



Edição 53



**DESAFIO:** Como criar um CRUD com web services  
Aprenda a implementar serviços RESTful com Jersey, JPA e MySQL

**Programando com exceções**  
Saiba como capturar,  
tratar e lançar exceções

# PRIMEIROS PASSOS COM JAVASERVER FACES

Aprenda a criar  
aplicações web  
ricas e flexíveis



# MVP

R\$ 1.000.000,00  
INVESTIDOS EM CONTEÚDO  
NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO  
NA SUA CARREIRA...

E MOSTRE AO MERCADO  
QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL  
QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's  
consumido + de **500.000** vezes

POR APENAS  
**R\$ 69,90\*** mensais

\*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR  
MAIS DO MERCADO!



 **DEV MEDIA**

# Sumário

## 06 - Aprenda a trabalhar com exceções no Java

[ John Soldera ]

Conteúdo sobre Boas Práticas, Artigo no estilo Solução Completa

## 16 - Primeiros passos com JavaServer Faces

[ Alessandro Jatobá ]

Artigo no estilo Solução Completa

## 26 - Criando um CRUD RESTful com Jersey, JPA e MySQL

[ Madson Aguiar Rodrigues ]

## 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](http://www.devmedia.com.br/curso/javamagazine)

(21) 3382-5038



Edição 53 • 2015 • ISSN 2173625-4



Assine agora e tenha acesso a todo o conteúdo da DevMedia:  
[www.devmedia.com.br/mvp](http://www.devmedia.com.br/mvp)

## EXPEDIENTE

### Editor

Eduardo Spínola ([eduspinola@gmail.com](mailto:eduspinola@gmail.com))

**Consultor Técnico** Diogo Souza ([diagosouzac@gmail.com](mailto:diagosouzac@gmail.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](http://www.devmedia.com.br/central), ou se preferir entre em contato conosco através do telefone 21 3382-5038.

### Publicidade

[publicidade@devmedia.com.br](mailto: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](http://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



**DEVMEDIA**



## 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](http://twitter.com/toolscloud)



**ToolsCloud**  
[toolscloud.com](http://toolscloud.com)

# Aprenda a trabalhar com exceções no Java

**Veja neste artigo como gerenciar o fluxo de erros na sua aplicação através do controle de exceções**

**U**ma exceção é um problema (erro) que ocorre durante a execução de um programa. Como sabemos, em qualquer programa de computador existem diversos motivos para a ocorrência de erros. Alguns dos mais comuns são destacados a seguir:

- O usuário entrou com dados inválidos na aplicação;
- Um arquivo não foi encontrado no momento da sua abertura;
- Foi acessado um arquivo que está em branco e era esperado que tivesse conteúdo;
- Erro de conversão de **String** para **float**, devido ao formato numérico incompatível;
- Queda de conexão com a internet durante a transmissão de dados por uma stream;
- Divisão de um número por zero.

A exceção, de modo simples, pode ser entendida como uma forma comum de informar a ocorrência de uma condição anormal no programa, ou seja, uma situação de erro. Quando acontece um problema em um método e o mesmo não pode resolver sozinho, ele pode lançar uma exceção, sinalizando e descrevendo o erro. Consequentemente, essa exceção pode receber tratamento próprio em outro ponto do programa. Entretanto, caso não seja tratada, o programa finalizará prematuramente e o método **main()** imprimirá um resumo da exceção na tela.

Como você poderá constatar, muitas linguagens de programação não apresentam o tratamento de exceções nativamente. Portanto, cada método ou função precisa disponibilizar algum tipo de tratamento, o que aumenta a complexidade da tarefa de codificação, pois são usados códigos de erros distintos, retornos nulos, *flags* e outras situações, sem haver um padrão. Por outro lado, o Java oferece uma abordagem simplificada, permitindo tratar qualquer tipo de ocorrência de erro através de uma hierarquia de classes de erros, onde a principal é a classe **Exception**. Ademais, permite ao programador definir

## Fique por dentro

Este artigo apresenta em detalhes como é realizado o tratamento de exceções na linguagem Java. Para isso, com exemplos, abordaremos os mecanismos que podem ser empregados para o controle de erros, mecanismos estes que simplificam o trabalho do programador no tratamento das diferentes condições adversas que ocorrem no decorrer da implementação de programas. Além disso, veremos como é possível criar exceções personalizadas, voltadas para necessidades específicas de acordo com as regras de negócio definidas na aplicação.

novos tipos de exceção, personalizados, que podem ser usados pela aplicação.

Como veremos no decorrer do artigo, o principal objetivo do gerenciamento de exceções é, no momento de um erro, transferir o controle do programa para um trecho de código que possa lidar com a exceção, realizando o tratamento do mesmo de forma conveniente. O código responsável por esse “reparo” é chamado de tratador de exceção (*exception handler*), visto que oferece uma solução para o erro ocorrido. Assim, durante a atividade de programação em Java, deve-se posicionar tratadores de exceção “estrategicamente” no código para que o programa possa capturar e resolver qualquer exceção possível de ser lançada, evitando o encerramento inesperado do aplicativo.

## Resposta a uma exceção

Considerando a ocorrência de uma exceção em um programa, o mesmo pode tratá-la de diferentes formas. A seguir analisamos as opções:

- **Ignorar a exceção:** Dessa forma, o programa terminará anormalmente ou produzirá resultados incorretos;
- **Atribuir a resolução do problema ao usuário:** Como o usuário não é programador, é muito provável que ele não saberá solucionar o problema;
- **Tratar o erro no programa:** O programador desenvolve a aplicação de forma que, quando o erro ocorrer, é acionado um bloco de código responsável pelo tratamento do erro.

É claro que a melhor solução é fazer o programa tratar a exceção, e se não houver uma solução disponível, deve-se notificar o usuário do ocorrido para que ele possa tomar alguma atitude. Por exemplo, em uma aplicação de uma grande corporação, na ocorrência de erros, é importante mostrar para o usuário uma mensagem apropriada, oferecendo possíveis soluções. Caso o erro não seja reparável, a sua ocorrência pode ser armazenada em um registro para consulta futura.

### A hierarquia de classes das exceções

Como em tudo em Java, criar uma exceção significa criar um objeto. Portanto, quando um programa lança uma exceção, ele está lançando um objeto, derivado de qualquer subclasse de **Throwable**, a classe mais alta na hierarquia das exceções. **Throwable** representa qualquer tipo de erro que pode acontecer em uma aplicação e se divide em duas categorias: **Exception** e **Error**.

A superclasse da maioria das exceções é a classe **Exception**. Esta sinaliza a ocorrência de erros comuns, que podem acontecer em qualquer aplicação, e está relacionada a erros que podem ser resolvidos com o devido tratamento, como um arquivo não encontrado, um arquivo em branco, divisão por zero, conversão inválida de classe, acesso a ponteiro nulo, conversão numérica inválida, fechamento inesperado de uma **stream** de dados, etc.

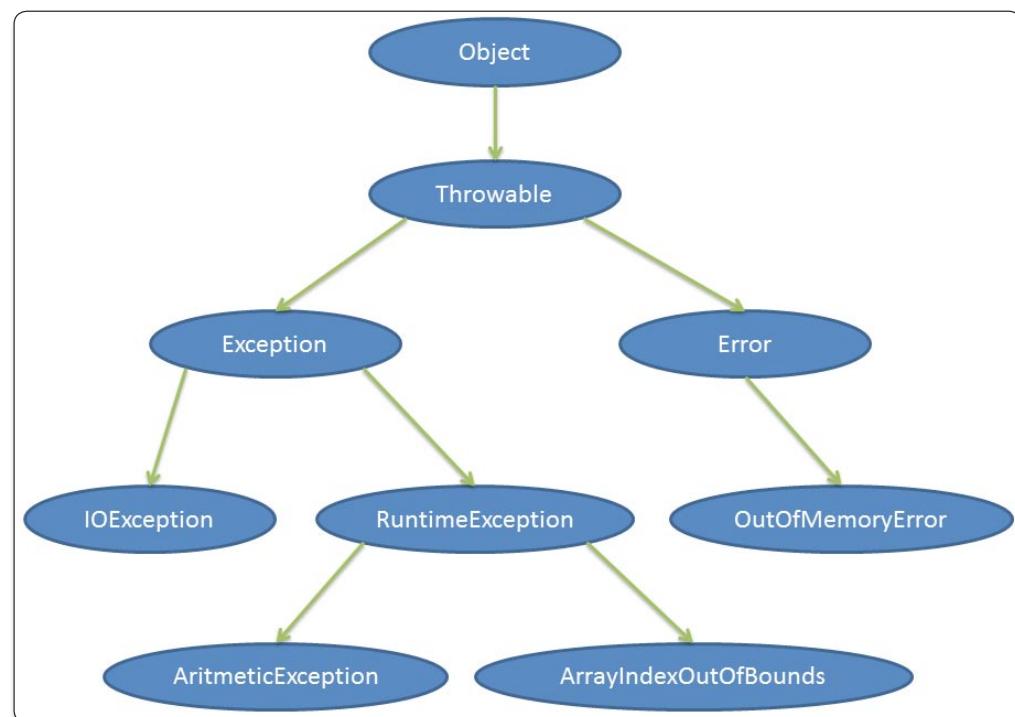
Por outro lado, **Error** indica um tipo mais rigoroso de erro, onde geralmente não é possível tratá-lo e nem continuar a execução do programa. Enquadram-se nessa categoria os erros de falta de memória, classe não encontrada, classe inválida, erro interno da máquina virtual, etc.

Ao lançar uma exceção em Java é criada uma instância de uma subclasse de **Throwable**, como **Exception**, **Error** ou suas subclasses. Cada tipo de exceção tem um papel específico, sendo necessário apenas instanciar e lançar a exceção do tipo correto (a que corresponde ao erro ocorrido). Uma parte da hierarquia de classes de exceções pode ser verificada na **Figura 1**.

### A classe **Throwable**

Com relação à classe **Throwable**, somente seus objetos (ou subclasses de **Throwable**) podem ser lançados pela máquina virtual Java na ocorrência de erros ou pelo programador, usando o comando **throw**. Os principais construtores da classe **Throwable** são:

- **public Throwable():** Constrói um objeto **Throwable** sem mensagem de erro;



**Figura 1.** Hierarquia de exceções da linguagem Java

- **public Throwable(String mensagem):** Constrói um objeto **Throwable** com a mensagem de erro informada por parâmetro.

Além disso, a classe **Throwable** possui uma série de métodos auxiliares, que são herdados pelas suas subclasses. São eles:

- **public String getMessage():** Retorna uma mensagem sobre o erro ou nulo se esse objeto não tem uma mensagem;
- **public String getLocalizedMessage():** Retorna uma mensagem relacionada ao **Locale** específico da máquina virtual Java. Como um **Locale** indica uma região política, geográfica ou cultural, será retornada uma mensagem no idioma (ou dialeto) onde encontra-se a JVM. A implementação **default** desse método retorna o mesmo que **getMessage()**, porém ambos podem ser sobreescritos pelas classes filhas de **Throwable**;
- **public String toString():** Retorna uma descrição curta desse objeto;
- **public void printStackTrace():** Imprime na saída padrão de erro a pilha de execução de métodos correspondente ao método que criou e lançou a exceção, chamada de **stacktrace**;
- **public void printStackTrace(PrintStream):** Imprime na stream indicada a pilha de execução de métodos correspondente ao método que criou e lançou a exceção;
- **public native Throwable fillInStackTrace():** Preenche a **stacktrace**. Esse método é usado para definir uma nova **stacktrace** quando a exceção for relançada, sendo útil em situações onde se deseja capturar uma exceção vinda de um determinado método e relançá-la como sendo de outro método.

## A classe Error

Por sua vez, **Error** define as situações de erro geralmente internas à máquina virtual, não sendo recomendado tentar capturar exceções desse tipo, visto que são erros irrecuperáveis. Os principais construtores de **Error** são:

- **Error():** Cria um erro sem mensagem;
- **Error(String mensagem):** Cria um erro com uma mensagem;
- **Error(String mensagem, Throwable causa):** Cria um erro com uma mensagem informando uma exceção como a causa desse erro;
- **Error(Throwable causa):** Cria um erro a partir de um **Throwable**.

Quanto a métodos, **Error** não define nenhum novo. Apenas herda todos os métodos de **Throwable**. Além disso, mesmo sendo possível capturar um **Error**, não será de grande valia para a aplicação tratá-lo, pois a situação corrente do programa é irrecuperável. Nesse caso, o ideal é finalizar a aplicação e esperar o usuário reiniciá-la. As principais classes filhas de **Error** são:

- **VirtualMachineError:** Esse erro acontece quando a máquina virtual fica sem recursos ou quando entra em um estado de erro. É irrecuperável e o programa finaliza;
- **ClassFormatError:** Esse erro é lançado quando a máquina virtual tenta carregar uma classe e a classe contém algum erro na sua definição que a impede de ser carregada ou ser interpretada como sendo uma classe;
- **NoClassDefFoundError:** Este erro é lançado quando a máquina virtual tenta carregar uma classe e a definição da classe não é encontrada na lista de definições de classes da JVM. Uma situação em que esse erro pode surgir é na falta de alguma dependência em algum momento da execução;
- **OutOfMemoryError:** Esse erro é lançado pela máquina virtual quando não é possível mais alocar memória para criar um novo objeto.

## A classe Exception

A outra classe filha de **Throwable** é **Exception**, usada para representar exceções comuns na execução de uma aplicação. As suas subclasses indicam cada possível tipo de exceção, sendo possível tratá-las com sucesso durante a execução do programa a partir da definição de um bloco de código para tal. Os principais construtores da classe **Exception** são:

- **Exception():** Cria uma exceção sem mensagem;
- **Exception (String mensagem):** Cria uma exceção com uma mensagem;
- **Exception (String mensagem, Throwable causa):** Cria uma exceção com uma mensagem informando outra exceção como a causa do erro;
- **Exception (Throwable causa):** Cria uma exceção a partir de um **Throwable**.

De modo semelhante a **Error**, **Exception** não define nenhum método novo. Apenas herda todos os métodos de **Throwable**.

Outra definição importante é que existem duas categorias de exceções (**Exception**) na linguagem Java. A primeira se refere às

exceções verificadas (*Checked Exceptions*). Com base nela, se um método declara que pode lançar uma exceção verificada, qualquer código que chame esse método deve, obrigatoriamente, incluir o tratamento para a exceção.

Por outro lado, não é obrigatório incluir no código o tratamento para exceções que sejam não-verificadas (*Unchecked Exceptions*). Entretanto, se uma exceção não-verificada for lançada e não for tratada, ela vai causar o término do programa como qualquer outra exceção.

## Exceções verificadas

Uma exceção verificada é, por definição, uma subclasse de **Exception**, porém a mesma não deve ser subclasse de **RuntimeException** – que também herda de **Exception** – por esta representar a exceção não-verificada. Algumas das principais exceções verificadas do Java são:

- **InstantiationException:** Esta exceção é lançada quando a aplicação tenta criar uma instância de uma classe, porém ela não pode ser instanciada porque é uma interface ou uma classe abstrata;
- **OException:** Esta exceção pertence ao pacote **java.io** e serve para indicar um erro de comunicação em uma stream de dados. Pode ser lançada durante a leitura e/ou escrita em arquivos;
- **ClassNotFoundException:** Esta exceção representa o erro de classe não encontrada. É lançada quando o **ClassLoader** é usado explicitamente para criar uma instância de uma classe e o arquivo da classe não existe;
- **CloneNotSupportedException:** É lançada quando o método **clone()** é chamado por um objeto derivado de uma classe que não implementa **Cloneable**. Para clonar um objeto é necessário que este implemente **Cloneable**;
- **InterruptedException:** É lançada quando uma thread, em espera, é interrompida por outra thread através da chamada ao método **interrupt()**;
- **IllegalAccessException:** Exceção lançada quando se tenta acessar uma classe da qual não se tem acesso, por exemplo, uma classe não visível.

## Exceções não-verificadas

Como visto anteriormente, todas as exceções não-verificadas são filhas de **RuntimeException**, que apresenta os mesmos métodos e construtores de **Exception**. Algumas das principais exceções não-verificadas são:

- **IndexOutOfBoundsException:** Esta exceção é criada quando uma aplicação tenta acessar um índice em um vetor, **array** ou **string** que esteja fora do intervalo de índices válidos;
- **ArithmaticException:** Exceção lançada quando ocorre uma condição anormal em uma operação aritmética, como divisão por zero;
- **ClassCastException:** Esta exceção ocorre quando um código tenta converter um objeto para um tipo que não é válido na hierarquia de classes. Por exemplo, ao tentar converter um **Integer** para uma **String**;

- **NullPointerException**: Exceção lançada na tentativa de acessar um campo de uma referência nula;
- **NumberFormatException**: Erro na conversão de uma **String** para um formato numérico.

## Os comandos try-catch

Agora que já vimos as principais exceções e sua hierarquia, podemos focar nos comandos responsáveis pela manipulação destas, a iniciar pelo **try-catch**, que é usado para capturá-las e tratá-las. A cláusula **try** é empregada para definir um bloco de código a ser monitorado quanto à ocorrência de exceções. Se alguma exceção ocorrer, a cláusula **catch** apropriada será chamada. Um exemplo de uso de **try-catch** é apresentado na **Listagem 1**.

**Listagem 1.** Exemplo com try-catch.

```
01. public class Teste1 {
02.   public static void main(String args[]){
03.     try{
04.       int vet[] = new int[10];
05.       System.out.println("Acessando elemento 20 :" + vet[20]);
06.     }catch(ArrayIndexOutOfBoundsException e){
07.       System.out.println("Exceção Lançada :" + e.getMessage());
08.     }
09.   }
10. }
```

Como podemos verificar, o comando **try** da linha 3 cria um bloco de código onde será monitorada a ocorrência de exceções – veja esse bloco nas linhas 4 e 5. Como consequência, se ocorrer um erro a cláusula **catch** (na linha 6) será executada, realizando o tratamento do mesmo.

É regra que toda a cláusula **catch** informe uma classe de exceção, que indica o tipo de exceção que será capturada por esse **catch** se ocorrer um erro no bloco especificado pelo **try**. Como consequência, o corpo do **catch** (linhas 6 a 8) define o tratamento para a possível exceção.

Analizando o código do bloco **try**, pode-se notar que na linha 4 é criado um **array** com 10 posições. No entanto, na linha 5 é acessada a vigésima posição deste **array**. Como esse índice é inválido, será lançada uma **ArrayIndexOutOfBoundsException** pela máquina virtual e esta será capturada pelo **catch** correspondente.

