



Spring in the backend

Spring en el backend

Sumario

- Aprender a filosofia de acesso a dados do Spring
- Configurando uma fonte de dados
- Usando JDBC com o Spring

Introdução

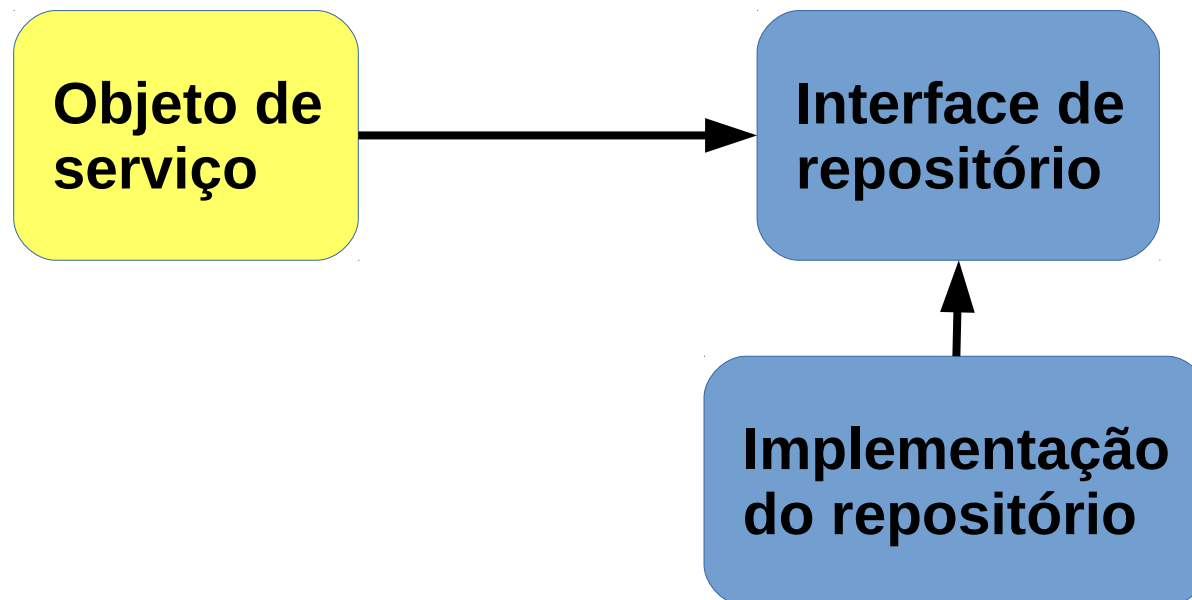
- Uma vez visto os aspectos fundamentais do núcleo do Spring, além disso do marco de trabalho para aplicações Web. Veremos um aspecto fundamental em todas as aplicações empresariais.
- **A persistência de dados**
- Para isso é necessário inicializar o marco de trabalho de acesso a dados, abrir conexões, dirigir vários tipos de exceções, e fechar conexões.
- Se um destes aspectos está mau, pode potencialmente corromper ou apagar dados valiosos de sua companhia.

Filosofia de acesso a dados no Spring

- Em todas as aplicações, é necessário ler e escrever dados em algum tipo de base de dados.
- Para evitar dispersar a lógica de persistência através de todos os componentes da aplicação, é boa prática fatorizar o acesso a base de dados em um ou mais componentes que estão enfocados nesta tarefa.
- Tais componentes são chamados Objetos de Acesso a Dados (DAOs) ou repositórios.

Filosofia de acesso a dados no Spring

- Para evitar o acoplamento da aplicação a qualquer tipo de estratégia de acesso a dados, os repositórios apropriadamente escritos devem expor suas funcionalidades através de interfaces.



Hierarquia de exceções de acesso a dados no Spring

- Spring provê um grande número de exceções de acesso a dados, cada uma descreve o problema pela que foi lançada.
- Spring tem uma exceção para virtualmente algo que possa ir mal lendo ou escrevendo a uma base de dados.
- Esta hierarquia não está associada com nenhum tipo particular de solução de persistência.
- O qual significa que pode contar com o Spring para lançar um conjunto consistente de exceções, independentemente de qual abastecedor de persistência utilize.

