



Edição 137 :: R\$ 14,90

Aplicações corporativas com JSF, JPA e CDI

Desenvolvendo um sistema para  
gerenciamento de bibliotecas

Avalie a arquitetura de seus sistemas

Conheça o método DCAR e  
aprimore a arquitetura do seu sistema

 DEV MEDIA

# SPRING + BOOTSTRAP

Construindo aplicações responsivas

ISSN 1676836-1



---

Programação funcional  
Domine este “novo” mundo  
e seu suporte no Java 8

---

Spring Framework  
Interface REST e estabeleça  
a comunicação com WebSockets

---

Explorando o Elasticsearch  
Prepare seus mecanismos de  
busca para lidar com Big Data

# 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!



 **DEVMEDIA**



Edição 137 • 2015 • ISSN 1676-8361



MVP

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

### Produção

**Jornalista Responsável** Kaline Dolabella - JP24185

**Capa** Romulo Araujo

**Diagramação** Janete Feitosa

### Distribuição

FC Comercial e Distribuidora S.A.

Rua Teodoro da Silva, 907, Grajaú - RJ

CEP 20563-900, (21) 3879-7766 - (21) 2577-6362

### 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.

*Java, o logotipo da xícara de café Java e todas as marcas e logotipos baseados em/ou referentes a Java são marcas comerciais ou marcas registradas da Sun Microsystems, Inc. nos Estados Unidos e em outros países.*

### 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 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 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



# Sumário

Conteúdo sobre Novidades

## 06 – Programação funcional com Java

[ Rômero Ricardo de Sousa Pereira ]

Artigo no estilo Curso

## 18 – Spring Framework: Criando uma aplicação web – Parte 2

[ Francisco A. Garcia ]

Conteúdo sobre Novidades, Artigo no estilo

## 28 – Spring MVC: Construa aplicações responsivas com Bootstrap – Parte 1

[ Pablo Bruno de Moura Nóbrega e Marcos Vinicios Turisco Dória ]

Artigo no estilo Curso

## 42 – Criando uma aplicação corporativa em Java – Parte 1

[ Bruno Rafael Sant'Ana ]

Conteúdo sobre Novidades

## 54 – Elasticsearch: realizando buscas no Big Data

[ Luiz Henrique Zambom Santana ]

Conteúdo sobre Engenharia de Software, Conteúdo sobre Boas Práticas

## 66 – Arquitetura de Software: Avaliando a arquitetura de seus sistemas

[ Regis Santos ]



Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

[www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)



## 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](mailto:toolscloud@toolscloud.com)



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



# Programação funcional com Java

Podemos pensar em um programa como uma estrutura sintática organizada de modo que, uma vez compilada e executada, gere um resultado semântico – neste caso, a execução de uma tarefa. Desta forma, percebe-se que a escrita de um software tem muito em comum com a escrita textual: para ambos os casos existem muitas maneiras de estruturar e transmitir uma ideia, chegando ao mesmo resultado. Sendo assim, como podemos identificar a melhor estrutura de escrita ou a melhor maneira de organizar um código? No campo da escrita textual, as recomendações estão sempre associadas a simplicidade, clareza e concisão, a fim de que o leitor consiga extrair o máximo de informação com o menor esforço de leitura. Na escrita de software, estas recomendações se encaixam perfeitamente através de códigos simples, limpos e concisos para a facilitação da leitura e obtenção de melhor performance na execução.

Desta forma, a evolução das linguagens de programação revela um esforço determinante nesta busca pelo melhor modelo de escrita de código, sendo a Orientação a Objetos um marco importante no estilo de codificação de softwares em comparação com os anteriores, contribuindo para esses três pilares com o reaproveitamento de código, reutilização de abstrações, acoplamento de componentes, encapsulamento, polimorfismo, entre outros. Até o Java 8 a programação funcional ainda era uma opção muito cara para o desenvolvimento em Java devido à natureza imperativa da linguagem que demandava muito esforço de estruturação, resultando muitas vezes em códigos menos legíveis. Agora esta nova versão está adicionando recursos importantes para facilitar o desenvolvimento de novas ideias através da introdução das sintaxes para expressões Lambda e Referências de Métodos, além da adição de classes utilitárias à API. Estas novidades flexibilizam a criação de códigos mais funcionais e oferecem ao desenvolvedor Java uma maneira diferente de expressar códigos com maior simplicidade e concisão, facilitando a sua manutenção e paralelização sem efeitos colaterais.

A programação funcional é um paradigma que visa estruturar a construção de softwares seguindo o modelo de funções matemáticas, objetivando a imutabilidade dos dados e o acoplamento de funções, que resultam

## Fique por dentro

O avanço tecnológico em diversos campos da sociedade tem exigido novas e mais modernas tecnologias para o acompanhamento das demandas mais complexas no desenvolvimento de softwares, bem como flexibilidade de manutenção, extensibilidade, adaptação para integração com outros sistemas e maior capacidade no processamento de grandes volumes de informações através de estruturas de sistemas mais eficientes.

A plataforma Java tem feito trabalhos importantes para auxiliar o desenvolvedor a suprir tais demandas. Neste artigo abordaremos as principais características da programação funcional, que se tornou uma possibilidade mais próxima do desenvolvedor Java após o lançamento do Java 8. Veremos como escrever um código mais conciso, declarativo e estético utilizando as novidades desta versão com a introdução de Lambdas, Referências de Métodos e as novas classes da API de suporte à programação funcional. Com estes novos recursos o desenvolvedor terá em suas mãos ferramentas eficientes para melhorar a qualidade do seu código e produzir aplicações com as características exigidas pelas demandas atuais.

em benefícios como consistência tanto do código quanto dos dados – veremos os porquês no decorrer do artigo. Semelhante a uma função matemática, um código funcional tem a sua saída condicionada exclusivamente à sua entrada, de modo que uma entrada garanta sempre a mesma saída após a execução do mesmo método, eliminando os chamados *side-effects*, que são as mudanças de estado ocasionadas por outros fatores que não dependam dos parâmetros do método.

As bases da programação funcional são da década de 1930, a partir dos estudos do matemático norte-americano Alonzo Church na aplicação de modelos de funções matemáticas em computação, mais precisamente no Cálculo de Lambdas. As primeiras aplicações práticas deste paradigma apareceram nas linguagens LISP (*List Processing*), quando elas começaram a ser influenciadas pelas ideias dos Cálculos de Lambda de Church.

Recentemente temos visto muitas linguagens que dão suporte à programação funcional emergindo e ganhando muitos desenvolvedores adeptos ao paradigma. Algumas delas rodando, inclusive, na JVM, como é o caso das linguagens Scala e Clojure.

A partir da versão 8 do Java, tanto a linguagem quanto a sua máquina virtual sofreram alterações consideráveis através de otimizações no núcleo da JVM e da modernização da linguagem

com a introdução de Lambdas e melhorias na API com a adição de novas classes para não só permitir, como também facilitar e otimizar a programação funcional. Com isso o Java ganha ainda mais flexibilidade e força para se manter no topo como uma das linguagens mais utilizadas no mundo.

Neste artigo conhceremos estes novos aspectos do Java 8. Para isso, serão apresentados e demonstrados os principais conceitos da programação funcional, suas aplicações e implementações utilizando a nova sintaxe e API introduzidas na linguagem.

## Mude o jeito de pensar

Para programar de forma funcional é preciso que o desenvolvedor mude a forma de pensar a construção dos seus algoritmos. Antes da versão 8, o modelo de programação na linguagem Java era imperativo; o que significa que para realizar uma iteração simples, por exemplo, o desenvolvedor tinha que se preocupar não somente com o que deveria ser feito com cada item da coleção iterada, mas também em como realizar a iteração e, em muitos casos, como filtrar os itens. Este modelo facilitava, por natureza, erros como variáveis não inicializadas ou que mudam no decorrer da iteração, possibilitando que a mesma iteração tivesse resultados diferentes em função das mutações de variáveis de controle. Além disso, esse modelo imperativo de programação dificulta a paralelização da tarefa efetivamente realizada dentro da iteração, visto que qualquer tentativa de paralelização pode envolver muitos controles de concorrência e de *thread safety*. Outra característica negativa do código imperativo é a sua legibilidade, já que a leitura do código envolve não somente decifrar a regra de negócio, mas também compreender as complexidades do controle de fluxo do algoritmo.

A programação funcional traz uma nova forma de pensar o fluxo do algoritmo ao introduzir o conceito de programação declarativa, cujos principais benefícios são a imutabilidade de variáveis, eliminação de *side-effects* – já que a garantia de que uma variável não vai mudar dentro de um fluxo assegura que não existirão efeitos colaterais no mesmo código quando paralelizado – e obtenção de códigos mais concisos. Um código conciso significa um código ao mesmo tempo breve, significativo e claro, de fácil legibilidade. Com todos estes aspectos reunidos, o resultado será um código muito mais fácil de manter e menos suscetível a erros.

## Como ser declarativo

Já que a palavra de ordem na programação funcional é programar de forma declarativa, torna-se necessário aprender como criar códigos que se utilizem desta técnica. Os aspectos de um código declarativo têm um paralelo bem conhecido dentro das melhores práticas do desenvolvimento orientado a objetos: o padrão de projeto *Strategy*. Na verdade, este padrão é uma das ideias-chave para a compreensão da programação funcional e, principalmente, sua implementação na linguagem Java, uma vez que todas as facilidades introduzidas no Java 8 são em essência implementações do padrão *Strategy* em conjunto com a utilização de Lambdas e Referências de Métodos –

ambos detalhados em seguida – para permitir que o código se torne ainda mais conciso; portanto, cabe uma breve explicação sobre este padrão de desenvolvimento.

Normalmente associamos as mudanças de estado de um objeto às alterações em atributos ao longo do seu ciclo de vida. Se pensarmos em uma abstração simples como um objeto do tipo **CarrinhoDeCompra**, podemos identificar a possibilidade de mudança em atributos como lista de produtos e quantidade. Sabemos que, conforme se adicionam produtos ao carrinho, a quantidade de produtos aumenta, o valor da compra aumenta e, portanto, o estado do objeto vai sofrendo alterações ao longo da execução de uma tarefa como a de efetuar compras.

Com o padrão *Strategy* torna-se possível criar atributos que modifiquem o funcionamento do objeto em vez de seu estado. Apesar de a mudança de estado estar indiretamente ligada à execução de um comportamento e vice-versa, a diferença criada pela utilização do *Strategy* é a possibilidade de atribuir e alterar outros comportamentos da mesma forma que fazemos com os estados do objeto; dando ao desenvolvedor a possibilidade de criar mecanismos para a atribuição de funcionalidades que nem mesmo ele tenha previsto no momento em que criou sua classe.

No exemplo do **CarrinhoDeCompra**, uma possibilidade de implementação do padrão *Strategy* seria a criação de um novo atributo do tipo **CalculadorDeDesconto**, cujas implementações pudessem ser decididas através do tipo de cliente. Assim o preço final do **CarrinhoDeCompra** poderia ser baseado neste novo atributo, cujo comportamento seria definir o valor do desconto. Desta forma, a mesma compra realizada por clientes diferentes poderia ter um valor final diferente com base no comportamento atribuído ao **CalculadorDeDesconto** devido ao tipo de cliente.

Esta possibilidade de parametrizarmos comportamento da mesma forma que parametrizamos estado (dados) tem papel fundamental na tarefa de programar de forma declarativa, uma vez que um código declarativo tem necessariamente esta característica de informar o comportamento esperado em uma função de um contexto mais amplo, sem se preocupar com o controle de fluxo da tarefa a ser realizada. O Java possui implementações muito conhecidas deste padrão, como a interface **java.util.Comparator**, que é utilizada no método **sort()** da classe utilitária **java.util.Collections**. Para exemplificar, digamos que um algoritmo deva ordenar uma lista de **Strings** pelo número de caracteres, dando prioridade às **Strings** com menor quantidade. Na **Listagem 1** podemos verificar a implementação.

Nesse código, uma implementação anônima de **Comparator** é utilizada pelo método **sort()** da classe **Collections** para definir o comportamento – ou estratégia – de ordenação que o método deve assumir. O resultado é o *print* da lista ordenada: “[*tiny*, *maior*, *grande*, *muito grande*]”, em ordem crescente de tamanho.

Com a introdução desse design baseado no padrão *Strategy*, o desenvolvedor Java consegue se aproximar mais da programação funcional e ser mais declarativo ao deixar responsabilidades de fluxos comuns em implementações que permitem

que o desenvolvedor se concentre mais na implementação do negócio, em vez de se preocupar com detalhes como controle de iteração.

## Listagem 1. Ordenação com Comparator.

```
1. public void sortTinyStrings() {
2.     List<String> list = Arrays.asList("muito grande", "maior", "grande", "tiny");
3.     Collections.sort(list, new Comparator<String>() {
4.         @Override public int compare(String value, String other) {
5.             return value.length() - other.length();
6.         }
7.     });
8.     System.out.println(list);
9. }
```

Fundamentalmente, o que o método `sort()` precisa é de uma função, com um método de comparação entre itens da coleção; porém, até o Java 7 não tínhamos uma forma de lidar diretamente com a função necessária para a tarefa. Por isso, o código da **Listagem 1** ainda exige muita codificação, pois estamos criando uma nova classe com todo o peso da declaração do tipo e do método a ser executado em uma função simples de ordenação. Para ser mais conciso, o código precisaria pular esta etapa de declaração de tipo e método e partir para o que o desenvolvedor está realmente interessado, ou seja, na definição de como ordenar o conteúdo da lista. Com isso o código ficaria mais limpo e ainda assim significante, autoexplicativo. No Java 8, Lambdas nos dão essa possibilidade. Observe na **Listagem 2** como podemos realizar a mesma tarefa de ordenação utilizando uma expressão Lambda para passar o método com a função de comparação.

## Listagem 2. Ordenação com Lambda.

```
1. public void sortTinyStringsWithLambda() {
2.     List<String> list = Arrays.asList("muito grande", "maior", "grande", "tiny");
3.     Collections.sort(list, (value, other) -> value.length() - other.length());
4.     System.out.println(list);
5. }
```

Semanticamente os dois exemplos se equivalem; porém, nesse último código, a partir da utilização de um Lambda no segundo parâmetro, o código fica muito mais conciso (nítido e significativo). Nesta notação estamos informando, entre parênteses, que a função recebe duas variáveis (`value` e `other`) e após o símbolo `->` indicamos que o resultado do comando seguinte deve ser retornado como resultado do método `compare()`.

O compilador utiliza as informações da declaração do método na interface para deduzir o tipo das variáveis, mas é possível definir o tipo diretamente na declaração do Lambda. No exemplo da **Listagem 2**, o argumento poderia ter sido alterado para `(final String value, final String other) -> value.length() - other.length()`. A vantagem de inferir o tipo dos argumentos é a possibilidade de utilizar o modificador `final` na declaração, a fim de evitar possíveis modificações da variável dentro do método – recomendável, mas não obrigatório.

Quando o desenvolvedor omite a inferência do tipo, o compilador assume que os parâmetros podem ser modificados e a responsabilidade de manter a imutabilidade fica a cargo do desenvolvedor. No entanto, se o parâmetro não for implicitamente `final`, o compilador apontará a mutação como erro, pois precisa garantir a consistência da função de segundo nível.

## Sintaxe de criação de Lambdas

Um Lambda é uma facilidade sintática que permite criar implementações de tipos com apenas um método, sem a necessidade de explicitar a criação através de uma classe. A sua estrutura básica segue o esquema **(parâmetros)->comando**.

A declaração de um Lambda que possua mais de um comando também é possível. Para isso, basta colocá-los entre chaves (`{}`) com os comandos separados por ponto-e-vírgula (`;`), como no esquema **(parâmetros)->{comando1; return comando2;}**. Note que é obrigatório o ponto-e-vírgula final antes do fechamento das chaves e que o comando de `return` seja explícito, como em qualquer outro método com retorno diferente de `void`.

Outra forma de declarar Lambdas é através da atribuição direta a variáveis. Nesse caso, o Lambda assume o tipo da variável sendo atribuída, com a condição de que o tipo seja uma interface com apenas um método e que os parâmetros informados na criação do Lambda e o seu tipo de retorno obedecam os tipos dos parâmetros e retorno da interface.

Finalmente, na declaração dos Lambdas com apenas um parâmetro é possível omitir os parênteses, conforme o esquema **parametro -> comando**. Por sua vez, em casos em que o Lambda não possua parâmetro, a declaração poderá ser feita conforme o esquema **() -> comando**.

## Código ainda mais conciso com Referências de método

Até agora vimos como a programação declarativa através da utilização de Lambdas promove um código mais conciso e funcional. Junto com a facilidade de criação dos Lambdas, o Java 8 também introduziu a parametrização de funções através das Referências de Métodos, que em alguns casos permitem a declaração direta do método a ser invocado no objeto parametrizado implicitamente. Para exemplificar, vamos voltar ao algoritmo de ordenação de `Strings` por tamanho e adicionar mais uma regra que consiste em modificar todos os itens da lista para maiúsculo.

Os códigos das **Listagens 3** e **4** são muito semelhantes entre si, com a diferença na linha 5 que, na **Listagem 4** utiliza a referência do método `toUpperCase()` da classe `String` para transformar em maiúsculo o item iterado. Os dois exemplos são bem particulares se comparados com os exemplos anteriores, por utilizarem recursos novos da API do Java 8 que facilitam o desenvolvimento funcional. O primeiro recurso utilizado nestes exemplos é o `java.util.stream.Stream`, cuja referência é obtida através da chamada ao método `stream()` da lista a ser ordenada. O pacote `java.util.stream` possui várias classes de suporte à programação funcional, sendo a classe `Stream` responsável por fornecer suporte a operações sequenciais e paralelas de agregação, mapeamento, filtro, junção, entre outros.

### Listagem 3. Ordenação e mapeamento com Lambda.

```
1. public void sortAndMapToUpper() {  
2.     List<String> list = Arrays.asList("muito grande", "maior", "grande", "tiny");  
3.     list = list.stream()  
4.         .sorted((value, other) -> value.length() - other.length())  
5.         .map(value -> value.toUpperCase())  
6.         .collect(Collectors.toList());  
7.     System.out.println(list);  
8. }
```

### Listagem 4. Ordenação e mapeamento com referência de método.

```
1. public void sortAndMapToUpperWithMethodRef() {  
2.     List<String> list = Arrays.asList("muito grande", "maior", "grande", "tiny");  
3.     list = list.stream()  
4.         .sorted((value, other) -> value.length() - other.length())  
5.         .map(String::toUpperCase)  
6.         .collect(Collectors.toList());  
7.     System.out.println(list);  
8. }
```

Em ambos os exemplos, a primeira operação executada é a ordenação dos itens da lista. Para tanto, é utilizada a mesma expressão Lambda que já exploramos em exemplos anteriores, mas desta vez utilizando o método `sorted()` que faz parte do objeto de tipo `Stream`. Já na segunda operação, no mapeamento, temos o código da **Listagem 4** utilizando uma referência de método para indicar de que forma os itens da lista devem ser mapeados. Esta é a grande diferença entre os dois exemplos de execução da mesma tarefa. O Lambda utilizado na **Listagem 3** realiza a mesma função da referência de método, sendo a referência utilizada na **Listagem 4** uma opção que melhora a legibilidade do código. Finalmente temos a chamada do método `collect()`, que faz a coleta dos itens já ordenados e convertidos em maiúsculo. Estes e outros métodos mais importantes para a programação funcional serão explorados mais adiante.

## Sintaxe das referências de métodos

A sintaxe das referências de métodos é bem simples, seguindo sempre o esquema `Tipo::nomeDoMétodo`. Em algumas situações, podemos utilizar referências de construtores, que funcionam da mesma forma que as referências de métodos, seguindo o esquema `Tipo::new`, sendo o sufixo `new` uma constante para indicar que estamos referenciando o construtor de uma nova instância do `Tipo` que será utilizada na tarefa a ser realizada.

Assim como os Lambdas, as referências de métodos podem ser utilizadas como argumentos de qualquer parâmetro em que uma interface com um único método seja esperado, desde que o método referenciado declare os mesmos parâmetros que o método da interface. O compilador reconhecerá a referência mesmo que ela não esteja declarada no tipo do item iterado e que o método não implemente a interface esperada.

Adicionalmente, é possível utilizar a referência de um método estático de outras classes – como classes utilitárias, por exemplo. No caso da **Listagem 4**, a linha 5 poderia facilmente ser modificada para algo como `.map(StringUtil::toUpperCase)`. Neste caso, a clas-

se `StringUtil` deveria ser implementada com um método estático `toUpperCase()` que recebesse como parâmetro uma `String` (já que os itens sendo mapeados na lista são do tipo `String`).

## Operações funcionais

A programação funcional sendo uma evolução – não revolução, como se pode imaginar – do paradigma de desenvolvimento orientado a objetos, nos permite reestruturar o raciocínio de construção de soluções, orientando o design do software para práticas de criação de fluxos comuns que, graças à natureza declarativa dos códigos funcionais, flexibilizam mudanças de comportamento sem alterar ou comprometer o controle de tais fluxos.

Seguindo este raciocínio, algumas estruturas comuns, principalmente em tarefas de iteração como mapeamento, filtro e coleta (redução), se consolidaram como práticas rotineiras na programação funcional e, portanto, foram implementados nativamente na API Java 8 de modo a flexibilizar e facilitar o desenvolvimento funcional. Sendo assim, faz-se necessário compreender e dominar tais conceitos e implementações para que seja possível tirar proveito destes poderosos recursos. Para entendermos estes conceitos, vamos explicar e em seguida exemplificar de forma funcional os principais recursos envolvidos nas tarefas mais comuns em um fluxo de software, sendo eles: ordenação, mapeamento, filtro e redução.

### Ordenação

A ordenação de uma coleção está sempre atrelada à iteração e comparação entre os itens utilizando o algoritmo mais apropriado dependendo do volume e da natureza dos dados. Um exemplo imperativo de ordenação de uma coleção simples pode ser verificado na **Listagem 5**, em que o algoritmo `bubble sort` é utilizado.



Na **Listagem 6**, a mesma tarefa é realizada de forma funcional. Perceba que no exemplo da **Listagem 5** o algoritmo se torna muito mais imperativo, com variáveis controlando o fluxo e tornando-o naturalmente sequencial. Diferentemente, na **Listagem 6** a mesma tarefa é escrita em um código mais orientado ao paradigma funcional, apropriando-se de filtros e mapeamentos para decidir o fluxo de recursão que mantém a iteração implícita.

## Listagem 5. Ordenação por Bubble Sort em modo imperativo.

```
01. public static final <T extends Comparable<T>> List<T>
    imperativeBubbleSort(List<T> list) {
02.     List<T> result = new ArrayList<>(list);
03.     int len = result.size();
04.     do {
05.         int newLen = -1;
06.         for(int i = 0; i < len-1; i++) {
07.             if(list.get(i+1).compareTo(list.get(i)) < 0) {
08.                 list.add(i, list.remove(i+1));
09.                 newLen = i;
10.             }
11.         }
12.         len = newLen + 1;
13.     } while (len > 0);
14.     return list;
15. }
```

## Listagem 6. Ordenação por Bubble Sort em modo declarativo.

```
01. @SuppressWarnings("unchecked") public static final <T extends
    Comparable<T>> List<T> declarativeBubbleSort(final List<T> list) {
02.     List<T> result = Collections.synchronizedList(new ArrayList<>(list));
03.     int len = result.size();
04.     Function<Function<Object, Object>, IntConsumer>
        consumer = recur -> length ->
05.         IntStream.range(0, length)
06.             .filter(i -> IntStream.range(0, len - i - 1)
07.                 .filter(j -> result.get(j+1)
08.                     .compareTo(result.get(j)) < 0)
09.                     .mapToObj(j -> {
10.                         T swap = result.remove(j+1);
11.                         result.add(j, swap);
12.                         return swap;
13.                     }).count() > 0)
14.             .max().ifPresent(IntConsumer.class
15.                 .cast(recur.apply(Function.class.cast(recur))));
16.             consumer.apply(Function.class.cast(consumer)).accept(len);
17.         return result;
18.     }
```

Em um caso em que a análise de execução do algoritmo demonstre que o paralelismo o melhore, é possível paralelizar o código de *swap* dos dados (troca de posições) da **Listagem 6** realizado no mapeamento da linha 9 à linha 12 ao incluir a chamada ao *pipe* de stream paralelo e ao aplicar uma sincronização à lista **result** no método **mapToObj()**, já que o seu acesso será paralelizado, sem o risco de qualquer tipo de efeito colateral. Já na **Listagem 5**, tal paralelismo parece muito mais difícil de ser alcançado. Além disso, a condição de finalização das iterações na **Listagem 5** está totalmente atrelada às mudanças nas variáveis **len** e **newLen**,

enquanto o código da **Listagem 6** torna tal condição implícita através do método **ifPresent()**, que verifica se o resultado do primeiro filtro possui algum item e só executa o código **IntConsumer.class.cast(recur.apply(Function.class.cast(recur)))** se algum item estiver presente.

Outro detalhe que precisa ser mencionado é a lógica de recursão estruturada no algoritmo, que é iniciada na chamada da linha 17 e consolidada no código das linhas 14 e 15. Tal esquema se deve ao fato de o Java não possuir nativamente o suporte à recursão de forma funcional, existente em muitas linguagens cujo suporte inclui uma palavra reservada – *recur* no caso da linguagem Clojure, por exemplo – que implicitamente realiza esta tarefa de recursão dentro de situações de iteração. A natureza da linguagem Java inviabiliza este tipo de execução recursiva, o que obriga o desenvolvedor a criar artifícios como este realizado nas linhas 14 e 15. Podemos perceber também que a avaliação tardia (*lazy evaluation* – ver **BOX 1**) dos **Streams** viabiliza a passagem da função recursiva como parâmetro dela mesma, já que a sua execução estará atrelada à condição da função de ordem mais baixa que será executada primeiramente.

## BOX 1. Lazy Evaluation (Avaliação tardia)

Lazy Evaluation ou Avaliação Tardia é uma técnica implementada por algumas linguagens de programação que possibilita a execução tardia, ou seja, execução posterior de uma função. Isto garante que a sua execução aconteça somente quando e, principalmente, se for necessária.

Através desta técnica, recursos importantes do Java 8, como os streams (**java.util.stream.Stream** e outros) se beneficiam de operações como filtros, mapeamentos, entre outros, ao percorrer somente a porção necessária de uma coleção e somente no momento final do encadeamento de chamadas. Com isso, torna-se possível realizar iterações em conjuntos teoricamente infinitos e realizar operações em porções menores que sejam importantes para uma determinada operação. Lazy Evaluation é uma característica importante de linguagens que pretendem ser funcionais, porque possibilitam trabalhar com a avaliação tardia de funções que, se não executadas de forma composta (daí a necessidade de ser tardio), pode conduzir a repetições infinitas e/ou resultados incorretos.

O exemplo a seguir, em Java, seria impossível de se realizar sem a avaliação tardia:

```
public List<Integer> getNthEven(Integer howMany) {
    return IntStream.iterate(0, i -> i+2)
        .limit(howMany)
        .collect(ArrayList::new,
            ArrayList::add,
            ArrayList::addAll);
}
```

Podemos ver que o trecho **IntStream.iterate(0, i -> i+2)** executaria infinitamente se fosse avaliado no mesmo momento da sua chamada. Porém, após limitarmos o tamanho da lista resultante ao valor da variável **howMany**, a redução pôde ser realizada sem risco de loop infinito.

## Mapeamento

O mapeamento é uma forma conveniente de efetuar operações de modificação e/ou transformação dos itens de uma coleção de forma funcional. Na implementação **Stream<T>**, o método **map()**

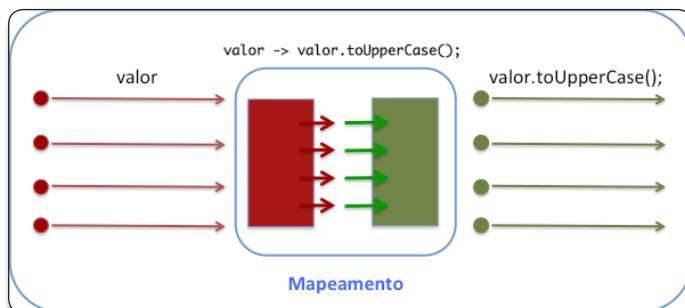
permite a realização do mapeamento através da parametrização de uma função `java.util.function.Function<? super T, ? extends R>`, que é uma interface funcional com o método `R apply(T)`, sendo `T` o tipo genérico iterado pela instância de `Stream<T>` e `R` o novo item da coleção no mapeamento.

#### Nota

O Java 8 introduziu o conceito de métodos defenders na criação de interfaces. Estes métodos podem ser criados com o comportamento padrão a ser adotado pelo método caso ele não seja declarado pela classe concreta que implemente a interface. Para todos os efeitos, os métodos declarados como defenders fazem parte do contrato a ser seguido pela classe que os implementa como qualquer outro método declarado na interface, porém a declaração do comportamento padrão pode garantir que uma interface com mais de um método ainda seja considerada funcional, caso apenas um dos métodos não seja defender.

O resultado final será um `Stream<R>`, sendo possível que genericamente `R` e `T` sejam o mesmo tipo (é o que ocorre quando o mapeamento realiza uma modificação ao invés de uma transformação). Vale lembrar que quando o argumento esperado por um método é uma interface funcional (seja explicitamente através da anotação `@FunctionalInterface`, ou implicitamente, por se tratar de uma interface com apenas um método sem *defender*), Lambdas e Referências de Métodos podem ser utilizadas, o que permite a escrita de um código mais conciso.

O esquema da Figura 1 elucida bem a ideia de transformação por trás do mapeamento.

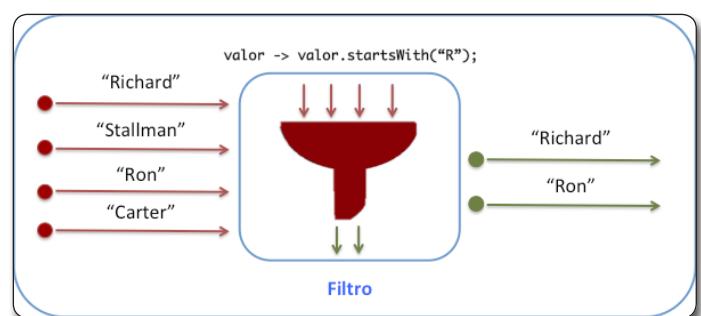


**Figura 1.** Esquema de funcionamento de um mapeamento em uma coleção de Strings

#### Filtro

Filtros são recursos essenciais em um processamento em que nem todos os itens de uma coleção sejam necessários. Devido à sua utilidade constante em muitos algoritmos, a classe `Stream` contempla o método `filter()`, que possibilita a definição de uma função de aceitação de um item da coleção iterada através da interface funcional `java.util.function.Predicate<? super T>`, esperada como parâmetro. Esta interface declara o método `boolean test(T)`, cuja implementação deve retornar um tipo `boolean` definindo se o item testado é elegível para permanecer na coleção resultante.

No esquema da Figura 2 é possível visualizar o funcionamento do filtro.



**Figura 2.** Esquema de funcionamento do filtro em uma coleção de Strings

Considerando o mecanismo do filtro baseado em um predicado que indica se o conteúdo deve permanecer na coleção resultante ou não, percebemos que, em qualquer filtro, a quantidade de itens resultantes da operação pode variar dependendo dos itens na coleção e do resultado dos chamados ao predicado, mas jamais ultrapassa a quantidade de itens da mesma.

#### Redução

Em programação funcional, a redução é utilizada para efetuar a sumarização das informações relevantes em uma coleção através da aplicação de uma operação de combinação. As operações de redução podem variar entre agrupamentos, junções, concatenações, coletas de máximos, mínimos, médias, somas, subtrações e outros. No Java 8, a redução pode ser realizada através dos métodos `reduce()` e `collect()` da classe `java.util.stream.Stream<T>`, cuja diferença principal está na forma de coleta, uma vez que a redução através do método `reduce()` ocorre de forma imutável e a redução através do método `collect()` acontece de forma mutável.

O mesmo exemplo de redução pode ser visto nas [Listagens 7 e 8](#). Ambos os exemplos executam a transformação de um texto simples, com espaços, para uma sentença no formato *camelcase*.



## Listagem 7. Implementação do algoritmo camelcase com reduce.

```
01. public static String toCamelCaseReduced(String text) {  
02.     return Arrays.asList(text.split(""))  
03.         .map(value -> value.length() < 2?  
04.             value.toUpperCase():  
05.             Character.toString(value.charAt(0))  
06.                 .toUpperCase()  
07.                 .concat(value.substring(1)  
08.                     .toLowerCase()))  
09.             .reduce("", String::concat, String::concat);  
10. }
```

## Listagem 8. Implementação do algoritmo camelcase com collect.

```
01. public static String toCamelCaseCollected(String text) {  
02.     return Arrays.stream(text.split(""))  
03.         .map(value -> value.length() < 2?  
04.             value.toUpperCase():  
05.             Character.toString(value.charAt(0))  
06.                 .toUpperCase()  
07.                 .concat(value.substring(1)  
08.                     .toLowerCase()))  
09.             .collect(StringBuilder::new,  
10.                 StringBuilder::append,  
11.                 StringBuilder::append).toString();  
12. }
```

### Nota

Camelcase é um modo de expressão textual muito conhecido entre desenvolvedores Java por ser uma boa prática como forma de nomear classes, métodos, variáveis e outros tipos de estruturas. O nome camelcase advém da alusão entre a percepção visual do texto transformado e as corcovas de um camelo.

O fluxo do algoritmo em exemplo inicia-se pelo `split()` da `String` a ser modificada. Após sua execução, o resultado será um `array` que é transformado em `Stream<String>` graças à introdução do método `stream()` à classe utilitária `java`



.util.Arrays no Java 8, que tem a função de criar uma instância de `Stream` com base no tipo do `array` passado como argumento ao método. Na próxima etapa do algoritmo, utilizamos o `Stream` resultante para mapear os itens da coleção, modificando-os para `Strings` iniciadas em maiúsculo. A última etapa do algoritmo é reduzir os itens da lista a uma `String` concatenada com todos os itens modificados pelo mapeamento anterior. Nesta etapa as Listagens 7 e 8 são diferentes entre si, sendo que o código da Listagem 7 executa o método `reduce()` em sua forma mais genérica, passando como argumento uma `String` com o valor inicial da concatenação e no segundo argumento passando a referência do método a ser utilizado para acumular – neste caso, concatenar – os itens em uma operação par-a-par. O terceiro argumento é uma referência de método a ser utilizada em caso de operação paralela para combinar os diferentes resultados das concatenações realizadas anteriormente pelo acumulador do segundo argumento.

Neste exemplo não existe mutação realizada em nenhuma variável. Isto é garantido pela forma de execução do método `reduce()`, já que os parâmetros enviados ao acumulador do segundo argumento serão sempre o resultado da última operação de acúmulo acompanhado do próximo item a ser concatenado. Em caso de paralelização da tarefa, o terceiro argumento, que é um combinador, funciona da mesma forma: os diferentes resultados de acumulações paralelas são combinados. Os dois parâmetros passados ao combinador são o resultado da última combinação junto de um novo valor – resultado de uma acumulação paralela anterior. Esse valor é combinado ao resultado final e o fluxo segue até que não haja mais valores a serem adicionados à combinação.

Portanto, neste exemplo, a tarefa de redução poderia facilmente ser paralelizada através da modificação da linha 9 para `.parallel().reduce("", String::concat, String::concat)`. A assinatura deste modo mais genérico do método `reduce()` é a seguinte: `<U> U reduce(U, java.util.function.BiFunction<U, ? super T, U>, java.util.function.BinaryOperator<U>)`. Se a paralelização não for uma preocupação, é possível simplificar o código utilizando a sobrecarga do método `reduce()` que requer apenas o valor inicial e o acumulador, conforme a assinatura: `T reduce(T, java.util.function.BinaryOperator<T>)`. Ao utilizar este método, podemos simplificar a linha 9 da Listagem 7 para `.reduce("", String::concat)` ou ainda tornar o código mais conciso ao utilizar a sobrecarga mais simplificada do método `reduce()` que recebe apenas o acumulador do tipo `BinaryOperator<T>` e retorna um `java.util.Optional<T>`, um tipo utilitário introduzido no Java 8 para ser utilizado em operações em que um valor nulo possa ser retornado, a fim de facilitar outras operações funcionais como executar um código opcional em caso de valores nulos para gerar um valor alternativo de retorno. Sendo assim, o melhor código a ser utilizado como substituição na linha 9 seria `.reduce(String::concat,orElse(text))`. O método `orElse()` da classe `Optional<T>` é responsável por fornecer o valor da variável `text` como retorno, caso o resultado da redução seja um valor nulo.

Já na **Listagem 8**, a tarefa de redução é executada através do método `collect()`, cuja assinatura é muito semelhante ao modo mais genérico do método `reduce(): <R> R collect(java.util.function.Supplier<R>, java.util.function.BiConsumer<R, ? super T>, java.util.function.BiConsumer<R, R>)`. A diferença no método `collect()` é que ele parametriza um tipo `Supplier<R>` que será responsável por fornecer uma instância nova de `R`, que será utilizada como container na coleta a cada execução. Esse container será responsável por armazenar os itens durante a redução e será o tipo retornado pelo método; portanto, deve ser uma instância mutável. Esta é a diferença a ser considerada ao escolher entre o método `reduce()` e o método `collect()` em uma operação de redução.

Em algumas situações será preferível optar pela perda da imutabilidade em uma redução com o método `collect()` a fim de ganhar em performance, como é o caso do exemplo na **Listagem 8**, que perde a imutabilidade garantida no exemplo da **Listagem 7**, mas, por outro lado, ganha pela performance inerente aos `StringBuilders`, que possibilitam trabalhar com `Strings` de forma mutável, reduzindo o problema de concatenação da **Listagem 7** que pode impactar em consumo excessivo de memória e perda de desempenho a cada concatenação.

