BandTec
DIGITAL SCHOOL

Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Linguagem de Programação

Introdução ao Spring JDBC

Sumário

Spring JDBC

Um dos motivos da popularização da plataforma Java no final da década de 1990 e no início dos anos 2000 foi a introdução de uma tecnologia chamada **JDBC** (***J****ava* ***D****ata****B****ase* ***C****onnectivity*) a partir da versão 1.1 da JDK, em 1997 - num momento onde o uso de bancos de dados relacionais estava consolidado e em expansão -. Trata-se de uma API com a qual seria possível usar as mesmas interfaces, classes e métodos para acessar virtualmente qualquer SGBD relacional.

Assim, caso fosse necessário acessar tabelas parecidas, mas em outro servidor, bastaria alterar uma mínima parte do código, relativa às configurações de acesso somente. Operações de criação, alteração, recuperação e exclusão de registros das tabelas teriam seus códigos mantidos, não importando o fabricante do banco de dados. Enfim, a JDBC trazia a real possibilidade de portabilidade de banco da dados em projetos Java.

O segredo é que a JDBC define um conjunto de interfaces com operações comuns a todos os SBGDs relacionais. Então, é possível criar implementações, chamadas de Drivers JDBC, para qualquer servidor de banco de dados. Esse comportamento está ilustrado na Figura 1.

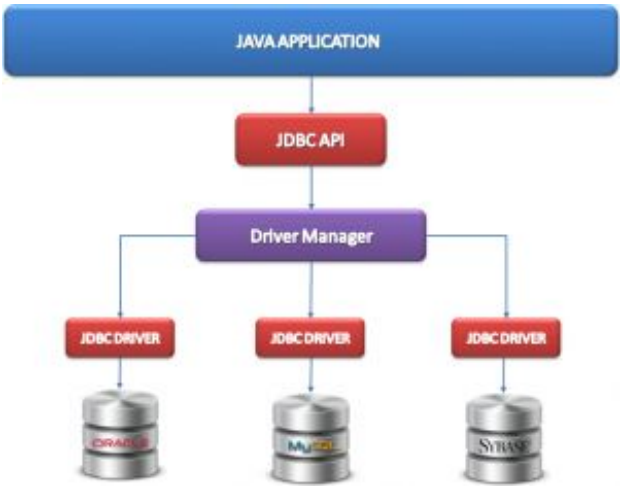

Figura 1. Funcionamento da JDBC
Fonte: https://avaldes.com/connecting-to-sqlserver-using-jdbc/

Apesar de facilitar a portabilidade de código entre diferentes SGBDs, a JDBC é uma API um tanto verbosa e sem grandes facilidades no controle de transações, na conversão de tipos e na conversão de resultados em objetos de determinada classe. Por conta disso, podemos usar o **Spring JDBC**, que é uma API que abstrai muitas funcionalidades da JDBC, raz funcionalidades muito úteis e é facilmente integrada a um projeto com Spring.

A seguir veremos como configurar o Spring JDBC num projeto Maven e como usar sua principais funcionalidades.

Configuração num projeto Maven

Num projeto com Maven, basta adicionar as seguintes dependências para usufruir da Spring JDBC:

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-jdbc</artifactId>
	<version>5.0.5.RELEASE</version>
</dependency>

<dependency>
	<groupId>org.apache.commons</groupId>
	<artifactId>commons-dbcp2</artifactId>
	<version>2.5.0</version>
</dependency>
```

Além da dependência do Spring JDBC, precisamos incluir uma dependência do driver de algum banco de dados. O código fonte de um **pom.xml** a seguir tem alguns exemplos de dependências de drivers de alguns dos principais bancos do mercado.

```
<!-- MySQL e MariaDB -->
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
	<version>6.0.6</version>
</dependency>

<!-- SQL Server -->
<dependency>
	<groupId>com.microsoft.sqlserver</groupId>
	<artifactId>mssql-jdbc</artifactId>
	<version>6.4.0.jre8</version>  <!-- ou versão 7.2.1.jre11, se sua JDK for a 11 -->
</dependency>
```

Com a inclusão da(s) dependência(s), devemos configurar a injeção do Spring JDBC no projeto. Para isso, basta criarmos um objeto da classe **BasicDataSource** (pacote **org.apache.commons.dbcp2**), como no código fonte a seguir, comum exemplo para um **SQL Server** no **Azure**.

