



Edição 158 :: R\$ 14,90

 DEVMEDIA



**Aplicações reativas em Java**  
**Primeiros passos com o Undertow**

**Modularização dinâmica com OSGi**

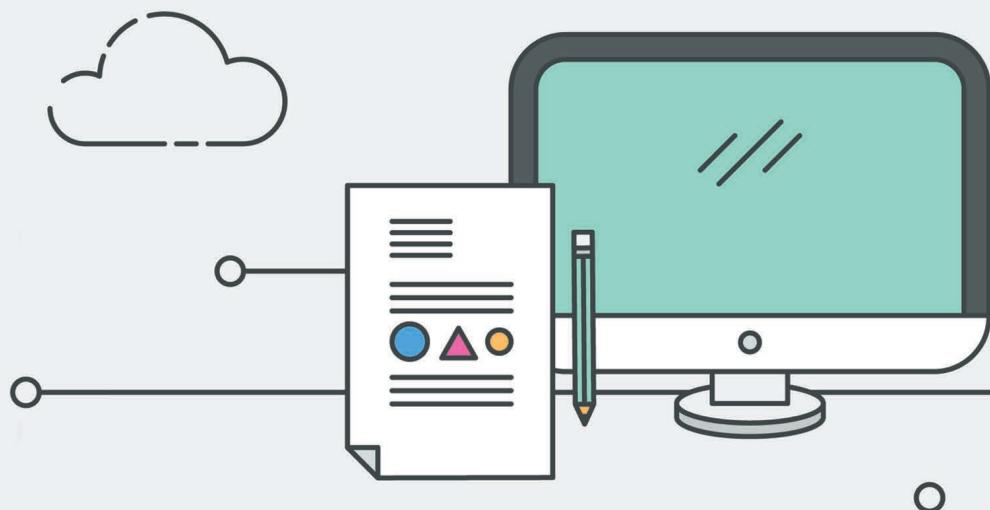
Conheça os benefícios que uma arquitetura modular pode oferecer

**Como tratar exceções na linguagem Java**

Desenvolva soluções que tratem suas próprias exceções

# JAVA EE 7

Aprenda a criar listagens com paginação por demanda



# 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 158 • 2018 • 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))

**Consultora Técnica** Anny Caroline ([annycarolinegnr@gmail.com](mailto:annycarolinegnr@gmail.com))

### Produção

**Jornalista Responsável** Kaline Dolabella - JP24185

**Capa e Diagramação** Romulo Araújo

### 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

# Sumário

### Conteúdo sobre Boas Práticas

#### 04 – Como tratar exceções na linguagem Java

[ José Fernandes A. Júnior ]

**Destaque – Mentoring** | **Artigo do tipo Mentoring**

#### 12 – Java EE 7: Como recuperar listagens sob demanda

[ Pablo Bruno de Moura Nóbrega e Joel Xavier Rocha ]

**Destaque – Vanguarda** | **Conteúdo sobre Novidades**

#### 27 – Construindo aplicações reativas com Undertow

[ Joel Backschat ]

### Artigo no estilo Solução Completa

#### 38 – Modularização dinâmica em Java com OSGi

[ André Luiz Martins Fabbro ]

# Como tratar exceções na linguagem Java

Aprenda o que é o mecanismo de exceções do Java, conheça as suas categorias e saiba como desenvolver programas que consigam tratar suas próprias exceções

Qualquer condição que interrompa o fluxo normal de um programa em execução é um erro ou uma exceção. Para compreender melhor esses dois termos, é importante, desde já, saber diferenciá-los. Um erro significa um problema no código ou em um arquivo de configuração que deve ser corrigido para que seja solucionado. Enquanto não houver correção, o problema voltará a acontecer sempre nas mesmas linhas. Alguns exemplos de erros são:

- Tentativa de conexão a uma base de dados inexistente;
- Acessar variáveis não estáticas a partir de métodos estáticos.

Por sua vez, uma exceção (ou evento excepcional) não está necessariamente relacionada com o código do programa, pode estar relacionada com condições externas que impedem o funcionamento normal da aplicação. Nesse caso, a criação do tratamento de exceções não é necessária para o correto funcionamento do programa, basta que a condição (ou condições) externa(s) que provoca(m) a exceção deixe(m) de existir. Entretanto, não queremos permitir que o programa termine inesperadamente, por isso podemos escrever código para lidar com as exceções e continuar o seu fluxo normal de execução. Alguns exemplos de exceções são:

- A base de dados com a qual o programa tenta se conectar está offline;
- Manipulação de operandos fora dos limites dos seus intervalos.

## Hierarquia de exceções

Todas as classes de exceções são subtipos da classe `java.lang.Exception`. Por sua vez, `Exception` é uma

## Fique por dentro

Durante a execução de um programa podemos ser surpreendidos por eventos inesperados. Quando esse tipo de evento acontece, o fluxo normal do programa é interrompido e o programa/aplicação é finalizado. Diante disso, para escrever um bom programa é preciso incluir um bom gestor de erros e recuperação.

O mecanismo de exceções da linguagem Java fornece uma abordagem simples e organizada para isso. Assim, ao invés de deixarmos o programa terminar devido a situações inesperadas, podemos escrever código para lidar com essas exceções e continuar a execução do programa normalmente.

Neste artigo, vamos conhecer as várias categorias de exceções da linguagem Java, aprender a utilizar o mecanismo de exceções que temos disponível para tratá-las, analisar a hierarquia de exceções e entender a importância das informações contidas no stack trace gerado por uma exceção.

subclasse de `Throwable`, assim como a classe `Error`. A **Figura 1** nos mostra a hierarquia das exceções em Java.

A classe `Error` é um caso especial e distinto das exceções, já que os erros são condições anormais que ocorrem em caso de falhas graves do sistema. Voltaremos a falar sobre esse assunto mais à frente.

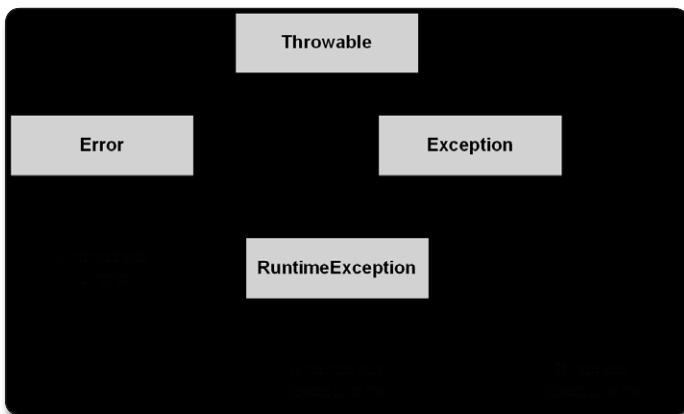
## Os três tipos de exceções

Para ajudar na organização das exceções, elas foram divididas em três categorias. É essencial entender essa divisão para compreender como o tratamento de exceções funciona na linguagem Java.

### *Checked exceptions*

O primeiro tipo de exceção é a checked exception. Uma checked exception é uma exceção que ocorre em tempo de compilação,

por isso também é chamada de exceção de tempo de compilação. Essas exceções não podem ser ignoradas, o programador deve, obrigatoriamente, tratar todas as exceções desse tipo, caso contrário é gerado um erro de compilação. A **Listagem 1** mostra um exemplo de checked exception.



**Figura 1.** Hierarquia das exceções na linguagem Java. Fonte: <http://theprogrammingsintroduction.blogspot.com.br/2014/07/what-is-exception-handling-in-java.html>

**Listagem 1.** Exemplo de checked exception.

```

1 public static void save(String filename, Object object) {
2   // Writing data...
3   ObjectOutputStream oos = null;
4
5   File file = new File(filename);
6   System.out.println("Path:" + file.getAbsolutePath());
7   oos = new ObjectOutputStream(new FileOutputStream(file));
8   System.out.println("Nome do arquivo:" + filename);
9   oos.writeObject(object);
10 }

-- Erros de compilação:
Unhandled exception type FileNotFoundException TestClassException.java
/TestProject line 7 Java Problem
Unhandled exception type IOException TestClassException.java /TestProject
line 7 Java Problem
Unhandled exception type IOException TestClassException.java /TestProject
line 9 Java Problem
  
```

Como podemos observar, na linha 7 tivemos dois erros de compilação, provenientes de duas exceções não tratadas: **FileNotFoundException** e **IOException**. Da mesma forma, na linha 9 temos um erro de compilação devido à exceção **IOException**, que, como dito, não foi tratada.

Conseguimos determinar as exceções que um método lança ao navegar pela API do Java, já que na assinatura do método temos obrigatoriamente que dizer que tipo de exceção (*checked exception*) ele pode lançar, através da instrução **throws**. Na **Listagem 2** podemos ver a assinatura do construtor **ObjectOutputStream** e a exceção que esse construtor pode lançar.

Como podemos notar, a classe **ObjectOutputStream** lança a checked exception **IOException**, daí um dos erros gerados na linha 7 do código da **Listagem 1**.

**Listagem 2.** Assinatura do construtor **ObjectOutputStream**.

```

1 public ObjectOutputStream(OutputStream out) throws IOException {
2   ...
  
```

**Listagem 3.** Assinatura do construtor **FileOutputStream**.

```

1 public FileOutputStream(File file) throws FileNotFoundException {
2   ...
  
```

Do mesmo modo, podemos verificar na **Listagem 3** que o construtor **FileOutputStream** lança a checked exception **FileNotFoundException**, o que explica o outro erro gerado na compilação da linha 7. Finalmente, como você poderá identificar na API da classe **ObjectOutputStream**, o método **writeObject(Object obj)** tem a instrução **throws IOException** em sua assinatura, o que provoca a mensagem de erro da linha 9 (devido à falta do código de tratamento para exceções desse tipo).

### Unchecked exceptions

Uma unchecked exception é uma exceção que ocorre em tempo de execução, por isso também é conhecida como exceção de runtime ou runtime exception. Esse tipo inclui bugs de programação, como erros lógicos, ou uso impróprio da API. Ao contrário das checked exceptions, uma unchecked exception não precisa (ou não deve) ser tratada e não gera erro de compilação. Por essa razão, um método que possa lançar um unchecked exception não precisa ter essa informação (**throws**) em sua assinatura. A **Listagem 4** mostra um exemplo simples de código que gera uma exceção de runtime.

Como podemos ver no exemplo apresentado, o resultado da iteração sobre o array de strings gerou uma exceção do tipo **ArrayIndexOutOfBoundsException**, que, como o próprio nome diz, significa que tentamos acessar uma posição fora dos limites do array.

**Listagem 4.** Código Java que gera uma exceção de runtime e seu respectivo output.

```

1 public class TestClassException {
2
3   public static void testException() {
4     String[] stringArray = {"Hello", "World", "Greetings"};
5
6     for (int i = 0; i < 4; i++) {
7       System.out.println(stringArray[i]);
8     }
9   }
10
11  public static void main(String[] args) {
12    testException();
13  }
14 }

-- Output:
Hello
World
Greetings
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
at TestClassException.testException(TestClassException.java:7)
at TestClassException.main(TestClassException.java:12)
  
```

# Como tratar exceções na linguagem Java

O stack trace da exceção contém informações extremamente úteis e detalhadas sobre o erro. Olhando para o output da **Listagem 4**, constatamos que a thread principal do programa tentou acessar a posição três do array de strings (**Exception in thread "main"** **java.lang.ArrayIndexOutOfBoundsException: 3**) e que o código que gerou a exceção é o que está na linha 7 da classe **TestClassException** (**at TestClassException.testException(TestClassException.java:7)**), que por sua vez foi chamado na linha 12 da mesma classe (**at TestClassException.main(TestClassException.java:12)**). Olhando para o código, podemos ver que o array de strings contém três posições: a posição 0 ("Hello"), a posição 1 ("World") e a posição 2 ("Greetings"). Quando chamamos o método **println()** para uma posição inexistente do array (posição 3), temos uma exceção.

## Errors

Como podemos verificar na **Figura 1**, um "error", na verdade, é um **throwable**, e não exatamente uma exceção. Os erros são condições anormais que ocorrem em caso de falhas graves do sistema. Eles geralmente não são e nem devem ser tratados pelos programas.

Os objetos de erros são criados para indicar erros gerados pelo ambiente em tempo de execução. Por exemplo, quando a Java Virtual Machine fica sem memória (quer seja Java heap space ou PermGen space), é criado e lançado o objeto de erro **java.lang.OutOfMemoryError**. Normalmente, os programas não podem se recuperar de erros.

## Tratamento de exceções

No caso de uma exceção acontecer e não ser tratada, o programa irá terminar com uma mensagem de erro. O tratamento de exceções (*exception handling*) permite ao programador capturar exceções e tratá-las sem interromper o fluxo normal de execução do programa. Esses casos excepcionais, quando ocorrem, são tratados em blocos de código separados, associados ao código da execução normal do programa, o que produz um código mais limpo, legível e de mais fácil manutenção.

### As instruções try e catch e finally

A linguagem de programação Java fornece um mecanismo que permite ao programa descobrir que tipo de exceção foi lançado e se recuperar da mesma. Para tratar exceções, podemos colocar as linhas de código que podem gerar uma exceção dentro de um bloco **try** e criar uma lista de blocos **catch** contíguos, um para cada exceção possível de ser lançada. As linhas de código de um bloco **catch** serão executadas se a exceção gerada for do mesmo tipo da listada nesse bloco de captura. Podem existir múltiplos blocos **catch** depois de um bloco **try**, cada um tratando uma exceção diferente.

A instrução **finally** define um bloco de código que é sempre executado, quer tenha sido lançada uma exceção, ou não. A **Listagem 5** mostra o código da **Listagem 1** novamente, mas dessa vez implementando o tratamento de exceções de forma que o código seja compilável.

**Listagem 5.** Tratamento de exceções com as instruções try/catch.

```
1 public static void save(String filename, Object object) throws IOException {
2     // Writing data...
3     ObjectOutputStream oos = null;
4     try {
5         File file = new File(filename);
6         System.out.println("Path: " + file.getAbsolutePath());
7         oos = new ObjectOutputStream(new FileOutputStream(file));
8         System.out.println("Nome do arquivo: " + filename);
9         oos.writeObject(object);
10    }
11    catch (FileNotFoundException e) {
12        _logger.error("File not found:" + filename + "\nMessage:" + e.getMessage());
13        throw e;
14    }
15    catch (IOException e) {
16        _logger.error("Erro while trying to save an object:" + e.getMessage());
17        throw e;
18    }
19    finally {
20        if (oos != null) {
21            try {
22                oos.close();
23            }
24            catch (Exception e) {
25                // Do nothing!
26            }
27        }
28    }
29 }
```

Como podemos notar, temos dois tratamentos distintos para dois tipos de exceções: um tratamento para resolver possíveis problemas com o arquivo não encontrado e outro para resolver possíveis problemas de input/output. No caso de o arquivo não ser encontrado (**FileNotFoundException**), vamos informar, no log, que o arquivo de nome "filename" não foi encontrado, e adicionar uma informação mais detalhada sobre o erro, usando o método **e.getMessage()**, disponível para objetos de exceção. Em seguida, lançamos o erro para quem chamou o método, já que ainda não sabemos o que o cliente (chamador) quer fazer no caso de não conseguir executar o método "save" com sucesso.

No caso de termos um erro de input/output (**IOException**), faremos basicamente a mesma coisa: inserimos informações no arquivo de log sobre a falha na escrita no filesystem e lançamos a informação para o método cliente para que ele tome alguma ação.

É importante entender que, para que possamos lançar a exceção de volta ao método cliente, temos que dizer explicitamente na assinatura do método quais exceções podemos lançar (apenas para checked exceptions), utilizando para isso a instrução **throws**, como podemos ver na linha 1 da **Listagem 5**; caso contrário teremos erros de compilação do tipo "unhandled exception", exatamente como vimos nos erros encontrados na **Listagem 1**. Devemos notar que o código da **Listagem 5** informa que o método **save()** pode lançar apenas exceções do tipo **IOException** (**throws IOException**) e não diz nada sobre exceções do tipo

**FileNotFoundException**, mas que mesmo assim não temos erros de compilação. Isso acontece porque as classes de exceção, como todas as classes do Java, estão organizadas de forma hierárquica, e a exceção **FileNotFoundException** é um subtipo da **IOException**, como nos mostra a **Figura 2**.

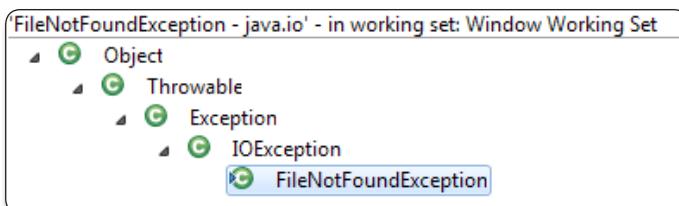


Figura 2. Hierarquia das exceções na linguagem Java

Do mesmo modo, sendo a **IOException** um subtipo da **Exception**, podemos ter a instrução **throws Exception** e até mesmo **throws Throwable**, mas não podemos ter a instrução **throws Object**, pois apenas classes e subclasses **Throwable** podem ser lançadas. Mesmo sendo possível lançar exceções de tipos mais genéricos, devemos sempre informar o tipo mais específico possível; neste caso, **IOException**.

Seguindo o mesmo princípio e sabendo que podemos usar uma classe sempre que uma subclasse é esperada, podemos capturar grupos de exceções e tratá-las com um mesmo **catch**. Por exemplo, usando ainda o código da **Listagem 5** e com a informação da **Figura 2**, podemos escrever apenas o código **catch (IOException e)**, da linha 15, para capturar exceções do tipo **IOException** e **FileNotFoundException** (visto que a segunda é um subtipo da primeira). Na verdade, a captura da exceção **FileNotFoundException** é opcional e serve para agregar informação à exceção capturada.

Sabendo disso, temos mais uma regra importante: a ordem com que as exceções são capturadas é importante. Se eu posso capturar exceções mais genéricas na árvore hierárquica das classes de exceção, então tenho que começar a capturar da exceção mais específica para a mais genérica. No caso de alterarmos a ordem de captura das exceções no exemplo demonstrado na **Listagem 5**, capturando primeiro a exceção mais genérica, teríamos um erro de compilação nos dizendo que o código contido no bloco de captura da exceção mais específica nunca será executado devido à exceção já ser tratada no bloco de tratamento da exceção mais genérica, como nos mostra a **Listagem 6**.

Avançando ainda mais na hierarquia de classes, podemos usar a instrução **catch (Exception e)** para capturar todas as exceções que podem ocorrer num bloco de código. Dessa forma, o código seria o que nos mostra a **Listagem 7**.

No entanto, por que devemos evitar esse tipo de aproximação? Porque se capturarmos exceções do tipo **Exception**, capturaremos, também, “unchecked exceptions”, como nos mostra a árvore hierárquica da **Figura 1**, o que não seria uma boa ideia já que capturariamos possíveis exceções de runtime. Além do mais, perderíamos as informações de erros por arquivo não encontrado e o nome do

arquivo, o que nos dificultaria na hora de procurar a causa do erro. Então, para reforçar o pensamento, quanto mais específica for a exceção capturada e quanto mais informação tivermos sobre o problema, melhor! Assim, vamos preferir o código da **Listagem 5** ao código da **Listagem 7** sempre que precisarmos de informações específicas das exceções. Caso contrário, se a informação contida no objeto de exceção não for importante, podemos agrupar o tratamento das exceções utilizando uma classe mais genérica.

**Listagem 6.** Exemplo demonstrando que a ordem de captura das exceções é importante.

```

1 catch (IOException e) {
2   _logger.error("Erro while trying to save an object:" + e.getMessage());
3   throw e;
4 }
5 catch (FileNotFoundException e) {
6   _logger.error("File not found:" + filename + "\nMessage:" + e.getMessage());
7   throw e;
8 }

-- Erro de compilação:
Unreachable catch block for FileNotFoundException. It is already handled by the
catch block for IOException.
  
```

**Listagem 7.** Tratamento de exceções com catch genérico.

```

1 public static void save(String filename, Object object) throws Exception {
2   //Writing data...
3   ObjectOutputStream oos = null;
4   try {
5     File file = new File(filename);
6     System.out.println("Path:" + file.getAbsolutePath());
7     oos = new ObjectOutputStream(new FileOutputStream(file));
8     System.out.println("Nome do arquivo:" + filename);
9     oos.writeObject(object);
10  }
11  catch (Exception e) {
12    _logger.error("Erro while trying to save an object:" + e.getMessage());
13    throw e;
14  }
15  finally {
16    ...
17  }
18  }
  
```

### Agrupamento de exceções sem generalização de classes

A partir da versão 7 do Java, foi introduzida uma nova característica à linguagem que nos permite agrupar o tratamento de exceções sem generalizar. Suponha que estamos lidando com arquivos e bases de dados e que, no nosso método, não interessa se o erro for de input/output ou de base de dados, qualquer um deles terá a mesma influência no resultado final. Além disso, não queremos capturar possíveis exceções de runtime, ou seja, não queremos generalizar e capturar exceções do tipo **Exception**. Para isso, podemos capturar mais de um tipo de exceção sem dependências hierárquicas num único bloco de tratamento de exceções. A **Listagem 8** ilustra o exemplo citado.

Como podemos reparar, a instrução **catch** na captura da exceção especifica os tipos das exceções que o bloco pode tratar, separando cada tipo de exceção com uma barra vertical “|”.

**Listagem 8.** Agrupamento de exceções sem generalização de classes.

```
1 catch (IOException|SQLException e){  
2     logger.error("Exception occurred:" + e.getMessage());  
3     throw e;  
4 }
```

Esse recurso da linguagem ajuda a reduzir a duplicação de código e a diminuir a tentação de se tratar uma exceção muito genérica e abrangente.

## Bloco finally

Agora, vamos olhar para o bloco da instrução **finally** da **Listagem 5**. Como vimos anteriormente, a instrução **finally** define um bloco de código que é sempre executado. Sendo assim, esse bloco foi criado para tentar garantir que o objeto **oos** (do tipo **ObjectOutputStream**) seja sempre fechado no fim da execução do método, quer tenha sido lançada uma exceção ou não. É importante notar que esse bloco contém o seu próprio tratamento de exceções. Isso acontece porque o método **close()**, que podemos ver na linha 22 da **Listagem 5**, da classe **ObjectOutputStream**, pode lançar uma exceção do tipo **IOException**. Isso aconteceria caso a chamada **new FileOutputStream(file)**, da linha 7, lançasse uma exceção do tipo **FileNotFoundException**, o que acionaria o código do bloco **catch (FileNotFoundException e)** da linha 11 e, posteriormente, o bloco de código definido no **finally** da linha 19. Como a exceção foi lançada antes da instrução **oos = new ObjectOutputStream** ter sido executada, o bloco **finally** geraria uma **IOException** na tentativa de fechar o objeto com a instrução **oos.close()** visto que o objeto **oos** seria nulo.

A fim de garantirmos que esse caso não aconteça, inserimos a instrução **if (oos != null)**. Ora, de qualquer forma essa exceção para nós é irrelevante, já que apenas teria a informação que tentamos fechar um objeto que não existe, quando o que importa é a informação que diz que o arquivo “filename” não existe. Não queremos manter qualquer informação de alguma possível exceção que possa acontecer no bloco **finally** e muito menos passá-la para a stack de chamadas. Por isso, no nosso tratamento de exceções, tratamos qualquer exceção com a instrução **catch (Exception e)** e não fazemos nada, sem deixar de garantir a correta execução do programa. Vale citar que é muito importante para a manutenção e legibilidade do código que tenhamos uma nota dizendo explicitamente que o bloco **catch** não vai executar nenhuma operação para esse caso. Isso é demonstrado na linha 25 da **Listagem 5** com a nota **// Do nothing!**.

Como o bloco de código contido na instrução **finally** é sempre executado, é uma boa prática de programação termos todo o código de limpeza do método situado nesse bloco. O **finally** é uma ferramenta fundamental para a prevenção de vazamentos de recursos.

## A instrução try-with-resources

Vamos estudar agora mais um “atalho” criado pela linguagem Java para facilitar nossa tarefa de manter um código limpo e livre

de vazamento de recursos. A instrução **try-with-resources** é uma instrução **try** que declara um ou mais recursos, sendo um recurso um objeto que deve ser fechado e liberado após o programa ter terminado o seu uso. Ao contrário da instrução **try**, essa nova instrução garante que cada recurso seja fechado no final das instruções contidas em seu bloco sem que tenhamos que escrever código para isso.

**Listagem 9.** Utilização da instrução try-catch-finally.

```
1 public static String readFirstLineFromFile(String path) throws IOException {  
2     BufferedReader br = new BufferedReader(new FileReader(path));  
3     try {  
4         return br.readLine();  
5     }  
6     finally {  
7         if (br != null) {  
8             try {  
9                 br.close();  
10            }  
11            catch (Exception e) {  
12                // Do nothing!  
13            }  
14        }  
15    }  
16}
```

Qualquer objeto que implementa **java.lang.AutoCloseable**, o que inclui todos os objetos que implementam **java.io.Closeable**, pode ser utilizado como um recurso. Para obter a lista de classes que implementam essas interfaces, consulte o Javadoc.

Vejamos um exemplo simples de como essa instrução funciona. Primeiro, utilizaremos o método tradicional **try-catch-finally** e em seguida produziremos o mesmo resultado com a instrução **try-with-resources** a fim de compararmos a facilidade de uso de uma em relação à outra. Vamos usar o objeto **BufferedReader**, que estende a classe **Reader**, que, por sua vez, implementa a interface **Closeable**. A **Listagem 9** mostra o código com esse exemplo.

Esse código lê a primeira linha de um arquivo utilizando um buffer de leitura (**BufferedReader**) construído sobre um leitor de arquivos (**FileReader**, linha 2 do código). Nesse caso, o objeto **BufferedReader** é um recurso que deve ser fechado assim que o método terminar de utilizá-lo, caso contrário poderemos ter vazamento de memória do sistema. Isso acontece porque o Garbage Collector não tem como liberar os recursos de objetos referenciados ou abertos, pois isso indica que eles ainda podem estar sendo utilizados pelo programa. O fechamento é feito no bloco **finally**, na chamada do método **close()**, que, por sua vez, tem o seu próprio tratamento de exceções, como explicado nos exemplos anteriores.

Vamos agora utilizar a instrução **try-with-resources** e ver como fica a implementação desse mesmo método. A **Listagem 10** apresenta o código em sua nova implementação.

A primeira diferença que podemos notar é que não temos mais de escrever o bloco **finally**. Isso porque, como vimos anteriormente, a instrução **try-with-resources** garante que todos os recursos declarados no bloco **try** sejam fechados no final da sua utilização.

Depois, em termos de linhas de código, diminuímos de 16 para cinco, o que nos permitiu criar um código aproximadamente três vezes menor do que o inicial. Podemos, ainda, declarar vários recursos de uma só vez, bastando separá-los por ponto e vírgula. A **Listagem 11** demonstra um exemplo.

**Listagem 10.** Utilização da instrução try-with-resources.

```
1 public static String readFirstLineFromFile(String path) throws IOException {
2     try (BufferedReader br = new BufferedReader(new FileReader(path))) {
3         return br.readLine();
4     }
5 }
```

**Listagem 11.** Exemplo de instanciação de mais de um recurso na instrução try-with-resources.

```
1 public static String readFirstLineFromFile(String zipFileName, String path)
throws IOException {
2     try (ZipFile zf = new ZipFile(zipFileName);
3          BufferedReader br = new BufferedReader(new FileReader(path))) {
4         ...
5     }
6 }
```

Nesse caso, os recursos **ZipFile** e **BufferedReader** serão fechados no fim do método, garantindo a limpeza de recursos do sistema sem termos de nos preocupar com isso. Dessa maneira, a instrução **try-with-resources** pode nos poupar muito trabalho de codificação, ajudando na criação de um código mais enxuto e de fácil manutenção.

