Artigo

Conhecendo o Hibernate Validator

Este artigo apresenta o Hibernate Validator, implementação de referência da JSR 303 – Bean Validation API. Esta API permite validar, de forma fácil, objetos de classes que representam o domínio de uma aplicação.

De que se trata o artigo:

Este artigo apresenta o Hibernate Validator, implementação de referência da JSR 303 – Bean Validation API. Esta API permite validar, de forma fácil, objetos de classes que representam o domínio de uma aplicação.

Este artigo serve para introduzir os desenvolvedores à API Hibernate Validator, que possibilita incorporar validação aos dados da aplicação através de uma API fácil de usar e de customizar.

Em que situação o tema é útil:

Os desenvolvedores que desejam aprender como o Hibernate Validator funciona e como integrá-lo em suas aplicações encontrarão neste artigo explicações detalhadas sobre o funcionamento desta API.

Conhecendo o Hibernate Validator:

A Bean Validation API, representada pela JSR 303, foi disponibilizada em dezembro de 2009. O Hibernate Validator surgiu como a implementação de referência desta API e permite utilizar anotações para validar dados de forma fácil e rápida. A grande vantagem é que a Bean Validation API independe da camada da aplicação onde é usada e até da forma de programar, podendo ser utilizada nos mais diversos cenários. Este artigo aborda em detalhes o funcionamento do Hibernate Validator.

Em dezembro de 2009, a versão final da JSR 303 foi disponibilizada. Trata-se da Bean Validation API, que permite validar dados de forma fácil e rápida através do uso de anotações. A proposta desta JSR é a validação dos dados presentes nas classes que modelam o domínio da aplicação, que comumente seguem o padrão JavaBeans. O que é mais interessante é que a Bean Validation API independe da camada da aplicação onde é usada e até da forma de programar. Ela pode ser usada tanto em aplicações web como desktop, além de não estar atrelada ao

A validação de dados sempre esteve presente em sistemas que recebem entrada de dados do usuário. Cada framework implementava um mecanismo proprietário para validar as informações, o que criava problemas de incompatibilidade e dificultava a integração.

Com o surgimento da JSR 303 foi estabelecida uma API padrão para validação, que é flexível o suficiente para ser utilizada pelos mais diversos tipos de frameworks. Além disso, a Bean Validation API possibilita a validação de dados nas classes do domínio da aplicação, que é mais simples do que quando o processo é feito por camada. Na validação por camada, é necessário verificar o mesmo dado diversas vezes (nas camadas de apresentação, negócio, persistência, etc.), a fim de garantir a consistência da informação. Ao validar os dados diretamente nas classes de domínio, o processo todo fica centralizado, uma vez que os objetos destas classes normalmente trafegam entre as camadas aplicação.

A implementação de referência da JSR 303 é o Hibernate Validator, que será explicado em detalhes no decorrer deste artigo.

Configurando o Hibernate Validator

Para trabalhar com a API é necessário utilizar a versão 5 do Java ou superior, já que o Hibernate Validator não é compatível com versões anteriores do JDK. A configuração da aplicação para usar a API é simples. O primeiro passo é fazer o download no site oficial do projeto, que pode ser consultado na seção de referências. A versão disponível até o momento da escrita deste artigo é a 4.1.0.

Após a descompactação do arquivo (que pode ser baixado nos formatos ZIP ou

Caso a sua aplicação use o Java 5 ao invés do 6, é necessário adicionar também os JARs da API JAXB: *jaxb-api-2.2.jar* e *jaxb-impl-2.1.12.jar*. Este conjunto de arquivos é o mínimo necessário para colocar o Hibernate Validator para funcionar.

Definindo restrições com o uso de annotations

As restrições são utilizadas para definir regras a respeito dos dados de um objeto. Você pode, por exemplo, definir que um determinado atributo não pode ser nulo ou que o valor de um atributo numérico deve pertencer a um intervalo bem definido. Quando o processo de validação dos dados é executado, é feita uma verificação para checar se os dados estão de acordo com as regras estabelecidas.