JDBC's exceptions	Spring's data-access exceptions
BatchUpdateException	BadSqlGrammarException
DataTruncation	CannotAcquireLockException
SQLException	CannotSerializeTransactionException
SQLWarning	CannotGetJdbcConnectionException
	CleanupFailureDataAccessException
	ConcurrencyFailureException
	DataAccessException
	DataAccessResourceFailureException
	DataIntegrityViolationException
	DataRetrievalFailureException
	DataSourceLookupApiUsageException
	DeadlockLoserDataAccessException
	DuplicateKeyException
	EmptyResultDataAccessException
	IncorrectResultSizeDataAccessException
	IncorrectUpdateSemanticsDataAccessException
	InvalidDataAccessApiUsageException
	InvalidDataAccessResourceUsageException
	InvalidResultSetAccessException
	JdbcUpdateAffectedIncorrectNumberOfRowsException
	LobRetrievalFailureException
	NonTransientDataAccessResourceException
	OptimisticLockingFailureException
	PermissionDeniedDataAccessException
	PessimisticLockingFailureException
	QueryTimeoutException
	RecoverableDataAccessException
	SQLWarningException
	SqlXmlFeatureNotImplementedException
	TransientDataAccessException
	TransientDataAccessResourceException
	TypeMismatchDataAccessException
	UncategorizedDataAccessException
	UncategorizedSQLException

Hierarquia de exceções de acesso a dados no Spring

- Todas estas exceções têm como raiz ao `DataAccessException`.
- O que a faz especial é que é uma exceção sem comprovação. Em outras palavras, não tem que capturar nenhum tipo de exceção lançada pelo Spring (embora o pode fazer se o deseja).

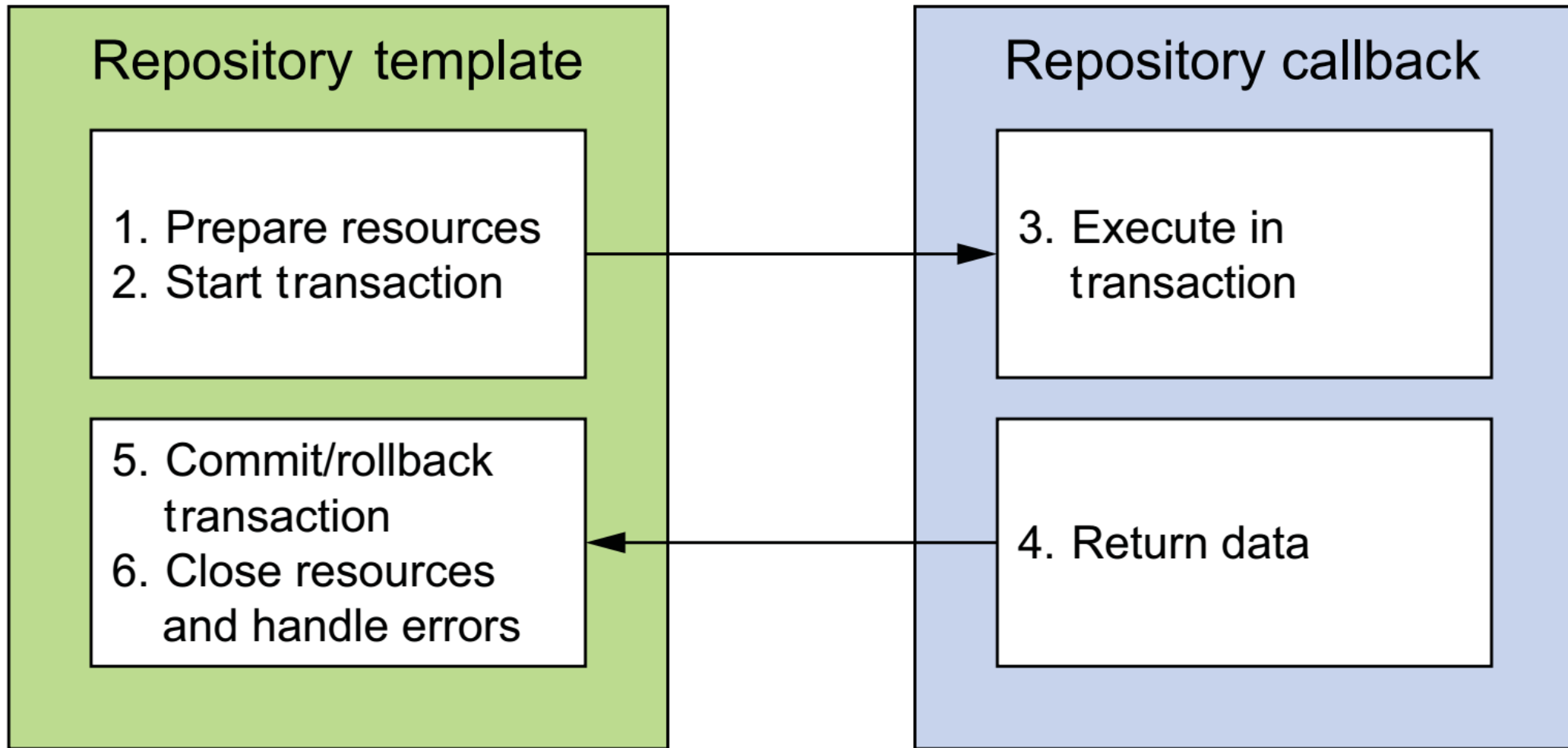
Moldes de acesso a dados

- Um molde de método define o esqueleto de um processo.
 - O processo em se mesmo for fixo; nunca troca
 - Alguns passados do processo também são fixos – acontecem da mesma maneira cada vez
 - Em certos pontos, o processo delega seu trabalho a uma sub-classe para encher em alguma implementação detalhe específicos.
- Este é o padrão que Spring aplica para o acesso a dados.