No bloco de código especificado pelo **catch**, realizamos o tratamento desta exceção. No entanto, se outro tipo de erro ocorrer, ele não será capturado, pois o **catch** captura apenas **ArrayIndexOutOfBoundsException**.

Outra regra sobre o comando **catch** é que ele captura qualquer subclasse da exceção informada, ou seja, é possível capturar uma determinada exceção e todas as exceções definidas na sua hierarquia de subclasses como, por exemplo, **Exception** e todas as suas subclasses, como exemplificado na **Listagem 2**.

Observe que na linha 6 é usado o comando **catch(Exception e)**, garantindo assim que **Exception** (e qualquer subclass da mesma) seja capturada. Outra regra sobre o uso do **catch** é que mais cláusulas **catch** podem ser incorporadas. Neste momento vale ressaltar

que a ordem de declaração deve ser da menos especializada para a mais especializada, como mostrado na **Listagem 3**.

**Listagem 2.** Exemplo de try-catch com Exception.

```
01. public class Teste2 {
02.   public static void main(String args[]){
03.     try{
04.       int vet[] = new int[10];
05.       System.out.println("Acessando elemento 20 :" + vet[20]);
06.     }catch(Exception e){
07.       System.out.println("Exceção Lançada :" + e.getMessage());
08.     }
09.   }
10. }
```

**Listagem 3.** Exemplo de try-catch com vários catches.

```
01. class Teste3 {
02.   public static void main(String args[]){
03.     try{
04.       int a[] = new int[10];
05.       System.out.println("Acessando elemento 20 :" + a[20]);
06.     }catch(NullPointerException e){
07.       System.out.println("NullPointerException :" + e.getMessage());
08.     }catch(ArrayIndexOutOfBoundsException e){
09.       System.out.println("ArrayIndexOutOfBoundsException :" +
10.         + 10. e.getMessage());
11.     }catch(RuntimeException e){
12.       System.out.println("RuntimeException :" + e.getMessage());
13.     }catch(Exception e){
14.       System.out.println("Exception :" + e.getMessage());
15.     }catch(Error e){
16.       System.out.println("Error :" + e.getMessage());
17.     }catch(Throwable e){
18.       System.out.println("Throwable :" + e.getMessage());
19.     }
20.   }
21. }
```

Esse código apresenta múltiplas ocorrências de **catch**, ordenados de acordo com o nível de especialização de cada exceção a ser capturada. Observe que **NullPointerException** e **ArrayIndexOutOfBoundsException** vêm antes de **RuntimeException**, exatamente por serem de tipos mais especializados. Por fim, **Throwable** é posicionado sempre como sendo o último **catch**, vindo depois de **Exception** e **Error**, suas subclasses mais diretas.

Outra explicação importante sobre múltiplos **catches** é a escolha do **catch** a ser executado quando ocorrer uma exceção. Neste momento, é escolhido o primeiro **catch** que é apto a capturar a exceção (considerando a ordem dos **catches** no código). Um exemplo desta regra é que a exceção **NullPointerException**, além de poder ser capturada normalmente por um **catch** com **NullPointerException**, também pode ser capturada por um **catch** com **RuntimeException**, **Exception** ou **Throwable**, pois as mesmas são suas superclasses. Vale lembrar que um **catch** que captura **Error** não capturará **NullPointerException**, pois o mesmo está fora da hierarquia de classes da mesma.

## O comando finally

Assim como **catch**, o **finally** é usado para criar um bloco de código ligado a um **try**. Porém, o **finally** é sempre executado,

independentemente de ter sido lançada ou não uma exceção. Dessa forma, este comando é muito útil em casos em que é aberta uma conexão para transmissão de dados, deixando a cargo do **finally** o fechamento da conexão.

Uma observação importante é que quando um **try** possui um **finally** o **catch** é opcional, ou seja, **try-finally** é válido. Um exemplo de código com **finally** é apresentado na **Listagem 4**, onde há um bloco **try** e dois **catches**, sendo um para cada possível exceção que pode ocorrer (**FileNotFoundException** ou **IOException**).

**Listagem 4.** Exemplo de try-catch-finally.

```
01. import java.io.File;
02. import java.io.FileInputStream;
03. import java.io.FileNotFoundException;
04. import java.io.FileOutputStream;
05. import java.io.IOException;
06.
07. public class Teste4 {
08.     public static void copiarArquivo(String origem, String destino) {
09.         FileInputStream in = null;
10.         FileOutputStream out = null;
11.         try {
12.             File f1 = new File(origem);
13.             File f2 = new File(destino);
14.             in = new FileInputStream(f1);
15.             out = new FileOutputStream(f2);
16.             byte[] buf = new byte[1024];
17.             int len;
18.             while ((len = in.read(buf)) > 0) {
19.                 out.write(buf, 0, len);
20.             }
21.         }
22.         catch (FileNotFoundException e) {
23.             System.out.println("FileNotFoundException:" + e.getMessage());
24.         }
25.         catch (IOException e) {
26.             System.out.println("IOException:" + e.getMessage());
27.         }
28.         finally {
29.             if (in != null)
30.                 try {
31.                     in.close();
32.                 }
33.                 catch (Exception e){}
34.             if (out != null)
35.                 try {
36.                     out.close();
37.                 }
38.                 catch (Exception e){}
39.         }
40.     }
41.     public static void main(String[] args){
42.         copiarArquivo("teste.txt","teste2.txt");
43.     }
44. }
```

Neste código é aberta uma stream para um arquivo de entrada (linha 12) e uma stream para um arquivo de saída (linha 13). Em seguida, os dados são copiados do arquivo de entrada para o arquivo de saída em blocos de 1024 bytes (linhas 18 a 20). Se algum arquivo não for encontrado, será acionado o **catch** na linha 22 e se ocorrer erro na comunicação, será acionado o **catch** na linha 25. Por fim, é declarado um bloco **finally** (linhas 28 a 39), que é responsável por fechar as conexões abertas anteriormente.

Como o **finally** é sempre executado e pode ser que algum erro aconteça na criação das streams, o que resulta nas variáveis **in** e **out** estarem nulas (ou apenas uma delas), é necessário verificar se elas não são nulas no **finally** (linhas 29 e 34) antes de fechar as streams. Além disso, como o método **close()** de ambas as streams também lança uma exceção, é preciso capturá-la usando mais um **try** (linhas 30 e 35), e no **catch** correspondente nada é feito por simplicidade.

Por fim, nas linhas 41 a 43, verifica-se que o método **main()** chama o método **copiarArquivo()**. Note, no entanto, que não é necessário definir um **try**, pois o método **copiarArquivo()** contém o tratamento das exceções que podem ocorrer no seu processamento.

## Os comandos **throw** e **throws**

Outro comando para manipular exceções é o **throw**, sendo usado para lançar uma exceção e sinalizar que houve erro no processamento no método que a lançou. Por sua vez, o comando **throws** é usado para omitir o tratamento de uma exceção em um método, delegando o tratamento do erro para o código externo, o que chamou essa função.

Na **Listagem 5** é apresentado um código com uma funcionalidade equivalente ao exemplo da **Listagem 4**, porém as exceções são capturadas e relançadas, sendo assim delegado o tratamento dos erros ao método chamador (**main()**).

Note que o método **copiarArquivo()** possui um **try**, dois **catches** e um **finally**. No entanto, se ocorrer uma exceção na execução do bloco **try**, o erro será capturado em um **catch** (linhas 23 a 29) e relançado para o método que chamou esse método (**copiarArquivo()**) usando o comando **throw** (linhas 25 ou 29), ou seja, o erro será lançado para o método **main()**, que obrigatoriamente define um **try-catch** para definitivamente tratar o erro, conforme verificado nas linhas 45 a 54.

Vale ressaltar ainda que esse código não compilaria sem o uso da cláusula **throws** (linha 9), pois como o método **copiarArquivo()** lança exceções (executa **throw**), ele pode lançar uma exceção do tipo **FileNotFoundException** ou **IOException**, sendo necessário declarar isso explicitamente através da cláusula **throws** para informar ambas as exceções.

Por causa disso, qualquer método que chamar **copiarArquivo()** deve especificar um **try-catch** para tratar a ocorrência das mesmas exceções ou ainda pode usar o mesmo procedimento com **throws**, deixando o tratamento do erro para o outro método chamador.

Na **Listagem 6** é apresentado um exemplo mais simples, mas com a mesma funcionalidade. Observe que não é necessário relançar a exceção com **throw**, uma vez que não existe **catch**, porém, como o erro não é capturado e tratado, implica-se no emprego do **throws**, pois esse método lança uma ou mais exceções.

Mesmo sem **catch** esse código compila normalmente, pois ele tem uma cláusula **finally** e sinaliza que lança exceções (linha 9), delegando para o método que o chama tratar as exceções, como visto nas linhas 36 a 46, onde o método **main()** usa um **try** para capturar cada um dos erros declarados no **throws**.

#### Listagem 5. Exemplo com throw e throws.

```
01. import java.io.File;
02. import java.io.FileInputStream;
03. import java.io.FileNotFoundException;
04. import java.io.FileOutputStream;
05. import java.io.IOException;
06.
07. public class Teste5 {
08.     public static void copiarArquivo(String origem, String destino)
09.             throws IOException, FileNotFoundException {
10.         FileInputStream in = null;
11.         FileOutputStream out = null;
12.         try {
13.             File f1 = new File(origem);
14.             File f2 = new File(destino);
15.             in = new FileInputStream(f1);
16.             out = new FileOutputStream(f2);
17.             byte[] buf = new byte[1024];
18.             int len;
19.             while ((len = in.read(buf)) > 0) {
20.                 out.write(buf, 0, len);
21.             }
22.         }
23.         catch (FileNotFoundException e) {
24.             System.out.println("FileNotFoundException: " + e.getMessage());
25.             throw e;
26.         }
27.         catch (IOException e) {
28.             System.out.println("IOException: " + e.getMessage());
29.         throw e;
30.     }
31.     finally {
32.         if (in != null)
33.             try {
34.                 in.close();
35.             }
36.             catch (Exception e){}
37.         if (out != null)
38.             try {
39.                 out.close();
40.             }
41.             catch (Exception e){}
42.     }
43. }
44. public static void main(String[] args){
45.     try {
46.         copiarArquivo("teste.txt","teste2.txt");
47.     }
48.     catch (FileNotFoundException e) {
49.         System.out.println("FileNotFoundException: " + e.getMessage());
50.     }
51.     catch (IOException e) {
52.         System.out.println("IOException: " + e.getMessage());
53.     }
54. }
55. }
```

#### Listagem 6. Exemplo com throw e throws simplificado.

```
01. import java.io.File;
02. import java.io.FileInputStream;
03. import java.io.FileNotFoundException;
04. import java.io.FileOutputStream;
05. import java.io.IOException;
06.
07. public class Teste6 {
08.     public static void copiarArquivo(String origem, String destino)
09.             throws IOException, FileNotFoundException {
10.         FileInputStream in = null;
11.         FileOutputStream out = null;
12.         try {
13.             File f1 = new File(origem);
14.             File f2 = new File(destino);
15.             in = new FileInputStream(f1);
16.             out = new FileOutputStream(f2);
17.             byte[] buf = new byte[1024];
18.             int len;
19.             while ((len = in.read(buf)) > 0) {
20.                 out.write(buf, 0, len);
21.             }
22.         }
23.         finally {
24.             if (in != null)
25.                 try {
26.                     in.close();
27.                 }
28.                 catch (Exception e){}
29.             if (out != null)
30.                 try {
31.                     out.close();
32.                 }
33.                 catch (Exception e){}
34.         }
35.     }
36.     public static void main(String[] args){
37.         try {
38.             copiarArquivo("teste.txt","teste2.txt");
39.         }
40.         catch (FileNotFoundException e) {
41.             System.out.println("FileNotFoundException: " + e.getMessage());
42.         }
43.         catch (IOException e) {
44.             System.out.println("IOException: " + e.getMessage());
45.         }
46.     }
47. }
```

## Criando uma exceção customizada

Existem situações em que é necessário criar classes de exceções personalizadas de acordo com a aplicação, de forma a adicionar novas opções ao tratamento padrão de exceções. Como exemplo, pode-se adicionar um novo campo, que representa o código do erro, sendo controlado pela aplicação, ou ainda adicionar informações mais detalhadas sobre o erro, como uma mensagem auxiliar.

Uma exceção pode ser definida através da criação de uma subclasse de **Throwable** ou **Exception** (mais comum), de modo a incluir novos atributos e métodos. Uma subclasse de **RuntimeException** será, consequentemente, uma exceção não-verificada. Um exemplo de exceção personalizada é apresentado na **Listagem 7**, que define um campo adicional **codigoErro** e uma **mensagemAuxiliar**, além da própria mensagem da classe **Exception**, pois em uma aplicação com controle mais avançado de erros, pode

ser útil definir regras de negócio para identificação de erros adicionais, incluindo mensagens de erro auxiliares dependendo do nível do usuário.

Para lançar uma instância da exceção criada, procede-se da mesma forma que uma exceção comum pertencente à linguagem Java, como demonstrado na [Listagem 8](#).

**Listagem 7.** Exemplo de exceção customizada, MinhaExcecao.

```
01. import java.io.File;
02. public class MinhaExcecao extends Exception {
03.     private static final long serialVersionUID = 1L;
04.     private int codigoErro = 0;
05.     private String mensagemAuxiliar;
06.     public MinhaExcecao(int codigoErro, String mensagem,
07.                         String mensagemAuxiliar){
08.         super(mensagem);
09.         this.codigoErro = codigoErro;
10.         this.mensagemAuxiliar = mensagemAuxiliar;
11.     }
12.     public int getCodigoErro() {
13.         return codigoErro;
14.     }
15.     public void setCodigoErro(int codigoErro) {
16.         this.codigoErro = codigoErro;
17.     }
18.     public String getMensagemAuxiliar() {
19.         return mensagemAuxiliar;
20.     }
21.     public void setMensagemAuxiliar(String mensagemAuxiliar) {
22.         this.mensagemAuxiliar = mensagemAuxiliar;
23.     }
```

**Listagem 8.** Exemplo com exceção customizada.

```
01. public class ExemploExcecao {
02.     public static void processar(int valor) throws MinhaExcecao{
03.         if (valor == 0)
04.             throw new MinhaExcecao(0, "MinhaExcecao", "MSG");
05.     }
06.     public static void main(String[] args) {
07.         try {
08.             processar(0);
09.         } catch (MinhaExcecao e) {
10.             e.printStackTrace();
11.             System.out.println("Codigo: " + e.getCodigoErro());
12.             System.out.println("Mensagem Auxiliar: " + e.getMensagemAuxiliar());
13.         }
14.     }
15. }
```

Como podemos verificar, na linha 4 é lançada uma exceção do tipo **MinhaExcecao** (definida na [Listagem 7](#)) a partir do comando **throw**. Como a exceção não é tratada no método **processar()**, o mesmo é declarado usando a cláusula **throws** (linha 2), o que faz com que o método **main()** use **try-catch** para capturar o possível erro (linha 7). O resultado da execução desse programa é mostrado na [Listagem 9](#).

A stacktrace é mostrada nas linhas 1 a 3, exibindo a pilha de chamada de métodos pertencente ao método onde o erro ocorreu. Logo após, na linha 4, é mostrado o código do erro e na linha 5 vemos a mensagem auxiliar.

**Listagem 9.** Resultado da execução da aplicação no console.

```
01. MinhaExcecao: MinhaExcecao
02. at ExemploExcecao.processar(ExemploExcecao.java:4)
03. at ExemploExcecao.main(ExemploExcecao.java:8)
04. Código: 0
05. Mensagem Auxiliar: MSG
```

## Exemplo com Threads

As exceções, como sabemos, são empregadas em qualquer situação que possua o risco da ocorrência de erros e paralização do programa. Diante disso, desempenham um papel muito importante em soluções que fazem uso de **Threads**. Quando uma thread entra em espera após chamar o método **wait()**, por exemplo, se ela for interrompida por outra thread, o mesmo lançará uma **InterruptedException**.

Um exemplo de código fonte para esta situação é mostrado na [Listagem 10](#), onde é usado **try-catch** na chamada do método **wait()**. Como **wait()** é um método que pode lançar **InterruptedException**, pois contém na sua declaração **throws InterruptedException**, torna-se necessário tratar a possível exceção.

**Listagem 10.** Exemplo de tratamento de exceções com threads.

```
01. public class Thread1 extends Thread {
02.     public static void main(String[] args) {
03.         Thread1 thread1 = new Thread1();
04.         thread1.start();
05.     }
06.     public void run() {
07.         ThreadUsuario b = new ThreadUsuario();
08.         b.start();
09.         synchronized (b) {
10.             b.interrupt();
11.         }
12.     }
13. }
14. class ThreadUsuario extends Thread {
15.     public void run() {
16.         synchronized (this) {
17.             try {
18.                 wait();
19.             }
20.             catch (InterruptedException e) {
21.                 e.printStackTrace();
22.             }
23.         }
24.     }
25. }
```

Neste código, é criada uma **Thread1** (linha 3) que inicia sua execução paralela após a chamada do método **start()** (linha 4), que por sua vez implica na invocação do método **run()** (linha 6). Como primeiro comando da execução desta thread, é criada uma **ThreadUsuario** (linha 7) e sua execução inicia com a chamada ao método **start()**, na linha 8 e, consequentemente, **run()**, na linha 15.

Agora, observe que existem três threads nesta aplicação: a thread principal, responsável por executar o método **main()** e que já está encerrada neste momento, **Thread1** e **ThreadUsuario**, criada por **Thread1**.

Seguindo com a execução do programa, a **ThreadUsuario** executa o método **wait()** na linha 18, ficando em espera. Logo após, essa espera é interrompida, pois **Thread1**, que ainda está executando paralelamente, invoca o método **interrupt()** de **ThreadUsuario** (vide linha 10).

Como o método **wait()** foi interrompido, ele lança uma **InterruptedException** (linha 20), imprimindo a stacktrace na saída padrão de erros, conforme demonstra a **Listagem 11**.

**Listagem 11.** Saída padrão de erros após a execução do código da Listagem 10.

```
java.lang.InterruptedException  
at java.lang.Object.wait(Native Method)  
at java.lang.Object.wait(Object.java:502)  
at ThreadUsuario.run(Thread1.java:20)
```

## NullPointerException

Uma exceção que pode ocorrer em qualquer trecho de código Java é a **java.lang.NullPointerException**, pois a mesma é lançada quando se tenta acessar um atributo ou método de uma referência a um objeto e esta referência é nula. Quando esta situação ocorre, a JVM identifica a situação de erro e lança a exceção **NullPointerException**, como exemplificado na **Listagem 12**.

