



Edição 42

DESAFIO:

Consistência de dados com Hibernate Validator
Saiba como evitar a entrada de dados incorretos

Estudando o Singleton e o Abstract Factory

Pratique o reuso de ideias e
soluções na criação de objetos

JSF SEM MISTÉRIOS

Simplificando
as soluções web



ISSN 2179625-4



DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

FORMAÇÃO DESENVOLVEDOR JAVA

DOMINE O DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETO COM **UML** E APRENDA TÓPICOS AVANÇADOS DE PROGRAMAÇÃO COMO O USO DE **SERVLETS, JSP, FRAMEWORKS WEB DENTRO DO PADRÃO MVC.**

➤ A linguagem Java está presente em complexos sistemas como em aplicativos mobile e desktop, servindo a milhares de pessoas. Inicie sua carreira e certifique-se no instituto que valoriza seu currículo. Faça Infnet!

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer



Sumário

Conteúdo sobre Boas Práticas

06 – Estudando os padrões Singleton e Abstract Factory

[Carlos Alberto Silva]

Artigo no estilo Solução Completa

14 – Java Server Faces (JSF): Desenvolvimento de aplicações web

[Lorena S. Dourado]

Artigo no estilo Solução Completa

28 – Hibernate Validator e a consistência dos dados

[Alessandro Jatobá]

CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



CONHEÇA ALGUNS DOS CURSOS:

- Curso de noSQL (Redis) com Java
- Curso Básico de JDBC
- Java Básico: Aplicações Desktop
- JSF com Primefaces
- Conhecendo o Apache Struts

Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Edição 42 • 2014 • ISSN 2173625-4



Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Davi Costa (davigc_08@hotmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse www.devmedia.com.br/central, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Easy Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

CURSOS ONLINE

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Feita para Desenvolvedores de Software e DBAs



CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :

<http://www.devmedia.com.br/cursos/banco-de-dados>
(21) 3382-5038



DEV MEDIA



CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

DESENVOLVA COM AS FERRAMENTAS QUE OS MELHORES DESENVOLVEDORES
USAM: GIT, REDMINE, JENKINS, SONAR, MAVEN E MUITO MAIS.

A suite ToolsCloud é um conjunto integrado das melhores ferramentas para desenvolvimento de software do mercado.

Um ambiente pronto para usar e com suporte, que os desenvolvedores curtem e os gerentes se surpreendem.

- Aumenta a produtividade;
- Facilita a colaboração e comunicação;
- Maior visibilidade e acompanhamento;
- Melhor planejamento e mais agilidade.

Na suite ToolsCloud você tem a sua disposição:

- As ferramentas mais usadas, integradas e de uso imediato;
- Repositório de código (git e svn), integração contínua (Jenkins/Hudson), Gerência de pendências (Redmine), e mais Maven, Nexus, Sonar;
- Infraestrutura robusta, em ambiente de nuvem, suporte e segurança;
- Facilita práticas de agilidade e lean startups;
- Ambiente isolado, customizável e flexível;
- Colaboração garantida: wiki, fóruns de discussão, chat (IRC).

Acompanhe a Toolscloud:

 toolscloud@toolscloud.com



twitter.com/toolscloud



ToolsCloud
toolscloud.com

Estudando os padrões Singleton e Abstract Factory

Praticando o reuso de ideias e soluções na criação de objetos

Padrão, no contexto geral, pode ser considerado como um modelo a ser seguido, um exemplo a ser copiado. Os roteiristas de cinema, na elaboração de suas obras, costumam utilizar alguns padrões. Basicamente, o roteiro deve apresentar três elementos: a descrição das cenas, a sequência das ações e as falas dos personagens. Esses elementos ajudam a direcionar o caminho que o roteirista deve seguir até a obtenção do resultado final do trabalho. No universo de projetos isso também ocorre. Existem alguns modelos que ajudam a nortear o caminho de um desenvolvedor de software diante de determinados problemas conhecidos para os quais soluções de sucesso foram implementadas e compartilhadas. Assim como os roteiristas possuem elementos os quais empregam em suas produções, os desenvolvedores também podem seguir caminhos que podem os ajudar na construção de seus sistemas.

No universo do desenvolvimento de software, os padrões ganharam relevância após a publicação do livro *Design Patterns: Elements of Reusable Object-Oriented Software*, em 1995. Neste livro foram catalogados e descritos vinte e três padrões para o desenvolvimento de software orientado a objetos. Seus autores, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, ficaram conhecidos como a Gangue dos Quatro (*Gang of Four*), ou mesmo GoF. A partir da divulgação deste trabalho, despertou-se o interesse de estudiosos e profissionais de tecnologia pelo assunto, criando-se assim um vocabulário comum na temática de projetos de software.

A principal vantagem do uso de padrões de projeto está no reuso das soluções propostas para determinado problema. Segundo GAMMA [1], “padrões resolvem problemas específicos de projetos e tornam os projetos orientados a objetos mais flexíveis e, em última instância, reutilizáveis. Eles ajudam projetistas a reutilizar projetos bem-sucedidos ao basear os novos projetos nas experiências anteriores. Um projetista que está familiarizado com tais padrões pode aplicá-los imediatamente a problemas de projeto, sem necessidade de redescobri-los”.

Fique por dentro

Neste artigo mostraremos de forma detalhada dois dos vinte e três padrões de projeto catalogados pela Gangue dos Quatro: Singleton e Abstract Factory. Abordaremos cada um dos vários elementos que caracterizam um design pattern e, por fim, apresentaremos uma situação onde o uso do padrão seria aplicável. Sendo assim, o conteúdo aqui analisado é útil para desenvolvedores e arquitetos que precisam dominar soluções simples e consagradas com o objetivo de poder reutilizá-las em problemas específicos de projetos de software orientados a objetos.

Além disto, podemos destacar a facilidade que eles proporcionam na manutenção dos sistemas, já que um padrão representa uma unidade de conhecimento comum entre os envolvidos.

A utilização de alguns padrões, apesar de ser benéfica na maioria dos casos, requer cuidados, pois pode tornar o código maior e mais complexo. Portanto, é necessário conhecer bem os padrões de projeto e realmente entendê-los para identificar em quais situações é possível empregá-los de forma positiva.

Neste artigo abordaremos, na teoria e na prática, dois padrões de projeto catalogados pela Gangue dos Quatro: Singleton e Abstract Factory. Esses padrões se encaixam na categoria criação. O objetivo dos padrões deste grupo é criar objetos das formas mais adequadas possíveis. Além dos padrões criacionais, temos também os padrões estruturais, que abordam o modo pelo qual as classes devem se organizar, e os comportamentais, que tratam das interações e divisões de responsabilidades entre as classes.

Características de um padrão de projeto

Embora um padrão seja a descrição de um problema, de uma solução genérica e sua justificativa, isso não significa que qualquer solução conhecida para um problema possa constituir um padrão, pois existem características obrigatórias que devem ser atendidas pelos padrões [1].

Não pode ser considerado um padrão de projeto trechos de código específicos, mesmo que para o seu criador ele reflita um padrão que soluciona um determinado problema, porque os

padrões devem estar em um nível maior de abstração e não limitado a recursos de programação. Um padrão de projeto nomeia, abstrai e identifica os aspectos chaves de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizável [1].

Para ser considerado um padrão, alguns elementos são essenciais, como sugere GAMMA em sua obra. O **Nome**, por exemplo, é usado para descrever um problema, sua solução e suas consequências em uma ou duas palavras. A **Intenção** representa o propósito do padrão. Já a **Motivação** diz respeito a um cenário que contém o problema que o padrão irá resolver. A **Aplicabilidade**, por sua vez, descreve as situações nas quais o padrão pode ser empregado. A **Estrutura** mostra a representação gráfica da estrutura de classes do pattern. **Participantes** é o elemento responsável por dizer quais classes e objetos participam do padrão, além de mencionar as responsabilidades de cada um. **Usos conhecidos** demonstram casos de sistemas reais em que os padrões foram utilizados. **Implementação** descreve como o padrão deve ser implementado. **Exemplo de código** apresenta códigos que ilustram o pattern. Por fim, **Consequências** são os resultados e decisões que a utilização do padrão em questão acarreta.

Assim, buscamos neste artigo através dos elementos supracitados analisar os padrões Singleton e Abstract Factory, visando, de forma didática e fácil, levar ao leitor o melhor entendimento possível.

O padrão Singleton

O padrão Singleton é um dos design patterns mais simples do GoF e um dos padrões criacionais mais conhecidos e empregados em projetos.

Nome

O nome vem do significado em inglês e quer dizer quando se resta apenas uma carta nas mãos, num jogo de baralho. Como o objetivo do padrão é possibilitar que exista apenas uma instância de um objeto naquele escopo de execução do software, surge a analogia da instância única do objeto com a carta que resta na mão do jogador na disputa de baralho.

Intenção

Este padrão garante a existência de apenas uma instância de determinado objeto na memória, mantendo um ponto global de acesso a esse objeto no código.

Você pode estar pensando nesse momento: "Mas isso é fácil! Basta eu ter uma variável global com a instância do objeto e sempre acessá-la.". O problema é que essa variável sendo inicializada no começo da execução do projeto consumirá recursos mesmo sem ser aproveitada. O Singleton possibilita que a classe seja instanciada somente quando for necessária.

Esse padrão possui um método responsável por limitar o número de instâncias da classe [2]. Além disso, o Singleton define o construtor da classe como privado, ou seja, sua invocação é vedada a

qualquer outra classe, para garantir que somente a própria classe instancie um objeto de seu tipo.

Motivação

A motivação para a utilização desse design pattern está na necessidade de se ter exatamente apenas uma instância de uma determinada classe na aplicação. Por exemplo, embora possam existir muitas janelas em uma aplicação, deve existir somente um gerenciador de janelas. A necessidade de se ter apenas um gerenciador de janelas referenciado por uma única instância da classe que o representa justifica a utilização do padrão Singleton.

Estrutura

A representação gráfica do padrão pode ser visualizada na **Figura 1**. Para que exista uma instância única do objeto na aplicação, precisamos criar um construtor privado na classe Singleton, uma variável estática e implementarmos a lógica para criação da instância e o seu retorno.

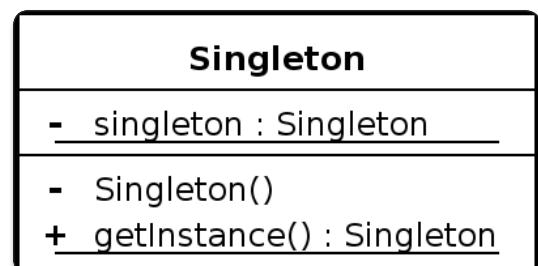


Figura 1. Estrutura do padrão Singleton

Participantes

Como mostra a **Figura 1**, o único participante do pattern é a classe **Singleton**. Ela disponibiliza a operação `getInstance()`, que permite aos clientes acessarem sua única instância.

Aplicabilidade

Deve-se utilizar o Singleton quando for necessário instanciar uma classe somente uma vez e que a instância criada seja acessada através de um ponto global [1]. Além disso, essa instância deve ter a capacidade de ser estendida, permitindo que os clientes usem a versão estendida sem modificar seus códigos-fonte. Uma situação onde podemos usar Singleton é quando temos que estabelecer conexão com algum banco de dados. Ao invés de abrirmos uma conexão com o banco a cada acesso, podemos utilizar uma variável global que armazena o objeto conexão. Assim, quando for necessário algum acesso ao banco, basta invocar a variável que armazena a conexão.

Consequências

Com a utilização do padrão Singleton não é necessário se preocupar com a possibilidade ou não da existência de uma instância da classe na qual o padrão está sendo aplicado, pois isso é controlado de dentro da própria classe [2].

Estudando os padrões Singleton e Abstract Factory

Outro efeito que a utilização do padrão proporciona é o espaço de nomes reduzido. O espaço de nomes é o local no código onde declaramos nossos identificadores (variáveis, funções, classes, estruturas, entre outros). Em vez de declarar variáveis globais que em muitos casos armazenam a mesma instância de um objeto, permite-se apenas um único nome externo visível, evitando a poluição do código.

Usos conhecidos

Um caso real em que o padrão Singleton pode ser empregado é o de uma aplicação que se utiliza de uma infraestrutura de log de dados. Nesse caso, é necessária a existência de apenas um objeto responsável pelo log em toda a aplicação e que seja acessível unicamente através da classe Singleton. Outra situação em que este padrão é útil é quando temos um objeto que armazena um pool de conexões, conforme veremos mais à frente.

Implementação e exemplo de código

A Listagem 1 mostra um exemplo de implementação em Java utilizando o padrão Singleton. Observe que na linha 3 da classe **ConnectionSingleton** é criado um objeto estático do tipo **ConnectionSingleton**. Este será a instância global de acesso. Na linha 6, o construtor encontra-se protegido e é nele que o algoritmo de conexão com o banco de dados é codificado.

Listagem 1. Implementação do Padrão Singleton.

```
01. public class ConnectionSingleton {  
02.  
03.     private static ConnectionSingleton instance = null;  
04.     private Connection connection;  
05.  
06.     private ConnectionSingleton() {  
07.         try {  
08.             this.connection = DriverManager.getConnection  
09.                 ("url", "user", "password");  
10.             System.out.println("Conexão realizada com sucesso.");  
11.         } catch (SQLException e) {  
12.             System.err.println(e.getMessage());  
13.         }  
14.  
15.         public static ConnectionSingleton getInstance(){  
16.             if (instance == null) { //se a instância não tiver sido criada, criá-la  
17.                 instance = new ConnectionSingleton();  
18.             }  
19.             return instance;  
20.         }  
21.  
22.         public Connection getConnection(){  
23.             return this.connection;  
24.         }  
25.     }
```

Note que o método **getInstance()**, iniciado na linha 15, verifica se foi atribuído algum valor ao atributo **instance**, que representa a instância global de acesso. Caso ainda não tenha sido atribuído, o método retorna um novo objeto **ConnectionSingleton**, e caso já tenha sido atribuído, retorna o objeto que já existe. Este método

deve, obrigatoriamente, ser estático, caso contrário não estaremos implementando um Singleton. Além disso, é ele que nos garante que quem está gerenciando a instância única é a própria classe.

Por fim, na linha 22, criamos o método **getConnection()** para obter a conexão. Este é um método muito simples, que apenas retorna a conexão criada no método construtor.

Apesar da simplicidade na implementação desse padrão, como podemos perceber no código utilizado como exemplo, devemos tomar alguns cuidados, pois em determinadas situações, como em ambientes multithreaded, podemos ter problemas. O Singleton não é uma solução thread-safe e pode não funcionar corretamente durante a execução simultânea de um mesmo trecho de código por várias threads. Imagine a situação em que uma thread chama o método **getInstance()** e, antes de realizar a instanciação, sofre interrupção e logo em seguida outra thread chama o método e realiza a instanciação. Neste caso, duas instâncias serão construídas. Isso fere a intenção do padrão. Para lidar com esse problema, precisamos sincronizar o método **getInstance()**, através da palavra-chave **synchronized**. Dessa forma, apenas uma thread por vez poderá acessar o método.

Considerações finais sobre o Singleton

Singleton é um padrão de projeto que tem como objetivo garantir que uma classe seja instanciada apenas uma vez. Trata-se de um padrão simples e que pode ser utilizado em situações em que normalmente temos componentes únicos do sistema como, por exemplo, o sistema de arquivo, a configuração de um software ou o gerenciador de janelas de um sistema operacional. Embora simples, cuidados devem ser tomados, pois Singleton não é thread-safe.

Abstract Factory

Para balancear um pouco, uma vez que no tópico anterior falamos sobre um padrão tido como mais simples, abordaremos agora um padrão um pouco mais complexo e que exigirá a análise de um número maior de conceitos e de linhas de código. Apresentaremos o Abstract Factory, ou Fábrica Abstrata.

Nome

O nome Fábrica Abstrata ocorre em virtude do cerne do padrão ser a existência de uma classe fábrica que é abstrata e responsável, através de operações, por encapsular a criação de objetos. As operações declaradas nessa classe são implementadas pelas Fábricas Concretas.

Portanto, o termo Fábrica se deve à existência de uma classe que propõe a criação de objetos e abstrata devido ao fato dessas operações serem implementadas por outras classes (subclasses).

Intenção

A intenção do Abstract Factory é fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas [1]. Assim, quando for necessário criar uma família de objetos, basta invocar a interface criada,

não sendo necessário conhecer os detalhes de implementação das classes concretas.

Motivação

A motivação para a utilização desse *Design Pattern* advém da necessidade de mudança dinâmica ou em tempo de execução da classe Factory. Esta classe tem a capacidade de alterar dinamicamente os objetos a serem usados na aplicação. Um exemplo disto ocorre na mudança de aparência e comportamentos de um Sistema Operacional.

Deste modo, o desenvolvedor, por exemplo, informa à classe Abstract Factory que precisa que seu programa seja parecido com o Linux, e ele recebe como retorno a fábrica responsável por objetos relativos à interface gráfica desse SO. Assim, toda vez que o usuário selecionar algum componente, será retornado esse componente para o Linux.

Estrutura

A Figura 2 apresenta uma representação gráfica desse padrão. A seguir teremos a oportunidade de entender o significado de cada um dos participantes do *Abstract Factory*.

Participantes

As classes e/ou objetos que participam do padrão, juntamente com as responsabilidades de cada um, são:

- **AbstractFactory:** declara uma interface para operações que criam produtos. Produto nesse caso é um objeto de uma subclasse determinada em tempo de execução;
- **ConcreteFactory:** implementa as operações para criar produtos. Para criar um produto, o cliente usa uma dessas fábricas;
- **AbstractProduct:** declara uma interface para uma família de produtos;
- **ConcreteProduct:** define o objeto produto a ser criado pela fábrica concreta correspondente e implementa a interface AbstractProduct;
- **Client:** utiliza as interfaces declaradas pelas classes AbstractFactory e AbstractProduct.