```
BasicDataSource dataSource = new BasicDataSource();

dataSource.setDriverClassName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
// exemplo para MySql: "com.mysql.cj.jdbc.Driver"

dataSource.setUrl("jdbc:mysql://meubanco.database.windows.net/meubanco");
// exemplo para MySql: "jdbc:mysql://localhost:3306/meubanco"

dataSource.setUsername("meulogin");

dataSource.setPassword("minhasenha");
```

No objeto **dataSource**, nós configuramos:

- A **classe do Driver JDBC** (Driver de Conexão com o SGBD);
- A **URL de conexão com o banco** (endereço pelo qual o Driver vai tentar a conexão);
- As **credenciais** de acesso ao banco (usuário e senha).

De onde vêm esses valores?

Cada Driver de conexão com banco de dados possui esses valores. Basta consultar a documentação de cada um. No geral, precisamos descobrir quais a classe do Driver JDBC e como criar uma URL de conexão com o banco.

Estudo de caso: Tabela “tbl_pais”

Para melhor exemplificar as funcionalidade do Spring JDBC, vamos usar uma tabela chamada “tbl_pais”, cuja estrutura é possível ver na Figura 1.

Figura 2: Estrutura da tabela “tbl_pais”

Classe para manipulação de “tbl_pais”

Vamos criar uma classe chamada **PaisCrud**, para aprendermos como usar o Spring JDBC. O código dessa classe ficaria como o do código a seguir.

```
public class PaisCrud {

    private JdbcTemplate jdbcTemplate;

    private PaisCrud() {
        BasicDataSource dataSource = new BasicDataSource();
        // configuração do dataSource, como visto antes

        jdbcTemplate = new JdbcTemplate(dataSource);
    }

}
```

Note que declaramos um atributo do tipo **JdbcTemplate** (pacote **org.springframework.jdbc.core**) usando um **dataSource** do tipo **BasicDataSource** como o criado no código anterior. Essa classe fornece métodos para a realização das principais operações junto a um banco de dados. Seus métodos atuarão usando as configurações de acesso ao banco definidas.

Consulta - Recuperando vários os registros da tabela

A seguir, vejamos as diferentes formas de realizar uma consulta que recupera vários registros de uma tabela. Vejamos isso num método que recupera todos os registros da tabela. Vide o próximo código.

```
import org.springframework.jdbc.core.BeanPropertyRowMapper;

public class PaisCrud {

    // atributo e construtor

    public List listarTodos() {
        List<Map<String, Object>> lista = jdbcTemplate.queryForList("select * from tbl_pais");
        return lista;
    }
}
```

O método **listarTodos()** usa o método **queryForList()** do **jdbcTemplate** para recuperar uma **List** de objetos do tipo **Map**. Em cada **Map** a chave será o nome do campo e o valor será o valor do registro.

O argumento usado no **queryForList()** é a instrução SQL que será enviada ao SGBD. Caso a consulta não encontre nenhum registro, o método retorna uma **List vazia** e não **null**.

Ao invocar esse método e imprimirmos seu retorno, teríamos como resposta parecida com um JSON como o do exemplo a seguir, caso existam registros na **tbl_pais**:

```
[
  {
    "id_pais": 1,
    "nome_pais": "BRASIL",
    "extensao_km2_pais": 8516000
  },
  {
    "id_pais": 2,
    "nome_pais": "JAPÃO",
    "extensao_km2_pais": 378000
  }
]
```

Consulta - Recuperando vários registros da tabela em objetos de Pais

Vamos supor que seja necessário usar uma classe para representar um registro da tabela **tbl_pais**. Vamos criar a classe **Pais**, cujo código está a seguir.

```
public class Pais {

        private Integer idPais;
        private String nomePais;
        private Double extensaoKm2Pais;

        // getters e setters
 }
```

Note que, basicamente, trocamos o padrão *snake_case* dos campos da tabela por *camelCase* nos atributos da classe (vide o Quadro 1). Fazendo isso temos usufruímos de uma interessante funcionalidade do Spring JDBC, que veremos a seguir.

	Tabela	**Classe**
Nome	tbl_pais	Pais
Campos x Atributos	id_pais	idPais
	nome_pais	nomePais
	extensao_km2_pais	extensaoKm2Pais

Quadro 1: Nomes de tabela, classe, campos e atributos entre “tbl_pais” e “Pais”

Então, nosso método **listarTodos()** passaria a recuperar todos os registros como no próximo código fonte.