Quando é necessário adicionar restrições de validação no código, isto é feito através da utilização de annotations. O Hibernate Validator já possui um conjunto de annotations para validações mais comuns, embora a API possibilite que o programador faça customizações. A criação de restrições customizadas será abordada na sequência deste artigo.

Existem dois locais onde as restrições podem ser aplicadas. A primeira é diretamente no atributo da classe. Neste caso o Java acessa diretamente o atributo via reflexão a fim de fazer a validação. A **Listagem 1** mostra um exemplo. A segunda forma de aplicação de restrições pode ser utilizada quando a classe segue a especificação de um JavaBean. Neste caso é possível usar a annotation no método getter do atributo. A **Listagem 2** mostra como isto pode ser feito. Você deve escolher apenas uma das opções, já que se ambas forem usadas a validação será feita duas vezes.

```
3   @NotNull
4   private String nome;
5 }
```

Listagem 2. Usando a restrição @NotNull no método getter.

```
1 public class Aluno {
2
3   @NotNull
4   public String getNome() {
5     return nome;
6   }
7 }
```

> Todos os elementos do Hibernate Validator (classes, interfaces, annotations, etc.) utilizados pelo seu código pertencem ao pacote **javax.validation**.

Além da annotation @NotNull mostrada no exemplo, que verifica se o dado não é nulo, existem outras annotations importantes presentes no Hibernate Validator. Na sequência são abordadas algumas delas com mais detalhes. Para mais informações sobre estas e outras annotations, consulte a JSR 303 e a documentação do Hibernate Validator.

@AssertFalse e @AssertTrue

Estas annotations validam se o dado é falso ou verdadeiro, respectivamente. Devem ser aplicadas em dados booleanos (o tipo primitivo boolean e a classe Boolean são suportados). A **Listagem 3** mostra ambas as restrições. O atributo

Listagem 3. Usando as annotations @AssertFalse e @AssertTrue.

```
1  public class Aluno {
2
3    @AssertFalse
4    private boolean possuiAdvertencia;
5
6    @AssertTrue
7    private boolean maiorDeIdade;
8  }
```

@Min e @Max

Estas annotations validam se um dado numérico tem um valor mínimo ou máximo, respectivamente. São aplicadas a dados dos tipos BigDecimal, BigInteger, String, byte, short, long e suas classes wrappers correspondentes. A **Listagem 4** mostra um exemplo onde o atributo nota deve ter um valor mínimo igual a 70 e máximo igual a 100. Este exemplo mostra que é possível utilizar mais de uma restrição para o mesmo atributo ou método getter.

Listagem 4. Usando as annotations @Min e @Max.

```
1  public class Aluno {
2
3    @Min(70)
4    @Max(100)
5    private int nota;
6  }
```

@Size

objetos dos tipos Collection e Map. A **Listagem 5** mostra um exemplo onde o atributo rua (do tipo String) deve ter um tamanho mínimo de 10 e máximo de 50 caracteres.

Listagem 5. Usando a annotation @Size.

```
1  public class EnderecoAluno {
2
3    @Size(min = 10, max = 50)
4    private String rua;
5  }
```

@Pattern

Esta annotation permite validar o dado de acordo com uma expressão regular, que é especificada pelo atributo regexp. Funciona para dados cujo tipo é String. A **Listagem 6** mostra como usá-la para validar um CEP.

Listagem 6. Usando a annotation @Pattern.

```
1  public class EnderecoAluno {
2
3    @Pattern(regexp = "[0-9]{5}-[0-9]{3}")
4    private String cep;
5  }
```

@Valid

É bastante comum a existência de classes que possuem atributos que referenciam

Listagem 7 mostra um exemplo onde, ao invocar a validação em um objeto da classe Aluno, o objeto da classe EnderecoAluno será validado também, em cascata. Neste caso, se o atributo rua do endereço estiver nulo, a validação irá falhar.