Filosofia de acesso a dados no Spring

- Spring separa as partes fixas e variáveis do processo de acesso a dados em duas classes distintas: **templates** e **callbacks**
 - Os templates administram a parte fixa do processo,
 - enquanto que o código específico de acesso a dados é dirigido nos callbacks.

Filosofia de acesso a dados no Spring



Filosofia de acesso a dados no Spring

- Como se pode apreciar, as classe molde do Spring dirigem a parte fixa do acesso a dados – controlando transações, dirigindo recursos, e dirigindo as exceções.
- Enquanto, as especificidades do acesso a dados pertencem a sua aplicação – criar instruções, atando parâmetros e organizar os conjuntos de resultados – são dirigidos na implementação do callback.
- Na prática, isto faz elegante a um marco de trabalho, porque tudo pelo que te tem que preocupar é por sua lógica de acesso a dados.

Filosofia de acesso a dados no Spring

- Spring vem com várias moldes para escolher, dependente da plataforma de persistência de sua eleição.
 - Se estas usando JDBC, então terá que usar `JdbcTemplate`.
 - Mas se quer usar um marco de trabalho de mapeo objeto – relacional então deve utilizar `HibernateTemplate` ou `JpaTemplate`

Template class (org.springframework.*)	Used to template . . .
<code>jca.cci.core.CciTemplate</code>	JCA CCI connections
<code>jdbc.core.JdbcTemplate</code>	JDBC connections
<code>jdbc.core.namedparam.NamedParameterJdbcTemplate</code>	JDBC connections with support for named parameters
<code>jdbc.core.simple.SimpleJdbcTemplate</code>	JDBC connections, simplified with Java 5 constructs (deprecated in Spring 3.1)
<code>orm.hibernate3.HibernateTemplate</code>	Hibernate 3.x+ sessions
<code>orm.ibatis.SqlMapClientTemplate</code>	iBATIS SqlMap clients
<code>orm.jdo.JdoTemplate</code>	Java Data Object implementations
<code>orm.jpa.JpaTemplate</code>	Java Persistence API entity managers

Configurando uma fonte de dados

- Independentemente de qual seja o acesso a dados suportado pelo Spring que use, é necessário configurar uma referência à fonte de dados.
- Spring oferece várias opções para configurar um bean de fonte de dados para sua aplicação Spring. Estes incluem:
 - Fonte de dados que são definidas por um controlador JDBC
 - Fonte de dados que são procurados por um JNDI.
 - Fonte de dados que agrupam conexões.

Usando fontes de dados JNDI

- As aplicações do Spring freqüentemente são desdobradas para executar-se em servidores de aplicação Java EE tais como WebSphere ou JBoss, ou inclusive em contêineres Web como Tomcat.
- Estes servidores permitem configurar as fontes de dados para ser obtidas por via JNDI.
- O benefício de configurar as fontes de dados desta maneira é que eles se podem administrar de forma completamente externa à aplicação, permitindo à aplicação perguntar por uma fonte de dados quando esta esta lista para acessar à base de dados.

Usando fontes de dados JNDI

- Com o Spring, pode configurar uma referência a uma fonte de dados que guarde no JNDI e unir esta nas classes que a necessitem como se esta for sozinho outro bean do Spring.
- O elemento `<jee:jndi-lookup>` do espaço de nome jee do Spring faz possível recuperar qualquer objeto incluindo fontes de dados, do JNDI e fazê-lo disponível como um bean do Spring.

Usando fontes de dados JNDI

- Por exemplo:
 - Se a fonte de dados de sua aplicação foi configurado no JNDI, poderia utilizar `<jee:jndi-lookup>` desta maneira no Spring:

```
<jee:jndi-lookup id="dataSource"  
  jndi-name="/jdbc/SpitterDS"  
  resource-ref="true" />
```

Usando fontes de dados JNDI

- O atributo JNDI-name é usado para especificar o nome do recurso no JNDI.
- Se somente a propriedade JNDI-name é estabelecida, então a fonte de dados procurará usando o nome dado tal e como é.
- Mas se esta aplicação executando-se em um servidor de aplicações Java, é desejável estabelecer a propriedade resource-ref a verdadeiro de modo que o valor dado no JNDI-name será anexado com `java:/comp/env/`.