```
// importação necessária:
import org.springframework.jdbc.core.BeanPropertyRowMapper;

List<Pais> lista = jdbcTemplate.query("select * from tbl_pais",
                                      new BeanPropertyRowMapper(Pais.class));
```

As diferenças começam no uso de outro método da **jdbcTemplate**, o **query()**. O primeiro argumento é a instrução SQL da consulta e o segundo é um objeto do tipo **BeanPropertyRowMapper** (pacote **org.springframework.jdbc.core**) que é criado a partir de um construtor que recebe a classe **Pais**.

A criação de um **BeanPropertyRowMapper** usa a classe informada no construtor para fazer o método **query()** retornar uma lista do tipo dessa classe. Para que a "mágica" ocorra:

a) Os tipos dos atributos na classe devem ser compatíveis com os tipos dos campos na tabela;
b) Os nomes dos atributos devem ser "iguais" aos dos campos da tabela. A diferença é que os atributos devem ser *camelCase* e não *snake_case*, como os campos.

A classe não precisa ter todos os campos da tabela "refletidos" nela. Bastam os desejados. E atributos da classe que são possuem "xará" na tabela serão simplesmente ignorados.

Consulta - Recuperando registros usando Pais e atributos de nomes diferentes

Vamos supor que seja necessário usar nomes de atributos da diferentes dos nomes dos campos da tabela. Vamos alterar a classe **Pais**, cujo novo código está a seguir.

```
public class Pais {

	private Integer id;
	private String nome;
	private Double extensao;

	// getters e setters

}
```

Note que os atributos agora estão com nomes bem diferentes dos campos da tabela. Basicamente "resumimos" os nomes. A seguir, precisamos criar uma classe que implemente a **RowMapper** (pacote **org.springframework.jdbc.core**), conforme o código fonte a seguir.

```java
import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class PaisRowMapper implements RowMapper<Pais>{

	@Override
	public Pais mapRow(ResultSet rs, int rowNum) throws SQLException {
		Pais pais = new Pais();
		pais.setId(rs.getInt("id_pais"));
		pais.setNome(rs.getString("nome_pais"));
		pais.setExtensao(rs.getDouble("extensao_km2_pais"));
		return pais;
	}

}
```

A classe **PaisRowMapper** implementa a **RowMapper** informado a classe **Pais** como "referência". Somos obrigados a implementar o método **mapRow()**, no qual "ensinamos" a preencher os atributos de **Pais** conforme os campos da tabela.

Basicamente preenchemos os atributos de **Pais** usando seus *setters* com valores recuperados do objeto **rs**, que é um **ResultSet** (pacote **java.sql.ResultSet**). Ao final, retornamos o objeto **pais** do tipo **Pais**, devidamente preenchido. Se uma consulta retornar vários países, esse método será invocado conforme a quantidade de registros encontrados.

Feito tudo isso, nosso método **listarTodos()** passaria a recuperar todos os registros como no próximo código fonte.

```java
// importar a classe PaisRowMapper caso tenha criado em outro pacote

List<Pais> lista = jdbcTemplate.query("select * from tbl_pais", new PaisRowMapper());
```

Usamos novamente o método **query()** do **jdbcTemplate**. A diferença é que o segundo argumento é um objeto da classe **PaisRowMapper**, que acabamos de criar.

Consulta - Recuperando registros numa consulta com parâmetros

O Spring JDBC oferece uma forma muito simples de realizar consultas que precisam de parâmetros. A seguir, vamos ver e analisar o código fonte de um método em **PaisCrud** que configura um EndPoint para recuperação de um registro de **tbl_pais** a partir de sua chave primária.