**Listagem 12.** Exemplo de tratamento de NullPointerException.

```
01. public class Teste7 {  
02.     public static Object criarObjeto() {  
03.         return null;  
04.     }  
05.     public static void main(String[] args) {  
06.         try {  
07.             Object obj = criarObjeto();  
08.             System.out.println(obj.toString());  
09.         } catch (NullPointerException e) {  
10.             e.printStackTrace();  
11.         }  
12.     }  
13. }
```

O código dessa listagem contém um método **main()** – iniciado na linha 5 – que chama o método **criarObjeto()** e esse retorna uma referência a **Object**. Entretanto, observando a implementação de **criarObjeto()** – nas linhas 2 a 4 –, verifica-se que ele retorna uma referência nula ao executar o comando **return null**.

Como resultado, a variável **obj** receberá o valor **null** (ou seja, **obj** será uma referência nula) e o problema acontecerá na linha 8, onde se tenta invocar o método **toString()**. Neste momento será lançada a exceção **NullPointerException**, que é capturada pelo **catch** na linha 9 e impressa na saída padrão de erros ao invocarmos o método **printStackTrace()**. Este erro é apresentado na **Listagem 13**.

Uma observação importante sobre **NullPointerException** é que ela pode ser lançada em decorrência do uso do operador ponto (para acessar um método ou atributo de uma referência a um objeto). Entretanto, não é necessário usar **try-catch** toda vez que o operador ponto for usado, mesmo **NullPointerException** sendo uma exceção verificada.

## ArithmaticException

Uma exceção que normalmente ocorre com menos frequência, mas que corriqueiramente aparece em questões de certificações Java é a **ArithmaticException**; exceção que indica a ocorrência de um erro em uma operação aritmética, como a divisão por zero.

De forma semelhante à **NullPointerException**, não é obrigatório usar **try-catch** para cada operação matemática para capturar a ocorrência de **ArithmaticException**, porém ela pode ser lançada pela JVM ao tentar realizar uma divisão por zero, como exemplificado na **Listagem 14**.

**Listagem 13.** Resultado da execução do código da Listagem 12.

```
java.lang.NullPointerException  
at Teste7.main(Teste7.java:8)
```

**Listagem 14.** Exemplo de tratamento de ArithmaticException.

```
01. public class Teste8 {  
02.     public static void main(String[] args) {  
03.         try {  
04.             int i = (int) Math.random();  
05.             int j = 10 / i;  
06.         } catch (ArithmaticException e) {  
07.             e.printStackTrace();  
08.         }  
09.     }  
10. }
```

Neste código, a variável **i** receberá o valor de **Math.random()** convertido para inteiro, como visto na linha 4. Porém, como o valor retornado por essa função está sempre entre zero e um, ao realizar a conversão para inteiro, as casas decimais após a vírgula serão eliminadas, transformando o valor em zero, o que implica na atribuição do valor zero para a variável **i**.

Em seguida, na linha 5 é executado o comando **int j = 10 / i**, ou seja, ocorre a divisão de 10 por zero, o que faz com que a JVM lance a exceção **ArithmaticException**, que será capturada pelo **catch** da linha 6 e impressa na linha 7:

```
java.lang.ArithmaticException: / by zero  
at Teste8.main(Teste8.java:5)
```

Como visto anteriormente, não é necessário usar o **try-catch** para cada operação matemática. Para demonstrar essa afirmação, um exemplo diferente é apresentado na **Listagem 15**, onde se optou por não realizar o tratamento de **ArithmaticException**.

**Listagem 15.** Exemplo sem tratamento de ArithmaticException.

```
01. public class Teste9 {  
02.     public static void main(String[] args) {  
03.         int i = (int) Math.random();  
04.         int j = 10 / i;  
05.     }  
06. }
```

## Aprenda a trabalhar com exceções no Java

Ainda assim, a exceção é lançada em virtude da divisão por zero, sendo impressa uma stacktrace diferente nesse caso, como descrito a seguir:

```
Exception in thread "main" java.lang.ArithmetricException: / by zero
at Teste9.main(Teste9.java:4)
```

### Assertions

Existem situações em que se deseja testar o código-fonte para verificar se ele está funcionando corretamente, porém em vez de inserir testes adicionais que lançam exceções decorrentes da detecção de situações anormais, pode-se usar **assertions**. Este recurso avalia expressões booleanas e se for confirmado o erro (expressão booleana verdadeira), é lançada uma exceção **AssertionException**.

Um exemplo pode ser verificado quando lidamos com aplicações Java que contêm muitas dependências, e onde pode ser esperado um determinado comportamento decorrente da chamada de certo módulo da aplicação, que pode ser outro projeto ou uma biblioteca externa. Nestes casos, através de **assertions**, testes são inseridos no código, alertando se ocorrer um comportamento inesperado na utilização desse módulo.

Caso o resultado seja **true**, é sinalizado que o processamento ocorreu com sucesso, caso contrário, é determinado que houve

erro e é lançada uma **AssertionError**. A **Listagem 16** apresenta um exemplo onde é verificada uma condição que se espera que seja sempre verdadeira.

#### Listagem 16. Exemplo com assertions.

```
01. public class AssertionTest {
02.     public static void main(String[] args) {
03.         int x = (int)(10 * Math.random());
04.         assert x < 10 && x >= 0;
05.     }
06. }
```

Nesse código, **random()** gera um número aleatório entre 0 e 1, que é multiplicado por 10 e convertido para inteiro, eliminando a parte após a vírgula e resultando em um número no intervalo entre 0 e 9. Como resultado, não são esperados que sejam gerados números fora desse intervalo. Portanto, é afirmado na linha 4 que **x** deve ser menor que 10 e maior ou igual a zero. É claro que a condição vai ser sempre verdadeira, porém se alguma situação anormal acontecer (variável com valor inválido, etc.), a condição pode se tornar falsa, assim disparando um **AssertionError**.

De forma geral, o comando **assert** é usado para testar condições que devem ser verdadeiras quando o programa

# FÓRUM DEVMEDIA

O lugar perfeito para você ficar por dentro  
de tudo o que acontece nas tecnologias do  
mercado atual

No novo Fórum da DevMedia você vai encontrar canais  
específicos de Delphi, ASP.NET, Java, Banco de Dados e  
Engenharia de Software; além de ter contato com  
profissionais qualificados da área para troca de  
informações, sugestões e muito mais.



**ACESSE AGORA**  
[www.devmedia.com.br/forum](http://www.devmedia.com.br/forum)

funcionar corretamente e se ocorrer algum erro inesperado, um **AssertionError** deve ser lançado e o programador notificado. Na **Listagem 17** é apresentado um exemplo que pode lançar uma **AssertionError**, pois a condição do **assert** pode ser falsa dependendo do número que for gerado.

**Listagem 17.** Exemplo com assertions que lança **AssertionError**.

```
01. public class AssertionTest2 {  
02.     public static void main(String[] args) {  
03.         int x = (int)(10 * Math.random());  
04.         assert x < 10 && x >= 5;  
05.     }  
06. }
```

Esse código cria um número aleatório no intervalo de 0 a 10, porém o teste da **assertion** indica que o número tem que estar no intervalo de 5 a 10 para ser validado. Neste cenário, o teste da **assertion** pode falhar se, por exemplo, for gerado 0, obtendo **false** na **assertion**, o que leva ao lançamento da **AssertionException**:

```
Exception in thread "main" java.lang.AssertionError  
at AssertionTest2.main(AssertionTest2.java:4)
```

Geralmente as assertions são usadas para testar o código durante a atividade de desenvolvimento. Portanto, é recomendado não colocar regras de negócio nas assertions, mas sim apenas em testes lógicos, a fim de evitar problemas quando elas forem desabilitadas na implantação do código no cliente. Outra observação importante sobre assertions é que para usá-las é necessário habilitá-las usando o comando da JVM: **-ea**.

As exceções em Java são um assunto de grande relevância, estando presentes em qualquer projeto, pois possibilitam o controle dos erros de forma a manter ao mesmo tempo a organização no código e a eficiência, permitindo capturar tanto erros da aplicação como erros da máquina virtual.

As exceções são usadas na maioria das classes para controlar situações adversas e podem auxiliar bastante o programador no desenvolvimento de sistemas que exigem um controle maior sobre os erros. Além disso, a criação de novas exceções ajuda no controle de erros que podem ocorrer em diferentes módulos de uma aplicação, sendo atreladas a situações de erros decorrentes de diferentes regras de negócio.

Por fim, vale ressaltar que as assertions são uma poderosa ferramenta de teste de código-fonte no desenvolvimento de software Java, pois simplificam o fluxo de testes de uma aplicação permitindo lançar exceções decorrentes da execução desses testes. Como resultado, é viabilizada a detecção de possíveis situações de erro antes da homologação do software através da execução em modo de teste com dados reais.

## Autor



**John Soldera**

[johnsoldera@gmail.com](mailto:johnsoldera@gmail.com)

É bacharel em Ciências da Computação pela UCS (Universidade de Caxias do Sul), mestre em Computação Aplicada pela Unisinos e cursa atualmente doutorado em Ciências da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul). Trabalha com Java há 12 anos e possui a certificação SCJP.



## Links:

### Javadoc da plataforma Java SE 8.

<https://docs.oracle.com/javase/8/docs/api/>

### Tutorial sobre Exceções Java.

[http://www.tutorialspoint.com/java/java\\_exceptions.htm](http://www.tutorialspoint.com/java/java_exceptions.htm)

### Conteúdo da Oracle sobre Exceções.

<https://docs.oracle.com/javase/tutorial/essential/exceptions/>

### Exceptions in Java.

<http://www.javaworld.com/article/2076700/core-java/exceptions-in-java.html>

### Checked versus unchecked exceptions.

<http://www.javapractices.com/topic/TopicAction.do?Id=129>

### Java Exception Handling.

<http://tutorials.jenkov.com/java-exception-handling/index.html>

### Java Exception Handling Tutorial with Examples and Best Practices.

<http://www.javacodegeeks.com/2013/07/java-exception-handling-tutorial-with-examples-and-best-practices.html>

## Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/esmag/feedback](http://www.devmedia.com.br/esmag/feedback)

Ajude-nos a manter a qualidade da revista!



# Primeiros passos com JavaServer Faces

## Implemente o padrão Model-View-Controller e crie interfaces web baseadas em componentes

**D**efinir a arquitetura de uma aplicação é sempre uma tarefa difícil. Deve-se levar em consideração um conjunto diverso de fatores para que o software seja eficiente, que cumpra seus objetivos, mas que também seja agradável para o usuário, não apenas visualmente, mas no que diz respeito à sua usabilidade. E também, deve ter performance adequada e utilizar os recursos disponíveis da maneira mais otimizada possível.

O projeto ainda deve ser otimizado de forma que a equipe de desenvolvedores seja capaz de cumprir suas metas de recursos e prazos, dividindo o trabalho adequadamente sem prejuízo para a qualidade do código e, consequentemente, para o produto final.

Portanto, os padrões de projeto destinados ao estabelecimento de arquiteturas de software – dentre os quais vale destacar o Model-View-Controller (MVC, ou Modelo-Visão-Controle) – são de grande importância por tornar a implementação do projeto menos desgastante, aumentando a qualidade do código e fazendo com que este alcance seus objetivos.

A arquitetura de software proposta pelo JSF para a implementação do padrão MVC tem como meta aprimorar a implementação de interfaces com o usuário, separando as representações internas de informação da camada com que o usuário da aplicação tem contato, o que diminui a necessidade de escrita de código e faz com que a camada de visão fique mais “magra”.

A característica fundamental do padrão MVC é a divisão em três camadas com responsabilidades específicas, ou seja, a camada de Modelo é responsável por representar o domínio – ou os dados – da aplicação; a camada de Visão é responsável pela mediação do contato com o usuário, normalmente através de interfaces gráficas; enquanto a camada de Controle determina o comportamento da aplicação, interpretando as ações do usuário e traduzindo suas ações.

Sendo assim, a divisão de responsabilidades entre as camadas permite a reutilização de código, escalabilidade,

### Fique por dentro

Esse artigo é útil por explorar os fundamentos da arquitetura do JavaServer Faces (JSF), apresentando os seus principais recursos, características, benefícios e limitações, além de, por meio de um exemplo prático, descrever como o padrão Model-View-Controller (MVC) é implementado sob essa arquitetura. Com isso, demonstramos as contribuições do JSF para o projeto de aplicações web, especialmente para a construção de camadas de visão magras e de bom desempenho. A partir do conteúdo exposto, o leitor será capaz de iniciar o desenvolvimento de aplicações web utilizando a tecnologia de referência e mais adotada do Java.

dade, além da divisão do trabalho entre os membros da equipe, facilitando o desenvolvimento em paralelo, permitindo a criação de diversas visões e aumentando a produtividade. Por exemplo, uma mesma aplicação pode ter diversas visões, como vemos nas versões para dispositivos móveis de *front-ends* de uma aplicação web.

Sobre a plataforma Java existem diversas implementações e frameworks para o padrão MVC, dentre os quais podemos destacar o pioneiro framework Apache Struts e o Spring. É nesse contexto que está posicionado o JavaServer Faces (JSF).

Embora tenha sido idealizado quase que ao mesmo tempo que a especificação de Servlets, a especificação inicial do JSF foi publicada em 2004 com a *Java Specification Request (JSR) 127* para fornecer características que correspondam ao padrão MVC complementadas por um modelo de componentes de interfaces gráficas orientado a eventos para web. Dessa forma, não é um equívoco fazer um paralelo entre a proposta do JSF para o desenvolvimento de componentes de interface baseados em eventos e a arquitetura usada há bastante tempo em aplicações *desktop*.

Funcionando como um framework MVC, as requisições numa aplicação JSF são atendidas por um controller único – o *FacesServlet* – que se responsabiliza por todas as respostas da aplicação, recebendo entradas dos usuários, validando dados, preenchendo objetos da camada de modelo e renderizando as respostas, livrando o programador da tarefa de escrever essas operações.

Sendo assim, a aplicação se resume basicamente às páginas na camada de visão e *JavaBeans* na camada de modelo.

No exemplo apresentado nesse artigo, a camada de visão será construída em documentos XHTML, seguindo a especificação *Facelets*, uma linguagem para a construção de páginas baseada em *templates* HTML capaz de utilizar a árvore de componentes da arquitetura JSF. Facelets é a implementação recomendada para a camada de visão no JSF em detrimento de páginas JSP, por possuir performance de compilação e renderização melhor. Por serem escritos em documentos XHTML, os facelets suportam o uso de taglibs, reuso de código por meio de *templates*, além de serem flexíveis à customização e extensíveis.

Embora forneça uma implementação mais rígida para o padrão MVC, quando comparado com frameworks como o Struts ou Spring, o JSF é mais conciso, na medida em que os eventos disparados pelo usuário na camada cliente possuem manipuladores prontos na camada servidor, reduzindo o trabalho de codificação. Por exemplo, no Struts a camada de controle é baseada em *Actions* que precisam ser estendidas como forma de atender às requisições, executar a lógica de negócio e retornar ao Struts qual página da camada de visão deve ser renderizada. Embora não utilize classes *Action*, o Spring MVC possui uma estrutura semelhante, na qual as classes da camada de controle interagem com o *DispatcherServlet* por meio de mapeamentos descritos em anotações.

Já no JSF isso não é necessário devido ao fato das respostas às requisições serem baseadas na renderização de componentes de interface, de forma semelhante ao que é feito em interfaces gráficas tradicionais, como as aplicações *desktop*. Esses componentes possuem um modelo de eventos que os permite reagir a mudança de valores, ações de formulário e a alterações de estado do ciclo de vida da aplicação JSF. Por isso, o JSF é considerado um framework baseado em componentes, enquanto Struts é considerado um framework baseado em ações (ou “action framework”).

Analizando a implementação das camadas, também há diferenças consideráveis entre o JSF e alguns frameworks MVC. Por exemplo, no Struts, fica restrito à camada de controle tratar as requisições por meio da execução da lógica de negócio e produzir uma resposta que ele seja capaz de traduzir em um direcionamento para a camada de visão. No entanto, no JSF, é possível misturar responsabilidades entre as camadas, o que significa que é possível implementar métodos de negócio em *JavaBeans* que, como parte da camada de modelo, deveriam conter apenas atributos, setters e getters. Para alguns, essa flexibilidade é um ponto positivo do JSF em relação ao Struts, enquanto para outros, isso pode levar a implementações menos elegantes no que diz respeito ao padrão MVC (ou até mesmo em relação ao paradigma de desenvolvimento orientado a objetos).

Outra comparação importante diz respeito à configuração. Tanto o Spring MVC quanto o Struts fazem uso considerável de arquivos XML, embora devamos ressaltar que a versão 2 do Struts tenha reduzido bastante essa necessidade. Por outro lado, no JSF, praticamente não há a necessidade de arquivos de configuração,

o que representa uma simplificação importante, diminuindo o trabalho do desenvolvedor.

Vale ressaltar também que, enquanto Struts e Spring são frameworks, JSF é uma especificação, e por fazer parte da plataforma Java EE, existem diversas implementações do JSF, dentre as quais gostaríamos de destacar a *MyFaces*, criada pela Apache (veja a seção **Links**). No entanto, nesse artigo mostraremos a implementação de referência, ou oficial, chamada *Mojarra*. Para isso, vamos explorar os recursos essenciais do JSF por meio do desenvolvimento de uma aplicação web que persiste alguns dados em um banco de dados MySQL. Embora não seja o foco de nosso estudo, a camada de persistência fará uso do Hibernate ORM. Além disso, a IDE que adotamos para implementar o exemplo é o Eclipse.

Enfim, o JSF é uma alternativa interessante para a implementação do padrão MVC na medida em que diminui consideravelmente a tarefa de programação e apresenta ganhos consideráveis de desempenho na camada de visão, que não foi tão extensivamente explorada em outras implementações do padrão MVC. Além disso, seu uso em conjunto com outros frameworks é feito de maneira bastante simples, o que permite o desenvolvimento de aplicações web mais robustas e eficientes.

## Instalando os pacotes necessários e criando o projeto

Considerando que foi utilizada a IDE Eclipse na implementação do projeto, vamos criar um *Dynamic Web Project* que chamaremos de *EJMJavaServerFaces*. Em seguida, vamos alterar suas propriedades para que ele inclua a API JavaServer Faces. Para isso, clique com o botão direito sobre o projeto e escolha a opção *Properties*. À esquerda na janela de propriedades do projeto, selecione *Project Facets*, marque a opção *JavaServer Faces* e clique no botão *Ok*, como mostra a **Figura 1**. No nosso caso, usaremos a versão 2.2. Se você não possui os pacotes do JavaServer Faces no seu *classpath*, é possível selecioná-los ou fazer o download na mesma tela. Se preferir, utilize gerenciadores de dependências como o Maven, especialmente porque as classes do Hibernate também precisam estar no seu *classpath*.

