Acceso a datos con JDBC

# Introducción a Spring Framework JDBC

Una de las tecnologías de acceso a datos más conocidas y utilizadas en Java es JDBC. Vamos a ver qué problemas presenta y cómo Spring nos ayuda a hacer un mejor uso de ella mediante la JdbcTemplate.

El valor agregado proporcionado por la abstracción de Spring Framework JDBC es quizás la mejor manera de mostrar la secuencia de las acciones indicadas en la tabla de abajo. La tabla muestra las acciones que Spring se hará cargo y las acciones que son responsabilidad suya, el desarrollador de la aplicación.

Tabla 1.1Spring JDBC - ¿quién hace qué?

|  |  |  |
| --- | --- | --- |
| **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** |  |
| Cerrar connection, statement y resultset. | **X** |  |

Spring Framework se encarga de todos los detalles de bajo nivel que puede hacer JDBC como una API tediosa para desarrollar.

El principal problema de JDBC es que gran parte del código que tenemos que escribir para acceder a los datos es reduntante, propenso a errores y el manejo de excepciones es complejo. Veamos un ejemplo:

|  |  |
| --- | --- |
|  | *public class PatientDaoJdbc implements PatientDao {*    *private DataSource dataSource;*    *public PatientDaoJdbc(DataSource dataSource) {*  *this.dataSource = dataSource;*  *}*    *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");*  *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;*    *}*    *}* |

Este es el DAO que hemos utilizado en ejemplos anteriores para acceder a nuestro repositorio de pacientes. Si revisamos este método, podemos ver que realmente el código relevante (resaltado) es sólo una porción del código total. El resto es gestión de recursos (conexión) y manejo de excepciones. Es más, una vez que recibimos una excepción SQLException, ¿qué hacemos con ella? Es un tipo de error demasiado general.

# JdbcTemplate

Una práctica común cuando se utiliza la clase JdbcTemplate, es configurar un DataSource en su archivo de configuración de Spring, y luego la dependencia-inyecta el bean DataSource que compartió en sus clases DAO, el JdbcTemplate es creado en el setter por el DataSource. Esto nos lleva a que los DAOs se vean en parte a lo siguiente:

Veamos como la JdbcTemplate nos ayuda a simplificar nuestro código y manejar las excepciones de una forma más organizada. Supongamos que queremos implementar un nuevo método que nos diese el número de pacientes en una habitación. Veamos cómo quedaría nuestro DAO utilizando JdbcTemplate:

|  |  |
| --- | --- |
|  | *public class PatientDaoJdbc implements PatientDao {*    *private JdbcTemplate jdbcTemplate;*    *public PatientDaoJdbc(DataSource dataSource) {*  *// Necesita un DataSource que apunte al repositorio*  *this.jdbcTemplate = new JdbcTemplate(dataSource);*  *}*    *public Integer countByRoom(Integer roomId) {*  *return jdbcTemplate.queryForInt("SELECT COUNT(\*) FROM T\_PATIENT WHERE ROOMID=?", roomId);*  *}*    *}* |

Como podemos observar, gran parte del código queda encapsulado dentro de la jdbcTemplate: establecer la conexión, transaccionalidad, ejecución de la query, procesamiento de la respuesta, manejo de excepciones y liberar la conexión. En esta consulta, además, estamos utilizando una variable bind (roomId) que pasamos como parámetro en la llamada.

Estamos realizando una consulta que devuelve un tipo básico (int). ¿Qué ocurre si queremos consultar para obtener varios campos? El primer acercamiento es utilizar un mapa genérico, en el ejemplo vamos a consultar el listado de todos los pacientes de una habitación:

|  |  |
| --- | --- |
|  | public List findByRoomGeneric(Integer roomId) {    return jdbcTemplate.queryForList("SELECT \* FROM T\_PATIENT WHERE ROOMID=?", roomId);  } |

El resultado será un objeto List que contendrá una serie de Map genéricos, si lo imprimimos por pantalla: *[{ID=1, NAME=Marshall, LASTNAME=Zieman, ROOMID=1}, {ID=2, NAME=Russ, LASTNAME=Dunham, ROOMID=1}]*

Este método del DAO utilizará la misma plantilla que el anterior, la idea es crear una única plantilla y reutilizarla una vez construida (JdbcTemplate es thread-safe).

Ahora veamos cómo podríamos devolver un listado de entidades de la clase Patient, para esto JdbcTemplate utiliza callbacks:

|  |  |
| --- | --- |
|  | public List findByRoom(Integer roomId) {    return jdbcTemplate.query("SELECT ID, NAME, LASTNAME FROM T\_PATIENT  WHERE ROOMID=?", new PatientMapper(), roomId);  }    ...    // Mapea cada fila del ResultSet a un objeto Patient  class PatientMapper implements RowMapper {    public Patient mapRow(ResultSet rs, int row) throws SQLException {      Patient patient = new Patient();      patient.setId(rs.getInt(1));      patient.setName(rs.getString(2));      patient.setLastName(rs.getString(3));      return patient;    }  } |
|  |  |

Lo único que hemos tenido que indicar a la JdbcTemplate además de la consulta, es la forma de mapear cada fila a un nuestro modelo Patient implementado el interfaz RowMapper.

¿Y si no hubiese que devolver ningún resultado? Por ejemplo, procesar una a una las filas del resultado y generar un informe con ellas. Para ello, Spring proporciona el callback RowCallbackHandler.

|  |  |
| --- | --- |
|  | public void generateReport() {    jdbcTemplate.query("SELECT \* FROM T\_PATIENT", new PatientRowCallbackHandler());  }  ...  // Procesa filas de una query  class PatientRowCallbackHandler implements RowCallbackHandler {      public void processRow(ResultSet rs) throws SQLException {      System.out.println("--Paciente--");      System.out.println("Nombre: " + rs.getString(2));      System.out.println("Apellidos: " + rs.getString(3));      System.out.println("Habitación: " + rs.getInt(4));    }    } |

¿Qué ocurre cuando queremos procesar todas las filas de una misma consulta a la vez? Para ello utilizaremos el interfaz ResultSetExtractor. Con este método somos nosotros los encargados de iterar el ResultSet:

|  |  |
| --- | --- |
|  | // Procesa el resultado de una sóla vez  private class PatientResultSetExtractor implements ResultSetExtractor<Set> {      public Set extractData(ResultSet rs) throws SQLException, DataAccessException {         Set patientsSet = new HashSet();       while(rs.next()) {          Patient patient = new Patient();      patient.setId(rs.getInt(1));      patient.setName(rs.getString(2));      patient.setLastName(rs.getString(3));      patientsSet.add(patient);       }       return patientsSet;    }  } |

Por último, vemos como realizar inserciones y actualizaciones:

|  |
| --- |
| public int save(Patient patient) {     return jdbcTemplate.update("INSERT INTO T\_PATIENT(ID, NAME, LASTNAME, ROOMID)  VALUES (?,?,?,?)",patient.getId(), patient.getName(),  patient.getLastName(), patient.getRoomId());  }    public int updateRoom(Patient patient) {     return jdbcTemplate.update("UPDATE T\_PATIENT SET ROOMID=? WHERE ID=?",  patient.getRoomId(), patient.getId());  } |
|  |

Por supuesto, JdbcTemplate mapea los errores a la jerarquía de excepciones de DataAccessException.