Na seção “Implementação e exemplo de código” teremos a oportunidade de ver o papel de cada um desses componentes considerando um exemplo real.

Aplicabilidade

O padrão Abstract Factory deve ser utilizado quando for necessário que um sistema funcione independentemente da forma como seus objetos são criados, compostos e representados. Dessa forma, disponibiliza-se uma biblioteca de classes de produtos, de

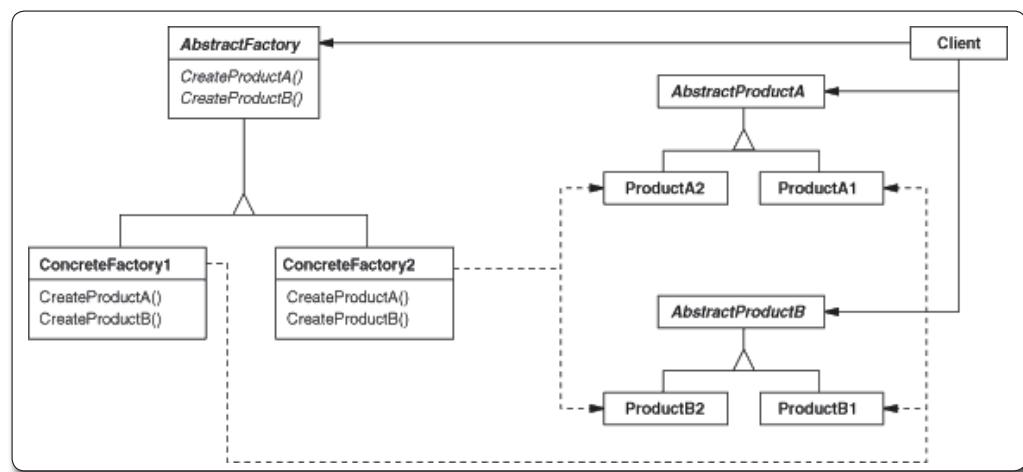


Figura 2. Estrutura do padrão Abstract Factory

acordo com as várias famílias de produtos existentes, revelando somente as suas interfaces, deixando detalhes referentes à criação dos objetos ocultos, não permitindo que objetos sejam diretamente criados com *new*.

Consequências

A utilização do padrão *Abstract Factory* permite isolar as classes concretas, pois os clientes manipulam as instâncias dessas classes

**Não perca tempo
reinventando a roda!**

COBREBEMX

**Componente completo para sua
Cobrança por Boleto Bancário
e Débito em Conta Corrente**

**Mais de 40 exemplos
em diversas linguagens
de programação**

**Geração e leitura de arquivos
(remessa e retorno) nos padrões
FEBRABAN e CNAB**

**Testes e Downloads
gratuitos em nosso site**

ACESSE E CONHEÇA O COMPONENTE EM:
WWW.COBBREM.COM

Estudando os padrões Singleton e Abstract Factory

apenas através de suas interfaces. Desse modo, facilita a permuta de famílias de produtos, pois a classe da fábrica concreta aparece apenas quando é invocada (instanciada), ou seja, em apenas um lugar da aplicação, o que elimina a complexidade ao se realizar mudanças nessas classes. Este padrão também promove a consistência entre produtos, garantindo que os objetos utilizados são todos de uma mesma família, representada pela fábrica concreta em questão.

Por fim, como um problema, este padrão dificulta a inserção de novos tipos de produtos, uma vez que a interface da fábrica abstrata torna fixo o conjunto de produtos que podem ser criados. Suportar um novo produto exige a extensão da interface da fábrica abstrata e a modificação de todas as suas subclasses.

Usos conhecidos

Um exemplo de uso clássico do Abstract Factory é o caso onde o sistema precisa suportar múltiplos tipos de interfaces gráficas, como Windows, Motif, Mac e Linux. O cliente informa ao factory que deseja que seu programa se pareça com o Windows e recebe como retorno a fábrica GUI com os objetos relativos ao Windows. Assim, quando você requisita um objeto específico como um botão, checkbox ou janela, a fábrica de GUI retorna as instâncias desses componentes para o Windows.

Outro uso conhecido se dá dentro da própria API do Java, com a biblioteca de Sockets do pacote **java.net**. Dentro desse pacote existe a classe abstrata **SocketFactory** que pode ser utilizada por outras fábricas de objetos para criar subclasses de Sockets que possuem regras específicas relacionadas.

Implementação e exemplo de código

Para exemplificar o padrão de projeto Abstract Factory, vamos imaginar que estamos construindo um sistema para uma loja de veículos e que em determinado momento precisamos de um comportamento específico para carros (produtos) com características semelhantes (família de produtos). As montadoras classificam os carros em várias “famílias”, isto é, categorias como Hatch, Sedan, Minivan, Perua, Picape, Utilitários, etc., sendo que cada uma possui características próprias. No nosso caso, como exemplo e visando simplificar a codificação para focar apenas no padrão de projeto, agruparemos os carros em: **Sedan** e **Hatch**.

Dito isso, começaremos nossa implementação pelo componente Abstract Factory, que será representado pela interface **FabricaDeCarro**. Esta interface declara as operações responsáveis pela criação dos produtos (carros) pelas fábricas concretas. Perceba, na **Listagem 2**, que temos os métodos **criarCarroSedan()** e **criarCarroHacth()** que criam instâncias dos produtos carro Sedan e carro Hatch, respectivamente.

Na **Listagem 3** temos a classe (fábrica) concreta **FabricaFiat** que utiliza a interface criada como fábrica abstrata. Nessa classe serão implementadas as operações para criar os carros da montadora Fiat. Já na **Listagem 4** é possível visualizar outra fábrica, que codifica os mesmos métodos, mas referentes a carros da Ford.

Listagem 2. Implementação da fábrica abstrata.

```
01. public interface FabricaDeCarro {  
02.     CarroSedan criarCarroSedan();  
03.     CarroHatch criarCarroHatch();  
04. }
```

Listagem 3. Código da fábrica concreta referente à montadora Fiat.

```
01. public class FabricaFiat implements FabricaDeCarro {  
02.  
03.     @Override  
04.     public CarroSedan criarCarroSedan() {  
05.         return new Siena();  
06.     }  
07.  
08.     @Override  
09.     public CarroHatch criarCarroHatch() {  
10.         return new Palio();  
11.     }  
12. }
```

Listagem 4. Código fábrica concreta referente à montadora Ford.

```
01. public class FabricaFord implements FabricaDeCarro {  
02.  
03.     @Override  
04.     public CarroSedan criarCarroSedan() {  
05.         return new Focus();  
06.     }  
07.  
08.     @Override  
09.     public CarroHatch criarCarroHatch() {  
10.         return new Fiesta();  
11.     }  
12. }
```

Se for necessário adicionar mais montadoras ao nosso sistema, basta que suas fábricas concretas implementem a interface **FabricaDeCarro**. Por exemplo, para trabalharmos com carros da Chevrolet, basta codificar a fábrica referente a essa montadora.

Nesse ponto, veremos como criar os produtos. No nosso exemplo, os produtos foram divididos em dois grupos: **Hatch** e **Sedan**, cada um com suas características. Para o nosso exemplo, vamos implementar um método para exibir informações de um carro **Sedan** e outro para exibir informações de um carro **Hatch**. Assim, simularemos através desses métodos um comportamento para cada família de carros e manteremos suas implementações ocultas do cliente. A interface para o tipo de carro Hatch fica conforme a **Listagem 5** e a interface para o tipo de carro Sedan fica conforme **Listagem 6**. Nessas interfaces podemos inserir operações particulares de acordo com o tipo do carro.

Os métodos **exibirInfoHatch()** e **exibirInfoSedan()** apresentam algumas informações do veículo como potência e capacidade do tanque de combustível.

A implementação dos produtos pode ser visualizada através das **Listagens 7** e **8**.

Com isso finalizamos a definição dos componentes do padrão. Na **Listagem 9** podemos visualizar o exemplo de uma classe cliente que usa a abstract factory **FabricaDeCarro** para criar uma referência para uma família de produtos. A partir da fábrica abs-

VOCÊ TAMBÉM
ESTÁ BOIANDO
NESTA SOPA?



JPA TDD JSON
HTML5 MADM

Estudando os padrões Singleton e Abstract Factory

trata, derivamos uma ou mais fábricas concretas que produzem carros de famílias diferentes. Dessa forma o cliente é desvinculado de qualquer especificação dos produtos concretos, pois toda a responsabilidade por instanciar carros foi passada para os métodos `criarCarroSedan()` e `criarCarroHatch()`.

Listagem 5. Código da interface CarroHatch.

```
1. public interface CarroHatch {  
2.     void exibirInfoHatch();  
3. }
```

Listagem 6. Código da interface CarroSedan.

```
1. public interface CarroSedan {  
2.     void exibirInfoSedan();  
3. }
```

Listagem 7. Código do produto concreto Palio.

```
1. public class Palio implements CarroHatch {  
2.  
3.     @Override  
4.     public void exibirInfoHatch() {  
5.         System.out.println("80 cavalos de potência e capacidade de 50 litros de  
combustível no tanque.");  
6.     }  
7. }
```

Listagem 8. Código do produto concreto Siena.

```
1. public class Siena implements CarroSedan {  
2.  
3.     @Override  
4.     public void exibirInfoSedan() {  
5.         System.out.println("78 cavalos de potência e capacidade de 45 litros de  
combustível no tanque.");  
6.     }  
7. }
```

Listagem 9. Código Cliente que consome a fábrica criada.

```
01 public static void main(String[] args) {  
02.  
03.     FabricaDeCarro fabrica = new FabricaFiat();  
04.     CarroSedan sedan = fabrica.criarCarroSedan();  
05.     CarroHatch hatch = fabrica.criarCarroHatch();  
06.     sedan.exibirInfoSedan();  
07.     System.out.println();  
08.     hatch.exibirInfoHatch();  
09.     System.out.println();  
10.  
11.     fabrica = new FabricaFord();  
12.     sedan = fabrica.criarCarroSedan();  
13.     hatch = fabrica.criarCarroHatch();  
14.     sedan.exibirInfoSedan();  
15.     System.out.println();  
16.     hatch.exibirInfoHatch();  
17. }
```

Como pode ser observado, na linha 3 criamos uma referência para uma fábrica abstrata e associamos a classe concreta referente à montadora Fiat. Na linha 4 criamos um objeto que representa um carro Sedan e na linha 5 um objeto para um carro Hatch.

Perceba que na linha 11 a fábrica abstrata passa a apontar para outra família de produtos, que no caso refere-se a carros da Ford. Na linha 12 foi criado um carro Sedan e na linha 13 um carro Hatch e nas linhas 14 e 16 foram invocados os métodos que exibem informações desses carros.

Como nosso código fica desvinculado do processo de criação dos produtos reais, pode-se criar ou modificar fábricas para obter comportamentos diferentes, como ocorreu na linha 11 substituindo a fábrica concreta Fiat pela fábrica concreta Ford, para obter os comportamentos da família de produtos que desejamos. Isso nos permite implementar uma variedade de fábricas que criam produtos para contextos diferentes.

Considerações finais sobre o Abstract Factory

Em resumo, o padrão Abstract Factory permite construir famílias de objetos relacionados, sendo um dos design patterns mais utilizados por arquitetos e desenvolvedores. Porém, precisa ser utilizado com cautela, pois como vimos, a inclusão de novos objetos acarreta na criação de diferentes fábricas e, consequentemente, o crescimento do número de classes no projeto.

O uso dos padrões analisados proporciona a construção de softwares orientados a objetos e estruturas de código de forma flexível e reutilizável. Ao invés de perder horas tentando reinventar a roda, o desenvolvedor pode utilizar soluções já consagradas em cenários específicos como os mencionados nesse artigo. Com isso, ganha-se em coesão, reusabilidade, tempo e facilidade no processo de manutenção.

Autor



Carlos Alberto Silva

casilvamg@hotmail.com

É Formado em Ciéncia da Computação pela Universidade Federal de Uberlândia (UFU) com especialização em Desenvolvimento Java pelo Centro Universitário do Triângulo (UNITRI). Trabalha na empresa Algar Telecom como Analista de Sistemas e atualmente é aluno do curso de especialização em Análise e Desenvolvimento de Sistemas Aplicados a Gestão Empresarial no Instituto Federal do Triângulo Mineiro (IFTM). Possui as seguintes certificações: OCJP, OCWCD e ITIL.



Links:

[1] GAMMA, Erich, HELM, Richard; JOHNSON, Ralph, VLISSIDES, John. "Padrões de Projeto: soluções reutilizáveis de software orientado a objetos". 1. ed. Porto Alegre: Bookman, 2000.

[2] SHALLOWAY, Alan; Trott, Richard. "Explicando Padrões de Projeto – Uma nova perspectiva em projeto orientado a objeto". 1. ed. Porto Alegre: Bookman, 2004.

[3] BRIZENO, Marcos. "Mão na massa: Abstract Factory".

<http://brizeno.wordpress.com/category/padroes-de-projeto/abstract-factory/>

ASSIM, O MERCADO TE ENGOLE!



PARAR DE SE ATUALIZAR PODE SER FATAL!

O mercado muda a todo momento e você precisa estar atento sempre. Seja um assinante MVP e devore tudo o que há de mais relevante no mercado de TI.

Invista na sua carreira!

TENHA ACESSO A:



+DE 290 CURSOS ONLINE



09 REVISTAS MENSais



9.074 VÍDEO-AULAS

POR APENAS **69,90** MENSais

QUEM TEM ESTÁ TRANQUILO.



DEVMEDIA

Acesse: www.devmedia.com.br/mvp

Java Server Faces (JSF): Desenvolvimento de aplicações web

Um pequeno guia para quem está começando no mundo Web

Aplicações web são hoje o principal tipo de aplicação desenvolvida por empresas de TI. Isso faz com que a cada dia surjam soluções que tentam ajudar os desenvolvedores no árduo trabalho da criação de sistemas. Com a concorrência do mercado de desenvolvimento de software, há diversas opções de frameworks Java para atender aos anseios dos desenvolvedores.

Como vimos no artigo publicado na Easy Java Magazine 36, de título “Introdução ao desenvolvimento de aplicações web”, existem frameworks baseados em Ações, ou Action Based, e frameworks baseados em Componentes, ou Component Based. Atualmente, os frameworks baseados em Componentes vêm se destacando no mercado pela facilidade de criarmos aplicações web com recursos visuais complexos. Assim, podemos nos preocupar apenas com qual componente devemos utilizar em uma determinada página como, por exemplo, um componente de calendário, e não mais em como iremos ter que programar tal componente usando HTML ou JavaScript.

A especificação do JSF, acrônimo de JavaServer Faces, atualmente na versão 2.2, definido na especificação **JSR-344**, veio para padronizar o uso de componentes ricos pelas aplicações, além de se tornar uma referência às empresas com garantia de extensibilidade de seus componentes, ou seja, é possível a criação de novos componentes que se encaixam nas necessidades de sua aplicação. Essa possibilidade de customização de componentes é um recurso poderoso de extensão para atender às demandas específicas de reuso presentes nas aplicações web atuais.

Nesse contexto do uso de componentes, iremos visualizar o que muda quando utilizamos Servlets e JSP na criação de aplicações Web adotando este novo paradigma de programação orientada ao uso de componentes.

Fique por dentro

Este artigo continuará a abordar o tema de introdução ao desenvolvimento de aplicações com foco na web, que foi tratado na edição 36 da revista Easy Java Magazine, onde discutimos o início do desenvolvimento de aplicações web utilizando Servlets e JSP. Ao longo do artigo serão apresentadas conceituações técnicas sobre as tecnologias necessárias à criação de um sistema web utilizando JSF, visando descomplicar a vida de quem está entrando neste mercado.

Protocolo HTTP: Stateless

O protocolo Hypertext Transfer Protocol (HTTP) é o método utilizado para enviar e receber informações na web. Está atualmente na versão 1.1, sendo definido pela especificação **RFC 2616**. Esta especificação é leitura obrigatória para todos os desenvolvedores que programam aplicações Web.

O protocolo HTTP é baseado em requisições e respostas entre clientes e servidores. O cliente — navegador ou dispositivo que fará a requisição; também é conhecido como user agent — solicita um determinado recurso (resource), enviando um pacote de informações contendo alguns cabeçalhos (headers) a uma URI ou, mais especificamente, URL, como vimos em nosso artigo anterior. O servidor recebe estas informações e envia uma resposta, que pode ser um recurso ou simplesmente outro cabeçalho.

Na **Listagem 1** temos um exemplo de como seria um cabeçalho enviado em uma requisição. É possível visualizar os cabeçalhos de uma requisição utilizando programas voltados para isto, que são conhecidos como sniffers. Um exemplo gratuito e muito utilizado seria o aplicativo **WireShark**.

Segundo este cabeçalho, estamos enviando algumas informações que identificam nosso cliente no caminho (path) e, o mais importante, qual o método da requisição. O servidor, por sua vez, identifica os cabeçalhos que lhe são convenientes e envia uma resposta.

Nos cabeçalhos de resposta, como mostrado na **Listagem 2**, você pode obter algumas informações muito importantes, dentre elas o código de resposta (Status). Este código identifica se uma requisição foi concluída com sucesso (200) ou se ela não existe (404), por exemplo.

Listagem 1. Exemplo de cabeçalho HTTP.

```
GET /HTTP/1.1
Host: google.com.br
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US) Gecko/20061201 Firefox/2.0.0.3
(Ubuntu-feisty)
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,image/png
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

Listagem 2. Exemplo de resposta de uma requisição HTTP.