## Quando devemos tratar a exceção e quando devemos lançá-la?

Nos exemplos vistos até aqui, implementamos códigos de tratamento de exceções utilizando as informações contidas nas mesmas, mas continuamos enviando esses mesmos objetos de exceções para o cliente, ou seja, para o método que chamou o método que gerou a exceção. Diante disso, quando devemos parar a pesquisa na pilha de chamadas para tratar as exceções levantadas de fato?

Vamos tomar como exemplo o código da **Listagem 5**, onde temos o método **save(String filename, Object object)**, que guarda o objeto **object** no arquivo indicado em **filename**. Esse tipo de método auxiliar do filesystem deve ser o mais genérico possível para que possa ser partilhado por todo código que queira guardar informações no filesystem. Sendo genérico, o método **save()** não tem como descobrir qual a relevância que o método que o chama dá à correta persistência do objeto enviado. Em outras palavras, persistir corretamente o objeto é algo que pode falhar algumas vezes, que é importante ou que é crítico para o método cliente? Para um método, pode ser crítico a ponto de não fazer sentido continuar a execução do programa se houver uma falha. Para outro, pode ser moderado, podendo ele tentar guardar novamente o objeto mais tarde. Para um terceiro método, ainda, esse problema pode ser ignorado, e a execução programar continuar normalmente. Sendo assim, o método **save()** informa ao usuário, através do log de erro, que houve um problema e passa o problema “para cima”, lançando-o para o cliente, que deve saber o que fazer.

Como exemplo de um possível cliente para o método **save()**, criamos um listener de uma aplicação em Java Swing que persiste arquivos de configuração de bases de dados de forma que essas configurações não sejam perdidas depois que a aplicação é fechada. A **Listagem 12** mostra esse código.

**Listagem 12.** Demonstração de código cliente do método **save()**.

```
1 public class ActionListenerSaveDatabasesSettings implements ActionListener {
2     ...
3     @Override
4     public void actionPerformed(ActionEvent e) {
5         _logger.info("Saving database settings...");
6         ...
7         SettingsDatabasesDTO settings =
8             new SettingsDatabasesDTO(databaseX, databaseY, databaseZ);
9
10        try {
11            FileSystemUtils.save(Constants.SETTINGS_FILE_PATH_DATABASE, settings);
12            JOptionPane.showMessageDialog(_tabPanelSettings,
13                Resources.INSTANCE.getString("settings_databases_saved"),
14                Resources.INSTANCE.getString("InfoDialog.title"),
15                JOptionPane.INFORMATION_MESSAGE);
16
17            _logger.info("Environments saved!");
18        }
19        catch (IOException e1) {
20            JOptionPane.showMessageDialog(_tabPanelSettings,
21                Resources.INSTANCE.getString("error_save_environment"),
22                Resources.INSTANCE.getString("error"),
23                JOptionPane.ERROR_MESSAGE);
24
25            _logger.error("Error while trying to save the environments!");
26        }
27    }
28 }
```

Como podemos verificar, na linha 11 do código é feita uma chamada ao método **save()**, dentro de um bloco **try**. Se a chamada for executada com sucesso, o método cria uma caixa de diálogos que avisa ao usuário que as informações foram guardadas com sucesso. Caso o método **save()** lance alguma exceção, o bloco **catch** trata a mesma e mostra uma caixa de diálogos com uma mensagem de erro informando ao usuário que não foi possível guardar a informação. O usuário, por sua vez, deverá tentar novamente mais tarde e pode olhar o log para conseguir mais detalhes sobre o erro. De qualquer forma, a exceção é tratada e resolvida, e o programa continua sua execução normal, sem interrupções.

Imaginemos agora um cliente onde a execução do método **save()** é essencial para a continuidade do programa. No caso de ser lançada uma exceção, o método poderia esperar algum tempo e tentar novamente até conseguir, ou poderia tentar certo número de vezes antes de desistir, informando ao usuário que a operação não executou com sucesso. O importante é que, mesmo que não seja possível completar a operação, o programa possa terminar normalmente.

A decisão de onde a exceção deve ser tratada deve ser sempre a que faça com que o código seja o mais reutilizável possível. Se a exceção fosse tratada e terminada no próprio método **save()**, os dois exemplos anteriores não poderiam chamar o mesmo método,

visto que um, qualquer que seja o resultado, chama o método, informa o cliente do resultado e sai, e o outro sai apenas se o método completar com sucesso, caso contrário continua tentando durante um certo número de vezes antes de sair. E por que lançar a exceção e não apenas retornar um erro? Porque a exceção contém informações detalhadas sobre o problema, que podem ou não serem utilizadas pelo cliente, quer seja apenas para informar ao usuário, quer seja para resolver o problema.

## Criando sua própria exceção

Como já vimos, uma exceção é uma classe Java como qualquer outra, com a particularidade de estender a interface `Exception`. Isso quer dizer que podemos escrever nossas próprias classes de exceção e instanciá-las da mesma maneira que criamos e instanciamos outros objetos. A primeira coisa que precisamos saber antes de criar a nossa própria exceção é se já existe alguma na biblioteca da linguagem que nos forneça o que precisamos (não queremos reinventar a roda). Depois, é importante saber quando criar uma exceção e quando apenas enviar um retorno com erro. Essa é uma grande discussão e que não tem uma única resposta.

A melhor dica é: represente uma exceção apenas se o acontecido for exatamente isso, uma exceção! O que pretendemos dizer com isso é que a linguagem Java é uma linguagem orientada a objetos, onde os objetos fazem a representação de coisas e acontecimentos reais. E o que é uma exceção? Como vimos, uma exceção é um evento excepcional, um desvio da regra geral, algo inesperado. Vamos colocar, então, alguns problemas e tentar entender onde devemos e onde não devemos criar uma exceção.

Suponha que trabalhamos para uma empresa onde temos um programa que valida a idade dos seus empregados. Para a construção do método de validação de idades, foram passadas as seguintes regras:

1. A idade do empregado será considerada válida se ele tiver mais de 18 anos;
2. Empregados com menos de 18 anos devem ser considerados inválidos, pois não são permitidos trabalhadores menores de idade.

Em uma primeira análise, o programador pode se sentir tentado a criar uma exceção para indicar que um trabalhador é menor de idade. A **Listagem 13** mostra o código de criação de uma exceção para esse caso.

**Listagem 13.** Classe de exceção para empregado com idade abaixo de 18 anos.

```
1 public class UnderAgeException extends Exception {  
2     private int _age;  
3  
4     public UnderAgeException(String message, int age) {  
5         super(message);  
6         this._age = age;  
7     }  
8  
9     // Use the getAge method to get the value that caused the exception.  
0     public int getAge() {  
11         return _age;  
12     }  
13 }
```

Como boa prática da linguagem Java, é importante que o nome da classe de exceção termine com a string `Exception`, melhorando a compreensão, legibilidade e manutenibilidade do código.

Como podemos ver pela **Listagem 13**, a exceção `UnderAgeException` estende a classe `Exception` e tem apenas um construtor, que chama o construtor da `Exception` e insere a idade que foi utilizada na criação da exceção. Na **Listagem 14** apresentamos um exemplo de como poderíamos utilizar essa exceção.

**Listagem 14.** Código que valida a idade de um empregado.

```
1 public static void checkEmployeeAge(int age) throws UnderAgeException {  
2     if (age > 18) {  
3         System.out.println("Employee age is OK! Employee has " + age + " years old.");  
4     }  
5     else {  
6         throw new UnderAgeException("Employee can't be under age!", age);  
7     }  
8 }  
9  
10 public static void testeEmployeeAge(int age) {  
11     try {  
12         checkEmployeeAge(age);  
13     }  
14     catch (UnderAgeException e) {  
15         System.out.print("Exception occurred! Error message: " + e.getMessage());  
16         System.out.println(" Employee age: " + e.getAge());  
17     }  
18 }  
19  
20 public static void main(String[] args) {  
21     testeEmployeeAge(28);  
22     testeEmployeeAge(16);  
23 }
```

-- Output:

Employee age is OK! Employee has 28 years old.

Exception occurred! Error message: Employee can't be under age! Employee age: 16

Como podemos notar, o método `checkEmployeeAge()` obedece às regras impostas. Primeiro, valida se a idade do empregado é maior do que 18 anos e, se for, envia uma mensagem dizendo que a idade está OK. Caso contrário, lança uma exceção `UnderAgeException`, que deverá ser tratada pelo método cliente. O método `testEmployeeAge()` — método cliente —, por sua vez, chama o método `checkEmployeeAge()` e trata a exceção `UnderAgeException`. Caso seja lançada uma exceção desse tipo, o tratamento de exceções irá indicar que ocorreu um problema e informar os seus detalhes. Essas informações estão guardadas no próprio objeto de exceção: a mensagem de erro, obtida através do método `e.getMessage()`, e a idade que foi enviada e que gerou a exceção, obtida através do método `e.getAge()`. Conforme podemos observar pelo output, o programa funcionou sem interrupções inesperadas, tendo gerado as mensagens corretas e terminado normalmente.

A solução funcional! Mas, então, qual o problema aqui? Alguns autores, entre eles o autor, não acham uma boa ideia gerar exceções para fazer o que chamamos de “passagem de mensagem”. Como foi dito anteriormente, uma exceção deve ser criada apenas se o acontecimento for exatamente isso, uma exceção.

Na vida real, seria uma exceção termos pessoas registradas com menos de 18 anos? Não! É um caso possível e aceitável. Então, por que simplesmente não dizer que pessoas menores de 18 anos não têm idade válida para trabalhar na empresa e deixar de tratar isso como um erro? Mas aí a pergunta: O que seria um erro provável de gerar uma exceção? Um input inválido! Um input inválido é um erro e, como tal, deve ser investigado. Onde aconteceu? Por quê? Como corrigi-lo?

Com base nisso, vamos refazer os exemplos das **Listagens 13** e **14** e mostrar como podemos aplicar uma melhor regra de criação e tratamento de exceções. A **Listagem 15** mostra uma nova exceção, criada para ser lançada em caso de input inválido.

**Listagem 15.** Classe de exceção para erros no valor da idade.

```

1 public class InvalidAgeException extends Exception {
2     private int _age;
3
4     public InvalidAgeException(String message, int age) {
5         super(message);
6         this._age = age;
7     }
8
9     // Use the getAge method to get the value that caused the exception.
10    public int getAge() {
11        return _age;
12    }
13 }
```

**Listagem 16.** Exemplo de uso da exceção InvalidAgeException.

```

1 public static boolean checkEmployeeAge(int age) throws InvalidAgeException {
2     if (age < 0) {
3         throw new InvalidAgeException("Input error! Age can't be less than zero!", age);
4     }
5     if (age < 18) {
6         return false;
7     }
8     else {
9         return true;
10    }
11 }
12
13 public static void testeEmployeeAge(int age) {
14     try {
15         if (checkEmployeeAge(age)) {
16             System.out.println("Employee age is OK! Employee has " + age + " years old");
17         }
18         else {
19             System.out.println("Employee is under age! Employee has " + age +
20                 " years old");
21         }
22     catch (InvalidAgeException e) {
23         System.out.print("Exception occurred! Error message: " + e.getMessage());
24         System.out.println(" Age value: " + e.getAge());
25     }
26 }
27
28 public static void main(String[] args) {
29     testeEmployeeAge(28);
30     testeEmployeeAge(16);
31     testeEmployeeAge(-2);
32 }

-- Output:
Employee age is OK! Employee has 28 years old.
Employee is under age! Employee has 16 years old.
Exception occurred! Error message: Input error! Age can't be less than zero! Age value: -2
```

A classe **InvalidAgeException** funciona exatamente como **UnderAgeException**: estende a classe **Exception**, possui um construtor que recebe informações sobre o erro e guarda a idade que gerou a exceção. Vejamos agora, na **Listagem 16**, como utilizar essa exceção numa melhor abordagem para a solução do problema proposto.

Como podemos observar, o método **checkEmployeeAge()** considera válida qualquer idade que seja maior ou igual a 0 e considera como um erro um input negativo, lançando a exceção **InvalidAgeException**, pois uma idade negativa nunca deveria ser enviada para ser validada. Isso deve desencadear um problema passível de ser investigado e resolvido. Isso é uma exceção! Mais uma vez, olhando para o output do programa, podemos validar que tudo ocorreu conforme planejado, tendo sido enviadas as mensagens de OK, não OK e erro, sem que o programa termine inesperadamente.

A plataforma Java oferece um mecanismo de tratamento de erros e exceções que nos permite construir um código limpo e organizado graças à sua abordagem simples, onde os tratamentos são separados por blocos de código através das instruções **try**, **catch** e **finally**.

Outra vantagem das exceções é a capacidade para propagar os objetos de exceção na pilha de chamadas dos métodos, ajudando na separação e organização do código e fazendo com que seja mais simples de se escrever código genérico e reutilizável, uma vez que os tratamentos específicos dos erros são feitos nos próprios clientes.

Na maior parte das situações, a melhor abordagem será fazer tratamentos de exceções o mais específico possível, no entanto, haverá situações em que o agrupamento no tratamento de exceções será vantajoso e de mais fácil implementação.

Não devemos esquecer que as exceções são destinadas a tratar casos excepcionais. Usá-las para controlar o fluxo normal de execução do código não só obscurece a intenção do código, como também o torna mais lento, uma vez que o tratamento de exceções é um processo relativamente pesado e que deve ser usado com precaução.

A plataforma Java percorreu um grande caminho para nos ajudar a lidar com condições de erros. O seu mecanismo de tratamento de exceções é uma ferramenta poderosa e simples que, quando bem utilizada, é essencial para desenhar uma boa solução de software.

## Autor



**José Fernandes A. Júnior**

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

Mestre em Engenharia Informática pela Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa (FCT-UNL). Trabalha há 10 anos com engenharia de software em diversos ramos e áreas, de core engines em empresas de telecomunicação a aplicações móveis. Formador autorizado na plataforma Java, SOA, Sistemas Unix e Shell Programming. Autor de artigos no site DevMedia para a revista Java Magazine. Particular interesse em programação para sistemas Android e code refactoring para reestruturação e simplificação de sistemas complexos em sistemas simples. Especializado em integração de sistemas com arquiteturas de soluções distribuídas, web services e programação em Java.



# Java EE 7: Como recuperar listagens sob demanda

Aprenda a apresentar as listagens do seu sistema utilizando paginação por demanda e economize recursos do servidor de aplicação

ESTE ARTIGO É DO TIPO MENTORING

SAIBA MAIS: [WWW.DEVMEDIA.COM.BR/MENTORING-SAIBAMAI](http://WWW.DEVMEDIA.COM.BR/MENTORING-SAIBAMAI)

**E**m geral, as aplicações que desenvolvemos — sejam elas *web*, *desktop* ou *mobile* — necessitam exibir listas de registros em tabelas. Essa é uma situação mais do que comum para a maioria dos programadores. O problema surge quando a quantidade de linhas a ser mostrada é muito grande. Nesses casos, colocar em memória todo o volume de dados retornado do banco pode ser extremamente perigoso, podendo inclusive derrubar a aplicação. Por esse motivo, precisamos tratar com bastante atenção como devemos recuperar o conteúdo a ser exibido para o usuário. Uma boa recomendação é dar uma lida em alguns artigos e discussões na *web* bem relevantes sobre o assunto. Entre essas postagens, podemos destacar “*Avoid the Pains of Pagination*” e “*The Impact of Paging vs. Scrolling on Reading Online Text Passages*” (consulte a seção **Links** para saber onde encontrá-las).

No que se refere à implementação da solução para o cenário descrito anteriormente, a estratégia comumente adotada para exibir dados em tabelas é a primeira que mencionamos: recuperar de uma única vez o conteúdo da base de dados, adicioná-lo em uma coleção e, a partir dessa coleção, permitir que o usuário possa visualizar os dados em uma tabela através de páginas, tendo cada uma delas uma quantidade predefinida de registros. Essa abordagem é chamada de paginação em memória porque os dados já estão todos disponíveis na RAM ou em disco

## Cenário

Imagine que você precisa criar uma aplicação que recupera um grande volume de dados e o exibe no formato de listagens. Para implementar esse requisito, temos duas abordagens: a primeira, considerada por muitos uma má prática, consiste em carregar todos os registros cadastrados no SGBD ao mesmo tempo e colocá-los em memória; já a segunda, bastante recomendada para listagens extensas, é aquela na qual recuperamos do banco de dados apenas os registros que estão sendo visualizados pelo usuário, ou seja, trazemos as informações sob demanda.

De forma intuitiva, sabemos que a segunda solução traz alguns benefícios para qualquer aplicação. Para demonstrar isso, criaremos um pequeno sistema Java explorando inicialmente a listagem em memória de uma grande quantidade de clientes cadastrados no SGBD. Em seguida, mostraremos como implementar a mesma operação utilizando a recuperação por demanda, para podermos comprovar as vantagens da segunda opção através da análise de apontamentos de performance e consumo de recursos

(se a memória virtual estiver sendo usada), mesmo que só uma parte dos registros esteja sendo mostrada. Fica claro, no entanto, que esse cenário é considerado uma má prática, pois pode consumir amplamente os recursos disponíveis do servidor de aplicação.

Para resolver esse problema, utilizamos o que chamamos de paginação por demanda, que consiste em obter da base de dados apenas o conteúdo visível para o usuário, ou seja, os registros que estão sendo mostrados na página. Assim, reduzimos largamente a quantidade de memória consumida e o processamento envolvido na operação, pois estamos criando um número pequeno de objetos por vez.

Neste artigo, veremos as duas implementações de listagem de dados descritas em funcionamento através de um pequeno sistema que iremos criar envolvendo JSF 2.2, CDI 1.1 para injeção de dependências, EJB 3.2 na camada de negócio, PrimeFaces 5.3 como biblioteca de interface rica, Facelets para utilização de *templates* (evitando que dupliquemos código em nossas páginas) e Hibernate 5.0 acessando uma base de dados no PostgreSQL. Além disso, o Maven será configurado para gerenciamento das dependências, e nosso servidor de aplicação será o WildFly 10.0.0. Finalizando o conteúdo, analisaremos, através de gráficos e relatórios gerados pelo VisualVM e pelo pacote de ferramentas do desenvolvedor do Google Chrome, a diferença no consumo de recursos e no tempo de processamento entre as soluções.

Iniciando nosso conteúdo teórico, teremos, neste primeiro momento, uma visão geral dos dois principais *frameworks* envolvidos na solução.

## JavaServer Faces 2.2

O JavaServer Faces (ou simplesmente JSF) é um *framework* orientado a eventos lançado em 2004 que adota o padrão MVC (*Model-View-Controller*) e atualmente se encontra na versão 2.2. É utilizado para a construção da interface com o usuário e, graças aos seus componentes de tela e *controllers* de utilização muito simples (chamados de *Managed Beans*), tem se consolidado como um dos frameworks mais populares entre os desenvolvedores Java. Também contribuem para essa aceitação o fato de ser um produto oficial Java EE e de apresentar um grande envolvimento de usuários na especificação da JSR.

Na versão 2.2 algumas inovações foram lançadas, como:

- Adição de suporte ao escopo *FlowScoped*;
- Maior integração com o CDI;
- Melhoria no suporte a HTML5;
- Proteção contra ataques do tipo CSRF (*Cross-Site Request Forgery*).

## PrimeFaces 5.3

O PrimeFaces é uma biblioteca de código aberto que possui diversos componentes adicionais aos já existentes no JSF. Foi lançado em 2008 por uma empresa turca especializada em desenvolvimento ágil, treinamento e consultoria Java EE chamada PrimeTek. O *framework* conta com uma comunidade bastante ativa e possui uma página de demonstração muito superior às dos concorrentes.

Embora a versão gratuita da biblioteca seja a mais popular, existem ainda dois planos de suporte diferenciados: o PrimeFaces Elite e o PrimeFaces Pro, que oferecem, entre outras coisas, *releases* com melhorias e correções mais constantes que a versão comunitária, acesso a temas exclusivos e um plano aperfeiçoado de correção de bugs.

Na versão 5.3 as seguintes novidades estão disponíveis:

- Mais de 100 correções feitas;
- Lançamento do componente *Signature*, que permite assinar documentos em uma tela *touchscreen*;
- Componentes *Captcha* e *Carousel* reescritos;
- Design responsivo aprimorado;

- Componentes de arrastar e soltar agora compatíveis com dispositivos móveis;
- Acessibilidade aperfeiçoada.

## Criando a aplicação

Como dito anteriormente, faremos uma pequena aplicação para exibir clientes cadastrados em um banco de dados. Nossa objetivo é implementar as duas formas de paginação que comentamos anteriormente (em memória e sob demanda) e comparar os resultados. Para criar o sistema, precisaremos dos seguintes softwares instalados:

- Servidor de aplicação WildFly 10.0.0 Final;
- Eclipse Mars para desenvolvedores Java EE;
- Java Development Kit 1.7;
- SGBD PostgreSQL 9.5.

Após ter configurado todos os softwares necessários, vamos primeiro criar o projeto Maven. Para isso, abra o Eclipse e clique no menu *File > New > Maven Project*. Na janela que surgiu, marque a opção *Create a simple project (skip archetype selection)* e clique em *Next*. Na próxima tela, insira no campo *Group Id* o valor “*br.com.javamagazine*”, no *Artifact Id* o valor “*paginacaojm*”, no *Packaging* o valor “*war*” e no *Name* o valor “*paginacaojm*”. Em seguida, confirme no botão *Finish*. Observe que após esses passos, o Maven criará um projeto que possui uma estrutura semelhante à mostrada na **Tabela 1**.

Diretórios do projeto	Conteúdo
<i>src/main/</i>	
java/	Classes Java.
resources/	Arquivos de propriedades.
webapp/	Conteúdo relacionado à parte web do projeto (páginas, arquivos XML de configuração, etc.).
<i>src/test/</i>	
java/	Classes de testes unitários. No nosso sistema, não terá conteúdo.
resources/	Arquivos de propriedades utilizados pelos testes. No nosso sistema, não terá conteúdo.
<i>target/</i>	Arquivos construídos pelo Maven.
<i>pom.xml</i>	Arquivo POM do Maven.

**Tabela 1.** Estrutura do projeto criado

## Modificando o pom.xml e gerando o web.xml

Para fazer com que nosso projeto funcione corretamente como uma aplicação *web*, vamos gerar o *web.xml*. Assim, clique com o botão direito na raiz do projeto, selecione *Java EE Tools* e clique em *Generate Deployment Descriptor Stub*. Observe que o Eclipse irá criar o arquivo na pasta *src/main/webapp/WEB-INF/*.

Agora, abra esse documento para especificar a página inicial do sistema e configurar o Servlet do JSF como nosso *Front Controller*, deixando o código igual ao da **Listagem 1**.

# Java EE 7: Como recuperar listagens sob demanda

## Listagem 1. Arquivo web.xml do projeto.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">
<display-name>paginacaojm</display-name>
<welcome-file-list>
    <welcome-file>/index.xhtml</welcome-file>
</welcome-file-list>
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
</web-app>
```

Nosso próximo passo será ajustar o *pom.xml* da aplicação. Como utilizaremos várias bibliotecas não fornecidas diretamente pelo JDK, precisaremos configurá-las para que fiquem disponíveis no projeto. Também adicionaremos a informação do compilador do projeto. Assim, modifique o código do arquivo *pom.xml* para ficar semelhante ao mostrado na **Listagem 2**.

- As principais configurações apresentadas nessa listagem são:
- Linhas 2 a 6: 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 raiz de pacotes do sistema. Já o **artifactId** é o nome utilizado para gerar o WAR;
  - Linha 7: informa o nome do projeto;
  - Linhas 8 a 19: configuram a versão 1.7 para o *plugin* de compilação do código fonte do sistema. Dessa forma, informamos ao Eclipse que o JDK 7 será usado e que ele deve constar no *classpath* da aplicação;
  - Linhas 20 a 43: definem todas as dependências utilizadas no projeto;
  - Linhas 21 a 26: informam que as bibliotecas do Java EE 7 usadas no projeto já estão no *classpath*. No nosso caso, são fornecidas pelo WildFly;
  - Linhas 27 a 31: informam a dependência do PrimeFaces 5.3;
  - Linhas 32 a 36: informam a dependência do Apache Tomahawk. A biblioteca será utilizada para salvar o estado dos objetos com escopo de requisição na página de listagem com paginação em memória (veremos mais detalhes adiante);
  - Linhas 37 a 42: informam que a biblioteca do Hibernate 5.0 já se encontra no *classpath*, pois também é fornecida pelo WildFly.

Após modificar o arquivo, 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.

## Listagem 2. Arquivo POM do projeto.

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
02     <modelVersion>4.0.0</modelVersion>
03     <groupId>br.com.javamagazine</groupId>
04     <artifactId>paginacaojm</artifactId>
05     <version>0.0.1-SNAPSHOT</version>
06     <packaging>war</packaging>
07     <name>paginacaojm</name>
08     <build>
09         <plugins>
10             <plugin>
11                 <artifactId>maven-compiler-plugin</artifactId>
12                 <version>2.0.2</version>
13                 <configuration>
14                     <source>1.7</source>
15                     <target>1.7</target>
16                 </configuration>
17             </plugin>
18         </plugins>
19     </build>
20     <dependencies>
21         <dependency>
22             <groupId>javax</groupId>
23             <artifactId>javaee-api</artifactId>
24             <version>7.0</version>
25             <scope>provided</scope>
26         </dependency>
27         <dependency>
28             <groupId>org.primefaces</groupId>
29             <artifactId>primefaces</artifactId>
30             <version>5.3</version>
31         </dependency>
32         <dependency>
33             <groupId>org.apache.myfaces.tomahawk</groupId>
34             <artifactId>tomahawk21</artifactId>
35             <version>1.1.14</version>
36         </dependency>
37         <dependency>
38             <groupId>org.hibernate</groupId>
39             <artifactId>hibernate-core</artifactId>
40             <version>5.0.7.Final</version>
41             <scope>provided</scope>
42         </dependency>
43     </dependencies>
44 </project>
```

## Configurando o banco de dados e o datasource

Com o PostgreSQL executando, crie uma base de dados chamada “*paginacaojm*” e rode o *script* da **Listagem 3** para gerar a tabela *cliente* e inserir quase 40 mil tuplas nela, com o objetivo de já termos como realizar nossas simulações (as linhas com os *inserts* foram omitidas e podem ser encontradas no arquivo completo que está na pasta *src/main/resources* do código-fonte que você pode baixar no site da editora).

Uma vez que já temos a base criada, podemos definir o *datasource* no WildFly. O primeiro passo para realizarmos esse procedimento é baixar o *driver* do PostgreSQL (confira a seção **Links** para saber onde fazer o *download*) e realizar o *deploy* dele como uma aplicação no WildFly para permitir que o JAR seja referenciado. Deste modo, após obter o arquivo, coloque-o dentro do diretório *<wildfly\_home>/standalone/deployments*.

Pronto, agora já podemos definir um novo *datasource*, o que é feito inserindo as linhas expostas na **Listagem 4** dentro da tag **<datasources>** do arquivo *standalone.xml*, localizado no diretório *<wildfly\_home>/standalone/configuration/*. Lembre-se de modificar a URL de conexão, o usuário e a senha pelos definidos na sua instalação do PostgreSQL.