Com essas alterações o projeto estará pronto para utilizar os recursos do JavaServer Faces, sendo, inclusive, criado o arquivo de configuração *WEB-INF\faces-config.xml*. Na sequência, vamos iniciar o desenvolvimento pela criação do pacote-base da aplicação. Dito isso, 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*.

Para implementar a arquitetura MVC, crie dentro do pacote-base os pacotes **control** e **model**, bem como o pacote onde serão armazenadas as classes da camada de persistência da nossa aplicação (**dao**). Nossas páginas (a camada *view*) ficarão no diretório *WEB-INF*. Como vamos usar recursos do Hibernate para a camada de persistência, crie também o arquivo *\src\hibernate.cfg.xml*. Dessa forma, a estrutura do projeto deve ficar como mostrado na **Figura 3**.

# Primeiros passos com JavaServer Faces

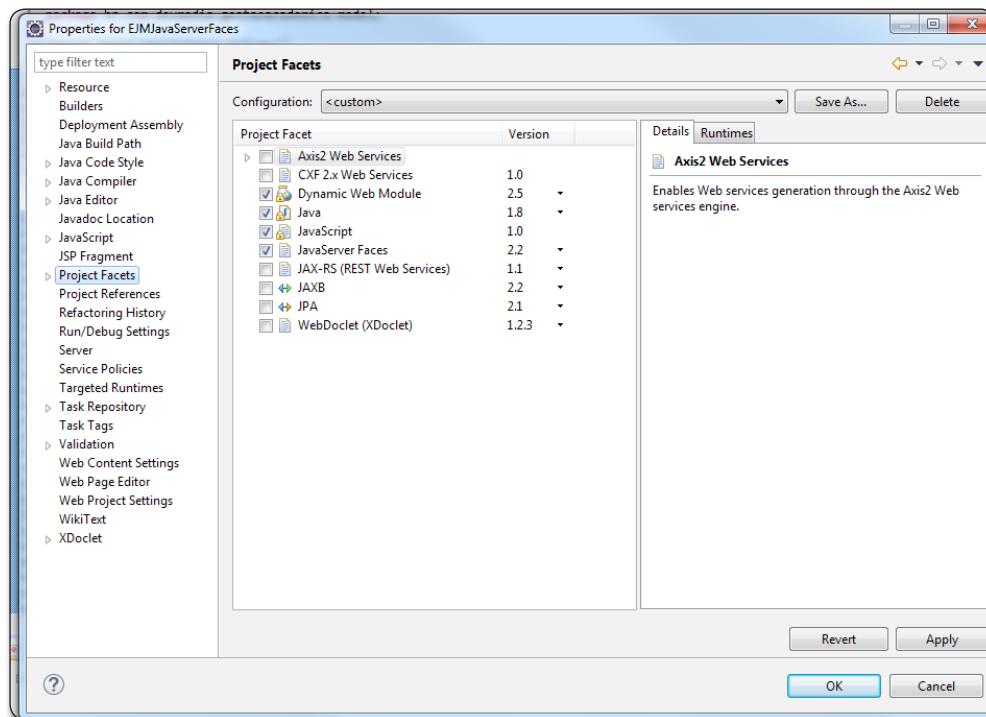


Figura 1. Incluindo os pacotes do JavaServer Faces na aplicação

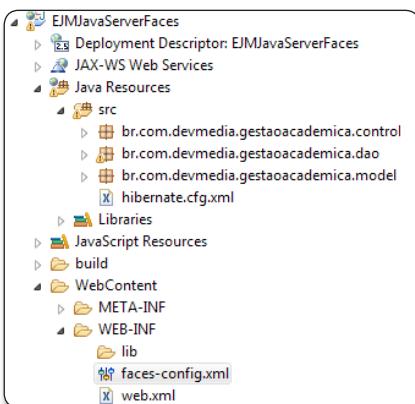


Figura 2. Criando o pacote-base da aplicação

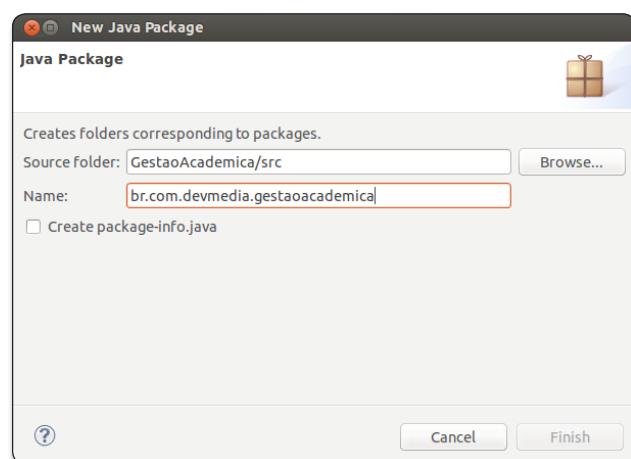


Figura 3. Estrutura do projeto

No arquivo `\src\hibernate.cfg.xml` manteremos as configurações de acesso ao banco de dados MySQL e o mapeamento das classes da camada de modelo no Hibernate. O código-fonte deste arquivo, no qual vemos propriedades como o endereço do banco de dados, usuário e senha, bem como o mapeamento das classes da camada de modelo, é apresentado na **Listagem 1**. Já o arquivo `faces-config.xml` foi criado automaticamente pelo Eclipse e não sofrerá alterações no nosso exemplo, mas seu código pode ser visto na **Listagem 2**.

## Implementando as camadas de modelo e persistência

O exemplo que construiremos nesse artigo é constituído de apenas um JavaBean na camada de modelo: **Docente**. O mapeamento desse bean pode ser visto no nó `mapping` no arquivo `hibernate.cfg.xml`, enquanto seu código-fonte pode ser visto na **Listagem 3**.

### Listagem 1. Código-fonte do arquivo `hibernate.cfg.xml`.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver
    </property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/
gestaoacademica</property>
    <property name="hibernate.connection.username">gestaoacademica</property>
    <property name="hibernate.connection.password">gestaoacademica</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.current_session_context_class">thread</property>

    <mapping class="br.com.devmedia.gestaoacademica.model.Docente"/>
  </session-factory>
</hibernate-configuration>
```

### Listagem 2. Código-fonte do arquivo `faces-config.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
  version="2.2">
</faces-config>
```

Nesse código, vemos anotações (vide **BOX 1**) do Hibernate que associam a classe **Docente** com sua respectiva tabela no banco de dados (através da anotação `@Table`). Como pode ser verificado, cada atributo da classe também é associado a um campo em sua respectiva tabela, por meio da anotação `@Column`.

**Listagem 3.** Código-fonte da classe Docente.

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

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

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

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

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

    @Column(name="MATRICULA")
    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;
    }
}
```

**BOX 1.** Anotações

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.

Com o *JavaBean* da camada de modelo criado, vamos implementar a camada de persistência em uma versão simplificada do padrão DAO, com uma interface (**DocenteDAO**) e sua implementação (**DocenteDAOImpl**) contendo os métodos para inserir e excluir registros do tipo **Docente** no banco de dados (**adicionarDocente()** e **excluirDocente()**, respectivamente), bem como recuperá-los em uma operação de listagem (**listarDocentes()**). O código-fonte da camada de persistência pode ser visto nas **Listagens 4** e **5**.

Como o uso do Hibernate não é o foco desse artigo, optamos por simplificar sua implementação deixando o método de obtenção da conexão (que usualmente é encapsulado em uma camada exclusiva ou implementado por algum framework) na própria classe **DocenteDAOImpl**. Dito isso, observe o método **buildSessionFactory()**. Este retorna uma **SessionFactory** (vide **BOX 2**) contendo a conexão com o banco que será utilizada por todos os métodos de persistência, por meio de chamadas ao método **getSessionFactory()**. De resto, são usados métodos do Hibernate como **save()**, **list()** e **delete()**, para adicionar, listar vários registros e excluir um único objeto do banco de dados, respectivamente.

**BOX 2.** SessionFactory

O conceito de SessionFactory estabelece que muitas threads podem acessar a conexão com o banco de dados de forma concorrente por meio da requisição de sessões. Um objeto do tipo **SessionFactory** é instanciado somente na inicialização da aplicação e, implementado por meio do padrão Singleton, pode ser acessado por outras camadas.

## ManagedBeans: Implementando a camada de controle

Com JSF, a camada de controle é composta pelos chamados **beans gerenciados** (ou ManagedBeans), que devem ser responsáveis por manipular os dados que trafegarão pela camada de visualização da aplicação. Esses beans gerenciados são nada mais do que JavaBeans comuns, normalmente utilizados na camada de modelo das aplicações Java, mas que aplicados sobre o framework JSF servem como modelos para componentes de interface e, por isso, podem ser acessados por uma página JSF.

Embora essa prática não possa ser considerada ruim, no exemplo demonstrado nesse artigo usaremos uma forma que consideramos mais elegante de implementação para manter a camada de modelo “magra”, ou seja, apenas com seus atributos, conforme já demonstrado na **Listagem 3**. Sendo assim, nossa camada de controle será composta por um ManagedBean que encapsulará as operações de negócio.

Portanto, vamos criar no pacote **control** a classe **DocenteController** e, nela, os métodos **excluirDocente()**, **listarDocente()** e **adicionarDocente()**, responsáveis por gerenciar os três tipos possíveis de requisições disponibilizadas por nossa aplicação. Vemos o seu código na **Listagem 6**.

**Nota**

Na versão atual do JSF não é necessário mapear o ManagedBean no arquivo faces-config.xml, uma vez que estes beans podem simplesmente ser registrados por meio de anotações.

A anotação `@ManagedBean` decora o bean, informando ao container que ele é gerenciado pelo JSF pelo nome especificado na propriedade `name`. Caso o nome não seja especificado, será adotado o nome da classe de forma qualificada, ou seja, com a primeira letra minúscula. Além desta, é possível definir outras propriedades como, por exemplo, o momento da inicialização do bean. Em nosso exemplo, no entanto, esses recursos não serão explorados.

Apesar disso, é necessário definir o escopo do bean, embora também seja adotado um escopo padrão caso essa propriedade não seja descrita. Em nosso exemplo, escolhemos o escopo “`session`”, definido por meio da anotação `@SessionScoped`. Isso determina que o ciclo de vida do `bean` se inicia com a primeira requisição HTTP e se encerra somente ao fim da sessão, ou seja, quando o usuário termina a navegação. Neste momento podemos destacar outros escopos muito utilizados, como:

- **@RequestScoped:** padrão para os casos em que o escopo não seja definido explicitamente. Nesse caso, o ciclo de vida do bean se ini-

**Listagem 4.** Código-fonte da interface DocenteDAO.

```
package br.com.devmedia.gestaooacademicaweb.dao;

import java.util.List;
import br.com.devmedia.gestaooacademicaweb.model.Docente;
public interface DocenteDAO {
    public void adicionarDocente(Docente docente);
    public List<Docente> listarDocentes();
    public void excluirDocente(Docente docente);
}
```

**Listagem 5.** Código-fonte da classe DocenteDAOImpl.

```
package br.com.devmedia.gestaooacademicaweb.dao;

import java.util.List;
import br.com.devmedia.gestaooacademicaweb.model.Docente;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class DocenteDAOImpl implements DocenteDAO {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    public static SessionFactory buildSessionFactory(){
        try {
            return new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public void adicionarDocente(Docente docente) {
        Transaction trns = null;
        Session session = getSessionFactory().openSession();
        try {

```

cia quando uma requisição HTTP é feita pelo usuário e se encerra quando a resposta associada a essa requisição é finalizada;

- **@ViewScoped:** com esta opção o bean vive enquanto o usuário permanecer visualizando a mesma página no navegador, ou seja, é destruído quando o usuário troca de página;
- **@ApplicationScoped:** com esta opção o ciclo de vida do bean é permanente enquanto a aplicação estiver executando, ou seja, se inicia na primeira requisição HTTP que envolva o bean e é destruído quando a aplicação é encerrada.

Outro detalhe importante na nossa implementação é a inicialização do objeto que será persistido. Lembre-se que o objeto que desejamos salvar no banco de dados é do tipo `Docente`. Portanto, a camada de controle precisa ser capaz de acessar um objeto desse tipo e preenchê-lo com os dados que virão da camada de visão, bem como enviá-lo para a camada de persistência.

Sendo assim, declaramos em `DocenteController` um atributo do tipo `Docente` (no trecho de código `private Docente docente`), implementando também seu `setter` e seu `getter`. Em seguida, criamos o método `init()`, que instancia `docente`. Utilizamos nesse método a anotação `@PostConstruct` para fazer com que ele seja chamado logo após qualquer instanciação da classe. No nosso caso, isso garante que o método `init()` será chamado toda vez que `DocenteController` for instanciado e, dessa forma, teremos sempre um objeto `Docente` inicializado.

A anotação `@PostConstruct` também garante que o método anotado só será executado uma vez durante todo o ciclo de vida

```
        trns = session.beginTransaction();
        session.save(docente);
        session.getTransaction().commit();
    } catch (RuntimeException e) {
        if (trns != null) trns.rollback();
    } finally {
        session.flush();
        session.close();
    }
}

public List<Docente> listarDocentes() {
    return getSessionFactory().openSession().createCriteria(Docente.class).list();
}

public void excluirDocente(Docente docente) {
    Transaction trns = null;
    Session session = getSessionFactory().openSession();
    try {
        trns = session.beginTransaction();
        session.delete(docente);
        session.getTransaction().commit();
    } catch (RuntimeException e) {
        if (trns != null) trns.rollback();
    } finally {
        session.flush();
        session.close();
    }
}
```

**Listagem 6.** Código-fonte da classe DocenteController.

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

import java.util.List;

import javax.annotation.PostConstruct;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.model.DataModel;
import javax.faces.model.ListDataModel;

import br.com.devmedia.gestaoacademica.dao.DocenteDAO;
import br.com.devmedia.gestaoacademica.dao.DocenteDAOImpl;
import br.com.devmedia.gestaoacademica.model.Docente;

@ManagedBean(name="docenteController")
@SessionScoped
public class DocenteController {

    private Docente docente;
    private DataModel<Docente> listaDocentes;
    private String msg;

    public Docente getDocente() {
        return docente;
    }

    public void setDocente(Docente docente) {
        this.docente = docente;
    }

    public DataModel<Docente> getListarDocentes() {
        DocenteDAO dao = new DocenteDAOImpl();
        List<Docente> lista = dao.listarDocentes();
        listaDocentes = new ListDataModel<Docente>(lista);
        return listaDocentes;
    }

    public void setListaDocentes(DataModel<Docente> listaDocentes) {
        this.listaDocentes = listaDocentes;
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }

    public String adicionarForm() {
        docente = new Docente();

        return "inserir_docente_form";
    }

    public String adicionarDocente() {
        DocenteDAO dao = new DocenteDAOImpl();
        dao.adicionarDocente(docente);

        setMsg("Salvo com sucesso!");
        return "inserir_docente_form";
    }

    public String excluirDocente() {
        Docente d = (Docente)(listaDocentes.getRowData());
        DocenteDAO dao = new DocenteDAOImpl();
        dao.excluirDocente(d);

        setMsg("Excluído com sucesso!");
        return "listar_docentes";
    }

    public String listarForm() {
        return "listar_docentes";
    }

    @PostConstruct
    public void init() {
        docente = new Docente();
    }
}
```

do *bean* gerenciado, mesmo que o bean seja instanciado várias vezes pelo container. É importante observar esse detalhe especialmente quando o bean gerenciado possui escopo de requisição (**@RequestedScoped**), uma vez que nesses casos seu ciclo de vida se encerra ao fim da requisição, tendo uma possibilidade maior de novas instanciações serem necessárias.

## Criando a camada de visão com Facelets

Um dos recursos mais interessantes do JSF é que as interações entre os JavaBeans e a camada de visão da aplicação são implícitas, ou seja, não precisam ser implementadas. Isso é viabilizado através de um padrão de projeto intitulado Inversão de Controle (vide **BOX 3**), que faz com que o container seja responsável pelo mapeamento entre o JavaBean e sua camada de visão correspondente.

Como dissemos anteriormente, nossa camada de visão será baseada em *Facelets*, em vez de páginas JSP. A principal diferença é que o processamento de uma página JSP se dá de cima para

### BOX 3. Inversão de Controle

Inversão de Controle é um padrão de projeto em que a chamada de métodos é invertida, ou seja, não é determinada pelo programador, mas delegada a um componente que possa tomar conta dessa execução – o container. Uma das maneiras de aplicar a inversão de controle é por meio da injeção de dependência, que torna o container responsável por injetar em cada componente as dependências que tiverem sido declaradas.

baixo, com os elementos JSP sendo interpretados na ordem em que estão colocados na página. Em *Facelets*, pelo contrário, o ciclo de vida dos componentes é mais complexo, pois sua instanciação e exibição ocorre em separado e em ordem definida, o que melhora a sua performance.

Hans Bergsten, autor do livro “JavaServer Faces”, apresenta esse fato – entre outros problemas na combinação de JSP com JSF – em seu artigo “*Improving JSF by Dumping JSP*” (“Aprimorando JSF jogando JSP fora”, em tradução livre), indicado na seção **Links**.

Também é importante ressaltar que, uma vez que são implementados em documentos XHTML, os *facelets* utilizam um mecanismo de validação semelhante ao de documentos XML, sendo, por exemplo, sensível às diferenças entre caracteres maiúsculos e minúsculos, exigindo o fechamento de todas as *tags* utilizadas, etc.

No nosso exemplo, a camada de visão terá duas telas: uma para a listagem dos docentes cadastrados e outra para o cadastro de docentes, como mostram as **Figuras 4 e 5**.

Nome	Matrícula	Titulação	
Catherine Burns	0545	Ph.D.	<a href="#">Excluir</a>
Mario Vidal	7890	Dr. Ing.	<a href="#">Excluir</a>
Paulo Carvalho	3423	D.Sc.	<a href="#">Excluir</a>

Figura 4. Listagem de docentes cadastrados

Nome:

Matrícula:

Titulação:

Figura 5. Formulário de cadastro de docentes

Enquanto a primeira exibe os registros cadastrados no banco, a segunda adiciona os registros quando o usuário clica no botão *Salvar*. Para implementá-las, vamos criar dois arquivos *.xhtml* no diretório \WEB-INF: *listar\_docentes.xhtml* e *inserir\_docente\_form.xhtml*, cujos respectivos códigos-fontes podem ser vistos nas **Listagens 7 e 8**.