Usando fontes de dados JNDI

- Alternativamente, se estas usando configuração Java pode usar JndiObjectFactoryBean para procurar um DataSource desde o JNDI.

```
@Bean
public JndiObjectFactoryBean dataSource() {
    JndiObjectFactoryBean jndiObjectFB =
        new JndiObjectFactoryBean();
    jndiObjectFB.setJndiName("jdbc/SpittrDS");
    jndiObjectFB.setResourceRef(true);
    jndiObjectFB
        .setProxyInterface(javax.sql.DataSource.class);
    return jndiObjectFB;
}
```

Usando um pool de fontes de dados

- Se não poder obter a fonte de dados desde o JNDI, a seguinte melhor costure é configurar um pool de fontes de dados diretamente no Spring.
- Embora Spring não provê nenhum, existem disponíveis vários, incluindo as seguintes opções de código aberto
 - Apache Commons DBCP
(<http://jakarta.apache.org/commons/dbcp>)
 - c3p0 (<http://sourceforge.net/projects/c3p0/>)
 - BoneCP (<http://jolbox.com/>)

Usando um pool de fontes de dados

- A maioria destas pool de conexões podem ser configuradas como uma fonte de dados no Spring de uma maneira que remonta a classe *DriverConnectionDataSource* ou *SimpleConnectionDataSource* do Spring.

Usando um pool de fontes de dados

- Por exemplo, assim é como podemos configurar um BasicDataSource do DBCP:

```
<bean id="dataSource"  
class="org.apache.commons.dbcp.BasicDataSource"  
p:driverClassName="org.h2.Driver"  
p:url="jdbc:h2:tcp://localhost/~/spitter"  
p:username="sa"  
p:password=""  
p:initialSize="5"  
p:maxActive="10" />
```

Usando um pool de fontes de dados

- Se preferir a configuração do Java, o DataSource para o pool deve ser declarado da seguinte maneira:

@Bean

```
public BasicDataSource dataSource() {  
    BasicDataSource ds = new BasicDataSource();  
    ds.setDriverClassName("org.h2.Driver");  
    ds.setUrl("jdbc:h2:tcp://localhost/~spitter");  
    ds.setUsername("sa");  
    ds.setPassword("");  
    ds.setInitialSize(5);  
    ds.setMaxActive(10);  
    return ds;  
}
```


Usando um pool de fontes de dados

- Se preferir a configuração do Java, o DataSource para o pool deve ser declarado da seguinte maneira:

Propriedades básicas do BasicDataSource



@Bean

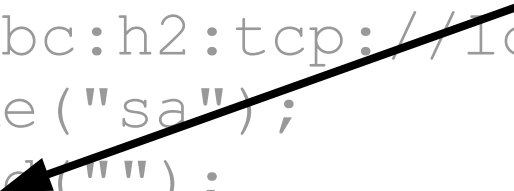
```
public BasicDataSource dataSource() {  
    BasicDataSource ds = new BasicDataSource();  
    ds.setDriverClassName("org.h2.Driver");  
    ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");  
    ds.setUsername("sa");  
    ds.setPassword("");  
    ds.setInitialSize(5);  
    ds.setMaxActive(10);  
    return ds;  
}
```

Usando um pool de fontes de dados

- Se preferir a configuração do Java, o DataSource para o pool deve ser declarado da seguinte maneira:

```
@Bean
public BasicDataSource dataSource() {
    BasicDataSource ds = new BasicDataSource();
    ds.setDriverClassName("org.h2.Driver");
    ds.setUrl("jdbc:h2:tcp://localhost:1521/");
    ds.setUsername("sa");
    ds.setPassword("");
    ds.setInitialSize(5);
    ds.setMaxActive(10);
    return ds;
}
```

Atributos para configurar o pool



Pool-configuration property	What it specifies
<code>initialSize</code>	The number of connections created when the pool is started.
<code>maxActive</code>	The maximum number of connections that can be allocated from the pool at the same time. If 0, there's no limit.
<code>maxIdle</code>	The maximum number of connections that can be idle in the pool without extras being released. If 0, there's no limit.
<code>maxOpenPreparedStatements</code>	The maximum number of prepared statements that can be allocated from the statement pool at the same time. If 0, there's no limit.
<code>maxWait</code>	How long the pool will wait for a connection to be returned to the pool (when there are no available connections) before an exception is thrown. If 1, wait indefinitely.
<code>minEvictableIdleTimeMillis</code>	How long a connection can remain idle in the pool before it's eligible for eviction.
<code>minIdle</code>	The minimum number of connections that can remain idle in the pool without new connections being created.
<code>poolPreparedStatements</code>	Whether or not to pool prepared statements (Boolean).

Usando fontes de dados apoiadas em driver JDBC

- A fonte de dados mais simples que se pode configurar no Spring é aquela definida através de um driver JDBC.
- Spring oferece três classes das quais se pode escolher
 - Todas disponíveis no pacote
`org.springframework.jdbc.datasource`

Usando fontes de dados apoiadas em driver JDBC

- **DriverManagerDataSource** – Devolve uma nova conexão cada vez que uma conexão é requerida.
- **SimpleDriverDataSource** – Trabalha muito parecido ao DriverManagerDataSource exceto ela trabalha com o driver JDBC diretamente para evitar problemas da carga de classes que podem surgir em certos ambientes, tais como em um contêiner do OSGi.
- **SingleConnectionDataSource** – Devolve a mesma conexão cada vez que uma conexão é requerida. Embora não é exatamente um pool de fontes de dados, pode pensar nela como um pool de exatamente uma conexão.

