01

Vários entradas para o relatório

Transcrição

Vimos nas vídeo aulas anteriores como usar o iReport, interface gráfica para criar a definição do relatório que é salva dentro de um arquivo `jrxml`. Esse arquivo XML foi compilado através do iReport ou programaticamente para um arquivo `jasper`. Depois preenchemos este arquivo com os parâmetros da aplicação e a conexão com o banco, resultando no objeto `JasperPrint`, que exportamos para PDF pelo `JRPdfExporter`.

Ainda há um pequeno problema nessa abordagem. Estamos amarrando o JasperPrint ao banco de dados pois passamos a conexão ao método `fillReport(..)`. Como existem várias saídas para o relatório, também podem existir várias entradas. Alguns exemplos disso são fontes de dados, como XML, JSON, CSV e muitas outras. Não queremos nos acoplar ao banco de dados. Queremos maior flexibilidade. Para isso existe a interface `JRDataSource`, que representa uma fonte de dados que abstrai da entrada concreta utilizada.

Introdução ao projeto

Para esta aula, utilizaremos o relatório que foi criado na primeira aula, aquele que lista movimentações. Nesse relatório configuramos a conexão com o banco MySQL executando uma simples query que selecionava todos os dados da tabela `movimentacoes`.

No Eclipse, já temos um projeto Java preparado que se chama `movimentacoes`. Dentro da pasta `src`, temos uma classe para gerar relatório. Na raiz do projeto encontra-se a definição do relatório, o arquivo `financas.jrxml`. Nele verificaremos se a linguagem padrão é Java.

Primeiro, analisamos a classe `TesteGeraRelatorio`, compilamos o arquivo `.jrxml`, definimos a entrada da aplicação, preenchemos o relatório e exportamos para PDF.

```
public class TesteGeraRelatorio {
    public static void main(String[] args) throws SQLException, JRException, FileNotFoundExcept

        JasperCompileManager.compileReportToFile("financas.jrxml");

        Map<String, Object> parametros = new HashMap<String, Object>();
        Connection connexao = new ConnectionFactory().getConnection();

        JasperPrint print = JasperFillManager.fillReport("financas.jasper", parametros, conexao

        JRExporter exporter = new JRPdfExporter();
        exporter.setParameter(JRExporterParameter.JASPER_PRINT, print);
        exporter.setParameter(JRExporterParameter.OUTPUT_STREAM, new FileOutputStream("financas
        exporter.exportReport();

        connexao.close();
    }
}
```

Ao chamar o método *main* e atualizar o projeto, o arquivo `financas.jasper` e a saída `financas.pdf` são criados. Abriremos mais uma vez o PDF para ver o resultado. Todas as movimentações foram listadas. Para evitar a

recompilação do relatório, comentaremos no código a linha responsável.

Desacoplando os dados e a interface JRDataSource

Ao investigar o método `fillReport(..)`, podemos ver que foi passado a conexão. Justamente nessa linha, amarramos o relatório ao banco de dados. Ao visualizar as versões do método `fillReport(..)` no Eclipse, percebemos que ele é sobrecarregado. Há uma versão que recebe um objeto do tipo `JRDataSource` no lugar da conexão. Essa `JRDataSource` é a abstração da fonte de dados.

Para conhecer melhor esta classe, vamos procurar pelo seu Javadoc na Internet:

http://jasperreports.sourceforge.net/api/net/sf/jasperreports/engine/JRDataSource.html (http://jasperreports.sourceforge.net/api/net/sf/jasperreports/engine/JRDataSource.html)

Ao seguir o link da API, estamos vendo todas as classes e *packages* disponíveis. Procuraremos pelo `JRDataSource` para chegar em seu javadoc. Na documentação, podemos ver que `JRDataSource` é uma interface que declara apenas dois métodos: `getFieldValue(..)` e `next()`.