#### **Listagem 3.** Script de criação do banco de dados.

```
01 CREATE TABLE "cliente"
02   id INTEGER NOT NULL,
03   nome VARCHAR(100) NOT NULL,
04   data_nascimento DATE NOT NULL,
05   cpf CHAR(11) NOT NULL,
06   qtd_filhos INTEGER NULL,
07   naturalidade VARCHAR(40) NOT NULL,
08   endereco VARCHAR(80) NOT NULL,
09   numero VARCHAR(10) NOT NULL,
10   complemento VARCHAR(30) NULL,
11   bairro VARCHAR(40) NOT NULL,
12   cep CHAR(9) NOT NULL,
13   cidade VARCHAR(30) NOT NULL,
14   uf CHAR(2) NOT NULL,
15   telefone_fixo VARCHAR(14) NULL,
16   telefone_celular VARCHAR(14) NULL,
17   telefone_comercial VARCHAR(14) NULL,
18   email VARCHAR(60) NOT NULL,
19   PRIMARY KEY (id);
20
21 // inserts omitidos. Baixe o código-fonte do projeto e pegue o script completo
//na pasta src/main/resources
```

#### **Listagem 4.** Configuração do novo datasource a ser adicionado no arquivo *standalone.xml* do WildFly.

```
<datasource jta="false" jndi-name="javajboss/datasources/PaginacaoDS"
  pool-name="PaginacaoDS" enabled="true" use-ccm="false">
  <connection-url>jdbcpostgresql://localhost:5432/paginacaojm</connection-url>
  <driver-class>org.postgresql.Driver</driver-class>
  <driver>postgresql-9.4.1208.jar</driver>
  <security>
    <user-name>postgres</user-name>
    <password>1234</password>
  </security>
</datasource>
```

## Configurando o Hibernate

Para configurarmos o Hibernate em nosso projeto, é necessário primeiro criarmos o XML de configuração da biblioteca. Deste modo, pressione o botão direito em *src/main/resources > New > Folder* e confirme em *Next*. No campo *Folder name* informe “META-INF” e clique em *Finish*. Agora, pressione o botão direito em cima da pasta criada, selecione *New* e depois, *Other*. Na tela apresentada, escolha a opção *XML > XML File* e confirme em *Next*. Na janela seguinte, no campo *File name* informe “persistence.xml” e, então, clique em *Finish*. Executados esses passos, modifique o código gerado automaticamente para ficar igual ao da **Listagem 5**.

Nesse código, podemos verificar que na linha 3 definimos o nome **paginacaoPU** para nossa **PersistenceUnit** e indicamos que o tipo de transação é JTA, o que traz a vantagem de evitar que precisemos manipular as transações manualmente. A linha 4 determina qual será a implementação do **EntityManager** e, como

estamos utilizando o Hibernate em nosso projeto, optamos pela classe **HibernatePersistence**. Na linha 5 configuramos o nome do *datasource* que será referenciado (veja que está igual ao que digitamos no arquivo *standalone.xml*) e na linha 7, o dialeto a ser usado pelo Hibernate (no caso, o do PostgreSQL).

#### **Listagem 5.** Código do arquivo *persistence.xml*.

```
01 <?xml version="1.0" encoding ="UTF-8"?>
02 <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">
03   <persistence-unit name="paginacaoPU" transaction-type="JTA">
04     <provider>org.hibernate.ejb.HibernatePersistence</provider>
05     <jta-data-source>java:jboss/datasources/PaginacaoDS</jta-data-source>
06     <properties>
07       <property name="hibernate.dialect"
08         value="org.hibernate.dialect.PostgreSQLDialect"/>
09       <property name="hibernate.show_sql" value="false"/>
10       <property name="hibernate.format_sql" value="false"/>
11     </properties>
12   </persistence-unit>
13 </persistence>
```

## Criando as classes Java

Agora vamos criar nossas classes Java. No entanto, precisamos gerar primeiro os pacotes do nosso projeto. Esse procedimento deve ser realizado da seguinte forma: clique com o botão direito em cima de *src/main/java* e depois em *New > Package*. Na janela que surge, informe no campo *Name* cada um dos valores a seguir, repetindo o procedimento até ter os quatro pacotes criados:

- **br.com.javamagazine.paginacaojm.domain;**
- **br.com.javamagazine.paginacaojm.ejb;**
- **br.com.javamagazine.paginacaojm.managedbean;**
- **br.com.javamagazine.paginacaojm.paginacao.**

O pacote **domain** será utilizado para colocar a única entidade do sistema, ou seja, a classe que será representada no banco de dados pela tabela *cliente*. O pacote **ejb** será destinado à camada de negócio da aplicação. No nosso caso, teremos apenas a classe **Cliente-EJB**, que possuirá métodos para retornar a listagem de clientes das duas formas (com paginação em memória e com paginação por demanda). No pacote **paginação**, criaremos uma classe chamada **LazyClienteDataModel**, que está relacionada com a paginação do objeto **Cliente**. Essa classe estende **LazyDataModel** (uma classe abstrata do PrimeFaces) e contém toda a lógica de recuperação de dados tomando como base o padrão de projeto conhecido como *Lazy Loading*. Para finalizar, o pacote **managedbean** terá duas classes, responsáveis por chamar a lógica de negócios (nossa EJB) e se comunicar com as páginas de listagem de clientes.

Para criar nossa primeira classe, **Cliente**, clique com o botão direito em cima do pacote **br.com.javamagazine.paginacaojm.domain**, selecione *New* e depois clique em *Class*. No campo *Name* informe “**Cliente**” e confirme em *Finish*. Modifique o código gerado inicialmente pelo Eclipse para ficar igual ao da **Listagem 6** (lembre-se de inserir os **get**s, **set**s, **equals()** e **hashCode()**, que foram omitidos).

# Java EE 7: Como recuperar listagens sob demanda

## Listagem 6. Código da classe Cliente.

```
01 package br.com.javamagazine.paginacaojm.domain;
02
03 import java.util.Date;
04 import javax.persistence.Column;
05 import javax.persistence.Entity;
06 import javax.persistence.Id;
07
08 @Entity
09 public class Cliente {
10     @Id
11     private Integer id;
12     private String nome;
13     @Column(name="data_nascimento")
14     private Date dataNascimento;
15     private String cpf;
16     @Column(name="qtd_filhos")
17     private Integer qtdFilhos;
18     private String naturalidade;
19     private String endereco;
20     private String numero;
21     private String complemento;
22     private String bairro;
23     private String cep;
24     private String cidade;
25     private String uf;
26     @Column(name="telefone_fixo")
27     private String telefoneFixo;
28     @Column(name="telefone_celular")
29     private String telefoneCelular;
30     @Column(name="telefone_comercial")
31     private String telefoneComercial;
32     private String email;
33
34 // gets e sets omitidos
35 // equals() e hashCode() omitidos
36 }
```

Como pode ser visto, a classe **Cliente** conta com exatamente 17 atributos, sendo cada um deles o mapeamento de uma coluna da tabela *cliente*. Os principais trechos desse código são:

- Linha 8: indica que a 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 10: informa que o atributo é a chave primária da tabela e identificador do objeto;
- Linhas 13, 16, 26, 28 e 30: informam alguns atributos sobre a coluna na tabela do banco de dados, como o nome, o tipo de dado, o tamanho máximo de caracteres (para campos que envolvem sequência de caracteres), a precisão (para campos numéricos), entre outros. No nosso caso, usamos para especificar o nome da coluna na tabela.

Agora vamos criar nosso EJB. O nome da classe será **ClienteEJB** e deve ficar no pacote **br.com.javamagazine.paginacaojm.ejb**. Após gerar a classe com o Eclipse, deixe o código igual ao da **Listagem 7** (não confunda as classes que devem ser importadas: **List**, **Iterator** e **Map** são do pacote **java.util**; **SortOrder**, do **org.primefaces.model**; **Order**, do **org.hibernate.criterion**; e **Session**, do **org.hibernate**).

Os principais trechos do código apresentado são:

- Linha 5: define a classe como um EJB Stateless (ver **BOX 1**) através da anotação **@Stateless**;

## BOX 1. EJB Stateless

Trata-se de um EJB que não precisa manter o estado das regras de negócio nas quais ele está envolvido. Por exemplo: um EJB que retorna o endereço baseado em um CEP fornecido não precisa ter informações do último CEP que foi consultado.

- Linhas 7 e 8: a anotação **@PersistenceContext** da JPA injeta a **Session** no EJB a partir dos dados de conexão informados no arquivo *persistence.xml*. Assim, podemos realizar as operações no banco de dados;
- Linha 11: o método **listaClientesEmMemoria()** retorna todos os clientes do banco de dados;
- Linha 18: o método **listaClientesPorDemandra()** retorna uma quantidade reduzida dos clientes cadastrados no banco de dados;
- Linhas 12, 22 e 48: cada linha cria um objeto do tipo **Criteria** a partir do método **createCriteria()** da **Session**. A classe **Criteria** oferece métodos para consulta, atualização e exclusão de tuplas, entre outras operações essenciais;
- Linhas 13, 30 e 33: mudam o tipo de ordenação da listagem a partir da chamada ao método **addOrder()** da classe **Criteria**;
- Linha 36: define a quantidade máxima de registros a ser retornada pela consulta a fim de obter o número de linhas correspondente ao que está sendo exibido em cada página do *DataTable* (veremos isso mais à frente);
- Linha 40: define a partir de que posição os dados devem ser retornados. Se, por exemplo, o usuário está na página 2 do *DataTable*, e são exibidos 10 registros por página, a 10<sup>a</sup> posição é o ponto inicial de onde os dados devem ser trazidos (o primeiro registro é o 0).
- Linhas 14 e 42: o método **list()** da classe **Criteria** é chamado para retornar uma listagem do banco de dados;
- Linha 53: indica que esta consulta retornará a contagem de registros de clientes que se encaixam nos eventuais filtros passados. Se nenhum filtro foi especificado pelo usuário, retorna a quantidade total de clientes;
- Linha 54: o método **uniqueResult()** da classe **Criteria** é chamado para retornar um único registro do banco de dados;
- Linhas 58 a 76: adiciona cláusulas no *where* da consulta que será executada, de acordo com os filtros preenchidos pelo usuário na tela.

Nosso próximo passo será criar a classe **LazyClienteDataModel**, responsável por chamar o EJB toda vez que for necessário carregar os clientes para preencher o *DataTable* através da paginação por demanda (isto envolve a primeira vez que a tela é chamada, mudanças no critério de ordenação, aplicação de um filtro ou troca da página no *DataTable*). A classe deve ficar no pacote **br.com.javamagazine.paginacaojm.paginacao** e o código precisa ficar semelhante ao da **Listagem 8**.

**Listagem 7.** Código da classe ClienteEJB.

```
01 package br.com.javamagazine.paginacaojm.ejb;
02
03 // imports omitidos
04
05 @Stateless
06 public class ClienteEJB {
07     @PersistenceContext
08     private Session session;
09
10    @SuppressWarnings("unchecked")
11    public List <Cliente> listaClientesEmMemoria() {
12        Criteria criteria = session.createCriteria(Cliente.class);
13        criteria.addOrder(Order.asc("nome"));
14        return criteria.list();
15    }
16
17    @SuppressWarnings("unchecked")
18    public List <Cliente> listaClientesPorDemandada(Map <String, Object>
19        filtros, String campoOrdenacao,
20        throws NoSuchFieldException, SecurityException {
21
22        Criteria criteria = session.createCriteria(Cliente.class);
23
24        // se algum filtro foi aplicado no datatable, adiciona as cláusulas e valores no where
25        if (!filtros.isEmpty())
26            adicionaClausulasEValoresNoWhere(filtros, criteria);
27
28        // se foi aplicado algum critério de ordenação no datatable
29        if (campoOrdenacao != null)
30            criteria.addOrder(tipoOrdenacao ==
31                SortOrder.ASCENDING ? Order.asc(campoOrdenacao) : Order.desc(campoOrdenacao));
32        else // senão, ordena pelo nome
33            criteria.addOrder(Order.asc("nome"));
34
35        // determina a quantidade de registros que deve ser retornada (qtd de linhas por
36        // página do datatable)
37        criteria.setMaxResults(qtdRegistrosPorPagina);
38
39        // Determina a partir de que registro deve retornar, por conta da página que o usuário
40        // selecionou no DataTable.
41
42        criteria.setFirstResult(registroInicial);
43
44
45        public Long retornaQuantidadeDeClientes(Map <String, Object> filtros)
46            throws NoSuchFieldException, SecurityException {
47            Long size = null;
48            Criteria criteria = session.createCriteria(Cliente.class);
49
50            // se algum filtro foi aplicado no datatable, adiciona as cláusulas no where e os
51            // parâmetros na consulta
52            if (filtros != null && !filtros.isEmpty())
53                adicionaClausulasEValoresNoWhere(filtros, criteria);
54            criteria.setProjection(Projections.rowCount());
55            size = (Long) criteria.uniqueResult();
56        }
57
58        private void adicionaClausulasEValoresNoWhere(Map <String, Object> filtros,
59            Criteria criteria)
60            throws NoSuchFieldException, SecurityException {
61
62            for (Iterator<String> it = filtros.keySet().iterator(); it.hasNext();) {
63                String propriedadeFiltrada = it.next();
64
65                if (Cliente.class.getDeclaredField(propriedadeFiltrada).getGenericType().equals(String.class))
66                    // Se estiver filtrando um campo do tipo String, transforma o valor pesquisado e o
67                    // valor do banco para maiúsculo para ignorar o case do campo e adiciona
68                    // um parâmetro
69                    // com o nome do campo pesquisado
70                    criteria.add(Restrictions.like(propriedadeFiltrada,
71                        (String) filtros.get(propriedadeFiltrada), MatchMode.ANYWHERE));
72                else
73                    // Se o campo não for String, adiciona uma cláusula para verificar se o valor do
74                    // campo no banco é igual ao valor passado adicionando um parâmetro com o nome
75                    // do campo pesquisado
76                    criteria.add(Restrictions.eq(propriedadeFiltrada, filtros.get(propriedadeFiltrada)));
77            }
78        }
79    }
```

**Listagem 8.** Código da classe LazyClienteDataModel.

```
01 package br.com.javamagazine.paginacaojm.paginacao;
02
03 import java.util.List;
04 import java.util.Map;
05 import org.primefaces.model.LazyDataModel;
06 import org.primefaces.model.SortOrder;
07 import br.com.javamagazine.paginacaojm.domain.Cliente;
08 import br.com.javamagazine.paginacaojm.ejb.ClienteEJB;
09
10 public class LazyClienteDataModel extends LazyDataModel<Cliente> {
11     private static final long serialVersionUID = 8310122015266723124L;
12
13     private List<Cliente> listaPaginadaDeClientes;
14     private ClienteEJB clienteEJB;
15
16     public LazyClienteDataModel(ClienteEJB clienteEJB) {
17         this.clienteEJB = clienteEJB;
18     }
19
20     @Override
21     public Cliente getRowData(String chave) {
22         for (Cliente cliente : listaPaginadaDeClientes) {
23             if (cliente.getId().equals(chave)) {
24                 return cliente;
25             }
26         }
27         return null;
28     }
29
30     @Override
31     public Object getRowKey(Cliente cliente) {
32         return cliente.getId();
33     }
34
35     @Override
36     public List<Cliente> load(int primeiroRegistro, int qtdRegistrosPagina,
37         String campoOrdenacao,
38         SortOrder tipoOrdenacao, Map <String, Object> filtros) {
39         try {
40             // Guarda na variável rowCount a quantidade de clientes retornada pelo EJB,
41             // levando em consideração possíveis filtros aplicados pelo usuário na tela
42             this.setRowCount(clienteEJB.retornaQuantidadeDeClientes(filtros).intValue());
43
44             // Retorna a lista de clientes devolvida pelo EJB levando em consideração
45             // possíveis filtros e o
46             // critério de ordenação definidos pelo usuário
47             return listaPaginadaDeClientes = clienteEJB.listaClientesPorDemandada(
48                 filtros, campoOrdenacao,
49                 tipoOrdenacao, qtdRegistrosPagina, primeiroRegistro);
50
51         } catch (NoSuchFieldException | SecurityException e) {
52             e.printStackTrace();
53         }
54     }
55 }
```

# Java EE 7: Como recuperar listagens sob demanda

Como dito anteriormente, essa classe nos permite seguir uma das boas práticas de programação do mercado ao utilizarmos o padrão de projeto *Lazy Loading*. Assim, um *Data Model* é um objeto que tem como função lidar com um grande volume de dados. Por sua vez, um *Lazy Data Model* — como a classe **LazyClienteDataModel** — é utilizado quando empregamos a paginação por demanda em uma solução. Assim, o código da **Listagem 8** pode ser explicado da seguinte forma:

- Linha 10: **LazyClienteDataModel** estende a classe abstrata **LazyDataModel** do PrimeFaces;
- Linhas 20 a 28: implementação do método **getRowData()**, que recebe como parâmetro uma **String** (a chave da linha do *DataTable*) e retorna o objeto associado a essa linha. De forma mais simples, recebe o *Id* de um cliente e retorna a instância da classe **Cliente**;
- Linhas 30 a 33: implementação do método **getRowKey()**, que recebe um **Cliente** como parâmetro e retorna a chave dele. No nosso caso, devolve o *Id* do objeto recebido;
- Linhas 35 a 52: implementação do método **load()**, que inicialmente recupera a quantidade total de registros existentes na tabela, obedecendo aos critérios dos filtros passados, e a guarda no atributo **rowCount** da super classe (linha 41). Em seguida retorna uma lista de clientes contendo uma quantidade correspondente ao número de registros por página no *DataTable* (na nossa implementação são 10). Lembrando que o método **load()** é chamado automaticamente pelo *DataTable* do PrimeFaces toda vez que o usuário clica em uma página diferente da atual, aplica um filtro, modifica o critério de ordenação ou troca a quantidade de registros por página da tabela.

Após finalizarmos este passo, precisamos criar nossos dois *managed beans*. Vamos iniciar pelo **ClienteBeanEmMemoria**. Portanto, crie uma classe com esse nome no pacote **br.com.javamagazine.paginacaojm.managedbean** e deixe o código igual ao da **Listagem 9** (lembre-se de inserir os gets e sets dos atributos **listaClientes** e **listaClientesFiltrados**).

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

- Linha 13: a anotação do CDI **@Named** transforma a classe em um *managed bean* ao dar um nome para que ele seja injetado em outra classe ou invocado via *expression language* (como não foi especificado nenhum nome, o CDI assume o nome da classe começando com a primeira letra em minúsculo);
- Linha 14: a anotação do CDI **@RequestScoped** define o objeto com escopo de requisição;
- Linhas 18 e 19: com a anotação **@EJB** definimos que o EJB **ClienteEJB** deve ser injetado em **ClienteBeanEmMemoria**;
- Linhas 20 e 21: declaração de duas *collections* para serem utilizadas pelo *DataTable* da página de listagem de clientes com paginação em memória. Tivemos que criar dois objetos pelo fato de o PrimeFaces criar um subconjunto da lista original quando um filtro é aplicado;
- Linhas 23 a 27: método que recupera todos os clientes do banco de dados, armazenando-os em **listaClientes**, e, no **return**, “chama” a página de listagem de clientes com paginação em memória.

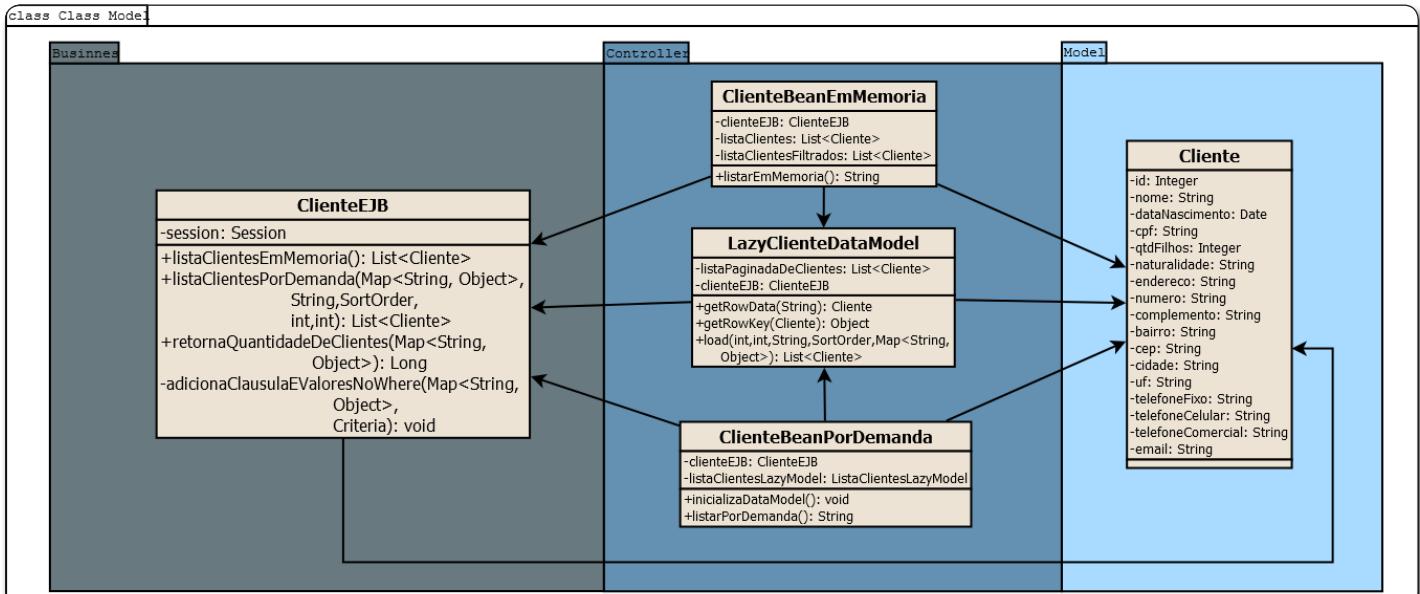
Depois de inserir o *managed bean* correspondente à paginação em memória, devemos criar a classe **ClienteBeanPorDemand** no mesmo pacote, deixando-a com o código igual ao da **Listagem 10** (lembre-se de inserir o get e o set do atributo **listaClientesLazyModel**).

**Listagem 9.** Código da classe **ClienteBeanEmMemoria**.

```
01 package br.com.javamagazine.paginacaojm.managedbean;
02
03 import java.io.Serializable;
04 import java.util.List;
05
06 import javax.ejb.EJB;
07 import javax.enterprise.context.RequestScoped;
08 import javax.inject.Named;
09
10 import br.com.javamagazine.paginacaojm.domain.Cliente;
11 import br.com.javamagazine.paginacaojm.ejb.ClienteEJB;
12
13 @Named
14 @RequestScoped
15 public class ClienteBeanEmMemoria implements Serializable {
16     private static final long serialVersionUID = 8310122015266723124L;
17
18     @EJB
19     private ClienteEJB clienteEJB;
20     private List<Cliente> listaClientes;
21     private List<Cliente> listaClientesFiltrados;
22
23     public String listarEmMemoria() {
24         listaClientes = clienteEJB.listaClientesEmMemoria();
25
26         return "listarClientesEmMemoria";
27     }
28
29 // gets e sets de listaClientes e listaClientesFiltrados omitidos
30 }
```

**Listagem 10.** Código da classe **ClienteBeanPorDemand**.

```
01 package br.com.javamagazine.paginacaojm.managedbean;
02
03 import java.io.Serializable;
04 import javax.annotation.PostConstruct;
05 import javax.ejb.EJB;
06 import javax.enterprise.context.RequestScoped;
07 import javax.inject.Named;
08 import br.com.javamagazine.paginacaojm.ejb.ClienteEJB;
09 import br.com.javamagazine.paginacaojm.paginacao.LazyClienteDataModel;
11
12 @Named
13 @RequestScoped
14 public class ClienteBeanPorDemand implements Serializable {
15     private static final long serialVersionUID = 5517414396849292509L;
16
15     @EJB
16     private ClienteEJB clienteEJB;
17     private LazyClienteDataModel listaClientesLazyModel;
18
19     @PostConstruct
20     public void inicializaDataModel() {
21         listaClientesLazyModel = new LazyClienteDataModel(clienteEJB);
22     }
23     public String listarPorDemand() {
24         return "listarClientesPorDemand";
25     }
26
27 // get e set de listaClientesLazyModel omitidos
28 }
```



**Figura 1.** Diagrama de classes do projeto

Observe que entre as linhas 19 e 22 foi criado um método com a anotação `@PostConstruct`. Isso significa que `inicializaDataModel()` será executado logo após a classe estar instanciada e todas as injetões de dependências terem sido resolvidas. No nosso caso, usamos esse método para instanciar o nosso *Lazy Data Model* passando o EJB que já foi injetado. O resto do código é bastante semelhante.

Após termos criado nossas classes Java, o diagrama de classes do nosso projeto ficou igual ao da **Figura 1**.

## Criando as páginas do sistema

Dando continuidade ao desenvolvimento da aplicação, nosso próximo passo será criar as páginas XHTML. Serão quatro arquivos, e em dois deles utilizaremos o componente **DataTable** para listar os clientes cadastrados na base de dados. Em uma das abordagens, empregaremos a paginação em memória e na outra, a paginação por demanda.

A primeira das nossas páginas será o *template* geral, no qual definiremos dois trechos comuns a todas as páginas do sistema utilizando o framework Facelets: uma instrução CSS para o PrimeFaces diminuir o tamanho da fonte dos seus componentes e o menu da aplicação. Isso permitirá codificar apenas o corpo interno das outras telas. Desta forma, clique com o botão direito em cima de `src/main/webapp`, selecione *New* e depois clique em *HTML File*. Na tela que surge, informe o nome “`layoutBase.xhtml`” e clique em *Next*. Em seguida, selecione a opção *New XHTML File (1.0 strict)* e clique em *Finish*. O Eclipse gerará um pequeno código que deve ser alterado para ficar igual ao da **Listagem 11**.