O funcionamento do método `collect()` é muito semelhante ao do método `reduce()`, contando com o acumulador e o combinador para respectivamente agrupar e combinar os itens para formar o resultado final. Ainda assim, diferentemente do acumulador no método `reduce()`, o seu acumulador recebe a instância mutável de `R` como parâmetro, junto do item da coleção a ser incluído no container, ou seja, trata-se de uma operação item-para-container ao invés de par-a-par.

#### Nota

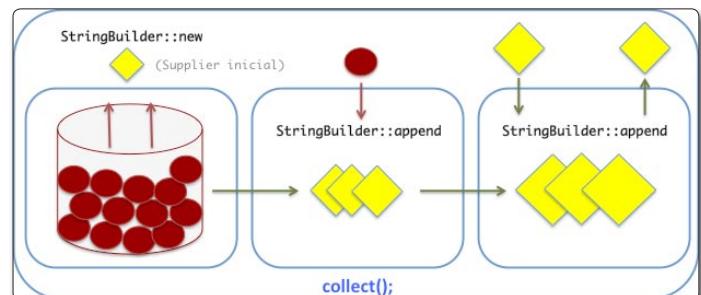
Estamos denominando item-para-container a redução em que cria-se um container (mutável) para armazenar os itens coletados sem geração de um novo item, ao passo que chamamos de par-a-par as operações em que dois itens da coleção (normalmente imutáveis) são combinados e resultam em um único item que virá a ser combinado novamente par-a-par em outras etapas até o final da redução.

Mesmo tendo o funcionamento atrelado a mutações nos containers, o método `collect()` pode ser paralelizado facilmente, devido à sua execução atrelada aos acumuladores e combinadores.

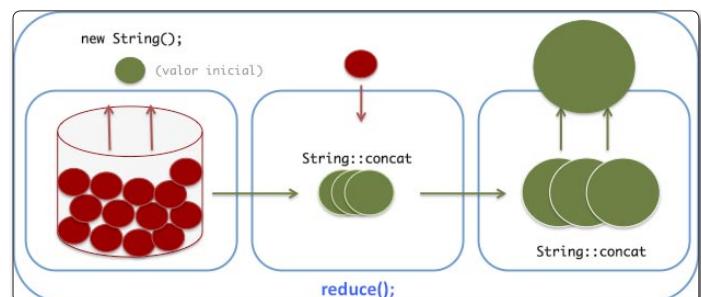
#### Nota

O conceito de redução não deve ser confundido com o método `reduce()`, que quando traduzido para o português tem o mesmo nome. O significado de redução é mais amplo e serve para explicar tanto o método `reduce()` quanto o `collect()`, sendo estes dois diferentes unicamente pela natureza imutável das variáveis na execução do `reduce()`, ao passo que o `collect()` tem sua execução atrelada a mutações nos containers.

Finalmente, na **Figura 3** podemos comparar a redução por `collect()` com a redução por `reduce()`, ilustrada na **Figura 4**.



**Figura 3.** Esquema de funcionamento da redução através do método `collect()`



**Figura 4.** Esquema de funcionamento da redução através do método `reduce()`

Com as figuras percebemos mais facilmente que, no caso do `reduce()` estamos tratando sempre do mesmo tipo de item que é associado de par em par a outro e os seus resultados parciais voltam a ser associados entre si para formar o resultado final. Enquanto isso, o `collect()` utiliza-se de um tipo diferente como container para armazenar resultados parciais dos itens que são associados a ele em uma operação de item a container e posteriormente gerar associações entre os containers (em caso de processamento paralelo) para um resultado final.



## A anotação @FunctionalInterface

`@java.lang.FunctionalInterface` é uma anotação introduzida no Java 8 utilizada para demarcar Interfaces Funcionais. Através desta anotação, o desenvolvedor obtém os benefícios de receber erros (*hints*) do compilador em situações em que a interface sendo criada não obedeça os requisitos para que seja considerada Funcional.

Esta anotação não é obrigatória para que uma interface seja considerada Funcional, ela serve para garantir que a interface obedeça aos requisitos para que seja uma; assim como a anotação `@java.lang.Override` também não é obrigatória para sobreescrita de métodos, apesar de ser útil para garantir que erros de digitação ou outros fatores impeçam que o método seja declarado erroneamente.

Uma interface é considerada Funcional quando declara apenas um método diferente daqueles já existentes na classe `java.lang.Object`; isto significa que uma interface que declare os métodos já existentes nesta classe, ainda poderá ser elegível como Funcional se declarar mais um método que não faça parte desta classe.

Interfaces que estendem outras também podem ser elegíveis como Funcionais, desde que não declarem métodos adicionais – a não ser que tais métodos estejam dentro da mesma regra de métodos existentes na classe `Object`.

Interfaces que adicionem métodos `default` – defenders – ainda serão consideradas Funcionais.

## Closures

Closures estão diretamente associados ao acesso por parte de uma Lambda ou uma classe anônima a variáveis que estejam no escopo do método em que uma ou outra esteja sendo declarada. Tal escopo é chamado de *enclosing scope* e não é uma novidade do Java 8, pois faz parte da linguagem desde a versão 5.



A criação de Closures garante uma flexibilidade importante ao desenvolvimento, mas quebra a ideia central da programação funcional. No exemplo da **Listagem 9**, em que um filtro é realizado em uma lista de nomes, podemos perceber que a lista resultante do filtro não depende somente de uma condição independente nos itens da coleção iterada, mas também da variável `letter`, que está em um *enclosing scope* e, portanto, cria uma dependência no filtro que altera o seu resultado final conforme as mudanças no valor da variável.

### Listagem 9. Exemplo de Closure.

```
1. public void pickNames(List<String> list, char letter) {  
2.     List<String> names = list.stream()  
3.         .filter(name -> name.startsWith(Character.toString(letter)))  
4.         .collect(Collectors.toList());  
5.     System.out.println(names);  
6. }
```

Assim, uma chamada ao método `pickNames()` conforme o exemplo: `pickNames(Arrays.asList("Richard", "Stallman", "Ron", "Carter"), 'R')`, imprimirá "[Richard, Ron]", enquanto a chamada com a mesma lista, mas com caractere diferente, conforme o exemplo: `pickNames(Arrays.asList("Richard", "Stallman", "Ron", "Carter"), 'C')`, imprimirá "[Carter]". Não há erro no resultado, mas conceitualmente temos a variável `letter` mudando o comportamento da execução do filtro. Esta mesma tarefa poderia ser executada de maneira mais declarativa se incluísse uma `Function<Character, Predicate<String>>` cuja aplicação resultasse um predicado a ser utilizado no filtro, conforme o código na **Listagem 10**.

### Listagem 10. Exemplo de como evitar Closures.

```
1. public void pickNamesFunctionalWay(List<String> list, char letter) {  
2.     Function<Character, Predicate<String>> function =  
3.         v -> name -> name.startsWith(Character.toString(v));  
4.     Predicate<String> predicate = function.apply(letter);  
5.     List<String> names = list.stream()  
6.         .filter(predicate)  
7.         .collect(Collectors.toList());  
8.     System.out.println(names);  
9. }
```

Neste código não temos mais acesso à variável `letter` no *enclosing scope*, o que é um ponto positivo do ponto de vista funcional. Apesar de não ter sido possível evitar totalmente o acesso ao *enclosing scope* devido à utilização da função de segundo nível `predicate` na função de primeiro nível `.filter()`, desta vez temos uma função se acoplando a outra para realizar a tarefa final de filtrar a coleção. Com esta mudança a JVM não precisará mais buscar o valor da variável `letter` em um escopo diferente, uma vez que o trecho `function.apply(letter)` está retornando outra função já com o valor da variável em seu escopo, garantindo a integridade da função no momento em que for executada pelo filtro da função de ordem mais alta.

Outra maneira funcional de eliminar o problema de *Closure* neste código seria parametrizar a função necessária ao filtro, conforme a **Listagem 11**, cuja chamada passaria a ser: `pickNames(Arrays.asList("Richard", "Stallman", "Ron", "Carter"), name -> name.startsWith(Character.toString('R')))`, sem alteração do valor impresso: “[Richard, Ron]”.

#### Listagem 11. Exemplo melhorado de como evitar Closures.

```
1. public void pickNames(List<String> list, Predicate<String> predicate) {
2.     List<String> names = list.stream()
3.         .filter(predicate)
4.         .collect(Collectors.toList());
5.     System.out.println(names);
6. }
```

A solução apresentada continua funcional, porém pode conduzir a duplicação de código caso o método `pickNames()` precise ser chamado em mais de um código, pois a Lambda passada como função necessária ao método poderia se repetir em diversos pontos da solução. Portanto, a **Listagem 10** continua sendo o melhor exemplo para evitar *enclosing scopes* de variáveis e priorizar o acoplamento de funções.

Paradoxalmente, *Closures* são necessários para a técnica de *currying*, que consiste em transformar a avaliação de uma função que exija dois ou mais parâmetros em uma sequência de avaliações de funções de um parâmetro só, com o benefício principal da avaliação tardia (*lazy evaluation*). Neste exemplo da **Listagem 10** estamos fazendo uso desta técnica ao deixar em uma função a dependência da vogal envolvida no filtro da função

de ordem mais alta. Com isso a avaliação tardia da função não corre risco de ter valores inconsistentes mesmo com a alteração da variável `letter`.

Se voltarmos ao exemplo da **Listagem 6**, podemos ver que a técnica de recursão para resolver o algoritmo *bubble sort* faz uso da técnica de *currying* para se beneficiar da avaliação tardia da primeira função, que é utilizada para compor a segunda função (gerando a recursão), que por sua vez recebe como parâmetro a variável `length`. Do ponto de vista matemático, se tentarmos escrever esta função, chegamos às expressões `consumer = g(Function)` (`Integer`) e `Function = h(Function, Integer)`. Deste modo, `consumer = g(h(Function, Integer))(Integer)`, ou, mais resumidamente:  $a = f(x)(y) = g(x, y) = x(y)$  e  $x = f(x)(y)$ ; o que prova que a avaliação tardia combinada com a condição do método `ifPresent()` possibilita que o código não fique em recursão infinitamente mesmo  $x$  sendo função de si próprio e de  $y$ .

A compreensão dos aspectos mais importantes da programação funcional com o Java 8 é de fundamental importância para o desenvolvedor que pretende ampliar o seu leque de conhecimentos para a criação de aplicações mais robustas e bem escritas. A partir deste ponto, valerá a pena um estudo mais aprofundado de classes importantes no desenvolvimento funcional utilizando Java 8; as principais são as presentes nos pacotes `java.util.function.*` e `java.util.stream.*`.

Temos visto que a programação funcional se impõe de forma declarativa através do acoplamento de funções para gerar estruturas de soluções flexíveis de fáceis manutenção e leitura através das Lambdas e das Referências de Métodos, trazendo consigo os benefícios de códigos mais concisos e a facilidade de paralelização.

## Conhecimento faz diferença!

The advertisement features several magazine covers from the 'Engenharia de Software' series:

- Edição 24 :: Ano 2:** Tema: Medição de Software: Um importante instrumento para o sucesso de projetos.
- Edição 25 :: Ano 2:** Tema: Agilidade: Negociação de contratos em projeto.
- Edição 26 :: Ano 2:** Tema: Processo: Definição + Ferramentas para Gerenciamento de Configuração.
- Edição 27 :: Ano 2:** Tema: Evolução do Software: Definições, preocupações e custo.
- Edição 28 :: Ano 2:** Tema: Automação de Testes: Cuidados a serem tomados na implantação.
- Edição 29 :: Ano 3:** Tema: SOA: Processo e levantamento de requisitos de negócios – Parte 2.
- Edição 30 :: Ano 3:** Tema: Qualidade de Software: Definição, características e importância.

**Características da revista:**

- Projeto: Diagrama de sequência na prática
- Projeto: Como inserir padrões de projeto através de refatorações – Parte 2
- Aulas: Estratégia de Teste Funcional baseada em Casos de Uso – Partes 5 a 9
- Atividades da turma: + de 290 vídeos para assinantes

**Call to Action:** 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!**



**DEV MEDIA**

Porém, é necessário ter cuidado para que a tentativa de escrever um programa orientado ao paradigma funcional não conduza a repetição de código como, por exemplo, a mesma lambda utilizada em vários lugares! Isso pode dificultar ainda mais a manutenção do software. Uma solução é associar a lambda a variáveis, criar funções de ordem alta ou criar métodos cujos retornos sejam lambdas e utilizá-los aonde for necessário, como foi exemplificado ao longo do artigo.

Vale lembrar que, mesmo com a introdução de Lambdas, Referências de Métodos e melhorias na API, o Java 8 não se tornou uma linguagem funcional; o seu suporte à programação funcional existe desde versões anteriores e vem melhorando a cada nova versão. Mesmo não sendo uma linguagem puramente funcional, com os avanços apresentados na versão 8, a linguagem Java se tornou muito mais flexível e suscetível a soluções mais funcionais através de códigos mais declarativos e concisos.

Caberá ao desenvolvedor mudar a forma de pensar a estrutura de suas soluções e tirar o máximo de proveito do paradigma funcional para escrever códigos mais limpos, de fácil manutenção e com maior suporte ao paralelismo através destas novidades e melhorias do Java 8.



## Autor



### Rômero Ricardo de Sousa Pereira

javeiro@uninove.edu.br | rpereira@atex.com

É formado em Ciência da Computação pela Universidade Nove de Julho, desde 2009, aonde atuou como professor em cursos extensivos de Java durante dois anos. Possui oito anos de experiência como desenvolvedor de sistemas, sendo quase sete atuando como desenvolvedor Java, passando por experiências desde o desenvolvimento de aplicações Standalone com Java SE, até soluções Web e Enterprise de alta demanda com Java EE. Atualmente é consultor Java na Atex Digital Media do Brasil. Possui as certificações OCJP e OCWCD.



## Links:

### Reduction Operation.

[http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html  
#Reduction](http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Reduction)

### Functional Interface Annotation.

<http://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html>

### Introduction to Functional Interfaces – A concept recreated in Java 8.

<http://java.dzone.com/articles/introduction-functional-1>

### Interface default methods in Java 8.

<http://java.dzone.com/articles/interface-default-methods-java>

### Lazy sequences implementation for Java 8.

<http://java.dzone.com/articles/lazy-sequences-implementation>

### Conception, Evolution, and Application of Functional Programming Languages.

<http://www.dbnet.ece.ntua.gr/~george/languages/books/p359-hudak.pdf>

### Currying vs Closures.

<http://java.dzone.com/articles/whats-wrong-java-8-currying-vs>

## Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



# RENOVE JÁ!

Sua assinatura pode estar acabando



Renovando a assinatura  
de sua revista favorita  
você ganha brindes  
e descontos exclusivos.

Faça a renovação de sua assinatura agora mesmo.



[www.devmedia.com.br/renovacao](http://www.devmedia.com.br/renovacao) ou (21)3382-5038

 **DEV MEDIA**

# Spring Framework: Criando uma aplicação web – Parte 2

ESTE ARTIGO FAZ PARTE DE UM CURSO

**N**a primeira parte deste curso apresentamos algumas funcionalidades introduzidas no Spring 4 por meio da implementação de uma aplicação Web para o controle de tarefas, simplesmente nomeada como **Controle de Tarefas**. A abordagem feita nesta parte do artigo serviu para mostrar o uso dos novos recursos do Spring aplicados ao desenvolvimento do *backend* deste tipo de aplicação.

Nesta segunda parte do curso vamos continuar demonstrando a utilização das novidades da versão 4 do Spring Framework, porém com maior atenção na implementação de funcionalidades que fazem parte da infraestrutura requerida para manutenção e apresentação das informações de tarefas da aplicação proposta.

A infraestrutura a ser apresentada inclui a elaboração de uma interface REST para troca de informações nas ações de gerenciamento de tarefas, como também a criação de um mecanismo de atualização destas informações em tempo real, com o uso da tecnologia WebSockets.

Apesar de não estar relacionado com a camada de apresentação, também será exposto o suporte oferecido pelo framework para alguns dos recursos existentes na plataforma Java 8, incluindo a utilização da API *Java Date and Time*, o tipo *Optional* e expressões *lambda*.

## Criando controladores REST

O Spring Framework oferece desde sua primeira versão uma solução MVC para o desenvolvimento da camada de apresentação de aplicações Web, o Spring MVC. Este framework vem evoluindo juntamente com a plataforma Spring, porém o princípio de funcionamento interno se mantém, baseando-se em alguns componentes.

## Fique por dentro

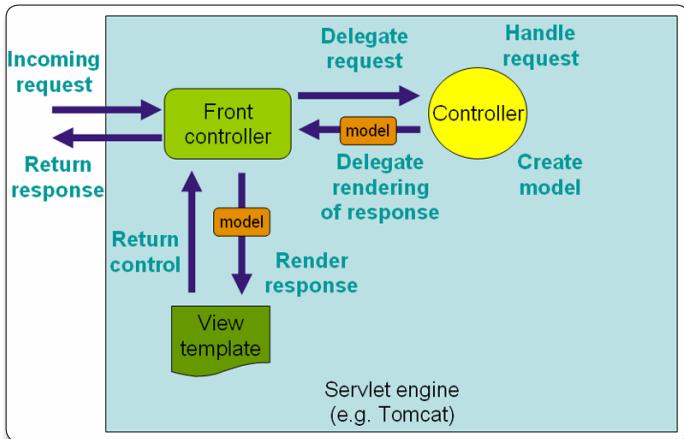
Nesta segunda e última parte da série conheceremos na prática mais recursos lançados com a versão 4 do Spring Framework e finalizaremos o desenvolvimento da aplicação exemplo. O foco estará no uso das novidades para implementação de funcionalidades da camada de apresentação da aplicação para controle de tarefas. Neste caso, o artigo será útil para expor ao leitor os recursos e tecnologias suportados e oferecidos pelo framework na criação de funcionalidades relacionadas à apresentação de dados e comunicação em tempo real em aplicações Web.

Os principais estão listados a seguir:

- **DispatcherServlet**: Componente baseado em uma especialização de Servlet (`javax.servlet.http.HttpServlet`) que atua como *Front Controller*, recebendo e direcionando as requisições para os controladores correspondentes;
- **HandlerMapping**: Componente responsável por manter o mapeamento e identificar qual *handler* (o método de um controlador) irá processar (tratar) a requisição;
- **Controllers**: O controlador, por meio de um de seus métodos, irá de fato processar uma requisição recebida, provavelmente delegando a execução para a camada de serviços, retornando a resposta para o chamador. A implementação de um controlador não está vinculada a interfaces e classes específicas do Spring MVC;
- **ViewResolver**: Este componente é responsável por identificar, através de um nome lógico retornado pelos métodos do controlador, qual elemento deverá ser utilizado na renderização da resposta, podendo ser um arquivo JSP.

O fluxo do processamento das requisições pelo funcionamento dos componentes listados anteriormente pode ser visualizado na **Figura 1**.

Na **Listagem 1** está o exemplo de um controlador que, por meio do método `home()`, fornece o tratamento para a renderização da página `home` através de requisições GET direcionadas à URI `/home`.



**Figura 1.** Fluxo de processamento de requisições no Spring MVC – Fonte: Documento de referência do Spring Framework

#### Listagem 1. Exemplo de implementação de um controlador Spring MVC.

```
//package e imports omitidos...

@Controller
public class HomeController {

    @RequestMapping(value = "/home", method = RequestMethod.GET)
    public String home() {
        return "home";
    }
}
```

Como pode ser verificado nesse código, a anotação `@org.springframework.stereotype.Controller` indica que a classe do exemplo é uma implementação de controlador. O mapeamento necessário para que as requisições HTTP sejam tratadas pelos métodos deste controlador são feitas com a anotação `@org.springframework.web.bind.annotation.RequestMapping`. Esta anotação pode ser aplicada tanto em nível de classe como em nível de método.

A configuração mais simples efetuada através da anotação `@RequestMapping` requer a indicação dos dois atributos presentes no código exemplo da **Listagem 1** e detalhados a seguir:

- **value:** Informa qual é o caminho da URI base quando a anotação é aplicada em nível de classe (tipo). Neste caso a URI será utilizada como base pelos demais métodos do controlador. Quando a anotação é aplicada em um método, indica qual URI específica este método irá processar;
- **method:** Informa qual método de requisição HTTP (GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE) será tratado pelo método do controlador.

A partir da versão 3 do Spring MVC a geração de respostas para renderização através dos métodos dos controladores se tornou mais versátil. Com a aplicação da anotação `@org.springframework.web.bind.annotation.ResponseBody` nos métodos

de um controlador, passou a ser possível escrever o resultado (retorno) destes métodos diretamente no corpo da resposta HTTP. Esse recurso pode ser utilizado, por exemplo, para a criação de interfaces REST que possibilitem a troca de informações no formato JSON ou XML. No entanto, até a versão 3 do framework a anotação `@ResponseBody` tinha que ser aplicada em cada método do controlador no qual este comportamento fosse requerido.

Na versão 4 do Spring MVC (módulo integrante do Spring Framework 4), as seguintes melhorias foram viabilizadas para a criação de controladores que expõem interfaces REST:

- A anotação `@ResponseBody` foi alterada para permitir sua aplicação em nível de classe. Neste caso, ao aplicar esta anotação em uma classe de controlador, todos os métodos terão o retorno escrito diretamente no corpo da resposta HTTP;
- A disponibilização da anotação `@org.springframework.web.bind.annotation.RestController`. Esta anotação define um novo estereótipo, pois é implementada na forma de uma anotação composta, agregando as anotações `@Controller` e `@ResponseBody`. Isto quer dizer que ao aplicar a anotação `@RestController` em uma classe, estaremos disponibilizando um controlador no qual todos os métodos já terão o comportamento de escrita no corpo da resposta HTTP, o ideal para o desenvolvimento de interfaces REST.

Na **Listagem 2** pode-se verificar a implementação do controlador `br.com.jm.tarefas.interfaces.web.GerenciamentoTarefasController`. Para evidenciar a criação e configuração do controlador REST, foram omitidos no código de exemplo os *imports* e a implementação dos métodos, focando na aplicação das anotações na classe e nos métodos.

O controlador mostrado oferece as operações REST para o gerenciamento das tarefas do sistema de Controle de Tarefas, sendo aproveitado pela camada de apresentação (página `home.jsp` e código JavaScript) para recuperar, exibir e efetuar modificações nas tarefas cadastradas.

O controlador `GerenciamentoTarefasController` utiliza a anotação `@RestController` para definir a interface REST que irá tratar informações no formato JSON nas respostas, conforme indicado no atributo `produces` da anotação `@RequestMapping`, empregada em nível de classe.

As operações listadas a seguir foram configuradas com o uso da anotação `@RequestMapping` em cada método existente no controlador e compõem a interface REST para o gerenciamento de tarefas. Esta interface, juntamente com a comunicação AJAX e a manipulação de dados via JavaScript, fornecem a infraestrutura necessária para a apresentação e manutenção das informações por meio do mecanismo de abas disponível no sistema de Controle de Tarefas:

- `/protected/tarefas/buscar`: Retorna todas as tarefas cadastradas no sistema. Por este motivo, apenas a chamada para o usuário com perfil Administrador é permitida;
- `/protected/tarefas/buscar/responsavel/{usuario}`: Retorna a lista de tarefas em que o usuário responsável corresponde ao identificador de usuário informado na variável `{usuario}`;

- /protected/tarefas/buscar/status/{status}: Retorna as tarefas em que o status corresponda ao status informado na variável {status};
- /protected/tarefas/{codigoTarefa}/iniciar: Inicia a tarefa que corresponde ao valor informado na variável {codigoTarefa};
- /protected/tarefas/{codigoTarefa}/concluir: Conclui a tarefa com o código correspondente ao valor informado na variável {codigoTarefa};
- /protected/tarefas/{codigoTarefa}/descartar: Descarta a tarefa com o código correspondente ao valor informado na variável {codigoTarefa};
- /protected/tarefas/criar: Cria uma nova tarefa. Esta operação está disponível apenas para o usuário com perfil Administrador. As informações necessárias para a criação da nova tarefa são recebidas no corpo da requisição HTTP (*payload*) em formato JSON.

## Supporte ao Java 8

O lançamento da versão 8 trouxe grandes mudanças e melhorias para a plataforma Java, desde à própria linguagem, às ferramentas que compõem o *kit* de desenvolvimento (JDK) e também ao ambiente de execução (JRE). Por se tratar de um assunto extenso, vamos abordar apenas alguns destes recursos, focando basicamente nos que foram adotados no projeto do sistema de Controle de Tarefas através do suporte oferecido pelo Spring Framework.

### Expressões Lambda

As expressões *lambda* (*Lambda Expressions*) tiveram sua inclusão na plataforma Java através da especificação contida na JSR 335. Esta especificação é dividida em oito partes: *Functional Interfaces*, *Lambda Expressions*, *Method References*, *Poly Expressions*, *Typing and Evaluation*, *Overload Resolution*, *Type Inference*, *Default Methods* e *Java Virtual Machine*.

Para o entendimento do funcionamento das expressões lambda, será detalhado apenas o conteúdo necessário relativo a interfaces funcionais (*Functional Interfaces*) e das próprias expressões *lambda*.

Uma expressão *lambda*, ou *closure*, pode ser definida como um bloco de código independente que representa uma função anônima, podendo, por exemplo, ser passado como argumento de um método, armazenado e executado futuramente. A sintaxe de uma expressão *lambda* é composta de uma lista de parâmetros, o *token* “->” e um bloco de código, conforme a sintaxe:

```
{(parametro1, parametro2) -> {bloco código;}}
```

Uma interface funcional, por sua vez, é uma interface que define apenas um método abstrato, firmando desta maneira um contrato com uma função única (*SAM – Single Abstract Method*). Não é mandatório, porém é indicado que as interfaces funcionais estejam anotadas com a anotação `@java.lang.FunctionalInterface`.

**Listagem 2.** Implementação do controlador GerenciamentoTarefasController.

```
package br.com.jm.tarefas.interfaces.web;

//imports omitidos...

@RestController
@RequestMapping(value = "/protected/tarefas",
produces = {MediaType.APPLICATION_JSON_VALUE})
public class GerenciamentoTarefasController {

    @Autowired
    private GerenciamentoTarefasService service;

    @Secured("ROLE_ADMINISTRADOR")
    @RequestMapping(value = "/buscar", method = RequestMethod.GET)
    public List<TarefaDTO> getTarefas() {
        //implementação omitida...
    }

    @RequestMapping(value = "/buscar/responsavel/{usuario}",
method = RequestMethod.GET)
    public List<TarefaDTO> getTarefasUsuario(@PathVariable("usuario")
String usuario,
        @AuthenticationPrincipal UsuarioAutenticacao autenticacao) {
        //implementação omitida...
    }

    @RequestMapping(value = "/buscar/status/{status}",
method = RequestMethod.GET)
    public List<TarefaDTO> getTarefasStatus(@PathVariable("status")
String status) {
        //implementação omitida...
    }

    @RequestMapping(value = "/{codigoTarefa}/iniciar",
method = RequestMethod.PUT)
    public MensagemDTO iniciarTarefa(@AuthenticationPrincipal
UsuarioAutenticacao usuario,
        @PathVariable("codigoTarefa") Long codigoTarefa) {
        //implementação omitida...
    }

    @RequestMapping(value = "/{codigoTarefa}/concluir",
method = RequestMethod.PUT)
    public MensagemDTO concluirTarefa(@AuthenticationPrincipal
UsuarioAutenticacao usuario,
        @PathVariable("codigoTarefa") Long codigoTarefa) {
        //implementação omitida...
    }

    @RequestMapping(value = "/{codigoTarefa}/descartar",
method = RequestMethod.PUT)
    public MensagemDTO descartarTarefa (@AuthenticationPrincipal
UsuarioAutenticacao usuario,
        @PathVariable("codigoTarefa") Long codigoTarefa) {
        //implementação omitida...
    }

    @Secured("ROLE_ADMINISTRADOR")
    @RequestMapping(value = "/criar", method = RequestMethod.POST)
    public MensagemDTO criarTarefa(@AuthenticationPrincipal
UsuarioAutenticacao usuario,
        @RequestBody NovaTarefaDTO novaTarefa) {
        //implementação omitida...
    }
}
```

Um exemplo concreto de interface funcional é a interface `java.lang.Runnable` da API Java, pois define um único método: `run()`.

De uma forma simples, em relação ao funcionamento das expressões *lambda* em Java, podemos afirmar que estas se baseiam inteiramente nas interfaces funcionais. O compilador, ao encontrar uma expressão *lambda* como argumento em um método, tentará fazer a coerção desta expressão para uma implementação da interface funcional definida como tipo na assinatura deste método.

No contexto do Spring Framework existem várias interfaces que, por definição, são interfaces funcionais, sendo, portanto, compatíveis para o uso através de expressões *lambda*.

Alguns exemplos destas interfaces são:

- `org.springframework.jdbc.core.RowMapper<T>`: Esta interface define o método `T mapRow(ResultSet rs, int rowNum)`, cujo contrato especifica a construção (transformação) da linha corrente do `ResultSet` para o objeto de retorno do tipo `T` (parametrizável). A interface `RowMapper` é utilizada pela classe utilitária `org.springframework.jdbc.core.JdbcTemplate`, que auxilia no desenvolvimento de código Java que executa operações SQL via JDBC;
- `org.springframework.transaction.support.TransactionCallback<T>`: Interface utilizada pela classe `org.springframework.transaction.support.TransactionTemplate` para externalizar a implementação do código que deverá ser executado dentro de um contexto transacional. A interface `TransactionCallback` estabelece o método `T doInTransaction(TransactionStatus status)`;
- `org.springframework.jms.core.MessageCreator`: A classe utilitária `org.springframework.jms.core.JmsTemplate`, que facilita a interação com a API JMS, emprega esta interface na implementação do código para criação de mensagens JMS através do método `javax.jms.Message createMessage(Session)`.

Conforme exposto no código da **Listagem 3**, temos, no projeto apresentado, um exemplo simples da utilização de expressões *lambda* com Spring. A classe `InicializadorDB`, que já foi analisada na primeira parte deste artigo, executa a persistência de dados de exemplo para testes durante o desenvolvimento do sistema. O método `inicializar()` desta classe usa uma instância da classe `TransactionTemplate` para o gerenciamento do contexto transacional durante a persistência destes dados. Ainda neste método, note o uso de uma expressão *lambda* compatível com a interface funcional `TransactionCallback` para a carga dos dados na base.

#### Java Date and Time

A JSR 310 estabeleceu a API *Java Date and Time* para a plataforma Java 8, fornecendo novos conceitos para a representação e realização de operações envolvendo datas e horas. Entre os novos pacotes relacionados a esta API, podemos destacar:

- `java.time`: Pacote principal, contendo as classes para representação de data, hora, data-hora, *timezones*, instantes e intervalos, baseando-se no padrão ISO-8601. Diferentemente da API antecessora, todas as classes deste pacote são imutáveis e *thread-safe*;

- `java.time.temporal`: Fornece as classes para acesso aos atributos de data e hora dos tipos `java.time`, como também oferece suporte à realização de cálculos a partir destas informações;
- `java.time.format`: Disponibiliza as classes para formatação e *parse* de data/hora;
- `java.time.zone`: Fornece classes para manipulação de informações de *timezone* dos tipos data/hora;
- `java.time.chrono`: Define uma API genérica para sistemas de calendários diferentes do padrão ISO.

#### Listagem 3. Implementação da classe InicializadorBD.

```
package br.com.jm.tarefas.infrastructure;

//imports omitidos...

public class InicializadorBD {

    @Autowired
    private PlatformTransactionManager transactionManager;

    @Autowired
    private GeradorCodigoTarefa geradorCodigoTarefa;

    @Autowired
    private JpaRepository<Usuario, Long> usuarioRepository;

    @Autowired
    private JpaRepository<Tarefa, Long> tarefaRepository;

    private DadosExemplo dadosExemplo;

    @PostConstruct
    public void inicializar() {
        dadosExemplo = new DadosExemplo(geradorCodigoTarefa);
        TransactionTemplate transactionTemplate =
            new TransactionTemplate(transactionManager);
        transactionTemplate.execute(ts) -> {
            carregarDados();
            return null;
        });
    }

    private void carregarDados() {
        usuarioRepository.save(dadosExemplo.getFulano());
        usuarioRepository.save(dadosExemplo.getBeltrano());
        usuarioRepository.save(dadosExemplo.getSicrano());
        usuarioRepository.save(dadosExemplo.getJohnDoe());
        tarefaRepository.save(dadosExemplo.getTarefaCadastrada());
        tarefaRepository.save(dadosExemplo.getTarefaIniciada());
    }
}
```

O suporte fornecido pelo Spring 4 à API `java.time` é feito por classes disponibilizadas no pacote `org.springframework.format.datetime.standard`, com destaque para:

- `DateTimeFormatterRegistrar`: Configura o sistema de formatação do Spring com o suporte adequado aos tipos definidos pela API `java.time`;
- `Jsr310DateTimeFormatAnnotationFormatterFactory`: Executa a formatação dos atributos de classes que utilizam os tipos definidos pela API `java.time` e que estejam anotados com `@org.springframework.format.annotation.DateTimeFormat`.

Com estas duas classes que fazem parte do sistema de formatação de datas do Spring 4, torna-se possível a utilização das classes do pacote `java.time` como tipos em atributos de classes, desde que estes estejam anotados com `@DateTimeFormat`. A classe `NovaTarefaDTO`, exibida na [Listagem 4](#), utiliza esta anotação no atributo `dataDevida`, do tipo `java.time.LocalDate`. Um exemplo do funcionamento do sistema de formatação Spring é a utilização do tipo `NovaTarefaDTO` como argumento em um método de controlador. Através de seu sistema de tratamento de argumentos (`org.springframework.web.method.support.HandlerMethodArgumentResolver`), o Spring irá converter o parâmetro da requisição correspondente ao atributo `dataDevida` para um objeto do tipo `java.time.LocalDate`. Obviamente este fluxo será seguido se cada informação correspondente aos atributos do objeto do tipo `NovaTarefaDTO` for recebida na forma de parâmetros na requisição HTTP.

No caso de recebimento da informação de nova tarefa em formato JSON, o suporte aos tipos da nova API `java.time` é realizado indiretamente pela biblioteca `jackson-datatype-jsr310`. Esta biblioteca fornece as implementações de serializadores e deserializadores JSON compatíveis com esses novos tipos.

Novamente, no código da [Listagem 4](#), o atributo `dataDevida` está anotado com `@com.fasterxml.jackson.databind.annotation.JsonDeserialize`, fornecendo qual tipo de deserializador deve ser utilizado na conversão deste atributo. Neste caso foi informado o deserializador `com.fasterxml.jackson.datatype.jsr310.deser.LocalDateDeserializer`, compatível com o tipo `java.time.LocalDate` do atributo.

#### Listagem 4. Implementação da classe NovaTarefaDTO.

```
package br.com.jm.tarefas.application.dto;

//imports omitidos...

public class NovaTarefaDTO implements Serializable {

    private static final long serialVersionUID = 1L;

    private String titulo;

    private String descricao;

    @DateTimeFormat(iso = ISO.DATE)
    @JsonDeserialize(using = LocalDateDeserializer.class)
    private LocalDate dataDevida;

    private IdentificadorUsuarioDTO autor;

    //getters e setters omitidos...
}
```

#### Optional

O tipo `java.util.Optional<T>` foi adicionado à plataforma Java para uso em situações em que seja necessário explicitar que a referência para um tipo de objeto pode ser nula. Esse tipo permite a criação de objetos que encapsulam (*wrapper*) um objeto para-

metrizável do tipo T com o propósito de minimizar a ocorrência de erros causados por referências nulas (`NullPointerException`). Entretanto, somente com o uso correto, o tipo `Optional` poderá ajudar a evitar esse tipo de erro.

Por meio do método `Optional.isPresent()` é possível verificar a existência de valor (referência não nula) para o objeto encapsulado. Já o método `Optional.get()` recupera esse objeto. Então, antes de acessar um objeto encapsulado por uma instância do tipo `Optional` através do método `get()`, deve-se verificar a existência do mesmo por meio do método `isPresent()` evitando, deste modo, a ocorrência do erro `NoSuchElementException` caso a referência seja nula.

O Spring Data JPA, adotado no sistema de Controle de Tarefas, suporta o tipo `Optional` como tipo de retorno em métodos de repositórios. Neste cenário, esse tipo irá encapsular uma referência a um objeto ou uma referência nula relacionada ao resultado da execução de uma consulta. No desenvolvimento dos repositórios `TarefaRepository` e `UsuarioRepository`, ambos apresentados na primeira parte deste artigo, foi empregado o tipo `Optional` no retorno de alguns métodos, como pode ser visto no código de exemplo constante da [Listagem 5](#), que contém a interface de repositório para a entidade `Usuario`.

#### Listagem 5. Repositório JPA para a entidade Usuario.

```
package br.com.jm.tarefas.domain;

// imports omitidos...

public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

    public Optional<Usuario> findByIdentificador
        (IdentificadorUsuario identificadorUsuario);

}
```

#### Atualizações em tempo real com WebSockets

Parte integrante da tecnologia HTML5, o protocolo e a API WebSockets definem uma estratégia para comunicação bidirecional entre cliente (navegador) e servidor através de um canal de comunicação *full-duplex* que utiliza uma conexão TCP. O protocolo WebSocket foi criado de modo a permitir seu funcionamento na plataforma Web, o que fica evidente no processo de obtenção de conexões. A conexão inicia-se com o protocolo HTTP e é promovida (*upgrade*) para o protocolo WebSocket após o *handshake* (acordo) entre as partes.

A comunicação bidirecional traz grande vantagem no desenvolvimento de aplicações com necessidade de atualização (exibição) de informações em tempo real em comparação às soluções elaboradas com base em *pooling*, *long-pooling* e *streaming*. Estas soluções geram tráfego desnecessário de dados, como também interferem na latência (tempo de resposta).