Listagem 7. Usando a annotation @Valid.

```
1  public class Aluno {
2
3    @Valid
4    private EnderecoAluno endereco;
5  }
6
7  public class EnderecoAluno {
8
9    @NotNull
10   private String rua;
11 }
```

Invocando o processo de validação

Após a definição das restrições em forma de annotations, o próximo passo é invocar o código que vai efetivamente fazer a validação dos dados de acordo com as regras que foram estabelecidas.

Para isto, o primeiro objeto necessário é uma instância de ValidatorFactory, que será utilizada para construir um objeto do tipo Validator. O código típico para obter um objeto Validator é mostrado na **Listagem 8**.

Listagem 8. Obtendo um objeto Validator.

pode ser feita de diferentes maneiras, que serão explicadas em detalhes na sequência e terão como base o código da **Listagem 9**. Perceba que as restrições são aplicadas no atributo nome e no método getNota().

Listagem 9. Restrições definidas na classe Aluno.

```
1  public class Aluno {
2
3    @NotNull
4    @Size(max = 40)
5    private String nome;
6
7    private int notaAluno;
8
9    @Min(70)
10   @Max(100)
11   public int getNota() {
12     return notaAluno;
13   }
14 }
```

A primeira maneira para invocar o processo de validação é chamar o método validate(), que recebe como parâmetro o objeto a ser validado. Para um objeto aluno da classe Aluno, a chamada validator.validate(aluno) realiza a validação no objeto de acordo com todas as restrições definidas.

A segunda opção é usar o método validateProperty(). Ao contrário do método validate(), ele permite invocar a validação em apenas uma propriedade do objeto. O nome da propriedade fornecido é o nome do atributo (caso a restrição esteja definida no atributo) ou o nome que segue a especificação de um JavaBean (caso a restrição esteja definida no método). Ao invocar o método

A terceira e última opção é fazer uma simulação de validação através do método validateValue(). Desta forma é possível verificar se a validação funcionaria para uma propriedade caso um determinado valor fosse atribuído a ela. Para verificar se a propriedade nota da classe Aluno poderia assumir o valor 60, por exemplo, basta invocar validator.validateValue(Aluno.class, “nota”, 60).

> Quando uma superclasse tem restrições definidas e um objeto da sua subclasse é submetido à validação, as restrições da superclasse também são validadas.

Os três métodos explicados anteriormente (validate(), validateProperty() e validateValue()) possuem como retorno um Set<ConstraintViolation<T>>. Através desta coleção é possível verificar se ocorreram problemas de validação e identificá-los. Quando não ocorre nenhum, o Set retorna vazio. Já se ocorrer um ou mais erros, cada item armazenado no Set representa um erro de validação. A Listagem 10 mostra um exemplo de código que pode ser utilizado para exibir no console os erros de validação ocorridos no objeto aluno.

Listagem 10. Exibindo os erros de validação do objeto no console.

```
1  Set<ConstraintViolation<Aluno>> erros = validator.validate(aluno);
2
3  for (ConstraintViolation<Aluno> erro : erros) {
4    String msgErro = erro.getMessage();
5    System.out.println(msgErro);
6  }
```

modelo (template). Este artigo abordará, na sequência, como trabalhar com a customização de mensagens de erro. Outros método úteis são o getInvalidValue(), que retorna o valor da propriedade do objeto que causou o problema de validação; e o getRootBean(), que retorna a referência do objeto cuja validação falhou.

> O método **validateValue()** não recebe como parâmetro um objeto a ser validado. Neste caso a chamada **getRootBean()** retorna **null**.

Usando grupos de validação

Muitas vezes é desejável separar em grupos as validações que devem ser realizadas. Isso permite a validação de apenas parte das restrições definidas na classe, e não todas elas de uma única vez.