Os principais trechos do código apresentado são:

- Linhas 5 a 7: instrução de folha de estilos para diminuir a fonte para 90% em todos os componentes do PrimeFaces. Como a instrução está no *template*, todas as páginas do sistema a receberão;

**Listagem 11.** Código da página `layoutBase.xhtml`.

```

01 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
02 <html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:p="http://primefaces.org/ui"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:ui="http://java.sun.com/jsf/facelets">
03 <h:head>
04   <title>Paginação JM</title>
05   <style type="text/css">
06     .ui-widget, .ui-widget .ui-widget { font-size: 90%; }
07   </style>
08 </h:head>
09 <h:body>
10   <ui:insert name="menu">
11     <h:form>
12       <p:menubar>
13         <p:menuitem value="Home" action="index.xhtml?faces-redirect=true"
           ajax="false"/>
14         <p:submenu label="Clientes">
15           <p:menuitem value="Listar Em Memória"
             action="#{clienteBeanEmMemoria.listarEmMemoria}" ajax="false"/>
16           <p:menuitem value="Listar Por Demanda"
             action="#{clienteBeanPorDemandada.listarPorDemandada}" ajax="false"/>
17         </p:submenu>
18       </p:menubar>
19     </h:form>
20   </ui:insert>
21   <ui:insert name="corpo"/>
22 </h:body>
23 </html>

```

- Linha 10: o componente `<ui:insert />` declara um possível ponto de substituição, ou seja, as demais páginas que usarem esta como *template* podem redefinir o código que está dentro dessa *tag*. No nosso caso, usamos este trecho para inserir o menu. Essa solução pode ser útil, por exemplo, no caso de uma página do sistema ter a necessidade de não exibir o menu;

# Java EE 7: Como recuperar listagens sob demanda

- Linha 11: adiciona um formulário do JSF. Sem ele, os itens do menu não funcionam;
- Linha 12: insere o componente de barra de menu;
- Linha 13: insere um item de menu na barra, o qual tem como ação redirecionar o usuário para a página inicial do sistema. A instrução `ajax="false"` configura a ação disparada no clique do mouse como uma requisição não AJAX;
- Linhas 15 e 17: adição de dois menus hierárquicos à barra de menu;

Listagem 12. Código da página index.xhtml.

```
01 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
02 <html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:p="http://primefaces.org/ui"
  xmlns:ui="http://java.sun.com/jsf/facelets">
03   <ui:composition template="layoutBase.xhtml">
04     <ui:define name="corpo">
05       <h2>Bem vindo. Escolha sua opção no menu acima.</h2>
06     </ui:define>
07   </ui:composition>
08 </html>
```

Listagem 13. Código da página listarClientesEmMemoria.xhtml.

```
01 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
02 <html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:p="http://primefaces.org/ui"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:t="http://myfaces.apache.org/tomahawk">
03   <ui:composition template="layoutBase.xhtml">
04     <ui:define name="corpo">
05       <t:saveState value="#{clienteBeanEmMemoria.listaClientes}" />
06       <t:saveState value="#{clienteBeanEmMemoria.listaClientesFiltrados}" />
07
08       <h1>Listagem de Clientes Em Memória</h1>
09
10      <h:form>
11        <p:dataTable id="listaClientes" widgetVar="listaClientes"
12          value="#{clienteBeanEmMemoria.listaClientes}"
13          filteredValue="#{clienteBeanEmMemoria.listaClientesFiltrados}" var="c"
14          filterEvent="enter"
15          rows="10" paginator="true" paginatorPosition="bottom" sortBy="#{c.nome}"
16          paginatorTemplate="{CurrentPageReport} {FirstPageLink} {PreviousPage
17          Link} {PageLinks} {NextPageLink} {LastPageLink} {RowsPerPageDropdown}"
18          currentPageReportTemplate="((currentPage) de (totalPages))">
19          <p:column headerText="Nome" filterBy="#{c.nome}" filterMatchMode="contains"
20            sortBy="#{c.nome}">
21            <h:outputText value="#{c.nome}" />
22          </p:column>
23          <p:column headerText="CPF" style="text-align: center" filterBy="#{c.cpf}"
24            filterMatchMode="startsWith" sortBy="#{c.cpf}">
25            <h:outputText value="#{c.cpf}" />
26          </p:column>
27          <p:column headerText="Qtd. Filhos" style="text-align: center" filterBy=
28            "#{c.qtdFilhos}"
29            filterMatchMode="exact" sortBy="#{c.qtdFilhos}">
30            <f:facet name="filter">
31              <p:selectOneMenu onchange="PF('listaClientes').filter();">
32                <f:selectItem itemLabel="Selecione" itemValue="#{null}">
33                  noSelectionOption="true"/>
34                <f:selectItem itemLabel="0" itemValue="0"/>
35                <f:selectItem itemLabel="1" itemValue="1"/>
```

- Linha 23: adicionamos outro componente `<ui:insert />` como possível ponto de substituição. Neste caso não há código dentro do componente. Assim, se uma página que usar este XHTML como *template* não redefinir a estrutura com o nome **corpo**, este trecho da página ficará sem conteúdo.

Após criarmos o *template*, nosso próximo passo será gerar a página inicial do sistema. Para isso, crie o arquivo *index.xhtml*, novamente no diretório *src/main/webapp*, e deixe o código igual ao da **Listagem 12**.

Podemos observar na linha 3 dessa listagem o componente `<ui:composition />`. Nele indicamos a página *layoutBase.xhtml* como *template* através da instrução `template="layoutBase.xhtml"`. Já na linha 4 inserimos a instrução `<ui:define />`, que indica que estamos redefinindo um trecho criado com `<ui:insert />` (neste caso, o `<ui:insert name="corpo" />` presente no *template*).

Com a tela inicial da aplicação pronta, nossas próximas páginas serão as relacionadas à listagem dos clientes. Seguindo a mesma ordem de criação dos *managed beans*, vamos primeiro gerar a que traz os objetos com paginação em memória. Desta forma, crie o arquivo *listarClientesEmMemoria.xhtml* na mesma pasta dos dois arquivos anteriores e deixe o código semelhante ao da **Listagem 13**.

```
33      <f:selectItem itemLabel="2" itemValue="2"/>
34      <f:selectItem itemLabel="3" itemValue="3"/>
35      <f:selectItem itemLabel="4" itemValue="4"/>
36      <f:selectItem itemLabel="5" itemValue="5"/>
37      <f:selectItem itemLabel="6" itemValue="6"/>
38    </p:selectOneMenu>
39  </f:facet>
40  <h:outputText value="#{c.qtdFilhos}" />
41 </p:column>
42 <p:column headerText="Naturalidade" style="text-align: center"
  filterBy="#{c.naturalidade}"
  filterMatchMode="startsWith" sortBy="#{c.naturalidade}">
43    <h:outputText value="#{c.naturalidade}" />
44 </p:column>
45 <p:column headerText="Data Nasc." sortBy="#{c.dataNascimento}">
46    <h:outputText value="#{c.dataNascimento}">
47      <f:convertDateTime pattern="dd/MM/yyyy"/>
48    </h:outputText>
49 </p:column>
50 <p:column headerText="Endereço" style="text-align: center">
51    <h:outputText value="#{c.endereco}" />
52    <h:outputText value="#{c.numero}" />
53    <h:outputText value="#{c.bairro}" />
54    <h:outputText value="CEP #{c.cep}" />
55    <h:outputText value="#{c.cidade}" />
56    <h:outputText value="#{c.uf}" />
57 </p:column>
58 <p:column headerText="Tel. Celular" filterBy="#{c.telefoneCelular}"
  filterMatchMode="startsWith" sortBy="#{c.telefoneCelular}">
59    <h:outputText value="#{c.telefoneCelular}" />
60 </p:column>
61 <p:column headerText="E-mail" filterBy="#{c.email}" filterMatchMode="contains"
  sortBy="#{c.email}">
62    <h:outputText value="#{c.email}" />
63 </p:column>
64 </p:dataTable>
65 </h:form>
66 </ui:define>
67 </ui:composition>
68 </html>
```

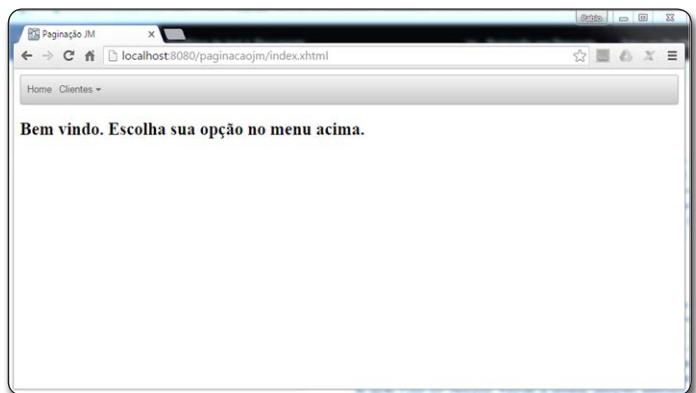
Os principais trechos desse código são:

- Linhas 5 e 6: adição do componente `saveState` do Tomahawk. Ele é necessário porque **ClienteBeanEmMemória** tem escopo de requisição e, devido ao comportamento do ciclo de vida do JSF, se não salvarmos o estado (ou seja, guardarmos os valores) de **listaClientes** e **listaClientesFiltrados**, ambos ficarão como `null` no *Managed Bean* ao trocarmos de página no **DataTable**. Existem duas soluções para não termos que usar o `saveState`: uma seria utilizarmos o escopo de sessão do CDI (neste caso o consumo de recursos de memória do servidor cresceria significativamente); a outra seria o escopo de visão do JSF, mas como estamos utilizando apenas os escopos nativos do CDI, fugimos dessa opção;
- Linhas 11 a 67: declaram um **DataTable** informando o campo de ordenação da listagem como o nome do cliente, definem que os filtros devem ser aplicados após a tecla ENTER ser pressionada, inserem o paginador da tabela no rodapé e colocam a instrução para que 10 clientes sejam exibidos por página do **DataTable**;
- Linhas 17 e 18: definem uma coluna no **DataTable** com o título do cabeçalho “Nome”, inserem um *input* de texto para aplicação de filtro pelo nome do cliente — levando em conta que o valor digitado pode estar em qualquer posição — e definem a possibilidade de ordenar a tabela pela mesma coluna;
- Linhas 25 a 41: declaram uma coluna no **DataTable** com o título do cabeçalho “Qtd. Filhos”, inserem um *select* para aplicação de filtro pela quantidade de filhos do cliente — levando em conta que o valor informado deve ser igual ao que se encontra no banco de dados — e definem a possibilidade de ordenar a tabela pela mesma coluna. Veja que no *select* foi colocada uma instrução para aplicar o filtro no momento em que o valor for modificado;
- Linhas 42 a 45: definem uma coluna no **DataTable** com o título do cabeçalho “Naturalidade”, inserem um *input* de texto para aplicação de filtro pela naturalidade do cliente e definem a possibilidade de ordenar a tabela pela mesma coluna;
- Linhas 46 a 50: adicionam mais uma coluna no **DataTable** — dessa vez com o título “Data Nasc.” —, definem a possibilidade de ordenar a tabela por ela e usam a *tag* nativa do JSF `<f:convertDateTime />` para formatar a data no padrão brasileiro;
- Linhas 51 a 58: declaram uma coluna no **DataTable** para exibir o endereço do cliente composto pelos atributos endereço, número, bairro, CEP, cidade e UF;
- Linhas 59 a 62: definem uma coluna no **DataTable** com o título do cabeçalho “Tel. Celular”, inserem um *input* de texto para aplicação de filtro pelo telefone celular do cliente — levando em conta que o valor digitado deve estar no início — e definem a possibilidade de ordenar a tabela pela mesma coluna;
- Linhas 63 a 66: declaram a última coluna do **DataTable** com o título do cabeçalho “E-mail”, inserem um *input* de texto para aplicação de filtro pelo e-mail do cliente e definem a possibilidade de ordenar a tabela pela mesma coluna.

Nosso próximo passo será criar a página que lista os clientes com paginação por demanda. Assim, gere o arquivo `listarClientesPorDemanda.xhtml` na mesma pasta dos arquivos anteriores e deixe o código como mostrado na **Listagem 14**.

Veja que o código é bastante semelhante ao da **Listagem 13**. O que temos de novidade é que declaramos que o **DataTable** usa paginação por demanda (instrução `lazy="true"` na linha 09) e colocamos a possibilidade de filtragem por data de nascimento (linhas 39 a 45). Vale esclarecer que o mesmo não foi feito na paginação em memória porque precisaríamos configurar mais alguns itens na aplicação e, desta forma, fugiríamos do escopo do artigo.

Como estamos com tudo pronto, já podemos rodar o sistema. É necessário, entretanto, que o WildFly já esteja configurado no Eclipse. Caso não o tenha configurado na IDE, na seção **Links** há um post do blog de um dos autores em que é possível ver como realizar esta etapa. Para executar a aplicação, clique com o botão direito em cima da raiz do projeto, selecione *Run As > Run On Server*, marque o item *WildFly 10* na tela que surge e pressione o botão *Finish* para confirmar. A tela inicial do sistema será exibida no seu navegador, conforme a **Figura 2**.



**Figura 2.** Tela inicial do sistema

Nessa página, vamos primeiro conferir a listagem de clientes com paginação em memória acessando o menu *Clientes* no topo da página e clicando em *Listar em Memória*. Veja que claramente há certa demora em trazer as quase 40 mil linhas a partir do banco de dados (discutiremos os resultados a seguir). Aproveite que está com a tela aberta e modifique a coluna utilizada para ordenação, aplique algum filtro ou avance nas páginas do **DataTable** para ver os clientes que estão mais à frente. Feito isso, acesse a página com paginação por demanda através da opção *Listar por Demanda* e execute as mesmas operações para verificar as diferenças.

## Comparando os resultados

Neste tópico, temos como objetivo analisar os resultados obtidos em cada um dos cenários apresentados pela aplicação para podermos comprovar por que a paginação por demanda é mais vantajosa quando lidamos com grandes volumes de dados. Para medir as variáveis desejadas (tempo de carregamento da tela, tamanho do *response*, consumo de memória e consumo de

# Java EE 7: Como recuperar listagens sob demanda

Listagem 14. Código da página listarClientesPorDemanda.xhtml.

```
01 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
02 <html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html" xmlns:p="http://primefaces.org/ui"
  xmlns:f="http://java.sun.com/jsf/core" xmlns:ui="http://java.sun.com/jsf/facelets">
03 <ui:composition template="layoutBase.xhtml">
04   <ui:define name="corpo">
05     <h1>Listagem de Clientes Por Demanda</h1>
06     <h:form id="form">
07       <p:dataTable id="listaClientes" widgetVar="listaClientes"
08         value="#{clienteBeanPorDemanda.listaClientesLazyModel}" var="c"
09         rows="10" paginator="true" paginatorPosition="bottom" lazy="true"
10         filterEvent="enter"
11         paginatorTemplate="{CurrentPageReport} {FirstPageLink} {PreviousPage
12           Link} {PageLinks} {NextPageLink} {LastPageLink} {RowsPerPageDropdown}"
13         currentPageReportTemplate="({currentPage} de {totalPages})"
14         <p:column headerText="Nome" filterBy="#{c.nome}" sortBy="#{c.nome}">
15           <h:outputText value="#{c.nome}" />
16         </p:column>
17         <p:column headerText="CPF" style="text-align: center" filterBy="#{c.cpf}"
18           sortBy="#{c.cpf}">
19           <h:outputText value="#{c.cpf}" />
20         </p:column>
21         <p:column headerText="Qtd. Filhos" style="text-align: center"
22           filterBy="#{c.qtdFilhos}" sortBy="#{c.qtdFilhos}">
23           <f:facet name="filter">
24             <p:selectOneMenu onchange="PF('listaClientes').filter();">
25               <f:selectItem itemLabel="Selecione" itemValue="#{null}"
26                 noSelectionOption="true"/>
27               <f:selectItem itemLabel="0" itemValue="0"/>
28               <f:selectItem itemLabel="1" itemValue="1"/>
29               <f:selectItem itemLabel="2" itemValue="2"/>
30               <f:selectItem itemLabel="3" itemValue="3"/>
31               <f:selectItem itemLabel="4" itemValue="4"/>
32               <f:selectItem itemLabel="5" itemValue="5"/>
33               <f:selectItem itemLabel="6" itemValue="6"/>
34               <f:converter converterId="javax.faces.Integer" />
35             </p:selectOneMenu>
36           </f:facet>
37           <h:outputText value="#{c.qtdFilhos}" />
38         </p:column>
39         <p:column id="dtNasc" headerText="Data Nasc."
40           sortBy="#{c.dataNascimento}" filterBy="#{c.dataNascimento}">
41           <f:facet name="filter">
42             <p:calendar id="filtro_data_nascimento" navigator="true"
43               pattern="dd/MM/yyyy" locale="pt_BR" yearRange="c-100:c+1">
44               <p:ajax event="dateSelect" oncomplete="PF('listaClientes').filter();"/>
45             </p:calendar>
46           </f:facet>
47           <h:outputText value="#{c.dataNascimento}" />
48         </p:column>
49         <p:column headerText="Endereço" style="text-align: center">
50           <h:outputText value="#{c.endereco}" />
51           <h:outputText value="#{c.numero}" />
52           <h:outputText value="#{c.bairro}" />
53           <h:outputText value="CEP #{c.cep}" />
54           <h:outputText value="#{c.cidade}" />
55           <h:outputText value="#{c.uf}" />
56         </p:column>
57         <p:column headerText="Tel. Celular" filterBy="#{c.telefoneCelular}"
58           sortBy="#{c.telefoneCelular}">
59           <h:outputText value="#{c.telefoneCelular}" />
60         </p:column>
61       </p:column>
62     </p:dataTable>
63   </h:form>
64 </ui:define>
65 </ui:composition>
66 </html>
```

processamento) foram utilizados o pacote de ferramentas do desenvolvedor do Google Chrome e o VisualVM, um aplicativo que acompanha nativamente o JDK. Os testes foram executados em uma máquina Core i5 com 4Gb de RAM rodando o Windows 7 (versão 64 bits) e com o JDK 1.7.

Na Figura 3 destacamos em vermelho o tempo de carregamento da tela e o tamanho do *response* com paginação em memória. Os apontamentos mostrados foram gerados pelo pacote de ferramentas do desenvolvedor embutido na versão 51.0.2704.84 do navegador Google Chrome.

Veja que a página levou 5,3 segundos para carregar a listagem inicial e o tamanho do *response* foi de 16,7KB. Aqui existe um detalhe: embora todos os clientes estejam salvos em memória (os quase 40 mil), o *DataTable* carrega apenas os clientes que estão sendo exibidos. Quando o usuário clica na página 2 da tabela para ver os clientes seguintes, por exemplo, o componente recupera os dados da memória e os exibe na tela. Assim, o tráfego de rede fica infinitamente menor.

A Figura 4 mostra o resultado da tela de paginação por demanda, obtido com a mesma ferramenta.

Como podemos verificar, o tempo de carregamento foi de 685 milissegundos (quase oito vezes menor!) e o tamanho do *response* cresceu

um pouco (para 17,4Kb). Vale lembrar que no caso da paginação por demanda colocamos mais uma opção de filtro na página, o que explica essa diferença no *response* e causa um pequeno aumento no tempo de carregamento da página. Deste modo, caso não houvesse esse filtro adicional, a diferença entre os tempos poderia ser ainda maior.

Nosso próximo teste foi feito em uma operação bastante utilizada em tabelas como a do nosso sistema: a troca do campo de ordenação. A Figura 5 mostra o tempo de carregamento da tela e o tamanho do *response* com paginação em memória ao definirmos que a ordenação dos dados deve ser feita pelo CPF do cliente.

Observe que a operação levou mais de 1,6 segundo e que o tamanho do *response* foi de 5,8KB. Já a Figura 6 mostra os resultados dessa mesma operação para a paginação por demanda.

Note que a operação levou 276 milissegundos (quase seis vezes mais rápido!) e que o tamanho do *response* ficou o mesmo (5,8KB).

Devemos salientar que o mesmo comportamento se aplica a outras operações, como troca de página no *DataTable* ou aplicação de algum filtro e, em todos esses casos, estamos falando de uma simulação local, em que apenas um cliente está requisitando os dados ao servidor. Em uma situação real, em que vários usuários acessam a aplicação, o tempo de carregamento da tela com paginação em memória tende a ficar bem maior.

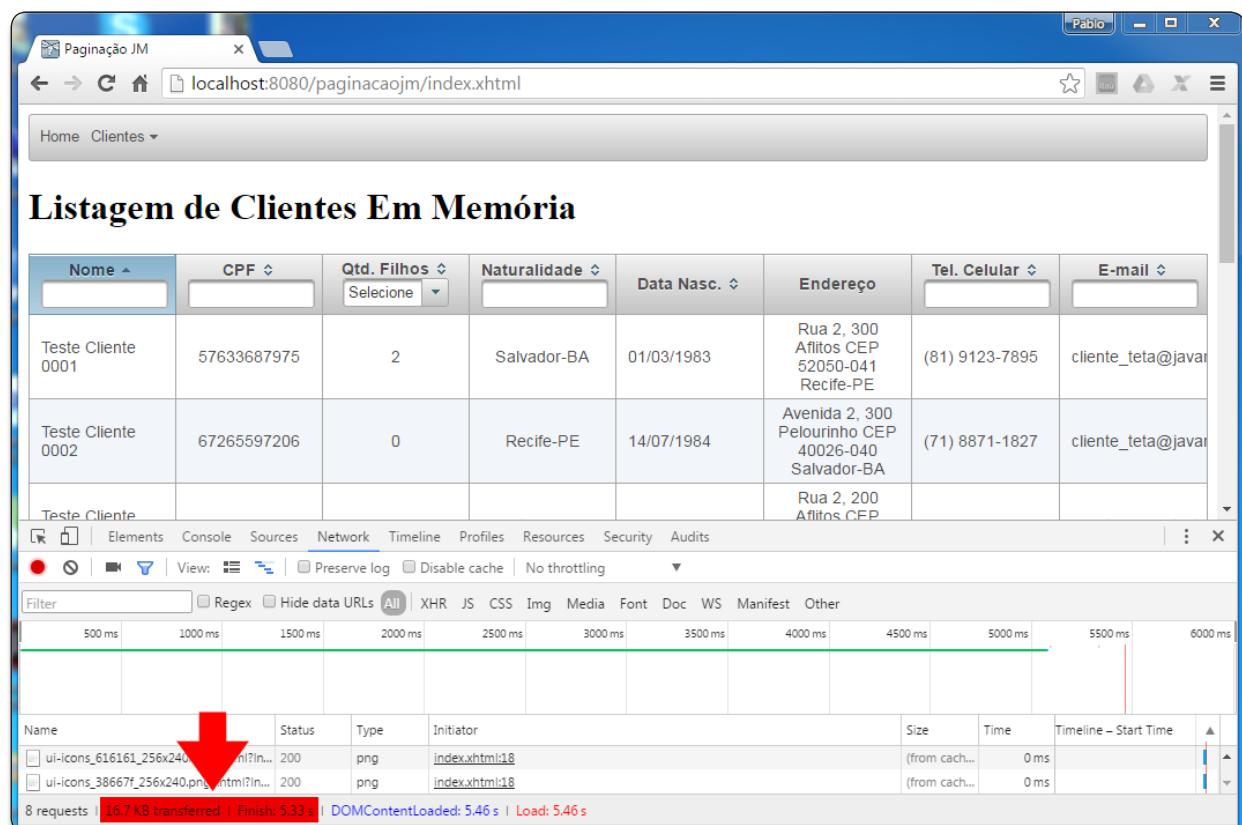


Figura 3. Resultado para a listagem de clientes na tela com paginação em memória

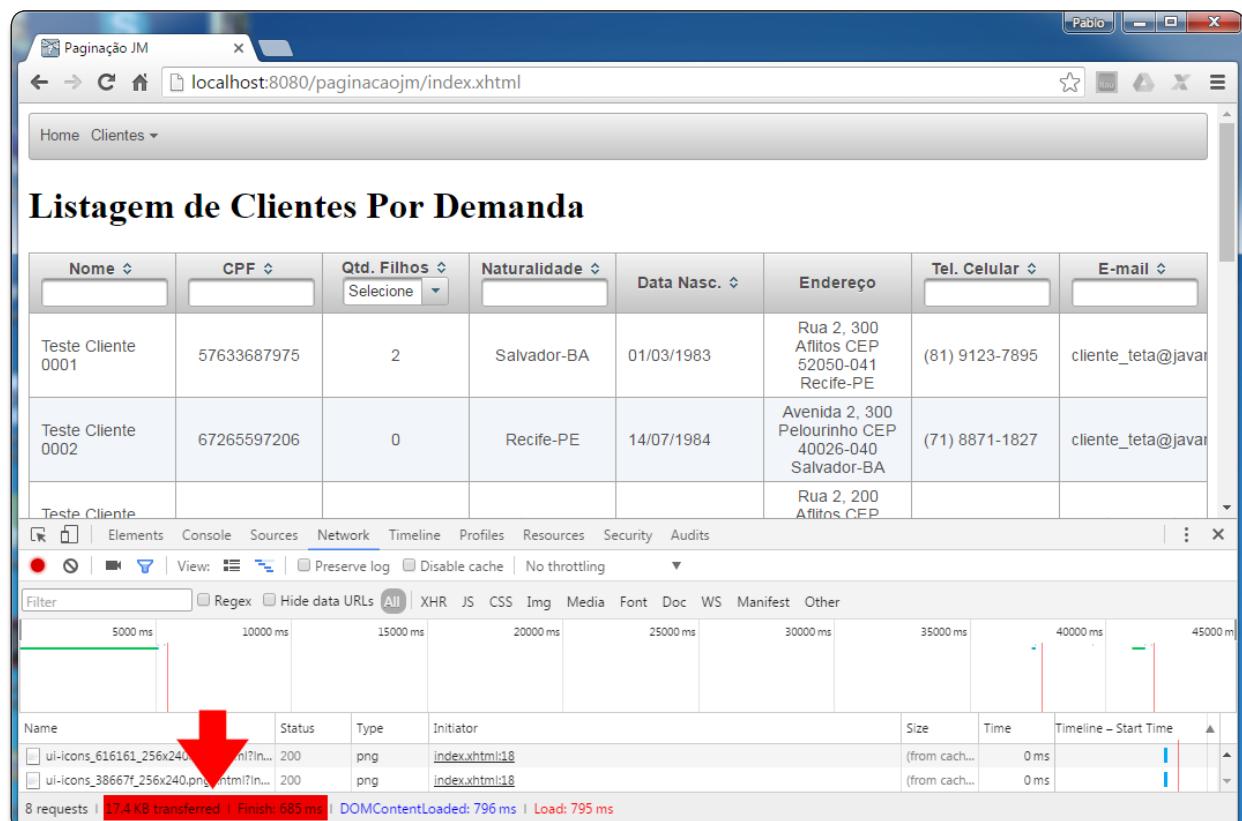


Figura 4. Resultado para a listagem de clientes na tela com paginação por demanda

# Java EE 7: Como recuperar listagens sob demanda

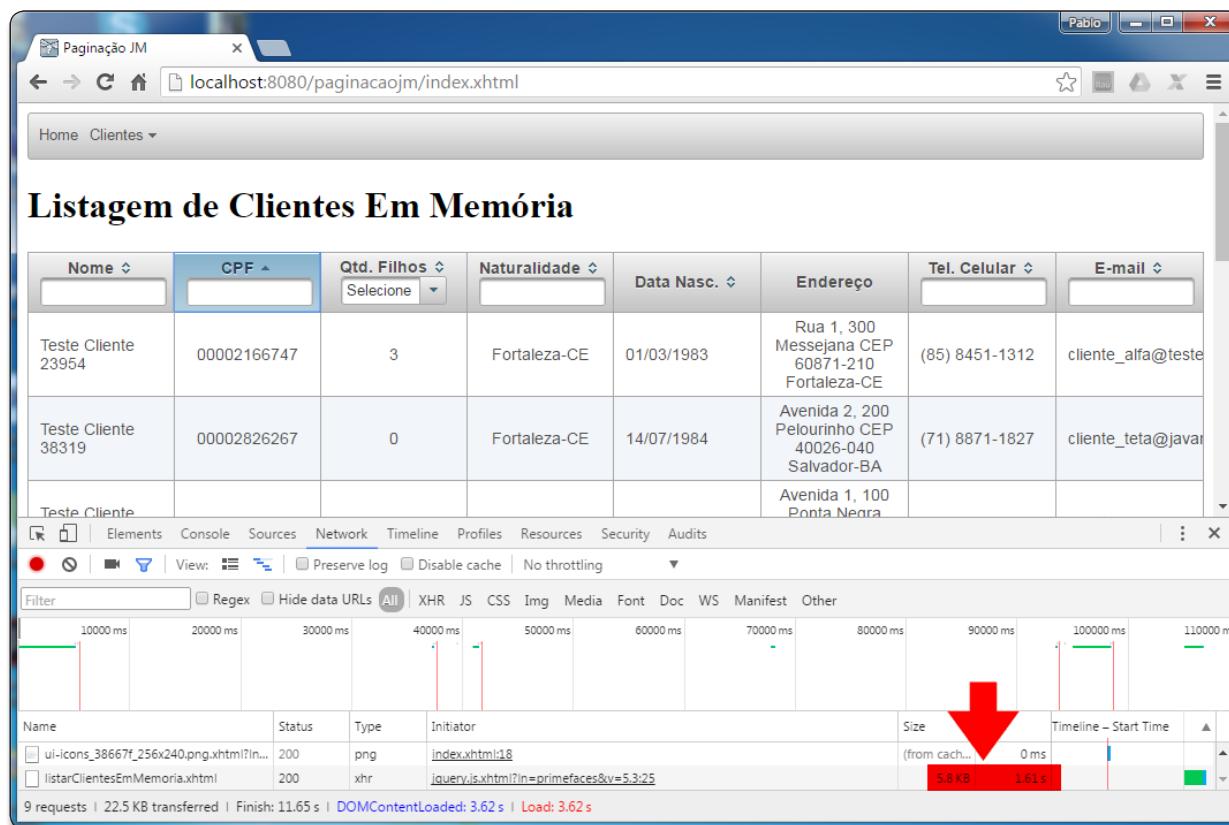


Figura 5. Resultado para a troca do campo de ordenação pelo CPF na tela com paginação em memória