A API WebSockets, suportada pela maioria dos navegadores atuais, permite o gerenciamento da conexão e a troca de mensagens com o servidor, porém em versões mais antigas, sem o devido

suporte a WebSockets, podem ser adotadas soluções alternativas como, por exemplo, o uso da biblioteca JavaScript SockJS. Esta biblioteca fornece a emulação da comunicação bidirecional realizada por WebSockets em navegadores que não possuem o suporte a esta tecnologia. Inteligentemente, a comunicação iniciada por meio de SockJS irá tentar utilizar o suporte nativo a WebSockets, e caso este não seja suportado, será aplicada a melhor opção de emulação oferecida pelo ambiente de execução, ou seja, a opção *fallback* (solução alternativa ou de contingência).

## STOMP

Enquanto o protocolo e a API WebSockets estabelecem o canal e o modo de comunicação entre cliente e servidor, o que implicitamente determina um contexto de troca de mensagens, ainda é necessário especificar qual será o formato utilizado na troca de mensagens entre as partes, o que pode ser feito através do uso de um subprotocolo de mensagens em nível de aplicação.

Durante a etapa de *handshake*, no estabelecimento de uma conexão WebSocket, é possível escolher um subprotocolo para troca de mensagens no qual tanto cliente quanto servidor tenham suporte. O protocolo STOMP (*Simple Text Oriented Message Protocol*), utilizado pelo Spring Framework 4 na comunicação por meio de WebSockets, é um exemplo concreto desta definição de subprotocolo de comunicação com mensagens em nível de aplicação.

O protocolo STOMP surgiu da necessidade de comunicação com servidores (*brokers*) de mensagens a partir de linguagens de *script*, como Ruby, Python e Perl. Este protocolo emprega um padrão simplificado de troca de mensagens, baseado no protocolo HTTP, permitindo a comunicação assíncrona entre clientes e servidores por meio de mensagens em formato texto.

O formato de comunicação definido pelo STOMP se baseia em *frames*, com uma estrutura composta de um comando (*command*), um conjunto opcional de cabeçalhos (*headers*) e um corpo (*body*) opcional. Esta estrutura está detalhada na **Listagem 6**.

**Listagem 6.** Estrutura de um frame STOMP.

```
COMMAND  
header1:value1  
header2:value2  
  
Body^@
```

No lado servidor, o modelo para comunicação é constituído com base em um conjunto de destinos (*destinations*), para os quais as mensagens podem ser enviadas. Geralmente esses destinos estão definidos em um sistema de mensagens com suporte ao protocolo STOMP, como é o caso dos *brokers* de mensagens ActiveMQ e RabbitMQ.

No lado cliente, a interação por mensagens pode ser realizada de duas maneiras:

1. Como um produtor, enviando as mensagens para um destino no servidor;

2. Como um consumidor, subscrevendo-se em um destino no servidor e recebendo as mensagens postadas neste.

Como já mencionado, o protocolo STOMP estabelece a comunicações por *frames*, estando estes divididos em dois conjuntos: os que são utilizados pelo cliente e os que são utilizados pelo servidor.

O conjunto representando os comandos para uso pelo cliente compreende os seguintes *frames* STOMP:

- **CONNECT:** Até a versão 1.1 este é o *frame* utilizado para estabelecer a conexão com o servidor;
- **STOMP:** Este *frame* é utilizado para conexão com o servidor a partir da versão 1.2, entretanto o *frame* CONNECT continua sendo suportado por compatibilidade;
- **SEND:** *Frame* utilizado para mandar mensagens ao servidor, exigindo, neste caso, a informação do cabeçalho *destination*, indicando o destino da mensagem;
- **SUBSCRIBE:** Este *frame* é utilizado para se subscrever em um determinado destino no servidor. Desta forma, o cabeçalho *destination* é obrigatório, além de um cabeçalho *id*, que identifica unicamente a subscrição;
- **UNSUBSCRIBE:** *Frame* utilizado para remover a subscrição de um determinado destino, exigindo, neste caso, a informação do cabeçalho *id*, contendo a identificação única da subscrição;
- **BEGIN:** *Frame* utilizado para iniciar uma transação, exigindo a informação do cabeçalho *transaction* contendo a identificação da transação;
- **COMMIT:** Este *frame* é utilizado para efetuar o *commit* da transação correspondente ao identificador recebido no cabeçalho *transaction*;
- **ABORT:** *Frame* utilizado para efetuar *rollback* da transação correspondente ao identificador recebido no cabeçalho *transaction*;
- **ACK:** Este *frame* é usado para fazer o *acknowledge* do consumo da mensagem e deve informar o cabeçalho *id*, identificando a qual mensagem a confirmação de consumo corresponde;
- **NACK:** *Frame* que possui lógica oposta ao ACK, informando que a mensagem não foi consumida e obrigatoriamente deve fornecer o cabeçalho *id* correspondente à mensagem;
- **DISCONNECT:** Este *frame* é utilizado para iniciar a desconexão com o servidor, informando o cabeçalho *receipt* e aguardando o *frame* de confirmação RECEIPT correspondente fornecido pelo servidor.

Os *frames* disponíveis para que o servidor comunique-se com o cliente são:

- **MESSAGE:** Este *frame* é utilizado para realizar o *broadcast* da mensagem para todos os clientes subscritos em um destino. Para isso, exige a informação dos cabeçalhos *destination* (destino da mensagem), *message-id* (identificador da mensagem) e *subscription* (identificador da subscrição que irá receber a mensagem);
- **RECEIPT:** *Frame* de confirmação enviado ao cliente informando que o *frame* enviado com o cabeçalho *receipt* foi recebido e processado;
- **ERROR:** Este *frame* enviado pelo servidor informa a ocorrência de erro. Pode incluir a mensagem informativa sobre o erro, como também a mensagem que originou o erro.

## Spring WebSocket e STOMP

O suporte a WebSockets e STOMP no Spring Framework 4 é feito por meio de dois novos módulos: **Spring WebSocket** (`spring-websocket`) e **Spring Messaging** (`spring-messaging`).

O módulo Spring WebSocket fornece o suporte de configuração e infraestrutura necessários para a manipulação dos recursos de WebSockets oferecidos pelo servidor de aplicações em uso, como também o mecanismo de *fallback* por meio do SockJS, no caso da indisponibilidade de WebSockets.

Já o módulo Spring Messaging foi criado a partir de abstrações do projeto Spring Integration para permitir a construção de fluxos de mensagens em uma aplicação. Alguns dos conceitos e implementações existentes no módulo Spring Messaging estão listados a seguir:

- **Message:** Representação genérica de uma mensagem, com cabeçalhos e corpo (*payload*);
- **MessageHandler:** Interface que define um contrato para a manipulação de mensagens;
- **MessageChannel:** Interface que estabelece um canal para o envio de mensagens;
- **SubscribableChannel:** Especialização de **MessageChannel** que agrupa o recurso de subscrição e implicitamente sugere o envio das mensagens para os subscritos no canal;
- **SimpleBrokerMessageHandler:** Esta é uma implementação simplificada de um *broker* de mensagens oferecido pelo Spring. Este mantém as mensagens em memória, permite a subscrição e realiza o *broadcast* destas mensagens. Esta implementação é destinada para uso em ambiente de desenvolvimento ou em aplicações mais simples;
- **StompBrokerRelayMessageHandler:** Esta implementação encaminha as mensagens para um *broker* de mensagens externo, por exemplo, o ActiveMQ;
- **SimpMessagingTemplate:** Este utilitário facilita a execução das operações para criação de uma mensagem (**Message**) e o envio para um destino (**MessageChannel**);
- Algumas anotações, como **@MessageMapping**, **@SendTo** e **@SendToUser**. Estas anotações são utilizadas em métodos de controladores para informar que a lógica para criação das mensagens (tratamento) será executada por estes métodos, antes do encaminhamento para o destino no *broker* de mensagens.

Para o uso de WebSockets através do suporte oferecido pelo Spring 4 no sistema de Controle de Tarefas, torna-se necessário realizar algumas configurações, conforme o código exposto na **Listagem 7** e detalhado na sequência:

- A classe de configuração **WebSocketConfig** estende a classe **AbstractSecurityWebSocketMessageBrokerConfigurer**, que já define algumas configurações padronizadas e habilita o mecanismo de autorização para a troca de mensagens via WebSockets;
- Com a aplicação da anotação **@EnableWebSocketMessageBroker** na classe **WebSocketConfig**, está sendo habilitado o mecanismo de *broker* de mensagens para um sistema de mensageria existente e também o uso do subprotocolo de mensagens STOMP;

- Através do método **configureMessageBroker()** foi habilitada a utilização do *broker* de mensagens oferecido pelo módulo Spring Messaging (**SimpleBrokerMessageHandler**), com uso do prefixo de destino */topic* e a definição do prefixo */app* para mensagens que devem ser tratadas pela aplicação (controladores);
- No método **configure()** está a definição de acesso (subscrição) para o destino */topic/updates* apenas aos usuários com perfil Administrador;
- No método **registerStompEndpoints()** foi criado o endpoint */ws*, no qual os clientes irão realizar as conexões via WebSockets. Neste endpoint também foi habilitada a opção de *fallback* por meio de SockJS;
- Através da sobrescrita do método **configureClientOutboundChannel()** é desabilitada a interceptação padrão para autorização da publicação de mensagens originadas do lado servidor, uma vez que as alterações geradas por todos os usuários logados no sistema serão propagadas.

## Listagem 7. Configuração de WebSockets e Message Broker.

```
package br.com.jm.tarefas.infrastructure.config;  
  
//imports omitidos...  
  
@Configuration  
@EnableWebSocketMessageBroker  
public class WebSocketConfig extends AbstractSecurityWebSocketMessage-  
BrokerConfigurer {  
  
    @Override  
    protected void configure(MessageSecurityMetadataSourceRegistry messages) {  
        messages.destinationMatchers("/topic/updates").hasAnyRole(  
            PerfilUsuario.ADMINISTRADOR.toString());  
    }  
  
    @Override  
    public void configureMessageBroker(MessageBrokerRegistry registry) {  
        registry.enableSimpleBroker("/topic");  
        registry.setApplicationDestinationPrefixes("/app");  
    }  
  
    @Override  
    public void registerStompEndpoints(StompEndpointRegistry registry) {  
        registry.addEndpoint("/ws").withSockJS();  
    }  
  
    @Override  
    public void configureClientOutboundChannel(ChannelRegistration registration) {  
    }  
}
```

O uso efetivo do fluxo de mensagens via WebSockets pelo sistema de Controle de Tarefas é bem simples: apenas os usuários com perfil Administrador irão acompanhar a evolução das tarefas cadastradas em tempo real. Com exceção da subscrição pelos administradores que inicia-se do lado cliente, o tráfego das atualizações de tarefas terá apenas o sentido servidor > cliente. O carregamento e atualizações de tarefas pelos usuários será feito por intermédio das operações existentes na interface REST, definida no código da **Listagem 2**.

O processo de subscrição de usuários com perfil Administrador irá funcionar da seguinte maneira:

- Ao efetuar o login no sistema o usuário será subscrito no destino `/topic/updates`;
- Assim que uma tarefa for cadastrada ou alterada, esta será recebida através do destino e incluída ou atualizada na aba *Tarefas*.

Na **Listagem 8** está o trecho de código JavaScript responsável pela subscrição, como também atualização e recebimento de novas tarefas.

**Listagem 8.** Código JavaScript para subscrição no destino `/topic/updates` e atualização de tarefas.

```
self.registrarListenerTarefas = function() {
    ws = new SockJS('/controle-tarefas/ws');
    self.stompClient = Stomp.over(ws);

    self.stompClient.connect({}, function(frame) {

        self.stompClient.subscribe("/topic/updates", function(message) {
            var tarefa = JSON.parse(message.body);
            self.atualizarTarefa(tarefa);
        });

        }, function(error) {
            console.log("Erro em conexão cliente STOMP " + error);
        });
};

self.atualizarTarefa = function(tarefa) {
    var existente = self.getTarefa(tarefa.identificador.codigo);
    if (existente) {
        existente.status(tarefa.status.nome);
        existente.responsavel(tarefa.responsavel.nome);
        existente.dataConclusao(tarefa.dataConclusao);
        self.mensagem.renderizarMensagemAviso
        ('Tarefa ['+ tarefa.identificador.codigo +'] atualizada!');
    } else {
        self.adicionarTarefa(tarefa);
        self.ordenarTarefas();
        self.mensagem.renderizarMensagemAviso
        ('Tarefa ['+ tarefa.identificador.codigo +'] adicionada!');
    }
};
```

Vejamos alguns detalhes sobre esse código:

- A conexão está sendo realizada por meio do utilitário SockJS, permitindo o funcionamento da aplicação em navegadores sem suporte a WebSockets;
- O uso da implementação JavaScript do cliente STOMP;
- Após a conexão com sucesso, a subscrição no destino `/topic/updates` é efetuada já fornecendo uma função de *callback* para o recebimento das mensagens enviadas pelo servidor. A mensagem recebida pode ser uma nova tarefa, que será a adicionado à lista de tarefas exibidas, ou uma atualização das informações de uma tarefa que já consta na lista de tarefas exibidas. A lógica para atualização ou inclusão de uma tarefa está implementada na função `atualizarTarefa()`.

Na **Listagem 9** está um exemplo de mensagem STOMP (*frame*) enviada do servidor ao cliente contendo as informações de uma tarefa em formato JSON.

**Listagem 9.** Mensagem STOMP com as informações em formato JSON.

```
MESSAGE
destination:/topic/updates
content-type:application/json;charset=UTF-8
subscription:sub-0
message-id:_0_bh7_x-0
content-length:527

{
    "identificador": {"codigo": 3},
    "titulo": "Nova Tarefa",
    "descricao": "Descrição Tarefa",
    "status": {"nome": "CADASTRADA"},
    "autor": {"identificador": {"email": "fulano@localhost"}, "nome": "Fulano de Tal", "perfil": "ADMINISTRADOR"},
    "responsavel": null,
    "dataCadastro": [2014, 12, 27, 11, 4, 4, 721000000],
    "dataDevida": [2014, 12, 31],
    "dataConclusao": null
}
```

Um detalhe a ser destacado é que do lado servidor não foi utilizado o tratamento de mensagens em métodos dos controladores, mas sim a publicação das ações de criação e atualização das tarefas por meio da classe `br.com.jm.tarefas.infrastructure.PublicadorAtualizacaoTarefa` (exibida na **Listagem 10**). Esta classe encapsula uma instância do tipo `SimpMessagingTemplate`, utilizada para propagar a atualização da tarefa para o destino `/topic/updates`.

**Listagem 10.** Implementação da classe `PublicadorAtualizacaoTarefa`.

```
package br.com.jm.tarefas.infrastructure;

//imports omitidos...

@Component
public class PublicadorAtualizacaoTarefa {

    private static final Log LOGGER = LogFactory.getLog
    (PublicadorAtualizacaoTarefa.class);

    @Autowired
    private SimpMessagingTemplate messagingTemplate;

    public void publicarAtualizacaoTarefa(TarefaDTO dto) {
        try {
            messagingTemplate.convertAndSend("/topic/updates", dto);
        } catch (MessagingException ex) {
            // Neste caso, apenas loga o erro de pulicação, garantindo a
            // efetivação de alteração na tarefa
            LOGGER.error(String.format("Erro ao publicar alteração na tarefa %s", dto), ex);
        }
    }
}
```

Para propagar todas as ocorrências de criação e atualização de tarefas em tempo real, foi utilizada uma instância do tipo **PublicadorAtualizacaoTarefa** na classe **GerenciamentoTarefasService**, que é o serviço oferecido para o gerenciamento de tarefas. A função do tipo **PublicadorAtualizacaoTarefa** é publicar as informações de tarefas no destino correto. O uso do publicador pode ser visto nos exemplos dos métodos **criarTarefa()** e **iniciarTarefa()**, ambos pertencentes ao serviço **GerenciamentoTarefasService** e detalhados na **Listagem 11**.

**Listagem 11.** Detalhe de uso do tipo PublicadorAtualizacaoTarefa no serviço GerenciamentoTarefaService.

```
@Secured("ROLE_ADMINISTRADOR")
public TarefaDTO criarTarefa(NovaTarefaDTO novaTarefa)
throws ControleTarefasException {
    Usuario usuario = getUsuario(novaTarefa.getAutor());
    Long codigoTarefa = tarefaRepository.proximoCodigoTarefa();

    Tarefa tarefa = usuario.criarTarefa(new IdentificadorTarefa(codigoTarefa),
        novaTarefa.getTitulo(), novaTarefa.getDescricao(), novaTarefa.getDataDevida());
    tarefaRepository.save(tarefa);

    TarefaDTO dto = MontadorTarefaDTO.montarTarefa(tarefa);
    publicadorAtualizacaoTarefa.publicarAtualizacaoTarefa(dto);

    return dto;
}

public TarefaDTO iniciarTarefa(AcaoTarefaDTO acaoTarefa)
throws ControleTarefasException {
    Usuario usuario = getUsuario(acaoTarefa.getIdentificadorResponsavel());
    Tarefa tarefa = getTarefa(acaoTarefa.getIdentificadorTarefa());
    tarefa.iniciar(usuario);

    TarefaDTO dto = MontadorTarefaDTO.montarTarefa(tarefa);
    publicadorAtualizacaoTarefa.publicarAtualizacaoTarefa(dto);

    return dto;
}
```

A publicação da criação e atualizações de tarefas a partir do serviço **GerenciamentoTarefasService** para o destino `/topic/updates` permite que os usuários com perfil Administrador logados no sistema recebam estas informações em tempo real através do mecanismo estabelecido no código da **Listagem 8**.

Conforme demonstrado, o uso de WebSockets e do protocolo de mensagens STOMP por meio do suporte no Spring Framework 4 permitiu construir uma solução simples e eficiente para exibição em tempo real das atualizações de informações das tarefas.

As tecnologias para o desenvolvimento de interfaces de usuário em aplicações Web evoluíram muito com o passar do tempo. Atualmente estão disponíveis *frameworks* que fornecem soluções para cada necessidade, desde a criação do *layout*, disponibilização

de componentes visuais, abstração de *patterns* para controlar a atualização e visualização do modelo de dados, desacoplamento de componentes, testes, entre outros. Esta variedade de recursos possibilita a construção de interfaces cada vez mais elegantes, sofisticadas e interativas.

Entretanto, para obter um bom resultado final para a camada de apresentação utilizando este conjunto de tecnologias citado, do mesmo modo como foi feito no sistema de Controle de Tarefas, é fundamental ter a infraestrutura necessária. O desenvolvimento da aplicação exemplo nos permitiu demonstrar e concluir que o Spring Framework se mantém atualizado, fornecendo as melhores soluções para a criação da infraestrutura desejada para a elaboração de interfaces de usuário modernas.

## Autor



Francisco A. Garcia

franciscogrc@gmail.com

É formado em Ciência da Computação pela Universidade Paulista. Trabalha como desenvolvedor Java desde 2006. Possui as certificações SCJP, SCWCD, OCBCD, OCEWSD.



## Links:

**Documentação de referência do Spring Framework versão 4.1.0 RELEASE (em formato PDF).**

<http://docs.spring.io/spring/docs/4.1.0.RELEASE/spring-framework-reference/pdf/spring-framework-reference.pdf>

**Página do JCP com a JSR 335, proposta da especificação de expressões lambda para a plataforma Java.**

<https://jcp.org/en/jsr/detail?id=335>

**Página da especificação do protocolo WebSocket.**

<https://tools.ietf.org/html/rfc6455>

**Página da especificação da API WebSocket.**

<http://dev.w3.org/html5/websockets/>

**Página do protocolo STOMP.**

<http://stomp.github.io/>

**Página do cliente JavaScript SockJS para emulação de WebSockets.**

<https://github.com/sockjs>

## Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



# CURSOS ONLINE



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

## 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>

# Spring MVC: Construa aplicações responsivas com Bootstrap – Parte 1

ESTE ARTIGO FAZ PARTE DE UM CURSO

O padrão de projeto MVC (Model-View-Controller) foi descrito pela primeira vez em 1979. A ideia surgiu com a necessidade de organizar uma aplicação, de forma que as responsabilidades dos componentes de um projeto fossem separadas umas das outras. Nesta arquitetura, a camada de controle processa as requisições e respostas e faz a ligação com a camada de modelo, que contém as regras de negócio da aplicação, e a camada de visão, responsável pela interação com o usuário.

Entre as vantagens do padrão proposto, tem-se a possibilidade de uma manutenção mais barata do código e torna-se muito mais prático estender as diversas funcionalidades do sistema, quando necessário. Não por acaso, o MVC começou a ser incorporado em tecnologias comuns e foi disponibilizado em diversos frameworks de desenvolvimento rapidamente.

Atualmente, existem dois tipos de implementações de frameworks MVC: os baseados em ações e os baseados em componentes. A diferença entre eles se dá principalmente pela forma como o fluxo de execução das solicitações é manipulado: os baseados em ações delegam a responsabilidade do gerenciamento de uma requisição a um controlador, cujo principal objetivo é reunir as informações de um modelo e devolver a resposta para o usuário através de uma visão; já os baseados em componentes utilizam a própria visão como responsável por extrair as informações de um modelo – por intermédio de uma classe gerenciável do framework – e assim apresentar a resposta ao cliente.

O Spring MVC é um destes frameworks baseados em ações e, quando utilizamos uma biblioteca *web* deste tipo, geralmente algum pacote de interface rica baseada em

## Fique por dentro

Conhecer produtos líderes em suas categorias é importante para qualquer desenvolvedor Java. Deste modo, neste artigo será apresentado o Spring MVC, framework Java web baseado em ações mais popular do mercado, e o Bootstrap, biblioteca para criação de páginas com layout responsivo com maior aceitação entre os desenvolvedores front-end. Nesta primeira parte vamos explorar mais profundamente o Spring MVC, analisando cada recurso que ele oferece através da criação do back-end de uma aplicação para uma clínica médica.

JavaScript é adicionado ao projeto para gerar telas dinâmicas e com um visual mais moderno. Na maioria dos casos, esta liberdade de escolha na construção da camada de visão torna possível migrar o *frontem* para outra plataforma de desenvolvimento quando necessário, o que representa uma grande vantagem para sistemas com tempo de vida longo. Não obstante, este não é o único diferencial: os frameworks MVC fornecem suporte ao REST, um serviço cada vez mais necessário nas aplicações atuais.

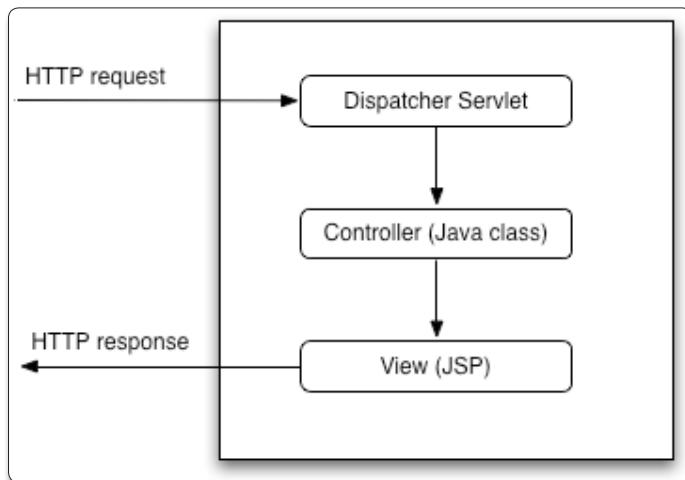
Na primeira parte deste artigo faremos uma análise do Spring MVC através da demonstração dos principais recursos presentes no framework. Para isso, iniciaremos a criação de uma pequena aplicação que simula o funcionamento de uma clínica médica utilizando o WildFly como servidor de aplicação e o PostgreSQL como banco de dados. No decorrer deste estudo todo o código do sistema será detalhado e explicado minuciosamente, para que possamos compreender o funcionamento da API da biblioteca *web* e como ocorre sua comunicação com as páginas JSP que serão montadas usando o Apache Tiles e o Bootstrap.

### O Spring MVC

O Spring MVC pertence ao Spring Framework e tem como principal proposta possibilitar uma maior produtividade no desenvolvimento de aplicações *web* corporativas utilizando a linguagem Java. Baseando-se no já comentado padrão de projeto MVC, ele consegue separar as principais fronteiras de uma aplicação *web*, fazendo com que nosso código fique bastante organizado.

Por se tratar de um framework leve e não intrusivo, ou seja, não requer que classes da aplicação tenham contato com as classes da biblioteca, e pelo fato de estar integrado ao Spring Framework, o Spring MVC tem à disposição serviços como injeção de dependências, gerenciamento de transações e de segurança, além da integração com frameworks de mapeamento objeto-relacional como o Hibernate.

O funcionamento básico do Spring MVC se dá através da classe **DispatcherServlet**, que é um *Front Controller* e estende **HttpServlet**. Ela tem como funções principais direcionar as requisições para outros controladores, tratá-las e repassar as respostas para o usuário. Após o recebimento da solicitação HTTP, o **DispatcherServlet** consulta o **HandlerMapping**, que redireciona a solicitação para outro controlador de acordo com suas configurações de comportamento. A **Figura 1** mostra como se dá o fluxo de uma requisição com o **DispatcherServlet**.



**Figura 1.** Funcionamento do Spring MVC

## 0 Bootstrap

O Bootstrap foi um framework criado para facilitar a elaboração de interfaces de usuário para sistemas *web*. A biblioteca surgiu com a necessidade de padronizar as estruturas de códigos de interfaces, de modo que se evitassem inconsistências geradas pelas diferentes formas de organizar os *layouts* dentro de um mesmo projeto. Com a padronização destas estruturas permitiu-se que toda a equipe de desenvolvimento envolvida trabalhasse com maior rapidez e eficiência, melhorando consequentemente a manutenibilidade do *front-end* da aplicação.

O framework foi projetado e desenvolvido pela equipe de engenheiros do Twitter e possui o código aberto. Estas características, aliadas aos consideráveis benefícios que o produto trouxe para os programadores *web*, fizeram com que ele se tornasse o projeto de desenvolvimento de *software* livre mais ativo do mundo, com uma comunidade gigantesca e bastante produtiva.

O Bootstrap disponibiliza uma coleção de elementos e funções personalizáveis empacotadas em uma única ferramenta que permitem a criação de *layouts* responsivos e o uso de *grids* personalizáveis.

As *grids* possibilitam mudar o *layout* das informações na página de forma muito prática, apenas modificando a largura das divs que funcionam como colunas. Os elementos escolhidos não conflitam entre si e são compatíveis com HTML5 e CSS3.

De forma padrão, o Bootstrap é formado por dois arquivos (um CSS e um JavaScript) que devem ser referenciados no código da aplicação. Entre os principais elementos HTML estilizados pela biblioteca, temos: *grids*, cabeçalhos, listas ordenadas e não ordenadas, formulários e botões. Em termos de componentes, vários foram criados pela equipe de desenvolvedores do projeto, dos quais podemos ressaltar: uma coleção de ícones baseados em fonte, botões *dropdown*, menus responsivos, *breadcrumbs*, paginadores, painéis e alguns componentes de *layout* para destacar uma área da página.

## Criando a aplicação

Como dito no início do artigo, criaremos um pequeno sistema de clínica médica para demonstrar o funcionamento do Spring MVC e alguns recursos do Bootstrap. As páginas do projeto serão baseadas em *templates* que fazem uso da biblioteca Tiles (ver **BOX 1**) e, em duas das telas teremos requisições Ajax para verificar como podemos efetuar este tipo de comunicação entre a *view* e o *controller*. O banco de dados utilizado será o PostgreSQL e o acesso se dará através da JPA 2.1, com o controle de transações efetuado pelo Spring ORM.

### BOX 1. Apache Tiles

O Tiles é uma biblioteca para construção de templates desenvolvida pela Apache que simplifica a construção da interface gráfica de aplicações web e viabiliza o reuso de código.

O ambiente requerido para implementar o sistema envolverá os softwares listados a seguir, que precisam estar previamente instalados (se necessário, confira a seção **Links** no final do artigo para saber onde baixá-los):

- Servidor de aplicação WildFly 8.0.0. Se quiser saber mais detalhes sobre como configurá-lo, consulte a Java Magazine 128;
- Eclipse Luna para desenvolvedores Java EE;
- Java Development Kit versão 1.7 configurada no Eclipse;
- SGBD PostgreSQL 9.1 ou superior.

Agora é hora de criar o projeto Maven. Para isto, na *view Project Explorer* do Eclipse, clique com o botão direito, selecione *New* e clique em *Other*. Na tela que surge, selecione *Maven* e depois clique em *Maven Project*. Marque a opção *Create a simple Project (skip archetype selection)* e clique em *Next*. Neste passo, informe no campo *Group Id* o valor “*br.com.javamagazine*”, no *Artifact Id* o valor “*clinicajm*”, no *Packaging* o valor “*war*” e no *Name* o valor “*clinicajm*”. Em seguida, clique em *Finish*.

Logo após estes passos iniciais, vamos configurar alguns parâmetros adicionais para a aplicação rodar corretamente. Assim, clique com o botão direito em cima da raiz do projeto, e depois na opção *Properties*. Na tela que surge, clique em *Project Facets*.

No item *Dynamic Web Module*, selecione a versão 3.1 para trabalharmos com a versão mais recente da API de Servlets e no item *Java*, selecione a opção 1.7 e clique em *OK*.

Após finalizarmos as alterações nas propriedades do projeto, nosso próximo passo será criar o arquivo *web.xml* com algumas configurações necessárias para o Spring MVC funcionar e a informação da página inicial do sistema. Para realizar esta alteração, crie primeiro uma pasta chamada *WEB-INF* dentro de *src/main/webapp* clicando com o botão direito em cima de *webapp*, selecione *New* e depois clique em *Folder*. Na tela que surge, informe “*WEB-INF*” em *Folder Name* e confirme. Agora clique com o botão direito em cima do diretório criado, selecione *New* e clique em *Other*. Na tela do *wizard*, selecione *XML*, *XML File* e clique em *Next*. No próximo passo, informe o nome “*web.xml*” e clique em *Finish*. Após a criação do arquivo, deixe o código igual ao da [Listagem 1](#).

**Listagem 1.** Código do arquivo *web.xml*.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
04   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
05   id="WebApp_ID" version="3.1">
06
07   <display-name>clinicajm</display-name>
08
09   <welcome-file-list>
10     <welcome-file>index.jsp</welcome-file>
11   </welcome-file-list>
12
13   <servlet>
14     <servlet-name>dispatcher</servlet-name>
15     <servlet-class>org.springframework.web.servlet.DispatcherServlet
16     </servlet-class>
17     <load-on-startup>1</load-on-startup>
18   </servlet>
19   <servlet-mapping>
20     <servlet-name>dispatcher</servlet-name>
21     <url-pattern>*.do</url-pattern>
22   </servlet-mapping>
23
24   <listener>
25     <listener-class>org.apache.tiles.web.startup.simple.SimpleTilesListener
26   </listener>
27 </web-app>
```

As principais configurações desse código são as seguintes:

- Linhas 9 a 11: informa a página inicial do sistema como *index.jsp*;
- Linhas 13 a 17: registra a classe **DispatcherServlet** como *Front Controller* da nossa aplicação e informa que ela deve ser carregada no momento em que o sistema subir;
- Linhas 18 a 21: informa que as requisições a serem processadas pelo *Front Controller DispatcherServlet* devem seguir o *pattern* “\*.do”;
- Linhas 23 a 25: configura o *listener* do Tiles com a classe **SimpleTilesListener**. Entre outras ações ele lê o arquivo *WEB-INF/tiles.xml*, fornecendo suporte para JSP, Servlets e Portlets, o que já atende nossos requisitos, uma vez que utilizaremos apenas *templates* nas páginas JSP.

## Criando o arquivo *dispatcher-servlet.xml*

Dentro da mesma pasta *WEB-INF*, crie um novo arquivo XML chamado *dispatcher-servlet.xml*. Este arquivo irá informar para o Spring MVC a localização das páginas JSP, as configurações da JPA e onde o Spring deve procurar por classes injetáveis. Após a geração do arquivo, deixe o código igual ao da [Listagem 2](#).

**Listagem 2.** Código do arquivo *dispatcher-servlet.xml*.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans xmlns="http://www.springframework.org/schema/beans"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xmlns:p="http://www.springframework.org/schema/p"
05   xmlns:context="http://www.springframework.org/schema/context"
06   xmlns:tx="http://www.springframework.org/schema/tx"
07   xmlns:mvc="http://www.springframework.org/schema/mvc"
08   xsi:schemaLocation="http://www.springframework.org/schema/beans
09     http://www.springframework.org/schema/beans/spring-beans.xsd
10    http://www.springframework.org/schema/context
11    http://www.springframework.org/schema/context/spring-context.xsd
12    http://www.springframework.org/schema/tx
13    http://www.springframework.org/schema/tx/spring-tx.xsd
14    http://www.springframework.org/schema/mvc
15    http://www.springframework.org/schema/mvc/spring-mvc.xsd">
16
17   <context:component-scan base-package="br.com.javamagazine.clinicajm"/>
18
19   <mvc:annotation-driven />
20
21   <bean id="entityManagerFactory"
22     class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"/>
23
24   <bean id="transactionManager"
25     class="org.springframework.orm.jpa.JpaTransactionManager">
26     <property name="entityManagerFactory" ref="entityManagerFactory"/>
27   </bean>
28
29   <tx:annotation-driven />
30
31   <bean id="viewResolver"
32     class="org.springframework.web.servlet.view.UrlBasedViewResolver">
33     <property name="viewClass"
34       value="org.springframework.web.servlet.view.JstlView"/>
35     <property name="prefix" value="/WEB-INF/jsp/" />
36     <property name="suffix" value=".jsp" />
37   </bean>
38 </beans>
```

As principais configurações apresentadas nesse código são as seguintes:

- Linha 17: informa que o Spring MVC deve procurar componentes injetáveis a partir do pacote **br.com.javamagazine.clinicajm**;
- Linha 19: define que os *controllers* da aplicação serão definidos através de anotação;
- Linha 21: informa que a **EntityManagerFactory** (utilizada para se conseguir uma instância do **EntityManager**) será gerenciada pela **LocalEntityManagerFactoryBean**, uma classe do Spring;
- Linhas 23 a 25: informa que o gerenciamento de transações com o banco de dados será feito pela classe **JPATransactionManager**, uma vez que utilizaremos a JPA para o acesso;
- Linha 27: define que o controle de transações será feito através de anotações;

- Linhas 29 a 34: no Spring MVC, precisamos de um *viewResolver* para direcionar o usuário para uma tela após a execução da lógica no *controller*. A classe **UrlBasedViewResolver**, declarada na linha 29, é útil quando o nome dos arquivos das páginas é igual à **String** de retorno dos métodos (veremos mais à frente que os métodos dos nossos *controllers* se comportarão desta forma). Na linha 30 informamos que nossas telas serão páginas JSP usando *tags JSTL*. Nas linhas 31 e 32 definimos que nossas páginas terão a extensão “.jsp” e que ficarão na pasta *WEB-INF/jsp*. A colocação das páginas dentro de *WEB-INF* é considerada uma boa prática de segurança, já que o usuário do sistema não tem acesso direto aos arquivos nela presentes.

### Criando o arquivo tiles.xml

Agora que o Spring MVC está configurado, ainda na pasta *WEB-INF*, crie outro arquivo XML chamado *tiles.xml*. Este arquivo será utilizado pela biblioteca Tiles para identificação dos *templates* de *layout* da aplicação e seus fragmentos (comumente cabeçalho, rodapé, menu e corpo). Após a geração do arquivo, deixe o código igual ao da **Listagem 3**.

Observe que na linha 4 informamos que temos um *layout* padrão disponível para as páginas do sistema utilizarem – chamado *template.jsp* –, o qual recebeu, através do atributo **name**, o nome de **template**. Isto significa que *template.jsp* terá algumas estruturas que são comuns a várias páginas do sistema (como cabeçalho

e menu). Nesta configuração a biblioteca Tiles se encarrega de injetar o código do *template* nas páginas que o utilizam, evitando a repetição de código em todas as *views* do sistema que seguem o mesmo padrão de *layout*.

### Listagem 3. Código do arquivo tiles.xml

```
01 <?xml version="1.0" encoding="ISO-8859-1"?>
02 <!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation
//DTD Tiles Configuration 3.0//EN"
"http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
03 <tiles-definitions>
04   <definition name="template" template="/template.jsp">
05   </definition>
06 </tiles-definitions>
```

### Modificando o pom.xml

Utilizaremos várias bibliotecas diferentes não fornecidas diretamente pelo JDK no código do sistema e, para que elas fiquem disponíveis no projeto, precisamos configurá-las no *pom.xml*. Deste modo, clique duas vezes sobre o arquivo e selecione a aba *pom.xml* para editar o código fonte, que deve ficar semelhante ao da **Listagem 4**.