```
// importação necessária:
import org.springframework.dao.EmptyResultDataAccessException;

public Map<String, Object> recuperar(int id) {
   try {
        Map<String, Object> registro = jdbcTemplate.queryForMap(
                                        "select * from tbl_pais where id_pais = ?", id);
        return registro;
   } catch (EmptyResultDataAccessException e) {
        return null;
   }
}
```

No método recém criado, o **recuperar()**, usamos o **jdbcTemplate** para recuperar um registro de **tbl_pais** usando o método **queryForMap()**. No primeiro argumento passamos a instrução SQL da consulta. No segundo argumento passamos o valor do argumento.

Note que, na instrução SQL o parâmetro é representado por uma interrogação (**?**). Como só há uma interrogação, o segundo argumento, o objeto **id**, será usado como seu valor. Se houvessem mais de um parâmetro, bastaria passarmos quantos valores forem necessários após a instrução SQL.

O retorno do **queryForMap()** é um objeto do tipo **Map** no qual as chaves são os nomes dos campos e os valores serão os valores dos registros. Caso a consulta não retorne valor algum será lançada uma **EmptyResultDataAccessException** (pacote **org.springframework.dao**), motivo pelo qual criamos o bloco **try-catch**.

Detalhes Importantes sobre os parâmetros

- A interrogação nunca deve estar entre aspas, mesmo que o parâmetro seja um valor alfanumérico;
- Não nos preocupamos com tipos. O Spring JDBC faz todas as conversões necessárias para que o parâmetro seja enviado da forma correta para o SGBD.

Consulta - Recuperando registros com parâmetros em objetos de Pais

Vamos supor que seja necessário usar uma classe para representar um registro da tabela **tbl_pais**. Tomemos como exemplo a classe **Pais** criada no tópico **Consulta - Recuperando vários registros da tabela em objetos de Pais**.

Então, nosso método **recuperar()** passaria a recuperar um registro como no próximo código fonte.

```
Pais registro = jdbcTemplate.queryForObject("select * from tbl_pais where id_pais = ?", new
BeanPropertyRowMapper<Pais>(Pais.class), id);
```

Como já vimos, o uso da **BeanPropertyRowMapper** junto do fatos dos atributos de **Pais** estarem "compatíveis" com os dos campos da **tbl_pais**, faz a "mágica" da criação do objeto com os valores corretamente preenchidos.

Consulta - Recuperando registros com parâmetros e atributos de nomes diferentes

Vamos supor que seja necessário usar nomes de atributos da diferentes dos nomes dos campos da tabela. Vamos alterar a classe **Pais** conforme já fizemos no tópico **Consulta - Recuperando registro usando Pais e atributos de nomes diferentes**.

Então, nosso método **recuperar()** passaria a recuperar um registro como no próximo código fonte.

```
Pais registro = jdbcTemplate.queryForObject("select * from tbl_pais where id_pais = ?", new
PaisRowMapper(), id);
```

A **PaisRowMapper** no último código é a mesma que criamos no tópico **Consulta - Recuperando registro usando Pais e atributos de nomes diferentes** e faz a mesma ação nele.

Consulta - Parâmetros nomeados

Nos últimos 3 exemplos, o parâmetro era representado na instrução SQL por uma interrogação (**?**). Porém, é possível dar nomes aos parâmetros. Para isso, temos que nomear os parâmetros com o prefixo "**:**" e usar a classe **SqlParameterSource** (pacote **org.springframework.jdbc.core.namedparam**). Vide como ficariam as últimas três consultas se usássemos esse recurso.

```
// importação necessária
import org.springframework.jdbc.core.namedparam.SqlParameterSource;

SqlParameterSource parametros = new MapSqlParameterSource().addValue("pid", id);

// Consulta que retorna um Map
Map<String, Object> registro = jdbcTemplate.queryForMap("select * from tbl_pais where
id_pais = :pid", parametros);

// Consulta que retorna um Pais com nomes de atributos parecidos com dos campos
Pais registro = jdbcTemplate.queryForObject("select * from tbl_pais where id_pais = :pid",
parametros,  new BeanPropertyRowMapper<Pais>(Pais.class));

