SQL_INSERT_SPITTER,
 spitter.getUsername(),
 spitter.getPassword(),
 spitter.getFullName(),
 spitter.getEmail(),
 spitter.isUpdateByEmail());
 spitter.setId(queryForIdentity());
 }
```

# Externalizar propiedades

- Interesante sacar las propiedades de configuración del **DataSource** a un **fichero .properties**.
- Por ejemplo, el fichero: **jdbc.properties**:  
database.url=jdbc:mysql://localhost:3306/ejemplos  
database.driver=com.mysql.jdbc.Driver  
database.user=root  
database.password=antonio



# Externalizar propiedades

- Dentro del fichero applicationContext.xml declaramos el bean que se encarga de la configuración de estas propiedades:

`<!-- El bean de configuración de la propiedades: -->`

```
<bean id="propertyConfigurer"
 class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
 <property name="location" value="jdbc.properties" />
</bean>
```

# Externalizar propiedades

- En la declaración de DataSource se referencia a las propiedades mediante las claves:

```
<bean id="ds"
 class="org.springframework.jdbc.datasource.DriverManagerDataSource">
 <property name="url" value="${database.url}"></property>
 <property name="username" value="${database.user}"></property>
 <property name="password" value="${database.password}"></property>
 <property name="driverClassName" value="${database.driver}"></property>
</bean>
```

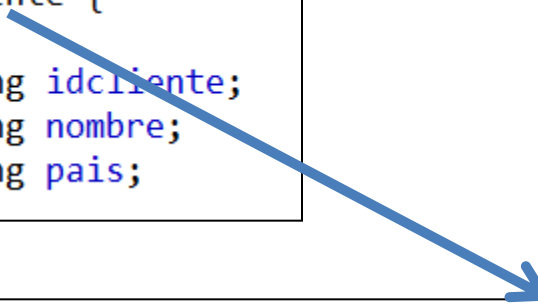
# SqlQuery

- Encapsula una consulta de SQL.
- Las subclases deben implementar el método `newRowMapper(..)` para proporcionar una instancia de `RowMapper` que pueda crear un objeto por fila para después iterar por los resultados.
- Recorre y accede directamente a los campos del `ResultSet`.

# Ejemplo

```
public class Cliente {

 private String idcliente;
 private String nombre;
 private String pais;
}
```



```
public class SelectCliente extends SqlQuery<Cliente>{

 public SelectCliente() {
 super();
 }
 public SelectCliente(DataSource ds, String sql) {
 super(ds, sql);
 }
 @Override
 protected RowMapper<Cliente> newRowMapper(Object[] parameters, Map context) {
 // TODO Auto-generated method stub
 return ParameterizedBeanPropertyRowMapper.newInstance(Cliente.class);
 }
}
```

# Ejemplo II

- En el contexto necesitamos un **datasource** para después **inyectarlo** en **SelectCliente**.

```
public static void main(String[] args) {
 ApplicationContext contexto = new ClassPathXmlApplicationContext("contexto.xml");

 DataSource ds = contexto.getBean("ds", DataSource.class);

 String sql = "select * from clientes";
 SelectCliente selectCliente = new SelectCliente(ds, sql);
 List<Cliente> resultados = selectCliente.execute();

 for (Cliente c : resultados)
 System.out.println(c);
}
```

# SqlFunction

- Con SqlFunction podemos ejecutar funciones del sistema (MySQL, Oracle, ...).
- O funciones propias que hemos creado en una base de datos.

# Ejemplo

```
public static void main(String[] args) {
 ApplicationContext contexto = new ClassPathXmlApplicationContext("contexto.xml");

 DataSource ds = contexto.getBean("ds", DataSource.class);

 String sql = "select 1+1 from dual";
 SqlFunction f = new SqlFunction(ds, sql);
 System.out.println(f.runGeneric());

 sql = "select calcularPedido2(10248)";
 f = new SqlFunction(ds, sql);
 System.out.println(f.runGeneric());
}
```

# StoreProcedure

- Encapsula la llamada a una función o procedimiento almacenado en la Base de datos.
- Disponemos de las clases:
  - SqlOutParameter: para definir parámetros de salida.
  - SqlParameter: Para definir parámetros de entrada.



# Ejemplo

```
public class ExecProcedure extends StoredProcedure {

 public ExecProcedure(DataSource ds, String name) {
 super(ds, name);
 setFunction(true);
 declareParameter(new SqlOutParameter("calcularPedido2", Types.DOUBLE));
 declareParameter(new SqlParameter("id_pedido", Types.INTEGER));
 compile();
 }

 public double execute(int id){
 Map<String, Object> params = new HashMap<>(1);
 params.put("id_pedido", id);
 Map<String, Object> paramsOut = execute(params);

 Double resul = (Double) paramsOut.get("calcularPedido2");
 return (resul != null) ? resul.doubleValue() : new Double(-1.0);
 }
}
```

# Ejemplo II

- Desde main:

```
public static void main(String[] args) {
 ApplicationContext contexto = new ClassPathXmlApplicationContext("contexto.xml");

 DataSource ds = contexto.getBean("ds", DataSource.class);
 ExecProcedure proc = new ExecProcedure(ds, "calcularPedido2");
 double resultado = proc.execute(10248);
 System.out.println("Resultado: " + resultado);
}
```