Podemos verificar no início dos códigos as primeiras diferenças entre páginas XHTML e o tradicional HTML, a começar pelas declarações XML nas primeiras linhas. Podemos verificar também as declarações das bibliotecas de *tags Facelets* na tag HTML, como no trecho de código *xmlns:h="http://java.sun.com/jsf/html"*, que declara o uso da *taglib h*.

A declaração dos campos de formulário usa a *taglib h*, como no trecho de código *<h:inputText id="tname">*, que cria um campo de texto. Vemos ainda outras tags sendo utilizadas, como *outputText*, que cria rótulos e *commandButton*, que cria botões, bem como *form*, para delimitar o formulário.

Por sua vez, a associação entre as páginas e os beans gerenciados pode ser vista nos *scriptlets* presentes no código. Por exemplo, vemos que a propriedade *value* dos campos é preenchida com valores vindos de classes na camada de controle, como demonstra o trecho de código *value="#{docenteController.docente.nome}"*. Nesse trecho, temos uma associação do valor do campo no formulário com o atributo *nome* da classe *Docente*.

**Listagem 7.** Código-fonte da listagem de docentes.

```
<%@page encoding="UTF-8"%>
<!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:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

<head>
    <title>Listagem de Docentes</title>
</head>
<body>

<h3>Docentes Cadastrados</h3>

<h3>
    <h:form>
        <h:commandLink value="+ Novo Docente"
                      action="#{docenteController.adicionarForm}"/>
    </h:form>
</h3>

<h4><h:outputLabel id="oplMensagem" value="#{docenteController.msg}"/></h4>

<h: dataTable value="#{docenteController.listaDocentes}" var="docente"
               border="1">

    <h:column>
        <f:facet name="header">Nome</f:facet>
        #{docente.nome}
    </h:column>

    <h:column>
        <f:facet name="header">Matrícula</f:facet>
        #{docente.matricula}
    </h:column>

    <h:column>
        <f:facet name="header">Titulação</f:facet>#{docente.titulacao}
    </h:column>

    <h:column>
        <f:facet name="header">&nbsp;</f:facet><h:form>
            <h:commandLink action="#{docenteController.excluirDocente}"
                           value="Excluir"/>
        </h:form>
    </h:column>
</h: dataTable>

</body>
</html>
```

Já na listagem de docentes, cujo código pode ser visto na **Listagem 7**, podemos notar o uso da *tag dataTable*, que fornece um controle capaz de criar tabelas HTML com dados provenientes de uma *Collection*. Usamos esse recurso para exibir uma tabela com os docentes cadastrados no banco, que são carregados para uma *List* por meio do método *getListaDocentes()* na classe *DocenteController*, como pode ser visto na **Listagem 6**.

Outro recurso interessante é que o JSF é capaz de obter a linha em que o cursor está posicionado (por meio do método *getRowData()* na classe *DataModel*). Com isso, pudemos implementar a exclusão de registros mais facilmente, simplesmente passando o registro selecionado para o método *delete()* do Hibernate, como pode ser visto no método *excluirDocente()* na classe *DocenteController*.

**Listagem 8.** Código-fonte do cadastro de docentes.

```
<?xml version='1.0' encoding='UTF-8'?>
<!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:h="http://java.sun.com/jsf/html"
 xmlns:f="http://java.sun.com/jsf/core">

<h:head>
 <title>Cadastro de Docentes</title>
</h:head>

<h:body>

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

<h3>
 <h:form>
 <h:commandLink value="Listagem de Docentes"
 action="#{docenteController.listarForm}"/>
 </h:form>
</h3>

<h4><h:outputLabel id="oplMensagem" value="#{docenteController.msg}"></h4>

<h:form>
 <table>
 <tr>
 <td><h:outputText id="lNome" value="Nome:" /></td>
 <td><h:inputText id="tNome"
 value="#{docenteController.docente.nome}"/></td>
 </tr>
 <tr>
 <td><h:outputText id="lMatricula" value="Matrícula:" /></td>
 <td><h:inputText id="tMatricula"
 value="#{docenteController.docente.matricula}"/></td>
 </tr>
 <tr>
 <td><h:outputText id="lTitulacao" value="Titulação:" /></td>
 <td><h:inputText id="tTitulacao"
 value="#{docenteController.docente.titulacao}"/></td>
 </tr>
 <tr>
 <td colspan="2">
 <h:commandButton id="submit" type="submit"
 action="#{docenteController.adicionarDocente}" value="Salvar"/>
 </td>
 </tr>
 </table>
</h:form>

</h:body>
</html>
```

Também observamos no código-fonte da listagem de docentes o uso da tag **column**, definida para criar as colunas da tabela, em que exibimos dados vindos do objeto por meio de *scriptlets* (como em `#{docente.nome}`). Além disso, usamos a tag **facet** para criar os cabeçalhos das colunas da tabela.

Feito isso, a aplicação está pronta e pode ser testada através da URL [http://localhost:8080/GestaoAcademicaWeb/faces/listar\\_docentes.xhtml](http://localhost:8080/GestaoAcademicaWeb/faces/listar_docentes.xhtml), que abrirá a página de listagem de docentes. Ao clicar no link *Cadastrar novo Docente*, somos direcionados para o formulário de cadastro. Nesta página, ao preencher os campos e clicar em

*Salvar*, os dados inseridos são salvos no banco e a mensagem “Salvo com sucesso!” é exibida, como mostra a **Figura 6**.

As mensagens de resposta das operações são carregadas no **msg** da classe **DocenteController**. Como mostram as **Listagens 6** e **7**, atribuímos valores à **msg** ao fim de cada operação, e esses valores são exibidos nas páginas por meio da tag **outputText**. Por exemplo, vemos na operação de exclusão que a mensagem “Excluído com sucesso!” é exibida quando um registro é excluído (vide **Figura 7**).

**Formulário de Cadastro de Docentes**

**Listagem de Docentes**

**Salvo com sucesso!**

Nome: Alain Wisner

Matrícula: 2134

Titulação: Ph.D.

Salvar

**Figura 6.** Cadastrando Docentes

**Docentes Cadastrados**

**+ Novo Docente**

**Excluido com sucesso!**

Nome	Matrícula	Titulação	
Catherine Burns	0545	Ph.D.	<a href="#">Excluir</a>
Paulo Carvalho	3423	D.Sc.	<a href="#">Excluir</a>
Alain Wisner	2134	Ph.D.	<a href="#">Excluir</a>

**Figura 7.** Excluindo um registro

Por fim, repare novamente na classe **DocenteController** que as operações de negócio retornam *strings*. Essas *strings* representam a página para a qual o usuário será direcionado ao fim da operação. Por exemplo, podemos ver que a operação **excluirDocente()** direciona o usuário para “listar\_docentes”, exatamente como acontece em nosso teste, em que o usuário é direcionado para a listagem de docentes após excluir um registro. Da mesma forma, criamos os métodos **listagem()** e **novoDocente()**, responsáveis por criar direcionamentos para a listagem de docentes e para o cadastro de docentes, respectivamente.

Em seu livro “*Ecological Interface Design*” (Projeto de Interfaces Ecológicas, ainda não publicado no Brasil), Catherine Burns e John Ajdukiewicz fazem uma breve comparação entre o projeto de interfaces com o usuário e uma arte. No entanto, os próprios autores não são conclusivos quanto a essa comparação, uma vez que é bastante difícil caracterizar o projeto de interfaces como uma arte. Mais do que isso, é importante ressaltar que aspectos não somente visuais, como também de desempenho e facilidade de implementação, devem ser levados em consideração no projeto de uma interface com o usuário.

O desenvolvimento de aplicações para web ainda é um segmento importante no mercado de desenvolvimento. Logo, é natural que a comunidade Java continue buscando novas implementações para atrair profissionais, especialmente no mercado corporativo. No entanto, embora haja de fato uma farta exploração do domínio das aplicações web pela plataforma Java, é possível verificar que o desenvolvimento de interfaces para a camada de visão não teve a mesma atenção se comparado com as outras camadas do padrão de projeto mais popular, o MVC.

Por exemplo, se observarmos com atenção as implementações do padrão MVC fornecidas pelos frameworks existentes, podemos notar que há pouca inovação no que diz respeito à implementação da camada de visão, como melhorias de desempenho e qualidade, aprimoramentos no ciclo de vida dos componentes de interface, limpeza de código, aprimoramento da experiência do usuário, etc. Isso quer dizer que embora os benefícios do padrão MVC sejam bastante claros – e suas implementações ilustram de forma satisfatória esses benefícios – a camada de visão não tem sido explorada no mesmo potencial que as demais camadas pelos frameworks mais populares do mercado.

Nesse sentido, o JSF faz uma contribuição clara, por fornecer implementações simples e de boa performance para a camada de visão. Sua biblioteca de componentes de interface é bastante

completa, escalável e de simples implementação, além de integrada, sendo compatível inclusive com outras implementações para o padrão MVC. Mais do que isso, o JSF fornece um mecanismo de escuta de eventos para seus componentes capaz de receber as ações do usuário, dando mais robustez à captação de eventos em aplicações web.

Embora nesse artigo tenhamos tratado da principal implementação do JSF, não exploramos bibliotecas de componentes que estendem seus recursos, dentre as quais vale destacar o PrimeFaces, bastante popular na comunidade de desenvolvedores. Lançado inicialmente em 2009, o PrimeFaces fornece diversos componentes de interface e *templates* compatíveis com o JSF, sendo, portanto, uma interessante oportunidade de aprofundamento para o leitor nesse tipo de tecnologia. Outra sugestão é estudar a integração do JSF com frameworks MVC como o Struts ou o Spring, além do aprofundamento na integração com o Hibernate para implementação da camada de persistência.

Dentre as características do JSF abordadas nesse artigo, podemos destacar a separação mais evidente entre o negócio e a visão em relação ao JSP, já que no código de nossos facelets só foram escritas as exibições dos componentes de interface. Isso representa um grande benefício não somente para a elegância do código produzido, mas também para a sua clareza, produzindo camadas de

## DÊ UM SALTO EM CONHECIMENTO!



Acesse o maior  
portal para  
desenvolvedores  
da América  
Latina!

20  
mil  
posts

430  
mil  
cadastrados

10  
milhões de  
page-views  
por mês

visão mais magras. Essa separação entre a lógica de negócio e a apresentação também traz benefícios para o projeto, na medida em que torna possível dividir as tarefas de implementação dos componentes da camada de visão entre os membros da equipe de desenvolvedores, até mesmo entre profissionais com níveis diferentes de experiência com programação.

Um exemplo para essa divisão de trabalho é a possibilidade de delegar a um grupo de desenvolvedores com menos experiência em Java a tarefa de construir a interface usando componentes de interface JSF, enquanto programadores mais experientes desenvolvem a camada *server-side* da aplicação, uma vez que unir essas camadas é algo simples na arquitetura JSF. No entanto, se a sua aplicação web faz um uso muito extenso de CSS ou JavaScript, além de recursos muito particulares de AJAX, talvez os componentes disponíveis no JSF não sejam suficientes, e o uso de XHTML pode trazer algumas dificuldades, fazendo com que Spring ou Struts se tornem mais atraentes, mesmo sendo necessário implementar todo o código CSS e JavaScript.

Com relação à arquitetura, a literatura especializada aponta algumas deficiências do JSF, especialmente na relação entre os *ManagedBeans* e a arquitetura orientada a serviços (SOA). Para alguns autores, o JSF enfraquece a SOA por não impedir que os programadores escrevam grandes quantidades de código no *ManagedBean*. No entanto, isso não necessariamente é um problema, uma vez que, caso o desenvolvedor realmente queira seguir a arquitetura orientada a serviços, basta implementar a lógica de negócio de acordo com os padrões definidos para a mesma.

Há também críticas à arquitetura de navegação fornecida no JSF, que embora traga alguns aprimoramentos em relação ao Struts, não permite, por exemplo, implementar redirecionamentos por meio de configurações, já que as *actions* são configuradas na página. Também não possui recursos nativos para o gerenciamento do fluxo de páginas, embora seja compatível com o Spring Web Flow, módulo do Spring framework que adiciona ao MVC a possibilidade de encapsular uma sequência de passos para guiar o usuário através de um conjunto definido de páginas que representem uma tarefa específica relacionada ao negócio da aplicação.

Por fim, podemos afirmar que o JavaServer Faces representa uma arquitetura que enriquece a plataforma Java, fornecendo implementações bastante interessantes para o gerenciamento de dados em componentes de interface, manutenção do estado de componentes, validação de dados de usuários e na captura de eventos em aplicações web.

## Autor



**Alessandro Jatobá**

jatoba@jatoba.org

É Mestre em Ciência 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 Projeto JavaServer Faces (Mojarra).**

<https://javaserverfaces.java.net/>

### **Site oficial do Apache MyFaces.**

<http://myfaces.apache.org/>

### **Site oficial do Hibernate.**

<http://www.hibernate.org>

### **Artigo "Improving JSF by Dumping JSP", por Hans Bergsten.**

<http://www.onjava.com/pub/a/onjava/2004/06/09/jst.html>

### **JavaServer Faces Technology.**

<http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>

### **Tutorial: JavaServer Faces Application.**

[http://docs.oracle.com/cd/E11035\\_01/workshop102/webapplications/jsf/jsf-app-tutorial/Introduction.html](http://docs.oracle.com/cd/E11035_01/workshop102/webapplications/jsf/jsf-app-tutorial/Introduction.html)

### **Understanding JSF as a MVC framework.**

<http://stackoverflow.com/questions/10111387/understanding-jsf-as-a-mvc-framework>

## Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/esmag/feedback](http://www.devmedia.com.br/esmag/feedback)

Ajude-nos a manter a qualidade da revista!



# Criando um CRUD RESTful com Jersey, JPA e MySQL

Aprenda neste artigo a implementar serviços em um CRUD RESTful para cadastro de clientes

Durante muito tempo os web services baseados em SOAP foram praticamente a única solução para a comunicação e implementação de sistemas distribuídos. Devido a isso e visando a qualidade dos serviços, essa opção passou por várias melhorias ao longo dos anos, principalmente relacionadas à segurança. No entanto, os web services SOAP acabaram ficando complexos, de difícil implementação e com custos elevados de adoção. A partir de então, a opção por esse tipo de web service passou a ser inviável em alguns cenários, seja pelo custo, por recursos de hardware e de software ou mesmo pelo grande consumo de banda da rede, principalmente por parte dos dispositivos móveis, que ainda não têm uma conexão de alta velocidade a preços acessíveis. O protocolo SOAP necessita de uma série de parâmetros e configurações no formato XML para viabilizar a troca de dados entre cliente e servidor e isso torna as mensagens longas tanto para tráfego na rede quanto para o dispositivo cliente processar a resposta.

O uso de web services surge da necessidade de se ter uma aplicação distribuída e escalável, necessidade esta que se espalha pelos mais diversos setores de negócio por todo o mundo, seja para integrar sistemas em diferentes plataformas, como os dispositivos móveis, seja para conectar sistemas web a sistemas legados, expor um canal de comunicação para clientes ou parceiros, dentre outras necessidades. Neste cenário, com o intuito de facilitar a comunicação entre sistemas, podemos fazer uso do protocolo HTTP e usufruir do padrão arquitetural REST para implementar os serviços web, simplificando assim a troca de dados entre cliente e servidor. Como um grande diferencial, o REST suporta os principais formatos para comunicação e troca de informações (JSON e XML), popularmente utilizados no desenvolvimento de sistemas distribuídos e em outras aplicações.

Conhecido por ser um estilo híbrido derivado de vários estilos arquiteturais baseados em rede, o REST tem como idealização e pilar a implementação de serviços web

## Fique por dentro

Este artigo é útil por demonstrar, passo a passo, como desenvolver um serviço RESTful com as operações de um CRUD para um cadastro de clientes totalmente web através de um projeto Java EE com Maven. Para que você possa compreender e aprender como utilizá-las, este projeto irá envolver uma gama de tecnologias da plataforma Java com o objetivo de criar uma solução de qualidade e escalável. Para isso, será apresentada a construção de um DER para o banco de dados MySQL seguido pelo desenvolvimento de um serviço REST utilizando o Jersey, implementação de referência da especificação JAX-RS. Além disso, será utilizada a especificação JPA e o Hibernate como ferramenta ORM para fazer o mapeamento objeto relacional entre as tabelas do banco de dados e as classes Java. Ao final, você saberá como implementar seus primeiros serviços web, recurso cada vez mais comum no mercado de software, que busca soluções capazes de prover diferentes interfaces com o usuário e de fácil integração.

baseados no protocolo HTTP. O termo surgiu nos anos 2000, na dissertação do coautor do protocolo HTTP, Dr. Roy Thomas Fielding, para obtenção do título de PhD, com a dissertação “Architectural Styles and the Design of Network-based Software Architectures”.

Neste artigo iremos desenvolver um serviço RESTful responsável por viabilizar um cadastro de clientes com todas as funcionalidades de um CRUD. O desenvolvimento desse cadastro consiste na implementação de um web service para que um cliente possa consumir e realizar as operações básicas de acesso ao banco de dados.

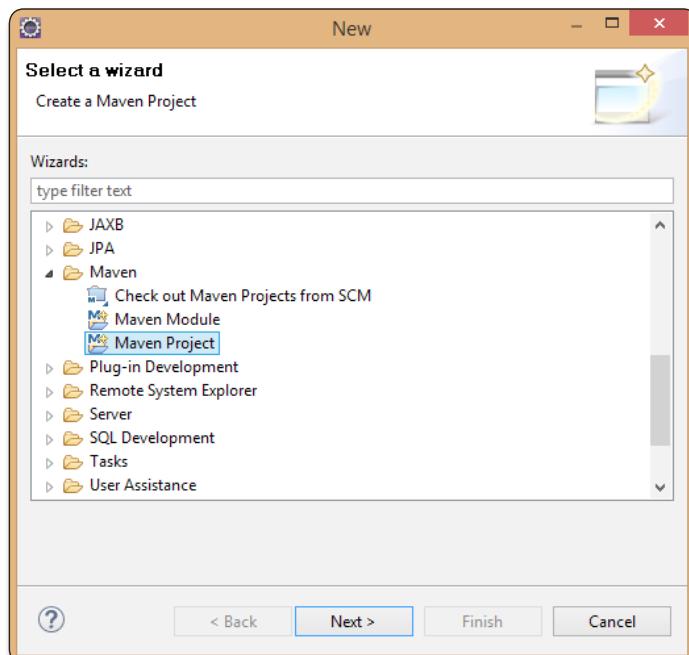
## Instalando o Eclipse Luna

Para o desenvolvimento do projeto “cadastro de clientes” iremos utilizar o Eclipse Luna. Sendo assim, baixe esta versão na página do Eclipse e então descompacte o arquivo em um diretório de sua preferência.