Usando fontes de dados apoiadas em driver JDBC

- Configurar qualquer dessas fontes de dados é similar a como se configuro o BasicDataSource do DBCP. Por exemplo, aqui se mostra como configurar um bean DriverManagerDataSource:

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource ds =
        new DriverManagerDataSource();
    ds.setDriverClassName("org.h2.Driver");
    ds.setUrl("jdbc:h2:tcp://localhost/~spitter");
    ds.setUsername("sa");
    ds.setPassword("");
    return ds;
}
```

Usando fontes de dados apoiadas em driver JDBC

- Usando XML, o bean DriverManagerDataSource pode ser configurado como segue:

```
<bean id="dataSource"  
class="org.springframework.  
    jdbc.datasource.DriverManagerDataSource"  
p:driverClassName="org.h2.Driver"  
p:url="jdbc:h2:tcp://localhost/~/spitter"  
p:username="sa"  
p:password="" />
```

Usando uma fonte de dados incorporado

- Uma base de dados embutida se executa como parte de sua aplicação em vez de como um servidor de base de dados a que sua aplicação se conecta.
- Embora não é muito útil em configurações de produção, uma base de dados embutida é uma opção perfeita para fins de desenvolvimento e prova.
- Por isso permite popular sua base de dados com dados de prova cada vez que reinicia sua aplicação ou executa suas provas.

Usando uma fonte de dados incorporado

- Para configurar o mesmo usando configuração Java:

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.H2)
        .addScript("classpath:schema.sql")
        .addScript("classpath:test-data.sql")
        .build();
}
```

Usando JDBC com o Spring

- JDBC não requer aprender outra linguagem de consulta pertencente a um marco de trabalho. Esta construído no topo do SQL, o qual é a linguagem de acesso a dados.
- Além disso, podem-se afinar o rendimento do acesso aos dados em virtualmente qualquer tecnologia de acesso a dados.
- JDBC permite tomar vantagem das características próprias da base de dados, onde outros Marcos de trabalho podem falhar ou simplesmente proibir estas práticas.

Usando JDBC com o Spring

- JDBC permite trabalhar com dados a mais sob nível que os Marcos de trabalho de persistência.
- Tem controle total de como a aplicação lê e manipula os dados.
 - Isto inclui permitir acessar e manipular colunas individuais em uma base de dados.
 - Esta aproximação de grão fino no acesso a dados é muito útil em aplicações como as aplicações de relatório, onde não tem sentido organizar os dados em objetos.

```

private static final String SQL_INSERT_SPITTER =
    "insert into spitter (username, password, fullname) values (?, ?, ?)";
private DataSource dataSource;
public void addSpitter(Spitter spitter) {
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(SQL_INSERT_SPITTER);
        stmt.setString(1, spitter.getUsername());
        stmt.setString(2, spitter.getPassword());
        stmt.setString(3, spitter.getFullName());
        stmt.execute();
    } catch (SQLException e) {
        // do something...not sure what, though
    } finally {
        try {
            if (stmt != null) {
                stmt.close();
            }
            if (conn != null) {
                conn.close();
            }
        } catch (SQLException e) {
            // I'm even less sure about what to do here
        }
    }
}

```

Get
connection

Create statement

Bind
parameters

Handle
exceptions
(somehow)

Clean up

Execute
statement

Insertar un objeto con JDBC

```

private static final String SQL_UPDATE_SPITTER =
    "update spitter set username = ?, password = ?, fullname = ?"
    + "where id = ?";
public void saveSpitter(Spitter spitter) {
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(SQL_UPDATE_SPITTER);
        stmt.setString(1, spitter.getUsername());
        stmt.setString(2, spitter.getPassword());
        stmt.setString(3, spitter.getFullName());
        stmt.setLong(4, spitter.getId());
        stmt.execute();
    } catch (SQLException e) {
        // Still not sure what I'm supposed to do here
    } finally {
        try {
            if (stmt != null) {
                stmt.close();
            }
            if (conn != null) {
                conn.close();
            }
        } catch (SQLException e) {
            // or here
        }
    }
}