```
HTTP/1.1 200 OK
Date: Mon, 23 Jun 2014 22:38:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
```

Outra informação importante sobre o protocolo HTTP é sua característica stateless – ele não é capaz por si só de reter informações entre requisições diferentes. Para persistir informações você precisa utilizar cookies, sessões, campos de formulário ou variáveis informadas na própria URL.

Assim, quando desenvolvemos aplicações web, precisamos superar o problema de armazenamento de informações necessárias aos próximos passos que o usuário dará dentro do contexto de nossa aplicação. Mas, como assim? Imagine uma aplicação web simples, onde você gostaria que o nome do usuário, que foi capturado do banco de dados no momento do login do usuário, estivesse sempre presente em todas as páginas que ele acessasse, sem a necessidade de efetuar uma nova consulta no banco de dados.

Pensando assim, poderíamos armazenar essas informações em algum lugar onde pudéssemos consultá-las sempre que necessário, e que estas não fossem perdidas a cada nova requisição do usuário. Para isto, poderíamos armazená-las então em uma sessão ou cookie, que será criado para o usuário, e que é a ligação dele com o servidor para saber que o mesmo cliente que iniciou o acesso à aplicação, está agora percorrendo as diferentes páginas da aplicação.

O JSF foi criado para tratar além de outros problemas, essa característica stateless do protocolo HTTP, provendo uma maneira de desenvolver aplicações web de forma stateful, com seus componentes se encarregando de armazenar o estado atual do contexto do usuário no momento em que este solicita uma requisição.

JSP x Facelets

Relembrando o artigo da edição 36, a tecnologia JSP, acrônimo de JavaServer Pages, é uma linguagem de script baseada em uma especificação Java, a JSR-245, que está atualmente na versão 2.1 e é utilizada para a criação de páginas com conteúdo estático ou dinâmico.

Quando o JSF foi criado, a intenção era utilizar o JSP como a principal tecnologia para o layout de páginas, uma vez que este já era o padrão utilizado na comunidade web.

A ideia foi juntar o útil ao agradável, simplificando a aprovação do JSF usando uma linguagem de tags familiar que já possuía uma grande aceitação em meio à comunidade Java.

Porém, infelizmente, JSP e JSF não eram naturalmente complementares, uma vez que o JSP é usado para criar conteúdo estático ou dinâmico na web, mas não para a criação de componentes, organizados em árvores. Os elementos de uma página JSP são processados de cima para baixo, ou seja, do início da página para o final, de forma estruturada, com o objetivo fundamental de criar uma resposta a uma requisição. Já no JSF deve-se ter um ciclo de vida mais complexo em que a geração do componente e a sua renderização precisa acontecer em fases claramente separadas.

Assim surgiu o Facelets, inicialmente como uma alternativa ao JSP, visando facilitar a construção de interfaces ao JSF, tornando-se o padrão da tecnologia JSF logo depois. Os principais motivos para utilização de Facelets à JSP são:

- Utilização adaptável a qualquer implementação e versão do JSF;
- Independência de container web, podendo ser utilizada sem a necessidade de usar o Java EE ou um container que já tenha o JSP;
- Utilização de templates, sendo possível reutilizar o código para simplificar o desenvolvimento e manutenção de aplicações de grande escala;
- Criação de componentes leves, que são fáceis de desenvolver, em comparação com os componentes JSF puros;
- Suporte a Unified Expression Language (EL), incluindo o suporte para funções EL e validação EL em tempo de compilação. A EL tem as características da JSP EL e acrescenta mais capacidades, tais como avaliação de expressões, JSTL iteration tags e invocação de métodos dentro de uma expressão;
- Relatório preciso de erros, mostrando informações detalhadas sobre a linha onde ocorre a exceção;
- Utilização do atributo `jsfc` (que é o equivalente ao `jwcid` do Tapestry), para fornecer integração com editores HTML existentes.

Com o uso de Facelets, é possível utilizarmos os componentes do JSF respeitando o ciclo de vida desta tecnologia.

Modelo, Ciclo de Vida e Árvore de Componentes

O JSF utiliza a arquitetura MVC, porém com uma organização um pouco diferente. Ele trabalha com a classe `javax.faces.webapp.FacesServlet`, que exerce o papel de Controller da arquitetura MVC, controla as requisições, roteando o tráfego e administrando o ciclo de vida dos beans e componentes de interface do usuário (UI). Assim, como podemos visualizar na

Java Server Faces (JSF): Desenvolvimento de aplicações web

Figura 1, a FacesServlet recebe a requisição originada no navegador e chama a View, representada por nossas páginas XHTML. Conforme nossa View é processada, esta efetua a chamada do que for necessário aos Managed Beans, através de **bindings** (será explicado posteriormente) entre componentes e beans. Os Managed Beans se encarregam de acessar as classes do Modelo, que são o domínio da aplicação.

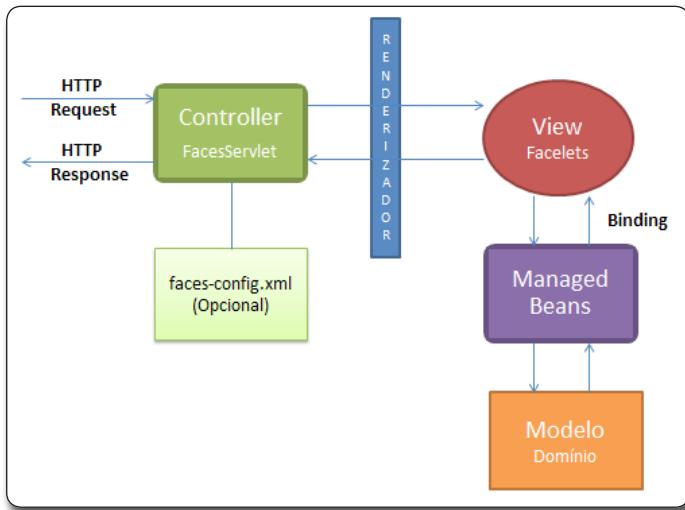


Figura 1. Arquitetura do JSF

O JSF também possui um Ciclo de Vida particular, que podemos visualizar na **Figura 2**. Assim, é de suma importância entender este ciclo para sabermos exatamente em que fase da renderização o JSF se encontra. Ao final deste artigo, quando efetuarmos a criação de nossa aplicação, ficará mais fácil visualizar este ciclo e entender suas etapas.

A seguir explicamos cada um dos itens do ciclo:

- **Restauração da View (Restore View)**: esta fase inicia o processamento da requisição do ciclo de vida por meio da construção da árvore de componentes do JSF, ou recuperação do estado da árvore anterior;
- **Aplicar os Valores da Requisição (Apply Requests)**: nesta fase, o JSF recupera os valores que vieram da requisição, e os aplica nos componentes apropriados. No entanto, os valores ainda são representados como **String**, não possuindo a representatividade do modelo de domínio;
- **Converter e Processar Validações (Process Validations)**: depois de recuperar os valores e associá-los ao componente correto, o JSF converte estes valores para o tipo adequado que cada componente espera receber. Após esta conversão, o valor já estará com o tipo correto e, portanto, estará pronto para ser validado. Um componente que necessita de validação deve fornecer a implementação da lógica de validação;
- **Atualização dos Valores dos Modelos (Update Model Values)**: nesta fase, após todos os componentes serem validados, os valores são aplicados aos seus respectivos modelos. Desta forma, as propriedades do Managed Bean serão populadas de acordo com

as informações que vieram na requisição. O JSF sabe para quais objetos enviar os valores através do binding que foi realizado na criação das telas, fazendo internamente a criação e população do objeto;

- **Invocação da Lógica (Invoke Application)**: com o Managed Bean populado na fase anterior, o JSF agora pode, com segurança, efetuar a invocação da lógica determinada pela requisição. Durante esta fase, a implementação JSF manipula quaisquer eventos do aplicativo, tal como enviar um formulário ou ir a outra página através de um link. Estes eventos são ações que retornam geralmente uma string que está associada a uma navegação ao qual se encarrega de chamar a página determinada por esta string;
- **Renderizar Resposta (Render Response)**: esta é a fase final, na qual é renderizada a página XHTML ao usuário. Se este é um pedido inicial para esta página, ou seja, primeira requisição, os componentes são acrescentados à apresentação neste momento. Se este é um postback (reenvio de dados de uma página para ela mesma), os componentes já foram acrescidos à apresentação, assim não precisam ser acrescidos novamente. Se há mensagens de conversão ou erros de validação e a página contém um ou mais destes componentes, estes serão exibidos.

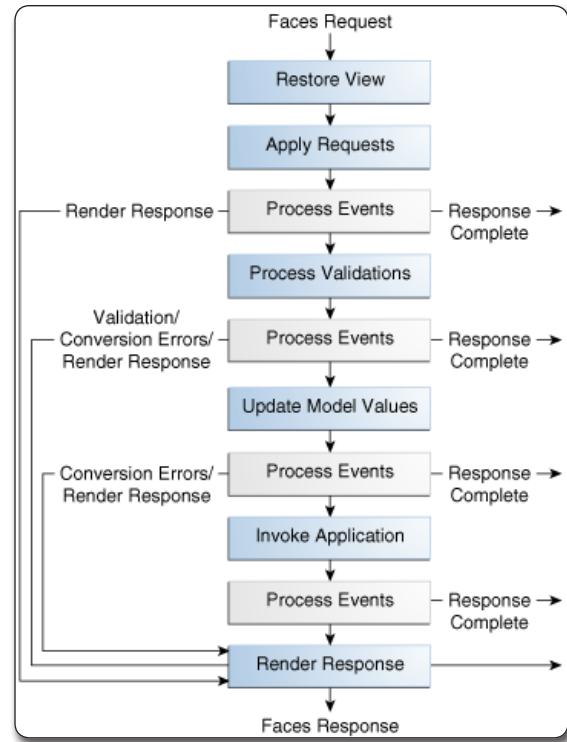


Figura 2. Ciclo de Vida do JSF

Como o JSF é todo baseado no uso de componentes, precisamos utilizá-los para manter o estado de nossas telas. Para isto, o JSF trabalha com o que chamamos de Árvore de Componentes (View ou UIViewRoot), que fica armazenada na sessão do usuário de nossa aplicação. Assim, quando criamos um simples formulário utilizando Facelets e JSF, como o mostrado na **Listagem 3**,

ao ser recebida a requisição pelo JSF, as tags **h:form**, **h:inputText** e **h:commandButton** são instanciadas como componentes, neste caso um **UIForm**, que seria o formulário, **UIInput**, que seria o nosso **inputText**, e por fim o **UIComand**, que seria o nosso button. É utilizada a nomenclatura de Árvore de Componentes, pois estes componentes são organizados como uma árvore, respeitando uma hierarquia, como mostrado na **Figura 3**.

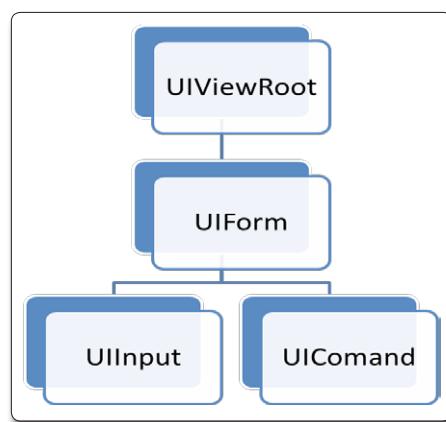


Figura 3. Árvore de componentes do formulário

Então cada vez que o controlador JSF recebe uma nova requisição para uma página, é criada uma árvore de componentes nova para esta página. Para distinção entre as árvores diferentes, o JSF renderiza no HTML sempre um campo a mais que representa a identificação da árvore dentro da sessão HTTP. Podemos ver um exemplo desta renderização na **Listagem 4**.

Verificamos no código que a variável **javax.faces.ViewState** armazena a identificação da árvore no lado do cliente, e a cada nova requisição será criada uma nova identificação, ou seja, todas as páginas terão uma ou mais árvores de componentes na sessão do usuário. Essa quantidade de árvores é limitada, a depender da implementação do JSF. A implementação de referência, o Mojarra, limita esta quantidade a 15 árvores, no máximo, na sessão do usuário. Assim, se o usuário tiver mais do que 15 abas abertas em seu navegador, e uma de suas abas for uma aplicação JSF, por exemplo, ao abrir a 16ª o controlador do JSF irá automaticamente tirar a árvore que

estiver sendo menos utilizada da sessão. Este cálculo de utilização é realizado por algoritmo (LRU) a fim de evitar perda de desempenho.

É possível também não utilizarmos a sessão para armazenar nossa árvore de componentes, através da configuração de um atributo no *web.xml* de nossa aplicação, como mostra a **Listagem 5**. Com este comportamento setado para **client**, o valor armazenado no campo **javax.faces.ViewState** será toda a árvore serializada, ou seja, toda a árvore de componentes serializada será armazenada no atributo **value**, algo como **value="/0RL6JBt8cvqShFdUYKZJMFYhH3aK5oYI4toUZvNpTR+xCEIE1Uu9gLy21nZ6Z9tJC3z6WQ4pGdWJuwOkbTP+Q=="**.

Configurando o ambiente e criando o projeto

Para demonstrar a criação de aplicações com JSF e o uso de seus componentes, iremos utilizar a IDE Eclipse Kepler e o servidor de aplicações web JBoss AS 7.1.1. Os endereços para download dessas ferramentas estão na seção **Links**.

Após o download e instalação das ferramentas, primeiramente iremos adicionar o contêiner web em nossa IDE. Para isto, na aba **Servers**, clique sobre a opção *No servers are available. Click this link to create a new server...*, como mostra a **Figura 4**. Ao clicar nesta opção será apresentada uma tela para escolha do servidor de aplicação web, como podemos visualizar na **Figura 5**.

Listagem 3. Código fonte de um formulário simples.

```

<h:form>
  <h:inputText value="#{formularioBean.valor}" id="valor"/>
  <h:commandButton value="Enviar" action="#{formularioBean.mostra}" />
</h:form>
  
```

Listagem 4. Identificação da árvore de componentes.

```
<input id="javax.faces.ViewState" value="-5331242445646946223:4134580279656748833" type="hidden"/>
```

Listagem 5. Configuração do atributo **STATE_SAVING_METHOD**.

