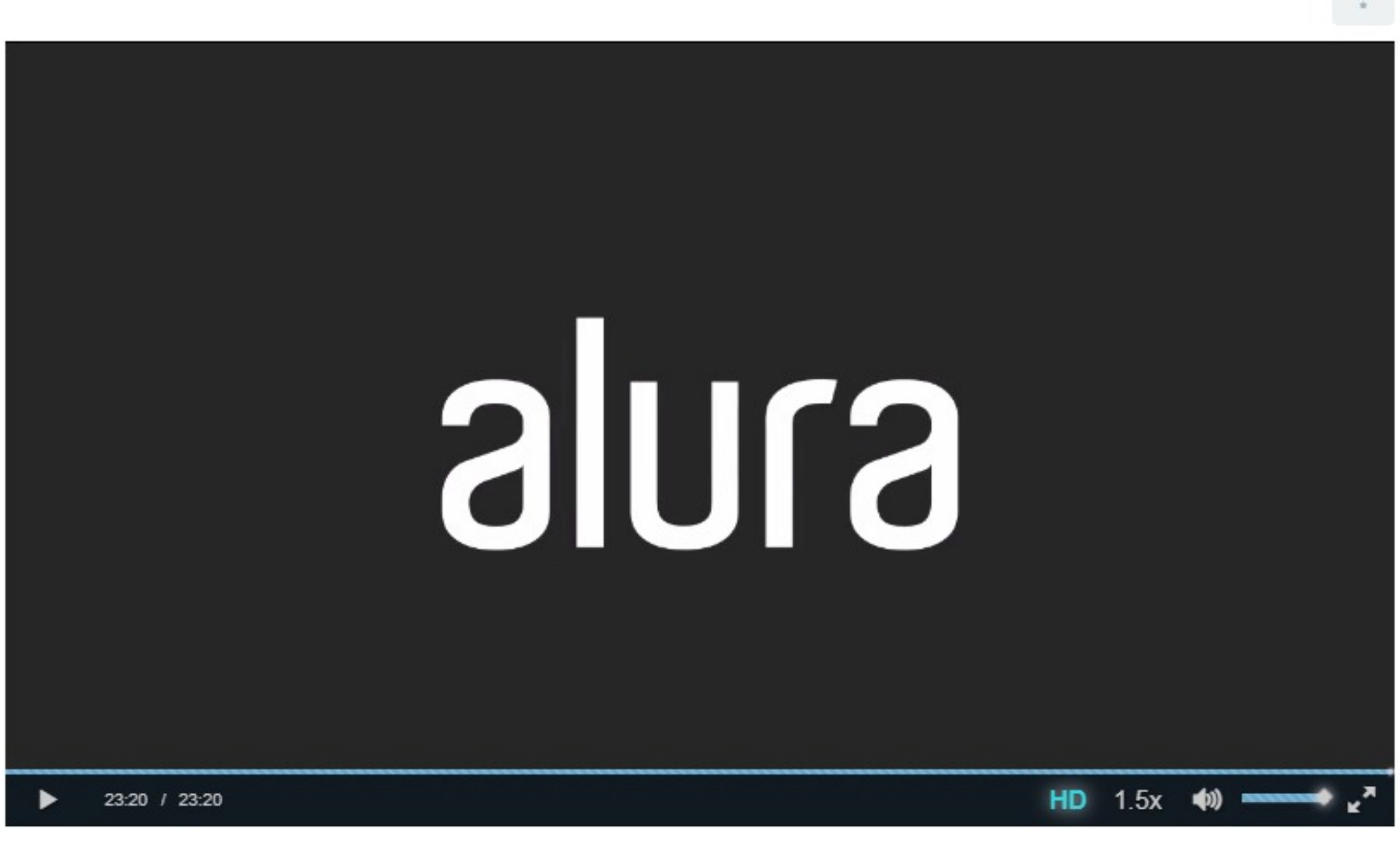
01 Validação e conversão de dados

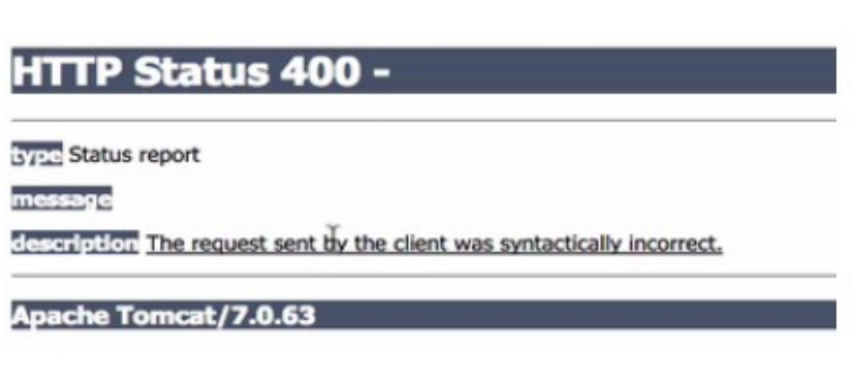
PRÓXIMA ATIVIDADE



Transcrição

Até aqui fizemos bastante coisa na nossa aplicação. Cadastramos livros, exibimos uma mensagem de sucesso e fizemos nossos livros aparecerem em uma listagem simples, mas que demonstra que tudo está funcionando perfeitamente.

Agora chegamos em um ponto em que precisamos validar os dados enviados pelo usuário. Esta validação é muito importante. Como seria se alguém tentasse cadastrar livros **sem preencher** os dados do formulário? Tente fazer isso pra ver o que acontece.



HTTP Status 400 -

type Status report

message

description The request sent by the client was syntactically incorrect.

Apache Tomcat/7.0.63

```
Erro 400 - The request sent by the client was syntactically incorrect.
```

A requisição foi enviada com uma sintaxe errada. Como assim? Tem algum problema acontecendo na hora de enviar os dados. Isso acontece porque não estamos enviando dados. Para corrigir isso, devemos validá-los e para isso existem algumas formas.

A primeira forma seria validar com ***JavasSript***, mas como esta aconteceria no navegador seria menos confiável, pois o usuário poderia modificar o código no navegador nas **ferramentas do desenvolvedor**. Nós podemos fazer a validação em nosso código **Java**, que é mais confiável porque o usuário não tem acesso a esse código.

O `Controller` de `Produto`, ou seja `ProdutosController`, parece ser um bom lugar para validar os dados dos produtos. E o método `gravar` parece ser o lugar onde precisamos de uma validação. É certo validar os dados antes de salvar o produto no banco de dados?