As principais configurações apresentadas nesse arquivo de configuração são as seguintes:

### Listagem 4. Código do arquivo pom.xml

```
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/xsd/maven-4.0.0.xsd">
03 <modelVersion>4.0.0</modelVersion>
04
05 <groupId>br.com.javamagazine</groupId>
06 <artifactId>clinicajm</artifactId>
07 <version>0.0.1-SNAPSHOT</version>
08 <packaging>war</packaging>
09 <name>clinicajm</name>
10 <build>
11   <plugins>
12     <plugin>
13       <artifactId>maven-compiler-plugin</artifactId>
14       <version>2.0.2</version>
15       <configuration>
16         <source>1.7</source>
17         <target>1.7</target>
18       </configuration>
19     </plugin>
20   </plugins>
21 </build>
22 <dependencies>
23   <dependency> 
24     <groupId>javax</groupId>
25     <artifactId>javaee-api</artifactId>
26     <version>7.0</version>
27     <scope>provided</scope>
28   </dependency>
29   <dependency>
30     <groupId>org.springframework</groupId>
31     <artifactId>spring-context</artifactId>
32     <version>4.1.2.RELEASE</version>
33   </dependency>
34   <dependency>
35     <groupId>org.springframework</groupId>
36     <artifactId>spring-webmvc</artifactId>
37     <version>4.1.2.RELEASE</version>
38   </dependency>
39   <dependency>
40     <groupId>org.springframeworkframework</groupId>
41     <artifactId>spring-orm</artifactId>
42     <version>4.1.2.RELEASE</version>
43   </dependency>
44   <dependency>
45     <groupId>org.apache.tiles</groupId>
46     <artifactId>tiles-jsp</artifactId>
47     <version>3.0.5</version>
48   </dependency>
49   <dependency>
50     <groupId>javax.servlet</groupId>
51     <artifactId>jstl</artifactId>
52     <version>1.2</version>
53     <scope>provided</scope>
54   </dependency>
55   <dependency>
56     <groupId>postgresql</groupId>
57     <artifactId>postgresql</artifactId>
58     <version>9.1-901.jdbc4</version>
59   </dependency>
60   <dependency>
61     <groupId>com.fasterxml.jackson.core</groupId>
62     <artifactId>jackson-core</artifactId>
63     <version>2.4.2</version>
64   </dependency>
65   <dependency>
66     <groupId>com.fasterxml.jackson.core</groupId>
67     <artifactId>jackson-databind</artifactId>
68     <version>2.4.2</version>
69   </dependency>
70 </dependencies>
71 </project>
```

- Linhas 5 a 8: especificações gerais do projeto informadas no início do artigo, ao criarmos a aplicação. O **groupId** identifica o grupo de projetos de uma organização e normalmente obedece à estrutura de pacotes do sistema. Já o **artifactId** é o nome utilizado para gerar o *war*;
- Linha 9: informa o nome do projeto;
- Linhas 10 a 21: configura a versão 1.7 para o *plugin* de compilação do código fonte do sistema. Desta forma, informamos ao Eclipse que o JDK 1.7 deve ser usado e que ele precisa ser colocado no *classpath* da aplicação;
- Linhas 22 a 70: define todas as dependências utilizadas no projeto;
- Linhas 23 a 28: informa que as bibliotecas Java EE 7 já estão no *classpath*. No nosso caso, são fornecidas pelo WildFly e a JPA será a única biblioteca do pacote utilizada pelo nosso sistema;
- Linhas 29 a 43: informa todas as dependências do Spring no nosso projeto. O módulo **spring-context** é considerado o *core* do Spring e traz o suporte para injecção de dependências, mensageria, entre outros.

**Listagem 5.** Script de criação do banco de dados.

```

01 CREATE SEQUENCE medico_id_seq INCREMENT BY 1;
02 CREATE SEQUENCE paciente_id_seq INCREMENT BY 1;
03 CREATE SEQUENCE consulta_id_seq INCREMENT BY 1;
04
05 CREATE TABLE "medico"
06   id INTEGER DEFAULT nextval('medico_id_seq') NOT NULL,
07   nome VARCHAR(120) NOT NULL,
08   especialidade VARCHAR(60) NOT NULL,
09   PRIMARY KEY (id);
10
11 CREATE TABLE "paciente"
12   id INTEGER DEFAULT nextval('paciente_id_seq') NOT NULL,
13   nome VARCHAR(120) NOT NULL,
14   data_nascimento DATE NOT NULL,
15   PRIMARY KEY (id);
16
17 CREATE TABLE "consulta"
18   id INTEGER DEFAULT nextval('consulta_id_seq') NOT NULL,
19   id_medico INTEGER NOT NULL,
20   id_paciente INTEGER NOT NULL,
21   sintomas VARCHAR(255) NULL,
22   receita VARCHAR(255) NULL,
23   data_consulta TIMESTAMP NOT NULL,
24   data_atendimento TIMESTAMP,
25   PRIMARY KEY (id);
26
27 ALTER TABLE "consulta" ADD CONSTRAINT medico_fk FOREIGN KEY
28 (id_medico) REFERENCES "medico"(id);
29 ALTER TABLE "consulta" ADD CONSTRAINT paciente_fk FOREIGN KEY
30 (id_paciente) REFERENCES "paciente"(id);

```

O módulo **spring-webmvc** fornece recursos do Spring MVC e o **spring-orm** mecanismos para acesso a dados;

- Linhas 44 a 48: define a dependência do Apache Tiles. Como vamos usar *templates* baseados em páginas JSP, basta utilizarmos o módulo **tiles-jsp**;
- Linhas 49 a 54: informa que a biblioteca JSTL já está no *classpath*. Em nosso ambiente ela será fornecida pelo WildFly;
- Linhas 55 a 59: define o *driver* do Postgres como dependência para podermos acessar o SGBD de código aberto;
- Linhas 60 a 70: define a biblioteca de processamento de dados Jackson como dependência do nosso projeto. Ela será utilizada para geração de dados no formato JSON nas requisições Ajax que iremos criar.

Após modificar o *pom.xml*, vamos atualizar as dependências do Maven no sistema. Para realizar esse procedimento, clique com o botão direito em cima da raiz do projeto, selecione *Maven* e clique em *Update Project*. Na tela que surge, clique em *OK* e aguarde alguns segundos para que o Eclipse realize o processo.

## Criando o banco de dados

Agora que as configurações básicas do projeto foram efetuadas, crie uma base de dados chamada **clinicajm** no PostgreSQL e rode o *script* da **Listagem 5** para gerar as três tabelas que iremos utilizar.

Após a criação das tabelas, o diagrama relacional da nossa base de dados ficará como o mostrado pela **Figura 2**.

## Configurando a JPA

Para inserirmos a JPA em nosso projeto, é necessário primeiramente criar o XML de configuração da biblioteca. Deste modo, crie inicialmente uma pasta chamada **META-INF** dentro de *src/main/resources*. Após realizar este passo, crie um XML chamado **persistence.xml** dentro do novo diretório e modifique o código gerado automaticamente para ficar igual ao da **Listagem 6**.

As principais linhas do arquivo *persistence.xml* são:

- Linha 6: definimos o nome **clinicaPU** para nossa **PersistenceUnit** e indicamos que o tipo de transação é **RESOURCE\_LOCAL**, o que significa que nós teremos que obter o **EntityManager** através da **EntityManagerFactory** e que o controle das transações será manual;
- Linha 7: determina qual será a implementação do **EntityManager**. Como o WildFly usa o Hibernate, optamos pela classe **HibernatePersistence**;



**Figura 2.** Diagrama relacional da base de dados

- Linhas 8 a 10: lista as entidades que nosso projeto terá;
- Linhas 11 a 16: configura o dialeto a ser usado pelo Hibernate (do PostgreSQL), o *driver* e os parâmetros da conexão com o banco.

**Listagem 6.** Código do arquivo persistence.xml.

```

01 <?xml version="1.0" encoding="UTF-8"?>
02 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
05   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
06   version="2.0">
07   <persistence-unit name="clinicaPU" transaction-type="RESOURCE_LOCAL">
08     <provider>org.hibernate.ejb.HibernatePersistence</provider>
09     <class>br.com.javamagazine.clinicajm.domain.Consulta</class>
10     <class>br.com.javamagazine.clinicajm.domain.Medico</class>
11     <class>br.com.javamagazine.clinicajm.domain.Paciente</class>
12     <properties>
13       <property name="hibernate.dialect"
14         value="org.hibernate.dialect.PostgreSQLDialect"/>
15       <property name="javax.persistence.jdbc.driver"
16         value="org.postgresql.Driver"/>
17       <property name="javax.persistence.jdbc.user" value="postgres"/>
18       <property name="javax.persistence.jdbc.password" value="1234"/>
19       <property name="javax.persistence.jdbc.url"
20         value="jdbc:postgresql://localhost:5432/clinicajm"/>
21     </properties>
22   </persistence-unit>
23 </persistence>

```

## Criando as entidades e o enum do projeto

Como vimos, tivemos que declarar três entidades no arquivo *persistence.xml* para mapear os atributos das classes com os campos de entrada de dados nas telas. São elas: **Medico**, **Paciente** e **Consulta**. Para criar a primeira delas (**Medico**), clique com botão direito em cima do projeto, selecione a opção *New* e depois clique em *Class*. Na tela que surge, informe em *Package* o valor “*br.com.javamagazine.clinicajm.domain*”, em *Name* informe “*Medico*” e clique no botão *Finish*. O código inicial que o Eclipse gera automaticamente deve ser modificado para ficar semelhante ao da **Listagem 7** (lembre-se de inserir os *gets*, *sets*, *equals()* e *hashCode()* que foram omitidos).

Por enquanto a classe ficará com erro, pois geraremos o enum **Especialidade** mais à frente. Os principais trechos desse código podem ser explicados da seguinte forma:

- Linha 9: a anotação **@Entity** indica que esta classe é uma entidade mapeada para uma tabela no banco de dados (por padrão, o nome da tabela é o mesmo nome da classe, mas com a primeira letra também em minúsculo);
- Linha 14: seta o atributo **id** como chave primária da tabela;
- Linha 15: define que o valor do atributo **id** será gerado automaticamente através de uma *sequence* no banco;
- Linha 16: configura a *sequence* **medico\_id\_seq**, que será usada pelo atributo **id**;
- Linhas 20 e 21: define que a classe **Medico** terá um atributo mapeado através de um enum e que ele será gravado no banco de dados no formato **String**.

Repetindo o mesmo procedimento para geração de uma classe, crie a entidade **Paciente** no pacote informado anteriormente e

substitua o código gerado automaticamente pelo Eclipse para ficar igual ao da **Listagem 8** (lembre-se de inserir os *gets*, *sets*, *equals()* e *hashCode()*).

A novidade nesse código está na linha 22, com o uso da classe **DateTimeFormat**, que define com qual formato o Spring MVC deve enviar o dado em uma requisição vinda da JSP e direcionada para o *controller*.

**Listagem 7.** Código da classe Medico.

```

01 package br.com.javamagazine.clinicajm.domain;
02
03 import java.io.Serializable;
04
05 import javax.persistence.*;
06
07 import br.com.javamagazine.clinicajm.domain.enumeration.Especialidade;
08
09 @Entity
10 public class Medico implements Serializable {
11
12   private static final long serialVersionUID = 714006989890638856L;
13
14   @Id
15   @GeneratedValue(generator="medico_id_seq",
16     strategy=GenerationType.SEQUENCE)
16   @SequenceGenerator(name="medico_id_seq",
17     sequenceName="medico_id_seq", allocationSize=1)
17   private Integer id;
18   private String nome;
19
20   @Enumerated(EnumType.STRING)
21   private Especialidade especialidade;
22
23   // gets e sets omitidos
24   // equals() e hashCode() omitidos
25 }

```

**Listagem 8.** Código da classe Paciente.

```

01 package br.com.javamagazine.clinicajm.domain;
02
03 import java.io.Serializable;
04 import java.util.Date;
05
06 import javax.persistence.*;
07
08 import org.springframework.format.annotation.DateTimeFormat;
09
10 @Entity
11 public class Paciente implements Serializable {
12
13   private static final long serialVersionUID = 3605331584324240290L;
14
15   @Id
16   @GeneratedValue(generator="paciente_id_seq",
17     strategy=GenerationType.SEQUENCE)
17   @SequenceGenerator(name="paciente_id_seq",
18     sequenceName="paciente_id_seq", allocationSize=1)
18   private Integer id;
19   private String nome;
20
21   @Column(name="data_nascimento")
22   @DateTimeFormat(pattern="dd/MM/yyyy")
23   private Date dataNascimento;
24
25   // gets e sets omitidos
26   // equals() e hashCode() omitidos
27 }

```

O framework se encarrega de fazer a conversão automática de **String** para **Date**, evitando que o desenvolvedor tenha que se preocupar com isto. Sem o uso da anotação, o procedimento dá erro porque a JSP envia o dado em um formato que o HTTP não aceita.

Nosso próximo passo será criar a terceira entidade, **Consulta**, no mesmo pacote das outras duas classes. Após esse procedimento, substitua o código gerado para ficar igual ao da **Listagem 9** (lembre-se novamente de inserir os gets, sets, **equals()** e **hashCode()**).

#### Listagem 9. Código da classe Consulta.

```
01 package br.com.javamagazine.clinicajm.domain;
02
03 import java.io.Serializable;
04 import java.util.Date;
05
06 import javax.persistence.*;
07
08 import org.springframework.format.annotation.DateTimeFormat;
09
10 @Entity
11 public class Consulta implements Serializable {
12
13     private static final long serialVersionUID = 7064809078222302493L;
14
15     @Id
16     @GeneratedValue(generator="consulta_id_seq",
17                     strategy=GenerationType.SEQUENCE)
17     @SequenceGenerator(name="consulta_id_seq",
18                         sequenceName="consulta_id_seq", allocationSize=1)
18     private Integer id;
19     private String sintomas;
20     private String receita;
21
22     @ManyToOne(fetch = FetchType.EAGER)
23     @JoinColumn(name="id_medico")
24     private Medico medico = new Medico();
25
26     @ManyToOne(fetch = FetchType.EAGER)
27     @JoinColumn(name="id_paciente")
28     private Paciente paciente = new Paciente();
29
30     @Column(name="data_consulta")
31     @Temporal(TemporalType.TIMESTAMP)
32     @DateTimeFormat(pattern="dd/MM/yyyy HH:mm:ss")
33     private Date dataConsulta;
34
35     @Column(name="data_atendimento")
36     @Temporal(TemporalType.TIMESTAMP)
37     private Date dataAtendimento;
38
39     // gets e sets omitidos
40     // equals() e hashCode() omitidos
41 }
```

Analisando esse código, podemos ver que os principais trechos são os seguintes:

- Linhas 22 a 24: define que a classe **Consulta** terá uma relação de muitos para um com **Medico**, expressa pelo atributo **medico**, e que a chave estrangeira na tabela **consulta** será mapeada pela coluna **id\_medico**. A instrução **fetch = FetchType.EAGER** indica que toda vez que uma consulta for recuperada do banco, o atributo **medico** deve estar preenchido com os dados correspondentes;

- Linhas 26 e 28: define que a classe **Consulta** terá uma relação de muitos para um com **Paciente**, expressa pelo atributo **paciente**, e que a chave estrangeira na tabela **consulta** será mapeada pela coluna **id\_paciente**;
- Linha 30: informa que o atributo **dataConsulta** mapeia a coluna **data\_consulta** da tabela **paciente**;
- Linhas 31 e 36: define que **dataConsulta** e **dataAtendimento** devem ser armazenados no banco como **timestamp**.

Por fim, vamos criar o nosso enum **Especialidade**, que poderia ser uma entidade no sistema, mas para ficarmos com o artigo menos extenso vamos criá-lo desta forma. Assim, clique com botão direito em cima do projeto, selecione a opção *New* e depois clique em *Enum*. Na tela que surge, informe em *Package* o valor “*br.com.javamagazine.clinicajm.domain.enumeration*”, em *Name* informe “**Especialidade**” e clique no botão *Finish*. Logo após, modifique o código para ficar semelhante ao da **Listagem 10**.

#### Listagem 10. Código do enum Especialidade.

```
01 package br.com.javamagazine.clinicajm.domain.enumeration;
02
03 import java.io.Serializable;
04
05 public enum Especialidade implements Serializable {
06
07     CARDIOLOGISTA("Cardiologista"),
08     CLINICO_GERAL("Clínico Geral"),
09     DERMATOLOGISTA("Dermatologista"),
10     OTORRINOLARINGOLOGISTA("Otorrinolaringologista"),
11     PEDIATRA("Pediatra"),
12     PNEUMOLOGISTA("Pneumologista");
13
14     private String descricao;
15
16     private Especialidade(String descricao){
17         this.descricao = descricao;
18     }
19
20     public String getDescricao() {
21         return descricao;
22     }
23
24     public void setDescricao(String descricao) {
25         this.descricao = descricao;
26     }
27 }
```

## Criando os repositórios

Nosso próximo passo na sequência da arquitetura será criar classes utilizando a anotação do Spring **@Repository** para realizarem as operações no banco de dados. Estas classes terão toda a lógica de acesso ao SGBD e devolverão o resultado para nossos **controllers**.

Seguindo a mesma sequência das entidades, a primeira será **MedicoRepository**. Para criá-la, clique com o botão direito em cima do projeto, selecione *New* e depois clique em *Class*. Na tela que surge, informe em *Package* o valor “*br.com.javamagazine.clinicajm.repository*”, em *Name* informe “**MedicoRepository**” e confirme no botão *Finish*.

Da mesma forma que fizemos anteriormente, precisamos modificar o código gerado pelo Eclipse para ficar semelhante ao da **Listagem 11**.

**Listagem 11.** Código da classe MedicoRepository.

```
01 package br.com.javamagazine.clinicajm.repository;
02
03 import java.util.List;
04
05 import javax.persistence.*;
06
07 import org.springframework.stereotype.Repository;
08 import org.springframework.transaction.annotation.Transactional;
09
10 import br.com.javamagazine.clinicajm.domain.Medico;
11 import br.com.javamagazine.clinicajm.domain.enumeration.Especialidade;
12
13 @Repository
14 public class MedicoRepository {
15
16     @PersistenceContext
17     private EntityManager entityManager;
18
19     @Transactional
20     public void salvaMedico(Medico medico) {
21         entityManager.persist(medico);
22     }
23
24     @Transactional
25     public void excluiMedico(Integer id) {
26         Medico medico = entityManager.find(Medico.class, id);
27         entityManager.remove(medico);
28     }
29
30     @SuppressWarnings("unchecked")
31     public List<Medico> listaMedicos() {
32         Query query = entityManager.createQuery(
33             "Select m from Medico m order by m.id");
34         return query.getResultList();
35     }
36     @SuppressWarnings("unchecked")
37     public List<Medico> listaMedicosPorEspecialidade(
38         Especialidade especialidade) {
39         Query query = entityManager.createQuery(
40             "Select m from Medico m where m.especialidade=:especialidade
41             order by m.nome");
42         query.setParameter("especialidade", especialidade);
43     }
44 }
```

Podemos verificar que os principais trechos no código do nosso primeiro repositório são os seguintes:

- Linha 13: define a classe como repositório através da anotação **@Repository**. Isto significa que **MedicoRepository** pertence à camada de persistência e que ela pode ser injetada em outras classes da nossa aplicação;
- Linhas 16 e 17: a anotação **@PersistenceContext** da JPA injeta o **EntityManager** na classe **MedicoRepository** a partir dos dados de conexão informados no arquivo *persistence.xml* para podermos realizar as operações no banco de dados;
- Linha 19: informa que o método deve ser transacional, ou seja, o *commit* na operação é realizado automaticamente, assim como um eventual *rollback*;

- Linha 21: invoca o método **persist()** do **EntityManager**. Este método realiza uma operação de *insert* no banco de dados, gravando o objeto **medico** passado na tabela correspondente;
- Linhas 24 a 28: exclui o médico com o **id** passado da tabela **medico**. Primeiro pesquisa o objeto **Medico** através do **id** invocando **find()** e depois chama o método **remove()** para removê-lo;
- Linhas 32 e 38: cria um objeto do tipo **Query** a partir do método **createQuery()** do **EntityManager**. A classe **Query** oferece métodos para consulta e atualização de dados no banco de dados;
- Linhas 33 e 41: o método **getResultSet()** da classe **Query** é chamado para retornar uma listagem do banco de dados;
- Linha 39: invoca o método **setParameter()** da classe **Query**. Este método substitui um parâmetro na consulta (através da expressão '**especialidade**') pelo valor passado na chamada do método.

Nosso próximo repositório, **PacienteRepository**, será responsável por realizar as operações que dizem respeito aos pacientes no banco de dados. Assim, crie a classe no mesmo pacote do repositório anterior e substitua o código gerado pelo da **Listagem 12**.

**Listagem 12.** Código da classe PacienteRepository.

```
01 package br.com.javamagazine.clinicajm.repository;
02
03 import java.util.List;
04
05 import javax.persistence.*;
06
07 import org.springframework.stereotype.Repository;
08 import org.springframework.transaction.annotation.Transactional;
09
10 import br.com.javamagazine.clinicajm.domain.Paciente;
11
12 @Repository
13 public class PacienteRepository {
14
15     @PersistenceContext
16     private EntityManager entityManager;
17
18     @Transactional
19     public void salvaPaciente(Paciente paciente) {
20         entityManager.persist(paciente);
21     }
22
23     @Transactional
24     public void excluiPaciente(Integer id) {
25         Paciente paciente = entityManager.find(Paciente.class, id);
26         entityManager.remove(paciente);
27     }
28
29     @SuppressWarnings("unchecked")
30     public List<Paciente> listaPacientes() {
31         Query query = entityManager.createQuery(
32             "Select p from Paciente p order by p.id");
33         return query.getResultList();
34     }
35 }
```

Observe nesse código que, da mesma forma que fizemos na classe **MedicoRepository**, chamamos os métodos **persist()** e **createQuery()** de **EntityManager** e **getResultSet()** da classe **Query**. O código, portanto, é bastante semelhante ao da classe anterior.

Nosso último repositório será **ConsultaService**. Sua função é realizar as operações relativas às consultas dos pacientes no banco de dados. Desta forma, crie a classe dentro do mesmo pacote **br.com.javamagazine.clinicajm.repository** e substitua o código gerado pelo da **Listagem 13**.

**Listagem 13.** Código da classe ConsultaRepository.

```
01 package br.com.javamagazine.clinicajm.repository;
02
03 import java.util.List;
04
05 import javax.persistence.*;
06
07 import org.springframework.stereotype.Repository;
08 import org.springframework.transaction.annotation.Transactional;
09
10 import br.com.javamagazine.clinicajm.domain.Consulta;
11
12 @Repository
13 public class ConsultaRepository {
14
15     @PersistenceContext
16     private EntityManager entityManager;
17
18     @Transactional
19     public void salvaConsulta(Consulta consulta) {
20         entityManager.persist(consulta);
21     }
22
23     @Transactional
24     public void atualizaConsulta(Consulta consulta) {
25         entityManager.merge(consulta);
26     }
27
28     public Consulta recuperaConsulta(Integer id) {
29         return entityManager.find(Consulta.class, id);
30     }
31
32     @SuppressWarnings("unchecked")
33     public List<Consulta> listarPorPaciente(Integer idPaciente) {
34         Query query = entityManager.createQuery(
35             "Select c from Consulta c where c.paciente.id=:idPaciente
36             order by c.dataConsulta");
37         query.setParameter("idPaciente", idPaciente);
38     }
39 }
```

Assim como nos repositórios anteriores, esse código traz chamadas aos métodos **find()**, **persist()** e **createQuery()** da classe **EntityManager**, além de **setParameter()** e **getResultSet()** da classe **Query**. A única novidade está na invocação do método **merge()**, na linha 25, utilizado para atualizar um objeto no banco de dados.

## Criando a classe para armazenar mensagens

Em nosso projeto será necessário devolver para o usuário mensagens de sucesso, erro ou aviso após alguma ação ser disparada. Desta forma, vamos criar uma classe que irá armazenar estas informações para que possamos exibi-las nas páginas. Para realizar este passo, clique com o botão direito em cima do projeto,

selecione **New** e depois clique em **Class**. Na tela que surge, informe em **Package** o valor “**br.com.javamagazine.clinicajm.util**”, em **Name** informe “**Mensagem**” e confirme no botão **Finish**. Feito isso, modifique o código gerado pelo Eclipse para ficar semelhante ao da **Listagem 14**.

**Listagem 14.** Código da classe Mensagem.

```
01 package br.com.javamagazine.clinicajm.util;
02
03 public class Mensagem {
04
05     public enum TipoMensagem {
06         ERRO("alert alert-danger"),
07         AVISO("alert alert-warning"),
08         SUCESSO("alert alert-success");
09     }
09
10     private String classeCss;
11
12     private TipoMensagem(String classeCss){
13         this.classeCss = classeCss;
14     }
15     public String getClasseCss() {
16         return classeCss;
17     }
18     public void setClasseCss(String classeCss) {
19         this.classeCss = classeCss;
20     }
21 }
22
23 private String texto;
24 private TipoMensagem tipoMensagem;
25
26 public Mensagem(String texto,TipoMensagem tipoMensagem) {
27     this.texto = texto;
28     this.tipoMensagem = tipoMensagem;
29 }
30
31 public String getTexto() {
32     return texto;
33 }
34 public void setTexto(String texto) {
35     this.texto = texto;
36 }
37 public TipoMensagem getTipoMensagem() {
38     return tipoMensagem;
39 }
40 public void setTipoMensagem(TipoMensagem tipoMensagem) {
41     this.tipoMensagem = tipoMensagem;
42 }
43 }
```

Note que criamos o enum interno **TipoMensagem** para indicar o tipo de mensagem (se erro, sucesso ou aviso) e as respectivas classes do CSS que devem ser colocadas na **div** a ser exibida na página JSP com a informação desejada. Para armazenarmos estes dados foi criado um atributo chamado **tipoMensagem** na classe **Mensagem** apontando para uma instância de **TipoMensagem**.

Veremos, na segunda parte do artigo, que haverá um tratamento no **template** das páginas JSP do projeto para verificar se existe mensagem de sucesso, erro ou aviso a ser mostrada para

o usuário. Em caso positivo, esta rotina adicionará uma *div* do Bootstrap na página. Esta *div* apresentará no atributo *class* o valor contido em *classeCss* da variável *tipoMensagem* presente nas instâncias de **Mensagem**. Já o conteúdo interno da *div* será o texto tratado pela aplicação e armazenado no atributo **texto** da classe **Mensagem**.

### Criando os Controllers

Agora que temos as entidades e os repositórios criados, precisamos dos *controllers* para realizar a comunicação com as páginas JSP e as classes de repositório. Nesta demonstração, definiremos quatro classes: **NavegacaoController**, **MedicoController**, **PacienteController** e **ConsultaController**. Para seguir esta mesma sequência, nosso primeiro *Controller* será **NavegacaoController**, a ser criado no pacote **br.com.javamagazine.clinicajm.controller**. Ele tem como objetivo preparar objetos e enviá-los para que as páginas de cadastro de médico, paciente, consulta e atendimento funcionem corretamente. Da mesma maneira que fizemos nas etapas anteriores, precisamos modificar o código gerado pelo Eclipse para ficar semelhante ao da **Listagem 15**.

Os principais trechos desse código podem ser explicados da seguinte forma:

- Linha 15: a anotação do Spring MVC **@Controller** transforma a classe em um *controller*, ou seja, permite que ele seja invocado a partir de requisições HTTP no sistema;
- Linhas 18 e 19: a anotação do Spring **@Autowired** define que o repositório **PacienteRespository** deve ser injetado em **NavegacaoController**;
- Linha 21: define que o método **redirecionaCadastroMedico()** deve ser chamado em toda requisição que tenha a URL terminada com `"/preparaCadastroMedico.do"`. O método adiciona na requisição atributos necessários para o preenchimento dos dados do médico no formulário da página JSP, chamada em seguida;
- Linhas 22 a 27: definição do método **redirecionaCadastroMedico()**, que coloca no objeto do tipo **Map<String, Object>** as especialidades de médico disponíveis no sistema e uma nova instância do objeto **Medico**. Observe que as chaves colocadas no **Map** são “especialidades” e “medico”. Isto significa que na página JSP que será chamada teremos duas variáveis com estes nomes – ambas com escopo de requisição, sendo uma do tipo **Especialidade[]** e outra do tipo **Medico** – para manipularmos. Na linha 26, veja que retornamos **cadastrarMedico**. Neste momento, o Spring MVC entende que a página **cadastrarMedico.jsp**, localizada dentro de **WEB-INF/jsp**, deve ser

## CURSOS ONLINE

A Revista Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



### CONHEÇA ALGUNS DOS CURSOS:

- **Curso de Multicamadas com Delphi e DataSnap**
- **Delphi para Iniciantes**
- **Criando componente Boleto em Delphi**
- **Loja Virtual em Delphi Prism**

Para mais informações :

[www.devmedia.com.br/cursos/delphi](http://www.devmedia.com.br/cursos/delphi)  
(21) 3382-5038

chamada (lembre-se da configuração que colocamos no arquivo `dispatcher-servlet.xml`).

Nosso segundo `controller` a ser criado realizará as operações de armazenar, listar e remover os médicos do banco de dados. Assim, crie a classe **MedicoController** no mesmo pacote do `controller`

anterior e deixe o código igual ao da **Listagem 16**.

Nesse código temos algumas novidades ainda não utilizadas no projeto. Verificando a classe, podemos perceber que os trechos mais relevantes são:

- Linha 15: a anotação `@RequestMapping`, com o valor `/medico`, colocada em nível de classe, indica que qualquer `@RequestMapping`

## Listagem 15. Código da classe NavegacaoController.

```
01 package br.com.javamagazine.clinicajm.controller;
02
03 import java.util.Map;
04
05 import org.springframework.beans.factory.annotation.Autowired;
06 import org.springframework.stereotype.Controller;
07 import org.springframework.web.bind.annotation.RequestMapping;
08
09 import br.com.javamagazine.clinicajm.domain.Consulta;
10 import br.com.javamagazine.clinicajm.domain.Medico;
11 import br.com.javamagazine.clinicajm.domain.Paciente;
12 import br.com.javamagazine.clinicajm.domain.enumeration.Especialidade;
13 import br.com.javamagazine.clinicajm.repository.PacienteRepository;
14
15 @Controller
16 public class NavegacaoController {
17
18     @Autowired
19     private PacienteRepository pacienteRepository;
20
21     @RequestMapping(value="/preparaCadastroMedico.do")
22     public String redirecionaCadastroMedico(Map<String, Object> map) {
23         map.put("especialidades", Especialidade.values());
24         map.put("medico", new Medico());
25
26     return "cadastrarMedico";
27     }
28
29     @RequestMapping(value="/preparaCadastroPaciente.do")
30     public String redirecionaCadastroPaciente(Map<String, Object> map) {
31         map.put("paciente", new Paciente());
32
33     return "cadastrarPaciente";
34     }
35
36     @RequestMapping(value="/preparaCadastroConsulta.do")
37     public String redirecionaCadastroConsulta(Map<String, Object> map) {
38         map.put("especialidades", Especialidade.values());
39         map.put("pacientes", pacienteRepository.listaPacientes());
40         map.put("consulta", new Consulta());
41
42     return "cadastrarConsulta";
43     }
44
45     @RequestMapping(value="/preparaCadastroAtendimento.do")
46     public String redirecionaCadastroAtendimento(Map<String, Object> map) {
47         map.put("pacientes", pacienteRepository.listaPacientes());
48
49     return "listarConsultas";
50     }
51 }
```

## Listagem 16. Código da classe MedicoController.

```
01 package br.com.javamagazine.clinicajm.controller;
02
03 import java.util.List;
04
05 import org.springframework.beans.factory.annotation.Autowired;
06 import org.springframework.stereotype.Controller;
07 import org.springframework.ui.Model;
08 import org.springframework.web.bind.annotation.RequestMapping;
09 import org.springframework.web.bind.annotation.RequestMethod;
10 import org.springframework.web.bind.annotation.ResponseBody;
11
12 // imports das classes do projeto omitidos
13
14 @Controller
15 @RequestMapping("/medico")
16 public class MedicoController {
17
18     @Autowired
19     private MedicoRepository medicoRepository;
20
21     @RequestMapping(value="/cadastrar.do", method=RequestMethod.POST)
22     public String cadastrar(Medico medico, Model model) {
23         medicoRepository.salvaMedico(medico);
24         model.addAttribute("medico", new Medico());
25         model.addAttribute("especialidades", Especialidade.values());
26         model.addAttribute("mensagem", new Mensagem ("Sucesso ao cadastrar o
médico", TipoMensagem.SUCESSO));
27
28     return "cadastrarMedico";
29     }
30
31     @RequestMapping(value="/listar.do", method=RequestMethod.GET)
32     public String listar(Model model) {
33         List<Medico> medicos = medicoRepository.listaMedicos();
34         model.addAttribute("medicos", medicos);
35
36     return "listarMedicos";
37     }
38
39     @RequestMapping(value="/excluir.do", method=RequestMethod.GET)
40     public String excluir(Integer idMedico, Model model) {
41         medicoRepository.excluiMedico(idMedico);
42         model.addAttribute("mensagem", new Mensagem("Sucesso ao excluir o
médico", TipoMensagem.SUCESSO));
43
44     return "forward:/medico/listar.do";
45     }
46
47     @RequestMapping(value="/listarPorEspecialidade.do",
method=RequestMethod.GET)
48     public @ResponseBody List<Medico> listarPorEspecialidade
(Especialidade especialidade) {
49         return medicoRepository.listaMedicosPorEspecialidade(especialidade);
50     }
51 }
```

presente nos métodos da classe automaticamente obriga que a URL de chamada tenha antes o prefixo **/medico**;

- Linhas 18 e 19: injeta uma instância de **MedicoRepository** na classe **MedicoController**;
- Linha 21: a instrução **method=RequestMethod.POST** informa ao Spring MVC que este método deve ser chamado através de uma requisição *post* e a instrução **value="/cadastrar.do"** que a requisição direcionada para esta ação deve terminar com **"/medico/cadastrar.do"**;
- Linhas 22 a 39: declara o método **cadastrar()** que realiza o cadastro do médico no banco de dados. O objeto do tipo **Model** permite que adicionemos atributos a ele para que sejam recuperados posteriormente na JSP. Na linha 30, por exemplo, adicionamos uma instância da classe **Mensagem** e damos ao atributo o nome **mensagem**. Desta forma, na página JSP ficamos com uma variável chamada **mensagem** para utilizarmos da forma que necessitarmos;
- Linha 44: declara que o retorno do método **excluir()** deve ser um *forward* para o endereço **/medico/listar.do**, ou seja, após executar a remoção do médico do banco de dados, chamamos novamente a operação de listagem;
- Linhas 47 a 50: declara o método **listaPorEspecialidade()** que lista, através de uma requisição Ajax, todos os médicos de uma determinada especialidade. Observe que na linha 52 usamos a anotação **@ResponseBody** para indicar que o retorno do método deve ser convertido para JSON. Esta conversão acontece automaticamente pelo Spring MVC porque configuramos em nosso *pom.xml* as dependências da biblioteca Jackson, fazendo com que a "mágica" toda aconteça.

O terceiro *controller* da aplicação será **PacienteController**. Ele conterá toda a lógica das operações relacionadas aos pacientes da clínica e deve ficar no mesmo pacote das duas classes anteriores. Pelo código exposto na **Listagem 17**, constatamos que as funcionalidades relacionadas aos pacientes no sistema serão poucas e bastante simples, existindo apenas os métodos **cadastrar()**, **listar()** e **excluir()**.

Esse código é bastante semelhante ao da classe **MedicoController**, com a diferença que não temos o método que retorna conteúdo JSON.

O último *controller* da nossa aplicação é o **ConsultaController**, responsável por realizar as operações de gravação da consulta e do atendimento dos pacientes. O pacote da classe deve ser o mesmo das três anteriores e o código precisa ficar igual ao da **Listagem 18**.

Assim como na classe **MedicoController**, temos um método que retorna conteúdo JSON. Trata-se do **listarPorPaciente()**, que devolve a lista de consultas de um paciente a partir do id fornecido. No próximo artigo, veremos que o retorno do método é tratado em uma página para exibir em uma tabela as consultas do paciente, após ele ser selecionado em um dos campos de formulário da tela.

Além do **listarPorPaciente()**, a classe **PacienteController** apresenta o método **agendar()**, que possui uma rotina para transformar a data e a hora da consulta informadas pelo usuário na tela – inicialmente no formato **String** – em valores do tipo **Date**. Esta é a única validação de campos de formulário na aplicação e serve para demonstrar o disparo de uma mensagem de erro para a página.

#### Listagem 17. Código da classe PacienteController.

```
01 package br.com.javamagazine.clinicajm.controller;
02
03 import java.util.List;
04
05 import org.springframework.beans.factory.annotation.Autowired;
06 import org.springframework.stereotype.Controller;
07 import org.springframework.ui.Model;
08 import org.springframework.web.bind.annotation.RequestMapping;
09 import org.springframework.web.bind.annotation.RequestMethod;
10
11 // imports das classes do projeto omitidos
12
13 @Controller
14 @RequestMapping("/paciente")
15 public class PacienteController {
16
17     @Autowired
18     private PacienteRepository pacienteRepository;
19
20     @RequestMapping(value="/cadastrar.do", method=RequestMethod.POST)
21     public String cadastrar(Paciente paciente, Model model) {
22         pacienteRepository.salvaPaciente(paciente);
23         model.addAttribute("paciente", new Paciente());
24         model.addAttribute("mensagem",
25             new Mensagem("Sucesso ao cadastrar o paciente", TipoMensagem.SUCESSO));
26
27     }
28
29     @RequestMapping(value="/listar.do", method=RequestMethod.GET)
30     public String listar(Model model) {
31         List<Paciente> pacientes = pacienteRepository.listaPacientes();
32         model.addAttribute("pacientes", pacientes);
33
34     }
35
36     @RequestMapping(value="/excluir.do", method=RequestMethod.GET)
37     public String excluir(Integer idPaciente, Model model) {
38         pacienteRepository.excluiPaciente(idPaciente);
39         model.addAttribute("mensagem", new Mensagem
40             ("Sucesso ao excluir o paciente", TipoMensagem.SUCESSO));
41
42     }
43 }
```

## Listagem 18. Código da classe ConsultaController.

```
01 package br.com.javamagazine.clinicajm.controller;
02
03 import java.text.ParseException;
04 import java.text.SimpleDateFormat;
05 import java.util.Date;
06 import java.util.List;
07
08 import org.springframework.beans.factory.annotation.Autowired;
09 import org.springframework.stereotype.Controller;
10 import org.springframework.ui.Model;
11 import org.springframework.web.bind.annotation.RequestMapping;
12 import org.springframework.web.bind.annotation.RequestMethod;
13 import org.springframework.web.bind.annotation.ResponseBody;
14
15 // imports das classes do projeto omitidos
16
17 @Controller
18 @RequestMapping("/consulta")
19 public class ConsultaController {
20
21     @Autowired
22     private ConsultaRepository consultaRepository;
23
24     @RequestMapping(value="/agendar.do", method=RequestMethod.POST)
25     public String agendar(Consulta consulta, String data, String hora, Model model) {
26         SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm");
27         Date dataConsulta;
28         try {
29             dataConsulta = sdf.parse(data + " " + hora);
30             consulta.setDataConsulta(dataConsulta);
31             consultaRepository.salvaConsulta(consulta);
32             model.addAttribute("mensagem", new Mensagem("Sucesso ao cadastrar a
33             consulta", TipoMensagem.SUCESSO));
34         } catch (ParseException e) {
35             model.addAttribute("mensagem", new Mensagem("Erro ao fazer a conver
36             são de data/hora. Observe os padrões a serem seguidos",
37             TipoMensagem.ERRO));
38         }
39
40     @RequestMapping(value="/detalharConsulta.do", method=RequestMethod.GET)
41     public String detalhar(Integer idConsulta, Model model) {
42         Consulta consulta = consultaRepository.recuperaConsulta(idConsulta);
43         model.addAttribute("consulta", consulta);
44
45         return "realizarAtendimento";
46     }
47
48     @RequestMapping(value="/atender.do", method=RequestMethod.POST)
49     public String gravarAtendimento(Consulta consulta, Model model) {
50         consulta.setDataAtendimento(new Date());
51         consultaRepository.atualizaConsulta(consulta);
52         model.addAttribute("mensagem", new Mensagem
53             ("Sucesso ao cadastrar o atendimento", TipoMensagem.SUCESSO));
54
55         return "forward:/preparaCadastroAtendimento.do";
56     }
57     @RequestMapping(value="/listarPorPaciente.do", method=RequestMethod.GET)
58     public @ResponseBody List<Consulta> listarPorPaciente(Integer idPaciente) {
59         return consultaRepository.listarPorPaciente(idPaciente);
60     }
61 }
```

A utilização de frameworks que adotam o uso de padrões de projetos e boas práticas tem influência direta na qualidade técnica e na produtividade da equipe de desenvolvimento de qualquer projeto. Como verificado nesta primeira parte do artigo, o Spring MVC é um dos produtos que seguem esta linha, e sua grande aceitação no mercado comprova isto, já que estamos tratando do framework baseado em ações mais popular do mundo, estando em cerca de 40% dos projetos Java web.