```

<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
  
```

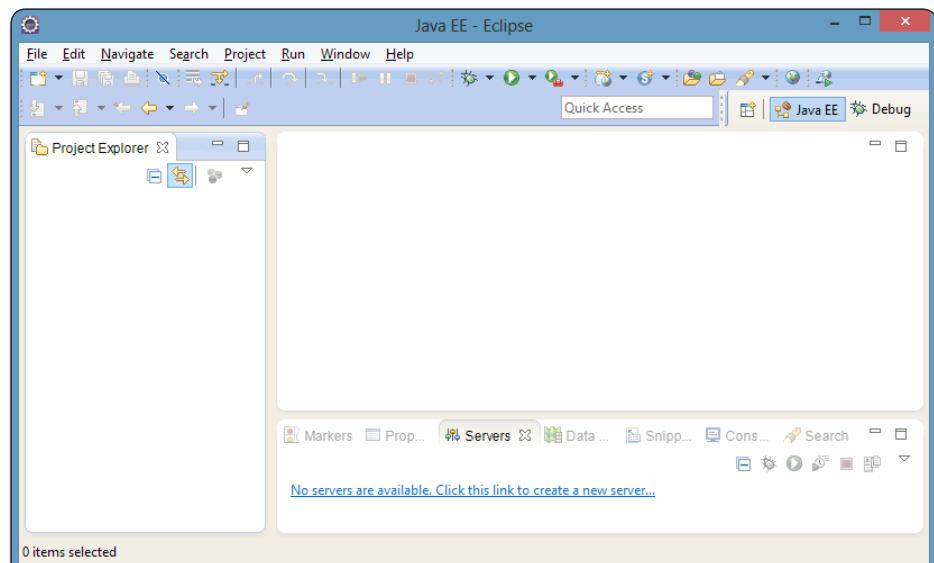


Figura 4. Inclusão do contêiner web na IDE Eclipse

Java Server Faces (JSF): Desenvolvimento de aplicações web

Porém, a versão 7.1 do JBoss não se encontra nesta listagem, sendo necessária a instalação do Plugin JBoss AS Tools. Para isto, clicamos na opção *Download additional server adapters*, como é mostrado na **Figura 5**, e selecionamos a opção JBoss AS Tools na próxima tela.

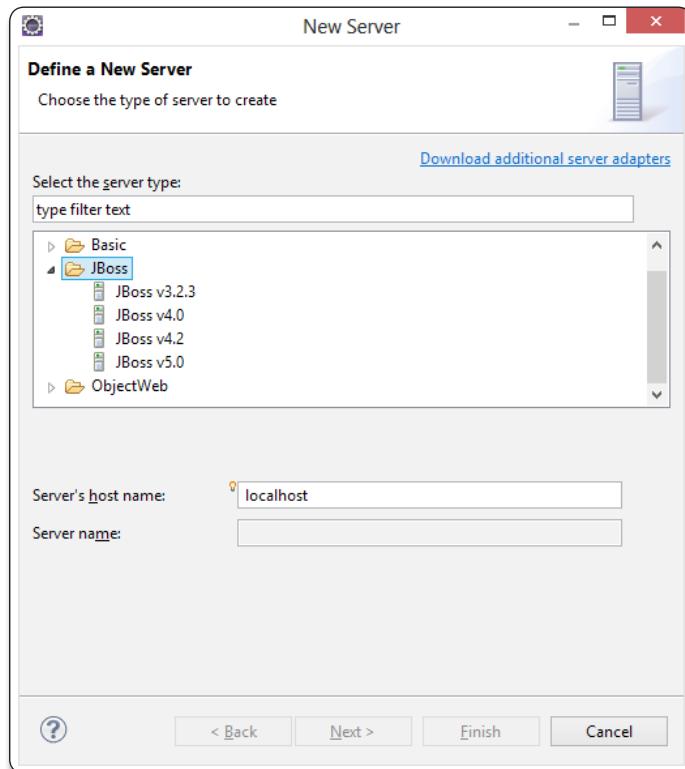


Figura 5. Acrescentando o adaptador para trabalharmos com o servidor JBoss AS 7

Em seguida, aceitamos os termos de instalação do plugin, como mostrado na **Figura 6**. Ao final da instalação será solicitado o reinício da IDE. Depois que esta tiver sido reiniciada, novamente na opção *No servers are available. Click this link to create a new server...*, como mostrado na **Figura 4**, e a opção JBoss Community terá sido acrescentada nas opções de escolha do servidor. Abaixo desta se encontra o servidor JBoss AS 7.1, que deverá ser selecionado, como mostrado na **Figura 7**. Logo após, clique no botão *Next* para partirmos para a próxima tela, onde a localização da instalação do servidor de aplicação web precisa ser informada (ver **Figura 8**). Após esta escolha, clique na opção *Finish*.

Com o servidor devidamente instalado em nossa IDE, iremos efetuar a criação de nosso projeto web. Para isto, acesse a opção *File > New > Dynamic Web Project*. Para a criação do projeto, precisamos definir o seu nome e escolher o servidor de aplicação web onde ele será instalado. Também é preciso verificar a versão da especificação de servlets que usaremos, neste caso a 3.0, e além disto, devemos selecionar a configuração inicial de nosso projeto, onde iremos optar pelo JavaServer Faces v2.0 Project, que já irá criar um projeto na estrutura utilizada pelo JSF, como pode ser visualizado na **Figura 9**. Depois disto, clicamos em *Next*.

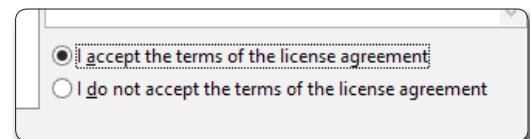


Figura 6. Seleção da opção de aceite dos termos do plugin no Eclipse

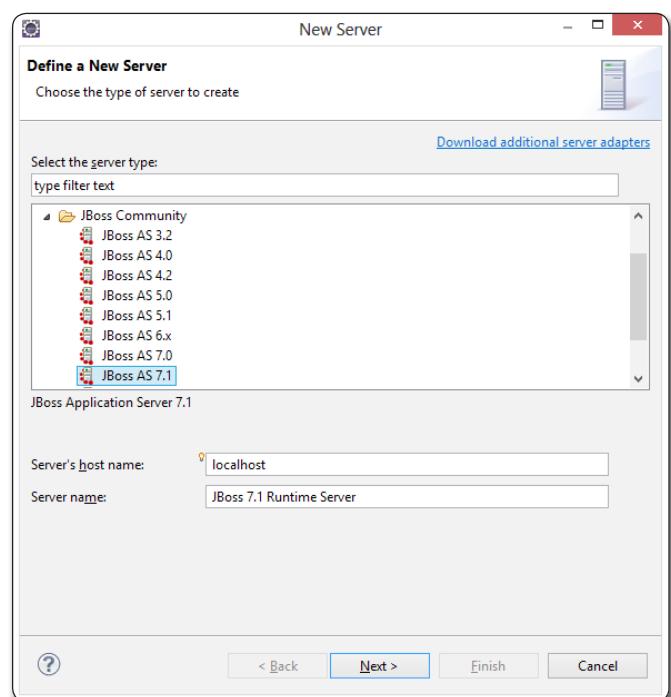


Figura 7. Seleção do servidor de aplicação web na IDE Eclipse

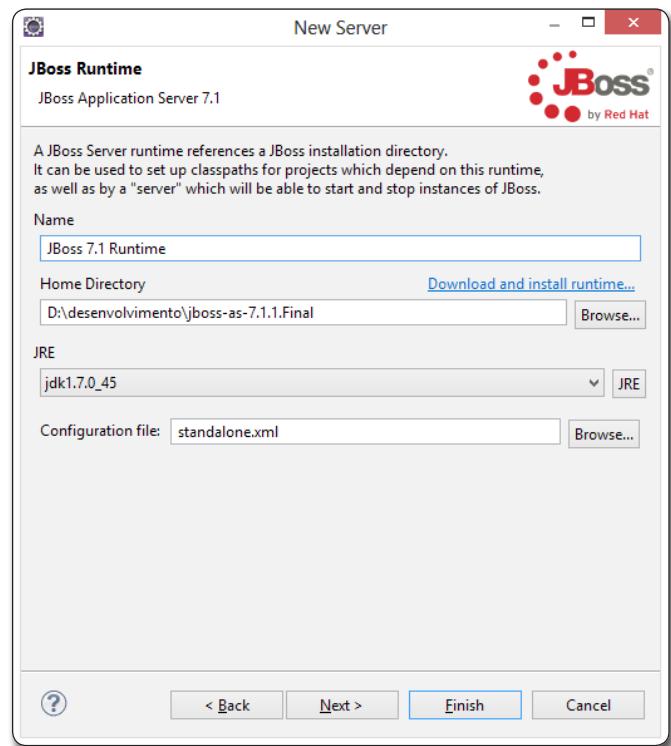


Figura 8. Informação da localização da instalação do servidor de aplicação web na IDE Eclipse

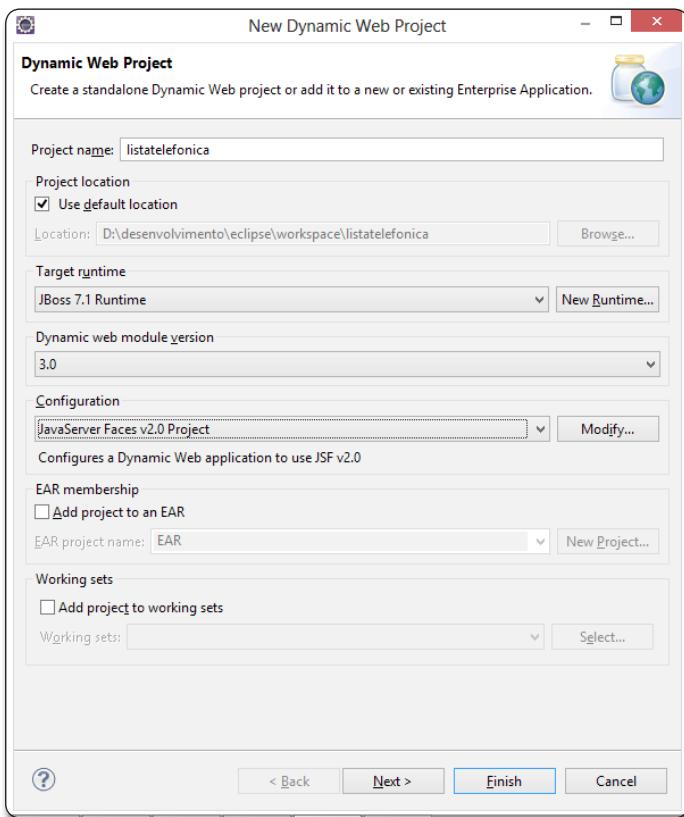


Figura 9. Criação de projeto Web na IDE Eclipse

Na próxima tela iremos configurar onde nossas classes Java serão armazenadas no projeto. Após isto, clicamos novamente em *Next*, e marcamos a opção *Generate web.xml deployment descriptor* para que o arquivo *web.xml* seja gerado automaticamente pela IDE em nosso projeto, como mostrado na **Figura 10**, e clicamos na opção *Next*.

Por fim, na próxima tela, que é responsável pela configuração do JSF no projeto, devemos selecionar a opção *Library Provided by Target Runtime*, onde estamos dizendo à IDE que iremos utilizar as bibliotecas do JSF nativas de nosso servidor de aplicações e, além disto, configuramos o restante das informações, como mostrado na **Figura 11**. Após estas configurações, clique na opção *Finish*. Pronto, nosso projeto está criado.

Com o projeto devidamente criado, vamos dar início à nossa aplicação criando uma lista telefônica. Para isto, acesse a opção *File > New > Other* na IDE. Será exibida uma lista, onde devemos selecionar na opção *Web*, o item *HTML File*, clicando na opção *Next* para prosseguirmos. Na próxima tela, iremos nomear o arquivo como *index.xhtml*, lembrando-se de alterar sua extensão para XHTML, como mostrado na **Figura 12**, e clicando em *Next* em seguida. Na próxima tela, devemos selecionar a opção *Use HTML Template*, e escolher a opção *New Facelet Composition Page*, de acordo com a **Figura 13**. Após isto, clique na opção *Finish*. Repita esta ação, criando outras duas páginas, a *pesquisar.xhtml* e a *cadastrar.xhtml*. O conteúdo de cada página é exibido nas **Listagens 6, 7 e 8**.

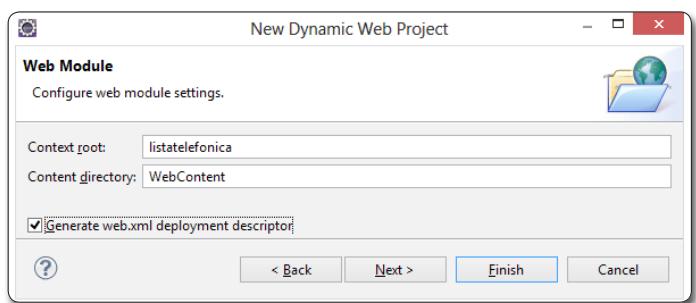


Figura 10. Seleção da geração automática do arquivo *web.xml*

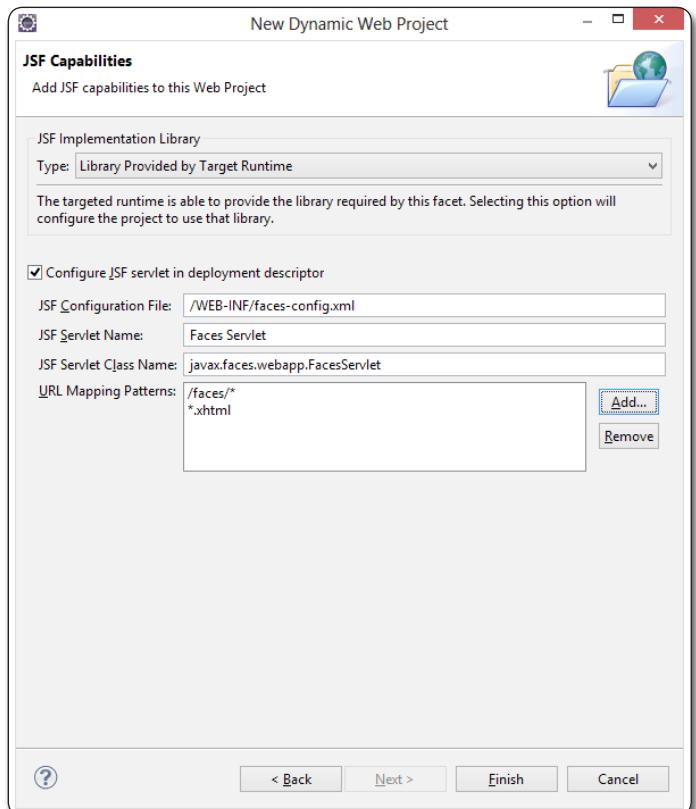


Figura 11. Configuração do JSF no projeto web na IDE Eclipse

Temos na **Listagem 6** nossa página principal, onde é exibido um pequeno menu para as opções de Cadastrar Contato e de Pesquisar Contatos. Utilizamos o componente *h:outputLink* para a criação dos links de nosso menu. Nas **Listagens 7 e 8** utilizamos o componente *h:outputText* apenas para exibirmos uma mensagem de acordo com a função de nossa página, Cadastrar e Pesquisar, apenas para esta parte inicial. Após criarmos nossas páginas, iremos acrescentar nossa página inicial *index.xhtml* como a página de boas vindas de nossa aplicação. Para isto, abra o arquivo *web.xml*, localizado na pasta *WebContent > WEB-INF* e acrescente a tag *<welcome-file>index.xhtml</welcome-file>*, conforme mostrado na **Listagem 9**.

Java Server Faces (JSF): Desenvolvimento de aplicações web

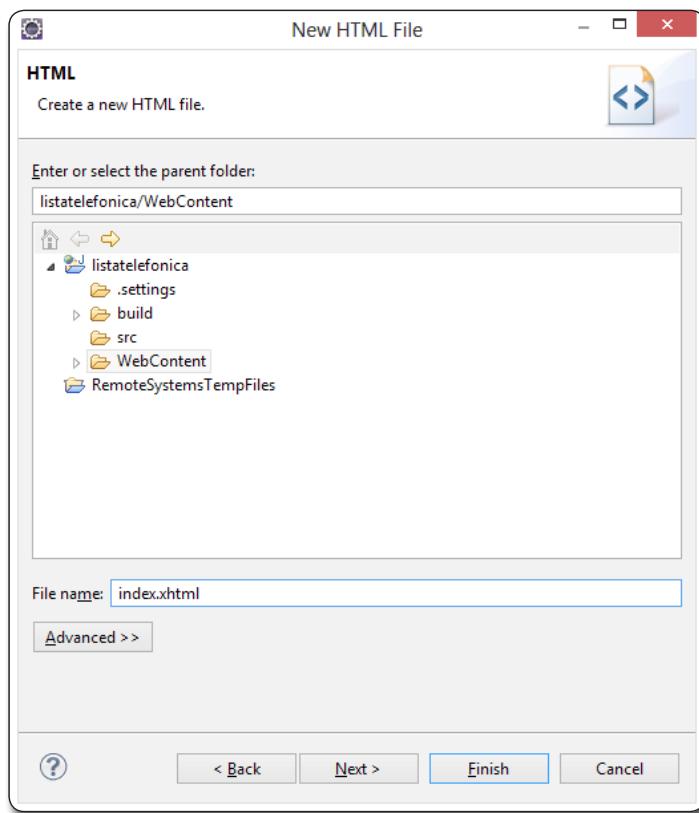


Figura 12. Tela de criação de uma página XHTML na IDE Eclipse

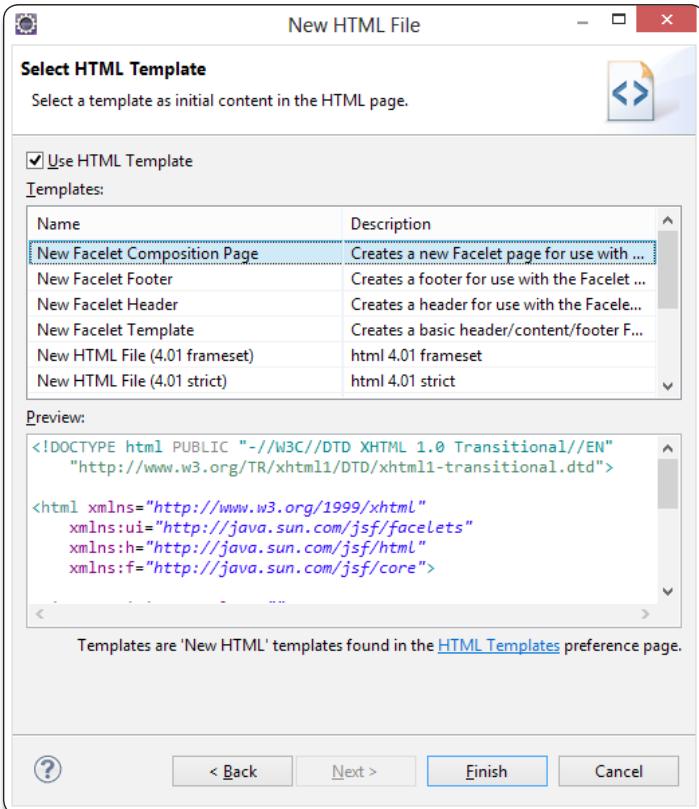


Figura 13. Tela de seleção de template para criação de uma página XHTML na IDE Eclipse

Listagem 6. Código da página index.xhtml.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:f="http://java.sun.com/jsf/core">

<head><title><h:outputText value="Lista Telefônica"/></title></head>
<h:body>
<h:form>
<h:outputText style="font-weight:bold" value="Lista Telefônica"/> <br/>
<h:outputText value="Selecione uma das opções abaixo:"/> <br/>
<h:outputLink value="cadastrar.xhtml">
<h:outputText value="Cadastrar Contatos"/>
</h:outputLink> <br/>
<h:outputLink value="pesquisar.xhtml">
<h:outputText value="Pesquisar Contatos"/>
</h:outputLink>
</h:form>
</h:body>

</html>
```

Listagem 7. Código da página cadastrar.xhtml.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:f="http://java.sun.com/jsf/core">

<head><title><h:outputText value="Lista Telefônica"/></title></head>
<h:body>
<h:form>
<h:outputText style="font-weight:bold" value="Lista Telefônica"/> <br/>
<h:outputText value="Cadastrar"/>
</h:form>
</h:body>
</html>
```

Listagem 8. Código da página pesquisar.xhtml.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:f="http://java.sun.com/jsf/core">

<head><title><h:outputText value="Lista Telefônica"/></title></head>
<h:body>
<h:form>
<h:outputText style="font-weight:bold" value="Lista Telefônica"/> <br/>
<h:outputText value="Pesquisar"/>
</h:form>
</h:body>
</html>
```