```

Execute statement →

← **Get connection**

← **Create statement**

← **Bind parameters**

← **Handle exceptions (somehow)**

← **Clean up**

Actualizar un objeto con JDBC

```

private static final String SQL_SELECT_SPITTER =
    "select id, username, fullname from spitter where id = ?";
public Spitter findOne(long id) {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(SQL_SELECT_SPITTER);
        stmt.setLong(1, id)
        rs = stmt.executeQuery();
        Spitter spitter = null;
        if (rs.next()) {
            spitter = new Spitter();
            spitter.setId(rs.getLong("id"));
            spitter.setUsername(rs.getString("username"));
            spitter.setPassword(rs.getString("password"));
            spitter.setFullName(rs.getString("fullname"));
        }
        return spitter;
    } catch (SQLException e) {
    } finally {
        if(rs != null) {
            try {
                rs.close();
            } catch (SQLException e) {}
        }

        if(stmt != null) {
            try {
                stmt.close();
            } catch (SQLException e) {}
        }

        if(conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {}
        }
    }
    return null;
}

```

Execute query →

← **Get connection**

← **Create statement**

← **Bind parameters**

← **Process results**

← **Handle exceptions (somehow)**

Clean up

Recuperar un objeto con JDBC

Usando JDBC com o Spring

- Como se pode apreciar só aproximadamente o 20% do código é para realizar as operações, o resto é código repetitivo para assegurar a conexão, etc.
- Apesar de tudo, este código repetitivo é importante, e é uma das principais fontes de equívoco.
- Esta é uma das razões principais para utilizar um marco de trabalho, o qual te assegura que o código se escreva uma só vez e se faça bem.

Trabalhando com as moldes JDBC

- O marco de trabalho JDBC do Spring limpará nosso código realizando o código repetitivo da administração de recursos e o manejo de exceções.
- Isto nos deixa libra para escrever somente o código necessário para mover nossos dados para e da base de dados.

Trabalhando com as moldes JDBC

- Spring vem com três moldes para o JDBC:
 - [JdbcTemplate](#) - É a molde mais básica, provê acesso simples à base de dados através do JDBC e consultas de parâmetros indexados.
 - [NamedParameterJdbcTemplate](#) – Esta molde JDBC permite realizar consultas onde os valores estão ligados a parâmetros nomeados no SQL, em vez de parâmetro indexados.
 - [SimpleJdbcTemplate](#) – Esta versão de molde JDBC toma vantagens das características do Java 5 como o auto-empacotado, genericidade e variáveis de lista de parâmetros para simplificar como JDBC é usado.

Inserindo dados usando JdbcTemplate

- Tudo o que necessita JdbcTemplate para trabalhar é um DataSource.
- Para configurar JdbcTemplate no Spring é:

```
@Bean
```

```
public JdbcTemplate jdbcTemplate(  
    DataSource dataSource) {  
    return new JdbcTemplate(dataSource);  
}
```

Inserindo dados usando JdbcTemplate

- Com o repositório JdbcTemplate a nossa disposição pode simplificar o código requerido.

```
public void addSpitter(Spitter spitter) {  
    jdbcOperations.update(INSERT_SPITTER,  
        spitter.getUsername(),  
        spitter.getPassword(),  
        spitter.getFullName(),  
        spitter.getEmail(),  
        spitter.isUpdateByEmail());  
}
```

Carregando dados com o JdbcTemplate

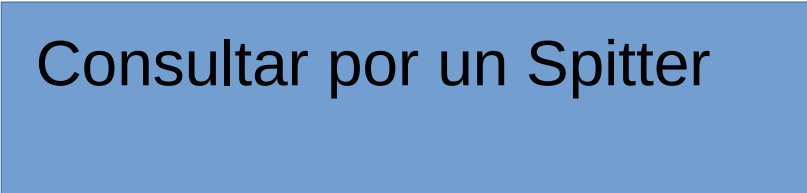
```
public Spitter findOne(long id) {
    return jdbcOperations.queryForObject(
        SELECT_SPITTER_BY_ID, new SpitterRowMapper(),
        id
    );
}

...

private static final class SpitterRowMapper
    implements RowMapper<Spitter> {
    public Spitter mapRow(ResultSet rs, int rowNum)
        throws SQLException {
        return new Spitter(
            rs.getLong("id"),
            rs.getString("username"),
            rs.getString("password"),
            rs.getString("fullName"),
            rs.getString("email"),
            rs.getBoolean("updateByEmail"));
    }
}
```

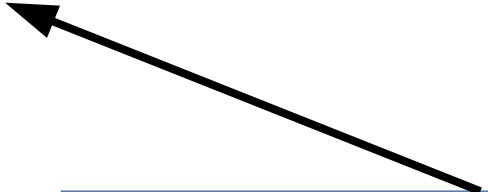
Carregando dados com o JdbcTemplate

```
public Spitter findOne(long id) {  
    return jdbcTemplate.queryForObject(  
        SELECT_SPITTER_BY_ID, new SpitterRowMapper(),  
        id  
    );  
}  
...  
private static final class SpitterRowMapper  
    implements RowMapper<Spitter> {  
    public Spitter mapRow(ResultSet rs, int rowNum)  
        throws SQLException {  
        return new Spitter(  
            rs.getLong("id"),  
            rs.getString("username"),  
            rs.getString("password"),  
            rs.getString("fullName"),  
            rs.getString("email"),  
            rs.getBoolean("updateByEmail"));  
    }  
}
```