O JasperReport já disponibiliza várias implementações dessa interface, por exemplo, a `JRXmlDataSource` para trabalhar com arquivos XML como fonte de dados, `JRCsvDataSource` para trabalhar com CSV ou `JRBeanCollectionDataSource` para usar a lista de *java beans* como datasource.

Usando uma lista como datasource

Ao investigar o Javadoc da classe `JRBeanCollectionDataSource`, podemos ver que ela recebe uma coleção de beans. Vamos voltar ao Eclipse. A primeira coisa que faremos é tirar a conexão da método `fillReport(..)`. Vamos passar uma *dataSource*. Como já falamos, a *dataSource* será do tipo `JRDataSource` e a implementação é do tipo `JRBeanCollectionDataSource`, que recebe no construtor uma lista de movimentações.

Vamos declarar a lista acima e carregar as movimentações usando o DAO que já está preparado no projeto. A classe `MovimentacaoDAO` possui o método `todos()` que devolve todas as movimentações do banco de dados.

```
Connection connexao = new ConnectionFactory().getConnection();
List<Movimentacao> movimentacoes = new MovimentacaoDAO(connexao).todos();
JRDataSource dataSource = new JRBeanCollectionDataSource(movimentacoes);

JasperPrint print = JasperFillManager.fillReport("financas.jasper", parametros, dataSource);
```

Antes de testar, vamos verificar o método `todos()` do DAO. Podemos ver o SQL e o uso do `PreparedStatement` para executar o SQL. Depois, usaremos o resultado da query (`ResultSet`) para criar em cada iteração uma nova movimentação. As movimentações ficam guardadas em uma lista que representa o retorno do método `todos()`.

Geração do relatório e as primeiras incompatibilidades

Vamos fechar a classe `MovimentacaoDAO` e executar a classe `TestGeraRelatorio` para testar a `DataSource`. Após a execução, é gerada uma exceção acusando um problema de cast entre `GergorianCalendar` e `java.sql.Date`.

O problema está no arquivo `jrxml`. O field data foi declarado como `java.sql.Date`. Essa definição ocorreu quando definimos o relatório com iReport. Usamos uma query SQL e o iReport associou automaticamente tipos com campos da query. O mais correto é definir os tipos, independentemente da fonte de dados. Ou seja, para a data, usando a classe `java.util.Date`.

Vamos rodar novamente a classe `Teste`, só descomentando a linha que compila o `jrxml`, pois alteramos o relatório. Novamente foi gerada uma exceção de cast, pois a incompatibilidade entre os tipos `GregorianCalendar` e `java.util.Date` continua.

Adaptar o modelo para atender o relatório

Uma opção possível é ajustar o arquivo `jrxml` para trabalhar com o tipo `java.util.Calendar`, alterando a classe e as expressões utilizadas no relatório. Outra possibilidade é ajustar o modelo. A classe `Movimentacao` para não fornecer a data do tipo `java.util.Calendar` e sim `java.util.Date`. Essa alteração se reflete no método `getData()` da classe, que deve devolver um `Date` chamando o método `getTime()` da classe `Calendar`. Esta forma faz com que o nosso modelo forneça os tipos corretos para o relatório funcionar.

Após outro teste, recebemos mais uma exceção, agora não mais relacionada com a data, mas com a enumeração `TipoMovimentacao` que é incompatível com o tipo `java.lang.String`.

Ao verificarmos o relatório podemos ver que o campo `tipoMovimentacao` realmente foi mapeado para a classe `String`. No outro lado, na classe `Movimentacao`, há o método `getTipoMovimentacao()` que devolve uma `Enum`. Alteramos o método para devolver uma `String` usando um método `name()` da enumeração.

```
public String getTipoMovimentacao() {
    return tipoMovimentacao.name();
}
```

Após executar, o PDF foi finalmente gerado.