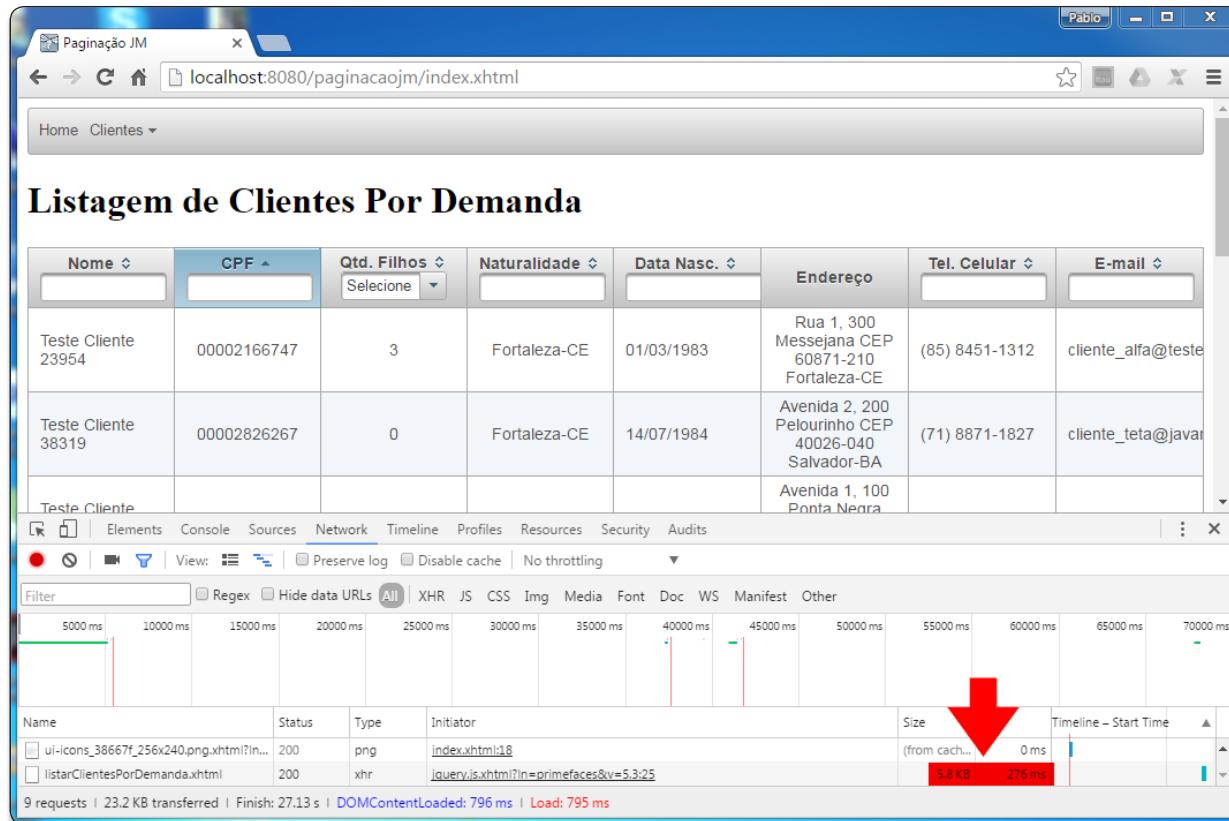


Figura 6. Resultado para a troca do campo de ordenação pelo CPF na tela com paginação por demanda

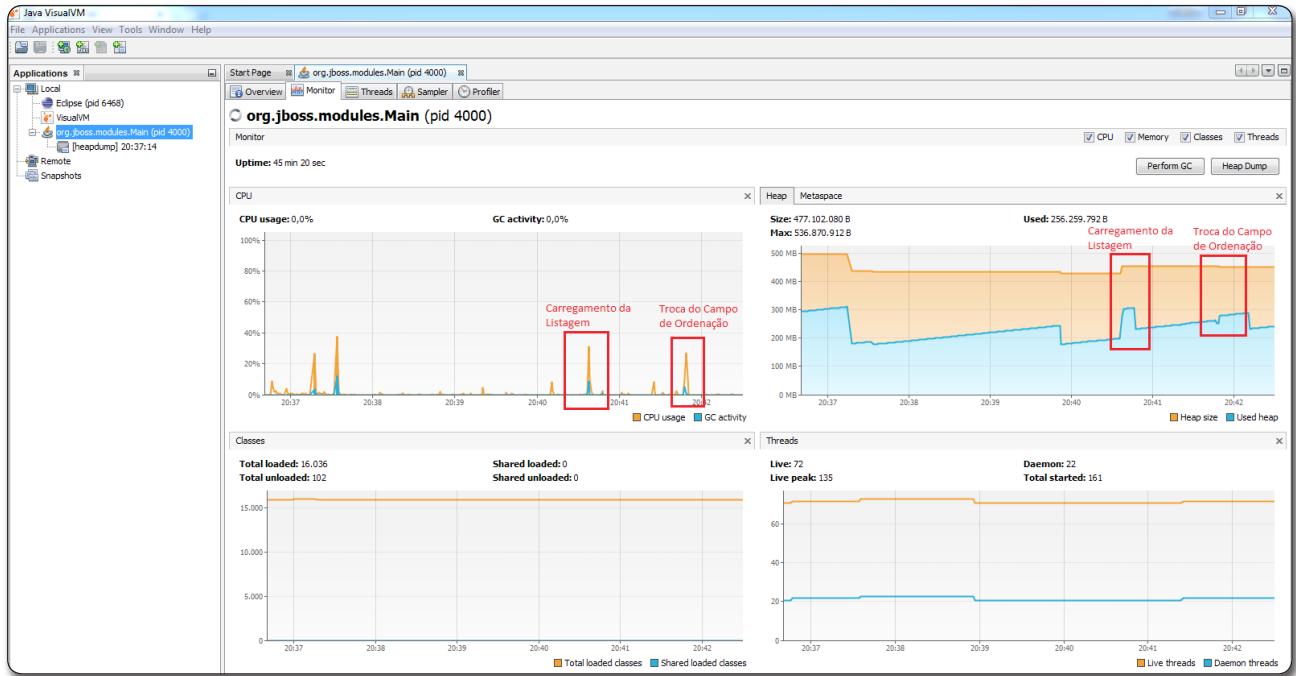


Figura 7. Consumo de recursos para a paginação feita em memória, de acordo com o VisualVM

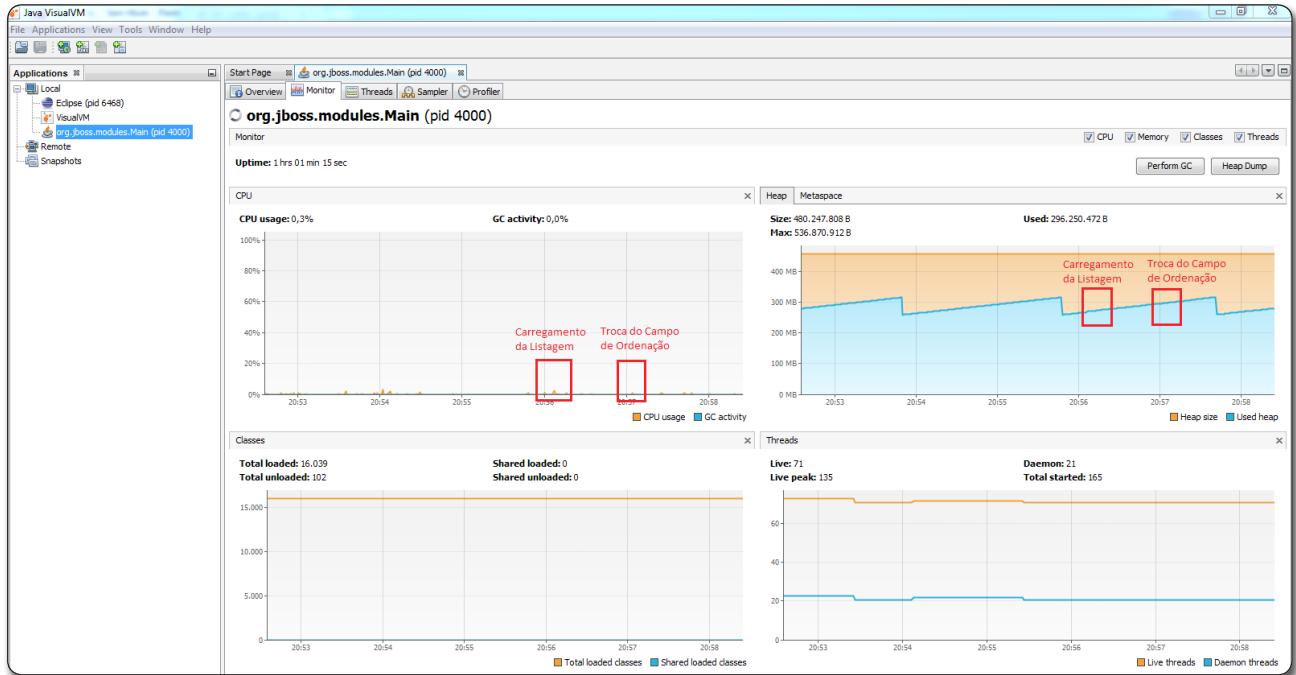


Figura 8. Consumo de recursos para a paginação sob demanda, de acordo com o VisualVM

Dando continuidade aos testes, nosso outro quesito de avaliação é o consumo de recursos da máquina (memória e processamento). Neste contexto, a tela gerada pelo VisualVM para a paginação em memória pode ser verificada na Figura 7. Já para a paginação por demanda, o resultado é exposto na Figura 8.

Pelos gráficos gerados, podemos identificar que o consumo de memória e processamento da solução de paginação em memória é

muito superior ao da paginação por demanda. Veja, por exemplo, que o uso de memória do *Heap*, que estava em aproximadamente 200MB, salta para mais de 300MB, e o número final poderia ser maior caso o *Garbage Collector* não tivesse entrado em ação no momento em que fizemos a requisição no sistema.

No que diz respeito ao processamento, chama a atenção os picos atingidos em ambas as operações da paginação em memória

# Java EE 7: Como recuperar listagens sob demanda

(listagem inicial e troca no critério de ordenação), chegando a atingir mais de 30% do uso de CPU. Na paginação por demanda, por sua vez, praticamente não há alteração no consumo de recursos nas duas chamadas.

Neste momento, é importante lembrar, novamente, que os testes foram feitos em *localhost*, com um único usuário por vez e uma quantidade pequena de dados cadastrada no banco de dados, pois sabemos que é extremamente comum empresas trabalharem com volumes muito maiores de informações.

Enfim, nossos testes mostraram que uma má prática utilizada pela maioria dos desenvolvedores pode ser facilmente substituída por uma solução simples e robusta em qualquer aplicação com apenas poucos passos. Neste cenário, comparar as duas abordagens mais comuns empregadas em aplicações *web Java* para a listagem de objetos nos deu a dimensão de como é importante economizar recursos através da recuperação de dados por demanda.

Como um grande auxílio, o DataTable do PrimeFaces facilita bastante a vida do programador ao fornecer a estratégia de Lazy Loading. Deste modo, vários detalhes da implementação estão encapsulados, restando apenas uma rápida consulta à documentação e a codificação de poucas linhas. Portanto, se o seu aplicativo não usa a paginação por demanda, está na hora de você reavaliar a estratégia de exibição de listagens.

## Autor



**Joel Xavier Rocha**

<http://joelxr.github.io> / [joelxr@gmail.com](mailto:joelxr@gmail.com)

Analista de Sistemas Java, certificado OCJP, bacharel em Engenharia da Computação pelo Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE). Trabalha na Secretaria de Finanças do Município de Fortaleza no Ceará (SEFIN) e desenvolve e mantém sistemas há mais de cinco anos.

## Autor



**Pablo Bruno de Moura Nóbrega**

<http://pablonobrega.wordpress.com>  
[pablonobrega2004@gmail.com](mailto:pablonobrega2004@gmail.com)

Gerente de Configuração de Software, certificado OCJP e OCWCD, Graduado em Ciências da Computação pela Universidade de Fortaleza – UNIFOR, Mestre em Computação pela Universidade Estadual do Ceará – UECE, Especialista em Gerenciamento de Projetos pela UNIFOR, trabalha na Secretaria Municipal de Finanças de Fortaleza – SEFIN e desenvolve sistemas há cerca de nove anos.



## Links:

### Artigo: Avoid the Pains of Pagination.

<http://uxmovement.com/navigation/avoid-the-pains-of-pagination/>

### Artigo: The Impact of Paging vs. Scrolling on Reading Online Text Passages.

<http://usabilitynews.org/the-impact-of-paging-vs-scrolling-on-reading-online-text-passages/>

### JavaDoc da anotação PostConstruct.

<http://docs.oracle.com/javaee/6/api/javax/annotation/PostConstruct.html>

### Pablo Nóbrega explica como configurar o WildFly 10 no Eclipse Mars.

<https://pablonobrega.wordpress.com/2016/06/06/configurando-o-wildfly-10-no-eclipse-mars/>

### Endereço para download do driver JDBC do PostgreSQL.

<https://jdbc.postgresql.org/download.html>

# Construindo aplicações reativas com Undertow

Aprenda a desenvolver aplicações modulares de alta performance com requisições que não param seu container

O mundo da computação está em constante mudança, e, com isso, muitas vezes precisamos deixar antigos paradigmas para trás e adotar novas opções para criarmos algo do zero. Hoje, com uma demanda cada vez mais crescente e diversificada, é exigido muito mais das soluções que disponibilizamos online, principalmente no que diz respeito ao modo de acesso, que não se limita mais a computadores.

A maneira como consumimos conteúdo também mudou, o que pode ser observado com o uso dos smartphones, responsáveis diretos por esse novo cenário. Atualmente, grande parte dos acessos se dá através de apps, e não mais através de navegadores. Isso tira de nossos serviços várias necessidades, como, por exemplo, servir conteúdo estático, como arquivos HTML, JavaScript, CSS, entre outros. O foco, então, passa a ser uma arquitetura voltada a disponibilizar serviços para atender a essa grande demanda gerada por apps, que são muitas vezes dependentes de serviços que os alimentem com informação.

Esse novo comportamento pode gerar um aumento sensível nos custos relacionados à infraestrutura do sistema, pois junto com a necessidade de disseminar conteúdo em diferentes formatos, existe também um aumento no custo para atender a essa nova demanda. Em casos mais extremos, requer que empresas reestruturem seus serviços para conseguirem manter a qualidade, como já aconteceu com a Netflix. Enfim, esse trabalho de repensar a maneira como utilizamos serviços impacta não apenas a infraestrutura do sistema, mas todo um mercado de desenvolvimento.

Quando falamos de mobile, por exemplo, o uso de Node.js para criar serviços que atendam aos aplicativos vem se tornando uma grande referência, justamente por fazer uso de eventos em suas threads. Ele é apresentado, em seu próprio site, como um framework focado em

## Fique por dentro

Este artigo apresenta o Undertow, novo web server do WildFly que trouxe a proposta de aumentar a performance do container da Red Hat e oferecer a possibilidade aos desenvolvedores de trabalhar com ele de forma isolada para construir serviços modularizados. Aqui, você verá como utilizá-lo para criar serviços mais rápidos e distribuídos, usufruindo de recursos para lidar com requisições de modo não bloqueante e garantido, assim, uma melhor performance a suas aplicações. Além disso, irá entender como, através da API do Undertow, é possível criar serviços utilizando Java e JavaScript. Profissionais que desejam aprender sobre o assunto podem adotar este artigo como ponto de partida para entender como migrar suas aplicações ou criá-las do zero utilizando essa ferramenta.

eventos e em processos não bloqueantes, ou seja, as requisições trabalham de maneira que o processo nunca fique parado esperando por um retorno de um banco de dados ou mesmo a leitura de um arquivo, assim como acontece com os novos containers Java, como o Vert.x.

O Undertow veio como uma resposta a essa nova maneira de desenvolver aplicações. Como veremos, ele fornece, de forma semelhante, os recursos que fizeram o Node.js ter a relevância que possui no mercado.

Contudo, apesar do espaço que essas ferramentas vêm conquistando, o modelo monolítico de containers continua tendo seu espaço, afinal, a adoção ou não deles depende não somente de uma premissa de tecnologia, mas também de um ponto de vista de gestão e cultura nas empresas. No momento em que é tomada a decisão de utilizar containers como o Undertow, que focam em programação reativa e microservices, isso não só acarreta na necessidade de conhecer a tecnologia em si, mas também em uma mudança na gestão e no modo de pensamento da equipe.

Portanto, é necessário que a empresa tenha mentalidade voltada para DevOps, porque há necessidade de gerenciar melhor toda a

estrutura na qual se baseia o desenvolvimento. Como exemplo, ações como o rollback de uma aplicação que apresentou problemas acabam não sendo mais um caminho de fácil acesso, pois, em sistemas que possuem vários serviços diferentes e dependentes uns dos outros, o rollback de um serviço pode acarretar na necessidade de realizar a mesma operação em vários outros. Dessa forma, quando utilizamos microservices pode ser mais fácil manter os serviços que estão funcionando da maneira esperada e lançar um novo pacote para corrigir um possível artefato que apresentou problemas.

Dado o alerta àqueles que querem entrar no mundo de microservices, é bom se habituar à ideia de mudar a sua forma de pensar. A princípio essa frase pode parecer um exagero, mas é a realidade, pois focar nesse novo modo de criar serviços exige uma maior organização da equipe, sendo necessário, também, saber o que as outras equipes estão criando para evitar a implementação de código duplicado. Outro item importante é o processo de deploy, que deve ser rápido e automatizado, pois cada serviço com problema pode acarretar na parada de vários outros.

Para evitar quedas e garantir uma alta disponibilidade, invista em monitoramento, afinal, se monitorar um único serviço monolítico já possuía seus custos e dificuldades, monitorar vários deles eleva consideravelmente essa complexidade. Esses são só alguns itens essenciais que precisam existir na cultura de sua empresa para que o trabalho com microservices seja realizado de forma eficiente.

Apesar de haver a necessidade de adequação, as vantagens de se trabalhar com microservices se sobressaem. O time como um todo sai mais fortalecido do processo de adequação, ganhando experiência e conhecimento, o que com certeza será um diferencial no dia a dia. Além da equipe, o próprio ecossistema da aplicação ganha em vários aspectos, como no tempo de disponibilidade, pois agora não teremos mais um monólito único representando toda a aplicação, mas sim pequenos serviços para atender a várias partes do negócio, e caso um deles fique indisponível, não significa que todo o sistema estará indisponível. Ademais, não há a obrigação de utilizar apenas uma linguagem ou plataforma. Serviços distintos podem ser desenvolvidos de maneiras distintas e, assim, resolver problemas através de tecnologias que atuem melhor em algum requisito específico.

O exemplo do JBoss AS se encaixa bem nesse caso. Há três anos esse produto possuía apenas duas versões: Community e Enterprise. Atualmente, oferece várias versões distintas, como o WildFly, que mesmo mantendo boa parte de sua estrutura, configuração e funcionamento fiel às suas versões mais antigas, tem seu core totalmente modificado. Seu web server, por exemplo, agora é o Undertow, enquanto nas versões mais antigas era utilizado um fork direto do Tomcat. Assim, desde 2013 o WildFly possui um web server novo, que fornece suporte a requisições não bloqueantes. Isso permite que requisições que exigem mais tempo do servidor, como um acesso a um banco de dados ou mesmo a leitura ou geração de um arquivo, sejam feitas em um pool de threads isolado e fora do ciclo principal de cada requisição.

Outro container que se beneficia das capacidades do Undertow como web server é o Swarm, que abordamos em outro artigo, publicado na edição 150 da Java Magazine.

## As vantagens do Undertow

Na documentação do Undertow, são listadas como suas principais características:

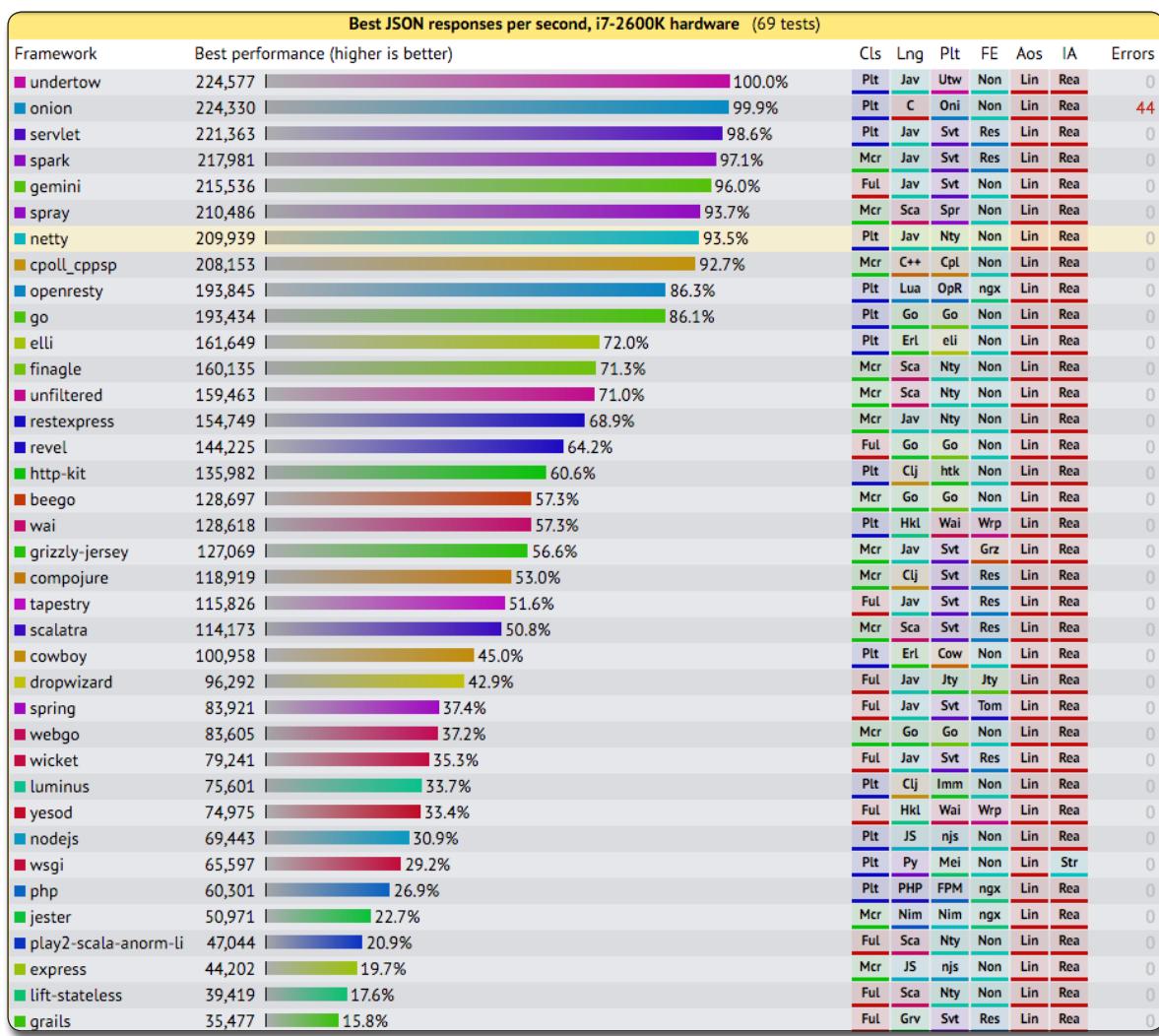
- A alta performance;
- Ser embeddable;
- O suporte a Servlets 3.1;
- O suporte a WebSockets;
- O proxy reverso.

A característica “alta performance” pode ser considerada um pouco abstrata, pois existem muitas situações diferentes que podem ocorrer durante uma requisição e que podem afetar diretamente o tempo de resposta. Por isso, a própria equipe do Undertow realizou, através do site [techempower.com](http://techempower.com), uma bateria de testes de performance com o Undertow e vários concorrentes. Em grande parte dos casos, levando em consideração também hardwares diferentes, o web server da Red Hat se mostrou superior, destacando-se, inclusive, na serialização e desserialização de objetos JSON, operações comumente utilizadas em serviços REST. Caso queira ver todos os testes realizados e a posição do container em cada um deles, na seção **Links** você encontrará o endereço para a matéria completa. A **Figura 1** demonstra o gráfico com a performance de uma requisição em JSON comparando o Undertow e vários concorrentes.

A segunda característica em nossa lista é ser embeddable. Assim, o Undertow pode ser anexado ao projeto e permitir que sua inicialização seja realizada junto com o start do projeto. Isso acaba por facilitar o deploy de novas instâncias e serviços, pois tanto o servidor quanto a aplicação podem ficar contidos em um único arquivo.

Antes de nos aprofundarmos no terceiro item da lista de características, precisamos entender o que a especificação de Servlets 3.1 trouxe, para só então entendermos os benefícios de se dar suporte a ela. A versão anterior de servlets, 3.0, nasceu como parte da Java EE 6 e teve como meta tornar mais fácil o seu uso. Nessa versão tivemos a adoção de novos recursos da linguagem, como anotações e generics, além da evolução da maneira de trabalhamos com a tecnologia. Assim, configurações antes só possíveis via *web.xml* foram disponibilizadas também através de anotações.

Já em 2013, na release 3.1, o foco passou a ser alguns recursos considerados fundamentais pelo time da Oracle. O principal deles foi a adoção de operações de I/O não bloqueantes, visto que na versão 3.0 era permitido requisições assíncronas, mas somente com recursos bloqueantes, o que restringia a performance das aplicações por fazer com que recursos tivessem que aguardar até que a requisição estivesse pronta. Na **Listagem 1** apresentamos um pequeno trecho de código que demonstra como funcionava a especificação de Servlet até então.



**Figura 1.** Comparativo de performance entre o Undertow e seus concorrentes

#### Listagem 1. Tratamento de uma requisição através de um servlet na versão 3.0.

```

1. public class TestServlet extends HttpServlet {
2.
3.     protected void doGet(HttpServletRequest request, HttpServletResponse response)
4.             throws IOException, ServletException {
5.
6.         ServletInputStream input = request.getInputStream();
7.         byte[] b = new byte[1024];
8.         int len = -1;
9.         while ((len = input.read(b)) != -1) {
10.             ...
11.         }
12.     }
13. }
```

#### Listagem 2. Tratamento de uma requisição através de um servlet na versão 3.1.

```

1. public class TestServlet extends HttpServlet {
2.
3.     protected void doGet(HttpServletRequest request, HttpServletResponse response)
4.             throws IOException, ServletException {
5.
6.         AsyncContext context = request.startAsync();
7.         ServletInputStream input = request.getInputStream();
8.         input.setReadListener(new MyReadListener(input, context));
9.
10.    }
11. }
```

Apesar de a implementação da classe ser um servlet simples, a linha 6 pode ser um problema. Existem várias situações que podem ocorrer nesse momento e forçar a thread a aguardar todos os dados chegarem para continuar sua execução, como um acesso ao banco de dados, a leitura de um arquivo, a consulta a outro serviço, etc. Ao utilizar os recursos da especificação 3.1, a nossa classe ficaria conforme a **Listagem 2**.