Carregando dados com o JdbcTemplate

```
public Spitter findOne(long id) {  
    return jdbcOperations.queryForObject(  
        SELECT_SPITTER_BY_ID, new SpitterRowMapper(),  
        id  
    );  
}  
  
...  
private static final class SpitterRowMapper  
    implements RowMapper<Spitter> {  
    public Spitter mapRow(ResultSet rs, int rowNum)  
        throws SQLException {  
        return new Spitter(  
            rs.getLong("id"),  
            rs.getString("username"),  
            rs.getString("password"),  
            rs.getString("fullName"),  
            rs.getString("email"),  
            rs.getBoolean("updateByEmail"));  
    }  
}
```



Mapear el resultado a un objeto

Carregando dados com o JdbcTemplate

```
public Spitter findOne(long id) {  
    return jdbcOperations.queryForObject(  
        SELECT_SPITTER_BY_ID, new SpitterRowMapper(),  
        id  
    );  
}
```

```
...  
private static final class SpitterRowMapper  
    implements RowMapper<Spitter> {  
    public Spitter mapRow(ResultSet rs, int rowNum)  
        throws SQLException {  
  
        return new Spitter(  
            rs.getLong("id"),  
            rs.getString("username"),  
            rs.getString("password"),  
            rs.getString("fullName"),  
            rs.getString("email"),  
            rs.getBoolean("updateByEmail"));  
        }  
    }  
}
```

Atar los parámetros del resultado de la consulta.



Carregando dados com o JdbcTemplate

- Este método `findOne()` usa o método `queryForObject()` do `JdbcTemplate` para consultar por um `Spitter` da base de dados. Este método toma três parâmetros:
 - Um `String` contendo o SQL para ser usado para selecionar dados da base de dados.
 - Um objeto `RowMapper` que extrai os valores de um `ResultSet` e constrói um objeto do domínio (neste caso um objeto `Spitter`).
 - Uma variável argumento que lista os valores a ser enlaçados aos parâmetros de indique da consulta.

Carregando dados com o JdbcTemplate

- A magia acontece no objeto SpitterRowMapper, o qual implementa a interface RowMapper.
- Para cada fila que resulta da consulta, o JdbcTemplate chama o método mapRow() da classe RowMapper, passando como parâmetro um ResultSet e um inteiro que representa o número de fila.
- No método mapRow() do SpitterRowMapper está o código que cria um objeto Spitter e o enche com os valores do ResultSet.

Usando Lambdas do Java 8 com o JdbcTemplate

- Como a interface `RowMapper` somente declara o método `addRow`, esta pode ser uma interface funcional.
- Isto significa que se estamos desenvolvendo nossa aplicação usando Java 8, podemos expressar a implementação do `RowMapper` como uma lambda em vez da implementação concreta de uma classe.

Usando Lambdas do Java 8 com o JdbcTemplate

```
public Spitter findOne(long id) {  
    return jdbcOperations.queryForObject(  
        SELECT_SPITTER_BY_ID,  
        (rs, rowNum) -> {  
            return new Spitter(  
                rs.getLong("id"),  
                rs.getString("username"),  
                rs.getString("password"),  
                rs.getString("fullName"),  
                rs.getString("email"),  
                rs.getBoolean("updateByEmail"));  
        },  
        id);  
}
```

Carregando dados com o JdbcTemplate

- Alternativamente pode utilizar a referência a métodos do Java 8 para definir o mapeo em um método separado:

```
public Spitter findOne(long id) {  
    return jdbcOperations.queryForObject(  
        SELECT_SPITTER_BY_ID, this::mapSpitter, id);  
}  
private Spitter mapSpitter(ResultSet rs, int row)  
    throws SQLException {  
    return new Spitter(  
        rs.getLong("id"),  
        rs.getString("username"),  
        rs.getString("password"),  
        rs.getString("fullName"),  
        rs.getString("email"),  
        rs.getBoolean("updateByEmail"));  
}
```

Usando parâmetros nomeados

- Nos exemplos anteriores se utilizam parâmetros indexados.
- Isto significa que tem que ter em conta a ordem dos parâmetros na consulta e listar os valores na ordem correta quando o passar a eles aos métodos `update()`.
- Se trocar a ordem dos SQL de tal maneira que trocar a ordem dos parâmetros, então também se deve trocar a ordem dos valores que recebe a consulta.