Listagem 9. Arquivo web.xml com configuração da página de boas-vindas.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.
  sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">
  <display-name>listatelefonica</display-name>
  <welcome-file-list>
    <welcome-file>index.xhtml</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
    <url-pattern>*.xhtml</url-pattern>
  </servlet-mapping>
</web-app>
```

Após criarmos as três páginas, vamos testar nossa aplicação. Devemos adicionar nosso projeto ao servidor de aplicação. Para isto, clicamos com o botão direito do mouse sobre o nome do servidor, na aba *Servers* da IDE Eclipse, e clicamos na opção *Add and Remove...*, como mostrado na **Figura 14**.

Na tela que se abrirá, selecione o projeto e clique no botão *Add* e depois em *Finish*. Agora iremos executar nossa aplicação web. Então, clique sobre o nome do projeto, selecione a opção *Run As* > *Run on Server*, como mostrado na **Figura 15**, e depois clique em *Finish*.

Para acompanharmos a execução de nosso servidor web, acesse a aba *Console* da IDE. Através desta opção visualizamos as informações de inicialização. O servidor estará pronto quando informar no log a frase: “INFO: Server startup in XXX ms”, onde XXX é o tempo que o servidor levou para terminar seu processo de inicialização.

Quando este processo tiver terminando, já teremos carregado nossa aplicação. Assim, nossa IDE abrirá a aba do browser interno, apresentando nossa tela inicial, mostrada na **Figura 16**.

Podemos acessar o menu da aplicação para verificarmos nossas funcionalidades. Ao clicar em Cadastrar Contatos, iremos para a página de cadastro de contatos de nossa lista, onde neste momento será apenas apresentado a palavra Cadastrar. O mesmo ocorre ao acessar a página Pesquisar Contatos, onde nos é apresentado Pesquisar.

Managed Beans e a persistência dos dados

Agora que já temos uma estrutura inicial, precisamos recuperar os dados que serão informados nos formulários pelo usuário para que possamos gravá-los no banco de dados e efetuar o cadastro de contatos em nossa lista.

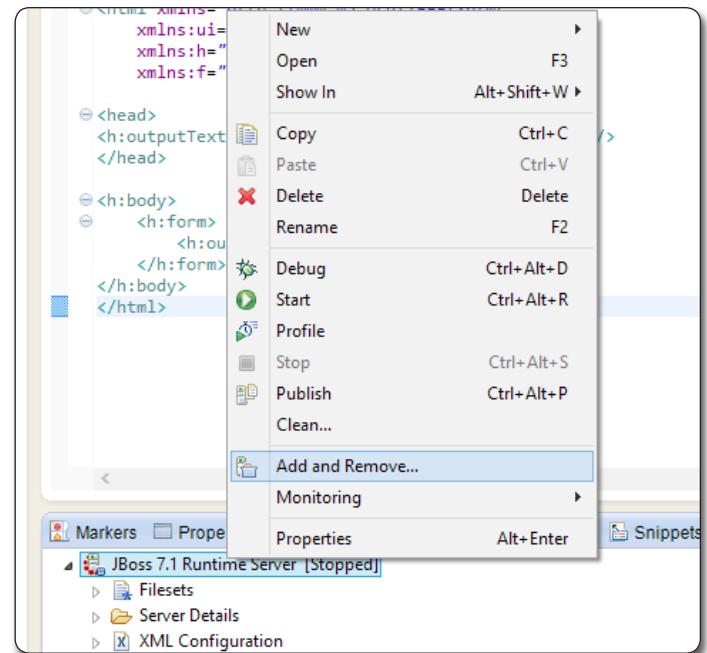


Figura 14. Tela para adição do projeto no servidor na IDE Eclipse

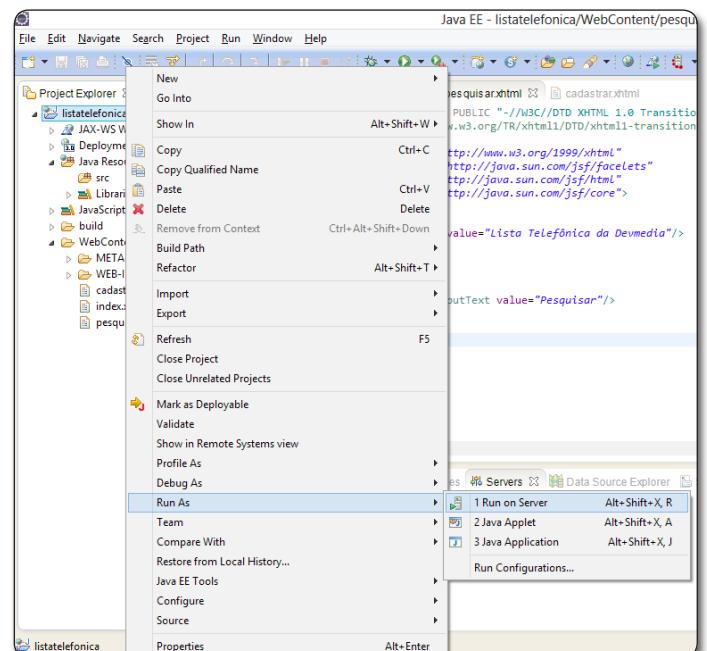


Figura 15. Tela para execução do projeto no servidor na IDE Eclipse

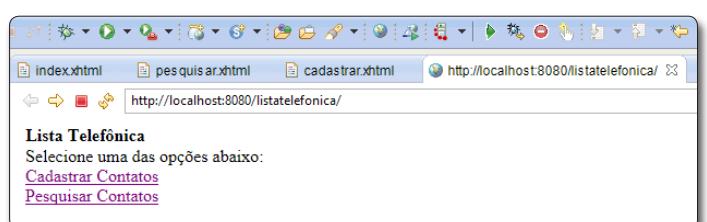


Figura 16. Tela principal da Lista Telefônica

Java Server Faces (JSF): Desenvolvimento de aplicações web

Para isto, precisaremos criar uma classe que receberá os valores vindos da requisição e as guardará. Esta classe é conhecida como Managed Bean, ou bean gerenciado, que é o que chamamos de POJO, acrônimo de Plain Old Java Object, que nada mais é do que um objeto simples.

Os objetos dessa classe terão seus métodos invocados a partir de comandos na web, se assemelhando ao estilo de programação Desktop, onde o clique de um botão, que gera um evento, invoca um método.

Para declararmos um Managed Bean usando JSF 2 utilizamos apenas a anotação `@ManagedBean`, sem a necessidade de configurações adicionais. Nas versões anteriores, esse processo envolvia configurações no arquivo `faces-config.xml`. Podemos verificar esta diferença nas **Listagens 10 e 11**. Note que na **Listagem 10** não precisamos especificar o nome e o escopo de nosso bean, pois utilizando a anotação `@ManagedBean`, por padrão, o nome do bean será o mesmo nome da classe com a primeira letra em minúsculo, e o escopo por padrão será o de `request`, porém podemos configurar estes dois itens a qualquer momento.

Listagem 10. Declaração do Managed Bean ContatoBean com JSF 2.

```
@ManagedBean  
public class ContatoBean {  
    // códigos aqui...  
}
```

Listagem 11. Declaração do Managed Bean ContatoBean em versões anteriores ao JSF 2.

```
<faces-config  
    xmlns="http://java.sun.com/xml/ns/javaee"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/  
    xml/ns/javaee/web-facesconfig_2_0.xsd"  
    version="2.0">  
  
<managed-bean>  
    <managed-bean-name>contatoBean</managed-bean-name>  
    <managed-bean-class>devmedia.mbs.ContatoBean</managed-bean-class>  
    <managed-bean-scope>request</managed-bean-scope>  
</managed-bean>  
  
</faces-config>
```

Agora iremos efetuar a ligação dos itens que estarão em nosso formulário de cadastro com os atributos de nosso Managed Bean. Para isto, iremos criar a classe **Contato**, disponível na **Listagem 12**, que será bem semelhante à classe **Usuario** utilizada em nosso artigo anterior. Ela contará com os dados a serem cadastrados para cada um dos contatos de nossa lista telefônica. Assim, esses dados deverão ser exibidos em nosso formulário para que o usuário entre com as informações que deverão ser cadastradas em nossa base de dados.

Como as informações deverão constar no formulário, estes atributos deverão fazer parte de nosso Managed Bean. Para fazer a ligação entre os valores dos componentes e os atributos do Managed Bean, usamos a Expression Language do JSF, efetuando o

que chamamos de **binding**, que é quando o valor do componente passa a ser representado pela propriedade do Managed Bean. Assim, quando o valor deste componente é alterado, a propriedade do Managed Bean é automaticamente modificada pelo JSF, e vice e versa.

Listagem 12. Código da classe Contato.

```
package devmedia.entity;  
  
public class Contato {  
  
    private String nome;  
    private String telefone;  
    private String endereco;  
  
    public Usuario(String nome, String telefone, String endereco) {  
        this.nome = nome;  
        this.telefone = telefone;  
        this.endereco = endereco;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public String getTelefone() {  
        return telefone;  
    }  
  
    public void setTelefone(String telefone) {  
        this.telefone = telefone;  
    }  
  
    public String getEndereco() {  
        return endereco;  
    }  
  
    public void setEndereco(String endereco) {  
        this.endereco = endereco;  
    }  
}
```

Vamos agora atualizar nossa página `cadastrar.xhtml`, criando nosso formulário de cadastro de acordo com a **Listagem 13**. Podemos verificar que na tag `<h:inputText value="#{contatoBean.contato.nome}" />` o binding é feito a partir do atributo `value`. Deste modo, dentro do **ContatoBean** estamos acessando a propriedade **contato**, e nesse **contato** queremos acessar a propriedade **nome**. Este acesso é realizado a partir dos métodos assessores `get`/`set` de cada atributo. Portanto, precisamos sempre criar os métodos assessores para que o JSF possa invocá-los.

Verificamos que o botão *Adicionar* de nosso formulário está fazendo o binding com o método `adicionar()` que existirá no **ContatoBean**. Por termos um escopo de requisição (`request`), será criada uma nova instância da classe **ContatoBean** para armazenamento dos dados do formulário no objeto **Contato** deste bean. Quando o método `adicionar()` é acionado, este efetua a gravação

do objeto **Contato** no banco de dados e retorna ao Managed Bean que irá apontar neste caso para a próxima página que o usuário será encaminhado após a gravação.

Listagem 13. Formulário de cadastro da página cadastrar.xhtml.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:f="http://java.sun.com/jsf/core">

<head>
<title><h:outputText value="Lista Telefônica"/></title>
</head>
<h:body>
<h:form>
<h:outputText style="font-weight:bold" value="Lista Telefônica" />
<br /><br/>
<h:outputLabel value="Nome:" />
<h:inputText id="nome" value="#{contatoBean.contato.nome}" /> <br/><br/>
<h:outputLabel value="Telefone:" />
<h:inputText id="telefone" value="#{contatoBean.contato.telefone}" />
<br /><br/>

<h:outputLabel value="Endereço:" />
<h:inputText id="endereco" value="#{contatoBean.contato.endereco}" />
<br /><br/>
<h:commandButton value="Adicionar" action="#{contatoBean.adicionar}" />
</h:form>
</h:body>
</html>
```

Temos nosso Managed Bean **ContatoBean** na **Listagem 14**. Como podemos verificar, o método **adicionar()** tem como retorno uma **String**, que neste caso indica o nome da próxima página para a qual o usuário será encaminhado. Como indicamos a string “sucesso”, o JSF irá procurar algum XHTML que possua este nome para o encaminhamento. Este comportamento também é válido diretamente no **commandButton** do formulário, onde podemos indicar diretamente para qual página o botão irá encaminhar na propriedade **action**.

Complementando nosso cadastro, temos as classes **ContatoDAO**, **FabricaConexao** e **AbstractDAO** (as duas últimas utilizadas em nosso artigo anterior), que serão usadas para a parte de persistências das informações no Banco de Dados. Elas estão disponíveis nas **Listagens 15, 16 e 17**, já analisadas no artigo Introdução ao desenvolvimento de aplicações web, publicado na Easy Java Magazine 36.

Como banco de dados, adotaremos a versão 5.6.19 do MySQL. Sendo assim, ele deve ser baixado e instalado, estando seu endereço para download e alguns materiais para referência na seção **Links**, em caso de dúvidas. Para conexão com o banco de dados, iremos utilizar a API JDBC. Esta contém classes para envio de instruções SQL ao banco de dados. Para finalizarmos, devemos baixar também o driver de conexão do banco de dados, cujo endereço também se encontra na seção **Links**. Este é um arquivo JAR e deve ser acrescentado à pasta *lib* da aplicação.

Listagem 14. Código da classe ContatoBean.

```
package devmedia.mbs;

import javax.faces.bean.ManagedBean;
import devmedia.dao.AbstractDAO;
import devmedia.dao.ContatoDAO;
import devmedia.entity.Contato;

@ManagedBean
public class ContatoBean {
    private Contato contato;
    private String erro;
    private AbstractDAO<Contato> contatoDAO;

    public ContatoBean() {
        contatoDAO = new ContatoDAO();
        contato = new Contato();
    }

    public String adicionar(){
        try{
            contatoDAO.adicionar(contato);
            this.contato = new Contato();
        }catch(Exception ex){
            System.out.println("Erro:" + ex);
            this.erro = ex.getMessage();
            return "erro";
        }
        return "sucesso";
    }

    public Contato getContato() {
        return contato;
    }

    public void setContato(Contato contato) {
        this.contato = contato;
    }

    public String getErro() {
        return erro;
    }

    public void setErro(String erro) {
        this.erro = erro;
    }
}
```

No MySQL, criaremos um banco de dados denominado *devmedia* e uma tabela de nome *usuário*, de acordo com a **Listagem 18**.

Na **Listagem 15** temos na classe **ContatoDAO** o método **adicionar()**, que será o responsável por persistir nossas informações no banco de dados. Este método é invocado em nosso Managed Bean **ContatoBean**, que após a gravação será o responsável por distribuir a navegação da aplicação de acordo com o processo de cadastro no banco, podendo ocorrer sucesso ou erro para esta ação.

Agora podemos executar nossa aplicação, acessando a opção *Cadastrar Contatos*. O resultado obtido pode ser visualizado na **Figura 17**. O formulário de cadastro pode então ser preenchido, e logo após, submetido clicando no botão *Adicionar*. Caso a gravação ocorra com sucesso, será apresentada a página de sucesso na gravação, mostrada na **Figura 18**.

Java Server Faces (JSF): Desenvolvimento de aplicações web

Listagem 15. Código da classe ContatoDAO.

```
package devmedia.dao;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import devmedia.entity.Contato;

public class ContatoDAO extends AbstractDAO<Contato> {

    private List<Contato> lista;

    public ContatoDAO() {
        super();
        lista = new ArrayList<Contato>();
    }

    public List<Contato> getLista() {
        return lista;
    }

    public void adicionar(Contato contato) {
        try {
            PreparedStatement ptmt = conn.prepareStatement("insert into contato
                (nome, telefone, endereco) values (?, ?, ?)");
            ptmt.setString(1, contato.getNome());
            ptmt.setString(2, contato.getTelefone());
            ptmt.setString(3, contato.getEndereco());
            ptmt.executeUpdate();
            ptmt.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Listagem 16. Código da classe FabricaConexao.

```
package devmedia.dao;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class FabricaConexao {
    String driverConexaoBD = "com.mysql.jdbc.Driver";
    String urlConexaoBD = "jdbc:mysql://localhost:3306/devmedia";
    String usuario = "root";
    String senha = "root";

    private static FabricaConexao fabricaConexao = null;
    private FabricaConexao() {
        try {
            Class.forName(driverConexaoBD);
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }

    public Connection getConexao() throws SQLException {
        Connection conn = null;
        conn = DriverManager.getConnection(urlConexaoBD, usuario, senha);
        return conn;
    }

    public static FabricaConexao getInstance() {
        if (fabricaConexao == null) {
            fabricaConexao = new FabricaConexao();
        }
        return fabricaConexao;
    }
}
```

Listagem 17. Código da classe AbstractDAO.

```
package devmedia.dao;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.List;

import devmedia.entity.Contato;

public abstract class AbstractDAO<T> {

    protected Connection conn;

    public AbstractDAO() {
        try {
            conn = FabricaConexao.getInstance().getConexao();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    public abstract List<T> getLista(Contato contato);

    public abstract void adicionar(T objeto);
}
```

Listagem 18. Script de criação do banco de dados devmedia e da tabela contato.