Na linha 6, criamos um **AsyncContext**, que será responsável por dizer quando terminamos de receber os dados e que a requisição pode retornar ao browser. Na linha 7, criamos um objeto da classe **ServletInputStream**, e, por fim, na linha 8, invocamos o novo método da classe **ServletInputStream**, que seta um listener responsável por ler os dados vindos através da variável **input**.

Vale destacar que existem dois listeners, representados por duas interfaces: uma para leitura, chamada **ReadListener**, e outra para escrita, chamada **WriteListener**. Elas são registradas na classe **ServletInputStream** através dos métodos **setReadListener()** e **setWriteListener()**, respectivamente, permitindo que todos os eventos de leitura e escrita sejam tratados de forma não bloqueante. Ambas as interfaces possuem métodos que servem de callback, e que são chamados quando o conteúdo estiver disponível para ser lido ou escrito sem nenhum tipo de bloqueio.

Com o objetivo de mostrar como os dados devem ser tratados pelos métodos da interface, vamos criar uma classe que implementará **ReadListener**, conforme mostra a **Listagem 3**.

**Listagem 3.** Implementação da interface ReadListener.

```
1. public class MyReadListener implements ReadListener {
2.
3.     private ServletInputStream input = null;
4.     private AsyncContext context = null;
5.
6.     public MyReadListener(ServletInputStream in, AsyncContext ac) {
7.         this.input = in;
8.         this.context = ac;
9.     }
10.
11.    @Override
12.    public void onDataAvailable() {
13.        try {
14.            StringBuilder sb = new StringBuilder();
15.            int len = -1;
16.            byte b[] = new byte[1024];
17.            while (input.isReady() && (len = input.read(b)) != -1) {
18.                String data = new String(b, 0, len);
19.                System.out.println("--> " + data);
20.            }
21.        } catch (IOException ex) {
22.            Logger.getLogger(MyReadListener.class.getName())
23.                .log(Level.SEVERE, null, ex);
24.        }
25.
26.        @Override
27.        public void onAllDataRead() {
28.            System.out.println("onAllDataRead");
29.            context.complete();
30.        }
31.
32.        @Override
33.        public void onError(Throwable t) {
34.            t.printStackTrace();
35.            context.complete();
36.        }
37.    }
```

Como podemos notar, a interface **ReadListener** possui três métodos que devem ser implementados. Além disso, declaramos na linha 6 um construtor que recebe dois parâmetros: um **ServletInputStream**, responsável por transportar os dados da requisição; e um **AsyncContext**, mencionado anteriormente e que é responsável por finalizar o processamento da requisição e retornar ao browser.

Dito isso, precisamos implementar os métodos que vão lidar com o ciclo de vida de nossas requisições.

No primeiro deles, **onDataAvailable()**, declarado na linha 12, implementamos a leitura dos dados. Ele só será acionado quando todos os dados puderem ser lidos sem nenhum tipo de bloqueio.

Já o segundo método, **onAllDataRead()**, declarado na linha 27, é acionado após a leitura de todos os dados da requisição. Nele, devemos implementar a operação que desejamos que seja executada antes de a requisição retornar ao browser. Na linha 29, invocamos o método **complete()** do objeto **AsyncContext**, liberando a requisição para retornar.

Por último, na linha 33, temos o método **onError()**, que só é invocado caso algum erro no processamento da requisição aconteça. Neste momento, vale destacar a necessidade de também invocar o método **complete()** para retornar a requisição (linha 35), da mesma maneira que fizemos na linha 29.

Saiba que tratar as requisições de forma assíncrona, utilizando as interfaces, em vez de tratá-las diretamente dentro do servlet, de forma síncrona, muda completamente o modo como o servidor lida com elas. O container não precisa manter recursos bloqueados enquanto cada uma das etapas é processada. Eles só serão alocaados quando forem realmente necessários e a requisição puder ser processada de uma vez. Assim, o suporte à versão 3.1 afeta diretamente na performance do Undertow, possibilitando que ele gerencie da melhor maneira os recursos e consiga lidar com mais requisições simultaneamente.

Na lista de benefícios temos ainda os WebSockets, algo muito bem-vindo para o desenvolvimento de aplicações. Implementado tanto no lado cliente quanto no lado servidor, esse protocolo permite que não somente o cliente invoque o servidor, mas também o caminho inverso, facilitando a troca de dados em tempo real. Outro ponto favorável é que atualmente a maioria dos browsers já dá suporte a esse protocolo. Assim, não precisamos nos preocupar em desenvolver algo que funcionará apenas em alguns navegadores.

Por último, citamos o proxy reverso. Ao contrário do proxy padrão, o proxy reverso tem a vantagem de não precisar de nenhum tipo de configuração no lado cliente. O serviço de proxy pode fazer requisições de conteúdo dentro da rede no lugar do cliente para um ou mais servidores responsáveis por tratar as requisições. Isso permite uma camada a mais de segurança em nosso ambiente, pois faz com que o cliente só tenha conhecimento de um servidor interno, dificultando, assim, o acesso a informações da arquitetura do sistema.

Agora que vimos um pouco sobre os principais motivos de o Undertow ser uma boa opção quando pensamos em um container para nossa aplicação web, vamos ver como podemos utilizá-lo e entender melhor seu funcionamento.

## Iniciando o desenvolvimento de uma aplicação com o Undertow

Para iniciar o desenvolvimento, vamos criar um projeto simples baseado no archetype QuickStart do Maven. Portanto, abra o prompt de comando no Windows, ou o terminal, caso esteja em um Linux ou Mac, navegue até a pasta onde deseja que seu projeto seja criado e execute a seguinte linha de comando:

```
mvn archetype:generate -DgroupId=br.com.devmedia -DartifactId>HelloUndertow  
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Depois, abra o Eclipse e importe o projeto que acabamos de criar. Logo após, precisamos adicionar o Undertow ao projeto. Para isso, basta adicionar as seguintes dependências:

- Core:** Como o próprio nome sugere, essa dependência é responsável por todo o funcionamento básico do Undertow, já dando suporte a requisições não bloqueantes;
- Servlet:** Utilize essa dependência caso seu projeto tenha a necessidade de trabalhar com servlets;
- WebSockets JSR:** Através dessa dependência é possível criar serviços que necessitam de uma comunicação de duas vias entre servidor e cliente, como serviços de chat e aplicações em tempo real.

A **Listagem 4** mostra a declaração das dependências que devem ser adicionadas ao projeto. Assim, garantimos que todas as funcionalidades do Undertow estarão disponíveis. Neste artigo foi utilizada a versão 1.3.21.

Além disso, certifique-se de que seu computador possua no mínimo a versão 7 do JDK instalada, assim como pelo menos a versão 3.1 do Maven.

Atendidos esses requisitos, podemos ver como iniciar uma instância do Undertow. Abra a classe **App**, criada pelo Maven ao gerar o projeto através do archetype selecionado, e substitua o código do método **main()** pelo conteúdo da **Listagem 5**.

#### Listagem 4. Dependências do Undertow a serem inseridas no projeto.

```
1. <dependency>  
2.   <groupId>io.undertow</groupId>  
3.   <artifactId>undertow-core</artifactId>  
4.   <version>${undertow.version}</version>  
5. </dependency>  
6. <dependency>  
7.   <groupId>io.undertow</groupId>  
8.   <artifactId>undertow-servlet</artifactId>  
9.   <version>${undertow.version}</version>  
10. </dependency>  
11. <dependency>  
12.   <groupId>io.undertow</groupId>  
13.   <artifactId>undertow-websockets-jsr</artifactId>  
14.   <version>${undertow.version}</version>  
15. </dependency>
```

#### Listagem 5. Método para criar uma instância do Undertow.

```
1. public static void main(final String[] args) {  
2.     Undertow server = Undertow.builder()  
3.         .addHttpListener(8080, "localhost")  
4.         .setHandler(new HttpHandler() {  
5.             public void handleRequest(final HttpServerExchange exchange)  
throws Exception {  
6.                 exchange.getResponseHeaders().put(Headers.CONTENT_TYPE,  
"text/plain");  
7.                 exchange.getResponseBody().send("Hello World");  
8.             }  
9.         }).build();  
10.    server.start();  
11. }
```

Na linha 2, criamos uma instância do Undertow através do método **builder()**, que encapsula toda a parte burocrática que existe ao iniciar o container. Por isso não precisamos nos preocupar com o XNIO, responsável por criar uma camada de abstração para o Java NIO. Se optar por não utilizar a classe **Undertow**, será necessária toda uma compreensão de como funciona a API XNIO para lidar com os vários tipos de conexões possíveis através de HTTP ou AJP, por exemplo, que vão além do escopo deste artigo. Assim, utilizamos essa abstração para focarmos nos recursos do Undertow especificamente.

Voltando ao código, na linha 3 adicionamos um listener na porta 8080 para podermos acessar o servidor através da URL *http://localhost:8080*. Na sequência, na linha 4, setamos um **HttpHandler**, que possibilita ao Undertow tratar uma determinada requisição HTTP. Na linha 5, sobrescrevemos o método **handleRequest()**, que, como veremos neste e nos próximos exemplos, além de lidar com as requisições, pode desviar o fluxo de nossas requisições conforme a necessidade. Na linha 6, iniciamos o preparo da resposta que será enviada ao browser através do objeto da classe **HttpServerExchange**. Primeiro, devemos adicionar ao header da resposta a informação de que vamos retornar um texto simples. Para isso, acessamos os headers através do método **getHeaders()**, e, então, ainda na linha 6, colocamos a identificação do tipo de retorno através do método **put()**. Para concluir o tratamento da requisição, enviamos a resposta na linha 7, adicionando o texto "Hello World" através do método **send()**.

Agora que já tratamos como vamos responder às chamadas ao nosso serviço, precisamos gerar o build do mesmo conforme a linha 9, através do método **build()**, e finalizar o processo de criação de nossa instância do Undertow, dando o start em nosso serviço através do método **start()** (linha 10).

Para rodar o projeto no Eclipse, basta executá-lo normalmente através do botão *Play*, do mesmo modo que um projeto Java padrão.

Feito isso, inicie o navegador de sua escolha e acesse a URL *http://localhost:8080* para ver a nossa mensagem ser exibida. Além disso, você pode gerar um JAR do projeto e executá-lo diretamente via linha de comando, ou mesmo via Maven, prática bem comum quando utilizamos integração contínua através de tasks. Para isso, no entanto, precisamos adicionar o plugin **exec**. Sendo assim, abra o *pom.xml* no Eclipse e adicione o trecho de código da **Listagem 6**.

Certifique-se de interromper o projeto que estava em execução no Eclipse, pois, como ambos estão configurados para rodar na porta 8080, se forem executados ao mesmo tempo ocorrerá um erro. Então, abra o terminal no Linux/Mac ou o prompt no Windows, navegue até a pasta do projeto e execute o comando:

```
mvn exec:java
```

Em seguida, ao acessar o endereço indicado no browser, você verá a mesma mensagem, o que indica que o servidor está no ar.

Com isso, nosso primeiro exemplo está pronto. Nossa classe irá tratar todas as requisições através da classe **HttpHandler**.

**Listagem 6.** Plugin que permite executar o projeto diretamente do Maven.

```
1. <build>
2.   <plugins>
3.     <plugin>
4.       <groupId>org.codehaus.mojo</groupId>
5.       <artifactId>exec-maven-plugin</artifactId>
6.       <version>1.2.1</version>
7.       <executions>
8.         <execution>
9.           <goals>
10.             <goal>java</goal>
11.           </goals>
12.         </execution>
13.       </executions>
14.     <configuration>
15.       <mainClass>br.com.devmedia.App</mainClass>
16.     </configuration>
17.   </plugin>
18. </plugins>
19. </build>
```

Note que o tratamento dado para todas as requisições é o mesmo, indiferente da URL acessada. Para saber qual caminho o usuário está acessando, podemos fazer uso do método `getHttpRequest()`, e, assim, gerar uma saída para cada um desses caminhos.

A classe `HttpHandler` também nos permite adicionar comportamentos default para as requisições. Através do método `handleRequest()`, por exemplo, conseguimos manipular todo o ciclo de vida delas. Esse método recebe como parâmetro um objeto `HttpServerExchange`, que utilizamos para acessar todas as informações de cada requisição.

## Utilizando fluxos alternativos

No próximo exemplo vamos montar uma página default que será criada em tempo de execução caso determinada requisição retorne o status code 500, informando ao usuário que um erro ocorreu. Nesse caso, primeiro vamos lidar com o fluxo da requisição e como podemos alterá-lo caso um erro ocorra. Para isso, vamos utilizar duas classes: uma que adicionará o comportamento que desejamos no caso de erro e outra que será responsável por executar o fluxo normal da requisição caso tudo ocorra bem. A **Listagem 7** mostra a implementação da classe `Errorhandler`, responsável por tratar o erro, caso ocorra, ou encaminhar a requisição para o padrão.

Como mostra a linha 11, criamos uma variável `final`, do tipo da interface `HttpHandler`, que será utilizada para invocar o método `handleRequest()`, na linha 36. Esse código, no entanto, somente será executado se a requisição não apresentar nenhum problema. Na linha 13, o construtor obriga qualquer um que deseje criar uma instância da classe a passar um objeto que implemente a interface `HttpHandler`.

Feito isso, precisamos implementar o método que vai, de fato, tratar a requisição: o `handleRequest()`, conforme a linha 17. Na linha 18, adicionamos a interface `DefaultResponseListener` ao objeto `exchange`, do tipo `HttpServerExchange`. Quando isso acontece, essa interface nos fornece a possibilidade de tratar requisições através de um ciclo normal de requisição/resposta ou devolver

uma resposta padrão caso ocorra algum tipo de comportamento não previsto. Em nosso caso, será criada uma resposta padrão caso seja detectado um erro durante o processamento da requisição.

O próximo passo, linha 19, é implementar o método `handleDefaultResponse()`, declarado na interface `DefaultResponseListener`. Esse método nos permite verificar o que ocorreu com cada requisição. Na linha 20, verificamos se o canal de resposta não está mais disponível. Se a assertiva for verdadeira, não devemos seguir com o tratamento da requisição, pois o mesmo já foi feito e não precisamos mais nos preocupar com ela, retornando o valor `false` na linha 21. Já na linha 24, verificamos se o objeto `exchange` possui o status code 500. Se esse for o caso, montamos uma página de erro para devolver ao usuário.

**Listagem 7.** Código da classe Errorhandler. Adiciona comportamento padrão no caso de erro.

```
1. package br.com.devmedia;
2.
3. import io.undertow.io.Sender;
4. import io.undertow.server.DefaultResponseListener;
5. import io.undertow.server.HttpHandler;
6. import io.undertow.server.HttpServerExchange;
7. import io.undertow.util.Headers;
8.
9. public class Errorhandler implements HttpHandler {
10.
11.   private final HttpHandler next;
12.
13.   public Errorhandler (final HttpHandler next) {
14.     this.next = next;
15.   }
16.
17.   public void handleRequest(final HttpServerExchange exchange) throws
Exception {
18.     exchange.addDefaultResponseListener(new DefaultResponseListener() {
19.       public boolean handleDefaultResponse(final HttpServerExchange exchange) {
20.         if (!exchange.isResponseChannelAvailable()) {
21.           return false;
22.         }
23.         //Exchange.setStatusCode(500);
24.         if (exchange.getStatusCode() == 500){
25.           final String errorPage = "<html><head><title>Error</title>
</head><body>Internal Error</body></html>";
26.           exchange.getResponseHeaders().put(Headers.CONTENT_LENGTH,
"" + errorPage.length());
27.           exchange.getResponseHeaders().put(Headers.CONTENT_TYPE,
"text/html");
28.           Sender sender = exchange.get_RESPONSESender();
29.           sender.send(errorPage);
30.           return true;
31.         }
32.         return false;
33.       }
34.     });
35.     // exchange.endExchange();
36.     this.next.handleRequest(exchange);
37.   }
38. }
```

Nosso próximo passo, nas linhas 25, 26 e 27, é montar uma string contendo o HTML que será retornado. Em seguida, na linha 28, obtemos a referência do objeto `sender`, contido no `exchange`, e na linha 29 enviamos o HTML como resposta através do método `send()`. Agora, na linha 30, como encontramos um erro, retornamos o valor `true`, que indica que nossa resposta é uma resposta padrão,

e não uma resposta que seria retornada caso o ciclo de vida da requisição tivesse sido concluído com sucesso.

Caso tudo ocorra bem com a requisição, o nosso **DefaultResponseListener** nunca será acionado. Sendo assim, na linha 36, pegamos nosso objeto **HttpHandler**, criado na linha 11, e invocamos o método **handleRequest()**. Dessa maneira, transferimos para o objeto **next** a responsabilidade de tratar a requisição.

Depois de criar a classe que lida com requisições que apresentem algum tipo de erro, vamos implementar a classe que irá processar a requisição em casos de não encontrarmos nenhum erro. Essa é a classe que será responsável por tratar a requisição e gerar uma resposta para o browser caso tudo ocorra conforme esperado. O código da classe **MyHandler** é mostrado na **Listagem 8**.

Observando a implementação do método **handleRequest()**, declarado na linha 9, notamos que seu conteúdo é o mesmo do nosso primeiro exemplo, apresentado na **Listagem 5**.

Encerrada essa implementação, para ver nosso exemplo em execução, precisamos realizar alguns ajustes na classe **App** criada anteriormente (vide **Listagem 9**).

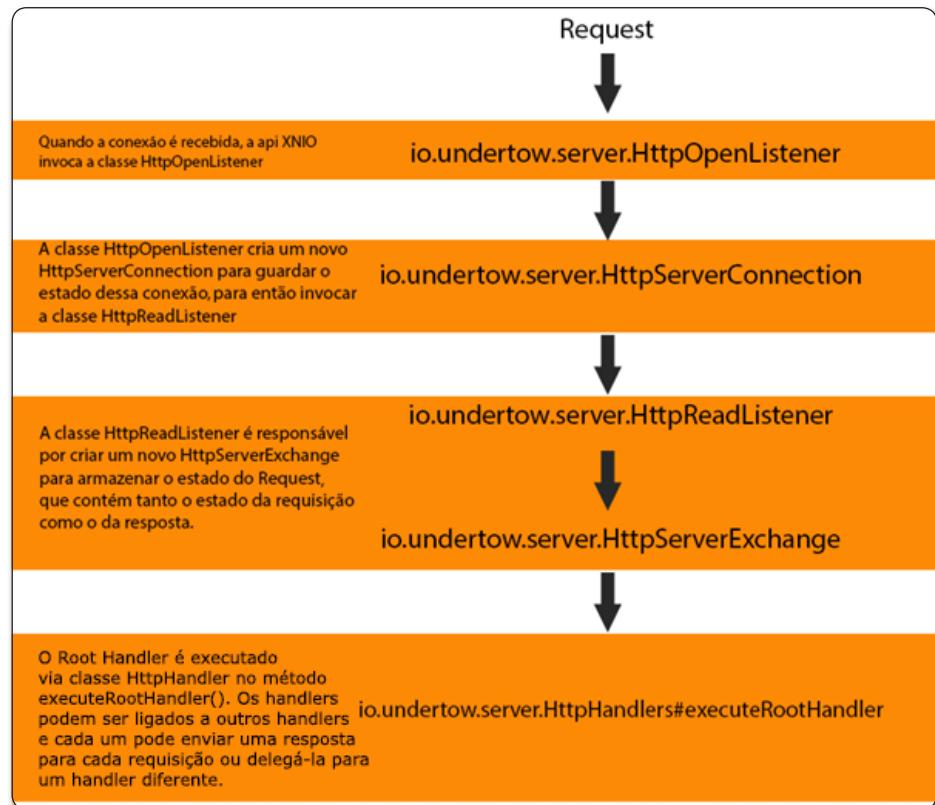
Na linha 7, criamos uma instância da classe **MyHandler**. Na linha 8, instanciamos um objeto da classe **Errorhandler** e em seu construtor passamos o objeto **myHandler**. Isso foi feito porque utilizamos esse objeto para invocar o método **handleRequest()** da nossa classe, caso não ocorra nenhum erro na requisição. A partir da linha 9, iniciamos normalmente a criação de uma instância do Undertow.

Caso você execute o projeto nesse momento, a aplicação continua retornando o mesmo Hello World da primeira versão (mas dessa vez seguido do caminho da requisição). No entanto, se quiser simular um erro descomente as linhas 23 e 35 da classe **Errorhandler** (**Listagem 7**). Com isso, finalizamos o processamento da requisição e retornamos um código de erro. Assim, **DefaultResponseListener** será acionado e a mensagem de erro padrão, criada por nós, será exibida na tela.

## Entendendo o fluxo de uma requisição no Undertow

Já vimos dois exemplos de como tratar requests no Undertow e como inserir comportamentos padrão neles. Contudo, existem vários caminhos por onde um request, no Undertow, pode seguir. Assim, vamos observar o seu fluxo e compreender seu ciclo de vida. Na **Figura 2** podemos observar cada passo e quais classes são acionadas durante o processo.

Como pode ser verificado, a requisição segue um fluxo único até o método **executeRootHandler()**.



**Figura 2.** Fluxo de uma requisição ao ser recebida pelo Undertow

**Listagem 8.** Código da classe referente ao Handler da requisição caso nenhum erro ocorra.

```

1. package br.com.devmedia;
2.
3. import io.undertow.server.HttpHandler;
4. import io.undertow.server.HttpServerExchange;
5. import io.undertow.util.Headers;
6.
7. public class MyHandler implements HttpHandler{
8.
9.     public void handleRequest(HttpServerExchange exchange) throws Exception {
10.         exchange.getResponseHeaders().put(Headers.CONTENT_TYPE, "text/plain");
11.         exchange.getResponseBody().send("Hello World "+exchange
12.             .getRequestPath());
12.     }
13. }
14. }
```

**Listagem 9.** Código da classe App modificada para incluir o Errorhandler.

```

1. package br.com.devmedia;
2.
3. import io.undertow.Undertow;
4.
5. public class App {
6.     public static void main(final String[] args) {
7.         MyHandler myHandler = new MyHandler();
8.         Errorhandler errorHandler = new Errorhandler(myHandler);
9.         Undertow server = Undertow.builder().addHttpListener(8080, "localhost")
10.             .setHandler(errorHandler).build();
11.         server.start();
12.     }
13. }
```

Após essa etapa, conforme a documentação do Undertow, algumas coisas podem ocorrer a cada requisição:

- O objeto **Exchange** pode ser finalizado. Isso acontece quando tanto a requisição quanto a resposta são fechadas. Caso o tamanho do conteúdo tenha sido setado, o canal será fechado automaticamente quando todos os dados forem lidos, e também pode ser fechado de forma explícita, através do método **endExchange()**, da classe **HttpServerExchange**. Caso nenhum dado seja escrito, o listener default gerado com o **Exchange** terá a oportunidade de criar uma resposta padrão, como uma página de erro. Então, quando o **Exchange** atual terminar, o seu listener interno irá executar finalizando qualquer operação. Por fim, a última operação realizada pelo **Exchange** indicará que pode ser iniciado o processamento da próxima requisição na conexão;
- O objeto **Exchange** pode ser enviado para ser processado fora da thread principal, através do método **dispatch()** da classe **HttpServerExchange**. Esse método é similar ao **startAsync()** dos servlets. Assim como mostramos no exemplo utilizando servlets, quando os dados terminarem de ser recebidos, como em um upload de arquivos, a tarefa é processada (por exemplo, escrever o arquivo recebido em disco), dando a possibilidade de executá-la em uma classe definida por você, bastando que a mesma implemente a interface **Executor** do pacote **java.util.concurrent**. Caso nenhum executor seja informado, o objeto **Exchange** rodará em um worker do XNIO. O mais comum para esse tipo de tarefa, que pode bloquear recursos por um período considerável de tempo, é mover a tarefa de uma thread IO (onde o bloqueio não é permitido) para um worker que pode ser bloqueado. Assim, o ciclo de requisições não precisa ficar parado aguardando a requisição atual terminar. A **Listagem 10** demonstra como é possível mover o objeto **Exchange** da thread principal.

#### Listagem 10. Verificação do tipo de thread e disparando a tarefa como assíncrona.

```
1. public void handleRequest(final HttpServerExchange exchange) throws Exception {  
2.     if (exchange.isInIoThread()) {  
3.         exchange.dispatch(this);  
4.         return;  
5.     }  
6.     //resto do código  
7. }
```

Como é possível observar na linha 2, o processo para verificar se o objeto **exchange** se encontra em uma thread de I/O é bem simples, basta invocar o método **isInIoThread()** do próprio objeto. Depois, conforme a linha 3, caso a assertiva da linha 2 seja verdadeira, invocamos o método **dispatch()** passando a referência **this** como parâmetro;

- A escrita e a leitura podem ser retomadas na requisição ou em um canal de resposta. Internamente, o tratamento é similar à quando invocamos o método **dispatch()** do objeto **Exchange** — quando a pilha da chamada retorna, o canal de resposta do objeto **Exchange** é notificado sobre os eventos de I/O. A razão pela qual a operação só terá efeito após o término da chamada é assegurar

que nunca vamos ter múltiplas threads atuando no mesmo objeto da classe **HttpServerExchange**;

- A chamada pode retornar sem o objeto **Exchange** ser enviado. Se isso acontecer, o método **endExchange()** da classe **HttpServerExchange** vai ser chamado e a request será finalizada;
- Uma exceção pode ser lançada. Se isso acontecer, toda a pilha de requisição é encerrada, e o objeto da classe **HttpServerExchange** retornará com um status code 500. Por esse motivo, na linha 24 da **Listagem 7** fazemos essa verificação. Se a assertiva for verdadeira, retornamos a página de erro.

## Entendendo os listeners

Um conceito importante quando trabalhamos com o Undertow são os listeners. Eles representam o ponto de entrada de uma aplicação desenvolvida para esse web server. Assim, ao chegarem ao servidor as requisições serão recebidas através de um listener, responsável por traduzi-las em um objeto da classe **HttpServerExchange**. A partir disso, o processamento da requisição será realizado e, então, o resultado será retornado ao cliente.

O Undertow fornece quatro tipos de listeners: HTTP, AJP, SPDY e HTTP2. Além desses, o HTTPS também é fornecido, através de um listener HTTP com conexão SSL habilitada. Essas opções podem ser adicionadas ao Undertow da mesma maneira que configuramos o listener HTTP em nossos exemplos: através da classe **Builder**, como mostra a linha 9 na **Listagem 9**.

Nesse modelo, cada listener possui uma ligação com uma instância de um **Worker** da API XNIO. Por padrão, uma única instância é compartilhada entre os listeners, mas é possível criar um novo **Worker** para cada listener. Assim, uma instância de cada **Worker** gerencia as threads de I/O de cada listener, além do pool de threads bloqueadas de cada tarefa.

Além disso, para aprimorar essa organização, é possível configurar um **Worker** de várias maneiras, onde cada configuração afeta diretamente o comportamento dos listeners correspondentes. Esses ajustes são feitos através do método **setWorkerOption()** de um objeto **Builder**. Na documentação oficial, duas delas recebem destaque por influenciarem diretamente na performance do servidor (veja a **Tabela 1**).

## Utilizando servlets em seus projetos

Tudo que vimos até o momento envolve o uso de recursos do próprio Undertow. Como mostrado nos exemplos, é possível lidar com requisições utilizando handlers e listeners do Undertow, que possibilitam um total controle do fluxo de nossa aplicação de uma maneira simplificada. Podemos tratar, inclusive, problemas em classes distintas, desacoplando nosso código. Contudo, nem sempre temos a possibilidade de começar um projeto do zero ou mesmo não queremos iniciar todo o desenvolvimento usando uma API totalmente nova. Por esses e tantos outros motivos, o Undertow também dá suporte a servlets de uma maneira bem similar ao que você faria caso estivesse utilizando servidores conhecidos, como o JBoss ou o Tomcat. Deste modo, veremos, a partir de agora, como podemos utilizar servlets em nosso projeto.

WORKER_IO_THREADS	Representa o número de threads de I/O a serem criadas. Elas executam tarefas não bloqueantes e nunca devem executar operações bloqueantes, pois são responsáveis por múltiplas conexões. Se as utilizarmos para tarefas bloqueantes, enquanto uma estiver bloqueada, as conexões a ela relacionadas terão que aguardar. O aconselhado é uma thread por CPU.
WORKER_TASK_CORE_THREADS	Número de threads definidas para tarefas bloqueantes. Quando você executa tarefas bloqueantes, como chamadas de servlets, threads desse pool serão usadas. Não se costuma oferecer um valor default para esse atributo, pois depende muito de como sua aplicação trabalha. Contudo, podemos dizer que se deve iniciar com um valor alto, como 10 threads por CPU.

**Tabela 1.** Configurações dos tipos de threads

#### Listagem 11. Servlet de exemplo que recebe um parâmetro em sua inicialização.