O Hibernate Validator suporta esta separação. Cada grupo é normalmente representado por uma interface de marcação, que não possui métodos ou atributos. É importante compreender que toda vez que o processo de validação é invocado, deve ser especificado um ou mais grupos de validação. Quando um grupo não é fornecido explicitamente, o Hibernate Validator assume o grupo javax.validation.groups.Default como padrão. A **Listagem 11** mostra a classe Aluno e as restrições feitas aos seus atributos. Como as restrições não referenciam um grupo específico, o padrão é utilizado.

Listagem 11. Restrições da classe Aluno definidas no grupo padrão de validação.

```
6     @Min(3)
7     private int idade;
8
9     @Size(max = 10)
10    private String numMatricula;
11
12    @NotNull
13    private Date dataMatricula;
14  }
```

A validação sempre deve estar associada a um ou mais grupos, que são passados como parâmetro para os métodos validate(), validateProperty() e validateValue(). Para validar o grupo padrão, basta omitir esta informação (as chamadas validate(aluno) e validate(aluno, javax.validation.groups.Default.class) têm exatamente o mesmo efeito neste caso). A invocação do processo de validação leva em consideração apenas as restrições dos grupos especificados e ignora as demais. Neste ponto, vale ressaltar que cada restrição pode pertencer a um ou mais grupos.

Sabendo agora que toda invocação de validação é feita em um ou mais grupos e que toda restrição pertence a pelo menos um grupo, fica fácil compreender o processo de agrupamento de restrições. O primeiro passo é a criação dos grupos, e a **Listagem 12** mostra a definição de dois deles: DadosPessoais e DadosMatricula.

Listagem 12. Definição dos grupos DadosPessoais e DadosMatricula.

```
1  public interface DadosPessoais {
2  }
3
4  public interface DadosMatricula {
```


annotations usadas para validação. A **Listagem 13** mostra os atributos nome e idade sendo atribuídos ao grupo DadosPessoais, e numMatricula e dataMatricula sendo atribuídos ao DadosMatricula. Caso seja necessário atribuir mais de um grupo a uma restrição, basta fornecer um array com as interfaces ao elemento groups da annotation. Se uma restrição @NotNull fosse aplicada a ambos os grupos, por exemplo, ela poderia ser declarada como @NotNull(groups = {DadosPessoais.class, DadosMatricula.class}).

Listagem 13. Organizando as restrições em grupos.

```
1  public class Aluno {
2
3    @NotNull(groups = DadosPessoais.class)
4    private String nome;
5
6    @Min(value = 3, groups = DadosPessoais.class)
7    private int idade;
8
9    @Size(max = 10, groups = DadosMatricula.class)
10   private String numMatricula;
11
12   @NotNull(groups = DadosMatricula.class)
13   private Date dataMatricula;
14 }
```

Por último, basta informar o grupo a ser validado no momento de invocar a validação. Aliás, é possível inclusive validar mais de um grupo em apenas uma invocação. A **Listagem 14** mostra três exemplos de chamada ao método validate(). As duas primeiras invocam a validação para um grupo de cada vez, e a terceira para os dois grupos. Lembre-se sempre de que as restrições que não fazem parte dos grupos onde a validação está sendo realizada são ignoradas.

```
3 | validator.validate(aluno, DadosPessoais.class, DadosMatricula.class);
```

Além da separação das validações em grupos, outro aspecto importante deve ser considerado. Por padrão, não existem garantias a respeito da ordem de validação das restrições. Em casos onde garantir a ordem é importante, pode-se usar a annotation @GroupSequence.

@GroupSequence permite definir uma ordem de execução para os grupos. Suponha que você deseja executar as validações nos grupos DadosPessoais e DadosMatricula, respeitando esta ordem. Para isso, você precisa criar um novo grupo e definir quais grupos ele irá validar e em qual ordem. A **Listagem 15** mostra a criação do grupo DadosCompletos, que define que primeiro será feita a validação no grupo DadosPessoais e depois no DadosMatricula. Para validar este novo grupo basta referenciá-lo no momento da validação. Ao invocar validate(aluno, DadosCompletos.class) serão executadas as validações dos grupos DadosPessoais e DadosMatricula, nesta ordem.