Desacoplar o modelo do relatório

Conseguimos usar um outro `DataSource` para abstrair a fonte de dados utilizando uma lista de beans para alimentar o relatório. Percebemos que alguns ajustes foram necessários. O problema era que os tipos do relatório não combinavam com os tipos do nosso modelo. Duas soluções eram possíveis: alterar a definição do relatório ou adaptar o modelo. Em nosso caso, a classe `Movimentacao`.

Isso gera um acoplamento entre o relatório e o modelo. Queremos que os dois lados evoluam separadamente. Ou seja, mudanças no relatório devem não influenciar o modelo e vice-versa. Os ajustes realizados no modelo foram apenas um exemplo e não eram necessários.

A ideia é continuar com a `JRBeanCollectionDataSource` que recebe uma lista de beans, mas esses beans não serão mais objetos da classe `Movimentacao`. Vamos desfazer as alterações na classe, devolvendo uma enumeração no método `getTipoMovimentacao()` e no método `getData()`. Daremos uma data do tipo `java.util.Calendar`.

Para fornecer os dados corretos, criaremos uma nova lista com elementos do tipo `MovimentacaoRelatorio`. Essa classe auxiliar é específica para o relatório, com a responsabilidade de fornecer os dados com os tipos corretos.

Vamos criar a classe pelo Eclipse definindo o atributo `Movimentacao`. Vamos gerar também um construtor que recebe a movimentação. Segue o primeiro esboço da classe:

```
public class MovimentacaoRelatorio {

    private Movimentacao movimentacao;

    public MovimentacaoRelatorio(Movimentacao movimentacao) {
```

```
            this.movimentacao = movimentacao;
        }

    }
```

Dentro da classe `MovimentacaoRelatorio`, vamos delegar os `getters`, novamente utilizando o Eclipse para gerar este código. Os método gerados são os que o relatório utilizará. Ou seja, `getData()`, `getDescricao()`, `getId()`, `getTipoMovimentacao()` e `getValor()`. Ao confirmar, o Eclipse criará todos os `getters` selecionados na classe.

As mudanças necessárias relacionadas com os tipos serão feitas nesta classe auxiliar e não no modelo da aplicação. Sendo assim, alteraremos primeiro o retorno do método `getData()`, novamente utilizando `java.util.Date` com o método `getTime()`. Vamos mudar o método `getTipoMovimentacao()`, que retorna uma `String` utilizando o método `name()` da enumeração.

Ao voltar à classe `TesteGeraRelatorio`, falta inicializar a `listaRelatorio` e depois copiar os elementos da lista de movimentações para a lista nova. Utilizaremos um laço em cada iteração, criando um objeto `MovimentacaoRelatorio` que guardamos na `listaRelatorio`. Passaremos esta lista para o `DataSource`.

```
List<Movimentacao> movimentacoes = new MovimentacaoDAO(connexao).todos();
List<MovimentacaoRelatorio> listaRelatorio= new ArrayList<MovimentacaoRelatorio>();

for (Movimentacao movimentacao : movimentacoes) {
    listaRelatorio.add(new MovimentacaoRelatorio(movimentacao));
}

JRDataSource dataSource = new JRBeanCollectionDataSource(listaRelatorio);
```

Antes de testarmos novamente, comentaremos a linha que compila o arquivo `jrxml`. Após a execução e atualizando o projeto, foi gerado o PDF. Vamos abrí-lo. Todos os dados aparecem, como esperado.

Resumo da aula

Resumindo, vimos como podemos abstrair a fonte de dados usando a interface `JRDataSource`. A implementação concreta que usamos foi a `JRBeanCollectionDataSource`. Com ela podemos utilizar qualquer coleção de Java Beans como fonte de dados. O objetivo era usar a lista de movimentações para o nosso relatório. Carregamos a lista pelo DAO e passamos ela para a `DataSource`. Percebemos que era preciso adaptar o relatório ou a classe `Movimentação`. Isso gerou um acoplamento que resolvemos através de uma classe auxiliar `MovimentacaoRelatorio`, e que também serviu como um adaptador entre o nosso modelo e o relatório.