Nos tempos atuais, é uma questão prioritária deixar os sistemas com uma implementação envolvendo códigos pequenos e de fácil manutenção para que a evolução do produto seja feita com o custo mais baixo possível. Neste contexto, o Spring MVC larga na frente, podendo ser uma ótima alternativa para adoção em qualquer organização. Esta sensação certamente pode ser comprovada na implementação do *back-end* do sistema de clínica médica, que apresentou um código bastante enxuto.

## Autor



### Marcos Vinícius Turisco Dória

É Analista de Sistemas Java, certificado OCJA e OCJP, Graduado em Ciências da Computação pela Universidade de Fortaleza - UNIFOR, trabalha na Secretaria de Finanças de Fortaleza - SEFIN e desenvolve há cerca de cinco anos.



## Autor



### Pablo Bruno de Moura Nóbrega

<http://pablonobrega.wordpress.com>

Líder de Projeto Java, certificado OCJP e OCWCD, Graduado em Ciências da Computação pela Universidade de Fortaleza – UNIFOR, Mestre em Computação pela Universidade Estadual do Ceará – UECE, cursa MBA em Gerenciamento de Projetos na UNIFOR, trabalha na Secretaria Municipal de Finanças de Fortaleza e desenvolve sistemas há cerca de nove anos.



## Links:

### Página de download do Eclipse Luna.

<http://www.eclipse.org/downloads/>

### Página de download do WildFly.

<http://wildfly.org/downloads/>

### Página de download do PostgreSQL.

<http://www.enterprisedb.com/products-services-training/pgdownload>

## Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!





# DEVMEDIA

## 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

# Criando uma aplicação corporativa em Java – Parte 1

ESTE ARTIGO FAZ PARTE DE UM CURSO

**S**e olharmos para a plataforma e para a linguagem Java desde as primeiras versões, podemos notar uma enorme evolução. Com o passar dos anos foram surgindo diversas tecnologias, frameworks e especificações que fazem parte desse longo e contínuo processo evolutivo.

Nesse artigo iremos conhecer um pouco mais sobre três importantes especificações Java: JSF, JPA e CDI, e desenvolver uma aplicação que utilize implementações dessas especificações para vermos como funciona a integração entre elas. Nesse contexto iremos entender porque surgiram, quais vantagens trazem e quais tipos de problemas resolvem.

A aplicação que será desenvolvida ao longo do artigo serve para fazer o gerenciamento de uma biblioteca. Nela poderão ser gerenciadas, por exemplo, as informações das editoras, dos livros, dos leitores, dos funcionários da biblioteca e dos empréstimos dos livros.

## Especificações Java

Foram citadas três especificações Java no texto introdutório deste artigo, mas quem está começando a programar nesta linguagem pode não estar familiarizado com o termo “especificação”. Então, o que seria isso? Em meio ao processo evolucionário do Java, muitas vezes uma grande ideia acaba tendo mais de uma implementação. Foi justamente isso que aconteceu quando surgiu a proposta de se ter um framework ORM (*Object-Relational Mapping*). Relacionadas a essa proposta vieram implementações como Hibernate, EclipseLink, OpenJPA, entre outras. Para que exista uma padronização entre essas implementações é definida uma especificação, que nada mais é do que um documento que garante que todas elas tenham um comportamento igual ou bem semelhante e que contemplam as funcionalidades especificadas.

## Fique por dentro

Nesse artigo será desenvolvida do zero uma aplicação para gerenciamento de bibliotecas, utilizando JSF, JPA e CDI, mostrando como integrar essas três tecnologias em um projeto. Para isso, serão abordados recursos como o PhaseListener do JSF, as anotações `@Inject`, `@Produces`, `@Disposes` e `@Interceptor` relacionadas ao CDI, como criar e mapear entidades usando JPA, dentre outras coisas. Como servidor de aplicação, adotaremos o JBoss WildFly, destacando como configurá-lo. Além disso, adotaremos o Java 8 e alguns dos seus novos recursos, como a nova API para manipulação de datas.

No caso dos frameworks ORM foi criada a especificação JPA. Dessa forma, se a implementação que está sendo usada em algum projeto começa a apresentar bugs ou é descontinuada, podemos trocá-la por outra sem muito esforço.

Às vezes, no entanto, as implementações surgem antes da especificação. Assim, para manter a padronização, as implementações passam por mudanças para se adequar ao que foi determinado. Um bom exemplo é o Hibernate, que surgiu antes da especificação JPA.

Para quem tiver interesse em saber mais sobre o processo de criação das especificações, recomendamos consultar o site do Java Community Process (JCP).

## Sobre o JSF

O JSF pode ser definido como um framework MVC padrão para construir interfaces com o usuário baseadas em componentes para aplicações web. A especificação JSF dita como deve funcionar esse framework. Deste modo, cada implementação do JSF, como o Mojarra e o MyFaces, agrupa as funcionalidades do framework sempre indo ao encontro da especificação.

O JSF surgiu de uma necessidade de termos uma ferramenta mais simples, eficaz e produtiva no que se refere à criação de interfaces gráficas para aplicações web. Atualmente, inclusive, conseguimos criar soluções web com recursos visuais complexos de forma simples ao adotar, por exemplo, o PrimeFaces, uma das bibliotecas de componentes mais famosas para JSF.

O PrimeFaces possui centenas de componentes ricos que podem ser adicionados facilmente à aplicação, porém nesse artigo não vamos utilizá-los, pois iremos nos ater aos componentes padrões do JSF.

Além de o JSF facilitar muito a criação da interface gráfica, ele provê uma maneira padrão de solucionar problemas comuns que encontramos durante o desenvolvimento de aplicações web, a saber: validações, conversões, criação de templates, etc.

Antes de chegarmos à origem do JSF, vamos voltar um pouco mais no tempo e olhar para as tecnologias que o antecederam.

### API de Servlets

A API de Servlets contribuiu bastante para a popularização do Java, sendo amplamente adotada no desenvolvimento de sistemas web. Hoje os Servlets funcionam “por baixo dos panos” nos frameworks mais novos, inclusive no JSF, e é parte importante na infraestrutura das aplicações web. Anteriormente, no entanto, eram usadas diretamente para gerar dinamicamente as páginas de resposta às requisições dos usuários.

Como verificado com o tempo, essa forma de criar conteúdo dinâmico se mostrava propensa a erros e complicada, pois misturava código HTML com código Java. Veja um exemplo de código que poderia aparecer em uma Servlet: `out.println("<body text=\"black\" bgcolor=\"blue\" link=\"yellow\">");`. Confuso, não?

Para piorar a situação, nesse exemplo e em muitas situações ainda temos que usar o caractere de “escape” barra invertida (\) para “escaparmos” das ocorrências de aspas duplas(“).

### JavaServer Pages

O próximo passo na evolução das tecnologias Java web foi a introdução das JavaServer Pages. Essa tecnologia consistia em criar páginas com extensão .jsp que aceitavam tanto conteúdo HTML como porções de código Java (scriptlets, declarations e expressions).

Apesar da JavaServer Pages ter significado uma grande melhoria na forma de gerar conteúdo dinâmico para ser apresentado ao usuário, pois as páginas JSP eram praticamente páginas HTML com tags especiais para incluir código Java, ainda assim continuávamos com códigos HTML e Java misturados no mesmo arquivo.

Visando resolver esse problema surgiu a JSTL (*Java Standard Tag Library*) e a EL (*Expression Language*). Assim os códigos Java poderiam ser eliminados das páginas JSP.

Com a introdução da JSTL e da EL passou a ser possível implementar o MVC nas aplicações Java web e separar as responsabilidades de cada componente. Sendo assim, classes Java comuns representavam a camada Model, as páginas JSP representavam a camada View e os Servlets representavam a camada Controller.

No entanto, apesar de naquela época implementar o MVC dessa maneira ter se tornado uma prática comum porque deixava as aplicações mais organizadas, ainda era muito trabalhoso desenvolver usando JSP e Servlets. Eis que surgem os frameworks MVC.

### Frameworks MVC

Ao falar de frameworks MVC, podemos dar destaque ao Struts, que por um bom tempo foi muito utilizado. Uma das razões pela grande adoção do Struts pela comunidade Java foi que ele resolia de forma inteligente o problema relacionado a separar a aplicação em camadas. Antes do Struts, com JSP e Servlets sem um framework, muitos desenvolvedores, principalmente os inexperientes, acabavam criando o mau hábito de colocar código Java na camada de visualização. Já com o Struts, devido a sua arquitetura MVC, os desenvolvedores eram impelidos a separar o código de apresentação (View) do restante do código (Controller e Model).

Apesar da boa arquitetura desta até então excelente opção, ainda era muito trabalhoso e custoso construir as interfaces gráficas. Então foi criado o JSF, para que se tivesse um framework que também fosse MVC, mas que facilitasse o trabalho de criação da camada de apresentação. Na época que o JSF surgiu o Struts era o framework mais utilizado, mas o JSF aos poucos foi ganhando seu espaço.

O profissional que está acostumado com outros frameworks como Struts ou Spring MVC, por exemplo, pode estranhar um pouco a forma como o JSF funciona, pois estes outros frameworks são considerados Action Based, enquanto o JSF é considerado Component Based. No entanto, não se preocupe com isso, pois na parte prática do artigo será apresentado o “jeitão” JSF de ser.

### Sobre a JPA

A especificação Java Persistence API, ou simplesmente JPA, foi criada para padronizar a forma como os frameworks ORM devem trabalhar. Essa especificação define uma série de anotações e interfaces, e os frameworks ORM devem implementar essas interfaces para que todos eles possam ser utilizados a partir de uma única API. A especificação também dita o que é esperado que cada método faça, e a implementação fica a cargo de cada provedor JPA – vide BOX 1.

#### BOX 1. Provedor JPA

Provedor JPA e persistence provider são termos usados para referenciar as implementações da especificação JPA, como Hibernate, EclipseLink, OpenJPA, dentre outras.

A seguir conheceremos alguns dos problemas que motivaram a criação dos frameworks de mapeamento objeto-relacional, que ao apresentarem as soluções para tais problemas facilitaram a vida dos programadores, elevando inclusive a produtividade durante o desenvolvimento.

Atualmente, a grande maioria das aplicações usa algum banco de dados relacional. No entanto, como veremos temos um pequeno problema nesse ponto. Quando falamos de bancos de dados relacionais, logo pensamos em tabelas, relacionamentos, chaves primárias, chaves estrangeiras, etc. Já quando o assunto é linguagem orientada a objetos, logo pensamos em classes, composição, herança, polimorfismo, etc. Como conciliar essas diferenças?

Ao utilizarmos JDBC para converter o conteúdo de um **ResultSet** em objetos, conseguimos notar as similaridades entre uma classe e uma tabela, mas apesar das semelhanças existem também diversas diferenças entre o modelo OOP e o relacional. Muitas vezes alguma coisa que existe no paradigma relacional não existe no orientado a objetos, e vice versa, o que remete a certa complexidade para fazer os dois modelos conversarem, exigindo que o desenvolver contorne essas diferenças programaticamente, fazendo para isso diversos ajustes no código. A essas diferenças fundamentais entre os dois paradigmas são dadas o nome de Impedância (*impedance mismatch*).

Em Java, uma das formas de se trabalhar com bancos de dados é utilizar a API JDBC. Entretanto, um dos problemas ao utilizarmos JDBC é que precisamos escrever código SQL misturado com nosso código Java. Assim, se for preciso alterar alguma instrução SQL, será necessário recompilar a classe na qual foi feita a alteração. Diante disso, alguns programadores adotaram estratégias como remover os códigos SQL da aplicação, movendo-os para um arquivo TXT ou XML. Nesse momento até surgiram frameworks, como o iBatis, para ajudar a realizar esse isolamento.

Outro problema que podemos enfrentar ao utilizar JDBC e escrever os SQLs diretamente em classes Java é verificado quando precisamos mudar de banco de dados. Apesar de existir o padrão ANSI, instruções SQL podem apresentar diferenças significativas dependendo do SGBD utilizado, o que dificulta a tarefa de migrar de um banco para outro.

Se olharmos para as dificuldades e problemas descritos, podemos concluir que faria sentido se trabalhássemos apenas com a nossa linguagem de programação, usando somente o paradigma OOP, e se fosse abstruída a responsabilidade de escrever SQL ou fazer conversões entre o modelo orientado a objetos e o relacional. Foi justamente para isso que foram criados os frameworks ORM, como o Hibernate, EclipseLink, Apache OpenJPA, dentre outros. A sigla ORM significa Object-Relational Mapping, ou

Mapeamento Objeto-Relacional em português, e como o nome sugere essa tecnologia viabiliza um mapeamento entre um modelo e outro, para que possa haver uma conversão automática entre os modelos OOP e relacional. O objetivo é que utilizando esse tipo de framework não precisemos mais nos preocupar com a escrita de instruções SQL, passando a trabalhar somente com objetos.

## Sobre o CDI

A especificação CDI, cujo nome completo é Context and Dependency Injection for Java EE, padroniza o processo de injeção de dependências (em inglês *Dependency Injection*, ou simplesmente DI) dentro do Java. Antes do CDI, existiam vários frameworks que realizavam injeção de dependências, como o Spring, JBoss Seam, Google Guice, PicoContainer, dentre outros, mas cada um deles funcionava de um jeito diferente e não existia uma forma padrão de fazer a injeção das dependências. Essa especificação, inclusive, sofreu várias influências dos frameworks que já existiam, principalmente do JBoss Seam 2 e do Google Guice.

Uma vantagem no uso do CDI, se comparado ao uso do JBoss Seam 2, é que a injeção das dependências é feita de modo type safe, e no caso do Seam 2 é feita baseada em Strings. Isso significa que utilizando CDI os beans a serem injetados são selecionados com base no tipo que é requerido no injection point e não com base no nome do bean, o que torna o processo de injeção mais seguro. Injection Point é um ponto ou lugar, geralmente dentro de uma classe, no qual a dependência pode ser injetada.

A especificação CDI (JSR 299) se relaciona com a especificação Dependency Injection for Java (JSR 330) e uma complementa a outra. Essa relação se dá pelo fato de utilizarmos as anotações `@Inject`, `@Named`, `@Qualifier`, `@Scope` e `@Singleton` – definidas na JSR 330 – no processo de injeção de dependências. Em outras palavras, ao utilizarmos CDI acabamos fazendo uso dessas anotações.

Dependency Injection é uma das formas de se aplicar uma técnica muito mais ampla chamada Inversão de Controle (em inglês *Inversion of Control*, ou simplesmente IoC). A ideia principal da IoC é inverter o controle das chamadas dos métodos da aplicação em relação à programação convencional, ou seja, as chamadas não são



determinadas diretamente pelo programador. Alguns frameworks usam o conceito de inversão de controle e controlam as chamadas de métodos específicos, que no caso do Java geralmente são marcados com alguma anotação. Quando usamos DI também ocorre inversão de controle, mas de uma forma um pouco diferente da mencionada anteriormente, pois nesse caso invertemos o controle da criação das dependências, que passa a não ser feita mais por nós, ou seja, já recebemos as dependências instanciadas e prontas para serem usadas.

A principal vantagem de se adotar a injeção de dependências é que esse design pattern diminui o acoplamento entre as classes e suas dependências. Felizmente ao utilizar CDI acabamos empregando esse padrão de projeto sem mesmo perceber, de forma transparente, e podemos focar nas regras de negócio da aplicação que estamos desenvolvendo, deixando de lado a preocupação de como as dependências serão satisfeitas.

Para que o leitor entenda a importância de se diminuir o acoplamento entre as classes (e a implementação do CDI que escolhermos irá nos ajudar nisso) é preciso entender o significado de acoplamento. Então vamos revisar dois conceitos importantes na orientação a objetos, que são o acoplamento e a coesão.

A coesão está relacionada ao que uma classe sabe fazer. Se a classe foi criada de forma que tenha um único e bem focado propósito, dizemos que ela é coesa. No entanto, se a classe sabe fazer muitas coisas, dizemos que ela é pouco coesa. A alta coesão é algo desejável, enquanto a baixa coesão é algo que devemos evitar.

O acoplamento, por sua vez, está relacionado com a forma como uma classe se relaciona e interage com as outras. Vejamos alguns exemplos para ajudar no entendimento. Se a classe A tem um método que recebe um objeto do tipo da classe B, podemos dizer que esse acoplamento entre as duas classes é menor do que se a classe A tivesse como membro uma variável de instância do tipo da classe B. Teríamos um acoplamento ainda maior que nos dois casos anteriores se a classe A herdasse da classe B. Outra situação seria se a classe A conhecesse a classe B unicamente por meio do que a classe B expõe através de sua interface. Deste modo podemos dizer que o acoplamento entre as duas classes é fraco. Mas se a classe A conhece detalhes de implementação da classe B e depende de partes da classe B que não fazem parte do que foi exposto em sua interface, então podemos dizer que o acoplamento entre elas é mais forte do que deveria.

O interessante é que a coesão e o acoplamento se inter-relacionam de certa forma. À medida que nos esforçamos para termos classes com alta coesão em nossa aplicação, deixamos de ter classes que sabem fazer muitas coisas, e consequentemente teremos mais relacionamentos entre nossas classes especializadas, o que propicia o aumento do acoplamento. Porém, se nos esforçarmos em excesso para diminuirmos o acoplamento, consequentemente voltaremos a ter classes que sabem fazer muitas coisas, ou seja, teremos menos classes se relacionando, mas classes pouco coesas. Levando em consideração o fato de que a coesão afeta o acoplamento e vice-versa, qual estratégia podemos adotar para atingirmos o cenário ideal que consiste em manter a alta coesão e o baixo acoplamen-

to? Uma possível resposta a essa pergunta seria projetar classes altamente coesas e diminuir o acoplamento entre elas através da injeção de dependências.

Suponhamos que estamos desenvolvendo nossa aplicação para gerenciamento de bibliotecas e criamos uma classe chamada **AutorBean** que tem as seguintes linhas de código:

```
EntityManager entityManager = ...código para conseguir um entityManager aqui  
AutorDao autorDao = new AutorDaoJpa(entityManager);
```

Note que a classe **AutorBean** depende de um **AutorDao**, ou seja, o **AutorDao** é sua dependência. Porém a classe **AutorBean** está resolvendo sua dependência e escolhendo uma implementação para **AutorDao**, que seria a classe **AutorDaoJpa**. Observe também que a classe **AutorDaoJpa** tem uma dependência que seria um **entityManager**, que também está sendo resolvida manualmente. Se já é ruim para a classe **AutorBean** gerenciar suas próprias dependências, imagine nesse caso que ela está gerenciando uma dependência de sua dependência.

Um problema que pode surgir é se alguém alterar a assinatura do construtor da classe **AutorDaoJpa**, modificando a lista de argumentos que ele recebe. Nesse caso a classe **AutorBean** irá parar de compilar.

Outro agravante que podemos ter com esse código é se um dia, por algum motivo, quisermos trocar a implementação de **AutorDao** e decidirmos usar um **AutorDaoJdbc** no lugar do **AutorDaoJpa**. Se as dependências são gerenciadas pela nossa aplicação, como nesse exemplo, teremos que alterar o código inteiro manualmente, procurando todo lugar onde é usado um **AutorDaoJpa** e substituir por um **AutorDaoJdbc**.

Os problemas que observamos são decorrentes do forte acoplamento entre nossa classe de exemplo e sua dependência, pois a classe conhece a implementação concreta da dependência e ainda precisa instanciá-la. Tudo poderia ficar ainda pior se crescesse o número de dependências. Se pensarmos bem, a responsabilidade de instanciar as dependências nem deveria recair sobre o programador, pois não fazem parte da lógica da aplicação.

Para solucionarmos esses problemas podemos utilizar Dependency Injection, e é ai que o CDI entra na jogada. Se escrevéssemos um código equivalente ao apresentado, mas utilizando CDI, poderíamos injetar um **AutorDao** na classe **AutorBean** da seguinte maneira:

```
@Inject  
AutorDao autorDao;
```

Na classe **AutorDaoJpa** poderíamos injetar o **entityManager** direto no construtor, conforme o código:

```
@Inject  
AutorDaoJpa(EntityManager entityManager){  
    ...  
}
```

## Nota

Para que a injeção desse entityManager funcione é preciso configurar uma classe produtora de entityManagers. Veremos mais sobre isso na hora de construir nossa aplicação para bibliotecas.

Ao criarmos nossas classes utilizando DI é comum configurarmos a injeção das dependências pelo construtor, como no trecho de código da classe **AutorDaoJpa** mostrado anteriormente. Com isso, a classe apenas espera que alguém injete as dependências que ela precisa. Isso facilita muito na hora de criarmos os testes de unidade, pois podemos injetar manualmente mock objects (ver **BOX 2**) pelo construtor da classe utilizando bibliotecas como o Mockito. Então, em suma, esse tipo de estratégia aumenta a testabilidade de nossas classes.

## BOX 2. Mock Objects

Mock Objects são objetos falsos que simulam o comportamento de objetos reais de forma controlada. É possível “ensinar” um mock object a se comportar da forma que precisamos. Normalmente eles são criados para testar o comportamento de outros objetos.

Ao utilizarmos CDI passamos a tirar proveito das duas principais vantagens atreladas à injeção de dependências: diminuir o acoplamento entre as classes e aumentar a testabilidade delas. Além destas, existem outras vantagens no uso de DI, mas não vamos entrar nesse mérito.

## Desenvolvendo a aplicação

Ao desenvolvermos uma aplicação é comum que acabemos criando grupos de arquivos muito similares entre si (os cadastros e as classes que fazem parte do DAO são exemplos disso). Por esse motivo e visando explicar detalhadamente cada código apresentado, não iremos exibir o código fonte de todos os arquivos.

Sendo assim, ao longo do artigo iremos analisar apenas os códigos mais importantes dos principais arquivos, para que o leitor possa construir seu projeto e no final ter um sistema de biblioteca que execute as duas principais operações: o empréstimo e a devolução de livros. Uma versão reduzida da aplicação igual à apresentada no artigo está disponível para download no GitHub. No entanto, uma versão completa, com todos os arquivos na íntegra, também está disponível para download no GitHub (os endereços de ambas as versões constam na seção **Links**).

A versão completa contém todos os arquivos relacionados aos cadastros, todas as classes pertencentes ao DAO, além de conter também a funcionalidade de renovação do empréstimo. Um ótimo desafio que lançamos ao leitor, com o objetivo de elevar seu nível técnico, é que desenvolva a aplicação se baseando no artigo e depois tente implementar sozinho o que só tem na versão completa presente no GitHub. Deste modo, caso sinta alguma dúvida neste processo, você pode verificar como tal recurso foi implementado.

## Configurando o ambiente de desenvolvimento

Para que o leitor possa criar o projeto proposto no artigo e executá-lo, é necessário primeiro configurar o ambiente de desenvolvimento. Sendo assim, a seguir serão mostrados os detalhes de cada uma das etapas da configuração do ambiente.

## Downloads e instalações

Para desenvolvermos nossa aplicação iremos usar a versão 8 do Java. Portanto, é necessário baixar e instalar o JDK 8. Como IDE, adotaremos o Eclipse Luna para Java EE. O servidor de aplicação escolhido foi o JBoss WildFly 8.1.0.Final. Já a implementação da especificação JSF que adotaremos é a Mojarra. Note que não será necessário fazer o download desta, pois ela já vem com o WildFly. A implementação da especificação JPA que usaremos é o Hibernate, e também já vem com o WildFly. Mantendo esse padrão, a implementação da especificação CDI que usaremos é o Weld. Por fim, como banco de dados, faremos uso do MySQL. Além disso, é necessário fazer o download do driver do MySQL. Veja o endereço para download dessas ferramentas na seção **Links**.

Após baixar e instalar os softwares mencionados, prossiga para o próximo passo, que é a configuração do JBoss.

## Configurando o JBoss WildFly

O WildFly possui uma arquitetura baseada em módulos, o que o torna mais flexível, pois o carregamento dos módulos é feito sob demanda e portanto serão carregados somente os módulos requeridos pelas aplicações. O fato de nem todos os módulos precisarem ser carregados resulta em menor tempo para inicialização do servidor de aplicação. Como a arquitetura é modular, primeiramente iremos adicionar um módulo no JBoss relacionado ao driver do MySQL, para viabilizar a conexão de nossa aplicação com o banco de dados. Sendo assim, na pasta *wildfly-8.1.0.Final\modules\*, crie a seguinte estrutura de pastas: *com\mysql\main*.

Note que dentro da estrutura de pastas criada, temos uma pasta chamada *main*, o que é obrigatório, pois se não existir um diretório com esse nome o WildFly não carrega o módulo. O diretório *com\mysql\main* deverá conter o driver do MySQL (por exemplo: *mysql-connector-java-5.1.32-bin.jar*) e um arquivo chamado *module.xml* que contém a definição do módulo (veja o código relacionado na **Listagem 1**).

**Listagem 1.** Conteúdo do arquivo *module.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.32-bin.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
  </dependencies>
</module>
```

Nesse arquivo, note que no atributo **name** da tag **module** é informado um nome que deve ser usado quando quisermos nos referir a esse módulo. Observe também que no atributo **path** da tag **resource-root** deve ser informado o nome do arquivo JAR do MySQL (o driver) que está no mesmo diretório.

O arquivo *wildfly-8.1.0.Final\standalone\configuration\standalone.xml* é um dos principais arquivos de configuração do JBoss e contém configurações relacionadas a drivers, datasources, portas, dentre outras. Dentro desse arquivo, entre as tags **<drivers>** e **</drivers>**, criaremos um elemento **driver** e o vincularemos ao módulo definido anteriormente (**Listagem 1**) configurando o atributo **module** desse elemento. O atributo **name** da tag **driver** é preenchido com um nome que será referenciado quando criarmos o datasource. As alterações feitas no arquivo *standalone.xml* são exibidas na **Listagem 2**.

**Listagem 2.** Trecho do arquivo *standalone.xml* com a configuração do driver MySQL.

```
<drivers>
  <driver name="com.mysql" module="com.mysql">
    <xa-datasource-class>com.mysql.jdbc.jdbc2.optional.MysqlXADatasource
    </xa-datasource-class>
  </driver>
</drivers>
```

O próximo passo é criarmos um datasource para podermos conectar na base de dados através dele. Sendo assim, no mesmo arquivo *wildfly-8.1.0.Final\standalone\configuration\standalone.xml*, entre as tags **<datasources>** e **</datasources>**, configure o datasource conforme a **Listagem 3**.

### Criando o banco de dados

Após o MySQL ter sido instalado poderemos utilizar o MySQL Command-Line Tool, um programa que funciona no prompt de comando do Windows e serve para estabelecer uma conexão com o servidor MySQL. Deste modo, a partir de uma janela do prompt de comando, conecte ao servidor MySQL com o comando: **mysql -u usuário -psenha**. Caso seu servidor não seja local você terá que adicionar o parâmetro **-h** ao comando informando o host do mesmo.

**Listagem 3.** Trecho do arquivo *standalone.xml* com a configuração do DataSource.

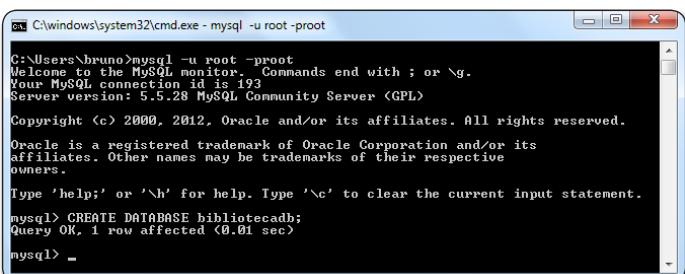
```
<datasources>
  <datasource jta="false" jndi-name="java:/bibliotecaDS"
    pool-name="bibliotecaPool" enabled="true" use-java-context="true">
    <connection-url>jdbc:mysql://localhost:3306/bibliotecaDb
    </connection-url>
    <driver>com.mysql</driver>
    <pool>
      <min-pool-size>10</min-pool-size>
      <max-pool-size>100</max-pool-size>
      <prefill>true</prefill>
    </pool>
    <security>
      <user-name>root</user-name>
      <password>root</password>
    </security>
  </datasource>
</datasources>
```

Podemos substituir a palavra **usuario** por **root**, que é o usuário com mais privilégios no MySQL, e substituir a palavra **senha** pela senha do root que foi definida na instalação do MySQL.

Depois de conectar no servidor, criamos o banco de dados que iremos utilizar em nosso projeto através da seguinte instrução SQL:

**CREATE DATABASE bibliotecadb;**

A **Figura 1** mostra a conexão sendo estabelecida e o banco de dados sendo criado.



The screenshot shows a command-line window titled 'C:\Windows\system32\cmd.exe - mysql -u root -proot'. The MySQL monitor is displayed, showing the connection information: 'Welcome to the MySQL monitor. Commands end with ; or \g.' and 'Your MySQL connection id is 193'. It also displays the server version: 'Server version: 5.5.28 MySQL Community Server (GPL)' and the copyright notice: 'Copyright (c) 2000, 2012, Oracle and/or its affiliates. All rights reserved.' Below this, it says 'Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.' At the bottom, it shows the MySQL prompt: 'Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.' A command is being typed: 'mysql> CREATE DATABASE bibliotecadb;'. The response shows 'Query OK, 1 row affected (<0.01 sec>)'. The MySQL prompt is shown again: 'mysql> \_'.

**Figura 1.** Banco de dados sendo criado através do MySQL Command-Line Tool



## Instalando o JBossAS Tools no Eclipse

Por padrão, ao tentarmos adicionar um novo servidor no Eclipse Luna, a opção WildFly 8.x não aparece. Para que isso aconteça, temos que fazer a instalação da extensão JBossAS Tools.

Para iniciarmos a instalação podemos clicar com o botão direito em qualquer lugar vazio na aba *Servers* e, no menu suspenso que se abre, apontar o mouse sobre a opção *New* e clicar em *Server*. Feito isso, será exibida uma caixa de diálogo onde temos que clicar na opção *Download additional server adapters*. Em seguida, devemos procurar por *JBossAS Tools* (veja a Figura 2) e logo após prosseguir com a instalação e aceitar os termos da licença. No final do procedimento o Eclipse será reiniciado.

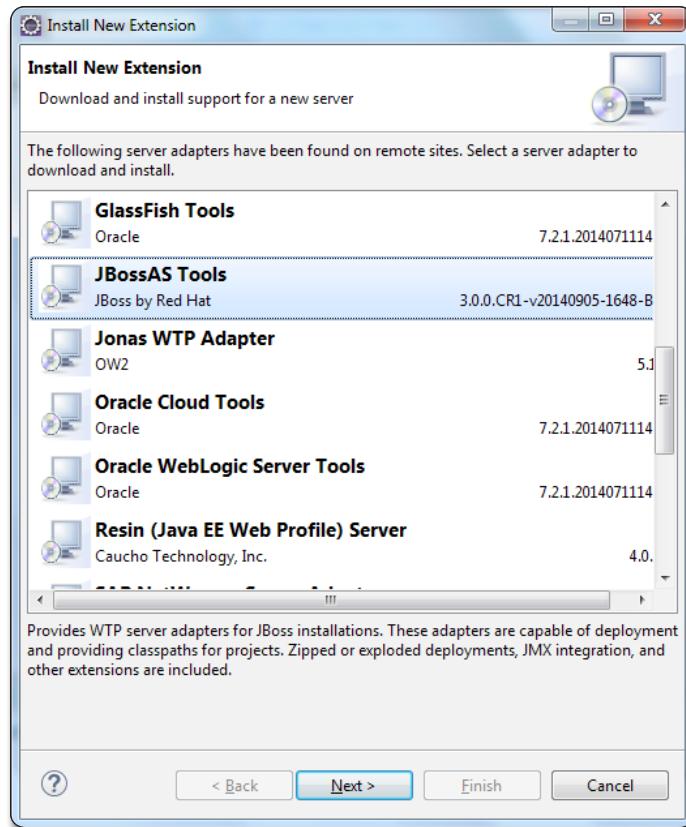


Figura 2. Instalação do JBossAS Tools

## Adicionando o JBoss WildFly no Eclipse

Após a instalação do JBossAS Tools conseguiremos adicionar o WildFly no Eclipse clicando com o botão direito em qualquer lugar vazio na aba *Servers* e, no menu suspenso que se abre, apontar o mouse sobre a opção *New* e clicar em *Server*. Dessa vez a opção *WildFly 8.x* será exibida, sendo ela a que adotaremos (veja a Figura 3).

Após selecionarmos a opção *WildFly 8.x* e clicarmos no botão *Next*, devemos seguir avançando pelas telas do wizard até a etapa em que será solicitado que você informe o *Home Directory* do JBoss WildFly 8.x. Como já o baixamos diretamente do site do WildFly, precisamos apenas informar o caminho onde ele

está (veja a Figura 4). O resto das opções pode ser mantido com as configurações padrão. Deste modo, prossiga até o término da instalação.

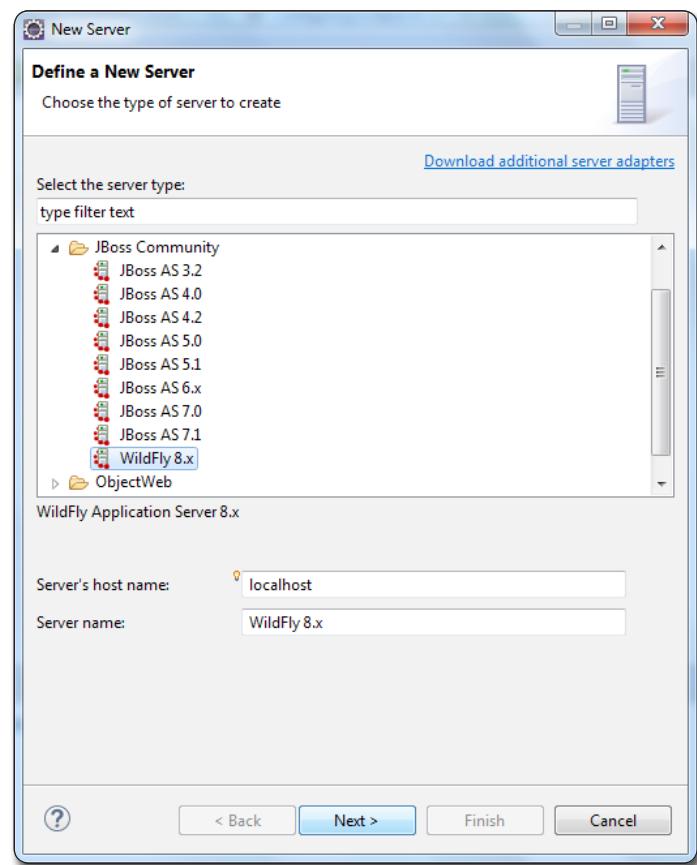


Figura 3. Adicionando o WildFly 8.x no Eclipse

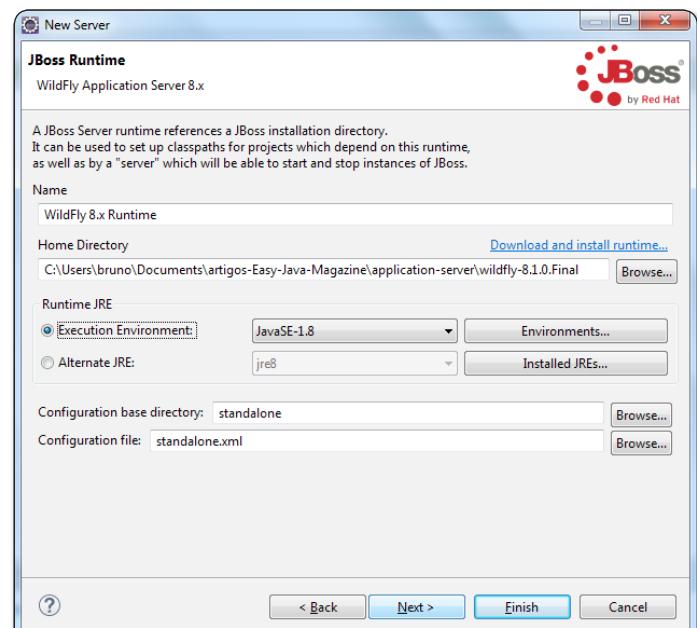


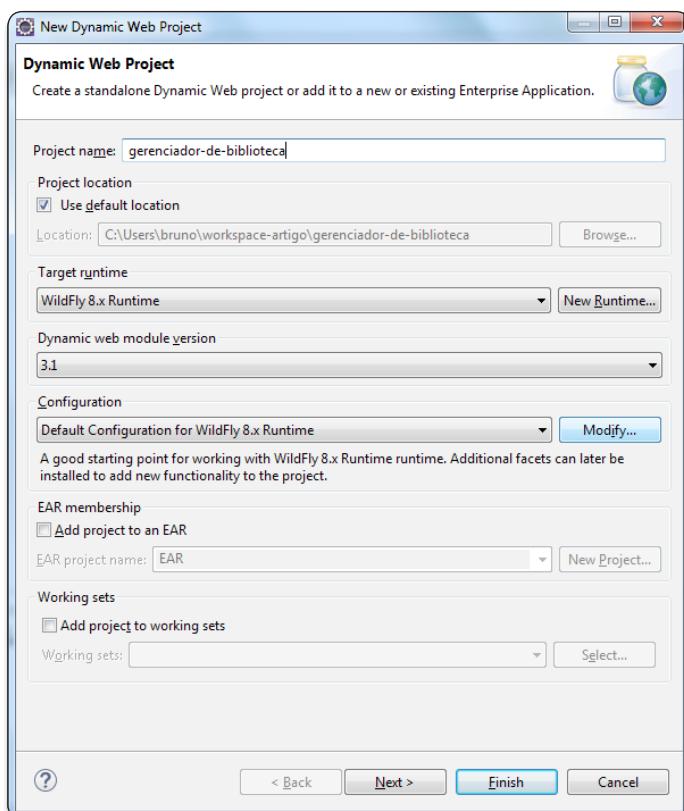
Figura 4. Informando o caminho do JBoss WildFly (seu Home Directory)

## Criando o projeto no Eclipse

Agora chegou o momento de criarmos dentro do Eclipse o projeto da aplicação para gerenciamento de bibliotecas.

Por ser uma aplicação Web, criaremos um novo projeto do tipo *Dynamic Web Project*. Para isso, podemos clicar no menu *File* e em seguida *New*, e então escolher a opção *Dynamic Web Project*. Feito isso, será exibida a primeira tela do wizard, onde poderemos configurar o projeto que está sendo criado. Devemos nomeá-lo como “gerenciador-de-biblioteca” e configurá-lo conforme a **Figura 5**.

Ainda na primeira tela do wizard devemos clicar no botão *Mostrar...* na seção *Configuration* para podermos configurar alguns aspectos relevantes ao projeto, como por exemplo, a versão do Java e definirmos que será usado JSF. Feito isso será aberta a tela *Project Facets*, onde devemos manter três opções marcadas e configurar suas respectivas versões (vide **Figura 6**).



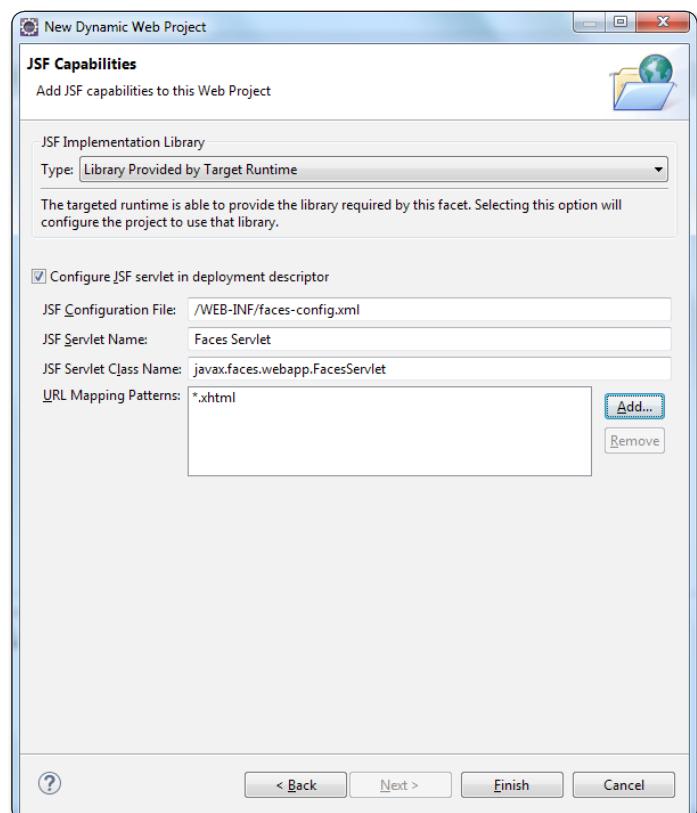
**Figura 5.** Configuração do projeto

Project Facet	Version
Axis2 Web Services	
CXF 2.x Web Services	1.0
<input checked="" type="checkbox"/> Dynamic Web Module	3.1
<input checked="" type="checkbox"/> Java	1.8
JavaScript	1.0
<input checked="" type="checkbox"/> JavaServer Faces	2.2
JAX-RS (REST Web Services)	1.1
<input checked="" type="checkbox"/> JAXB	2.2
<input checked="" type="checkbox"/> JPA	2.1
WebDoclet (XDoclet)	1.2.3

**Figura 6.** Opções selecionadas na tela Project Facets

Na etapa *Web Module* do wizard, marque a opção *Generate web.xml deployment descriptor* para que seja gerado automaticamente o arquivo *web.xml* na pasta *WEB-INF* do projeto.

Em seguida, na etapa *JSF Capabilities*, a configuração deve ficar igual à configuração da **Figura 7**.



**Figura 7.** Configurações do JSF no projeto



Note que na opção *JSF Implementation Library* informamos que os jars que contêm a implementação do JSF serão fornecidos pelo servidor de aplicação. Note também que na opção *URL Mapping Patterns* removemos o mapeamento */faces/\** e inserimos o mapeamento *\*.xhtml*, pois nossas páginas JSF terão extensão XHTML. Depois de configurada essa etapa podemos finalizar, clicando em *Finish*.

## Configurando o persistence.xml

O *persistence.xml* é o arquivo de configuração padrão para JPA. Dentro dele podemos definir um ou mais persistence units, onde cada um reúne um agrupamento lógico que inclui:

- As configurações para uma entity manager factory e para seus entity managers;
- O conjunto das entidades gerenciadas pelos entity managers;
- O mapeamento objeto-relacional para as entidades.

Dito isso, na pasta *src* de nosso projeto devemos criar uma pasta chamada *META-INF* e dentro dela criar o arquivo *persistence.xml*, que deverá ter o conteúdo apresentado na **Listagem 4**.

Dentro do arquivo *persistence.xml* usamos a tag *<persistence-unit>* para definirmos o único persistence unit do nosso projeto. A tag *<provider>* serve para indicar qual o persistence provider que iremos adotar, nesse caso o Hibernate. Na sequência temos a tag *<jta-data-source>*, que define o datasource a ser utilizado pelo persistence provider. Em seguida temos algumas tags *<class>* que declaram as entidades. Essas entidades ainda não foram criadas, no entanto, não se preocupe com isso, pois criá-las será exatamente o nosso próximo passo. Ao final do arquivo temos a tag *<properties>*, que pode conter tanto configurações de propriedades padrão como propriedades específicas do persistence provider em uso.

Nesse momento devemos entender a importância da propriedade *hibernate.hbm2ddl.auto*, também presente no *persistence.xml*, pois

dependendo de como a configuremos podem ocorrer alterações indesejáveis no banco de dados. Por esse motivo é recomendado dar uma atenção especial a ela, principalmente quando a aplicação for para produção. Essa propriedade aceita os seguintes valores:

### Listagem 4. Conteúdo do arquivo *persistence.xml*.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="bibliotecaPersistence"
    transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/bibliotecaDS</jta-data-source>

    <class>br.com.javamagazine.entidades.Autor</class>
    <class>br.com.javamagazine.entidades.Categoria</class>
    <class>br.com.javamagazine.entidades.Editora</class>
    <class>br.com.javamagazine.entidades.Emprestimo</class>
    <class>br.com.javamagazine.entidades.Endereco</class>
    <class>br.com.javamagazine.entidades.FuncioarioBiblioteca</class>
    <class>br.com.javamagazine.entidades.Leitor</class>
    <class>br.com.javamagazine.entidades.Livro</class>
    <class>br.com.javamagazine.entidades.Pessoa</class>
    <class>br.com.javamagazine.entidades.Telefone</class>
    <class>br.com.javamagazine.entidades.Usuario</class>
    <class>br.com.javamagazine.conversores.LocalDateConversor</class>

    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLInnoDBDialect" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="false" />
      <property name="hibernate.hbm2ddl.auto" value="create" />
    </properties>
  </persistence-unit>
</persistence>
```



