02

Recebendo dados da requisição

Vamos começar nosso sistema de contas a pagar e receber. O primeiro passo será representar uma `Conta`. Uma conta contém uma descrição, um valor, um tipo (entrada ou saída), uma data de pagamento, e um booleano informando se ela foi paga ou não. Em Java:

```
public class Conta {

    private Long id;
    private String descricao;
    private boolean paga;
    private double valor;
    private Calendar dataPagamento;
    private TipoDaConta tipo;

    // getters e setters
}

public enum TipoDaConta {
    ENTRADA,
    SAIDA
}
```

Com a `Conta` representada, vamos ao próximo passo. Nossa primeira tela será a tela de inserção de conta. Para isso, precisamos de um formulário HTML, com os campos para o usuário digitar os dados dessa nova conta. Vamos criar um JSP chamado `formulario.jsp`, com os campos descrição, valor e uma lista de opções para ele escolher entre ENTRADA ou SAÍDA:

```
<html>
<body>
    <h3>Adicionar Contas</h3>
    <form action="adicionaConta" method="post">
        Descrição: <br/>
        <textarea name="descricao" rows="5" cols="100"></textarea>
        <br/>
        Valor: <br/>
        <input type="text" name="valor" /><br/>
        Tipo: <br/>
        <select name="tipo">
            <option value="ENTRADA">Entrada</option>
            <option value="SAIDA">Saída</option>
        </select>
        <br/><br/>
        <input type="submit" value="Adicionar"/>
    </form>
</body>
</html>
```

Observando o formulário acima, é possível perceber que, ao ser postado, as informações serão enviadas para a ação "adicionaConta". Como aprendemos na aula passada, ela cairá em um Controller, que saberá responder por ela. Como

vimos, para que o Spring MVC reconheça uma classe como Controller, ela precisa estar anotada com `@Controller`, e ter um método com a anotação `@RequestMapping`, com o conteúdo "adicionaConta". Esse método, por sua vez, salvará a Conta no banco:

```
@Controller
public class ContaController {
  @RequestMapping("/adicionaConta")
  public String adiciona() {
    ContaDAO dao = new ContaDAO();
    dao.adiciona(conta);
    return "conta-adicionada";
  }
}
```

Um ponto importante aqui é discutir de onde vem a variável `conta`. Se estivéssemos usando Servlets puras, precisaríamos preencher esse objeto na mão (por meio de vários `request.getParameter(..)`). O Spring MVC facilita nossa vida e popula automaticamente esses objetos. Basta recebermos ele na assinatura do método:

```
public String adiciona(Conta conta) {
  // ...
}
```

Como ele sabe como preencher? Basta olhar o nome dos atributos no HTML. Veja: `conta.descricao`, `conta.tipo` e assim por diante. Ele preencherá o atributo `descricao` dentro da variável com nome `conta` na Action. Como a classe `Conta` é um Java Bean, e possui getters e setters, tudo funcionará.

Precisamos exibir a mensagem de confirmação, afinal o usuário precisa saber que a conta foi inserida. Se olharmos a nossa Action, estamos redirecionando para "conta-adicionada". Então, vamos criar o JSP com esse nome:

```
<html>
  <body>
    Nova conta adicionada com sucesso!
  </body>
</html>
```

Por fim, precisamos encontrar uma maneira de exibir o formulário que criamos (e está em `formulario.jsp`). Para isso, basta criarmos uma Action, que simplesmente aponte para esse JSP:

```
    @RequestMapping(value="/form")
    public String form() {
        return "formulario";
    }
```

Pronto. Ao acessarmos a URL `http://localhost:8080/contas/form (http://localhost:8080/contas/form)`, ele nos apontará para o formulário HTML. Ao preencher o formulário e clicar em Salvar, uma requisição será feita para "/adiciona". Nesse momento, o Spring MVC preencherá o objeto conta, e salvará no banco. Por fim, redirecionará para a view, mostrando a mensagem de sucesso.

Um detalhe final é que o Spring MVC ele sabe converter bastante coisa, mas não sabe converter enums. Para isso, precisamos ensiná-lo. Na própria documentação do Spring MVC, ele sugere que você tenha a seguinte classe no seu projeto:

```
@SuppressWarnings({"rawtypes", "unchecked"})
final class StringToEnumConverterFactory implements ConverterFactory<String, Enum> {

    public <T extends Enum> Converter<String, T> getConverter(Class<T> targetType) {
        return new StringToEnumConverter(targetType);
    }

    private final class StringToEnumConverter<T extends Enum> implements Converter<String, T> {

        private Class<T> enumType;

        public StringToEnumConverter(Class<T> enumType) {
            this.enumType = enumType;
        }

        public T convert(String source) {
            return (T) Enum.valueOf(this.enumType, source.trim());
        }
    }
}
```

Não há necessidade de entrar nos detalhes dela nesse momento. Tudo que essa classe faz é que, quando o Spring pegue um enum para ser convertido, ele consiga ver a String postada, e a transforme no enum esperado. Para que ele saiba usar essa classe, precisamos configurá-la no spring-context.xml:

```
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <list>
            <bean class="br.com.caelum.contas.StringToEnumConverterFactory"/>
        </list>
    </property>
</bean>
```

Pronto. É bem comum termos essa classe em nossos projetos, já que enums são bastante úteis. Tudo agora funciona.