Usando parâmetros nomeados

- Opcionalmente, pode-se usar parâmetros nomeados.
- Estes parâmetros nomeados lhe permitem lhe dar a cada parâmetro no SQL um nome explícito e te referir a ele pelo nome quando passar os valores à declaração.
- Por exemplo suponhamos que a consulta SQL_INSERT_SPITTER está definida como segue:

```
private static final String SQL_INSERT_SPITTER =  
"insert into spitter (username, password, fullname) "  
+ " values (:username, :password, :fullname)";
```

Usando parâmetros nomeados

- Com as consultas com nomes de parâmetros, a ordem em que aparecem os parâmetros não é importante.
- Pode passar cada valor por nome. E se a consulta troca e a ordem dos parâmetros não é a mesma, já não tem que preocupar por isso.
- A classe `NamedParameterJdbcTemplate` é uma molde especial do `JDBC` que suporta o trabalho com parâmetros nomeados.

Usando parâmetros nomeados

- A classe NamedParameterJdbcTemplate pode ser declarada no Spring da mesma maneira que a classe JdbcTemplate:

```
@Bean
public NamedParameterJdbcTemplate jdbcTemplate (
    DataSource dataSource) {
    return
        new NamedParameterJdbcTemplate (dataSource);
}
```

- Agora o método addSpitter() deve ser da seguinte maneira.

Usando parâmetros nomeados

```
private static final String INSERT_SPITTER =  
    "insert into Spitter " +  
    " (username, password, fullname, email,  
updateByEmail) " +  
    "values " +  
    " (:username, :password, :fullname, :email,  
:updateByEmail)";  
  
public void addSpitter(Spitter spitter) {  
    Map<String, Object> paramMap =  
        new HashMap<String, Object>();  
    paramMap.put("username", spitter.getUsername());  
    paramMap.put("password", spitter.getPassword());  
    paramMap.put("fullname", spitter.getFullName());  
    paramMap.put("email", spitter.getEmail());  
    paramMap  
        .put("updateByEmail", spitter.isUpdateByEmail());  
    jdbcOperations.update(INSERT_SPITTER, paramMap);  
}
```

Usando parâmetros nomeados

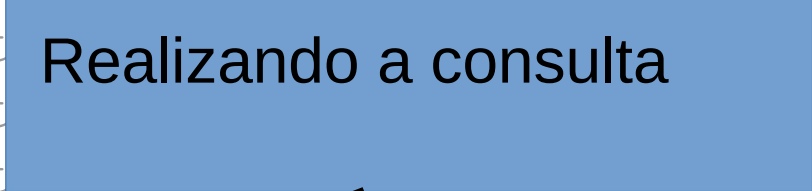
```
private static final String INSERT_SPITTER =  
    "insert into Spitter " +  
    " (username, password, fullname, " +  
    "updateByEmail) " +  
    "values " +  
    " (:username, :password, :fullname, :email, " +  
    ":updateByEmail)";
```

Parâmetros da consulta.

```
public void addSpitter(Spitter spitter) {  
    Map<String, Object> paramMap =  
        new HashMap<String, Object>();  
    paramMap.put("username", spitter.getUsername());  
    paramMap.put("password", spitter.getPassword());  
    paramMap.put("fullname", spitter.getFullName());  
    paramMap.put("email", spitter.getEmail());  
    paramMap  
        .put("updateByEmail", spitter.isUpdateByEmail());  
    jdbcOperations.update(INSERT_SPITTER, paramMap);  
}
```

Usando parâmetros nomeados

```
private static final String INSERT_SPITTER =  
    "insert into Spitter " +  
    " (username, password, fullname, email,  
updateByEmail) " +  
    "values " +  
    " (:username, :password, :fullname, :email,  
:updateByEmail)";  
  
public void addSpitter(Spitter spitter) {  
    Map<String, Object> paramMap =  
        new HashMap<String, Object>();  
    paramMap.put("username", spitter.getUsername());  
    paramMap.put("password", spitter.getPassword());  
    paramMap.put("fullname", spitter.getFullname());  
    paramMap.put("email", spitter.getEmail());  
    paramMap  
        .put("updateByEmail", spitter.isUpdateByEmail());  
    jdbcTemplate.update(INSERT_SPITTER, paramMap);  
}
```



Resumem

- Os dados é o sangue de uma aplicação.
- É importante que desenvolva a porção de acesso a dados de sua aplicação de uma maneira robusta, simples e clara.
- JDBC é a maneira mais básica de trabalhar com dados lhes relacione no Java. Mas como está definida na especificação pode ser muito trabalhosa de usar.
- Spring tira a maior parte da dor de trabalhar com o JDBC, eliminando o código repetitivo e simplificando o manejo de exceções do JDBC, te deixando pouco mais que tratar que com a escritura do SQL que deve ser executado.
- Vimos como é o suporte do Spring para a persistência. Especificamente as moldes do Spring apoiadas no JDBC.

Leitura recomendada

- Walls, C. (2014). **Spring in Action** (Fourth Edi). Manning Publications. Retrieved from <https://www.manning.com/books/spring-in-action-fourth-edition>
 - Chapter 11 - Persisting data with object-relational mapping
 - Chapter 12 - Working with NoSQL databases
 - Chapter 13 - Caching data
 - Chapter 14 - Securing methods



Spring in the backend

Spring en el backend