Ao executar esta IDE pela primeira vez é solicitado ao usuário que informe um diretório para servir como ambiente de trabalho. O workspace nada mais é do que uma pasta adotada pelo Eclipse para salvar os projetos que ele está gerenciando.

## Criando o projeto com Eclipse e Maven

Para criar o projeto no Eclipse, clique no menu *File > New > Other*. Logo após, será exibida uma nova janela, conforme a **Figura 1**, para que seja selecionado o wizard que auxiliará na criação do projeto. Neste caso, selecione a opção *Maven > Maven Project* e clique em *Next*.



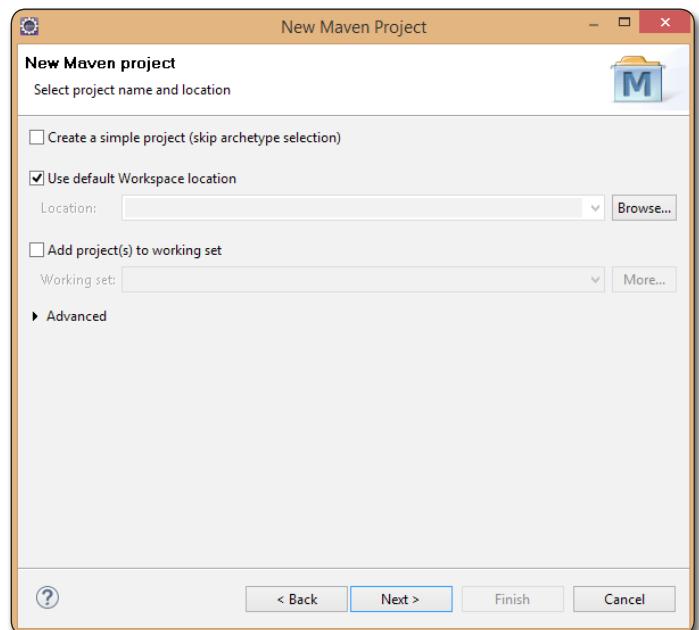
**Figura 1.** Selecionando o wizard Maven Project

Na próxima tela, de criação do projeto Maven, deixe marcada a opção *Use default workspace location* para que seja utilizado o workspace configurado ao executar o Eclipse pela primeira vez. Em seguida, clique mais uma vez em *Next* (vide **Figura 2**).

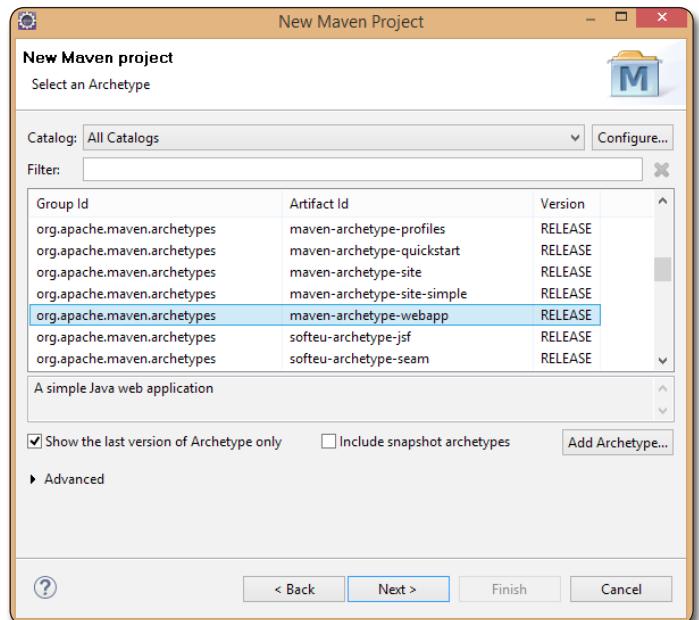
Agora, conforme apresentado na **Figura 3**, deve ser selecionado o Archetype do Maven a ser utilizado para criar o projeto. Neste caso, selecione a opção *maven-archetype-webapp*, pois iremos criar um projeto Java EE. O Archetype é uma espécie de template que viabiliza a criação de projetos com base em uma tecnologia ou especificação, como JSF, JPA, Spring, Struts, dentre outras. Feito isso, clique em *Next*.

A última tela do wizard, apresentada na **Figura 4**, mostra alguns campos que devem ser preenchidos para finalizar a criação do projeto. Nela, deve ser informado o group id (geralmente informase o site da organização), o id do artefato (identificador do projeto), a versão deste e o nome completo do pacote base.

Após clicar em *Finish*, em algumas ocasiões pode ocorrer um erro de acesso à pasta do repositório local do Maven, informando que não foi possível definir esse repositório. Para solucioná-lo você deve deletar a pasta *repository* para que o Eclipse possa criá-la novamente com as permissões de acesso corretas. O local do repositório pode ser verificado na opção *Local Repository*, conforme apresentado na **Figura 5**, nas preferências do Eclipse.



**Figura 2.** Setando o workspace para o projeto



**Figura 3.** Selecionando o Archetype do Maven para o projeto cadastro de clientes

O repositório nada mais é que uma pasta na máquina local onde o Maven irá baixar de seu servidor ou servidor personalizado todas as dependências do projeto, como frameworks, plugins e bibliotecas.

## Configurando as dependências e plugins no pom.xml

Para que possamos construir o projeto será necessário utilizar várias bibliotecas Java, referentes ao Jersey, Hibernate, MySQL e plugin do Tomcat. Sendo assim, devemos informar essas bibliotecas e suas respectivas versões no *pom.xml*. Feito isso, o Maven

# Criando um CRUD RESTful com Jersey, JPA e MySQL

se responsabilizará por baixá-las e gerenciá-las a partir de então. Na **Listagem 1** é apresentado o código do arquivo *pom.xml* do nosso projeto.

Vejamos a seguir algumas considerações sobre a listagem:

- **Linhas 4 a 8:** Neste trecho são definidas algumas informações referentes ao projeto para que o Maven possa realizar o controle

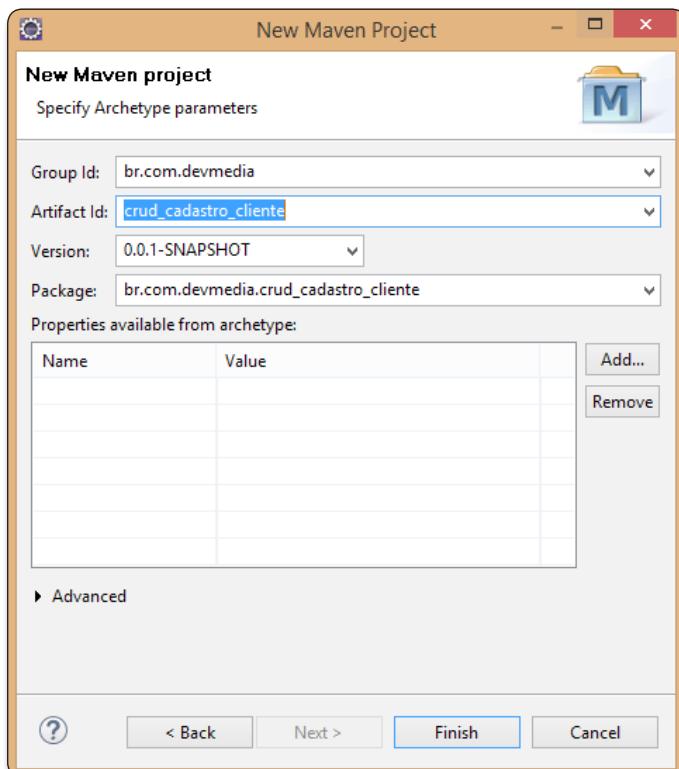


Figura 4. Definição dos dados para identificação do projeto

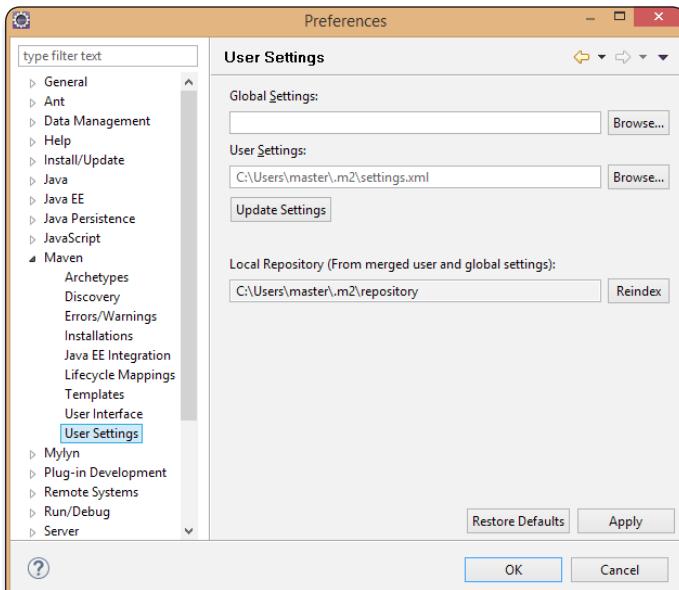


Figura 5. Definindo a nova pasta repository

do ciclo de vida do desenvolvimento. Assim sendo, é informado o identificador do grupo, o identificador do artefato, a versão e o nome do projeto;

- **Linhas 11 a 16:** Aqui foi inserida uma dependência ao JUnit para testes de unidade no projeto. O Maven define a mesma automaticamente no ato da criação do projeto;
- **Linhas 17 a 21:** Neste bloco consta a dependência ao servidor Jersey, framework responsável por tratar as requisições realizadas ao serviço. Seu código assume o papel de Servlet Container para receber e tratar todas as requisições realizadas via protocolo HTTP. Neste projeto iremos implementar as requisições HTTP com os verbos GET, POST, PUT e DELETE para as respectivas operações de um CRUD. Assim será possível inserir e realizar a manutenção dos dados no banco MySQL;
- **Linhas 22 a 26:** Neste bloco é definida a dependência para o suporte à troca de dados no formato JSON. Desta forma será possível enviar e receber dados no formato JSON entre as requisições e respostas do cliente ao servidor;
- **Linhas 27 a 47:** Neste intervalo são informadas todas as dependências do Hibernate. O Hibernate é um framework que viabiliza o mapeamento objeto relacional entre as tabelas do banco de dados e as classes Java. Lembre-se que as classes Java representam as entidades do mundo real no sistema. É importante frisar ainda que embora tenhamos adotado o Hibernate, programaticamente iremos utilizar as interfaces da especificação JPA para acesso ao banco;
- **Linhas 48 a 53:** Aqui é definida a dependência da biblioteca do MySQL. O Hibernate fará uso desta para acessar o banco de dados.

Definidas as dependências, vamos configurar um servidor web que rode aplicações Java EE. Para isso, podemos usar o WildFly, GlassFish, dentre outros, porém, neste artigo será adotado o Tomcat 7. Como a Apache disponibiliza o mesmo através de um plug-in para projetos desenvolvidos com o Maven, a sua instalação no Eclipse é bastante simples. Basta adicionar o bloco de código referente ao plug-in do Tomcat no *pom.xml* e executar o projeto definindo o campo *Goals* com o valor *tomcat7:run*, como podemos verificar na janela *Run Configurations* do Eclipse (vide **Figura 6**). Feito isso o Maven irá baixar todos os arquivos necessários para o repositório local e depois irá levantar o servidor no endereço *localhost:8080*. As linhas 68 a 78 mostram como configurar esse plug-in, onde informamos o Group Id, Artifact Id, version e o path para formar a URL a ser acessada pelas requisições HTTP.

Realizadas as configurações, execute o projeto para verificar o resultado no browser. Durante a compilação do projeto você pode acompanhar o download do plug-in do Tomcat 7 através do console do Eclipse. Ao final será apresentada a URL do servidor. A partir de então podemos informar a mesma no browser para verificar a aplicação rodando, como demonstra a **Figura 7**.

## Diagrama Entidade Relacionamento da aplicação

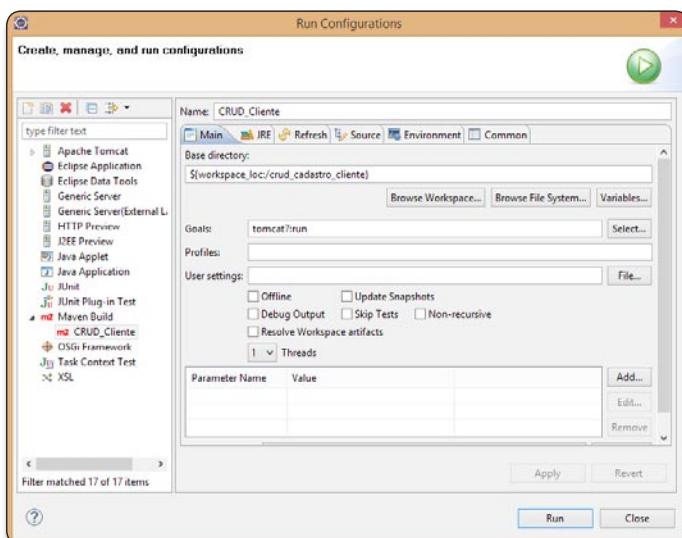
Para desenvolver o Diagrama Entidade Relacionamento (DER) da aplicação e posteriormente gerar a(s) tabela(s) no banco de dados,

**Listagem 1.** Arquivo pom.xml com as dependências do projeto.

```

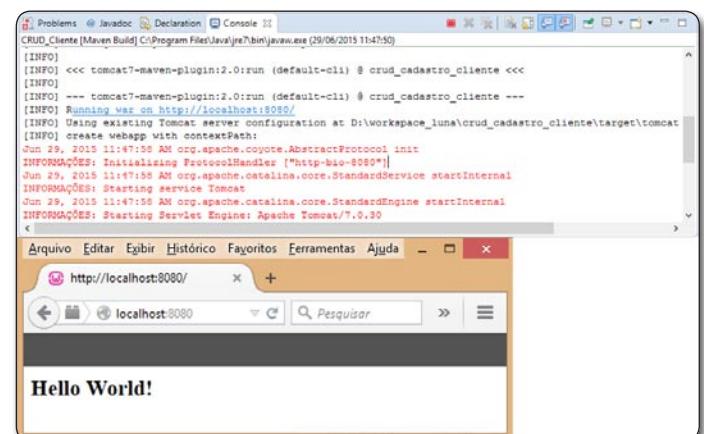
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
02   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0.xsd">
03 <modelVersion>4.0.0</modelVersion>
04 <groupId>br.com.devmedia</groupId>
05 <artifactId>crud_cadastro_cliente</artifactId>
06 <packaging>war</packaging>
07 <version>0.0.1-SNAPSHOT</version>
08 <name>crud_cadastro_cliente Maven Webapp</name>
09 <url>http://maven.apache.org</url>
10 <dependencies>
11 <dependency>
12   <groupId>junit</groupId>
13   <artifactId>junit</artifactId>
14   <version>3.8.1</version>
15   <scope>test</scope>
16 </dependency>
17 <dependency>
18   <groupId>com.sun.jersey</groupId>
19   <artifactId>jersey-server</artifactId>
20   <version>1.8</version>
21 </dependency>
22 <dependency>
23   <groupId>com.sun.jersey</groupId>
24   <artifactId>jersey-json</artifactId>
25   <version>1.8</version>
26 </dependency>
27 <dependency>
28   <groupId>org.hibernate</groupId>
29   <artifactId>hibernate-validator</artifactId>
30   <version>4.2.0.Final</version>
31 </dependency>
32 <dependency>
33   <groupId>org.hibernate.common</groupId>
34   <artifactId>hibernate-commons-annotations</artifactId>
35   <version>4.0.1.Final</version>
36   <classifier>tests</classifier>
37 </dependency>
38 <dependency>
39   <groupId>org.hibernate.jaxb.persistence</groupId>
40   <artifactId>hibernate-jpa-2.0-api</artifactId>
41   <version>1.0.1.Final</version>
42 </dependency>
43 <dependency>
44   <groupId>org.hibernate</groupId>
45   <artifactId>hibernate-entitymanager</artifactId>
46   <version>4.0.1.Final</version>
47 </dependency>
48 <dependency>
49   <groupId>mysql</groupId>
50   <artifactId>mysql-connector-java</artifactId>
51   <version>5.1.6</version>
52   <scope>compile</scope>
53 </dependency>
54 </dependencies>
55 <build>
56   <finalName>crud_cadastro_cliente</finalName>
57   <plugins>
58     <plugin>
59       <groupId>org.apache.tomcat.maven</groupId>
60       <artifactId>tomcat7-maven-plugin</artifactId>
61       <version>2.0</version>
62       <configuration>
63         <path>/</path>
64         <port>8080</port>
65       </configuration>
66     </plugin>
67   </plugins>
68 </build>
69 </project>

```



**Figura 6.** Configurando o build do projeto

Iremos utilizar a ferramenta CASE DBDesigner Fork, uma solução gratuita que gera scripts SQL para diversos SGBDs, como MySQL, SQL Server, Oracle, SQLite, PostgreSQL, dentre outros. Você pode realizar o download do DBDesigner através do endereço indicado na seção [Links](#).



**Figura 7.** Projeto cadastro de clientes em execução no browser

A **Figura 8** apresenta o DER da nossa aplicação sendo composto por apenas uma tabela, de nome *cliente*, responsável por armazenar os dados dos clientes a serem cadastrados.

Para gerar o script SQL para o MySQL, clique no menu *File > Export > SQL Create Script*. Feito isso, será aberta uma nova tela conforme apresentado na **Figura 9**. Então, selecione a opção *MySQL* no *Target Data Base* e clique no botão *Copy script to Clipboard* para copiar o script. Logo após, abra alguma ferramenta de gerência

# Criando um CRUD RESTful com Jersey, JPA e MySQL

do banco de dados, como o MySQL Workbench, crie um banco de dados chamado “bd\_cliente” e execute o script da **Listagem 2**, gerado pelo DBDesigner Fork.