Listagem 15. Definição do grupo DadosCompletos com uma ordem de validação definida.

```
1 | @GroupSequence({ DadosPessoais.class, DadosMatricula.class })
2 | public interface DadosCompletos {
3 | }
```

> Quando um erro de validação é encontrado em um grupo, os próximos grupos não são validados.

definição, basta usar @GroupSequence no nível da classe. A **Listagem 16** mostra o seu uso em Aluno. Agora, quando a validação for invocada para o grupo padrão, os grupos DadosPessoais e DadosMatricula serão chamados e essa ordem continuará sendo respeitada. Outro ponto importante é sempre adicionar a classe que está sendo anotada como integrante do grupo definido por @GroupSequence, como foi feito com a classe Aluno na **Listagem 16**. Caso essa condição não seja atendida, uma GroupDefinitionException será lançada.

Listagem 16. Redefinindo o grupo padrão com a annotation @GroupSequence.

```
1 @GroupSequence({ Aluno.class, DadosPessoais.class, DadosMatricula.cl
2 public class Aluno {
3   //...
4 }
```

Customizando mensagens de erro

Quando você usa o Hibernate Validator e ocorrem problemas de validação você se depara com mensagens do tipo: "*may not be null*", "*must be greater than or equal to 70*" ou "*size must be between 2 and 10*". Fica evidente que nem sempre estas mensagens, que são definidas por padrão na API, são ideais para a sua aplicação.

Por este motivo, o Hibernate Validator permite que customizações sejam feitas nas mensagens. Para entender como este processo funciona, você deve ter em mente que cada restrição tem um *message descriptor* associado. Um message descriptor nada mais é do que um template que define como será a mensagem resultante.

{javax.validation.constraints.NotNull.message}, {javax.validation.constraints.Min.message} e {javax.validation.constraints.Size.message}. Esta informação pode ser facilmente obtida se o método getMessageTemplate() for invocado em uma ConstraintViolation (que representa um erro de validação, como já explicado anteriormente). Toda informação do message descriptor declarada entre chaves é, na verdade, um parâmetro que será substituído no momento de gerar a mensagem final, num processo conhecido como interpolação.

Para realizar a interpolação o Hibernate Validator consulta o valor do parâmetro no arquivo *org/hibernate/validator/ValidationMessages.properties*, que está localizado dentro do JAR do Hibernate Validator. A **Listagem 17** mostra o trecho que corresponde às mensagens das annotations @NotNull, @Min e @Size. Este arquivo mapeia uma chave a um valor, onde a chave é o parâmetro do message descriptor e o valor é o que deve ser gerado como saída quando ocorrer o processo de interpolação.

Listagem 17. Parte do arquivo ValidationMessages.properties disponibilizado pelo Hibernate Validator.

```
1 javax.validation.constraints.NotNull.message=may not be null
2 javax.validation.constraints.Min.message=must be greater than or equ
3 javax.validation.constraints.Size.message=size must be between {min}
```

Para customizar as mensagens de erro, você deve criar um arquivo *ValidationMessages.properties* e redefinir as mensagens. Ele deve estar na raiz do classpath da sua aplicação e tem precedência de leitura sobre o arquivo padrão do

Listagem 17. Arquivo ValidationMessages.properties da aplicação.

```
1 javax.validation.constraints.NotNull.message=O dado não pode ser nul
2 javax.validation.constraints.Min.message=O valor deve ser maior ou i
3 javax.validation.constraints.Size.message=O tamanho do dado deve ser
```

Outra possibilidade que a API disponibiliza é sobrescrever o message descriptor de apenas uma restrição. Isto é feito através do atributo message da annotation que representa a restrição. A **Listagem 18** mostra como é feita a redefinição do message descriptor da annotation @NotNull para o nome do aluno. Como o parâmetro {aluno.nome} é utilizado, deve existir uma entrada correspondente no arquivo *ValidationMessages.properties*. Considerando que esta entrada seja aluno.nome = nome do aluno, ao ocorrer um erro de validação neste atributo a mensagem gerada seria “O nome do aluno não pode ser nulo”.