- **create:** cria as tabelas necessárias para persistirmos nossas entidades, destruindo qualquer dado existente;
- **create-drop:** cria as tabelas de forma igual à opção create, porém remove as tabelas no final da sessão;
- **update:** faz com que as alterações feitas nas entidades sejam refletidas nas tabelas do banco. Por exemplo: ao adicionar um novo atributo em uma entidade, será criada a respectiva coluna na tabela;
- **validate:** valida se as tabelas da base de dados estão de acordo com nossas entidades, porém não efetua nenhuma alteração no banco. Pelo fato dessa opção não efetuar alterações na base de dados, podemos considerá-la a mais segura para ser usada em ambiente de produção.

A estrutura do projeto nesse momento deve estar semelhante à estrutura exposta na **Figura 8**.

## Criando e mapeando as entidades

Dentro da JPA existe o conceito de entidade, que nada mais é do que um POJO (*Plain Old Java Object*) cujos atributos que não são marcados com a anotação `@Transient` devem ser persistidos em um banco de dados relacional através de um **EntityManager**. Em outras palavras, uma entidade representa os dados armazenados em um banco de dados relacional. Dessa forma, trabalhamos na grande parte do tempo apenas com nossos objetos e não com SQL, e as mudanças feitas nesses objetos irão refletir no banco de dados. Com isso, a JPA provê um mecanismo simples de manipularmos os dados que estão no banco.

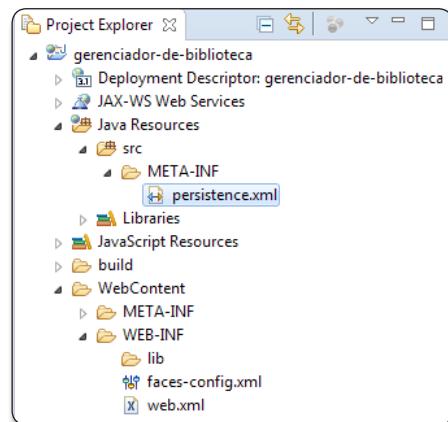
Para que possamos trabalhar com as entidades da forma descrita no parágrafo anterior, devemos primeiramente indicar quais de nossas classes são entidades e fazer o mapeamento objeto/relacional. Assim, o JPA provider usado será capaz de fazer as conversões entre o mundo OOP e o mundo relacional. Para isso, podemos adotar anotações ou configurações em arquivos XML (`orm.xml` e `persistence.xml`). Em nosso estudo de caso faremos uso de anotações.

A **Figura 9** demonstra um diagrama que representa como deverá ficar nosso banco de dados depois de pronto.

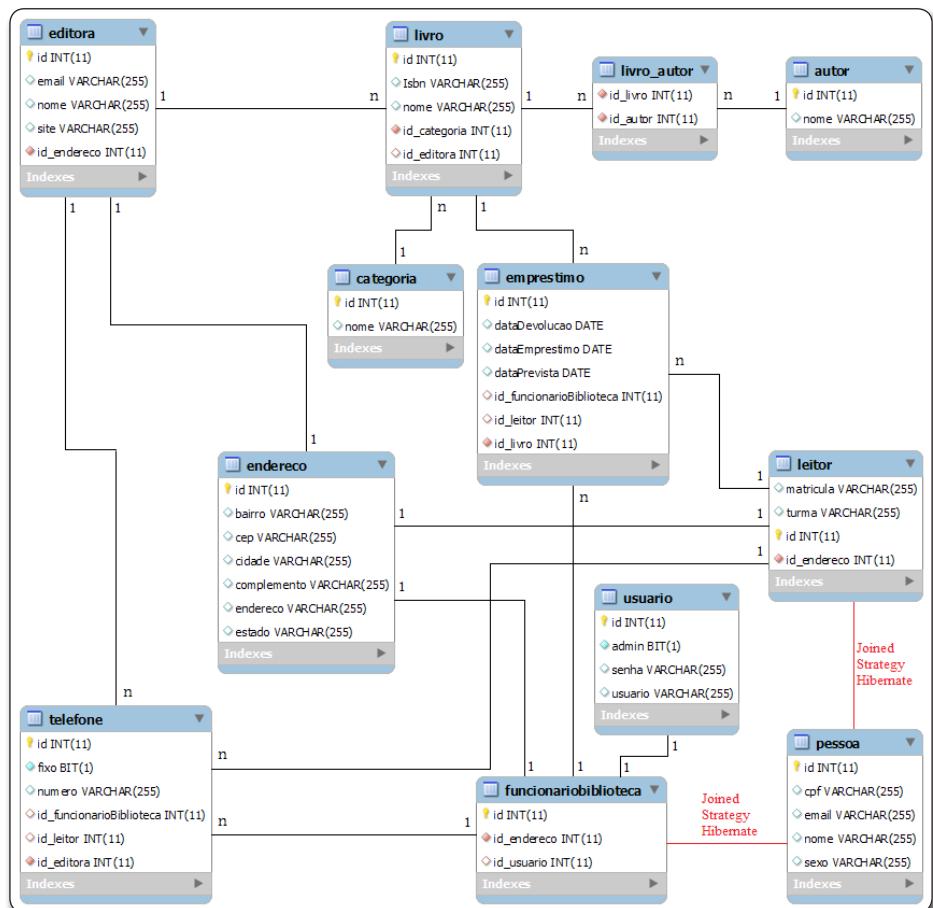
Olhando para um diagrama como esse fica mais fácil criarmos e mapearmos nossas entidades, pois visualizar as tabelas (assim como suas colunas, chaves primárias e estrangeiras), os relacionamentos entre as tabelas e a cardinalidade de cada relacionamento, nos possibilita ter uma visão geral do banco e, portanto, sabermos

quais anotações devemos usar nas entidades para alcançar esse resultado final.

Veja que os relacionamentos entre as tabelas ocorrem de forma convencional, ou seja, por meio de chaves primárias (*primary keys*) e estrangeiras (*foreign keys*). No entanto, no caso das tabelas *pessoa*, *leitor* e *funcionarioBiblioteca*, o relacionamento entre elas é feito pelo Hibernate através das *primary keys*, pois nesse caso o



**Figura 8.** Estrutura do projeto



**Figura 9.** Diagrama de como deve ficar nosso banco de dados pronto

mapeamento será feito através da anotação `@Inheritance` (com a estratégia `JOINED`), que deve ser usada quando existe uma relação de herança entre entidades. Veremos mais detalhes sobre isso adiante.

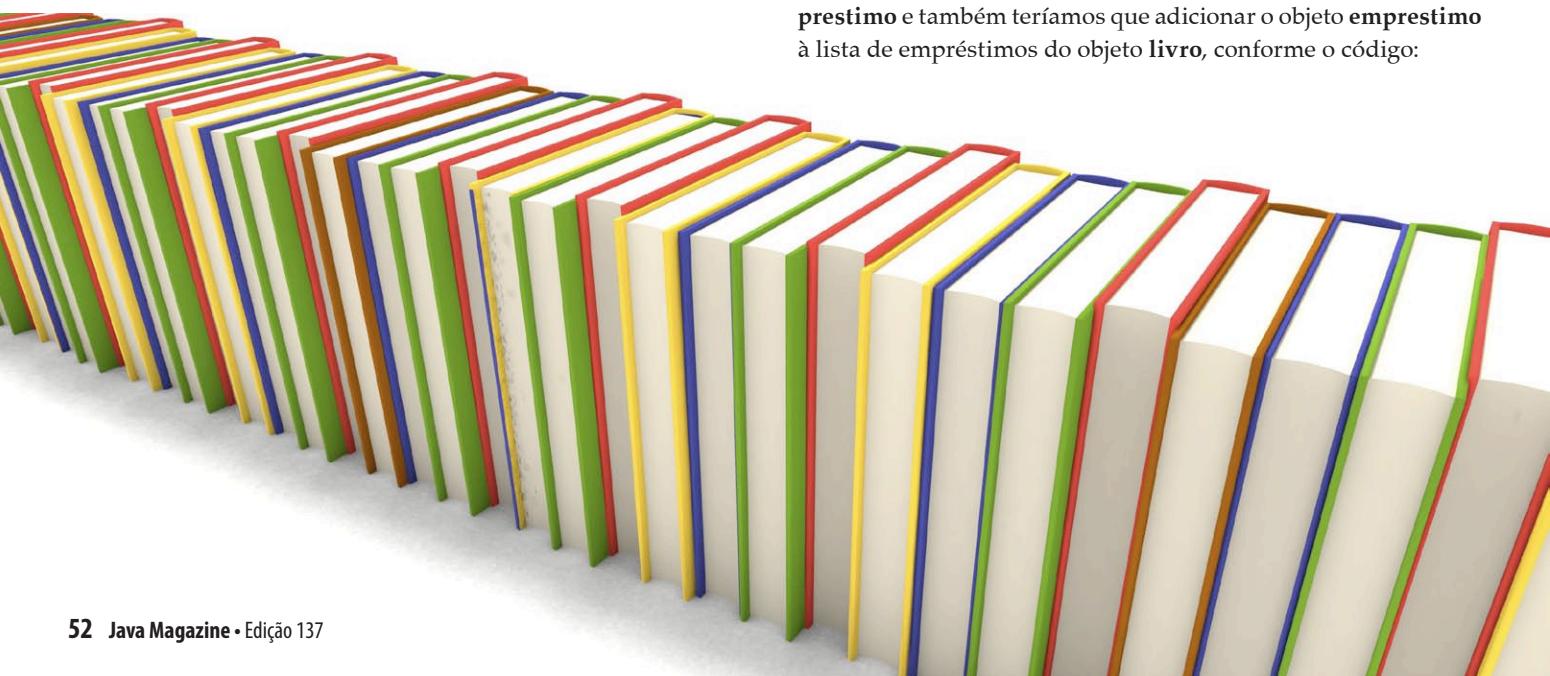
Antes de criarmos nossas entidades é importante entender quais tipos de relacionamentos podem existir entre elas, sendo que esses relacionamentos podem ser unidirecionais ou bidirecionais. Existem, ao todo, sete tipos de relacionamentos, então vejamos um exemplo de cada um para facilitar o entendimento:

- **OneToOne unidirecional:** tipo de relacionamento empregado entre as entidades `Editora` e `Endereco`. Em nossa aplicação é certo que podemos precisar recuperar o endereço de uma editora (`editora.getEndereco()`), mas não precisaremos recuperar a editora de um endereço (`endereco.getEditora()`);
- **OneToOne bidirecional:** adotamos esse tipo de relacionamento entre as entidades `Usuario` (que representa a combinação de usuário e senha usados para logar no sistema) e `FuncionarioBiblioteca`. Em nossa aplicação, a partir do usuário que foi usado para logar no sistema, podemos descobrir qual o funcionário da biblioteca que está associado a ele (`usuario.getFuncionarioBiblioteca()`) e assim poderíamos, por exemplo, gerar logs das operações realizadas por esse funcionário. Também é interessante conseguirmos fazer o inverso e assim saber qual o usuário de um determinado funcionário da biblioteca (`funcionarioBiblioteca.getUsuario()`). Assim, poderíamos, por exemplo, ter uma tela de alteração de senha que permita buscar o funcionário pelo nome e em seguida mostre um campo read-only com seu usuário e campos para preencher sua senha atual e senha nova;
- **OneToMany unidirecional:** Utilizamos esse relacionamento entre as entidades `Leitor` e `Telefone`. Note que faz sentido recuperarmos os telefones de um leitor (`leitor.getTelefones()`), mas não faz muito sentido recuperarmos o leitor de um telefone (`telefone.getLeitor()`);
- **OneToMany bidirecional e ManyToOne bidirecional:** apesar de `OneToMany` e `ManyToOne` serem nomes diferentes, representam o mesmo tipo de relacionamento quando este é bidirecional.

Adotamos esse tipo de relacionamento entre as entidades `Livro` e `Emprestimo`. Se o relacionamento for bidirecional, podemos, por exemplo, gerar um relatório informando em quais datas o livro foi emprestado ou devolvido (podemos usar `livro.getEmprestimos()` e iterar na lista de empréstimos recuperando as datas de empréstimo e devolução). Também podemos recuperar todos os empréstimos e descobrir qual livro foi emprestado em cada um dos empréstimos (`emprestimo.getLivro()`). São duas perspectivas diferentes de um mesmo conceito;

- **ManyToOne unidirecional:** Fizemos uso desse relacionamento entre as entidades `Livro` e `Editora`. Assim, em nosso sistema será possível recuperarmos a editora de um determinado livro (`livro.getEditora()`), porém não poderemos recuperar todos os livros de uma determinada editora (`editora.getLivros()`). Caso quiséssemos buscar os livros pela editora, seria necessário adotar o relacionamento `ManyToOne` bidirecional;
- **ManyToMany unidirecional:** Esse foi o relacionamento utilizado entre as entidades `Livro` e `Autor`. A partir dele, em nossa aplicação será viável recuperarmos os autores de um determinado livro (`livro.getAutores()`), porém não há necessidade de recuperarmos os livros de um determinado autor (`autor.getLivros()`);
- **ManyToMany bidirecional:** por fim, podemos citar novamente o relacionamento entre as entidades `Livro` e `Autor`. Em nossa aplicação será implementado um relacionamento unidirecional entre essas duas entidades, mas nada impediria que configurássemos um relacionamento bidirecional se houvesse essa necessidade.

Como você já deve ter percebido, o desenvolvedor deve analisar cuidadosamente os requisitos do sistema e com base nisso decidir se os relacionamentos devem ser unidirecionais ou bidirecionais. O motivo pelo qual devemos evitar esse tipo de relacionamento é que o código fica um pouco mais complicado, pois temos sempre que “amarra” as duas pontas, mantendo um estado consistente para o nosso modelo. Por exemplo, se fossemos realizar um empréstimo, teríamos que setar o objeto `livro` em nosso objeto `emprestimo` e também teríamos que adicionar o objeto `emprestimo` à lista de empréstimos do objeto `livro`, conforme o código:



```
emprestimo.setLivro(livro);
livro.getEmprestimos().add(emprestimo);
```

É muito importante que seja definido com cuidado se um relacionamento deve ser unidirecional ou bidirecional. Também é de suma importância ter uma boa base conceitual sobre os tipos de relacionamentos existentes para sempre escolher o tipo mais adequado para cada situação durante o desenvolvimento de um sistema. Na próxima parte do artigo, tendo esses conceitos-chave em mente, daremos continuidade em nosso projeto criando e mapeando as entidades.

A plataforma Java EE tem evoluído constantemente e significativamente ao longo dos anos e atualmente é uma ótima opção fazer uso dela para desenvolver aplicações corporativas de larga escala.

## Autor



Bruno Rafael Sant'Ana

[bruno.santana.ti@gmail.com](mailto:bruno.santana.ti@gmail.com)

Graduado em Análise e Desenvolvimento de Sistemas pelo SENAC. Possui as certificações OCJP e OCWCD. Atualmente trabalha na Samsung com desenvolvimento Java. Entusiasta de linguagens de programação e tecnologia.



## Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Conhecida como especificação guarda-chuva, a Java EE inclui referências a diversas outras especificações que, juntas, compõem a plataforma. Especificações como JSF, JPA e CDI fazem parte da Java EE.

Entender como essas tecnologias se complementam e como podem trabalhar em conjunto é algo essencial para um desenvolvedor. Fazendo uso desses recursos da plataforma conseguimos reduzir drasticamente a complexidade envolvida no desenvolvimento de software e podemos focar na implementação das regras de negócios.

## Links:

### Aplicação reduzida (igual a desenvolvida no artigo) no GitHub.

<https://github.com/brunosantanati/javamagazine-app-reduzida>

### Aplicação completa no GitHub.

<https://github.com/brunosantanati/javamagazine-app-completa>

### Endereço para download do JDK 8.

<http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html>

### Endereço para download do Eclipse Luna.

<https://www.eclipse.org/downloads/>

### Endereço para download JBoss WildFly.

<http://wildfly.org/downloads/>

### Endereço para download do instalador do MySQL.

<http://dev.mysql.com/downloads/installer/>

### Endereço para download do driver do MySQL.

<http://dev.mysql.com/downloads/connector/j/>

# Elasticsearch: realizando buscas no Big Data

**B**ig Data é um termo que engloba uma larga quantidade de conceitos, técnicas e ferramentas, e cujo foco é gerenciar grandes quantidades de dados. Visto como um novo paradigma computacional, o Big Data promete capturar, armazenar, analisar e compartilhar dados da ordem de *petabytes* gerados por aplicações nos mais distintos domínios. Frequentemente associado ao Big Data, o objetivo do Elasticsearch é apoiar o desenvolvimento de aplicações centradas em texto, como redes sociais, sistemas de e-commerce, sites de notícias e canais de educação. A grande vantagem do ES reside na sua arquitetura, projetada para ser escalável e para gerenciar grandes quantidades de dados de forma simples e eficiente.

A primeira versão foi lançada em 2010, pelo israelense Shay Banon. Desde então, muitas versões foram liberadas e empresas do calibre da Wikipedia, GitHub, Foursquare e Globo.com passaram a utilizar o framework. O código do ES é desenvolvido em Java e está baseado principalmente em dois frameworks da Fundação Apache: o Lucene e o Hadoop. O Lucene é utilizado como o motor de indexação e buscas em documentos desestruturados, e grande parte dos conceitos de programação deste é igualmente aplicável ao ES. O Hadoop, por sua vez, é utilizado para escalar o sistema fazendo uso de jobs Map e Reduce, um modelo de programação paralela introduzido pelo Google. Como um dos seus diferenciais, o Elasticsearch possibilita que mesmos clientes não Java utilizem suas funcionalidades, via REST e JSON.

Com base nisso, este artigo tem como objetivo apresentar os conceitos básicos do ES e discutir sua API Java. Por motivos didáticos, o ES será apresentado através de uma comparação com um banco de dados relacional (ou BDR). Sendo assim, serão analisados a partir de agora: os conceitos básicos e a arquitetura do ES; a instalação, a inserção e a busca de documentos utilizando comandos REST; e, finalmente, a API Java desta solução.

## Conceitos básicos

O ES é uma ferramenta distribuída para mineração e tratamento de textos. Sua função principal é permitir que documentos desestruturados fossem armazenados e recuperados de forma simples e eficiente.

## Fique por dentro

Esse artigo é útil para estudantes e profissionais que tenham alguma experiência em Java e queiram dar os primeiros passos no desenvolvimento para Elasticsearch. O artigo apresenta uma visão geral dos conceitos de mineração de texto, um resumo das ideias básicas contidas no framework, os primeiros passos para o desenvolvimento neste segmento e a criação de buscas avançadas. Ao final, as ferramentas de filtros, agregações e sugestões também serão apresentadas, a fim de melhorar o resultado das pesquisas.

A arquitetura do ES, que foi projetada para sempre trabalhar em cluster, suporta grandes quantidades de dados.

Elasticsearch é, de forma geral, uma versão distribuída do Lucene – framework para mineração e tratamento de texto desenvolvido pela Fundação Apache. Isto porque cada nó de um cluster ES contém tal framework para o gerenciamento das informações armazenadas.

Devido a essa relação, alguns dos conceitos básicos de ES são derivados do Lucene, a saber:

- **Índice:** define o endereço para acesso às informações guardadas no ES. De forma parecida com o esquema nos BDRs, necessitamos saber o nome e a localização na rede de um índice (por exemplo, `localhost:9200/nome_indexe`) para conectar-se e manipular as estruturas de armazenamento do ES;
- **Type:** recurso usado para nomear conjuntos de documentos armazenados em um índice, podendo ser comparado ao conceito de tabela em BDRs, pois contém vários documentos que obedecem a uma mesma estrutura de campos;
- **Documento:** é um texto plano – isto é, não corresponde a formatos binários como .doc ou .pdf – organizado em campos delimitados por chaves e vírgulas, de acordo com o padrão JSON. Sua função é similar ao das linhas de tabelas, já que é sobre os documentos que as operações de manuseio de dados (inserção, recuperação, alteração e exclusão) são realizadas;
- **Campo (field):** é a unidade mínima de informação armazenada em um documento. Deve possuir um tipo, que pode ser padrão – por exemplo: string, integer/long, float/double, boolean, ou null – ou criado pelo desenvolvedor. O campo tem a mesma função de uma coluna no BDR;
- **Mapeamento (mapping):** define a estrutura de um documento, contendo campos e a maneira como cada um deve ser armazena-

do e recuperado. O mapeamento funciona como a definição de colunas nas tabelas em BDRs;

- **Query DSL:** é a linguagem de busca (para mais detalhes, veja a seção [Links](#)). Está para o Lucene como o SQL está para os BDRs;

- **Score:** valor numérico que representa quanto bem um documento está relacionado a uma busca em Query DSL;

- **Analizador:** mecanismo para transformação de texto – por exemplo: a conversão de letras maiúsculas em minúsculas, o tratamento de espaços em branco – durante o armazenamento e recuperação de informações.

Para facilitar o entendimento, a **Tabela 1** sumariza o relacionamento entre os conceitos do Lucene e os conceitos de BDRs apresentados nesta seção.

Lucene	Banco de dados relacional (BDR)
Índice (Index)	Esquema
Type	Tabela
Documento (JSON)	Linha
Campo (Field)	Coluna
Mapeamento (Mapping)	Estrutura da tabela
Query DSL	SQL

**Tabela 1.** Mapeamento entre conceitos de bancos de dados relacionais e ES

Outros conceitos, igualmente importantes, foram adicionados pelo ES para estender o Lucene e permitir sua execução em cluster. Dentre eles, os principais são:

- **Nó:** um servidor – virtual ou físico – que contém certo número de shards e réplicas;

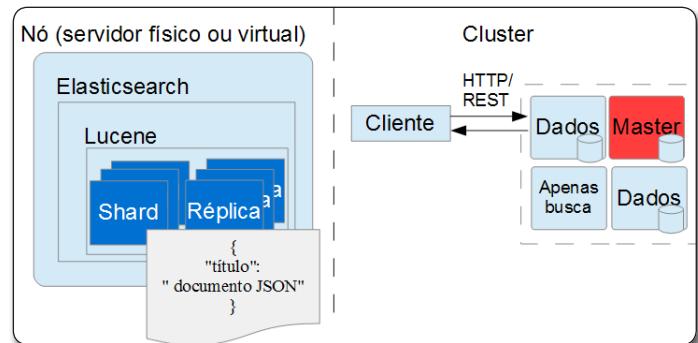
- **Shard:** um índice do Lucene que gerencia as informações armazenadas em um nó;

- **Réplica:** também um índice Lucene, porém gerenciado pelo ES como uma cópia completa de algum dos outros shards do cluster. A réplica contém os mesmos dados e é responsável pelas mesmas funções de um shard. Ela pode ser utilizada em dois casos: para melhorar o desempenho das buscas e para garantir a disponibilidade do cluster. O primeiro caso permite que haja um平衡amento de carga entre os shards do ES, diminuindo o tempo de resposta, já que o processamento das buscas será dividido entre os shards e suas réplicas. O segundo caso possibilita que o cluster continue funcionando mesmo ante uma falha do shard original. Assim, quando o shard do qual uma réplica foi criada não estiver mais disponível por causa de uma falha técnica ou um desastre, a réplica deverá assumir as funções do shard original, garantindo que o ES continue operando normalmente;

- **Filter:** oferece, em situações específicas – por exemplo, verificar se uma informação existe no índice sem que seja necessária sua recuperação completa do documento –, uma opção de melhor desempenho em relação às buscas com Query DSL;

- **Aggregation:** mecanismo para sumarizar dados em estatísticas relevantes, como a contagem, média, diferença de tempo, entre outras.

A **Figura 1** descreve a arquitetura do ES. Como pode ser observado, cada nó de um cluster ES contém um ou mais shards e réplicas, gerenciados pelo Lucene. E distribuídos em diversos shards, podem estar diferentes índices contendo documentos descritos em JSON. O ES ainda permite que os nós do cluster sejam configurados para armazenar ou não dados. Os nós que não armazenam dados são responsáveis apenas pelas atividades de processamento das buscas. Com o propósito de trabalharem de forma autogerenciada, os nós do cluster necessitam eleger um mestre (*master*), que será responsável por tarefas de administração, como a criação de um novo índice e adição de um novo nó. Entretanto, o cliente não necessita saber o tipo do nó – ou seja, se contém dados, se é mestre, ou se apenas realiza busca – para enviar uma requisição HTTP/REST, já que todos os servidores conhecem a topologia do cluster, a localização dos documentos e podem redirecionar tais requisições para o servidor que contém as informações desejadas.



**Figura 1.** Arquitetura do Elasticsearch

Em resumo, o ES adiciona funções de cluster ao Lucene, a fim de manipular informação textual e atender as necessidades do Big Data. Entender essas funções é muito importante para o desenvolvedor, porém não é suficiente para a implementação de buscas de forma efetiva. Sendo assim, antes de continuar com exemplos práticos, se faz necessário conhecer um pouco sobre análise de texto e entender como o ES avalia, armazena e recupera informações.

Para facilitar, a análise de texto será explicada através de exemplos usando dois conceitos centrais do Lucene/ES: as listas invertidas e os tokens. Como ilustrado na **Figura 2**, se documentos contendo um campo com as frases “O Elasticsearch é desenvolvido em Java” e “Elasticsearch é uma ferramenta BigData” fossem inseridos em um índice ES, diferentemente de um BDR, antes de serem armazenados esses documentos seriam analisados e divididos em pequenos pedaços de informação chamados *tokens*. Nesse exemplo, a partir da quebra das frases nos espaços em branco, seriam gerados e armazenados em uma lista inversa os seguintes tokens: *Elasticsearch*, *é*, *desenvolvido*, *em*, *Java*, *O*, *uma*, *ferramenta* e *BigData*. As listas inversas são utilizadas porque evitam a comparação textual da busca com todos os campos de todos os documentos de um índice – uma atividade que pode ser muito lenta, já que um índice pode conter muitos documentos.

# Elasticsearch: realizando buscas no Big Data

Para ilustrar, se uma busca por documentos que contenham a palavra “Elasticsearch” fosse enviada ao índice da **Figura 2**, não seria necessária uma busca textual completa em cada um dos campos de texto dos dois documentos para verificar que ambos contêm a palavra procurada, pois através de uma simples pesquisa diretamente na lista invertida – muito mais rápida, já que é internamente implementada como uma tabela hash –, os dois documentos seriam recuperados.

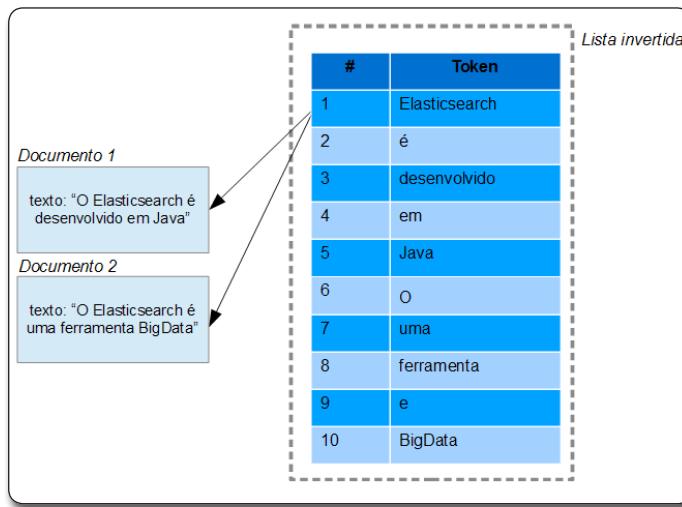


Figura 2. Exemplo de índice inverso

O processo de criação de tokens pode ser bastante complexo, incluindo análises de texto, remoção de palavras indesejadas e criação de novos tokens derivados das palavras encontradas no documento original. Tais transformações são realizadas por analisadores, como por exemplo, o analisador do tipo *stemmer*, que transforma as palavras em sua forma raiz.

Na **Figura 3** estão ilustradas as transformações das palavras *trabalhador*, *trabalho* e *trabalhar* para a forma raiz: *trabalh*. O ES oferece um grande conjunto de ferramentas para transformação de texto, por exemplo: transformação do texto para caracteres ascii, modificação de palavras para letras maiúsculas e remoção de palavras muito longas, o que aumenta sua capacidade de busca, já que em uma busca que contenha uma dessas palavras (no exemplo, *trabalhador*, *trabalho* e *trabalhar*), internamente será a forma raiz (“*trabalh*”, neste caso) que será utilizada nas comparações.

Além disso, os analisadores podem ser combinados para obter tokens que sejam mais representativos para o domínio de negócios da aplicação que está sendo desenvolvida. Por exemplo, em um site de e-commerce, as palavras usadas na busca de certo produto podem ser transformadas para sua forma raiz e em letras minúsculas, a fim de que mais resultados sejam retornados. Para obter esse resultado, podemos combinar analisadores como *whitespaces* (que divide as palavras de acordo com os espaços em branco entre elas), *lowercase* (que transforma todas as letras de uma palavra em minúscula), *stopwords* (que remove palavras que tenham pouca

relevância) e *stemmer* (que transforma a palavra na sua forma raiz), conforme ilustrado na **Figura 4**.