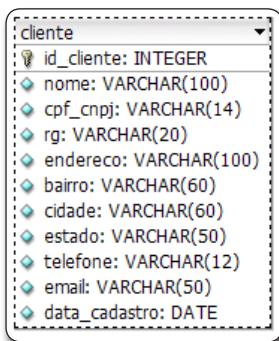


Figura 8. DER do banco de dados cadastro de clientes

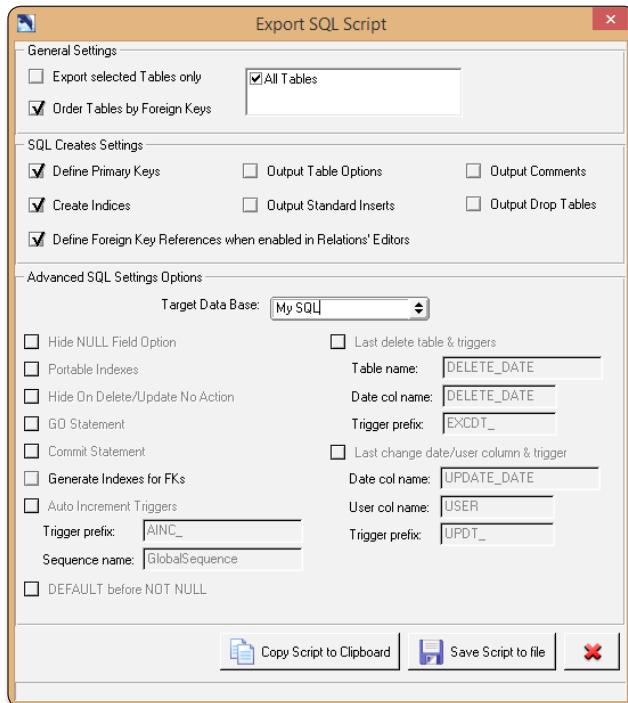


Figura 9. Gerando script SQL

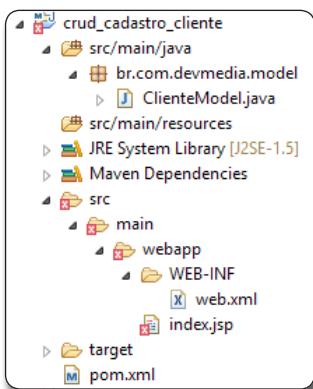


Figura 10. Estrutura do projeto

## Listagem 2. Script SQL para criar a tabela cliente.

```
CREATE TABLE cliente (
    id_cliente INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
    nome VARCHAR(100) NULL ,
    cpf_cnpj VARCHAR(14) NULL ,
    rg VARCHAR(20) NULL ,
    endereco VARCHAR(100) NULL ,
    bairro VARCHAR(60) NULL ,
    cidade VARCHAR(60) NULL ,
    estado VARCHAR(50) NULL ,
    telefone VARCHAR(12) NULL ,
    email VARCHAR(50) NULL ,
    data_cadastro DATE NULL ,
    PRIMARY KEY(id_cliente));
```

## Criando a entidade cliente

Para tornar possível a manipulação de clientes na aplicação é necessária ter uma classe que viabilize essa abstração. A partir dela podemos receber os dados do cliente armazenados no banco de dados, exibir na interface do usuário e vice-versa.

Sendo assim, vamos criar uma classe de nome **ClienteModel** para representar a entidade cliente. Nesta classe declararemos os atributos referentes às colunas da tabela *cliente* do banco de dados, processo este que é conhecido como mapeamento objeto relacional. Para isso, primeiro criaremos uma pasta chamada *java* dentro da pasta *main* do projeto. É importante seguir estas instruções porque caso você crie uma pasta com um nome diferente de *java*, o Jersey e o Hibernate podem não encontrar as classes **ClienteModel** e **ClienteService**, que serão criadas para expor o serviço REST com as operações do CRUD.

Para criar a pasta *java*, clique com o botão direito na pasta *main* e depois na opção *New > Folder*. Em seguida, defina o nome como *java* e clique em *Finish*. Agora, clique com o botão direito na pasta *src/main/java* e depois, ao selecionar a opção *New > Class*, defina o nome da classe como **ClienteModel**, no pacote **br.com.devmedia.model**, e clique em *Finish*.

Realizado este procedimento, teremos o projeto com a estrutura apresentada na **Figura 10**.

Criada a classe **ClienteModel**, você deve implementá-la conforme o código apresentado na **Listagem 3**.

Como podemos verificar, esse código tem algumas particularidades e anotações da JPA que merecem destaque. Assim, a seguir analisamos os detalhes dessa implementação:

- **Linha 13:** Nesta linha é informada a anotação **@Entity** da JPA. Ela é responsável por definir que a classe **ClienteModel** é uma estrutura de mapeamento objeto relacional e corresponde a uma referência a uma tabela na base de dados;
- **Linha 14:** Aqui é definida a anotação **@Table** com a propriedade **name** especificada com o valor “cliente”. Deste modo, informamos que a classe **ClienteModel** corresponde à tabela de nome *cliente* do banco de dados;
- **Linhas 16 a 18:** Neste intervalo foi definida a propriedade **id\_cliente** com as anotações **@Id**, para especificar que é chave primária, e **@GeneratedValue**, para especificar que o valor da chave primária deve ser gerado automaticamente pelo banco de dados;

**Listagem 3.** Código da classe ClienteModel.

```
01 package br.com.devmedia.model;
02
03 import java.util.Date;
04 import javax.persistence.Column;
05 import javax.persistence.Entity;
06 import javax.persistence.GeneratedValue;
07 import javax.persistence.GenerationType;
08 import javax.persistence.Id;
09 import javax.persistence.Table;
10 import javax.persistence.Temporal;
11 import javax.persistence.TemporalType;
12
13 @Entity
14 @Table(name="cliente")
15 public class ClienteModel {
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     int id_cliente;
19
20     @Column(name="nome")
21     String nome;
22
23     @Column(name="cpf_cnpj")
24     String cpf_cnpj;
25
26     @Column(name="rg")
27     String rg;
28
29     @Column(name="endereco")
30     String endereco;
31
32     @Column(name="bairro")
33     String bairro;
34
35     @Column(name="cidade")
36     String cidade;
37
38     @Column(name="estado")
39     String estado;
40
41     @Column(name="email")
42     String email;
43
44     @Column(name="dataCadastro")
45     @Temporal(TemporalType.TIMESTAMP)
46     Date dataCadastro;
47
48     public int getId_cliente() {
49         return id_cliente;
50     }
51
52     public void setId_cliente(int id_cliente) {
53         this.id_cliente = id_cliente;
54     }
55
56     public String getNome() {
57         return nome;
58     }
59
60 }
```

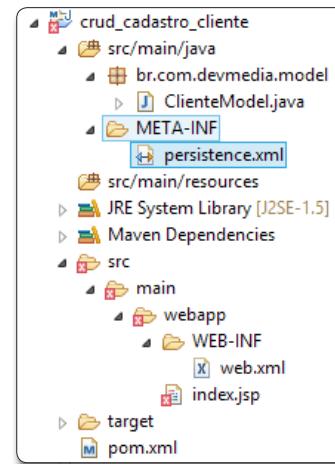
- **Linhas 20 a 42:** Neste intervalo são determinadas as demais propriedades da classe **ClienteModel**. Perceba que todas são anotadas com **@Column** para informar que representam colunas na tabela *cliente* da base de dados;
- **Linha 45:** Esta linha expõe a anotação **@Temporal** para a propriedade **data\_cadastro**. Assim, é definido que este campo irá trabalhar com valores no formato data;

- **Linhas 48 a 58:** Neste intervalo são implementados os métodos de acesso (getters e setters) de algumas propriedades da classe **ClienteModel**. Você pode gerar estes métodos automaticamente no Eclipse. Para isso, basta clicar com o botão direito na opção *Source > Generate Getters and Setters* e marcar todos os atributos para os quais deseja gerar os respectivos métodos.

## Criando e configurando o persistence.xml

Para que nossa aplicação de cadastro de clientes consiga realizar o acesso a dados, devemos criar o arquivo *persistence.xml*. Este possui as configurações utilizadas pela JPA para viabilizar a comunicação com o banco de dados. Tal arquivo deve ser criado na pasta de nome *META-INF*, que por sua vez deve ser criada dentro da pasta *java* para que as classes **EntityManagerFactory** e **EntityManager** da JPA possam encontrá-lo e assim fazer uso das configurações informadas.

Dito isso, crie a pasta *META-INF* dentro de *java* e depois crie o arquivo *persistence.xml*. Após este procedimento a estrutura de pastas do projeto deve estar semelhante à apresentada na **Figura 11**.



**Figura 11.** Estrutura de pastas para inclusão do arquivo *persistence.xml*

Com o *persistence.xml* em mãos, podemos configurá-lo para informar a classe **ClienteModel**, o provedor de acesso a dados, o dialeto do SGBD e os dados para conexão com o banco de dados, ou seja, o endereço do banco, o usuário e senha. A **Listagem 4** mostra como deve ficar esse arquivo.

Vejamos os detalhes da configuração do *persistence.xml*:

- **Linha 6:** Nesta linha é informado o **persistence-unit**, a base para iniciar a configuração do *persistence.xml*. Perceba que a propriedade **name**, na mesma linha, recebeu o valor “**app\_crud\_cliente**”. Portanto, é este nome que informaremos na configuração do **EntityManagerFactory** que iremos criar mais adiante no artigo, para que possa ser identificada a configuração do *persistence.xml* e assim criar o **EntityManager** para acesso ao banco de dados;
- **Linha 7:** Aqui é informado o provedor ORM que implementa o JPA, neste caso o Hibernate;
- **Linha 8:** Nesta linha é informada a classe **ClienteModel**, responsável pelo mapeamento com o banco de dados;

# Criando um CRUD RESTful com Jersey, JPA e MySQL

- **Linhas 9 a 15:** Neste bloco de código são informadas as propriedades de conexão com o banco de dados, como o driver do MySQL, a URL de acesso ao servidor do banco de dados seguida pelo nome do banco, o usuário e a senha.

Listagem 4. Código do arquivo persistence.xml.

```
01 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
02 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
04 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
05 version="2.0">
06   <persistence-unit name="app_crud_cliente">
07     <provider>org.hibernate.ejb.HibernatePersistence</provider>
08     <class>br.com.devmedia.model.ClienteModel</class>
09     <properties>
10       <property name="javax.persistence.jdbc.driver"
11         value="com.mysql.jdbc.Driver"/>
12       <property name="javax.persistence.jdbc.url"
13         value="jdbc:mysql://localhost/bd_cliente"/>
14       <property name="javax.persistence.jdbc.user" value="root"/>
15       <property name="javax.persistence.jdbc.password" value="root"/>
16       <property name="hibernate.dialect"
17         value="org.hibernate.dialect.MySQLInnoDBDialect"/>
18     </properties>
19   </persistence-unit>
20 </persistence>
```

## Criando o EntityManager

O próximo passo é criar a classe que tenha um método responsável por retornar um objeto do tipo **EntityManager**. É o **EntityManager** que disponibiliza todos os métodos que precisamos para conseguir acesso ao banco de dados e que veremos mais à frente. Sendo assim, criaremos uma classe, chamada **JpaEntityManager**, para gerenciar a conexão com o banco de dados MySQL. Dentro da mesma devemos declarar dois objetos para setar a fábrica de objetos, isto é, setar **EntityManagerFactory** com as configurações do arquivo *persistence.xml*. Assim, podemos obter uma instância do **EntityManager** para, de fato, chamar seus métodos e realizar as operações do CRUD.

Portanto, crie a classe **JpaEntityManager** no pacote **br.com.devmedia.EntityManager**. Seu código fonte é apresentado na [Listagem 5](#).

Listagem 5. Código da classe JpaEntityManager.

```
01 package br.com.devmedia.EntityManager;
02
03 import javax.persistence.EntityManager;
04 import javax.persistence.EntityManagerFactory;
05 import javax.persistence.Persistence;
06
07 public class JpaEntityManager {
08
09   private EntityManagerFactory factory = Persistence.
10     createEntityManagerFactory("app_crud_cliente");
11
12   private EntityManager em = factory.createEntityManager();
13
14   public EntityManager getEntityManager(){
15     return em;
16   }
17 }
```

Como podemos verificar, ele é bem simples. Vejamos os seus detalhes:

- **Linha 9:** Nesta linha é criado um objeto chamado **factory** do tipo **EntityManagerFactory**. Este foi definido para receber uma instância do próprio **EntityManagerFactory**. Para isso, chamamos o método **createEntityManagerFactory()** da classe **Persistence**, que recebe como parâmetro o nome “*app\_crud\_cliente*”, nome este que foi informado no *persistence.xml*, na propriedade **persistence-unit**;
- **Linha 10:** Neste trecho foi criado mais um objeto, chamado **em** e do tipo **EntityManager**. O mesmo recebe uma instância do **EntityManager** através da chamada ao método **createEntityManager()** do objeto **factory** criado anteriormente;
- **Linhas 12 a 14:** Neste bloco foi definido um método chamado **getEntityManager()**. Este retorna um objeto **EntityManager** para realizar as operações do CRUD no banco de dados através de seus métodos.

## Definindo o servlet do Jersey no web.xml

Em toda aplicação Java EE precisamos criar um arquivo chamado *web.xml*, local onde podemos configurar o servlet que irá receber as requisições HTTP. Neste arquivo também são definidas as informações sobre o serviço REST, pacotes, classes do serviço, dentre outras configurações que fogem do escopo deste artigo.

O nosso *web.xml* deve ser configurado conforme o código da [Listagem 6](#), onde são informados os dados da API do Jersey, isto é, onde é definido que o servlet do Jersey que será o responsável por receber as requisições HTTP referentes às chamadas ao serviço.

Listagem 6. Definição do servlet do Jersey no arquivo web.xml.

```
01 <!DOCTYPE web-app PUBLIC
02 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
03 "http://java.sun.com/dtd/web-app_2_3.dtd">
04
05 <web-app>
06   <display-name>Archetype Created Web Application</display-name>
07
08   <servlet>
09     <servlet-name>CRUD Cadastro de Clientes</servlet-name>
10     <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer
11   </servlet-class>
12   <init-param>
13     <param-name>com.sun.jersey.config.property.packages</param-name>
14     <param-value>br.com.devmedia.service</param-value>
15   </init-param>
16   <init-param>
17     <param-name>com.sun.jersey.api.json.POJOMappingFeature
18     </param-name>
19     <param-value>true</param-value>
20   </init-param>
21   <load-on-startup>1</load-on-startup>
22 </servlet>
23
24 <servlet-mapping>
25   <servlet-name>CRUD Cadastro de Clientes</servlet-name>
26   <url-pattern>/apirest/*</url-pattern>
27 </servlet-mapping>
28 </web-app>
```

Para entender melhor esse código XML, vejamos sua análise:

- **Linha 10:** Nesta linha é definida a classe que representa o servlet da aplicação. Neste caso foi informado **ServletContainer**, do Jersey;
- **Linhas 11 a 14:** Neste bloco é informado o pacote que irá conter as classes que representam o serviço e contêm os métodos a serem expostos. Neste caso foi informado o pacote **br.com.devmedia.service**, que ainda vamos criar;
- **Linhas 16 a 19:** Neste bloco são definidos outros parâmetros para o serviço. Dessa vez para a API de JSON do Jersey, para que ela possa realizar o mapeamento que viabiliza a conversão das requisições no formato JSON em classes Java. O mesmo vale para as respostas do servidor ao cliente, na transformação objeto Java → JSON;
- **Linhas 24 a 27:** Neste bloco é configurada a URL para acesso aos recursos do servidor, ou melhor, às operações do CRUD a serem expostas pelo serviço. Através da tag **url-pattern** é definido o caminho padrão de acesso às URLs do serviço, como: *http://localhost:8080/apirest/cliente/cadastrar*.

## Implementando o serviço REST

Chegou o momento de implementar a classe que irá expor o serviço REST e consequentemente as operações CRUD para o cadastro de clientes. Para desenvolver esse serviço, crie uma classe denominada **ClienteService** no pacote **br.com.devmedia.service**. Após esse passo, a estrutura do projeto deve estar conforme a **Figura 12**. O código da classe **ClienteService** é apresentado na **Listagem 7**.

Observando essa imagem podemos notar uma organização na estrutura dos pacotes do projeto. Sendo assim, há um pacote para representar as classes que representam as entidades, outro para as classes de serviço e, por fim, o pacote que encapsula o acesso dados. Desta forma o projeto se torna bem estruturado e com responsabilidades bem definidas, facilitando possíveis atualizações e manutenções.

Neste código podemos verificar que os verbos do protocolo HTTP (GET, POST, PUT e DELETE) são utilizados para cada operação do CRUD. O verbo POST, por exemplo, é empregado

**Listagem 7.** Código do serviço REST – classe ClienteService.

```
01 package br.com.devmedia.service;
02
03 @Path("/cliente")
04 public class ClienteService {
05     private JpaEntityManager JPAEM = new JpaEntityManager();
06     private EntityManager objEM = JPAEM.getEntityManager();
07
08     @GET
09     @Path("/listar")
10    @Produces("application/json")
11    public List<ClienteModel> listar(){
12
13        try {
14            String query = "select c from ClienteModel c";
15            List<ClienteModel> clientes = objEM.createQuery(query, ClienteModel.class).
16                getResultList();
17            objEM.close();
18            return clientes;
19        } catch (Exception e) {
20            throw new WebApplicationException(500);
21        }
22
23        @GET
24        @Path("/buscar/{id_cliente}")
25        @Produces("application/json")
26        public ClienteModel buscar(@PathParam("id_cliente") int id_cliente){
27            try {
28                ClienteModel cliente = objEM.find(ClienteModel.class, id_cliente);
29                objEM.close();
30                return cliente;
31            } catch (Exception e) {
32                throw new WebApplicationException(500);
33            }
34        }
35
36        @POST
37        @Path("/cadastrar")
38        @Consumes("application/json")
39        public Response cadastrar(ClienteModel objClinte){
40            try {
41                objEM.getTransaction().begin();
```

```
42                objEM.persist(objClinte);
43                objEM.getTransaction().commit();
44                objEM.close();
45                return Response.status(200).entity("cadastro realizado.").build();
46            } catch (Exception e) {
47                throw new WebApplicationException(500);
48            }
49        }
50
51        @PUT
52        @Path("/alterar")
53        @Consumes("application/json")
54        public Response alterar(ClienteModel objClinte){
55            try {
56                objEM.getTransaction().begin();
57                objEM.merge(objClinte);
58                objEM.getTransaction().commit();
59                objEM.close();
60                return Response.status(200).entity("cadastro alterado.").build();
61            } catch (Exception e) {
62                throw new WebApplicationException(500);
63            }
64        }
65
66        @DELETE
67        @Path("/excluir/{id_cliente}")
68        public Response excluir(@PathParam("id_cliente") int id_cliente){
69            try {
70                ClienteModel objClinte = objEM.find(ClienteModel.class, id_cliente);
71
72                objEM.getTransaction().begin();
73                objEM.remove(objClinte);
74                objEM.getTransaction().commit();
75                objEM.close();
76
77                return Response.status(200).entity("cadastro excluído.").build();
78            } catch (Exception e) {
79                throw new WebApplicationException(500);
80            }
81        }
82    }
```

# Criando um CRUD RESTful com Jersey, JPA e MySQL

para a operação relacionada ao cadastro de clientes. Para alteração dos dados foi adotado o PUT, DELETE para exclusão e GET para obter a listagem dos clientes. No código ainda podemos verificar algumas anotações do JAX-RS e o uso de recursos da JPA.