```

1. package br.com.devmedia;
2.
3. import java.io.IOException;
4.
5. import javax.servlet.ServletException;
6. import javax.servlet.ServletConfig;
7. import javax.servlet.http.HttpServlet;
8. import javax.servlet.http.HttpServletRequest;
9. import javax.servlet.http.HttpServletResponse;
10.
11. public class ExampleServlet extends HttpServlet{
12.     private static final long serialVersionUID = 1L;
13.     private String initMessage;
14.
15.     @Override
16.     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
17.         resp.getWriter().write("Mensagem recebida como parâmetro inicial:
"+this.initMessage);
18.     }
19.     @Override
20.     public void init(ServletConfig config) throws ServletException {
21.         super.init(config);
22.         this.initMessage = config.getInitParameter("message");
23.     }
24. }
```

Nosso primeiro passo é criar um servlet que recebe como parâmetro de inicialização uma mensagem. Essa mensagem demonstrará que podemos ter a mesma interação com o servlet sem ter um *web.xml* em nosso projeto. Sendo assim, crie uma classe conforme a **Listagem 11**.

Note que o servlet implementa o método **doGet()**, na linha 16. Nele iremos referenciar a variável de classe **initMessage**, declarada na linha 13, que será usada para devolver a mensagem ao browser. O valor da variável **initMessage** é definido pelo método **init()**, executado na inicialização do servlet. A partir desse método, declarado na linha 20, podemos receber informações que seriam carregadas diretamente do *web.xml* do projeto.

Contudo, quando utilizamos o Undertow não temos esse arquivo, mas mesmo assim podemos passar informações pré-definidas ao servlet. Essa tarefa pode ser realizada através do nosso código, isto é, as informações que definirmos podem ser carregadas no método **init()**.

Para isso, no momento que criamos a instância do Undertow, temos a possibilidade de carregar também os parâmetros. A **Listagem 12** demonstra como passar alguns parâmetros para o servlet.

#### Listagem 12. Carregando informações iniciais em nosso servlet.

```

1. package br.com.devmedia;
2.
3. import javax.servlet.ServletException;
4. import io.undertow.Handlers;
5. import io.undertow.Undertow;
6. import io.undertow.server.handlers.PathHandler;
7. import io.undertow.servlet.Servlets;
8. import io.undertow.servlet.api.DeploymentInfo;
9. import io.undertow.servlet.api.DeploymentManager;
10. import io.undertow.servlet.api.ServletInfo;
11.
12. public class App {
13.     public static final String appPath = "/devmedia";
14.
15.     public static void main(final String[] args) {
16.         try{
17.             DeploymentInfo servletBuilder = Servlets.deployment()
18.                 .setClassLoader(App.class.getClassLoader())
19.                 .setContextPath(appPath)
20.                 .setDeploymentName("test.war");
21.
22.             ServletInfo servlet1 = Servlets.servlet("MessageServlet",
23.                 ExampleServlet.class);
24.             servlet1.addInitParam("message", "Hello World");
25.             servlet1.addMapping("/myservlet1");
26.             servletBuilder.addServlets(servlet1);
27.
28.             ServletInfo servlet2 = Servlets.servlet("MyServlet", ExampleServlet.class);
29.             servlet2.addInitParam("message", "MyServlet");
30.             servlet2.addMapping("/myservlet2");
31.             servletBuilder.addServlets(servlet2);
32.
33.             DeploymentManager manager = Servlets.defaultContainer().
34.                 addDeployment(servletBuilder);
35.             manager.deploy();
36.
37.             PathHandler path = Handlers.path(Handlers.redirect(appPath))
38.                 .addPrefixPath(appPath, manager.start());
39.
40.             Undertow server = Undertow.builder()
41.                 .addHttpListener(8080, "localhost")
42.                 .setHandler(path)
43.                 .build();
44.             server.start();
45.         }catch (ServletException e) {
46.             e.printStackTrace();
47.         }
48.     }
49. }
```

Na linha 13, declaramos a variável **appPath**, que guarda o caminho que será registrado no servidor para a nossa aplicação. Para realizar o deploy do servlet no Undertow, utilizamos a interface **DeploymentManager** na linha 32. Porém, como ela pode ocasionalmente gerar uma exceção no método **deploy()**, na linha 33, colocamos nosso

código dentro de um bloco `try`, declarado na linha 16. Na linha 17, iniciamos o processo para gerar o deploy no Undertow. Para isso, utilizamos a classe `Servlets`, que nos ajuda a criar o objeto da classe `DeploymentInfo` e definir alguns parâmetros, como o caminho de nossa aplicação, na linha 19, e setar o nome do WAR que será criado em memória, na linha 20.

Estruturado o WAR que irá conter nosso projeto, podemos instanciar dois servlets, ambos baseados na mesma classe que implementamos na **Listagem 11**, mas que serão mapeados em URLs diferentes para que possamos distinguir quando um ou outro for acessado. Sendo assim, vamos informar valores diferentes no parâmetro inicial de cada classe.

O próximo passo, na linha 22, é criar um objeto da classe `ServletInfo`. Através de seu método `servlet()` criamos o primeiro servlet, passando como parâmetros o nome e a referência da classe `ExampleServlet` (**Listagem 11**). Na linha 24, especificamos o caminho que será monitorado pelo container e que usaremos para acessar o servlet; no caso, `myservlet1`. Feito isso, precisamos apenas adicionar nosso servlet à variável  `servletBuilder`, na linha 25, incluindo o servlet ao deploy que será realizado.

Partindo para o segundo servlet, da linha 27 à linha 30, repetimos o processo que acabamos de fazer, alterando somente os valores de alguns atributos. Com isso, temos dois servlets baseados na mesma classe, mas que recebem valores iniciais diferentes e respondem a URLs diferentes.

Agora, precisamos criar um objeto da classe `DeploymentManager`, que nos permite realizar o deploy do nosso objeto `DeploymentInfo` conforme o mostrado nas linhas 32 e 33. O método `deploy()`, na realidade, não inicia nenhum processo de inicialização do projeto no container. Ele apenas cria os metadados que auxiliarão a efetuar a operação que realizaremos em seguida. Assim, terminamos o preparo de nosso deploy.

Na linha 35, por sua vez, criamos uma instância da classe `PathHandler`, que mapeia o path registrado para nossa aplicação, definido na linha 13 de nossa classe. Ao invocar o método `start()` do objeto `manager` (vide linha 36), damos início ao processo de deploy. Já na linha 40, utilizamos um objeto da classe `PathHandler` para fazer um redirect. Desse modo, quando iniciarmos nossa aplicação, ele redirecionará automaticamente para `http://localhost:8080/devmedia`. Essa classe, além do redirect, possibilita executar várias ações nas nossas aplicações, como o bloqueio ou permissão de acesso a determinados IPs e alteração dos response headers no retorno das chamadas recebidas pelo Undertow.

Por fim, da linha 38 à linha 42, assim como já explicado nos outros exemplos, preparamos o Undertow para executar. O destaque aqui vai para a linha 40, onde passamos como `Handler` a variável `path` criada na linha 35 e que contém o nosso `manager`. Ele será responsável por passar ao container todas as informações do caminho de nosso deploy.

Com isso, se executarmos nosso projeto pelo Eclipse e, no browser, acessarmos a URL `localhost:8080`, seremos redirecionados para a URL da nossa aplicação, `localhost:8080/devmedia/`. Contudo, nenhum servlet será acionado e o que você verá será somente

uma resposta padrão do Undertow, dizendo que nada pode ser encontrado naquela URL. Para visualizarmos os servlets, temos que acessar uma das duas URLs que definimos: `localhost:8080/devmedia/myservlet2` ou `localhost:8080/devmedia/myservlet1`. Cada uma delas retornará uma das mensagens definidas nas linhas 22 e 27.

## Utilizando JavaScript para criar aplicações com o Undertow

O JavaScript é uma linguagem que vem ganhando cada vez mais espaço. Pensando nisso, o Undertow oferece a possibilidade de você criar aplicações utilizando uma API desenvolvida nessa linguagem. O mais interessante é que mesmo programando todo o back-end em JavaScript, você terá acesso a recursos como chamadas REST, WebSockets, JDBC, hot reload e integração com o Java EE, incluindo recursos como injeção de dependências.

Para demonstrar a facilidade de criação de um projeto do zero utilizando JavaScript, crie uma pasta em qualquer local do seu computador e dê o nome de “`UndertowJS`”. Em seguida, nessa pasta, crie um arquivo chamado `app.js` e uma nova pasta, chamada `WEB-INF`. O código a ser inserido nesse arquivo é apresentado na **Listagem 13**.

**Listagem 13.** Registrando um serviço no Undertow com JavaScript.

```
1. $undertow.onGet("/hello", {headers: {"content-type": "text/plain"}},
2.   [function ($exchange) {
3.     return "Hello World UndertowJS";
4.   }])
```

Nesse código é possível visualizar como é trivial criar um serviço em JavaScript. Na linha 1, temos o objeto `$undertow` representando uma instância do próprio container. Ainda nessa linha, registramos um endpoint HTTP com o método `onGet()`, informando no primeiro parâmetro que nosso serviço deve ser acionado quando a URL `/hello` for acessada. No segundo parâmetro, criamos um objeto para informar que o tipo de retorno será um texto. Logo após, na linha 2, no terceiro parâmetro, informamos um array, cuja única posição especifica uma função que recebe como parâmetro um objeto `exchange`, o mesmo que vimos nos exemplos anteriores. Por último, na linha 3, dentro da função, apenas retornamos uma mensagem para o cliente. Vale salientar que essa função será chamada automaticamente pelo Undertow quando ele receber uma requisição.

Com o código do serviço pronto, no entanto, ainda precisamos de alguns arquivos e configurações. Assim, dentro da pasta `WEB-INF`, crie um arquivo chamado `undertow-scripts.conf` e adicione a seguinte linha ao mesmo:

`app.js`

Como você pode estar pensando, nesse arquivo, mapeado automaticamente pelo Undertow, é feita a verificação dos arquivos JavaScript que possuem serviços registrados. Sendo assim, basta informar nele o caminho para cada um desses arquivos.

Como só temos um arquivo, que fica na raiz do projeto, só precisamos informar seu nome.

Basicamente, para fazer nosso projeto funcionar, precisamos somente desse código. Contudo, para fazer com que o WildFly realize o deploy do nosso projeto automaticamente a cada vez que um arquivo *js* for alterado, precisamos criar mais um arquivo na pasta *WEB-INF*, chamado *undertow-external-mounts.conf*, e em seu conteúdo especificar o caminho para a pasta raiz do projeto. Dessa forma, o projeto estará pronto. O que precisamos agora é copiar a pasta *UndertowJS* e depois acessar o local onde o WildFly 10 foi instalado. Em seguida, no diretório *standalone/deployments/*, cole a pasta copiada.

Voltando à pasta raiz do WildFly, acesse a pasta *bin* e clique duas vezes no arquivo *standalone.bat*, caso esteja no Windows, para iniciar o servidor de aplicação. Caso esteja no Mac/Linux, abra o terminal, navegue até a mesma pasta e execute o arquivo *standalone.sh*. Logo após, basta iniciar o browser e acessar <http://localhost:8080/UndertowJS/hello> para ver a mensagem que definimos em nossa função.

Como você pode notar, parece que essa funcionalidade de criar projetos em JavaScript é uma funcionalidade do WildFly 10, mas na realidade ela só é possível por causa do Undertow. Uma prova disso é o próprio objeto **exchange**, que faz parte da API desse web server.

Na seção **Links** você encontrará um endereço para o repositório do desenvolvedor Stuart Douglas, que faz parte tanto do projeto Undertow quanto do projeto WildFly Swarm. Nesse repositório estão disponíveis alguns exemplos de implementações em JavaScript para o Undertow que vão de simples serviços REST a exemplos mais complexos, com acesso a bancos de dados e classes Java nativas.

Enfim, seja em JavaScript ou Java, o Undertow é uma ótima escolha para a construção de serviços. Através de sua API, temos a possibilidade de integração com os mais variados recursos do

mundo Java. Ademais, some isso a sua velocidade e versatilidade e verá que com ele você terá também uma ótima ferramenta de apoio para seus projetos. Portanto, não deixe de aprofundar seus estudos nessa tecnologia, pois estamos longe de esgotar o assunto. Para isso, recomendamos a leitura da documentação oficial e também dos exemplos postados pela própria equipe do Undertow. Bons estudos!

## Autor



### Joel Backschat

[joel@cafecomjava.com.br](mailto:joel@cafecomjava.com.br)

Bacharel em Sistemas da Informação pela Universidade da Região de Joinville, possui certificação SCJP e Adobe FLEX. Já trabalhou em empresas como TOTVS e Supero. Desde 2014 é arquiteto de software da Fcamara Formação e Consultoria nas áreas de inovação e logística portuária. Entusiasta de novas tecnologias, mantém o site [cafecomjava.com.br](http://cafecomjava.com.br) para compartilhar suas experiências, que vão do hardware ao software.



## Links:

### Manifesto reativo.

<http://www.reactivemanifesto.org/>

### Exemplos utilizando Undertow.

<https://github.com/stuardouglas/undertow.js-examples>

### Resultados dos testes de benchmark utilizando o Undertow.

<http://www.techempower.com/benchmarks/#section=data-r6&hw=i7&test=json>

### Documentação oficial do Undertow.

<http://undertow.io/undertow-docs/undertow-docs-1.3.0>

### Ciclo de vida das requisições no Undertow.

<http://undertow.io/undertow-docs/undertow-docs-1.3.0/undertow-request-lifecycle.html>

# Modularização dinâmica em Java com OSGi

Conheça os benefícios que uma arquitetura modular em OSGi pode oferecer para suas aplicações

Pode-se dizer que os conceitos fundamentais para o desenvolvimento de um *software* baseiam-se na busca pela flexibilidade, robustez e reutilização. Assim, características como coesão, acoplamento, modularização e extensibilidade costumam estar no topo do *checklist* de arquitetos que querem que o código de uma determinada solução de *software* seja definitivamente de boa qualidade.

Nesse contexto, enquanto as ferramentas que apoiam o desenvolvimento de *software*, bem como os meios nos quais o mesmo é executado, mudaram significativamente, os princípios básicos para o desenvolvimento de uma boa arquitetura continuam os mesmos. A título de exemplo, observamos todos os novos recursos disponíveis em IDEs e na Java Virtual Machine (JVM), que apoiam o desenvolvimento, e também a evolução proporcionada pelo ecossistema baseado em *Cloud Computing*, no qual muitos *softwares* são executados.

No entanto, os conceitos que moldam a qualidade na construção de um sistema não sofrem mudanças significativas há pelo menos vinte e cinco anos. Isso se mostra evidente ao lembrar que, mesmo antes da publicação do livro “*Design Patterns: Elements of Reusable Object-Oriented Software*”, em 1994 — cujos autores ficaram conhecidos como “Gangue dos Quatros” (*Gang Of Four*, frequentemente abreviado como “GoF”) —, os engenheiros Kent Beck e Ward Cunningham iniciaram, em 1987, um experimento para aplicar padrões baseados em ordem, organização e forma nas linguagens de programação. Tal experimento foi baseado na obra do arquiteto e matemático Christopher Alexander, denominada “*Notes on*

## Fique por dentro

O conceito de modularização dinâmica para a tecnologia Java, proposto pela iniciativa OSGi, vem sendo adotado há um bom tempo por soluções de software cuja execução não deve ser interrompida, seja devido à necessidade de instalação de novos módulos, seja para alcançar resiliência quanto à disponibilidade da aplicação. Independentemente disso, existem ótimos benefícios pela escolha de uma arquitetura modular, e o framework OSGi é o caminho para alcançar tais vantagens. Com base nisso, este artigo tem como objetivo apresentar o contexto no qual a especificação OSGi foi concebida, bem como um exemplo prático para o entendimento dos principais conceitos.

*the Synthesis of Form*”, publicada em 1964. Curiosamente, Alexander também influenciou na criação do famoso jogo “SimCity”, a partir de sua obra “*A Pattern Language*”, conforme declarado pelo próprio Will Wright, designer do jogo e co-fundador da Electronic Arts. Voltando ao experimento de Beck e Cunningham, eles apresentaram seus resultados no mesmo ano de 1987 em uma conferência anual dedicada à pesquisa e desenvolvimento de linguagens orientadas a objetos, conhecidas como OOPSLA, que naquele ano ocorreu em Orlando (Flórida, EUA). Tais resultados contribuíram para o início de um movimento para a busca por padrões de projetos na área da ciência computacional, que finalmente ganhou popularidade a partir da publicação do livro pelos membros do GoF, em 1994.

Depois disso, poucas publicações alcançaram tamanha popularidade como a obra do GoF. É claro que surgiram novos Design Patterns nos últimos anos, mas os princípios nos quais esses se apoiam continuam os mesmos.

Entre os princípios mais conhecidos, é possível citar cinco que atualmente delineiam os parâmetros qualitativos na engenharia de um sistema. São eles: Princípio da Responsabilidade Única (*Single Responsibility Principle* — SRP); Princípio da Segregação de Interfaces (*Interface Segregation Principle* — ISP); Princípio da Substituição de Liskov (*Liskov Substitution Principle* — LSP); Princípio da Inversão de Dependências (*Dependency Inversion Principle* — DIP) e o Princípio do Aberto Fechado (*Open Closed Principle* — OCP).

Não está no escopo deste artigo se aprofundar sobre cada um desses conceitos. Contudo, vale dizer que três deles, o SRP, DIP e o ISP, estão diretamente ligados à determinação por um código modular. Não por acaso a modularização oferece excelentes benefícios quando se pretende alcançar uma arquitetura com índices de manutenibilidade e reusabilidade satisfatórios.

Entre os benefícios obtidos pelo desenho de um código modular, observamos a facilidade com que componentes independentes podem ser extraídos a partir dele. Tal vantagem mostrou-se muito importante no ecossistema do desenvolvimento de *software*. Isso porque a produção de códigos modulares, empacotados em componentes — na forma de bibliotecas do tipo JAR, por exemplo —, promoveu o reuso genuíno além das fronteiras de um ou outro grupo de desenvolvedores. Desse modo, centenas de *frameworks* foram construídos, compartilhados e combinados, como num lego, com o objetivo de alavancar o desenvolvimento de sistemas complexos com um esforço relativamente reduzido frente ao que seria necessário para construir-los inteiramente do zero. Assim, grande parte da comunidade de desenvolvedores beneficiou-se (e vem se beneficiando) da contribuição mútua de código reutilizável, muitas vezes chamados de *frameworks*, como o JUnit, SLF4J, Log4j, Google Guava, Apache Commons, Spring, Hibernate e milhares de outros.

Entretanto, não foi por acaso que tal modelo para construção de código modular encaixou-se perfeitamente na tecnologia Java. A forte aderência dessa linguagem de programação aos princípios da Orientação a Objetos tornou a aplicação desse modelo muito efetiva, e, como já foi dito nos parágrafos anteriores, bibliotecas empacotadas em arquivos JAR se proliferaram, permitindo a forte reutilização de seus módulos.

O uso de um *software* denominado Maven também alavancou a prática de reuso de componentes, que são disponibilizados em centenas de repositórios públicos. Tal comportamento também tem sido observado em outras tecnologias, como o JavaScript, que foi marcado pelo surgimento da plataforma Node.js. Essa plataforma possui um gerenciador de pacotes denominado *npm*, o qual disponibiliza também repositórios públicos contendo componentes que podem ser reutilizados. Tal gerenciador é mantido pela empresa NPM Inc.

Pode-se dizer que um grande percentual das aplicações construídas em JavaScript atualmente são dependentes desse modelo. Para se ter uma ideia, em março de 2016, o programador Azer Koçulu removeu cerca de 250 módulos que havia compartilhado no *npm*, isso porque se enfezou quando a equipe da NPM Inc., juntamente com advogados de uma empresa responsável por um aplicativo

de mensagens instantâneas, denominado Kik, solicitou que ele alterasse o nome de um de seus componentes, também batizado de Kik, alegando direitos autorais sobre esse nome. O problema é que, entre os módulos removidos, estava o popular “*left-pad*”, composto somente por onze linhas de código. A remoção desse módulo comprometeu milhares de aplicações na *web* que o possuíam como dependência, entre elas o Facebook, Netflix, Airbnb e milhares de outras. Assim, tais aplicações ficaram indisponíveis por um curto período de tempo até que a NPM Inc. conseguiu reestabelecer tal dependência. Tal episódio levantou inclusive uma discussão sobre esse tipo de fragilidade dos sistemas *web*.

No caso da tecnologia Java, o modelo de reutilização baseado em componentes empacotados em JARs possui um detalhe que pode vir a se tornar uma desvantagem dependendo do tipo de sistema a ser construído. Esse detalhe é a necessidade de reinicialização da Máquina Virtual Java (JVM) para os casos de inclusão, remoção ou atualização dos componentes que compreendem a aplicação. Tal fato pode ser um problema para aplicações construídas para servirem outras aplicações, como o WebLogic, JBoss AS, GlassFish, TomEE, etc., e para plataformas para o desenvolvimento de sistemas, onde é preciso gerenciar a instalação de plug-ins e extensões, como o Eclipse IDE. Para esses tipos de sistemas, é necessário que sejam implantados módulos em tempo de execução e que os mesmos possam ser removidos e atualizados sem a necessidade de reinício da JVM, ou seja, um modelo de modularização dinâmico.

Diante dessa latente necessidade, surgiu a especificação OSGi (*Open Services Gateway initiative*), também conhecida por *Dynamic Module System for Java*, desenhada por uma organização denominada OSGi Alliance. Essa organização foi fundada em março de 1999, por um consórcio formado pelas empresas Ericsson, IBM, Motorola, Sun Microsystems, entre outras.



Atualmente, a OSGi Alliance conta com a participação de mais de 35 empresas de diferentes áreas. Entre elas, pode-se citar a Adobe, IBM, Liferay, Oracle e Salesforce. Tal aliança possui um *board* de diretores que define a governança geral, e diversos escritórios com diferentes papéis e responsabilidades. O trabalho técnico é conduzido por grupos de especialistas, chamados de EGs (*Expert Groups*), responsáveis por definir a especificação, implementações de referência e testes de conformidade. Tais grupos de especialistas já chegaram a produzir seis *Major Releases* da especificação e são divididos por áreas, como *enterprise*, *mobile*, veículos, residencial, *telematics*, entre outras. O grupo responsável por tecnologias voltadas à área residencial, por exemplo, trabalha em especificações para o gerenciamento remoto de residências.

Em outubro de 2003, a Nokia, Motorola, IBM, ProSyst e outros membros da OSGi formaram um grupo de especialistas para especificar uma plataforma de serviços usada no mercado de *smartphones*. Além disso, empresas como a NASA e SAP, e até o sistema de gerenciamento das ferrovias na Suíça vêm utilizando intensamente OSGi.

Em resumo, não restam dúvidas que tal tecnologia vem sendo usada por grandes empresas em diferentes áreas de negócio, como citado no próprio *website* oficial da iniciativa OSGi: "... tem sido distribuída em muitas das 500 maiores empresas globais listadas pela revista Fortune ...".

Entretanto, mesmo com a chancela de grandes empresas globais, ainda é difícil encontrar desenvolvedores especialistas nessa tecnologia, até porque não se trata de uma especificação definida oficialmente por algum JCP. Na verdade, houve alguns esforços da comunidade para definir a especificação oficial responsável por tratar módulos dinâmicos em Java, começando pela JSR-277, em 2005, também conhecida como *Java Module System*, que foi retirada

e hoje se encontra inativa. Em seguida, tivemos a JSR-291, proposta pela IBM em 2006 e entregue em 2007. Também denominada *Dynamic Component Support for Java SE*, tal JSR foi aprovada por 12 votos a dois (aliás, um dos votos contra foi da Oracle), tendo Thomas Watson, da IBM (projeto Equinox), como líder. O objetivo era trazer o OSGi para o *mainstream Java SE*, assim como foi feito no JSR-232 para a plataforma Java ME.

Ainda em 2006, tivemos a JSR-294 (*Improved Modularity Support in the Java Programming Language*), que na verdade não definia um sistema modular em Java, mas propunha alterações na JVM que permitissem alavancar sistemas modulares. A ideia era estender a linguagem Java incluindo novos construtores que possibilitassem uma organização modular hierárquica. Porém, ela também foi retirada, em janeiro de 2016, para ser substituída pela JSR-376.

Denominada *Java Platform Module System*, a JSR-376 é o componente central do famoso Projeto Jigsaw, que tem como meta incorporar o sistema modular dinâmico no JDK. Apesar de ter sido diversas vezes adiado, está previsto para ser incorporado na versão 9 da JDK.

Apesar dos esforços da comunidade em definir uma JSR oficial, o OSGi continua navegando imponente no oceano da modularização dinâmica para a tecnologia Java, mesmo depois de sofrer algumas "alfinetadas" do próprio James Gosling em sua entrevista para Darryl K. Taft, do site eWEEK, em junho de 2009. Na entrevista ele diz que o: "... OSGi vem de um universo diferente, é um tanto pesado, e não se adapta muito bem em espaços menores ...", o que gerou muitas críticas pelos defensores da especificação OSGi na época. Mesmo Gosling deve levar em consideração que tal especificação está muito bem incorporada pelo mercado, já que quase todos os servidores de aplicações *enterprise* suportam ou planejam suportar OSGi. Além disso, o Spring Framework também suporta OSGi através do projeto *Spring Dynamic Modules for OSGi Service Platforms*, que oferece uma camada de infraestrutura para facilitar o desenvolvimento de aplicações baseadas em Spring com OSGi. Enfim, trata-se de uma tecnologia que vem obtendo cada vez mais espaço no universo do desenvolvimento de *software*.

## Construindo um módulo OSGi

Como dito anteriormente, o OSGi vem se tornando cada vez mais comum no contexto de desenvolvimento de *software*, já que muitas soluções têm sido baseadas nesse modelo. Por esse motivo, parece um bom negócio compreender os princípios básicos dessa tecnologia e, talvez, até caminhar para um desenvolvimento focado em modularização, fato que por si só já contribuirá para uma melhoria da qualidade do produto final. Sendo assim, neste artigo demonstraremos o funcionamento básico de um módulo OSGi, bem como o seu ciclo de vida. Antes disso, no entanto, faz-se necessário um entendimento básico da arquitetura modular proposta pela especificação. Tal arquitetura é implementada pelos chamados *containers*. Os *containers* permitem o gerenciamento do ciclo de vida dos módulos e a interdependência entre eles. Desse modo, para criar uma aplicação OSGi é necessário, além de usar a API da especificação, realizar o *deploy* em um *container*.

Na perspectiva de um desenvolvedor, um *container OSGi* oferece as seguintes vantagens:

- É possível instalar, desinstalar, iniciar e parar seus módulos dinamicamente, sem a necessidade de reiniciar o *container* ou a JVM;
- Sua aplicação poderá conter duas versões de um módulo em particular, rodando ao mesmo tempo e oferecendo serviços aos outros módulos em ambas as versões;
- OSGi oferece uma infraestrutura para o desenvolvimento de aplicações orientadas a serviços, ou seja, o que será usado para promover a interoperabilidade entre os módulos estará na forma de serviços, preferencialmente.

### Os containers para OSGi

Considerando que um *servlet container*, como o Tomcat, é usado para construir aplicações *web*, e um *container EJB* é usado para a construção de aplicações transacionais, um *container OSGi* é usado exclusivamente para a construção de aplicações em Java compostas por módulos. Ademais, os *containers OSGi* geralmente têm a característica de serem extremamente leves e poderem ser incorporados em qualquer solução *enterprise*. Dito isso, a primeira ação a ser tomada, antes da criação de um módulo, é a de escolher um *container* apropriado. Entre os mais conhecidos, podemos citar:

- **Equinox:** trata-se da implementação de referência oficial do OSGi. É o coração do Eclipse IDE, e, além de implementar todos os requisitos mandatórios da especificação, implementa também a maioria dos opcionais;
- **Knopflerfish:** implementação *open source*; assim como o Equinox, também implementa todos os requisitos mandatórios e alguns dos opcionais;
- **Apache Felix:** implementação oferecida pela Apache Foundation, também *open source*.

Para esse projeto, optamos pelo Apache Felix, por apresentar um console extremamente leve e simplificado. Nesse momento é importante ressaltar que, no universo do OSGi, o *software* é distribuído na forma de