Em suma, internamente no ES, a seguinte sequência é executada:

- Uma inserção ou atualização de documento é recebida via PUT ou POST;
- Os analisadores são executados e cada documento é convertido em um ou mais tokens indexados;
- Os tokens são armazenados em uma lista com ponteiros para a versão completa do documento.

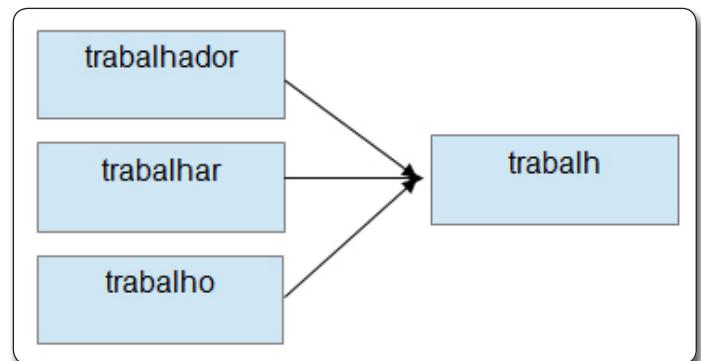


Figura 3. Análise do tipo stemmer

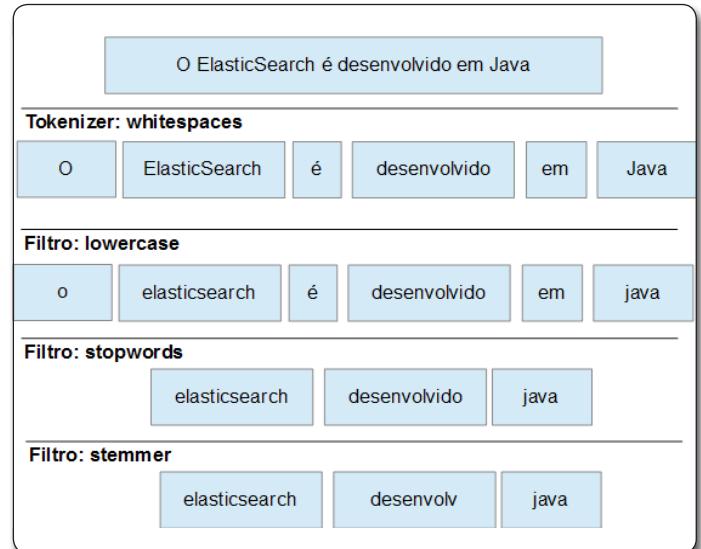


Figura 4. Exemplo de uso de analisadores

## Instalação e comandos REST

Com os conceitos básicos já abordados, podemos iniciar um exemplo prático. Para tal, necessitamos do ES instalado em um servidor. Sendo assim, faça o download do arquivo contendo o framework diretamente no site (veja o endereço na seção **Links**), descompacte este arquivo no servidor, que deve possuir o JDK 1.6 instalado, e, na pasta *bin*, execute o comando *bin/elasticsearch*.

Caso tudo esteja funcionando bem, o Elasticsearch estará disponível no endereço <http://localhost:9200> e poderemos enviar comandos para o servidor. Como já mencionado, a comunicação

com o ES é baseada na tecnologia REST, cujas vantagens são o suporte de qualquer linguagem que possa enviar requisições HTTP, a facilidade de integração e a escalabilidade. No contexto do ES, cada comando REST possui um equivalente em BDR, como ilustrado na **Tabela 2**.

Operação	Elasticsearch	BDR
Create (inserir)	POST ou PUT	INSERT
Retrieve (recuperar)	GET	SELECT
Update (atualizar)	PUT	UPDATE
Delete (excluir)	DELETE	DELETE

**Tabela 2.** Mapeamento entre comandos SQL e Elasticsearch

Dito isso, a partir de agora vamos ilustrar o seu uso através de um exemplo que simula a criação de uma biblioteca para artigos publicados na Java Magazine. Para facilitar o desenvolvimento, os exemplos utilizarão o Sense (vide seção **Links**), um plugin para o Google Chrome que atua como um cliente enviando chamadas REST/HTTP.

O primeiro passo na utilização do ES é a criação de um índice no servidor. Conforme ilustrado na **Figura 5**, o comando *PUT /javamagazine/* irá criar um índice chamado *javamagazine*.

```
PUT /javamagazine/
1 PUT /javamagazine/ 1 [Box I] acknowledged: true
2
3
```

**Figura 5.** Criação de um índice

Com o índice definido, o próximo passo é a criação de um *type* com o mapeamento dos campos, como demonstrado na **Listagem 1**. Nesse código, o *type biblioteca* é criado e nele, os campos **autor**, **título**, **texto** e **assunto** do artigo são definidos usando o tipo *string*.

O mapeamento define também como cada campo será analisado durante seu armazenamento e recuperação. Para isto, analisadores são definidos para cada campo ou o analisador padrão é utilizado, como no caso da **Listagem 1**, onde não definimos nenhum analisador.

Da mesma forma que no tópico anterior, o ES permite que analisadores sejam combinados para aumentar a capacidade de busca das aplicações. Para tal, devemos adicionar um novo analisador nas configurações do índice, que estão disponíveis no endereço */javamagazine/\_settings*. Esse endereço não contém um arquivo,

mas sim um documento JSON que especifica configurações válidas para todo o índice, como é o caso de analisadores e filtros. A **Listagem 2** mostra como criar um analisador chamado **analisador\_título**, que combina o analisador *whitespaces*, do tipo *tokenizer*, com os analisadores *trim* e *lowercase*, do tipo *filter*. É importante notar que antes de realizar a alteração das configurações o índice deve ser fechado (com o comando *POST /javamagazine/\_close*), ficando assim indisponível para buscas, e depois da alteração deve ser reaberto (com o comando *POST /javamagazine/\_open*), para que a alteração da configuração tenha efeito e para que os dados voltem a estar disponíveis.

#### Listagem 1. Mapeamento dos campos.

```
PUT /javamagazine/biblioteca/_mapping
{
  "biblioteca": {
    "properties": {
      "autor": {"type": "string"},
      "título": {"type": "string"},
      "texto": {"type": "string"},
      "assunto": {"type": "string"}
    }
  }
}
```

#### Listagem 2. Criação do analisador customizado.

```
POST /javamagazine/_close
PUT /javamagazine/_settings
{
  "index": {
    "analyzer": {
      "analisador_texto": {
        "type": "custom",
        "tokenizer": "whitespace",
        "filter": ["trim", "lowercase"]
      }
    }
  }
}
POST /javamagazine/_open
```



# Elasticsearch: realizando buscas no Big Data

A fim de que o novo analisador seja utilizado, precisamos modificar o mapeamento de campos conforme a **Listagem 3**. Nesse exemplo, além do **analisador\_titulo**, especificado na listagem anterior, são utilizados também os analisadores *keyword* e *standard*. Os campos **autor** e **assunto** recebem o analisador *keyword*, que simplesmente transforma uma **string** em um *token* único e o campo **texto** recebe o analisador *standard*, que combina os analisadores *lowercase* e *stopword*. Finalmente, o campo **titulo** recebe o analisador customizado *analisador\_titulo*.

Com o mapeamento definido, podemos utilizar o comando da **Listagem 4** para inserir um novo artigo no índice criado para armazenar a biblioteca da Java Magazine. O valor “1” no comando *POST /javamagazine/biblioteca/1* define o id do documento inserido. Esse id pode conter qualquer valor alfanumérico e caso omitido será gerado automaticamente pelo ES.

Após inserir o documento no índice, podemos recuperá-lo diretamente através do seu id. Para tal, executamos o comando *GET /javamagazine/biblioteca/1*.

Como em bancos de dados relacionais, o ES também permite a atualização e exclusão de valores do índice. Na **Listagem 5**, o comando *PUT* atualiza o nome do autor para “Luiz Santana”. Posteriormente, podemos utilizar o comando *DELETE /javamagazine/biblioteca/1* para excluir esse registro.

Com o intuito de recuperar as informações manipuladas utilizando os conceitos apresentados até aqui, devemos utilizar o comando *GET /javamagazine/biblioteca/\_search?q=assunto:elasticsearch*. Este comando busca no type *biblioteca* do índice *javamagazine* documentos que contenham a palavra *Elasticsearch* no campo *assunto*. O termo *\_search* deve sempre ser utilizado para enviar ao servidor as consultas que são definidas usando Query DSL no campo *q*.

Essa busca irá retornar o documento JSON da **Listagem 6**, cujas propriedades mais importantes são:

- **took**: descreve o tempo total gasto na execução da busca;
- **shards**: conta quantos shards foram acessados para executar a busca;

- **hits**: representa a lista dos resultados;
- **score**: informa a relevância do resultado de acordo com o algoritmo de similaridade utilizado.

## Listagem 3. Uso de analisador customizado.

```
PUT /javamagazine/biblioteca/_mapping
{
  "biblioteca": {
    "properties": {
      "autor": {"type": "string", "analyzer": "keyword"},
      "titulo": {"type": "string", "analyzer": "analisador_titulo"},
      "texto": {"type": "string", "analyzer": "standard"},
      "assunto": {"type": "string"}
    }
  }
}
```

## Listagem 4. Comando para inserção de documentos no ES.

```
POST /javamagazine/biblioteca/1
{
  "autor": "Luiz",
  "titulo": "Sua primeira aplicação em Elasticsearch...",
  "texto": "O objetivo do Elasticsearch é apoiar ...",
  "assunto": "Elasticsearch"
}
```

## Listagem 5. Comando PUT para alteração de documentos.

```
PUT /javamagazine/biblioteca/1
{
  "autor": "Luiz Santana",
  "titulo": "Sua primeira aplicação em Elasticsearch...",
  "texto": "O objetivo do Elasticsearch é apoiar ...",
  "assunto": "Elasticsearch"
}
```

Esses parâmetros devem ser aproveitados pelo desenvolvedor para entender o comportamento do ES após uma busca.

Um exemplo é o *score*, que indica quão significativo foi a busca em relação aos documentos presentes no índice. O algoritmo padrão para o cálculo desse parâmetro é o TF/IDF, que se baseia simplesmente na semelhança entre os itens procurados na busca e os documentos do índice. Modificando esse algoritmo podemos manipular o score de acordo com as necessidades do domínio de negócios para o qual a aplicação está sendo desenvolvida. Por exemplo, na biblioteca para a Java Magazine que estamos desenvolvendo nesse artigo, podemos alterar o *score* para priorizar documentos que contenham mais páginas em vez da simples similaridade textual.

Outra informação importante são os *hits*, que representam os documentos que possuem textos similares aos da busca enviada ao índice e estão, por padrão, ordenados do maior para o menor *score*. Além disso, em relação ao array de *hits*, que o desenvolvedor deverá percorrer no momento de utilizar os dados retornados, podemos observar que cada *hit* indica o índice do qual foi recuperado (no caso do exemplo, *javamagazine*), seu tipo (no caso do exemplo, *biblioteca*), seu id, o *score* em relação a essa busca e,



mais importante, o documento JSON contendo o resultado para os valores consultados.

**Listagem 6.** Resposta da busca no índice Java Magazine.

```
{  
  "took": 2,  
  "timed_out": false,  
  "_shards": {  
    "total": 5,  
    "successful": 5,  
    "failed": 0  
  },  
  "hits": {  
    "total": 1,  
    "max_score": 1.4054651,  
    "hits": [  
      {  
        "_index": "javamagazine",  
        "_type": "biblioteca",  
        "_id": "1",  
        "_score": 1.4054651,  
        "_source": {  
          "autor": "Luiz Santana",  
          "titulo": "Sua primeira aplicação em Elasticsearch...",  
          "texto": "O objetivo do Elasticsearch é apoiar ...,"  
          "assunto": "Elasticsearch"  
        }  
      }  
    ]  
  }  
}
```

## Java API para Elasticsearch

A API Java para ES permite realizar os mesmos comandos expostos em REST. Para isto, é indicado utilizar o Maven como gerenciador de pacotes, já que toda a documentação disponibilizada baseia-se nesse padrão. O ES está disponível no Maven Central (vide seção **Links**) e a **Listagem 7** apresenta a dependência Maven que deve ser adicionada ao *pom.xml* do seu projeto.

**Listagem 7.** Dependência Maven para ES no pom.xml.

```
<dependency>  
  <groupId>org.elasticsearch</groupId>  
  <artifactId>elasticsearch</artifactId>  
  <version>1.3.2</version>  
</dependency>
```

Com o ambiente de desenvolvimento configurado, o primeiro passo na criação de uma aplicação é estabelecer a conexão com o cluster Elasticsearch, que pode estar em execução na própria máquina local ou remotamente. Sendo assim, lembre-se que antes de executar a aplicação o servidor deve ser iniciado, como explicado no tópico anterior.

Existem duas formas de conectar-se ao cluster: criando sua aplicação como um nó, usada quando necessitamos estender as capacidades do cluster ES; ou como um cliente puro. Neste caso,

o cliente acessará o cluster remoto através de conexões REST/HTTP, de forma parecida aos BDRs comuns. A classe **Connection**, exposta na **Listagem 8**, apresenta as duas formas de conexão. Em ambos os casos o ES deve estar disponível para conexão, isto é, necessitamos previamente instalar e configurar o cluster como apresentado na seção anterior.

**Listagem 8.** Código para conexão de uma aplicação Java como cliente puro ou como nó do cluster.

```
package example.javamagazine;  
  
import org.elasticsearch.client.Client;  
import org.elasticsearch.client.transport.TransportClient;  
import org.elasticsearch.common.settings.ImmutableSettings;  
import org.elasticsearch.common.settings.Settings;  
import org.elasticsearch.common.transport.InetSocketAddress;  
import org.elasticsearch.node.Node;  
  
import static org.elasticsearch.node.NodeBuilder.*;  
  
public class Connection {  
  
  private Client client;  
  
  //conexão como nó do cluster  
  public Client createNode(){  
  
    Node node = nodeBuilder().clusterName("clusterJavaMagazine").  
    client(true).node();  
    client = node.client();  
  
    return client;  
  }  
  
  //conexão como cliente  
  public Client createClient(){  
  
    Settings settings = ImmutableSettings.settingsBuilder().  
      put("cluster.name","clusterJavaMagazine").build();  
    TransportClient client = new TransportClient(settings);  
  
    client.addTransportAddress(new  
      InetSocketAddress("localhost",9300));  
  
    return client;  
  }  
}
```

A primeira forma de conexão é explicitada no método **createNode()**, onde utilizamos o objeto **nodeBuilder** para criar um nó que fará parte de um cluster denominado **clusterJavaMagazine**. Caso o nome do cluster seja omitido, o default, **elasticsearch**, será utilizado. Apesar dessa facilidade, é importante que o nome seja definido para que nosso nó não se conecte a um cluster qualquer que casualmente esteja acessível – o que poderia causar um problema conhecido como *split brain*. Outro ponto importante é que o novo nó não tente armazenar informações – por isso definimos **.client(true)** no método **createNode()** da **Listagem 8** – e concentre

# Elasticsearch: realizando buscas no Big Data

seus esforços nas atividades relacionadas a buscas. Apesar de estar disponível, a opção de criar um nó que armazene informações deve ser utilizada apenas em casos específicos, nos quais as funcionalidades do ES necessitem ser estendidas, ou seja, quando queremos, por exemplo, controlar como os dados são armazenados no índice, como o cluster é gerenciado ou como requisições são respondidas.

A outra forma de estabelecer a conexão com o ES é criar um cliente puro, como ilustrado no método `createClient()`. Essa é a maneira mais simples de utilizar a API. O cliente criado não será um nó do cluster, ele apenas acessará as funcionalidades de um ES remoto. A partir de um socket que será conectado ao cluster denominado `clusterJavaMagazine`, configuramos o cliente através de um comando `put settings` encapsulado pela API no método `ImmutableSettings.settingsBuilder().put()`. Para realizar essa conexão (entre cliente e cluster), devemos definir o nome do cluster ao qual queremos conectar, seu endereço de rede (no exemplo, `localhost`) e a porta de conexão (no exemplo, usamos 9300, a padrão do ES).

Em ambos os casos, o cliente desenvolvido será do tipo da interface `Client` (`org.elasticsearch.client.Client`) e realizará todas

as atividades de manipulação de dados na aplicação que estamos desenvolvendo, como veremos na sequência do artigo.

Com a conexão estabelecida, podemos desenvolver métodos para manipular as informações no ES. Na Listagem 9 são apresentados, através dos métodos `create()`, `retrieveAll()`, `update()` e `delete()`, os seguintes comandos da interface `org.elasticsearch.client.Client`: `prepareIndex()`, para inserção; `prepareSearch()`, para busca; `prepareUpdate()`, para atualização; e `prepareDelete()`, para exclusão.

Da mesma maneira que no tópico anterior, devemos utilizar documentos JSON nos comandos de inserção e alteração. Por isso o ES oferece, através da sua API, a classe `XContentBuilder`, que facilita a criação de documentos JSON, evitando que o desenvolvedor tenha que escrever tais documentos a partir da concatenação de strings. Os documentos JSON criados com essa classe são inseridos ou atualizados respectivamente pelos métodos `prepareIndex()` e `prepareUpdate()`, através da função `setSource()` presente em ambos, ou seja, devemos criar o documento utilizando o `XContentBuilder` e incluí-lo como parâmetro do método `setSource()`.

Listagem 9. Exemplo de conexão com o cluster e os métodos de um CRUD.

```
package example.javamagazine;

import static org.elasticsearch.common.xcontent.XContentFactory.jsonBuilder;

import java.io.IOException;

import org.elasticsearch.ElasticsearchException;
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.client.Client;
import org.elasticsearch.common.xcontent.XContentBuilder;
import org.elasticsearch.search.SearchHit;

public class BibliotecaDAO {

    private Client client;
    private final static String index = "javamagazine";
    private final static String type = "biblioteca";

    public BibliotecaDAO(){
        Connection connection = new Connection();

        client = connection.createClient();
    }

    //método para inclusão de documento no índice
    public void create(String id, String autor,
                       String texto, String assunto) throws IOException {
        XContentBuilder builder = jsonBuilder()
            .startObject()
            .field("autor", autor)
            .field("titulo", texto)
            .field("texto", texto)
            .field("assunto", assunto)
            .endObject();

        client.prepareIndex(index, type, id)
            .setSource(builder).execute()
            .actionGet();
    }

    //método para alteração de documento no índice
    public void update(String id, String autor,
                       String texto, String assunto) throws ElasticsearchException, Exception {
        XContentBuilder builder = jsonBuilder()
            .startObject()
            .field("autor", autor)
            .field("titulo", texto)
            .field("texto", texto)
            .field("assunto", assunto)
            .endObject();

        client.prepareUpdate(index, type, id)
            .setSource(builder).execute()
            .actionGet();
    }

    //método para recuperação de todos os documentos do índice
    public void retrieveAll() {
        SearchResponse response = client.prepareSearch(index)
            .execute().actionGet();

        for (SearchHit hit : response.getHits().getHits()) {
            System.out.println("Id:" + hit.getId());
            System.out.println("Título:" + hit.getSource().get("titulo"));
        }
    }

    //método para exclusão de documento no índice
    public void delete(String id) {
        client.prepareDelete("javamagazine", "biblioteca", id);
    }
}
```

Como ilustrado no método `retrieveAll()` da **Listagem 9**, o método `prepareSearch()` do objeto `client` é responsável por realizar uma consulta que retorna todos os documentos do índice. As informações são recuperadas como uma lista de `SearchHit`, sendo que cada elemento dessa lista representa um objeto contendo um documento JSON que pode ser acessado como um documento texto completo no método `hit.getSource()` ou através dos seus campos pelo método `hit.getSource().get()`. Este último método receberá como parâmetro o nome do campo que está sendo consultado, por exemplo: `hit.getSource().get("titulo")`.

Finalmente, para excluir um documento, devemos utilizar `prepareDelete()`, como ilustrado no método `delete()` do nosso exemplo. De forma bastante similar à seção anterior – onde excluímos valores do índice através de REST/HTTP apenas usando o `id` –, o método `prepareDelete()` exclui um documento de acordo com o `id` deste.

Da mesma maneira que nos BDRs, normalmente o índice e o mapeamento do ES são definidos diretamente no servidor, de forma prévia ao desenvolvimento. Entretanto, caso seja necessário, o ES oferece também uma API para administração e configuração do cluster. Essa API que permite, por exemplo, criar, alterar e excluir índices e mapeamentos, também está disponível na interface `org.elasticsearch.client.Client`, porém suas funções não são acessíveis através do objeto cliente (`client`) como nos exemplos anteriores, e sim através do método `client.admin()`.

A **Listagem 10** apresenta o código para criação programática do índice `javamagazine` e do seu mapeamento. Para a criação do índice utilizamos o método `prepareCreate()`, ao qual nesse exemplo incluímos, a título de ilustração, a definição da configuração (`setSettings()`) para a alteração do número de shards para 1. Finalmente, usamos a classe `XContentBuilder` para criar um documento JSON definindo o tipo `biblioteca` e um mapeamento contendo os campos `autor`, `titulo`, `texto` e `assunto`, e posteriormente usamos o método `preparePutMapping()` para adicionar o tipo e o mapeamento ao índice `javamagazine`.

Agora que já sabemos como criar e administrar um cluster, e realizar as funções básicas de manipulação de informações em um índice, podem aprofundar nossos conhecimentos e melhorar os resultados das buscas.

## Buscas usando a API Java para ES

Busca é a atividade central do ES! Todas as outras operações, como inserção, atualização, mapeamento e administração do cluster, visam tornar essa atividade mais rápida e simples, e atender de forma eficiente as necessidades da aplicação independente do seu domínio. Por esse motivo, o ES oferece dezenas de opções para recuperar documentos de um índice. No entanto, mesmo assim, as soluções mais utilizadas são: `term` e `match`.

A busca do tipo `term` pode ser comparada às consultas em bancos de dados relacionais, já que ela procura coincidências exatas entre os termos desejados e os valores do índice. A busca do tipo `match`, por outro lado, avalia as palavras da consulta enviada – utilizando

os analisadores citados anteriormente – e retorna valores mesmo que a coincidência não seja exata. Por exemplo, se existe um documento no índice contendo o termo “Java” (com a primeira letra maiúscula) e for realizada uma busca por “java”, o `match` poderá encontrar o documento, mas o `term` não vai encontrá-lo por conta da diferença entre letras maiúsculas e minúsculas.

### Listagem 10. Código para criação de índice e mapeamento no ES.

```
package example.javamagazine;

import static org.elasticsearch.common.xcontent.XContentFactory.jsonBuilder;
import java.io.IOException;
import org.elasticsearch.ElasticsearchException;
import org.elasticsearch.client.Client;
import org.elasticsearch.common.settings.ImmutableSettings;
import org.elasticsearch.common.xcontent.XContentBuilder;

public class ClusterAdmin {

    private Client client;

    public void createIndex() throws ElasticsearchException, IOException {
        client.admin().indices()
            .prepareCreate("javamagazine")
            .setSettings(
                ImmutableSettings.settingsBuilder()
                    .put("number_of_shards", 1))
            .execute().actionGet();

        XContentBuilder builder = jsonBuilder().startObject()
            .startObject("biblioteca")
            .startObject("properties")
                .field("autor", "string")
                .field("titulo", "string")
                .field("texto", "string")
                .field("assunto", "string")
            .endObject()
            .endObject()
            .endObject()
            .endObject();

        client.admin().indices()
            .preparePutMapping("javamagazine")
            .setType("biblioteca")
            .setSource(builder)
            .execute().actionGet();
    }
}
```



A **Listagem 11** mostra o uso desses dois tipos de busca. O método `retrieveArtigosByAutor()` procura artigos do índice usando `term` para retornar documentos que contenham no campo autor a palavra exata passada como parâmetro de busca, enquanto `retrieveArtigosByTitulo()` procura os artigos do índice usando `match` para retornar documentos que contenham no campo título palavras que sejam similares às palavras informadas para a busca.

**Listagem 11.** Exemplos de códigos de busca utilizando term e match.

```
public void retrieveArtigosByAutor(String autor){  
    SearchResponse response = client.prepareSearch(index) 1|  
        .setTypes(type)  
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)  
        .setQuery(QueryBuilders.termQuery("autor", autor))  
        .execute()  
        .actionGet();  
  
    for (SearchHit hit : response.getHits().getHits()) {  
        // Manusear resultados  
    }  
  
    public void retrieveArtigosByTitulo(String titulo){  
        SearchResponse response = client.prepareSearch(index)  
            .setTypes(type)  
            .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)  
            .setQuery(QueryBuilders.matchQuery("titulo", titulo))  
            .execute()  
            .actionGet();  
  
        for (SearchHit hit : response.getHits().getHits()) {  
            // Manusear resultados  
        }  
    }  
  
    public void retrieveArtigos(String query) {  
        SearchResponse response;  
  
        while (true) {  
            response = client  
                .prepareSearch(index)  
                .setTypes(type)  
                .setSearchType(SearchType.SCAN)  
                .setQuery(  
                    QueryBuilders.multiMatchQuery(query,  
                        "titulo", "texto", "autor"))  
                .setSize(100)  
                .execute()  
                .actionGet();  
  
            for (SearchHit hit : response.getHits()) {  
                // Manusear resultados  
            }  
            // Condição de parada do while anterior  
            if (response.getHits().getHits().length == 0) {  
                break;  
            }  
        }  
    }  
}
```

O código Java para consulta de ambos os tipos – `term` ou `match` – têm a mesma estrutura: utilizando o objeto cliente, enviamos uma consulta ao índice através do método `setQuery()`. Essas consultas são definidas pelos métodos `termQuery()` e `matchQuery()` da classe `QueryBuilders` e, ao final da execução, retornarão como resultado uma lista de `SearchHit`, cuja manipulação foi omitida por ser igual à do exemplo da **Listagem 9**.

O método `retrieveArtigos()` realiza uma busca do tipo `match`, porém a estende aos campos `titulo`, `texto` e `autor`, ou seja, a mesma consulta vai considerar esses três campos na sua execução e por isso ela é chamada de *multi match*, sendo definida com o método `QueryBuilders.multiMatchQuery()`. Além disso, esse método ilustra o uso do `SCAN`, que permite paginar os resultados da busca a fim de limitar a quantidade de informação retornada. Nesse exemplo, os valores serão retornados em grupos de 100 por `shard` (tamanho este que é definido no método `setSize()`) até que todos os resultados sejam recuperados, isto é, quando nenhum `SearchHit` seja retornado. Novamente, o código para manipulação da lista de `SearchHit` foi omitido por já ter sido apresentado.

## Adicionando filtros às buscas com a API

Os filtros atuam de forma muito parecida com as buscas, mas devem ser utilizados para situações em que as respostas são do tipo sim ou não (por exemplo, no caso da consulta sobre a existência de um valor no índice) e na pesquisa por termos exatos. A vantagem do seu uso é o desempenho, pois os filtros não calculam o `score` dos documentos e seus resultados podem ser armazenados em cache.

Na API Java para ES, um filtro é criado utilizando a classe `FilterBuilder`, como ilustrado na **Listagem 12**. Neste exemplo, a função do filtro é limitar as respostas de uma pesquisa de acordo com um autor e um assunto específico. Também usamos dois `termFilters` que filtram documentos de acordo com um assunto e um autor. Além disso, esses `termFilters` são implementados através de um `boolFilter`, que os combina de acordo com as regras da lógica booleana.

Em cada uma das regras definidas por `boolFilter` (`must`, `should` e `mustNot`) podem ser adicionados um ou mais filtros que serão analisados em conjunto. Tais regras são analisadas da seguinte forma: serão retornados os documentos que atendam aos filtros da regra `must`; não serão retornados os documentos que atendam aos filtros da regra `mustNot`; e os documentos que atendam aos filtros da regra `should` só serão retornados caso nenhuma regra `must` seja atendida e atendam a uma quantidade mínima de regras `should` (o valor mínimo padrão é 1).

Assim, o código da **Listagem 12** irá retornar documentos que contenham exatamente o nome do autor, já que o `termFilter` para autor está na regra `must`, mas que não possuam o assunto passado para o método como parâmetro, já que o `termFilter` para assunto está na regra `mustNot`.

## Adicionando agregadores às buscas com a API

Agregar estatísticas é parte essencial de uma ferramenta como o ES. Por exemplo, em um site de e-commerce, necessitamos saber quantos produtos existem no índice, o preço médio e o valor mínimo desses produtos. No ES, as *Aggregations*, ou agregações, permitem calcular esse tipo de informações analíticas para summarizar os dados a partir de um conjunto de documentos. Representantes dessa categoria de cálculo são: o valor máximo, o valor mínimo, a média e a soma.

O código da **Listagem 11**, por exemplo, pode ser modificado para retornar o número de artigos de acordo com o assunto, resultando no código da **Listagem 13**. Como ilustrado nesse código, o ES oferece a classe **AggregationBuilders**, com a qual podemos criar diferentes tipos de agregadores que são adicionados a uma busca pelo método **addAggregation()**. Nesse caso, que contabiliza a quantidade de artigos que um autor possui em relação a determinado assunto, é criado um agregador do tipo contador (**AggregationBuilders.count()**) para o campo *assunto*. Após a execução dessa consulta, a resposta estará no método **getAggregations()**, que possui o resultado das agregações acessíveis por seu nome. No exemplo, usamos **getAggregations().get("counter")** para recuperar a quantidade, dividida por assuntos, de artigos presentes no índice.

**Listagem 12.** Código para criação programática de um filtro.

```
public void retrieveUsingFilter(String autor, String assunto){  
    SearchResponse response = client.prepareSearch(index).setTypes(type)  
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)  
        .setPostFilter(  
            FilterBuilders.boolFilter()  
                .must(FilterBuilders.termFilter("autor", autor))  
                .mustNot(FilterBuilders.termFilter("assunto", assunto))  
        )  
        .execute()  
        .actionGet();  
  
    for (SearchHit hit : response.getHits()) {  
        // Manusear resultados  
    }  
}
```

**Listagem 13.** Aggregations aplicado ao índice de artigos.

```
public void countArtigosBySubject(String autor) {  
    SearchResponse response = client  
        .prepareSearch(index)  
        .setTypes(type)  
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)  
        .setQuery(  
            QueryBuilders.matchQuery("autor", autor))  
        .addAggregation(  
            AggregationBuilders  
                .count("counter").field("assunto"))  
        .execute().actionGet();  
  
    System.out.println(response.getAggregations().get("counter"));  
}
```

## Adicionando sugestões às buscas com a API

Além de ser uma ferramenta simples e eficiente, o ES oferece alguns mecanismos interessantes para a criação de aplicações mais amigáveis aos usuários. Um desses mecanismos é conhecido como *suggestions*, ou sugestões. As sugestões funcionam quando um usuário deseja realizar uma busca, mas não sabe como escrever as palavras dessa busca. Por exemplo, digamos que o usuário quer buscar um artigo que contenha no título a palavra "Java Magazine". Usando as sugestões, o usuário poderá digitar "Ja" que o ES se encarregará de encontrar entre os documentos do índice possíveis complementos para as letras digitadas.

Como ilustrado na **Listagem 14**, podemos modificar os códigos anteriores a fim de permitir sugestões durante a busca por títulos de artigos presentes no índice *javamagazine*. Para tal, a função **addSuggestion()** adiciona um objeto da classe **TermSuggestionBuilder** que terá a função de procurar no campo *titulo* dos documentos do índice as palavras passadas como parâmetro para a função do exemplo (**retrieveWithSuggestions()**). Podemos verificar na listagem que o construtor da classe **TermSuggestionBuilder** recebe como parâmetro um nome para essa sugestão – no exemplo, foi escolhido "sug".

**Listagem 14.** Exemplo de uso do recurso de sugestões com a API Java.

```
public void retrieveWithSuggestions(String query) {  
    SearchResponse response = client  
        .prepareSearch(index)  
        .setQuery(QueryBuilders.matchAllQuery())  
        .addSuggestion(  
            new TermSuggestionBuilder("sug")  
                .text(query).field("titulo")).execute()  
        .actionGet();  
  
    for (Entry<? extends Option> entry : response.getSuggest()  
        .getSuggestion("sug").getEntries()) {  
  
        System.out.println("Para o termo: " + entry.getText() + ". As opções são:");  
        for (Option option : entry.getOptions()) {  
            System.out.println("\t" + option.getText());  
        }  
    }  
}
```



# Elasticsearch: realizando buscas no Big Data

Os termos encontrados como sugestão estarão disponíveis em `getSuggest()`, que possui distintas entradas (`entry`) representando cada um dos campos para os quais foi criada uma sugestão – nesse exemplo possuímos apenas a sugestão “sug”. Cada uma dessas entradas conterá, por sua vez, um conjunto de opções, que estão presentes em `getOptions()`. As opções são levantadas de acordo com os documentos presentes no índice, isto é, o ES faz uma consulta ao índice para verificar como seria possível completar a palavra baseando-se nos caracteres enviados pelo usuário. No exemplo anterior, se o usuário procura por “Ja”, o ES poderia sugerir opções como “Java”, “Java Magazine”, “Java SE” ou “JavaScript”.

Em apenas cinco anos o Elasticsearch deixou de ser uma solução desconhecida para conquistar grandes players do mercado de Big Data. A maturidade dessa tecnologia pode ser demonstrada pela recente criação de uma empresa, também chamada Elasticsearch, com o objetivo de guiar o desenvolvimento, divulgar, dar suporte e construir ferramentas – por exemplo, para gerenciamento do cluster, análise de logs de execução, integração com BDRs e clientes – que auxiliem a criação de aplicações corporativas com alto nível de qualidade. Em vista disso, espera-se que projetos de distintos domínios cada vez mais incluam o ES para gerenciar o armazenamento e a busca de suas informações textuais.

Do ponto de vista do desenvolvimento Java, é fundamental que, após entender os conceitos apresentados neste artigo, o leitor avance seu conhecimento sobre os mecanismos internos do ES, já que as possibilidades de combinação de mapeamento, analisadores e tipos de busca são enormes. Para tirar proveito das capacidades desta solução como um todo é importante também conhecer bem o domínio dos dados que serão inseridos no ES e ter em mente que durante o processo de desenvolvimento a aplicação deve ser calibrada de acordo com suas peculiaridades.

Para manter-se atualizado, o leitor pode acompanhar, e eventualmente participar, do processo de desenvolvimento dos novos comandos e modificações no ES, que são previamente discutidos através da lista da comunidade de usuários (vide seção **Links**).

Outra possibilidade muito interessante é acessar o código do ES diretamente no GitHub, o que permite verificar detalhes da implementação que muitas vezes não estão explicados na documentação oficial.

Finalmente, muitos outros artigos ainda podem ser escritos sobre ES. A seguinte lista oferece ao leitor uma ideia dos conceitos que ainda podem ser explorados: criação de *plugins* que estendam as funcionalidades do ES (por exemplo, para conexão com BDRs, para análise de textos em português, para apresentação das informações armazenadas no índice); os mecanismos internos (como o processo de descoberta de servidores no cluster, o armazenamento de informações no *shard* e a recuperação do *shard* após uma falha); conceitos avançados de busca (por exemplo, ordenação, *boosting*, scripts para alteração do score); clientes para outras linguagens de programação (por exemplo, para PHP, Ruby, C#); *tuning* para melhoria de desempenho; monitoramento do cluster; e aplicações baseadas em informações geográficas.

## Links:

**GitHub do autor, com os códigos apresentados no artigo.**

<https://github.com/lhzsantana/es-javamagazine>

**GitHub do projeto Elasticsearch.**

<https://github.com/elasticsearch/elasticsearch>

**Site da empresa Elasticsearch.**

[www.elasticsearch.com](http://www.elasticsearch.com)

**Lista da comunidade de usuários do Elasticsearch.**

<https://groups.google.com/forum/?fromgroups#!forum/elasticsearch>

**Plugin Sense.**

<https://github.com/bleskes/sense>

**ES no Maven Central.**

<http://search.maven.org/#search%7Cga%7C1%7Ca%3A%22elasticsearch%22>

**Query DSL.**

<http://www.querydsl.com>

## Autor



**Luiz Henrique Zambom Santana**

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



É bacharel e mestre em Ciência da Computação. Possui mais de 10 anos de experiência em programação Java e há dois anos atua com tecnologias de Big Data. Trabalhou em projetos para grandes empresas no Brasil, Argentina e Alemanha. Atualmente é startupper e consultor em Elasticsearch na Emergi.net.

## Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



# FÓRUM DEVMEDIA

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



No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!

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

# Arquitetura de Software: Avaliando a arquitetura de seus sistemas

Conheça neste artigo um método para avaliação de arquitetura de software que ajuda nas tomadas de decisão

**E**laborar a arquitetura de um software pode ser uma tarefa árdua para muitos profissionais. Dependendo do tamanho e da missão do sistema, são milhares de linhas de código, centenas de componentes, bibliotecas de terceiros, relatórios, integração com sistemas legados, servidores de aplicações, tabelas de banco de dados, segurança de acesso, alta disponibilidade e demais elementos que aumentam a complexidade da arquitetura.

Como verificado, o desafio é grande e desenvolver um sistema sem o devido planejamento não é a melhor solução. Se já é complicado para um arquiteto de software e sua equipe colocarem todos esses artefatos funcionando em situações normais de operação, imagine em momentos de instabilidade como excesso de usuários, falha de hardware, indisponibilidade de rede ou banco de dados, etc. Agora, imagine como colocar um sistema semelhante em funcionamento sem planejar sua arquitetura?

Durante o desenvolvimento de um sistema o arquiteto irá se deparar com perguntas como: Qual componente deve ser utilizado para integrar com o legado? Será que a interface de comunicação com o ERP deverá ser feita com web services ou será melhor utilizar uma API proprietária do fabricante que possui um melhor desempenho? Há restrições legais para fazer uso de código open source? Qual será o comportamento do sistema se a conexão com o banco de dados ficar indisponível? Perguntas como estas precisam ser respondidas para que seja definida a arquitetura do sistema.