Listagem 18. Redefinindo o message descriptor de @NotNull.

```
1 public class Aluno {
2
3   @NotNull(message = "O {aluno.nome} não pode ser nulo")
4   private String nome;
5 }
```

É possível também utilizar atributos da própria annotation como parâmetros no message descriptor. Considere o exemplo da annotation @Size, onde os elementos min e max definem o intervalo de tamanho permitido. A **Listagem 19** mostra como definir um message descriptor que lê estes valores do intervalo. Deste modo, quando ocorrer um erro de validação no tamanho do número da

```
1  public class Aluno {
2
3    @Size(min = 5, max = 10, message = "Matrícula deve ter tamanho ent
4    private String numMatricula;
5  }
```

> Ao realizar o processo de interpolação, o Hibernate Validator procura valores para serem substituídos pelos parâmetros do message descriptor numa ordem bem definida. Primeiro a busca é feita no ValidationMessages.properties da aplicação. Se a informação não for encontrada, a busca passa a ser feita no ValidationMessages.properties do Hibernate Validator. Se ainda assim a informação não for encontrada, o algoritmo busca nos elementos da annotation.

Criando restrições customizadas

Como já foi abordado anteriormente, o Hibernate Validator possui um conjunto de restrições que podem ser usadas para validar dados (como @NotNull, @Size, @Min, @Max, entre outras). Mas nem sempre estes validadores são suficientes para todos os tipos de aplicação. A saída nestes casos é que o desenvolvedor crie suas próprias restrições, o que é suportado pela API.

Para exemplificar o processo de criação de uma restrição de validação, será criada a annotation @Numeric. Ela permitirá validar se o conteúdo de uma string é um dado numérico que pode ser convertido para os tipos long ou double.

O primeiro passo é criar a annotation propriamente dita, como mostra a

devem ser declarados.

O primeiro é o message(), que define um descritor de mensagem para a restrição, neste caso {app.Numeric.Message}. Lembre-se que o descritor define como será a mensagem de erro de validação gerada pelo processo de interpolação. O outro atributo é o groups(), que define os grupos de validação aos quais a restrição pertence. É obrigatório que o valor padrão seja definido como uma string vazia.

O atributo payload é um array de objetos Class que implementam a interface de marcação Payload. Ele pode ser usado em situações onde é necessário associar outras informações à restrição. Um exemplo prático onde este atributo pode ser útil é citado na especificação do Bean Validation e leva em conta o seguinte cenário. Considere uma restrição declarada como @NotNull(payload = ErrorLevel1.class). Se ocorrer um erro de validação, é possível recuperar o payload através do objeto ConstraintViolation, invocando getConstraintDescriptor().getPayload(). Considere também que podem existir outras interfaces, como ErrorLevel2 ou ErrorLevel3 (cada uma representa um nível de erro), que podem ser associadas a outras restrições. Um framework que atua na camada de apresentação poderia ler o payload associado à restrição que ocasionou o problema de validação e mostrar a mensagem de erro para o usuário de formas diferentes, de acordo com este atributo (para cada nível poderia ser adotada uma cor diferente para a mensagem, por exemplo). Normalmente, o valor padrão utilizado para o atributo payload na declaração da restrição é um array vazio.

Os atributos citados anteriormente são obrigatórios e precisam ser declarados, mas outros atributos podem ser criados. No exemplo da **Listagem 20**, o atributo

Listagem 20. Definição da annotation @Numeric.