// Consulta que retorna um Pais com nomes de atributos de nomes personalizados
Pais registro = jdbcTemplate.queryForObject("select * from tbl_pais where id_pais = :pid",
parametros, new PaisRowMapper());
```

Criamos a configuração do parâmetro nomeado criando um objeto **parametros** do tipo **SqlParameterSource**. Ao instanciar um **MapSqlParameterSource** nós imediatamente adicionamos o parâmetro **pid**. Caso houvessem outros parâmetros, bastaria invocar o método **addValue()** para cada um deles, informando sempre seu nome e seu valor.

No primeiro código de consulta, o que retorna um **Map**, o parâmetro nomeado na instrução SQL é o "**:pid**". Na invocação do **queryForObject()**, o segundo argumento passou a ser o objeto de parâmetros nomeados, o **parametros**.

No segundo código de consulta, o que retorna uma **List** de **Pais** com atributos de "mesmo" nome dos campos da tabela, o parâmetro nomeado na instrução SQL é o "**:pid**". Na invocação do **queryForObject()**, o segundo argumento passou a ser o objeto de parâmetros nomeados, o **parametros**.

No terceiro código de consulta, o que retorna uma **List** de **Pais** com atributos de nomes personalizados, o parâmetro nomeado na instrução SQL é o "**:pid**". Na invocação do **queryForObject()**, o segundo argumento passou a ser o objeto de parâmetros nomeados, o **parametros**.

Criação - Inserindo um registro na tabela

A seguir, vejamos as diferentes formas de realizar um "insert" numa tabela. Vejamos isso num EndPoint que recebe um objeto e usa seus valores para inserir um registro na tabela. Vide o próximo código.

```
public void incluir(Pais novoPais) {

   jdbcTemplate.update("insert into tbl_pais (nome_pais, extensao_km2_pais) values (?,?)",
                        novoPais.getNomePais(), novoPais.getExtensaoKm2Pais());

}
```

O método **incluir()** usa o método **update()** do **jdbcTemplate** para enviar um DML (no caso, uma instrução "insert") para o SGBD.

O primeiro argumento no **update()** é a instrução SQL parametrizada e os demais, os valores que deverão substituir os parâmetros. O **novoPais.getNomePais()** vai substituir a primeira **?** da instrução e o **novoPais.getExtensaoKm2Pais()**, a segunda **?**.

Usando parâmetros nomeados

Assim como fizemos na consulta, é possível dar nomes aos parâmetros, precedendo eles com "**:**" e usando a classe **SqlParameterSource**. Vejamos como ficaria o método **incluir()** caso usássemos essa abordagem.

```
public void incluir(Pais novoPais) {

   SqlParameterSource parametros = new MapSqlParameterSource()
                                       .addValue("nome", novoPais.getNomePais())
                                       .addValue("extensao", novoPais.getExtensaoKm2Pais());

   jdbcTemplate.update(
        "insert into tbl_pais (nome_pais, extensao_km2_pais) values (:nome, :extensao)",
        parametros);
}
```

Note que a instrução SQL de "insert" possui 2 parâmetros: **nome** e **extensao**. Os objetos que deverão ser usados para dar-lhes valor são indicados na criação do objeto **parametros**, usando-se o valor de **novoPais.getNomePais()** para **nome** e o valor de **novoPais.getExtensaoKm2Pais()** para **extensao**.

Prática: Método para exclusão de registro

Com base nos exemplos aqui descritos, programe um novo método na **PaisCrud** que efetue o "delete" de um registro a partir da chave primária de **tbl_pais** informada via argumento no método.

Prática: Método para atualização de registro

Com base nos exemplos aqui descritos, programe um novo método na **PaisCrud** que efetue o "update" de um registro a partir da chave primária de **tbl_pais** informada via argumento no método *param* e de um objeto do tipo Pais informado no corpo da requisição. A assinatura do método ficaria assim:

```
public void atualizar(int id, Pais paisAlterado)
```

Use a versão que quiser de **Pais** (com nomes de atributo "iguais" aos dos campos da tabela ou não)

Bibliografia

GUTIERREZ, Felipe. Pro Spring Boot. New York (EUA): Apress, 2016.

KONDA, Madhusudhan. Just Spring Data Access: Covers JDBC, Hibernate, JPA and JDO. Sebastopol, CA, EUA: O'Reilly Media, Inc, 2012.

Oracle. Lesson: JDBC Basics (The Java™ Tutorials > JDBC(TM) Database Access) Disponível em: <https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>. Acesso em: 19/04/2019.