```
CREATE DATABASE devmedia;
USE devmedia;
CREATE TABLE contato(id BIGINT NOT NULL AUTO_INCREMENT, nome
VARCHAR(255), telefone VARCHAR(255), endereco VARCHAR(255), primary key
(id));
```

The screenshot shows a web application interface titled 'Lista Telefônica'. It contains three input fields: 'Nome' with the value 'Beatrix', 'Telefone' with the value '3333-4444', and 'Endereço' with the value 'Rua sem curva, 03'. Below these fields is a blue 'Adicionar' button. The browser's address bar shows the URL <http://localhost:8080/listatelefonica/cadastrar.xhtml>. The top of the browser window displays tabs for 'cadastrar.xhtml', 'Contato.java', and 'FabricaConexao.java'.

Figura 17. Tela de cadastro da Lista Telefônica

The screenshot shows a web application interface titled 'Lista Telefônica'. It displays a green success message: 'Cadastramento realizado com sucesso!' (Registration successful!). Below the message is a blue 'Voltar' (Back) link. The browser's address bar shows the URL <http://localhost:8080/listatelefonica/cadastrar.xhtml>. The top of the browser window displays tabs for 'cadastrar.xhtml', 'Contato.java', and 'FabricaConexao.java'.

Figura 18. Tela de sucesso no cadastramento de contatos da Lista Telefônica

Componente DataTable e a listagem de contatos

Após criarmos o cadastro de contatos de nossa Lista Telefônica, queremos efetuar buscas destes contatos, deixando assim nossa lista telefônica com a principal funcionalidade para a qual foi desenvolvida.

Para desenvolvermos a listagem dos contatos, iremos utilizar um componente voltado para esta função, chamado **dataTable**. No **dataTable**, temos que indicar qual será a fonte dos dados a serem exibidos. Esta indicação é feita na propriedade **value** do componente, que espera receber uma referência para um método do Managed Bean que retorna uma Collection, como um **List** ou um **Set**.

O componente **dataTable** funciona como um **for**, no qual dizemos qual a Collection que será percorrida, e qual a variável em que ele armazenará o item atual durante a interação do laço. A variável temporária é definida através da propriedade **var**, que neste caso definimos como **var="contato"**.

Iremos atualizar a página *pesquisar.xhtml* para inserirmos esta lógica de pesquisa de nossos contatos, conforme mostrado na **Listagem 19**. Nesta listagem destacamos a utilização do **h:dataTable**, onde temos o atributo **value="#{contatoBean.contatos}"** que será a fonte dos dados apresentados na tabela. Por ser uma tabela, temos a tag **<h:column>** que indica a criação de uma coluna em nossa tabela, sendo utilizada a tag **<f:facet name="header">Nome</f:facet>** para destacarmos o título da coluna em questão.

Para melhorarmos a apresentação dos dados, criaremos um arquivo de estilos para *zebrarmos* a nossa tabela. Para isto, criamos o arquivo *estilos.css* dentro de uma pasta *css* criada abaixo da pasta *Web-Content* do projeto. O arquivo de estilos é exibido na **Listagem 20**. As explicações de como zebrar tabelas com JSF podem ser conferidas na seção **Links**.

Observe na **Listagem 19** que em nosso **dataTable** inserimos na propriedade **rendered** a seguinte condição: **rendered="#{!empty contatoBean.contatos}"**,

onde estamos dizendo ao JSF para renderizar este componente apenas se a condição for satisfeita, ou seja, a tabela apenas será exibida se nossa lista de contatos não estiver vazia. Uma lógica parecida é feita com o componente **outputText**: **<h:outputText value="Não foram encontrados registros com este critério de busca!" rendered="#{empty contatoBean.contatos and !empty contatoBean.contato.nome}" />**, onde exibiremos a mensagem que não foram encontrados registros apenas se nossa lista estiver vazia e se o atributo **nome** não estiver vazio.

Após atualizarmos nossas páginas, iremos atualizar nosso Managed Bean e DAO para acrescentarmos a listagem de contatos, como mostrado nas **Listagens 21 e 22**. Podemos ver na **Listagem 21** que acrescentamos o método **pesquisar()** que será acionado pelo botão de nossa página de pesquisa. Neste método efetuamos a chamada à consulta de contatos que será realizada na classe **ContatoDAO** da **Listagem 22**. Nesta classe o método **getLista()** é o responsável por efetuar a consulta no banco de dados e retornar a lista de contatos cadastrados.

Podemos perceber na **Listagem 21** que apenas fazemos a nossa lista de contatos do Managed Bean receber o retorno da consulta no banco de dados, **this.contatos = contatoDAO.getLista(contato)**, e pronto, nossa página, através do **getContatos()** de nossa lista, irá exibir os dados constantes nesta lista para o usuário.

Agora podemos executar nossa aplicação, acessando a opção *Pesquisar Contatos*. O resultado obtido pode ser visualizado na **Figura 19**. O formulário de pesquisa realiza buscas com parte do nome do contato digitado no campo *Nome*, trazendo todas as opções do banco que satisfaçam este critério, como mostra a **Figura 20**. A busca também poderá não retornar informações caso estas não sejam encontradas, conforme mostra a **Figura 21**.

Listagem 19. Página *pesquisar.xhtml* atualizada para a listagem de contatos.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">

  <head>
    <title><h:outputText value="Lista Telefônica" /></title>
    <link rel="stylesheet" type="text/css" href="css/estilos.css"/>
  </head>
  <h:body>
    <h:form>
      <h:outputText style="font-weight:bold" value="Lista Telefônica" />
      <br/><br/>
      <h:outputLabel value="Nome:" />
      <h:inputText id="nome" value="#{contatoBean.contato.nome}" />
      <h:commandButton value="Pesquisar" action="#{contatoBean.pesquisar}" />
      <br /><br />
      <h:dataTable styleClass="tabela" value="#{contatoBean.contatos}"
        headerClass="cabecalho" columnClasses="ultimo,primeiro"
        var="contato" rendered="#{!empty contatoBean.contatos}">
        <h:column>
          <f:facet name="header">Nome</f:facet>
          #{contato.nome}
        </h:column>
        <h:column>
          <f:facet name="header">Telefone</f:facet>
          #{contato.telefone}
        </h:column>
        <h:column>
          <f:facet name="header">Endereço</f:facet>
          #{contato.endereco}
        </h:column>
      </h:datatable> <br />
      <h:outputText value="Não foram encontrados registros com
        este critério de busca!"
        rendered="#{empty contatoBean.contatos and !empty contatoBean.contato.nome}" />
      <br /><br />
      <h:outputLink value="index.xhtml">
        <h:outputText value="Voltar"/>
      </h:outputLink> <br />
    </h:form>
  </h:body>
</html>
```

Java Server Faces (JSF): Desenvolvimento de aplicações web

Listagem 20. Arquivo de estilos css, estilos.css.

```
/* Definição de estilos para toda a tabela*/
.tabela {
border: 1px solid green;
}

/* Definição de estilos para linha de cabeçalho da tabela*/
.cabecalho {
text-align: center;
font: 11px Arial, sans-serif;
font-weight: bold;
color: Snow;
background: #008B45;
}

/* Definição de estilos para coluna */
.primeiro {
text-align: center;
font: 11px Arial, sans-serif;
background: #A2CD5A;
}

/* Definição de estilos para coluna */
.ultimo {
font: 11px Arial, sans-serif;
text-align: center;
background: #BCEE68;
}
```

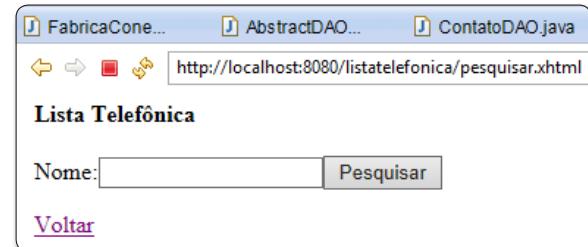


Figura 19. Tela de pesquisa de contatos da Lista Telefônica

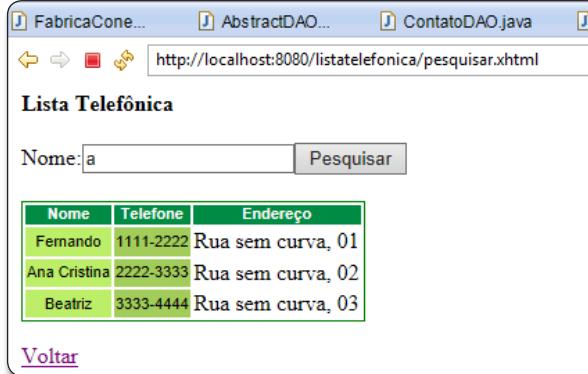


Figura 20. Tela com a pesquisa de contatos da Lista Telefônica

Listagem 21. Classe ContatoBean atualizada com a listagem de contatos.

```
package devmedia.mbs;

import java.util.List;

import javax.faces.bean.ManagedBean;

import devmedia.dao.AbstractDAO;
import devmedia.dao.ContatoDAO;
import devmedia.entity.Contato;

@ManagedBean
public class ContatoBean {

private List<Contato> contatos;
private Contato contato;
private String erro;
private AbstractDAO<Contato> contatoDAO;

public ContatoBean() {
contatoDAO = new ContatoDAO();
contato = new Contato();
}

public String adicionar(){
try{
contatoDAO.adicionar(contato);
this.contato = new Contato();
}catch(Exception ex){
System.out.println("Erro:" + ex);
this.erro = ex.getMessage();
return "erro";
}

return "sucesso";
}

public String pesquisar(){
```

```
try{
this.contatos = contatoDAO.getLista(contato);
}catch(Exception ex){
System.out.println("Erro:" + ex);
this.erro = ex.getMessage();
return "erro";
}

return "pesquisar";
}

public List<Contato> getContatos() {
return contatos;
}

public void setContatos(List<Contato> contatos) {
this.contatos = contatos;
}

public Contato getContato() {
return contato;
}

public void setContato(Contato contato) {
this.contato = contato;
}

public String getErro() {
return erro;
}

public void setErro(String erro) {
this.erro = erro;
}
```