## Fique por dentro

Este artigo é útil para arquitetos de software que desejam conhecer técnicas que lhes ajudem a avaliar a arquitetura de um sistema, identificando suas principais camadas, componentes e seus relacionamentos, e como alterá-los sem afetar a continuidade do negócio. Durante o ciclo de vida de um software, uma coisa é certa: haverá mudanças. Como consequência dessas mudanças, temos as decisões, que precisam ser tomadas para atender aos requisitos especificados, decisões estas que devem ser cuidadosamente estudadas para não impactar negativamente na arquitetura do software, trazendo prejuízos ao desempenho, escalabilidade e outros atributos de qualidade do sistema.

Deste modo, o ideal é que as respostas sejam obtidas de forma consciente e racional, baseadas em boas práticas, padrões e experiência do arquiteto e da equipe, avaliando para cada decisão o risco envolvido. Porém, as respostas que estão corretas hoje podem não mais representar a melhor escolha amanhã, pois o negócio do cliente exige mudanças frequentes. Sendo assim, como avaliar o impacto dessas mudanças dentro da atual arquitetura? A inclusão de uma nova funcionalidade poderá degradar o desempenho do sistema?

Felizmente, a busca pelas respostas de questões como estas não necessita de adivinhações ou outro tipo de inspiração que não seja o conhecimento. Para ajudar a obter essas respostas deve-se escolher um método de avaliação arquitetural e aplicá-lo em seu trabalho. A adoção de um método não exige grandes transformações na empresa e, na maioria das vezes, sua utilização se dá de modo direto, sem dispendiosas adaptações ao modelo de negócio,

e abrange qualquer tipo de sistema de diversos segmentos de mercado. Entretanto, os métodos divergem no estilo e forma de trabalho, podendo um método ser mais detalhista e levar mais tempo na sua execução em relação a outro.

Com base nisso, neste artigo será apresentado um método que auxilia os arquitetos de software e suas equipes na avaliação arquitetural e na tomada de decisões sobre a arquitetura. Este método foi escolhido por ser mais adequado a times que utilizam metodologias ágeis e desejam que o impacto da avaliação seja o menor possível no prazo e custo durante o desenvolvimento do software.

## O que é arquitetura de software?

Todo sistema de software tem uma arquitetura. Talvez ela não seja do conhecimento de todos os envolvidos e não esteja documentada. Talvez não tenha sido planejada e tenha sido evoluída mediante diversas modificações durante o ciclo de vida do software. Mesmo assim, ainda podemos assegurar categoricamente que a afirmativa é verdadeira.

Para iniciar o entendimento sobre o que é arquitetura de software, usaremos a definição do livro Software Architecture in Practice. Apesar disso, existem dezenas de outras definições publicadas por outros autores, mas que acabam por passar um mesmo conceito: “Arquitetura de software é a estrutura (ou estruturas) de um sistema, construída por elementos (componentes) de software, suas propriedades visíveis externamente e dos relacionamentos entre esses elementos.”.

Nessa definição, vale destacar o trecho “relacionamento entre esses elementos”, ou seja, como ocorre a interação entre as partes, pois a arquitetura precisa ser elaborada buscando o baixo acoplamento para facilitar a substituição dessas partes, quando necessário, e não acarretar em elevados custos de manutenção na reescrita de código. Em sistemas modernos, a interação ocorre através de interfaces, o que proporciona um baixo acoplamento, visto que um componente não precisa conhecer os detalhes de implementação de outro para estabelecer uma comunicação.

Para guiar a arquitetura do sistema é possível adotar alguns padrões, como client-server e 3-tier. Como recomendação para a escolha de um padrão arquitetural deve-se levar em consideração, principalmente, os requisitos não funcionais, pois estão associados a atributos de qualidade do sistema, como disponibilidade, manutenibilidade, flexibilidade, segurança, entre outros, e que por vezes são difíceis de identificar ou não estão bem documentados quando comparados aos requisitos funcionais.

Além disso, usar os requisitos não funcionais como ponto de partida na definição de uma arquitetura traz benefícios, afinal, não há uma regra que determine que uma arquitetura seja melhor do que outra sem que haja uma análise de tais requisitos. Por exemplo, para um sistema de e-commerce, escalabilidade e portabilidade de plataformas são essenciais devido a sua característica de acesso realizada por qualquer computador conectado à internet. Isso permite que um grande número de usuários acesse



o sistema simultaneamente utilizando um navegador web em plataformas heterogêneas (Windows, Linux, Mac). Neste cenário, uma arquitetura 3-camadas é forte candidata como padrão arquitetural da solução, visto que a separação das camadas cliente e negócios (serviços) proporciona a utilização de uma interface web acessível a qualquer navegador, alcançando assim a portabilidade desejada, como também proporciona a escalabilidade da camada de negócios utilizando servidores em *cluster* para atender a demanda dos usuários simultâneos.

Por sua vez, um sistema batch que precisa importar informações de arquivos texto para um banco de dados relacional tem como requisitos fundamentais trabalhar com grandes volumes de dados e ter baixa latência de rede. Neste cenário, uma arquitetura client-server poderá ser o padrão escolhido para atender às necessidades da solução.

Portanto, é importante que a escolha de um padrão arquitetural seja cuidadosa, para evitar que seja tomada uma decisão equivocada e que pode levar a uma mudança muito cara no futuro.

## Avaliação arquitetural

É sabido que alterações em sistemas são mais baratas se forem realizadas no início. Quanto mais cedo forem encontrados os problemas, menor será o custo envolvido na correção dos mesmos. Esta máxima também é válida para a arquitetura de software, pois tentar incluir qualidade posteriormente a um sistema é um grande desafio, de maneira que a qualidade deve estar incorporada desde o princípio.

Sendo assim, avaliar a arquitetura de um software não é uma tarefa a ser realizada somente no início do desenvolvimento do sistema. A avaliação deve ocorrer durante todo o seu ciclo de vida.

À medida que um software necessita de mudanças para atender ao negócio do cliente, a arquitetura precisa ser reavaliada para ajudar na tomada de decisão sobre onde e como devem ser realizadas essas mudanças, de modo que não haja degradação nos atributos de qualidade do software.

Uma arquitetura mal projetada ou que não tenha recebido a devida atenção pode levar um projeto ao fracasso. Se um sistema é construído com uma arquitetura que não permita a escalabilidade, ele poderá funcionar bem para uma dezena de usuários simultâneos. Entretanto, se este mesmo sistema necessitar de expansão para atender a milhares de usuários, podem ocorrer problemas se não houver uma avaliação arquitetural adequada para viabilizar esta mudança. Afinal, a arquitetura não foi projetada para este novo cenário e o sistema não funcionará adequadamente e apresentará falhas como lentidão e indisponibilidade para seus usuários, trazendo prejuízos e insatisfação ao cliente.

Analizando cuidadosamente as alterações necessárias para evitar problemas como os mencionados, certamente serão observados benefícios que justificam o investimento de tempo na avaliação de uma arquitetura. Entre eles, podemos destacar:

- **Financeiro:** em média, estima-se que uma avaliação arquitetural resulte na redução de 10% dos custos de um projeto;

- **Compreensão:** a tarefa de analisar a arquitetura força os envolvidos a documentar e compreender a arquitetura do sistema, convergindo o conhecimento da equipe;
- **Justificar decisões:** se um arquiteto é questionado sobre um sistema não possuir alta disponibilidade, é fácil justificar a decisão desse requisito não ter sido incluído na arquitetura por algum fator limitante (ex.: custo do projeto). Logo, as decisões tomadas e registradas pelo arquiteto com base em documentos e reuniões durante a fase de definição da arquitetura do sistema estão embasadas em critérios técnicos e de conhecimento de todos;
- **Detectar problemas com antecedência:** identificar os principais problemas na fase inicial, reduzindo os custos de correção no futuro;
- **Validação dos requisitos:** possibilita confrontar a arquitetura com os requisitos especificados, de modo a verificar a sua aderência;
- **Melhoria contínua:** a constante avaliação arquitetural proporcionará à empresa o domínio das técnicas de avaliação e maior conhecimento por parte da equipe. Deste modo, resultará em arquiteturas com maior qualidade, além de disseminar a cultura entre os desenvolvedores.

Apesar dos benefícios citados, boa parte das empresas ainda não praticam a avaliação arquitetural em seus sistemas. Um dos principais fatores para que isso ocorra é a grande adoção de metodologias ágeis, que não encorajam a utilização de métodos de avaliação arquitetural por acreditar que será consumido muito tempo e recurso.

## Métodos de avaliação arquitetural

Pesquisadores de universidades e institutos internacionais realizam pesquisas com o intuito de criar métodos para avaliação arquitetural de software e então aplicá-los em projetos reais de diferentes segmentos e tamanhos. Através desses estudos, tais métodos são testados e melhorados para atender ao mercado.

Infelizmente, não existe um método que atenda a todas as equipes e metodologias de desenvolvimento de software. É preciso experimentar alguns até encontrar aquele que mais se adapte ao seu cenário e à sua forma de trabalho para que seja produtivo e ajude a resolver problemas ao invés de causar prejuízos.

Dentre as dezenas de métodos disponíveis, podemos destacar o já consagrado ATAM e o recém-desenvolvido DCAR. Ambos serão abordados rapidamente neste artigo, e o foco principal será dado ao DCAR, por apresentar uma proposta mais próxima às metodologias utilizadas atualmente nas empresas desenvolvedoras de software.

### ATAM

Dentre os vários métodos que podem ser adotados para avaliar uma arquitetura de software, um dos mais conhecidos é o ATAM (*Architecture Tradeoff Analysis Method*), desenvolvido em 2000 pelo Software Engineering Institute (SEI) da Universidade Carnegie Mellon.

Seu principal foco está em realizar uma avaliação da arquitetura considerando os atributos relacionados à qualidade de software. Tem como característica ser um método baseado em cenários, onde são criadas situações de uso do sistema. Por exemplo, elabora-se um cenário no qual o sistema precise processar 10 mil transações por segundo e então se avalia o comportamento do sistema nesse contexto. Se o sistema não atingir o atributo de qualidade de desempenho, então se estuda a arquitetura para realizar as mudanças que permitam que o cenário se concretize.

Neste exemplo, ao tentar alcançar o desempenho desejado, o arquiteto perceberá que isto somente será viável se o sistema perder a flexibilidade na geração de *queries* dinâmicas em algumas partes. Esta situação demonstra que ocorreu um conflito entre os atributos flexibilidade e desempenho, o que chamamos de “*tradeoff*”. Em momentos como esse o arquiteto deve resolver o “*tradeoff*”, optando por diminuir a flexibilidade do sistema para conseguir o desempenho necessário caso esta seja a decisão mais adequada para o negócio.

O tempo de avaliação de uma arquitetura, segundo dados do ATAM, é de três a quatro dias, contando com uma equipe de avaliadores treinados, arquitetos e interessados no sistema que possam contribuir com a avaliação.

#### DCAR

Um método de avaliação arquitetural mais recente é o DCAR (*Decision-Centric Architecture Reviews*). Esse método promete ser uma alternativa ao ATAM, sendo ideal para projetos que trabalham com metodologias ágeis, na tentativa de aproximar as equipes de desenvolvimento da prática de avaliação arquitetural.

A proposta do DCAR é ser um método leve, de baixo impacto no tempo e nos recursos do projeto e permitir a avaliação da arquitetura com base em decisões arquiteturais, sem a necessidade de testar a arquitetura em cenários.

Segundo estimativas, as sessões de avaliação do DCAR são curtas (metade de um dia de trabalho), contando com a presença de três a cinco participantes, incluindo o arquiteto. Assim, acaba sendo uma opção para projetos com tempo e orçamento limitados ou quando os interessados não têm disponibilidade total para participar das reuniões.

Neste artigo, por ser mais recente do que o ATAM e mais adequado às equipes que utilizam metodologias ágeis em seu ciclo de desenvolvimento de sistemas de software, será apresentado o método DCAR para avaliação arquitetural.

#### Momento de decisão

Ao desenvolver um sistema a equipe terá que tomar uma série de decisões para a construção deste. As escolhas vão desde a linguagem a ser adotada, o gerenciador de banco de dados, frameworks e se serão utilizados componentes open source.

Basicamente, essas decisões arquiteturais são as escolhas que um arquiteto de software deverá tomar baseado em forças como custo e segurança que atuam no projeto. Tais forças exercem influência direta, mas nem sempre estão claras para os interessados.

A equipe responsável pelo sistema não deve analisar isoladamente uma decisão, pois ela pode estar inter-relacionada a outras. Assim, deve-se analisar o conjunto e procurar combinações para atingirem um mesmo objetivo.



Um exemplo dessa influência é a opção por utilizar um banco de dados em memória para obter um rápido tempo de resposta. Note que esta decisão gera um impacto negativo na confiabilidade, pois caso o sistema entre em colapso pode ocorrer perda de informação. Para minimizar o problema o arquiteto poderá optar por manter uma réplica dos dados.

Compreender essa relação entre as decisões é de grande importância, pois auxilia na identificação de decisões que sofrem influências de outras e podem ter consequências em grande parte da arquitetura.

Portanto, toda decisão que for identificada precisa ser avaliada pela equipe. Ao analisá-las, restrições como riscos, considerações da organização, preferências pessoais, experiência e objetivos de negócios influenciarão nas decisões arquiteturais, sendo o arquiteto o responsável por resolver possíveis conflitos, sempre buscando o consenso para chegar a uma decisão final.

É importante ressaltar que tais forças não são imutáveis. Sendo assim, durante a execução do projeto uma força pode perder relevância por mudanças em requisitos ou fatores externos à empresa, como leis e regulamentações impostas pelo governo.

Nesse contexto, o DCAR irá ajudar apoiando a análise das decisões arquiteturais. E como detalhamento disso, serão mostrados seus passos, templates e sua dinâmica na avaliação arquitetural de um sistema de software.

## Introdução ao DCAR

O DCAR é um método sequencial composto por apenas nove passos bem definidos que guiam a sua execução. Cada um desses passos é similar a um processo, onde são definidas as entradas, os participantes e as saídas que servirão de entrada para o passo seguinte. A **Figura 1** mostra o fluxo desse método.

Esse diagrama demonstra que o DCAR é um método iterativo, sendo cada iteração representada por um passo. Isso torna o método simples de aplicar, pois ele segue um fluxo contínuo, direto, sem desvios.

Outro ponto positivo sobre o DCAR é que se trata de um método enxuto, que exige uma equipe reduzida para sua execução. Além disso, em muitos dos passos as equipes não precisam estar reunidas ao mesmo tempo, permitindo com isso uma otimização dos recursos humanos envolvidos.

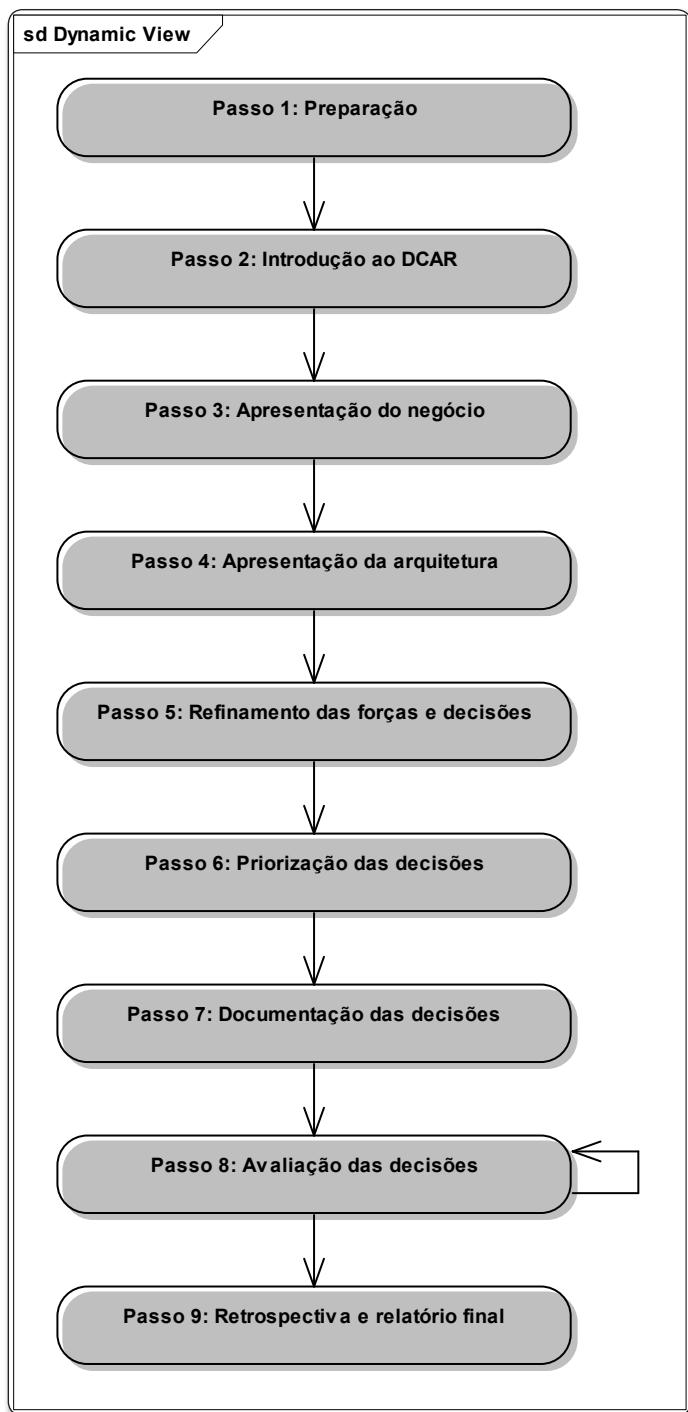


Figura 1. Passos do método DCAR

Basicamente, recomenda-se a participação do arquiteto e um ou dois membros do time de desenvolvimento que tenham diferentes papéis ou responsabilidades. Por exemplo, podem participar um desenvolvedor e um analista de requisitos. Também é importante que alguém represente o papel do cliente, podendo ser o Product Owner de uma equipe ágil, pois ele possui um amplo conhecimento do negócio do cliente e poderá identificar as forças do ponto de vista do negócio que influenciarão nas

decisões da arquitetura, como a força “time-to-market”, que sinaliza o tempo entre o planejamento de um sistema e sua disponibilização no mercado.

Note que em cada passo do diagrama do DCAR há uma tarefa de revisão das decisões que foram escolhidas. Para as revisões, a equipe responsável deve ser externa ao time, podendo ser da mesma organização ou de uma consultoria especializada. É importante que essa equipe de revisores tenha boa experiência em projetos de arquitetura. Se possuírem conhecimentos na mesma área de domínio a qual o sistema está sendo desenvolvido, excelente, mas isso não é imprescindível.

A seguir serão abordados com mais detalhes cada passo do DCAR apresentado no diagrama.

### **Passo 1 - Preparação**

Este passo é bem simples e não há necessidade de realizar uma reunião entre os envolvidos. Nele, o arquiteto deve preparar uma apresentação do sistema a ser avaliado contendo os pontos mais relevantes da arquitetura, numa visão de alto nível. Para ajudar na compreensão da informação pela equipe é recomendável que o arquiteto utilize padrões e tecnologias conhecidas ou previstas na arquitetura, como servidores de aplicação e de banco de dados, de modo a ilustrar o problema com artefatos comuns e que já fazem parte do conhecimento das pessoas, e apresente as principais decisões tomadas durante o design da arquitetura, as mudanças esperadas, os riscos, etc.

O representante do cliente (Product Owner, por exemplo) deve preparar uma apresentação mostrando a perspectiva do cliente, buscando descrever as principais características do produto, diferenciais para o negócio, possíveis restrições e atributos de qualidade desejáveis, como desempenho, disponibilidade, segurança, capacidade de mudança, interoperabilidade, etc.

O time de revisão deve receber essas apresentações antes da próxima etapa para que possam estudá-las e avaliá-las. O objetivo é encontrar potenciais decisões que ajudem a definir a arquitetura e as principais forças que atuam sobre o sistema. Se existirem documentações adicionais sobre o sistema, estas poderão ser encaminhadas para o time de revisão para ajudar na compreensão.

### **Passo 2 - Introdução ao DCAR**

O objetivo aqui é apresentar o método DCAR a todos os envolvidos. Para isso, realiza-se uma breve introdução ao DCAR, explicação do fluxo dos nove passos do método, denominação dos papéis e atribuição das responsabilidades a cada participante. A ideia é fazer com que todos os participantes conheçam como o método funciona, a importância de cada um dentro do processo e a dedicação que será necessária.

Evidentemente, esta introdução é muito importante para as primeiras experiências com o DCAR. À medida que o método torna-se parte integrante do processo de desenvolvimento e as pessoas envolvidas já estejam habituadas, este passo pode ser pulado ou reduzido sem prejuízos ao projeto.

### **Passo 3 - Apresentação do negócio**

A apresentação do negócio é um passo importante da avaliação. Através dele, o representante do cliente vai mostrar a todos os membros da equipe qual a importância do sistema do ponto de vista do negócio, utilizando os slides elaborados no primeiro passo.

O objetivo é identificar as principais forças de decisão que estão relacionadas ao negócio e que devem ser consideradas durante a avaliação. Ou seja, trabalha-se para encontrar forças que vão exigir da arquitetura soluções para resolver questões como redução de custos de implementação, alta disponibilidade, escalabilidade para atender a picos de demanda, entre outras forças ligadas ao negócio do cliente. Uma dica é sempre associar essas forças a uma possível perda financeira caso a arquitetura não cumpra o papel exigido para resolver esses problemas, pois se o impacto financeiro é maior, maiores são as chances de investimento da empresa na solução do problema.

Durante um período de aproximadamente 20 minutos de apresentação o time de revisores pode tirar dúvidas, anotando potenciais forças apresentadas e identificando outras novas com base em suas experiências.

Após a apresentação não há necessidade do representante do cliente permanecer na sessão, mas poderá ficar caso considere que ajudará com esclarecimentos.

O resultado deste passo é a elaboração de uma lista com as forças que influenciarão as decisões arquiteturais do próximo passo.

### **Passo 4 - Apresentação da arquitetura**

Utilizando os slides gerados no Passo 1 - Preparação, o arquiteto apresenta a arquitetura para os participantes durante aproximadamente 50 minutos. A ideia principal é demonstrar uma visão geral da arquitetura proposta. Assim como ocorre na apresentação do negócio, o time de revisão deverá questionar a arquitetura para esclarecer pontos identificados durante a análise dos slides que todos os envolvidos receberam antecipadamente ao final do primeiro passo.

Nessa etapa do método é importante que a equipe de revisão tenha experiência em identificar decisões arquiteturais para poder questioná-las junto ao arquiteto. Deve-se trabalhar com foco na identificação das tecnologias utilizadas propostas pela arquitetura, como servidores, linguagens, frameworks e componentes de terceiros, e na busca de padrões arquiteturais.

Ao final desse passo, os revisores deverão elaborar uma lista com as forças e decisões identificadas nos passos 1, 3 e 4, pois a equipe já teve seu primeiro contato com a arquitetura proposta e os requisitos do negócio. Essa lista será revisitada no passo seguinte.

### **Passo 5 - Refinamento das forças e decisões**

De posse das informações levantadas anteriormente, o objetivo é identificar a relação entre as decisões arquiteturais e resolver conflitos que possam ocorrer entre os demais relacionamentos definidos nos passos anteriores.

Para ajudar nessa etapa, um diagrama UML pode ser utilizado como ferramenta para ilustrar essa relação entre as decisões, pois sem uma ferramenta desse tipo pode ser muito difícil de capturá-las, visualizá-las. Na **Figura 2** temos um exemplo de diagrama que mostra os relacionamentos. Cada elipse representa uma decisão contendo uma breve descrição para facilitar a identificação.

O diagrama de relacionamento de decisão deve ser ajustado e criticado pelos envolvidos até que todos os participantes tenham uma compreensão clara das decisões e de seus impactos na arquitetura. Enquanto houver dúvidas sobre os relacionamentos ou as pessoas do time não chegarem a um acordo se uma decisão deve ser realmente descartada, o grupo deve alterar o diagrama até que haja consenso sobre as decisões e os relacionamentos.

Vale ressaltar que uma decisão arquitetural pode surgir influenciada por outra, como também pode perder importância porque outra decisão inviabilizou sua escolha. Apesar de existirem diferentes tipos de relacionamentos, as relações *caused by* e *depends on* exercerão maior influência nas decisões arquiteturais que serão escolhidas ao final.

Além disso, é importante reforçar que para cada decisão deve haver uma ou mais forças que justifiquem a sua escolha, pois as forças estão diretamente associadas ao negócio. E se a arquitetura

não atende ao negócio, o resultado não trará os benefícios esperados pelo cliente.

É preciso ter conhecimento sobre quais forças determinada decisão ataca. Para isso, elas precisam estar detalhadas considerando o ponto de vista do negócio, não havendo ambiguidades. Nesse contexto, a utilização de termos que fazem parte do domínio do cliente ajuda a evitar erros de interpretação. Sendo assim, prefira descrever as forças como “emissão de 120 notas fiscais por segundo”, em vez de “cinco mil transações por segundo”.

#### Nota

As relações *caused by* indicam que uma decisão surgiu por influência de outra. Por exemplo, no diagrama decidiu-se pela utilização de EJB 3.1 devido à relação causada pelo uso de JPA no mapeamento objeto/relacional. Porém, seria possível utilizar JPA com Hibernate ao invés de EJB. Assim, a decisão causadora (JPA) exerce influência pelo uso de EJB, mas permite alternativas com considerável grau de flexibilidade.

As relações *depends on* são mais rígidas quando comparadas às *caused by*. No exemplo, ao decidir sobre o uso de EJB, automaticamente criou-se a dependência de um container Java EE 6, sendo o GlassFish o escolhido. Mesmo que fosse utilizado outro container, este deveria suportar a especificação Java EE 6 do mesmo modo que o GlassFish para permitir a execução de componentes EJB. Assim, a flexibilidade é limitada ou, em alguns casos, até mesmo nula.

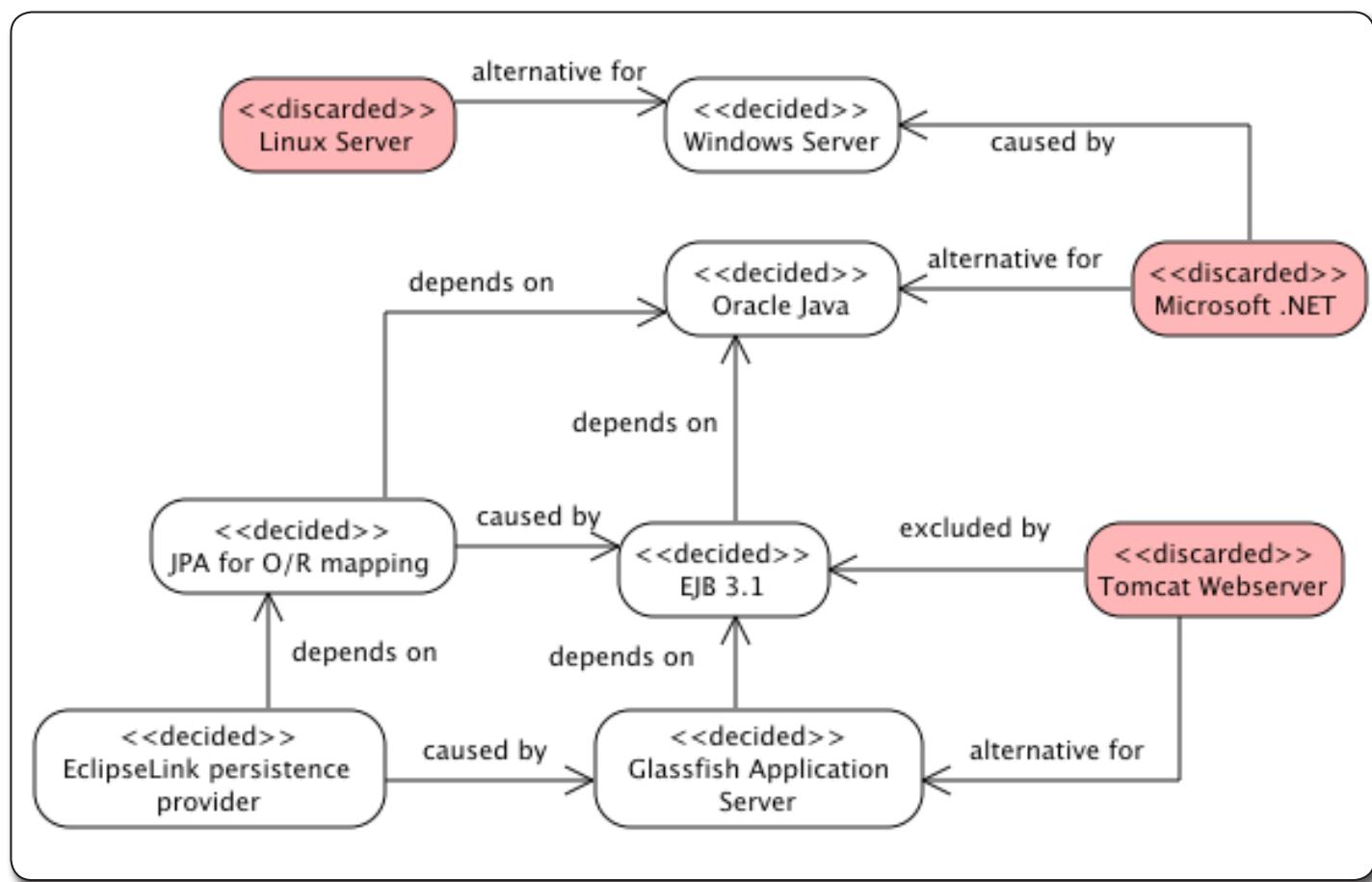


Figura 2. Diagrama de relacionamento de decisão

Nome	Redundância de servidores			
Problema	A aplicação deve executar mesmo se o servidor falhar			
Solução ou descrição da decisão	O sistema é implantado em dois servidores com a mesma configuração, porém um está ativo e outro inativo. O servidor ativo provê todos os serviços do sistema, enquanto o segundo servidor atua em modo passivo, nos bastidores. Quando o servidor ativo falhar, o inativo passa a ser o ativo e assume as atividades. Durante a operação do sistema uma atualização dos dados será realizada periodicamente entre os servidores para manter o status das informações. A solução proposta segue o padrão de redundância de funcionalidade.			
Soluções alternativas consideradas	Ambos os servidores estão ativos. Externamente, há um balanceamento de carga que decide qual dos servidores atenderá as requisições dos serviços. Entretanto, esta solução causará mudanças maiores no sistema. Apesar do aumento da disponibilidade oferecido pela solução, isso causará um impacto nos custos no qual o cliente não está preparado para assumir. Além disso, o mecanismo de balanceamento de carga pode apresentar um ponto de falha único. Por esses motivos, esta alternativa foi descartada.			
Forças a favor da decisão	<ul style="list-style-type: none"> <li>Implementação mais fácil e com menor custo, se comparada à solução alternativa;</li> <li>Pode ser entendido facilmente para outras versões do sistema, onde a redundância não é utilizada;</li> <li>Sem incidência de custos adicionais.</li> </ul>			
Forças contra a decisão	<ul style="list-style-type: none"> <li>O processo de troca de servidores é lento.</li> <li>Dificuldade em oferecer alta disponibilidade maior que a atual, que é de 99,99%.</li> </ul>			
Classificação	Verde	Amarelo	Amarelo	Vermelho
Justificativa	A solução proposta é aceitável	Preocupa o tempo de troca dos servidores em caso de falha	Solução baseada em um padrão reconhecido. Disponibilidade pode vir a ser um problema no futuro.	Essa decisão precisa ser reavaliada, pois o próximo release do sistema possui requisitos de alta disponibilidade.

**Tabela 1.** Exemplo de documentação de decisão do DCAR

#### Passo 6 - Priorização das decisões

Dependendo do tamanho e da complexidade do sistema que se está estudando a arquitetura, a lista de decisões pode ficar um pouco grande para ser analisada, podendo passar de uma centena de decisões. Deste modo, torna-se praticamente inviável analisar todas as decisões, pois demandará muito tempo e dificilmente o grupo chegará a um consenso. Assim, haverá a necessidade de realizar uma negociação para eleger as decisões mais prioritárias.

Um bom critério para priorizar as decisões é considerar aquelas de missão crítica, de alto risco e que tenham um impacto maior no custo do projeto e maior retorno financeiro.

Para auxiliar nesse processo de priorização, deve-se fazer uma votação das decisões, onde cada participante terá 100 pontos para distribuir entre as decisões listadas considerando os critérios de priorização. Ao final, levam-se para o próximo passo entre sete e dez decisões com maior pontuação.

#### Passo 7 - Documentação das decisões

É o momento de documentar as decisões que receberam pontuação mais alta no passo anterior. Para isso, o arquiteto e demais participantes devem dividi-las entre si de modo que cada participante tenha de duas a três decisões para documentar dentro da arquitetura. Vale ressaltar que os participantes devem documentar as decisões que eles possuam mais conhecimento sobre o assunto. Por exemplo, se uma decisão for sobre a utilização de um banco de dados NoSQL, então a pessoa da equipe que irá documentá-la deve conhecer o assunto bancos de dados não relacionais.

Neste documento, a equipe deve descrever a solução arquitetural adotada, o problema que essa decisão propõe solucionar, demonstrar alternativas viáveis e as forças que devem ser consideradas para avaliação da decisão. A lista de forças utilizadas é a que foi elaborada no passo 5, mas novas forças podem ser adicionadas caso seja necessário. Na **Tabela 1** é apresentado um modelo para documentação das decisões do DCAR.

As decisões são documentadas conforme o modelo da **Tabela 1**. Para cada decisão deve haver informações como nome, o problema ao qual ela pretende solucionar, a solução proposta, uma solução alternativa e forças que vão pesar a favor e contra essa decisão. Depois de documentadas, as decisões passam ao próximo passo para serem classificadas e justificadas.

#### Passo 8 - Avaliação das decisões

Após a documentação das decisões é necessário avaliá-las conforme sua escala de prioridades. Nesta etapa, cada participante responsável por documentar a decisão de acordo com seu grau de experiência deverá apresentar a solução para os demais. Nesse momento, todos os envolvidos devem ouvir a explicação e a solução proposta pelo membro da equipe. Havia necessidade, é possível ajustar o documento durante a explicação para corrigir alguns pontos que não ficaram claros na escrita.

Em seguida, os participantes avaliam a solução proposta levando em consideração as forças que atuam a favor e contra a decisão. Com base nessas informações, cada participante que está avalian-

do a solução deverá dar o seu parecer escolhendo uma cor da linha de classificação e apresentando a sua justificativa.

As cores da classificação (verde, amarelo e vermelho) irão indicar se a solução será implementada ou terá que ser revista. A cor verde indica que o avaliador considerou a solução boa e que deve ser adotada. O amarelo indica que a solução é aceitável, mas existem algumas observações que precisam ser tratadas. E o vermelho indica que a solução deve ser reavaliada.

Se houver alguma situação em que os avaliadores classifiquem a solução com cores diferentes, ou seja, não existindo uma unanimidade por parte da equipe, então a decisão é discutida por cerca de 20 minutos. Durante esse tempo, se ainda não houver um consenso, a solução deverá aguardar mais alguns instantes e ser reavaliada em um segundo momento.

Esse passo tem uma característica diferente dos demais passos do método DCAR porque possui um loop. Isso sinaliza que o processo só poderá seguir para o passo seguinte após todas as decisões prioritárias terem sido avaliadas e terem alcançado um consenso. Enquanto existir uma decisão em estudo pelos avaliadores, o processo permanecerá nesta etapa.

## Passo 9 - Retrospectiva e relatório final

Nessa última etapa, todas as decisões de maior prioridade já foram documentadas e avaliadas. Todos os documentos de decisão elaborados, planilhas com as votações de priorização das decisões e outros documentos importantes para a definição da arquitetura que foram gerados durante a execução do método DCAR, serão compilados em um relatório pelo time de revisão e será apresentado ao arquiteto do sistema em até duas semanas, não devendo se estender mais do que esse período para não comprometer o andamento e o custo do projeto. Nesse novo encontro entre o arquiteto e o time de revisão, serão checadas as decisões e as soluções escolhidas para serem implementadas. Eventualmente, algum refinamento pode ser realizado ainda neste passo para melhoria da solução.

Enfim, vale ressaltar que a realização da avaliação arquitetural pode trazer reais benefícios para a construção de um sistema de software, seja qual for a linguagem, plataforma ou tipo de negócio do cliente.

Esses benefícios vão desde uma melhor compreensão dos requisitos não funcionais por parte da equipe de desenvolvimento, até uma redução de custos do projeto e uma melhoria da qualidade do sistema.

A partir do método DCAR, a realização da avaliação arquitetural pode ser feita de forma simples, rápida e sem altos custos de execução. Estatísticas apontam que a avaliação de um sistema com 600 mil linhas de código pode ser feita por uma equipe com oito pessoas consumindo em média 40 horas por integrante. Com base nestes números não é difícil convencer o cliente ou o patrocinador do projeto a realizar a avaliação da arquitetura durante ciclo de vida do sistema, tendo em vista os resultados que podem ser alcançados.

Além disso, a avaliação auxilia na criação indolor da documentação da arquitetura, que é de grande importância para o arquiteto quando o mesmo for questionado sobre as decisões tomadas no passado e também o ajudará a implementar soluções no futuro para atender a novas necessidades de negócio.

## Autor



**Regis Santos**

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

Desenvolvedor Java com mais de 10 anos de experiência e pós-graduação em Arquitetura de Soluções. Possui certificações na plataforma Java (SCJP, SCWCD e SCBCD).



## Links:

**Decision-Centric Architecture Evaluation.**

<http://www.dcar-evaluation.com/>

**Artigo Decision-Centric Architecture Reviews**

<http://www.cs.rug.nl/paris/papers/IIEEESW14b.pdf>

## Você gostou deste artigo?

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



## Supporte 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

# DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

## FORMAÇÃO DESENVOLVEDOR JAVA

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

### CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer