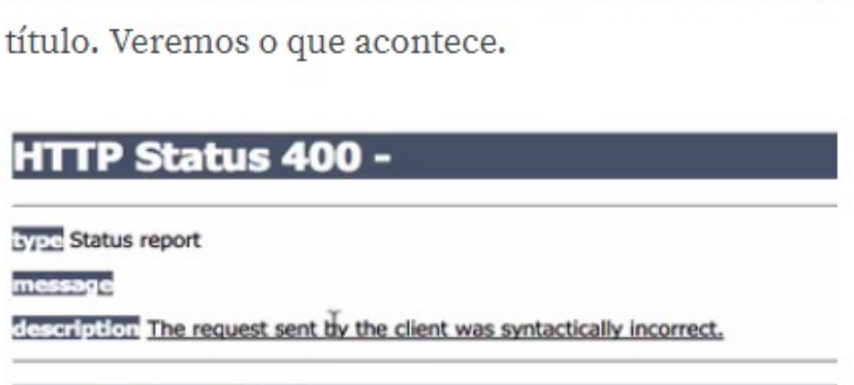
Validações geralmente envolvem `ifs`, que avaliam uma condição e garantem se está tudo certo. Nosso método `grava` atualmente está assim:

```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView gravar(Produto produto, RedirectAttributes redirect
    System.out.println(produto);
    produtoDao.gravar(produto);
    redirectAttributes.addFlashAttribute("message","Produto cadastrado
    return new ModelAndView("redirect:produtos");
}
```

Vamos primeiro tirar esse `System.out.println(produto)`, porque ele não é mais necessário. Então vamos fazer um `if` antes de salvar o `produto`, verificando se o título é `null` ou vazio. Devemos validar pelo menos o título, a descrição e o preço, mas vamos começar com o título. Caso um dos casos da validação dê `false`, retornaremos para o formulário. Veja como fica nosso código com o `if`:

```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView gravar(Produto produto, RedirectAttributes redirect
    if(produto.getTitulo() == null || produto.getTitulo().isEmpty()){
        return form();
    }
    produtoDao.gravar(produto);
    redirectAttributes.addFlashAttribute("message","Produto cadastrado
    return new ModelAndView("redirect:produtos");
}
```

Vamos fazer dois testes agora, um com o título preenchido e outro, sem preenchê-lo. Lembre-se de deixar os outros campos em branco, pois estamos validando somente o título. Veremos o que acontece.



HTTP Status 400 -

type Status report

message

description The request sent by the client was syntactically incorrect.

Apache Tomcat/7.0.63

Nos dois casos que testamos agora, teremos o mesmo erro de antes e não voltaremos para o formulário. Por quê?

O erro acontecerá antes mesmo de chegar ao nosso código de validação. O **Spring** não está conseguindo fazer o `bind` dos dados do formulário para o nosso **objeto** `produto`. Vejamos então os atributos da nossa classe `Produto`.

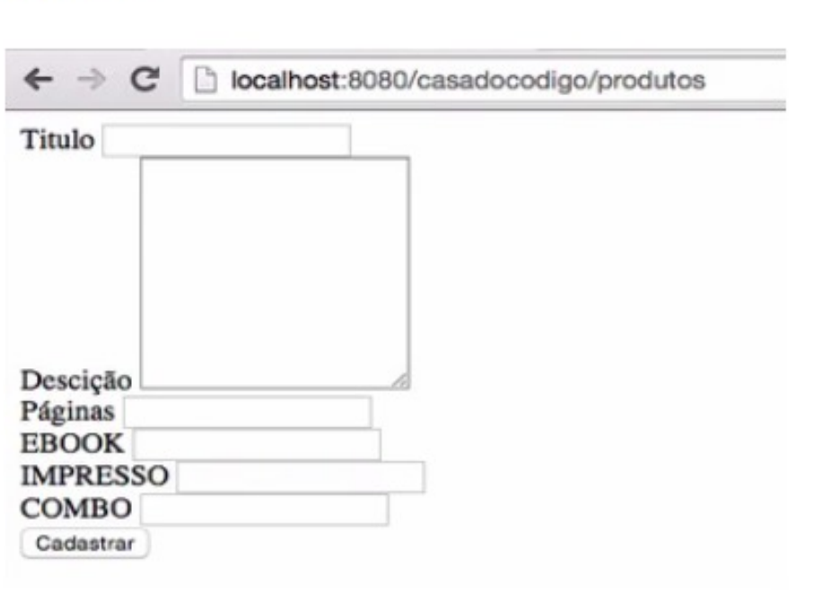
```
@Entity
public class Produto {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;

    private String titulo;
    private String descricao;
    private int paginas;

    @ElementCollection
    private List<Preco> precos;
    [...]
}
```

Se olharmos bem, temos o atributo `paginas` que é do tipo `int`. Este é um tipo primitivo, ele não pode receber uma valor `null`. No entanto, os outros como `String` e `List` podem, pois são objetos. O `id` não nos importa nesse momento, porque é gerenciado pelo **Spring**. Vamos, então, enviar o formulário preenchendo somente o número de páginas.



Agora sim, voltamos para o nosso formulário como fizemos em nossa validação. Perceba que apesar de preencher o número de páginas e o formulário ter enviado os dados sem problemas, não temos nenhuma mensagem de erro.

O problema da validação ser feita desta forma é que se for preciso validar mais campos, aquele `if` em nosso `controller` vai ficar muito grande e isso não é bom. Pior ainda se for preciso validar o `produto` em outra parte da aplicação. Teríamos que ficar copiando e colando código. Sabemos que isso também não é bom. A melhor ideia é isolar o código em outro lugar e sempre que precisarmos, usá-lo.

Vamos criar então uma classe para esse código, que chamaremos de `ProdutoValidation` e ela ficará no pacote `br.com.casadocodigo.loja.validation`. Esta classe terá um método `boolean` chamado `valida` que fará a mesma coisa que nosso `if` do `controller` e retorna `true` caso o produto tenha os dados corretos e `false` caso não tenha. Nossa classe `ProdutosValidation` ficará assim:

```
public class ProdutoValidation {
    public boolean valida(Produto produto) {
        if (produto.getTitulo() == null || produto.getTitulo().isEmpty(
            return false;
        }
        return true;
    }
}
```

Este código terá o mesmo efeito do outro que estava no nosso `ProdutosController` e continuará com um problema que quase não notamos. Por hora, estamos validando somente o título, mas se fizermos a validação da descrição e depois tentarmos cadastrar um produto sem nenhuma dessas informações, como vamos saber qual delas deu erro? O código não nos informa isso. Apesar de funcionar, o código não nos ajuda muito. Então, vamos usar algo do **Spring**. O **Framework** deve poder nos ajudar. Neste caso, o **Spring** tem uma classe chamada `ValidationUtils`, com alguns métodos que validam dados.