Listagem 22. Classe ContatoDAO atualizada com a listagem de contatos.

```
package devmedia.dao;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import devmedia.entity.Contato;

public class ContatoDAO extends AbstractDAO<Contato> {

    private List<Contato> lista;

    public ContatoDAO() {
        super();
        lista = new ArrayList<Contato>();
    }

    public List<Contato> getLista(Contato contato) {
        try {
            PreparedStatement ptmt = conn
                .prepareStatement("select * from contato where nome like ?");
            ptmt.setString(1, "%" + contato.getNome() + "%");
            ResultSet rs = ptmt.executeQuery();
            while (rs.next()) {
                contato = new Contato(rs.getString("nome"), rs.getString("telefone"),
                    rs.getString("endereco"));
                lista.add(contato);
            }
            rs.close();
            ptmt.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
        return lista;
    }

    public void adicionar(Contato contato) {
        try {
            PreparedStatement ptmt = conn.prepareStatement(
                "insert into contato(nome, telefone, endereco) values (?, ?, ?)");
            ptmt.setString(1, contato.getNome());
            ptmt.setString(2, contato.getTelefone());
            ptmt.setString(3, contato.getEndereco());
            ptmt.executeUpdate();
            ptmt.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

FabricaCone... AbstractDAO... ContatoDAO.java

http://localhost:8080/listatelefonica/pesquisar.xhtml

Lista Telefônica

Nome: 8

Não foram encontrados registros com este critério de busca!

[Voltar](#)

Figura 21. Tela mostrando que dados não foram encontrados na pesquisa da Lista Telefônica

Uma das características de destaque do JSF é o quesito produtividade, onde podemos perceber a redução de complexidade ao não termos que manipular diretamente a requisição e sua resposta, visto que nos atemos apenas à criação de Managed Beans que concentram as atividades a serem realizadas na aplicação de acordo com o negócio.

O JSF é uma especificação muito completa para desenvolvimento web, e merece todo papel de destaque neste quesito. Trabalhamos apenas com uma pequena parte de seu conteúdo. O exemplo apresentado em nosso artigo pode simplesmente ser adaptado para uma utilização mais complexa, como um controle de alunos de uma instituição de ensino. Entretanto, é preciso estudar outros conceitos não abordados ainda, como controle de escopo, validações, controle de erros, entre outros. Também podemos efetuar a atualização da parte de acesso aos dados no banco de dados

utilizando alguma especificação ou framework voltado para este fim, como a JPA, por exemplo.

Lembrando que o JSF pode ser totalmente integrado com tecnologias como JPA, EJB, CDI, e demais especificações da plataforma Java EE (agora na versão 7), dando a segurança de continuidade que as empresas precisam para criar suas aplicações.

Autora

Lorena S. Dourado
dourado.lore@gmail.com
Especialista no desenvolvimento de aplicações Web com Java pela Universidade Norte do Paraná. Trabalha com Java há 10 anos, atualmente como Ingeniera Sistemas Senior na Indra Company, multinacional de Tecnologia Espanhola atuante nas áreas de Energia e Indústria, Serviços financeiros, Administrações Públicas e Saneamento, Telecom e Mídia, Transporte e Tráfego, bem como Defesa e Segurança.



Links:

Página oficial da especificação JSR-344 - JavaServer Faces 2.2.
jcp.org/en/jsr/detail?id=344

Página oficial do aplicativo Wireshark, utilizado para visualizar o conteúdo de uma requisição HTTP.
wireshark.org

Endereço para download do MySQL.
dev.mysql.com/get/Downloads/MySQL-5.6/mysql-5.6.19-win32.zip

Endereço para download do Driver do MySQL.
dev.mysql.com/downloads/connector/j/5.1.html

Hibernate Validator e a consistência dos dados

Utilize validações do Hibernate para estabelecer as regras e verificar seu cumprimento

Em qualquer tipo de aplicação, receber dados sempre foi um problema. Não é possível prever o que se passa na cabeça do usuário ou a interpretação que ele faz da interface da aplicação. Dessa forma, não podemos garantir que o sistema receberá as entradas no formato que precisa. É comum usuários digitarem números ou caracteres especiais em campos que não estão preparados para receber esse tipo de entrada. Se falarmos de formatos mais específicos, como datas, por exemplo, esse problema fica ainda mais evidente.

Isso ocorre porque o usuário não tem ideia de que a interface que ele visualiza é apenas a “ponta do iceberg”. Existem outras camadas que darão todo tipo de tratamento aos dados que ele acaba de inserir.

Dessa forma, se não for feito o tratamento adequado por parte das camadas da aplicação – não apenas na camada de interface, mas em todas as camadas -, erros podem ocorrer, sejam exceções que interromperão a execução do sistema ou problemas de integridade ou corrupção dos dados.

Cada entrada de dados do usuário segue um propósito, ou seja, é informação que será armazenada e processada pelo software com o intuito de fornecer uma resposta ao usuário. Neste contexto, é natural que algumas regras sejam estabelecidas para moderar a interação entre o usuário e o sistema. É para garantir o cumprimento dessas regras que o conceito de validação aparece. Validar é determinar se os dados capturados pelo sistema cumprem as regras lógicas definidas para manter a sua consistência.

Também é preciso ressaltar que a consistência dos dados influencia diretamente a qualidade do software, interferindo na forma como o usuário interage com a aplicação. Por exemplo, a experiência de receber uma mensagem de erro do sistema ao deixar um campo em branco ou preenchê-lo incorretamente é bastante desagradável, podendo deixar o usuário confuso em relação à forma de utilizar o software.

Fique por dentro

Nesse artigo exploramos o módulo do framework Hibernate que fornece ferramentas de validação, garantindo que a aplicação receba entradas de acordo com regras estabelecidas e, dessa forma, aumente a qualidade dos dados. O exemplo é útil por demonstrar em detalhes como utilizar o Hibernate Validator em uma aplicação Web baseada em Spring MVC.

Como um problema clássico do desenvolvimento de software – pode-se dizer até da interação homem/máquina –, a validação das entradas dos usuários já foi abordada de diversas formas. Podemos citar como exemplo o uso de “máscaras” em campos de dados, que limitam as opções de digitação. Exemplos simples de restrições podem ser datas em um formato específico, a obrigatoriedade de um campo, campos que só podem receber números, etc.

Para validar essas restrições, diversas abordagens podem ser utilizadas, como o uso de exceções para impedir que dados inconsistentes ultrapassem determinadas camadas da aplicação ou o uso de JavaScript nos campos de um formulário na Web.

A principal distinção entre os exemplos citados anteriormente reside justamente na camada em que cada uma das validações ocorre. É muito comum definir validações na camada de visão, que tem mais contato com o usuário (consequentemente, sobre a qual o usuário tem mais controle), e dependendo da arquitetura sobre a qual o software funcione isso pode torná-lo vulnerável. Há casos de validações escritas em *JavaScript*, portanto executadas no navegador, que são simplesmente desabilitadas pelo usuário.

O framework Hibernate fornece uma alternativa interessante, chamada **Validator**, para a validação de dados em qualquer arquitetura (Web, Desktop, etc.). De acordo com o site do Hibernate (veja o endereço na seção **Links**), o Validator permite expressar as regras de maneira padronizada através do uso de anotações (vide **BOX 1**) que representem as restrições desejadas.

Como o Validator expressa as regras e restrições da aplicação por meio de anotações que podem ser estendidas, a implementação

das validações é transparente, ou seja, não está relacionada a uma camada específica da aplicação ou a qualquer padrão de projeto.

O Hibernate Validator parte da premissa estabelecida pelo DRY (*Don't Repeat Yourself*, veja o **BOX 2**), para que as regras e respectivas verificações sejam escritas uma única vez e gerenciadas de maneira centralizada. Dessa forma, mesmo que a arquitetura adotada seja em múltiplas camadas, não será necessário duplicar as validações.

BOX 1. Anotações (Annotations)

Anotações são recursos para a declaração de metadados – dados que descrevem outros dados – úteis para localizar dependências, configurações ou para fazer verificações lógicas. Essas definições serão, então, interpretadas pelo compilador para realizar uma determinada tarefa.

BOX 2. DRY (Don't Repeat Yourself)

É um conceito de programação, descrito por Andy Hunt e Dave Thomas – ambos cofundadores do Manifesto Ágil –, no livro “The Pragmatic Programmer”, que estabelece como meta a redução das repetições de informação de todo e qualquer tipo, especialmente entre as camadas da aplicação. De acordo com esse princípio, na medida em que cada elemento de negócio ou conhecimento do software é implementado univocamente, a modificação de um elemento não acarreta a necessidade de modificar outros.

A necessidade de validar os dados de um objeto é bastante antiga e está presente na arquitetura Java há bastante tempo. Um marco importante nessa história é a disponibilização, em 2009, da *Java Specification Request 303* (JSR-303, vide o site da Java Community Process na seção **Links**), que define um padrão para a implementação de validações em aplicações Java. Com a JSR-303 surge a *Java Bean Validation 1.0* (JBV), um framework definido como parte da especificação Java EE 6, estabelecendo que as validações sejam feitas nas classes da camada de modelo – normalmente implementadas como *JavaBeans* – por meio de anotações cujos metadados podem ser estendidos ou sobreescritos por meio de XML.

Cada anotação corresponde a uma validação, que por sua vez faz a verificação do objeto e responde se este obedece ou não a regra estabelecida. De fato, o Hibernate já realiza essa operação na camada de persistência quando uma inserção ou atualização está sendo realizada, mas com o Validator essa validação pode ser acionada em qualquer momento.

Para isso, a especificação JBV é compatível com diversos tipos de frameworks e, portanto, pode ser utilizada tanto em aplicações Web quanto em aplicações Desktop, sendo o Hibernate Validator a referência de implementação. Seu uso é a **boa prática** recomendada para a utilização de *Java Bean Validation*.

Atualmente estamos na versão Java Bean Validation 1.1, presente na JSR-349. Nesta versão foram feitos vários aprimoramentos, dos quais podemos destacar a validação em “nível de método” (o que permite validar os parâmetros recebidos pelo método, bem como seu valor de retorno), a interpolação de mensagens de erro por meio de Expressões de Linguagem (*EL Expressions*), bem como um foco maior no uso de Injeção de Dependência para os *JavaBeans* validados.

No Hibernate Validator existem dois níveis de validação: em nível de aplicação ou em nível de persistência. Em nível de aplicação as instâncias de classes são validadas em tempo de execução. Em nível de persistência, as validações são incorporadas pelo modelo implementado pelo Hibernate no momento da criação do banco de dados, por meio das especificações das entidades criadas como campos nulos, valores únicos, tipos de campos etc.

Uma vez que a validação de dados é uma tarefa constante e comum em todas as camadas da aplicação (da apresentação à persistência), vamos demonstrar ao longo das próximas seções desse artigo como utilizar o Hibernate Validator utilizando um exemplo.

Instalação e criação do projeto

Não há procedimentos de instalação para o Hibernate Validator. Na verdade, é necessário baixar os pacotes, colocá-los em seu *Classpath* e seguir alguns procedimentos de configuração como demonstraremos a seguir.

Para baixar o Hibernate Validator, vá ao site do Hibernate (endereço na seção **Links**) e clique no link *More* na caixa *Hibernate Validator*. Você será redirecionado para a página do Validator, na qual basta clicar no link *Download* para baixá-lo. Faremos o download da versão 5.1.1.

O site do Hibernate indica o uso do Maven em projetos com o Validator. De fato, o Maven é bastante útil em *builds* de projetos desse tipo, que geralmente têm muitas dependências. No entanto, o uso do Maven deve ser explorado com detalhes que não são foco desse artigo, portanto, vamos criar simplesmente um *Dynamic Web Project* na IDE Eclipse e copiar os *Jars* do Validator para nosso *Classpath*.

No Eclipse, escolha a opção *File > New > Dynamic Web Project*. Na janela que é exibida, como pode ser visto na **Figura 1**, digite no campo *Project Name* o nome do nosso projeto – “GestaoAcademica”. Nosso projeto deve utilizar o *container* Tomcat na versão 6 ou superior. Logo, certifique-se de que no campo *Target Runtime* uma dessas opções está selecionada e clique em *Finish*.

Em seguida, descompacte o arquivo ZIP do Hibernate Validator que foi baixado anteriormente e verá dois diretórios com *jars* – *optional* e *required*. Copie para o *Classpath* do projeto o conteúdo de *required*. Esses *jars* são suficientes para que nosso exemplo funcione.

Nosso projeto seguirá a arquitetura *Model-View-Controller* e, para isso, faremos uso do framework Spring MVC. A integração do Hibernate Validator com o Spring MVC – ou qualquer outro framework – é transparente, mas para que o exemplo funcione, não deixe de fazer as configurações necessárias do Spring também. Nesse artigo demonstraremos algumas configurações básicas do Spring MVC, em especial aquelas que interferem diretamente no funcionamento do nosso exemplo. Para mais detalhes sobre o Spring MVC, visite o site do Spring Framework (veja o endereço na seção **Links**).

Vamos agora criar o pacote base de nossa aplicação. Com o botão direito sobre a pasta *Java Resources\src* no *Package Explorer* do Eclipse, selecione *New > Package*. Conforme demonstra a **Figura 2**, digite “br.com.devmedia.gestaoacademica” no campo *Name* e clique em *Finish*.

Hibernate Validator e a consistência dos dados

Para implementar a arquitetura MVC, crie dentro do pacote base os pacotes **control** e **model**. Nossas páginas (a camada *view*) ficarão no diretório *WEB-INF/views*. Dessa forma, a estrutura do projeto deve ficar como mostrado na **Figura 3**.

No Hibernate Validator as restrições são aplicadas por meio de anotações. Para que o código suporte as anotações do Spring, a instrução `<tx:annotation-driven>` é colocada no arquivo *spring-context.xml*. Essa tag diz ao Spring para examinar todos os Beans e procurar por aqueles que possuem anotações.

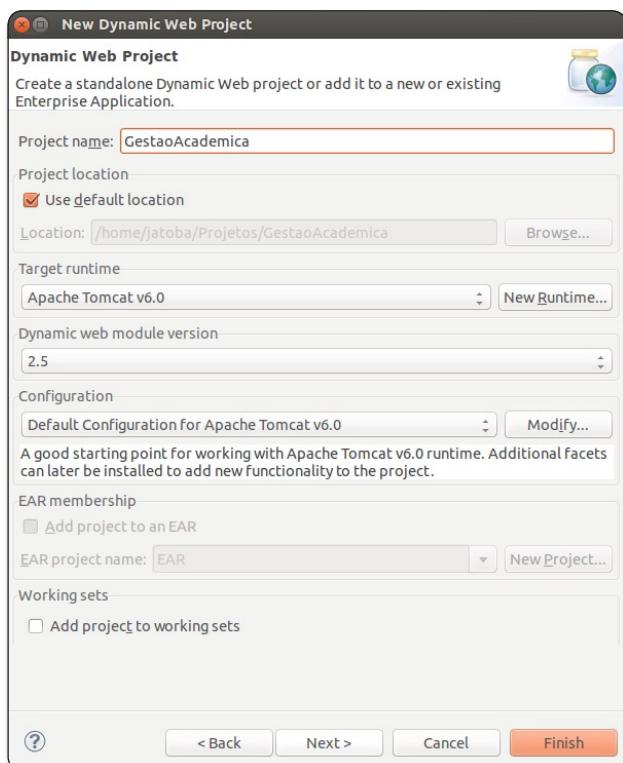


Figura 1. Criando um Dynamic Web Project no Eclipse

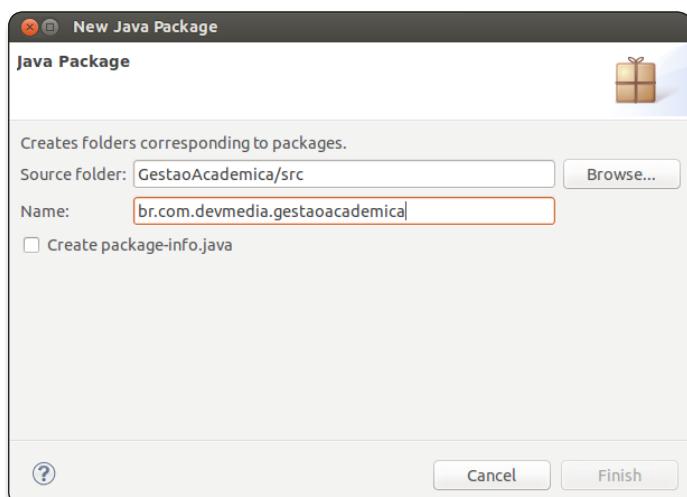


Figura 2. Criando o pacote-base da aplicação

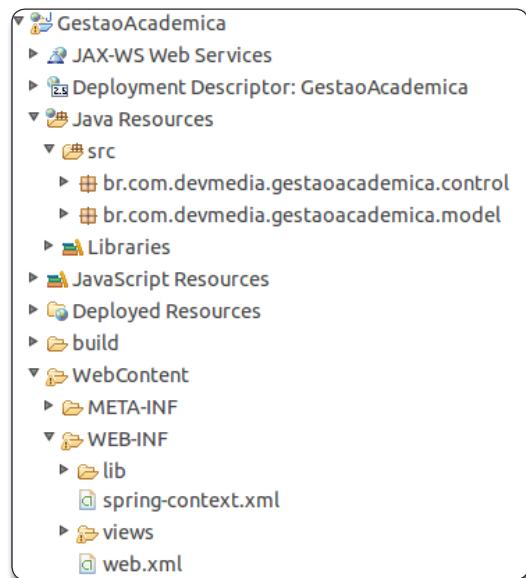


Figura 3. Estrutura do projeto após a criação do pacote base

Da mesma forma, para que possamos utilizar anotações do Hibernate Validator, ou seja, tornar nosso projeto compatível com as anotações descritas na JSR-349, devemos utilizar a instrução `<mvc:annotation-driven>`. A **Listagem 1** apresenta como o arquivo *spring-context.xml* deve ficar.

Além das configurações de anotações que citamos anteriormente, este arquivo possui configurações essenciais do Spring, como o pacote base da aplicação e outras configurações relacionadas à camada de visualização.

Desenvolvendo um primeiro exemplo e aplicando restrições no modelo

Vamos criar uma única classe na camada de modelo – **Docente** –, que será justamente a classe que passará pela validação. Esta classe – um JavaBean – possui quatro atributos – **id**, **nome**, **matricula** e **titulacao** – e seu código fonte pode ser visto na **Listagem 2**.

Na introdução desse artigo dissemos que há dois níveis de validação: em nível de aplicação ou em nível de persistência. Embora não seja o que vem à mente quando falamos em validação, podemos ver as validações em nível de persistência expressas, por exemplo, pela anotação `@NotNull` no atributo **id**. Esta anotação diz ao Hibernate que este campo não pode ser nulo em hipótese alguma e, dessa forma, quando o respectivo mecanismo do Hibernate criar as tabelas no banco de dados, o campo terá essa característica.

No nosso exemplo, faremos a validação em nível de aplicação do atributo **matricula**. Queremos que esse campo nunca esteja vazio e que seu valor tenha entre 8 e 16 caracteres. Para fazer essas restrições, podemos utilizar anotações nativas da *Java Validation API*. A anotação `@NotEmpty` garante que um atributo não esteja vazio e a anotação `@Size` estabelece o tamanho do valor que um atributo pode receber, como pode ser visto no trecho de código `@Size(min = 8, max = 16)` na **Listagem 2**.

Listagem 1. Código-fonte do arquivo spring-context.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee.xsd
        http://www.springframework.org/schema/lang
        http://www.springframework.org/schema/lang/spring-lang.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
    <context:annotation-config />
    <context:component-scan base-package="br.com.devmedia.gestaoacademica"/>
    <tx:annotation-driven />
    <mvc:annotation-driven />

    <bean id="jspViewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
            value="org.springframework.web.servlet.view.JstlView"/>
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

O Hibernate Validator é a implementação de referência da *Java Bean Validation API* desde sua origem, no entanto, usar validações desta API – contidas no pacote **javax.validation** – garantem a portabilidade do código. Por outro lado, utilizamos também a validação **@NotEmpty**, contida no pacote **org.hibernate.validator**.

O catálogo de anotações do Hibernate Validator possui diversas restrições que podem ser usadas livremente, dentre as quais gostaríamos de destacar:

- **@Email:** verifica se a **String** é um e-mail válido;
- **@Length(min=, max=):** verifica se o tamanho de uma **String** está dentro dos limites estabelecidos nas propriedades **min** e **max**;
- **@Max(value=)** e **@Min(value=):** verificam, respectivamente, se o valor é maior ou menor que o estabelecido na propriedade **value**;
- **@Past** e **@Future:** verificam, respectivamente, se uma data é anterior ou posterior à data corrente;
- **@AssertFalse:** testa se um valor booleano é falso;
- **@AssertTrue:** testa se um valor booleano é verdadeiro;

Listagem 2. Código-fonte da classe Docente.

```
package br.com.devmedia.gestaoacademica.model;
import javax.persistence.*;
import javax.validation.constraints.*;
import org.hibernate.validator.constraints.NotEmpty;

@Entity
@Table(name="DOCENTES")
public class Docente {

    @Id
    @Column(name="ID")
    @NotNull @GeneratedValue
    private Integer id;

    @Column(name="NOME")
    private String nome;

    @Column(name="MATRICULA")
    @NotEmpty
    @Size(min = 8, max = 16)
    private String matricula;

    @Column(name="TITULACAO")
    private String titulacao;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getMatricula() {
        return matricula;
    }

    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }

    public String getTitulacao() {
        return titulacao;
    }

    public void setTitulacao(String titulacao) {
        this.titulacao = titulacao;
    }
}
```

- **@Pattern(regex="regexp", flag=):** compara um valor com uma expressão regular;
- **@Range(min=, max=):** verifica se um valor numérico está entre o mínimo e o máximo.

Na próxima seção veremos como validar as restrições aplicadas na camada de modelo, a partir da camada de controle de nossa aplicação Web.

Validando as restrições aplicadas

Na seção anterior aplicamos algumas restrições na classe **Docente**, que faz parte da camada de modelo da nossa aplicação.

Hibernate Validator e a consistência dos dados

Considerando que nossa aplicação Web está sendo desenvolvida de acordo com o padrão MVC, vamos demonstrar a seguir o que deve ser feito nas camadas de controle e visualização para testar as regras estabelecidas nas restrições aplicadas na camada de modelo e exibir respostas para o usuário.

Camada de controle: verificando as restrições

A camada de controle será responsável por executar a validação, ou seja, verificar se as restrições estão sendo obedecidas e enviar respostas à camada de visualização, que as exibirá para o usuário. Sendo assim, vamos criar a classe **DocenteController** na camada de controle do nosso projeto. Para isso, com o botão direito sobre o pacote **br.com.devmedia.gestaoacademica.control**, selecione *New > Class*. No campo *Name*, digite “**DocenteController**”, como mostra a **Figura 4**. Para finalizar, clique em *Finish*.

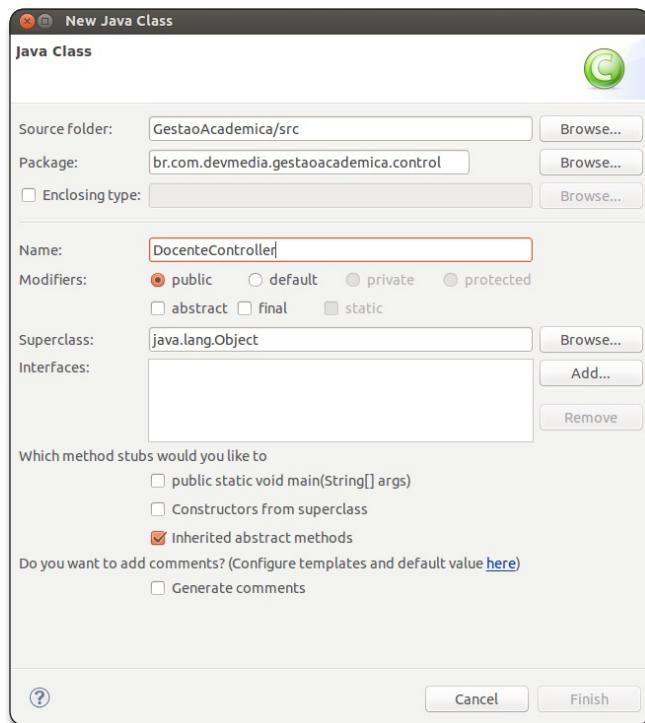


Figura 4. Criando a classe DocenteController na camada de controle

A classe **DocenteController**, cujo código fonte pode ser visto na **Listagem 3**, terá um único método de negócios – **adicionarDocente()** – que realiza a validação do objeto **Docente** e, caso as regras estejam sendo obedecidas, redireciona para uma página de resposta que informa que a operação foi realizada com sucesso – *sucesso.jsp*. Essa classe é anotada com **@Controller**, anotação do Spring MVC que “decora” a classe indicando que ela faz parte da camada de controle.

Da mesma forma, o método **form()** funciona como um mapeamento para o formulário de cadastro de docentes, que também recebe uma anotação do Spring MVC – **@RequestMapping**.

Listagem 3. Código fonte de DocenteController.

```
package br.com.devmedia.gestaoacademica.control;

import java.util.Map;

import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import javax.validation.Valid;

import br.com.devmedia.gestaoacademica.model.Docente;

@Controller
public class DocenteController {

    @RequestMapping("/form")
    public String form(Map<String, Object> map) {
        map.put("docente", new Docente());
        return "inserir_docente_form";
    }

    @RequestMapping(value = "/adicionar", method = RequestMethod.POST)
    public String adicionarDocente(@ModelAttribute("docente")
        @Valid Docente docente, BindingResult result) {

        if (result.hasErrors()) {
            return "inserir_docente_form";
        } else {
            return "sucesso";
        }
    }
}
```

Já no método **adicionarDocente()**, observe que a validação é acionada através da anotação **@Valid** – que faz parte do pacote **javax.validation** – que precede o objeto **docente** na assinatura do método **adicionarDocente()**. Isso significa que esse objeto será validado de acordo com as restrições que aplicamos na camada de modelo.

Com isso, a camada de controle da nossa aplicação de exemplo está pronta. Podemos partir, então, para a camada de visualização.

Camada de visualização: exibindo respostas para o usuário

Uma vez concluída a camada de controle, podemos partir para a implementação da camada de visualização de nossa aplicação de exemplo. Sendo assim, a camada de visualização terá duas páginas: um formulário para inserção de dados do docente e uma página de resposta que será exibida caso a operação de inserção seja bem sucedida. Nesse exemplo, o sucesso da operação significa que o preenchimento do formulário não violou as restrições que foram aplicadas à classe **Docente**.

Para criar a página JSP que corresponde ao formulário de cadastro de docentes, com o botão direito sobre o diretório **WEB-INF/views**, selecione *New > JSP File*. No campo *File Name*, digite “*inserir_docente_form.jsp*”, como mostra a **Figura 5**. Logo após, clique em *Finish* e o arquivo será criado.

Uma vez criado o arquivo do formulário, vamos criar três campos – **Nome**, **Matrícula** e **Titulação** – utilizando tags do Spring para gerar o código HTML. Como pode ser visto no código fonte na **Listagem 4**, utilizaremos a tag `<form:errors>`, com a propriedade `path` recebendo o valor `"**"` para exibir uma caixa com as todas as ocorrências de erro no topo da página. Da mesma forma, utilizamos a mesma tag, mas com a propriedade `path` recebendo `matricula`, para exibir ao lado do campo **Matrícula** a mensagem de erro correspondente.

Listagem 4. Código fonte de inserir_docente_form.jsp.