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = NumericValidator.class)
public @interface Numeric {
  String message() default "{app.Numeric.message}";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
  NumberType value();
}
```

Listagem 21. Definição do enum NumberType.

```
public enum NumberType {
  LONG,
  DOUBLE
}
```

Além dos atributos, é possível perceber que a própria annotation @Numeric é anotada. @Target(ElementType.FIELD) indica que @Numeric só pode ser usada em atributos. @Retention(RetentionPolicy.RUNTIME) indica que a JVM consegue, em tempo de execução, identificar a presença da annotation em um atributo. Isto é imprescindível, pois o Hibernate Validator precisa identificá-la através do uso de reflexão. E @Constraint(validatedBy = NumericValidator.class) indica que a classe NumericValidator é quem implementa a lógica que vai validar a restrição.

Terminado este primeiro passo, é preciso agora implementar a classe NumericValidator, que pode ser vista na **Listagem 22**. Esta classe deve implementar a interface ConstraintValidator<A extends Annotation, T>. O

String.

> Caso a restrição se aplique a vários tipos de dados, é necessário implementar um **ConstraintValidator** para cada um deles. Em situações como essas é preciso fornecer um array de objetos **Class** para o atributo **validatedBy** da annotation **@Constraint**.

Listagem 22. Implementação da classe NumericValidator.

```
1  public class NumericValidator implements ConstraintValidator<Numeric
2    private Numeric constraint;
3
4    public void initialize(Numeric numeric) {
5      this.constraint = numeric;
6    }
7
8    public boolean isValid(String str, ConstraintValidatorContext ctx)
9      if (str == null) {
10       return true;
11     }
12
13     boolean isOk = true;
14
15     if (constraint.value() == NumberType.LONG) {
16       try {
17         Long.parseLong(str);
18       } catch (NumberFormatException e) {
19         isOk = false;
20       }
21     } else if (constraint.value() == NumberType.DOUBLE) {
22       try {
23         Double.parseDouble(str);
24       } catch (NumberFormatException e) {
```


```
31
32  }
```

Ao criar a classe, é preciso implementar dois métodos: initialize() e isValid(). O initialize() é invocado quando o objeto da classe é inicializado. Este método recebe como parâmetro uma referência à annotation correspondente, que é normalmente copiada para um atributo para ser usada futuramente no método isValid(). O método isValid() é onde fica a lógica de implementação da validação e é invocado quando o dado deve ser validado. O retorno é um booleano, onde true indica que não houve erros de validação e false indica que o dado não é válido, considerando as regras estabelecidas. Como parâmetro, o método isValid() recebe o dado a ser validado (neste caso um objeto da classe String) e um ConstraintValidatorContext, que permite sobrescrever o descritor de mensagem definido para a restrição.

O algoritmo da **Listagem 22** é bastante simples. Ele usa o atributo value() da annotation @Numeric, que indica o tipo numérico esperado na string, e tenta fazer a conversão para o tipo de dado correspondente (long ou double). Se conseguir, o algoritmo retorna true. Caso não consiga, retorna false. Perceba que se a string for nula o algoritmo considera que a string é válida. Esta é uma recomendação da própria especificação da Bean Validation API, que diz que a validação não deve falhar quando os valores forem nulos (para validar se o dado é nulo, basta usar a restrição em conjunto com @NotNull).

O último passo antes que a nova restrição criada possa ser usada é adicionar ao arquivo *ValidationMessages.properties* a chave *app.Numeric.message* associada a uma mensagem de erro. Esta chave foi definida como padrão no atributo message

Agora a nova restrição @Numeric está pronta para ser utilizada. Observe um exemplo de uso na **Listagem 23**. Ele valida se o número de matrícula do aluno, armazenado em uma string, é um número válido do tipo long.

Listagem 23. Exemplo de uso da annotation @Numeric.

```
1  public class Aluno {
2
3    @Numeric(NumberType.LONG)
4    private String numMatricula;
5  }
```

Compondo restrições

Para entender o objetivo da composição de restrições, observe o código da **Listagem 24**. Neste exemplo, o atributo numMatricula, para ser válido, deve passar por três validações: @NotNull, @Size e @Numeric. Agora imagine que sua aplicação possui outros números de matrícula em outros objetos, que devem ser validados da mesma forma. Seria necessário adicionar as três restrições em vários lugares diferentes, o que pode gerar dificuldades de manutenção futura.

Listagem 24. Aplicação de várias restrições em um atributo.

```
1  public class Aluno {
2
3    @NotNull
4    @Size(min = 5, max = 10)
5    @Numeric(value = NumberType.LONG)
6    private String numMatricula;
7  }
```


A criação de uma nova restrição que engloba outras segue o mesmo processo de criação de restrições abordado anteriormente. A **Listagem 25** mostra a annotation @Matricula, que agrupa as restrições @NotNull, @Size e @Numeric. A única diferença é que neste caso o atributo validatedBy de @Constraint é definido como um array vazio. Isto porque não é necessário usar uma classe com a lógica de validação, uma vez que @Matricula apenas aproveita as validações já existentes das restrições que ela compõe.

Listagem 25. Definição da restrição @Matricula.

```
1  @NotNull
2  @Size(min = 5, max = 10)
3  @Numeric(value = NumberType.LONG)
4  @Target(ElementType.FIELD)
5  @Retention(RetentionPolicy.RUNTIME)
6  @Constraint(validatedBy = {})
7  public @interface Matricula {
8
9    String message() default "{app.Matricula.message}";
10   Class<?>[] groups() default {};
11   Class<? extends Payload>[] payload() default {};
12 }
```

> É preciso fazer uma pequena modificação na definição da annotation **@Numeric** para que **@Matricula** possa ser definida desta forma. Em **@Target**, é preciso usar tanto **FIELD** como **ANNOTATION_TYPE** para permitir que **@Numeric** possa ser usada para anotar uma annotation (antes ela podia anotar apenas atributos). A configuração fica então desta forma: **@Target({ ElementType.FIELD, ElementType.ANNOTATION_TYPE })**.

caracteres (@Size) e outro dizendo que a matrícula não é um número válido (@Numeric). No entanto, é possível configurar a @Matricula para que o erro de validação gerado seja da própria restrição, e não de cada uma das restrições da composição. Para isto, basta anotar @Matricula com @ReportAsSingleViolation. A partir de agora apenas um erro de validação será reportado, e este erro será referente ao descritor de mensagem {app.Matricula.message}, que foi definido para a restrição @Matricula (essa mensagem poderia ser configurada como "matrícula inválida", por exemplo).

Conclusões

Este artigo abordou em detalhes o funcionamento do Hibernate Validator, que é a implementação de referência da JSR 303 – Bean Validation API, e permite validar de forma fácil dados contidos em objetos. Além de possuir diversas restrições já disponíveis, o Hibernate Validator permite ainda uma ampla customização, de forma que novas restrições possam ser criadas e todas as mensagens de erro possam ser customizadas. O artigo mostrou também outras funcionalidades interessantes, como a criação de grupos de validação para validar dados separadamente, além da composição de restrições para evitar o uso repetitivo de annotations.

Uma das grandes vantagens da Bean Validation API, e consequentemente do Hibernate Validator, é a sua total independência da camada da aplicação onde pode ser aplicada, do modelo de programação (pode ser utilizada em aplicações desktop ou web, por exemplo) e do mecanismo de persistência. Diversos frameworks e APIs que atuam nas mais variadas frentes já fazem integração com

> Site oficial do projeto Hibernate Validator.
>
> **jcp.org/en/jsr/detail?id=303**
>
> Site oficial da JSR 303 – Bean Validation API.

Tecnologias:

Hibernate Java

Por **Carlos**

Em 2010