Dentre os métodos disponíveis em `ValidationUtils`, teremos um que se encaixa exatamente com nosso caso, que é o `rejectIfEmpty`, que traduzido para português significa "rejeite se for vazio". É exatamente o que queremos. Este método recebe três parâmetros: Um **objeto** `errors` que contém os erros da validação; O nome do campo que iremos validar passado como **String**; e um `errorCode` que também é passado como **String**, mas que é reconhecido pelo **Spring**. Neste último parâmetro, usaremos o `errorCode` com o valor `"field.required"` para informar ao **Spring** que aquele campo é obrigatório.

Note que o **objeto** `errors` não é gerenciado por nós, mas sim pelo **Spring**. Nosso método agora não precisa mais retornar nenhum valor, já que o objeto `errors` terá as informações se a validação falhou ou não. Vejamos como fica o método `valida` após essas modificações.

```
public void valida(Produto produto, Errors errors) {
    ValidationUtils.rejectIfEmpty(errors, "titulo", "field.required");
}
```

Bem mais simples não acha? Vamos aproveitar o momento e já validar da mesma forma o nosso campo `descricao`. Veja como fica:

```
public void valida(Produto produto, Errors errors) {
    ValidationUtils.rejectIfEmpty(errors, "titulo", "field.required");
    ValidationUtils.rejectIfEmpty(errors, "descricao", "field.required"
}
```

Para validar o campo `paginas` nós precisamos de uma outra estratégia, pois nosso campo precisa de um número maior que zero não é verdade? Neste caso, teremos que fazer um `if` verificando isso. Caso o número de páginas seja **menor ou igual a zero**, usaremos o objeto `errors` para **rejeitar o valor**, passando também o `errorCode` de campo obrigatório. Nosso método `valida` ficará dessa forma:

```
public void valida(Produto produto, Errors errors) {
    ValidationUtils.rejectIfEmpty(errors, "titulo", "field.required");
    ValidationUtils.rejectIfEmpty(errors, "descricao", "field.required"
    if(produto.getPaginas() <= 0){
        errors.rejectValue("paginas", "field.required");
    }
}
```

Está quase tudo pronto para nossa validação ser finalizada. Iremos agora usar a validação em nosso controller e mostrar os devidos erros em nossa `view`. Para não nos preocuparmos em ficar fazendo `ifs` em nosso código, podemos dizer para o **Spring** usar a nossa classe de validação para validar o `produto`. Para isso precisamos de algumas configurações.

Vamos usar a especificação do Java chamada **Bean Validation**. A implementação desta especificação que vamos usar será a **Hibernate Validator** . Usaremos algumas bibliotecas para facilitar nosso trabalho. No `pom.xml` adicionaremos essas dependências:

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.0.0.GA</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.2.0.Final</version>
</dependency>
```

Nosso próximo passo é configurar nosso projeto para usar essas bibliotecas e fazer com que o **Spring** passe a usar nossa classe **ProdutoValidation** para validar nosso produto. No `ProdutosController` vamos usar uma nova anotação. No método gravar, antes da assinatura do Objeto `produto` adicione a anotação `@valid` do pacote: `javax.validation`. A assinatura do método deve ficar dessa forma:

```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView gravar(@Valid Produto produto, RedirectAttributes r
    [...]
}
```

Isso ainda não é o suficiente para nossa validação funcionar. Agora precisamos criar um método em nosso controller chamado `InitBinder` que terá uma anotação com o mesmo nome do método `@InitBinder`. Este método recebera um **objeto** do tipo `WebDataBinder` que chamaremos apenas de `binder`. O objeto `binder` tem um médoto chamado `addValidators` que recebe uma instância de uma classe que implemente a **interface** `Validator` do pacote `org.springframwork.validation`.

Observação: O **Binder**, por assim dizer, é o responsável por conectar duas coisas. Por exemplo, os dados do formulário com o objeto da classe `Produto`, como já fizemos anteriormente.

Dito isso, vamos implementar esse método `InitBinder` em nosso `ProdutosController`, então. Nosso código deve ficar parecido com esse:

```
@InitBinder
public void InitBinder(WebDataBinder binder){
    binder.addValidators(new ProdutoValidation());
}
```

Com este passo pronto, notamos que o Eclipse irá reclamar que o método `addValidators` não é um objeto `Validator`. Isso é justo, já que nossa classe `ProdutoValidation` não implementa essa interface. Vamos então modificar nossa classe `ProdutoValidation` para que ela implemente a interface correta. A interface `Validator` correta é a que está no pacote `org.springframework.validation`.Fazendo com que a nossa classe `ProdutoValidation` implemente esta interface, adicionaremos os métodos da interface, nossa classe fica dessa forma:

```
public class ProdutoValidation implements Validator {
    public void valida(Produto produto, Errors errors) {
        ValidationUtils.rejectIfEmpty(errors, "titulo", "field.required
        ValidationUtils.rejectIfEmpty(errors, "descricao", "field.requi
        if(produto.getPaginas() <= 0){
            errors.rejectValue("paginas", "field.required");
        }
    }

    @Override
    public boolean supports(Class<?> arg0) {
        return false;
    }

    @Override
    public void validate(Object arg0, Errors arg1) {

    }
}
```

Perceba que o método `Validate` é onde realmente acontecerá a validação. Este método se parece bastante com o método `valida`, criado anteriormente. Então, vamos usar o médoto da interface. Primeiro renomearemos os parâmetros, `arg0` se chamará `target`, este será o objeto alvo da validação. O `arg1` será o `errors` igual ao método `valida`. Depois deste passo, vamos copiar o código dentro do método `valida` e colá-lo no método `validate`. Podemos apagar o método `valida` depois disso. Até aqui nosso código deve estar parecido com esse:

```
public class ProdutoValidation implements Validator {
    @Override
    public boolean supports(Class<?> arg0) {
        return false;
    }

    @Override
    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmpty(errors, "titulo", "field.required
        ValidationUtils.rejectIfEmpty(errors, "descricao", "field.requi

        if(produto.getPaginas() <= 0){
            errors.rejectValue("paginas", "field.required");
        }
    }
}
```

O Eclipse vai reclamar, pois no `if` estamos usando um objeto `produto` e não temos mais esse objeto em mãos. O que podemos fazer então? Não podemos simplesmente mudar o nome `target` porque estamos recebendo um objeto do tipo `Object` agora. A forma mais simples é fazer um `cast` antes do `if`. Nosso código ficará assim então:

```
public class ProdutoValidation implements Validator {
    @Override
    public boolean supports(Class<?> arg0) {
        return false;
    }

    @Override
    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmpty(errors, "titulo", "field.required
        ValidationUtils.rejectIfEmpty(errors, "descricao", "field.requi

        Produto produto = (Produto) target;
        if(produto.getPaginas() <= 0){
            errors.rejectValue("paginas", "field.required");
        }
    }
}
```

O método `supports` também precisa ser implementado. A implementação desse método indica a qual classe a validação dará suporte. Sabemos que será a classe Produto. Vamos então fazer essa implementação da seguinte forma:

```
public class ProdutoValidation implements Validator {
    @Override
    public boolean supports(Class<?> clazz) {
        return Produto.class.isAssignableFrom(clazz);
    }
}
```

O que esse código faz é verificar se o objeto recebido para a validação tem uma assinatura da classe `Produto`. Com isso o **Spring** pode verificar se o objeto é uma instância daquela classe ou não.

Um último passo é necessário agora, para nossa validação estar completa. Falta recebermos o resultado da verificação, da validação em sí em nosso controller e verificar se houve algum erro. Faremos isto recebendo na assinatura do nosso método `gravar` um objeto do tipo `BindingResult` que tem um método chamado `hasErrors`, que informa se houve erros de validação ou não. Com isso poderemos fazer um simples `if`, para ver se tudo funciona bem. Caso a validação não tenha encontrado erros, salvaremos o produto. Porém, se houver erros, voltaremos para o formulário. Nosso código no método `gravar` ficará dessa forma:

```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView gravar(@Valid Produto produto, BindingResult result
    if(result.hasErrors()){
        return form();
    }
    produtoDao.gravar(produto);
    redirectAttributes.addFlashAttribute("message","Produto cadastrado
    return new ModelAndView("redirect:produtos");
}
```

Note que o `BindingResult` vem logo após o atributo que está assinado com a anotação `@Valid`, essa ordem não é por acaso, precisa ser desta forma para que o **Spring** consiga fazer as validações da forma correta. Teste o formulário mais uma vez!

Agora podemos enviar o formulário sem preencher nenhum dos campos e nada acontecerá. Ainda falta mostrar as informações de erro - as mensagem. Mas logo faremos isto. Até o momento, basta acreditar que a validação está acontecendo.

TIRAR DÚVIDA
PRÓXIMA ATIVIDADE