```

<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
<head>
<style>
.label_erro {
    color: #088A29;
}
.mensagem_erro {
    color: #000;
    background-color: #58FA82;
    border: 2px solid #088A29;
    padding: 8px;
    margin: 16px;
}
</style>
</head>
<title>Cadastro de Docentes</title>
</head>
<body>

<h3>Formulário de Cadastro de Docentes</h3>

<form:form method="post" action="adicionar.html" commandName="docente">
<form:errors path="**" cssClass="mensagem_erro" element="div"/>
<table>
<tr>
<td>Nome:</td>
<td><form:input path="nome"/></td>
</tr>
<tr>
<td>Matrícula:</td>
<td>
<form:input path="matricula"/>
<table>
<tr>
<td>
<form:errors path="matricula" cssClass="label_erro" />
</td>
</tr>
<tr>
<td>Titulação:</td>
<td><form:input path="titulacao"/></td>
</tr>
<tr>
<td colspan="2">
<input type="submit" value="Salvar"/>
</td>
</tr>
</table>
</form:form>
</body>
</html>

```

Como mostra a **Figura 6** o formulário pode ser acessado pelo navegador na URL `/GestaoAcademica/form`, como foi determinado no mapeamento (vide classe `DocenteController`, na **Listagem 3**).

Podemos testar seu funcionamento preenchendo os campos e clicando no botão `Salvar`. Por exemplo, se deixarmos o campo matrícula em branco, mensagens de erro de validação serão exibidas, como mostra a **Figura 7**.

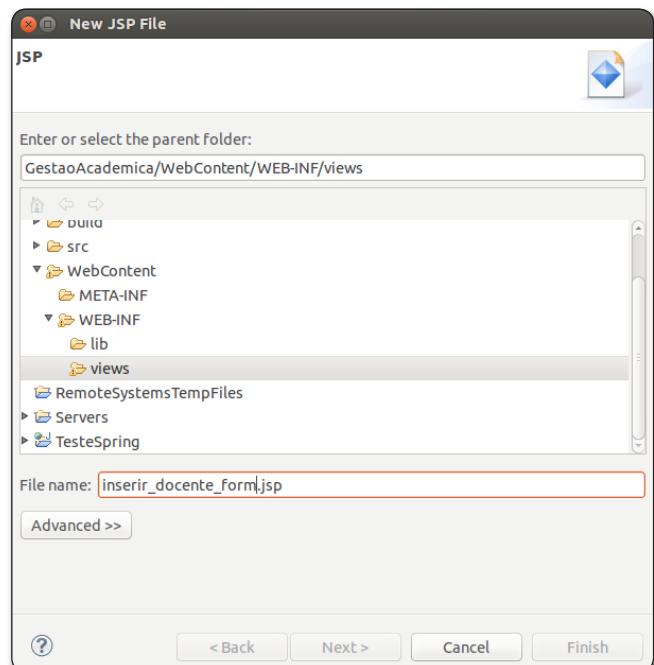


Figura 5. Criando o arquivo JSP para o formulário de cadastro de docentes

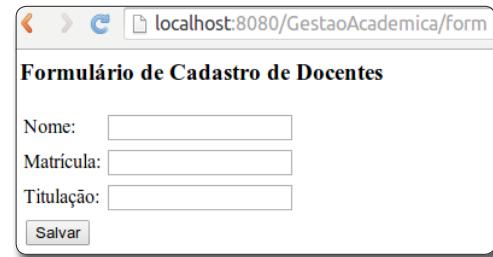


Figura 6. Formulário de cadastro de docentes visto no navegador

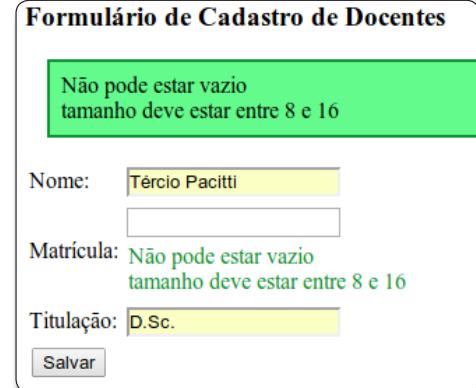


Figura 7. Mensagens de erro de validação sendo exibidas

Hibernate Validator e a consistência dos dados

Da mesma forma, caso as restrições sejam obedecidas – preenchendo o campo **Matrícula** com um valor que tenha entre 8 e 16 caracteres - o usuário é redirecionado para a página de sucesso demonstrada na **Figura 8**.

Como pudemos ver, a aplicação Web utilizando o Hibernate Validator está completa e pronta para uso. Na próxima seção, veremos como escrever restrições personalizadas, ou seja, como criar nossas próprias regras para não nos limitarmos às restrições existentes no catálogo de anotações do Hibernate Validator.



Figura 8. Página de sucesso

Desenvolvendo restrições personalizadas

Embora a biblioteca de restrições do Hibernate Validator seja bastante vasta, não precisamos nos limitar a ela. Dependendo do negócio de nossa aplicação, pode ser conveniente implementar nossas próprias regras. Por exemplo, imaginemos que o campo **Titulação** dos docentes deva estar obrigatoriamente em letras maiúsculas.

Para implementar essa restrição, vamos começar criando um pacote **br.com.devmedia.gestaoacademica.validator** que armazenará as classes e anotações de nossa validação personalizada. Feito isso, o primeiro passo para escrever a validação é criar uma forma de expressar as restrições que o campo receberá. No nosso caso, um campo do tipo **String** deverá receber caracteres somente maiúsculos.

Assim, com o botão direito sobre o pacote **validator**, selecione *New > Enum*. Como mostra a **Figura 9**, no campo *Name* digite “**CaseSense**”, que será o nome desse nosso **Enum**, e feito isso, clique em *Finish*.

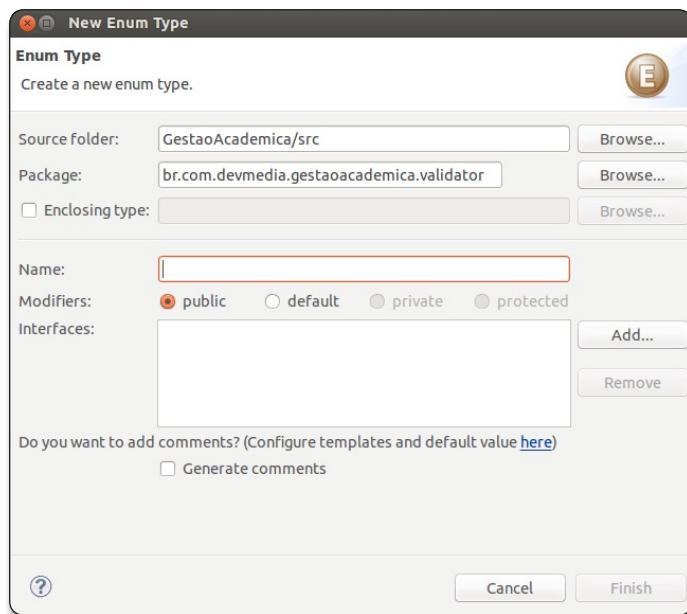


Figura 9. Criando a restrição

O código fonte do **Enum CaseSense** pode ser visto na **Listagem 5**.

Em seguida, precisamos escrever a anotação em si. Com o botão direito sobre o pacote **validator**, selecione *New > Annotation* e será mostrada a janela de propriedades da anotação, como apresenta a **Figura 10**. No campo *Name*, digite “**VerificaCase**” e então clique em *Finish*.

Listagem 5. Código fonte de CaseSense.

```
package br.com.devmedia.gestaoacademica.validator;  
  
public enum CaseSense {  
    UPPER, LOWER;  
}
```

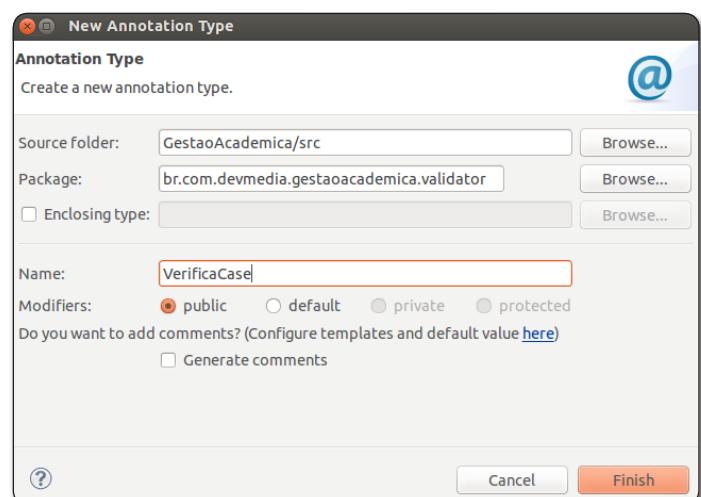


Figura 10. Criando a anotação

Como pode ser visto no código fonte descrito na **Listagem 6**, anotações são definidas através da palavra-chave **@interface** e, de acordo com a especificação da Java Bean Validation API, toda anotação deve ter um atributo **message** que retorne a mensagem (ou uma chave para que a mensagem seja encontrada em um arquivo de propriedades) para o caso da restrição ser violada; um atributo **groups**, que indica a que grupo sua restrição pertence – no nosso exemplo deixamos o valor **default**, que deixa a restrição no grupo raiz (vide **BOX 3**); um atributo **payload**, que permite que os clientes que utilizem essa restrição possam determinar o quanto severa uma violação pode ser (vide **BOX 4**) – também deixamos o valor **default** para esse atributo no nosso exemplo.

No código fonte de **VerificaCase**, a anotação **@Target** define que tipos de elementos podem ser validados com a anotação que estamos criando. No nosso caso, a anotação pode ser usada em atributos (**FIELD**), métodos (**METHOD**) e parâmetros (**PARAMETER**).

Já **@Retention(RetentionPolicy.RUNTIME)** determina que anotações desse tipo estarão disponíveis em tempo de execução, enquanto **@Constraint** “decora” a anotação como sendo uma restrição, que nesse caso será executada pelo validador **VerificaCaseValidator**, que escreveremos a seguir.

BOX 3. Agrupamento de Validações

Já a partir da JSR-303, as validações podem ser organizadas dentro de uma estrutura, compondo uma hierarquia de conjuntos e subconjuntos de restrições. Dessa forma, é possível definir quais restrições devem ser verificadas por cada validação, ou seja, as restrições de um subgrupo incluem as restrições de seu grupo – e assim sucessivamente.

BOX 4. Severidade de Validações

A severidade de uma violação a uma restrição é definida em seu atributo “payload”. Esse atributo é utilizado para associar metadados à declaração da restrição para indicar sua gravidade. Por exemplo, uma violação pode resultar em um “aviso” (por não ser tão grave), enquanto outras podem resultar em um “erro” (por serem mais graves).

Listagem 6. Código fonte de VerificaCase.

```
package br.com.devmedia.gestaooacademica.validator;

import java.lang.annotation.*;
import javax.validation.*;

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.
PARAMETER, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = VerificaCaseValidator.class)
@Documented
public @interface VerificaCase {

    String message() default "O campo é obrigatoriamente maiúsculo";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    CaseSense value();
    @Target({ElementType.FIELD, ElementType.METHOD,
    ElementType.PARAMETER, ElementType.ANNOTATION_TYPE})
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    @interface List {
        VerificaCase[] value();
    }
}
```

A anotação **@Documented** define que essa restrição aparecerá no *JavaDoc* das classes que a utilizarem.

O passo seguinte é criar o validador para nossa restrição. Desse modo, ainda no pacote **validator**, crie a classe **VerificaCaseValidator**, que implementa a interface **ConstraintValidator**, como mostra a **Listagem 7**.

Como pode ser visto, nesta classe, o método de retorno booleano **isValid()** é responsável por verificar se a restrição está sendo obedecida, retornando **true** em caso positivo e **false** em caso negativo. Feito isso, a restrição está pronta para ser utilizada e validada em nossa aplicação de exemplo.

Implementando e testando a restrição personalizada

Utilizar a restrição que acabamos de criar não é diferente de utilizar qualquer outra restrição do catálogo Java Bean Validation ou do Hibernate Validator. Para usá-la, basta acrescentar a anotação ao respectivo atributo na camada de modelo, como mostra a **Listagem 8**.

Para a camada de controle, a aplicação de uma nova restrição é completamente transparente – como recomenda o conceito DRY (vide **BOX 2**) –, não sendo necessário fazer qualquer tipo de modificação na classe **DocenteController**. Precisamos apenas acrescentar as linhas de código para que a eventual mensagem de erro apareça corretamente na página de formulário, como mostra a **Listagem 9**.

Listagem 7. Código fonte de VerificaCaseValidator.

```
package br.com.devmedia.gestaooacademica.validator;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class VerificaCaseValidator implements ConstraintValidator<VerificaCase,
String> {

    private CaseSense caseSense;

    @Override
    public void initialize(VerificaCase constraintAnnotation) {
        this.caseSense = constraintAnnotation.value();
    }

    @Override
    public boolean isValid(String object, ConstraintValidatorContext constraint
Context) {
        if (caseSense == CaseSense.UPPER) {
            return object.equals(object.toUpperCase());
        } else {
            return object.equals(object.toLowerCase());
        }
    }
}
```

Listagem 8. Acrescentando a restrição personalizada à classe Docente.

```
...
@Column(name="TITULACAO")
@VerificaCase(CaseSense.UPPER)
private String titulacao;
```

Listagem 9. Acrescentando a eventual mensagem de erro ao formulário de cadastro de docentes.

```
...
<tr>
    <td>Titulação:</td>
    <td>
        <form:input path="titulacao"/> <form:errors path="titulacao"
cssClass="label_erros"/>
    </td>
</tr>
...
```

Podemos, então, testar novamente nossa aplicação para verificar se a nova restrição foi corretamente construída e está sendo perfeitamente aplicada e verificada. Para tanto, digitemos valores em minúsculo no campo *Titulação* e cliquemos em *Salvar* (ver **Figura 11**).

Hibernate Validator e a consistência dos dados

Formulário de Cadastro de Docentes

O campo é obrigatoriamente maiúsculo

Nome:

Matrícula:

Titulação: O campo é obrigatoriamente maiúsculo

Figura 11. Testando a restrição personalizada e sua validação

Como podemos verificar, o campo *Titulação* passa a exigir obrigatoriamente valores maiúsculos, exibindo uma mensagem de erro na página quando essa restrição não é obedecida.

Garantir a consistência dos dados é uma necessidade fundamental de todo desenvolvedor. Por mais que tentemos deixar claro na interface das nossas aplicações as características dos dados que desejamos receber, deixar essa tarefa nas mãos do usuário pode ser perigoso, uma vez que este não tem nenhum compromisso com a manutenção e nem mesmo com o funcionamento do software.

No entanto, mais importante do que escrever validações é entender a tênue distinção entre a necessidade de manter a consistência dos dados – por meio de regras e restrições – e implementar o propósito da aplicação – por meio das regras de negócio. A melhor forma de compreender essa distinção, por mais limitadora que seja, é se concentrar no formato dos dados. Em geral, a consistência dos dados está relacionada ao seu formato, enquanto as regras de negócio, também de forma geral, estão relacionadas a fluxos que a aplicação deve respeitar.

O desenvolvedor deve, portanto, ter o cuidado para não manter regras de negócio encapsuladas no lugar errado, assim como regras de validação que, de uma maneira ou de outra, prendem esses objetos a tais implementações.

Além de tudo o isso, o Hibernate Validator ainda traz muito a ser explorado como, por exemplo, a integração com as APIs de validação do Spring Framework ou de outros *frameworks* para desenvolvimento Web ou MVC. Da mesma forma, recursos mais simples podem ser explorados, como a personalização de mensagens de erro ou a determinação de níveis de severidade para cada violação de restrição.

Todas essas ferramentas podem ser utilizadas não só para garantir a consistência de sua aplicação Web, como uma melhor experiência para o usuário. O que devemos fazer é simplificar a interação do usuário com o sistema, viabilizando mecanismos para que os dados fornecidos estejam dentro das regras estabelecidas e da maneira mais transparente possível para que o software funcione perfeitamente. O Hibernate Validator tem esse papel e pode nos ajudar a fazer isso de maneira simples e elegante.

Autor



Alessandro Jatobá

jatoba@jatoba.org

É Mestre em Ciéncia da Computação pela UFRJ, Doutorando em Engenharia também pela UFRJ com Doutorado-Sanduíche pelo Departamento de Engenharia e Projeto de Sistemas da Universidade de Waterloo, no Canadá. Trabalha com Java há mais de 15 anos, a maior parte deles liderando equipes de desenvolvedores em projetos Web. Também é professor universitário lecionando temas ligados ao desenvolvimento orientado a objetos.



Links:

Site oficial do Hibernate.

<http://www.hibernate.org>

Site oficial da Java Community Process.

<http://www.jcp.org>

Site oficial do Spring Framework.

<http://www.spring.io>

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

Ele está trabalhando em seu saque e não na sua nova aplicação.



Mais de 95%* dos times de desenvolvimento e testes de aplicativos relatam tempos de espera e atrasos para acessar os sistemas que necessitam para realizar suas atividades. A Virtualização de Serviços elimina estas dependências gerando ambientes simulados similares à realidade, permitindo o desenvolvimento em paralelo. Isto significa que suas aplicações serão lançadas mais rapidamente, com maior qualidade e menor custo. Game over!

Conheça mais em ca.com/br/GoDevOps

ca
technologies

*De acordo com o estudo 2012 North America/Europe Service Virtualization Study, Coleman-Parkes.