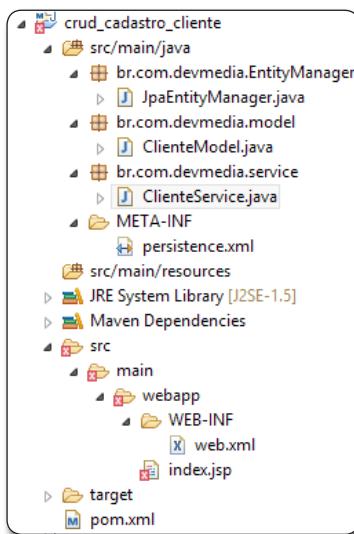


Figura 12. Estrutura do projeto após criar a classe ClienteService

A seguir são apresentados os primeiros detalhes da classe **ClienteService**, que será analisada mais a fundo nos próximos tópicos:

• **Linha 03:** Local onde definimos que a classe **ClienteService** será um serviço. Para isso foi inserida a anotação **@Path** passando como parâmetro o nome do serviço que irá compor a URL de acesso aos recursos do servidor. A partir disso, o caminho para acesso ao serviço será: <http://localhost:8080/apirest/cliente/>. Lembre-se que **apirest** e a porta foram definidos no mapeamento do servlet do Jersey no **web.xml**;

- **Linha 05 e 06:** Na linha 5 foi criado um objeto da classe **JpaEntityManager**. Logo depois, na linha 6 é criado outro objeto, chamado **objEM**. Este recebe um **EntityManager** através da chamada ao método **getEntityManager()** de **JpaEntityManager**. É o **objEM** que irá possibilitar o acesso a dados.

Para melhor entendimento e mostrar na prática o consumo dos métodos (recursos) do nosso serviço de cadastro de cliente, iremos instalar um complemento do Firefox chamado de **Http requester**. Com esse intuito, acesse a opção **Complementos** deste navegador, procure por “**Http requester**” e então clique em **Instalar**. Este possibilita realizar requisições a serviços REST utilizando vários verbos do protocolo HTTP.

## Consumo do recurso listar clientes

O primeiro recurso a ser implementado será o de listagem dos dados. Desta forma, começaremos analisando o código relacionado à listagem (vide método **listar()** da **Listagem 7**) e como consumir este serviço.

A implementação deste método, equivalente à operação Read do CRUD, é explicada a seguir:

- **Linha 08 a 21:** Este bloco de código define o primeiro método do serviço, o **listar()**, que retorna uma lista de **ClienteModel**. Na linha 08 foi utilizada a anotação **@GET** para informar que este método só poderá ser requisitado por uma requisição do tipo GET no protocolo HTTP. Na linha 09 é informada a anotação **@Path**, que recebe por parâmetro o nome do recurso do serviço; neste caso o nome final da URL que aponta para o recurso de listagem de clientes no servidor, **listar**. Vale ressaltar que o valor da propriedade **name** de **@Path** não precisa ser o mesmo nome do método iniciado na linha 11, mas deixamos o mesmo apenas para facilitar o entendimento. Na linha 10 foi especificada a última anotação do método **listar**: **@Produces**. Esta serve para informar ao Jersey que ele deve retornar ao cliente a listagem da linha 17 no formato JSON;

- **Linha 13 a 20:** Esta é a implementação do código que efetivamente irá buscar a listagem de clientes, onde é definida uma **String** chamada **query** que recebe a consulta a ser realizada no banco de dados. Em seguida, na linha 15 o objeto **clientes** recebe o resultado da consulta através da chamada aos métodos **createQuery()** e **getResultSet()**, disponíveis no objeto **EntityManager**, e na linha 16 é encerrado o **EntityManager** chamando o método **close()**. Por fim, na linha 19 é criado um **throw** através da classe **WebApplicationException** para recuperarmos possíveis erros durante a execução e informar ao solicitante do serviço.

Na **Figura 13** expomos o resultado de uma requisição do tipo GET ao recurso **listar clientes**. Para isso, perceba que precisamos apenas

Figura 13. Consumo do recurso listar clientes

informar a URL do serviço e selecionar o verbo GET. Antes de realizar o teste, no entanto, abra algum gerenciador de banco de dados MySQL e insira alguns clientes na tabela *cliente*.

### Consumo do recurso buscar cliente

Outro importante recurso do nosso serviço é a busca de clientes na base de dados pelo id. Relacionado à operação Read do CRUD, a requisição a este recurso geralmente é realizada através de uma requisição do tipo GET.

No código do projeto exemplo, o método **buscar()** recebe o **id** do cliente a ser pesquisado e retorna os dados do mesmo no formato JSON, quando encontrado. Façamos uma análise do seu código:

- Linhas 23 a 26:** Nestas linhas são declaradas as anotações **@GET**, **@Path** e **@Produces** para indicar, respectivamente, o tipo de método HTTP, o caminho de acesso ao recurso e o tipo dos dados que serão retornados (JSON, neste caso). Perceba ainda que na linha 24 é informado um parâmetro chamado **id\_cliente** para receber o código do cliente na requisição GET. É através deste parâmetro que é obtido o código a ser pesquisado no banco de dados;

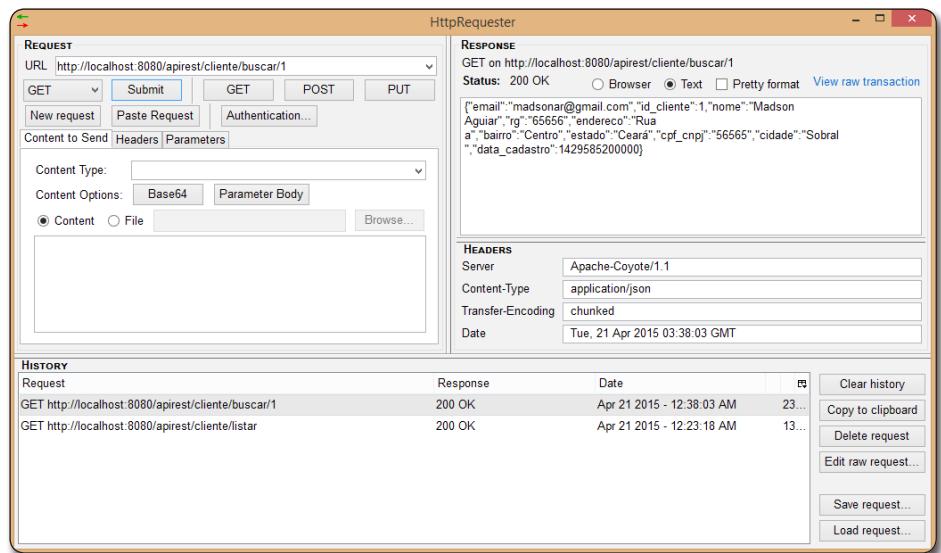
- Linha 26:** Aqui temos a declaração do método **buscar()**, local onde também fazemos uso da anotação **@PathParam**, que recebe o nome do parâmetro a ser passado com o código do cliente junto à URL de requisição;

- Linha 28:** Nesta linha é criado um objeto do tipo **ClienteModel** para receber os dados do cliente pesquisado na base de dados. Veja que a consulta foi realizada através do método **find()**, que recebeu o tipo do model a ser pesquisado e o id do cliente. Nas linhas 29 e 30 o **EntityManager** é encerrado e é retornado o cliente pesquisado no formato JSON.

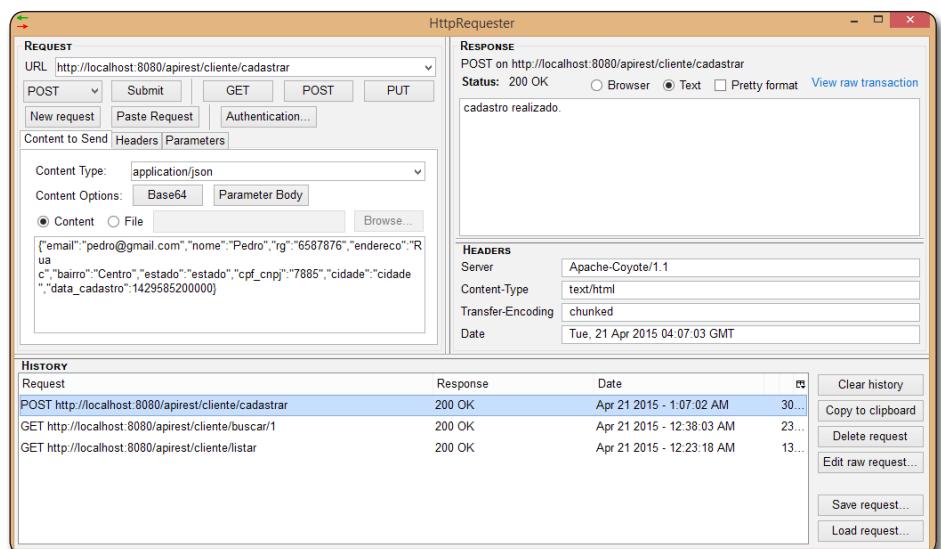
Na **Figura 14** podemos verificar o resultado do consumo do recurso buscar cliente através de uma requisição do tipo GET com a ferramenta Open HttpRequirer.

### Consumo do recurso cadastrar cliente

A próxima operação do CRUD que codificamos como um serviço é o Create, para cadastrar um novo cliente na base



**Figura 14.** Consumo do recurso buscar cliente



**Figura 15.** Consumo do recurso cadastrar cliente

de dados. Para isso, devemos realizar uma requisição do tipo POST passando no corpo da mesma os dados do novo cliente. Os detalhes de sua implementação são apresentados a seguir:

- Linha 38:** O primeiro ponto a destacar neste código é a presença de uma nova anotação, chamada **@Consumes**. Esta tem como função definir o tipo de dado a ser recebido pela requisição POST, neste caso o JSON;
- Linha 39:** Nesta linha temos a declaração do método, que como parâmetro recebe o objeto cliente a ser cadastrado no banco;
- Linhas 41 a 43:** Neste intervalo é iniciada uma nova transação com o banco de dados.

Na linha 42 um novo cliente é persistido. Por fim, é realizado um commit para que o mesmo seja gravado permanentemente;

- Linha 46:** Nesta linha é utilizado o método **status()** da classe **Response** para retornar ao cliente o código 200 do HTTP. Isto informa que o cliente foi cadastrado com sucesso.

Na **Figura 15** podemos verificar o resultado da requisição POST ao recurso cadastrar. Perceba que os dados do novo cliente estão no formato JSON, pois este é o formato esperado pelo servidor, conforme explicitado pela anotação **@Consumes("application/json")**.

# Criando um CRUD RESTful com Jersey, JPA e MySQL

## Consumo do recurso alterar cliente

Outra operação do nosso serviço, relacionada ao Update do CRUD, é a alteração dos dados de um cliente. Como esperado, o método relacionado altera os dados do cliente e depois retorna uma mensagem informando o sucesso da execução. Analisemos o seu código:

- **Linha 51:** Nesta linha é informada a anotação @PUT, definindo que a requisi-

ção deve usar o método PUT do protocolo HTTP;

- **Linha 53:** Aqui é especificada a anotação @Consumes para informar que os dados recebidos pela requisição devem estar no formato JSON. Deste modo o Jersey poderá fazer a conversão para objetos Java;
- **Linhas 56 a 58:** Neste intervalo é iniciada uma nova transação (vide linha 56). Em seguida os dados do cliente são alterados

na base através da chamada ao método merge() do EntityManager, que recebe por parâmetro o cliente a ser atualizado. Por fim, na linha 58 é realizado o commit para persistir essa mudança;

- **Linha 60:** Nesta linha é utilizada novamente a classe Response, que através do método status() retorna o código 200 informando que o cadastro do cliente foi alterado com sucesso.

Na **Figura 16** é possível verificar o consumo do recurso que altera as informações de um cliente no banco de dados.

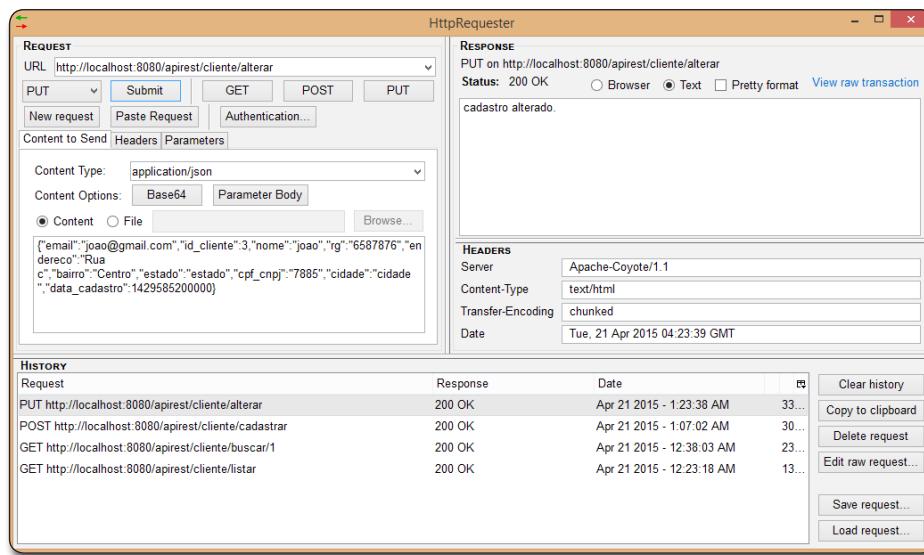


Figura 16. Consumo do recurso alterar cliente.

## Conhecimento faz diferença!

The advertisement highlights several articles:

- Edição 28 :: Ano 2: Gerência de Configuração - Definição + Ferramentas
- Edição 29 :: Ano 3: Agilidade: Negociação de contratos em projeto
- Edição 29 :: Ano 3: engenharia de software magazine
- Edição 29 :: Ano 3: SOA - Processo e levantamento de requisitos de negócios - Parte 2
- Edição 29 :: Ano 3: Qualidade de Software - Definição, características e importância
- Edição 29 :: Ano 3: Automação de Testes - Definições, preocupações e custo
- Edição 29 :: Ano 3: Cuidados a serem tomados na implantação
- Edição 29 :: Ano 3: Estratégia de Teste Funcional baseada em Casos de Uso - Partes 5 a 9
- Aulas desta edição: + Atividades da semana
- + Aulas desta edição: + Atividades da semana
- + de 290 vídeos para assinantes

Faça já sua assinatura digital! | [www.devmedia.com.br/es](http://www.devmedia.com.br/es)

## Faça um upgrade em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**



- **Linha 70:** Nesta linha podemos verificar a busca no bando de dados do cliente a ser excluído;
- **Linhas 72 a 74:** Neste intervalo declaramos uma transação que exclui o cliente do banco de dados através da chamada ao método **remove()** de **EntityManager**. Como parâmetro este método recebe o cliente. Na linha 74 efetuamos o **commit()**;
- **Linha 77:** Por fim, é retornado o código 200 para indicar que o cliente foi excluído com sucesso.

Veja na **Figura 17** o resultado do consumo do recurso relacionado à exclusão de um cliente no banco de dados.

O desenvolvimento de aplicações que adotam o padrão arquitetural REST segue em alta e provavelmente continuará assim até que uma solução mais simples e leve seja apresentada à comunidade. Deste modo, se você ainda não está utilizando esta solução em suas aplicações que requerem web services, este artigo serve como um ótimo ponto de partida para isso.

Com a necessidade de soluções cada vez mais integradas, o uso de serviços já se tornou uma forte vertente no mundo do desenvolvimento. Conhecer e saber como implementá-los é, portanto, um requisito essencial a todo profissional que atua na área.

## Autor



**Madson Aguiar Rodrigues**

[madsonar@gmail.com](mailto:madsonar@gmail.com)



Formado em Análise e Desenvolvimento de Sistemas pela UNOPAR, pós-graduado em Engenharia de Sistemas pela ESAB e especialista em Tecnologias para aplicações Web pela UNOPAR.

Trabalha com desenvolvimento de software há sete anos com uso da plataforma .Net, Java e Mobile com Android. Atua no mercado com prestação de serviços e consultoria em TI, fornecendo soluções em software, tutor do curso de graduação em Análise e Desenvolvimento de Sistemas na UNOPAR e autor de artigos e vídeos no portal DevMedia.

The screenshot shows the 'REQUEST' tab with the URL 'http://localhost:8080/apirest/cliente/excluir/3' and the 'DELETE' button selected. The 'RESPONSE' tab shows a 200 OK status with the message 'cadastro excluido.'. The 'HEADERS' tab lists the server as Apache-Coyote/1.1, Content-Type as text/html, Transfer-Encoding as chunked, and Date as Tue, 21 Apr 2015 04:39:55 GMT. The 'HISTORY' tab shows a list of previous requests and responses.

Request	Response	Date
DELETE http://localhost:8080/apirest/cliente/excluir/3	200 OK	Apr 21 2015 - 1:39:55 AM
PUT http://localhost:8080/apirest/cliente/alterar	200 OK	Apr 21 2015 - 1:23:38 AM
POST http://localhost:8080/apirest/cliente/cadastrar	200 OK	Apr 21 2015 - 1:07:02 AM
GET http://localhost:8080/apirest/cliente/buscar/1	200 OK	Apr 21 2015 - 12:38:03 AM
GET http://localhost:8080/apirest/cliente/listar	200 OK	Apr 21 2015 - 12:23:18 AM

Figura 17. Consumo do recurso excluir cliente

## Links:

### Endereço para download do Eclipse Luna.

<https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunasr2>

### Página do projeto DBDesigner Fork.

<http://sourceforge.net/projects/dbdesigner-fork/>

### Dissertação de Roy Fielding.

[https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation.pdf](https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf)

### Página do projeto Jersey.

<https://jersey.java.net/>

### Endereço para download do MySQL.

<http://www.mysql.com/downloads/>

### Java Persistence API.

<http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html>

### What are RESTful Web Services?

<https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html>

## Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/esmag/feedback](http://www.devmedia.com.br/esmag/feedback)

Ajude-nos a manter a qualidade da revista!



# 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](http://porta80.com.br) | [comercial@porta80.com.br](mailto:comercial@porta80.com.br) | [twitter.com/porta80](http://twitter.com/porta80)

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