```
-rw-r--r--@ 1 andrefabbro staff 987B Oct 13 2015 DEPENDENCIES  
-rw-r--r--@ 1 andrefabbro staff 11K Oct 13 2015 LICENSE  
-rw-r--r--@ 1 andrefabbro staff 1.1K Oct 13 2015 LICENSE.kxml2  
-rw-r--r--@ 1 andrefabbro staff 550B Oct 13 2015 NOTICE  
drwxr-xr-x@ 3 andrefabbro staff 102B Sep 5 19:03 bin/  
drwxr-xr-x@ 6 andrefabbro staff 204B Sep 5 19:03 bundle/  
drwxr-xr-x@ 3 andrefabbro staff 102B Sep 5 19:03 conf/  
drwxr-xr-x@ 20 andrefabbro staff 680B Oct 13 2015 doc/  
drwxr-xr-x 9 andrefabbro staff 306B Sep 5 19:58 felix-cache/  
Andres-MBP:felix-framework-5.4.0 andrefabbro$ █
```

Figura 1. Pastas e arquivos da distribuição do Apache Felix

```
Andres-MBP:felix-framework-5.4.0 andrefabbro$ java -jar bin/felix.jar  
Hello world  
-----  
Welcome to Apache Felix Gogo  
g! █
```

Figura 2. Output da execução do binário felix.jar

```
Welcome to Apache Felix Gogo  
g! lb  
START LEVEL 1  
ID|State |Level|Name  
0|Active | 0|System Bundle (5.4.0)|5.4.0  
1|Active | 1|Apache Felix Bundle Repository (2.0.6)|2.0.6  
2|Active | 1|Apache Felix Gogo Command (0.16.0)|0.16.0  
3|Active | 1|Apache Felix Gogo Runtime (0.16.2)|0.16.2  
4|Active | 1|Apache Felix Gogo Shell (0.10.0)|0.10.0  
g! █
```

Figura 3. Output da execução do comando lb

um *bundle*. Um *bundle* consiste em uma série de classes Java e outros artefatos que oferecem serviços e pacotes para outros *bundles*, ou seja, é o equivalente ao módulo. O Eclipse oferece um excelente suporte para o desenvolvimento de *bundles* OSGi: não apenas através de *wizards* para a criação desse tipo de projeto, mas também incorporando um *container* Equinox embutido para executar e depurar plugins. Vale dizer que todo plugin do Eclipse é um *bundle* OSGi, com algum código específico para o Eclipse.

Portanto, a primeira tarefa a ser feita é acessar o site do Apache Felix e realizar o *download* da distribuição 5.4.0. Após o *download*, extraia os arquivos para qualquer pasta em seu ambiente. Você verá uma estrutura de pastas similar à mostrada na Figura 1.

Para iniciar o *container*, basta executar o arquivo *felix.jar* (pasta *bin*) na JVM usando o seguinte comando: *java -jar bin/felix.jar*. Com isso você verá o *output* mostrado na Figura 2.

O que é apresentado na imagem é o console administrativo do *container*, denominado Gogo Shell. Nele você poderá instalar, desinstalar, atualizar ou remover *bundles*. Para listar todos os *bundles* instalados, por exemplo, basta executar o comando *lb*, e o *output* será conforme demonstra a Figura 3. Veja que a distribuição do Apache Felix já traz cinco módulos por padrão: o primeiro se trata do módulo do sistema, ou seja, da implementação OSGi propriamente dita; o segundo gerencia o repositório dos *bundles*; e os outros referem-se ao próprio console de administração.

## Ciclo de vida de um bundle OSGi

Uma vez iniciado o *container*, podemos criar nosso primeiro *bundle* OSGi e realizar seu *deploy*. Antes disso, no entanto, é necessário entender o ciclo de vida de um *bundle*. Tal ciclo de vida é demonstrado na **Figura 4**. Note que os retângulos representam os estados em que um *bundle* pode se encontrar, e as setas que ligam os estados representam o fluxo de um estado para outro, onde a descrição da seta mostra o comando que deve ser executado no console para alteração de estado. Dito isso, a ideia para esse primeiro *bundle* que iremos construir é simplesmente imprimir o famoso “Hello World” na inicialização do *bundle* (ou seja, no estado “STARTING”), e imprimir “Goodbye World” na parada do mesmo (estado “STOPPING”).

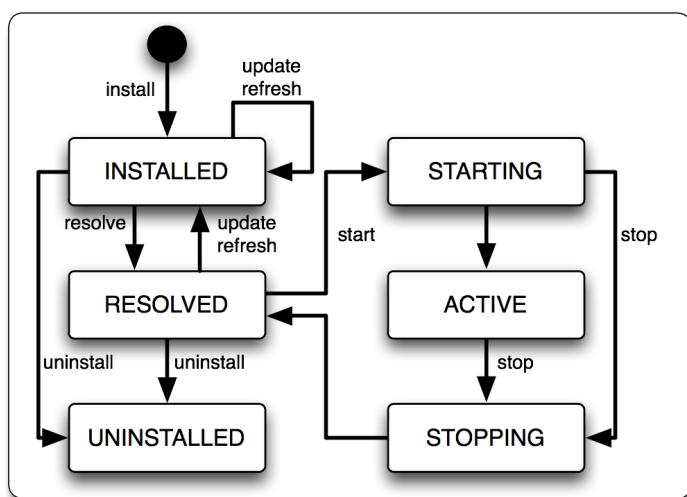


Figura 4. Ciclo de vida de um bundle OSGi

## Estrutura do projeto

Para iniciar a construção do projeto, crie uma pasta qualquer em seu ambiente, e então, a partir de sua raiz, crie uma árvore de pastas com a seguinte estrutura: *devmedia osgi exemplo*/src/main/java/br/com/devmedia/osgi/exemplo/. Para isso, você pode usar o seguinte comando: `mkdir devmedia osgi exemplo\src\main\java\br\com\devmedia\osgi\exemplo`. Desse modo, a pasta raiz do seu projeto se chamará *devmedia osgi exemplo*. Note, ainda, que já foi criada uma estrutura padrão de pastas para servir como o pacote principal, o qual será denominado **br.com.devmedia.osgi.exemplo**. Sendo assim, a partir da pasta raiz do projeto (*devmedia osgi exemplo*), crie uma classe Java, denominada **Activator**, no caminho *src/main/java/br/com/devmedia/osgi/exemplo/* (você pode usar o seguinte comando: `touch src/main/java/br/com/devmedia/osgi/exemplo/Activator.java`). Agora, abra essa classe para edição e copie o conteúdo indicado na **Listagem 1**.

### Listagem 1. Código da classe Activator.

```
01 package br.com.devmedia.osgi.exemplo;
02
03 import org.osgi.framework.BundleActivator;
04 import org.osgi.framework.BundleContext;
05
06 public class Activator implements BundleActivator {
07
08     @Override
09     public void start(BundleContext context) throws Exception {
10         System.out.println("Hello world");
11     }
12
13     @Override
14     public void stop(BundleContext context) throws Exception {
15         System.out.println("Goodbye World");
16     }
17 }
```

Ao analisar o código dessa classe, notamos que ela implementa a interface **BundleActivator**, contida na API OSGi, como visto na linha 06. Isso fará com que essa classe seja automaticamente instanciada pelo *container* OSGi através de uma chamada do tipo `Class.newInstance()`. Feito isso, o *container* executará o método `start()` (linha 09) quando o *bundle* transitar do estado “RESOLVED” para o estado “STARTING”. Portanto, o método `start()` é a oportunidade que você tem para implementar o código que realizará tarefas de inicialização do módulo, como abrir uma conexão com um banco de dados. Tal método também recebe como argumento um objeto do tipo **BundleContext**, que servirá para prover o acesso a informações específicas do *container* OSGi. Se alguma exceção for lançada durante a execução desse método, o *container* marcará o *bundle* como “stopped” e não irá exportar seus serviços. No caso desse código, estamos apenas imprimindo a mensagem “Hello World” no *output*.

Ainda referente ao código da **Listagem 1**, nota-se o método `stop()`, na linha 14. Esse método será executado pelo *container* no momento da parada do módulo. Dessa forma, ele fornece a oportunidade de implementar o código referente a tarefas de



limpeza, como liberar uma conexão com o banco de dados. Veja que o código implementado nessa classe apenas imprimirá a mensagem “Goodbye World” no *output*.

### Empacotando o bundle

Uma vez que a classe **Activator** esteja pronta, podemos passar para a tarefa de compilação e empacotamento do *bundle*. Aqui, há um detalhe que é muito importante dentro do processo de *deploy* de uma aplicação OSGi: o conteúdo do arquivo *MANIFEST.MF*. Muitos desenvolvedores não estão habituados a manipular o conteúdo desse arquivo, já que muitas das informações são irrelevantes na maioria das arquiteturas Java Enterprise quando tratamos apenas de bibliotecas empacotadas em arquivos JAR. No entanto, no universo da tecnologia OSGi, o conteúdo desse arquivo é muito importante, já que os *containers* se baseiam nas informações dele para, por exemplo, descobrir os pacotes a serem exportados e/ou importados pelo módulo, suas versões, dependências, entre outras informações. Dito isso, um exemplo para o conteúdo do arquivo *MANIFEST.MF* da aplicação que acabamos de criar seria a **Listagem 2**.

**Listagem 2.** Exemplo do arquivo *MANIFEST.MF* para o módulo Hello World.

```
01 Manifest-Version: 1.0
02 Bundle-ManifestVersion: 2
03 Bundle-Name: Devmedia OSGi Exemplo
04 Bundle-SymbolicName: devmedia osgi exemplo
05 Bundle-Version: 1.0.0
06 Bundle-Activator: br.com.devmedia.osgi.exemplo.Activator
07 Bundle-Vendor: DEVMEDIA
08 Import-Package: org.osgi.framework;version="[1.5,2]"
```

Os parâmetros mais relevantes indicados no arquivo *MANIFEST.MF*, demonstrado na **Listagem 2**, são descritos a seguir:

- **Bundle-ManifestVersion:** essa propriedade diz para o *container* qual a versão da especificação que esse *bundle* se apoia. O valor “2” diz que esse *bundle* é compatível com a especificação OSGi Release 4. Para a Release 3 ou anterior, deve-se usar o valor “1”;
- **Bundle-Name:** essa propriedade define o nome curto do *bundle*, humanamente amigável;
- **Bundle-SymbolicName:** Tal propriedade especifica um nome único para o *bundle*. Esse será o nome pelo qual os outros *bundles* o identificarão;
- **Bundle-Version:** especifica a versão do *bundle*;
- **Bundle-Activator:** essa é a propriedade (opcional) que indicará para o *container* qual é a classe *listener* que será notificada quando houver eventos de inicialização e parada. Em nosso caso, o valor é o nome da classe **Activator**, criada na **Listagem 1**;
- **Bundle-Vendor:** é o nome amigável do fornecedor do *bundle*. Nesse caso estamos usando DEVMEDIA;

```
Andres-MBP:devmedia osgi exemplo andrefabbro$ gradle init
:wrapper
:init

BUILD SUCCESSFUL

Total time: 0.593 secs
Andres-MBP:devmedia osgi exemplo andrefabbro$
```

**Figura 5.** Output do comando *gradle init*

drwxr-xr-x	3	andrefabbro	staff	102B	Sep 6 01:39	.gradle/
-rw-r--r--	1	andrefabbro	staff	1.2K	Sep 6 01:39	build.gradle
drwxr-xr-x	3	andrefabbro	staff	102B	Sep 6 01:39	gradle/
-rwxr-xr-x	1	andrefabbro	staff	5.1K	Sep 6 01:39	gradlew
-rw-r--r--	1	andrefabbro	staff	2.2K	Sep 6 01:39	gradlew.bat
-rw-r--r--	1	andrefabbro	staff	657B	Sep 6 01:39	settings.gradle
drwxr-xr-x	4	andrefabbro	staff	136B	Sep 5 18:52	src/

**Figura 6.** Pastas e arquivos criados pelo comando *gradle init*

- **Import-Package:** define os pacotes a serem importados por esse *bundle*. Nesse caso será somente o *framework* OSGi. Veremos essa configuração em detalhes mais adiante, quando demonstrarmos o gerenciamento de dependências entre bundles.

Entretanto, mesmo diante do esclarecimento das informações contidas no arquivo *MANIFEST.MF*, não seria produtivo que o desenvolvedor criasse tal arquivo manualmente. Sendo assim, é recomendável o uso de alguma ferramenta de compilação e empacotamento, ou algum recurso da IDE, para auxiliá-lo nessa tarefa. Pode-se usar, por exemplo, o Maven ou o Gradle, que dispõem de plugins específicos para automatizar a criação do *MANIFEST.MF* específico para *bundles* OSGi. A ferramenta que escolhemos para o projeto exemplo neste artigo é o Gradle.

### Configurando o Gradle

Caso ainda não possua o Gradle instalado em seu ambiente, visite o site do projeto para realizar o *download* e adicione o binário como uma variável de ambiente em seu sistema operacional. Feito isso, navegue para a pasta raiz do projeto criado (denominado *devmedia osgi exemplo*) e execute o comando *gradle init*. O *output* será conforme o indicado na **Figura 5**.

Note que serão criados vários arquivos e diretórios na raiz do seu projeto, conforme demonstra a **Figura 6**.

Logo após, abra o arquivo *build.gradle* para edição e substitua todo o seu conteúdo pelo conteúdo da **Listagem 3**. Esse arquivo é responsável por definir as instruções de compilação e empacotamento de um projeto do tipo Gradle, sendo equivalente ao POM do Maven.

Ao analisar o conteúdo dessa listagem, nota-se que as configurações são organizadas em trechos separados por chaves. As duas primeiras linhas indicam a aplicação dos plugins que serão usados para ajudar na construção do projeto. O primeiro é referente ao compilador Java, e o segundo, denominado “osgi”, permite a inserção de instruções referentes à especificação OSGi no arquivo

# Modularização dinâmica em Java com OSGi

**Listagem 3.** Conteúdo do arquivo build.gradle.

```
01 apply plugin:'java'  
02 apply plugin:'osgi'  
03  
04 repositories {  
05   jcenter()  
06 }  
07  
08 dependencies {  
09   compile'org.osgi:org.osgi.core:6.0.0'  
10 }  
11  
12 jar {  
13   manifest {  
14     name'Devmedia OSGi Exemplo'  
15     version'1.0.0'  
16     vendor'DEV MEDIA'  
17     instruction'Bundle-Activator,'  
18     'br.com.devmedia.osgi.exemplo.Activator'  
19 }
```

*MANIFEST.MF*. No trecho entre as linhas 04 e 06 podem ser incluídos os repositórios (remotos ou locais) de onde o Gradle irá fazer *download* das dependências. Tais repositórios também podem ser do tipo Maven ou Ivy. Nesse caso, optamos pelo repositório público Java Central do Maven.

Na seção “*dependencies*”, entre as linhas 08 e 10, são definidas as dependências do projeto; aqui, incluímos a API OSGi. Finalmente, entre as linhas 13 e 18, são indicadas as informações a serem adicionadas no *MANIFEST.MF*. A maioria dessas informações, padrões, é obtida automaticamente pelo Gradle, como o fato de não haver a necessidade de incluir a propriedade *Bundle-SymbolicName*, já que

```
Andres-MBP:devmedia osgi-exemplo andrefabbro$ gradle build  
:compileJava UP-TO-DATE  
:processResources UP-TO-DATE  
:classes UP-TO-DATE  
:jar UP-TO-DATE  
:assemble UP-TO-DATE  
:compileTestJava UP-TO-DATE  
:processTestResources UP-TO-DATE  
:testClasses UP-TO-DATE  
:test UP-TO-DATE  
:check UP-TO-DATE  
:build UP-TO-DATE  
  
BUILD SUCCESSFUL  
  
Total time: 0.601 secs  
Andres-MBP:devmedia osgi-exemplo andrefabbro$
```

**Figura 7.** Output do comando gradle build

será usado o nome do projeto conforme definido no arquivo *settings.gradle* (gerado automaticamente quando o comando *gradle init* foi executado).

Após a edição do *build.gradle*, navegue até a pasta raiz do projeto e execute o comando: *gradle build*. Isso fará com que seu projeto seja compilado, empacotado em um arquivo JAR e, então, disponibilizado na pasta *build/libs/* com o nome *devmedia osgi-exemplo.jar*. O *output* do comando *build* deverá ser parecido ao mostrado na **Figura 7**.

## Deploy do bundle

Uma vez que o projeto foi construído, basta realizar a instalação do *bundle* no *container* através do console Gogo demonstrado anteriormente. Para isso, execute o comando *install \$Caminho-DO-PROJETO/build/libs/devmedia osgi-exemplo.jar*, onde *\$Caminho-DO-PROJETO* deve ser substituído pelo caminho da pasta raiz do seu projeto no sistema operacional, conforme demonstrado na **Figura 8**. Note que, após a instalação, é impresso no console o código identificador único do novo *bundle* instalado, por exemplo, *Bundle ID: 5*.

Para visualizar o novo *bundle* instalado no *container*, execute o comando *lb* como citado anteriormente; veja no exemplo da **Figura 9** os dados dele na última posição.

Note que o estado do novo *bundle* consta como “*Installed*”, enquanto que os outros se encontram no estado “*Active*”. Isso porque o módulo foi instalado, mas não está em execução.

## Iniciando o bundle

Para inicializar o módulo, deve-se usar o comando “*start*” seguido do ID do mesmo, por exemplo: *start 5*. Para realizar a parada, deve-se usar o comando “*stop*” seguido também do ID, por exemplo: *stop 5*. Tais comandos são demonstrados na **Figura 10**. Veja que no momento da inicialização do *bundle* é impressa a mensagem “Hello world” (indicada pela primeira seta vermelha) conforme foi implementado no método *start()* da classe **Activator**. Além disso, após a execução do comando para listar os módulos, veja que o estado foi alterado para “*Active*”.

```
Andres-MBP:felix-framework-5.4.0 andrefabbro$ java -jar bin/felix.jar  
Welcome to Apache Felix Gogo  
g! install /Users/andrefabbro/projetos/devmedia osgi-exemplo/build/libs/devmedia osgi-exemplo.jar  
Bundle ID: 5  
g!
```

**Figura 8.** Output do comando install no Gogo Shell

```
Andres-MBP:felix-framework-5.4.0 andrefabbro$ java -jar bin/felix.jar  
Welcome to Apache Felix Gogo  
g! install /Users/andrefabbro/projetos/devmedia osgi-exemplo/build/libs/devmedia osgi-exemplo.jar  
Bundle ID: 5  
g! lb  
START LEVEL 1  
ID|State |Level|Name  
0|Active | 0|System Bundle (5.4.0)|5.4.0  
1|Active | 1|Apache Felix Bundle Repository (2.0.6)|2.0.6  
2|Active | 1|Apache Felix Gogo Command (0.16.0)|0.16.0  
3|Active | 1|Apache Felix Gogo Runtime (0.16.2)|0.16.2  
4|Active | 1|Apache Felix Gogo Shell (0.10.0)|0.10.0  
5|Installed | 1|Devmedia OSGi Exemplo (1.0.0)|1.0.0  
g!
```

**Figura 9.** Output do comando lb no Gogo Shell após a instalação do novo bundle

Ainda na **Figura 10**, note que no momento da parada do *bundle* é impressa a mensagem “Goodbye World” (indicada pela segunda seta vermelha) conforme implementado no método `stop()`, e o estado do *bundle* foi modificado para “Resolved”. Além desses comandos básicos de instalação, inicialização e parada de um *bundle*, existem ainda os comandos “`update <ID>`” (para atualizar o JAR de um *bundle*), “`uninstall <ID>`” (para remover um *bundle*) e “`bundle <ID>`” (detalhes do *bundle*).

## Gerenciamento de dependências

Uma vez que a especificação OSGi permite que você construa a sua aplicação baseada em múltiplos módulos, ela fornece também uma forma de gerenciar a relação de dependência entre esses módulos. Esse gerenciamento é feito através do chamado *bundle scope*, que é o escopo das classes que serão especificamente compartilhadas entre os *bundles*. Isso porque, por padrão, nenhuma classe contida em um *bundle* é visível por outro, pois cada um possui seu próprio Classloader, e dentro dele são seguidas as mesmas regras de qualquer aplicação Java quanto à visibilidade de classes. Então, o mecanismo para gerenciar quais classes de um arquivo JAR serão visíveis e quais classes não serão visíveis é através da criação de múltiplos Classloaders pelo *container OSGi* para cada *bundle*.

Um *bundle* pode acessar classes a partir das seguintes origens:

- **Boot classpath:** Contém os pacotes base, ou seja, todos aqueles dos pacotes `java.*`;
- **Framework classpath:** Geralmente os *containers* separam um Classloader apenas para as classes de implementação do *framework OSGi*, bem como as interfaces de serviço mais importantes;
- **Bundle space:** Consiste nas classes contidas no arquivo JAR associado ao *bundle*, além de outros JARs que são incorporados ao módulo, na forma de fragmentos;
- **Imported packages:** Refere-se aos pacotes expostos por outros *bundles*, e que devem ser explicitamente importados por eles a partir de diretivas no `MANIFEST.MF`.

Na prática, para promover o compartilhamento das classes entre os módulos, é necessário exportar pacotes de um *bundle* origem e importá-los em um *bundle* destino. Esse processo é feito através das diretivas `Export-Package` e `Import-Package`, respectivamente, no arquivo `MANIFEST.MF`. As classes a serem compartilhadas entre os módulos devem ser interfaces de serviços e/ou classes POJO. Quanto às implementações dessas interfaces de serviço, elas não devem estar contidas nos pacotes identificados para exportação. Isso porque tais implementações devem ser desacopladas, podendo ser livremente modificadas sem afetar outros módulos clientes de tais serviços.

```

g! start 5
Hello world ←
g! lb
START LEVEL 1
ID|State |Level|Name
0|Active | 0|System Bundle (5.4.0)|5.4.0
1|Active | 1|Apache Felix Bundle Repository (2.0.6)|2.0.6
2|Active | 1|Apache Felix Gogo Command (0.16.0)|0.16.0
3|Active | 1|Apache Felix Gogo Runtime (0.16.2)|0.16.2
4|Active | 1|Apache Felix Gogo Shell (0.10.0)|0.10.0
5|Active | 1|Devmedia OSGi Exemplo (1.0.0)|1.0.0
g! stop 5
Goodbye World ←
g! lb
START LEVEL 1
ID|State |Level|Name
0|Active | 0|System Bundle (5.4.0)|5.4.0
1|Active | 1|Apache Felix Bundle Repository (2.0.6)|2.0.6
2|Active | 1|Apache Felix Gogo Command (0.16.0)|0.16.0
3|Active | 1|Apache Felix Gogo Runtime (0.16.2)|0.16.2
4|Active | 1|Apache Felix Gogo Shell (0.10.0)|0.10.0
5|Resolved | 1|Devmedia OSGi Exemplo (1.0.0)|1.0.0
g! []

```

**Figura 10.** Output dos comandos `start` e `stop` no Gogo Shell para o novo *bundle* instalado

## Serviços OSGi

Como mencionado anteriormente, a arquitetura OSGi é uma boa opção para implementar soluções orientadas a serviços, já que permite que os *bundles* exportem os seus serviços para serem consumidos por outros *bundles*, sem que estes tenham qualquer conhecimento sobre os detalhes de implementação daqueles. Isso é possível devido à habilidade de expor apenas interfaces, escondendo os detalhes da implementação. Essa é uma característica fundamental para aplicações que sejam orientadas a serviços.



# Modularização dinâmica em Java com OSGi

## Criando um serviço OSGi

Para demonstrar a relação de dependências entre módulos, bem como o compartilhamento de serviços entre eles, vamos criar um *bundle* denominado “Hello Service”, no qual exportaremos um pacote e registraremos um serviço. Tal serviço será consumido pelo *bundle* construído no tópico anterior (denominado “Devmedia OSGi Exemplo”). Para isso, crie um novo projeto do tipo Gradle conforme os passos descritos anteriormente, e nomeie-o como “devmedia-hello-service”. Nesse projeto, crie uma interface com o nome **HelloService** em um pacote cujo *namespace* seja **br.com.devmedia.hello.service**. Essa interface deverá conter apenas um método, denominado **sayHello()**, como demonstra o conteúdo da **Listagem 4**.

**Listagem 4.** Interface HelloService do projeto “devmedia-hello-service”.

```
01 package br.com.devmedia.hello.service;
02
03 public interface HelloService {
04     String sayHello();
05 }
```

Em seguida, crie uma implementação para essa interface, denominada **HelloServiceImpl**, no pacote **br.com.devmedia.hello.service.impl** e inclua o conteúdo da **Listagem 5**. É possível notar que a implementação do método **sayHello()** imprime uma mensagem dizendo “Executando o método sayHello()” e logo em seguida retorna uma **String** com o conteúdo “Say Hello”.

**Listagem 5.** Código da classe HelloServiceImpl, que implementa a interface HelloService.

```
01 package br.com.devmedia.hello.service.impl;
02
03 import br.com.devmedia.hello.service.HelloService;
04
05 public class HelloServiceImpl implements HelloService {
06
07     @Override
08     public String sayHello() {
09         System.out.println("Executando o metodo sayHello()");
10         return "Say Hello";
11     }
12 }
```

Após a criação da interface e da sua implementação, conforme as **Listagens 4 e 5**, é necessário informar, no arquivo **MANIFEST.MF**, qual o pacote que contém a interface **HelloService** para que o *container* a deixe disponível para ser importada por outros módulos. Isso é feito adicionando a diretiva **Export-Package**. Além disso, deve-se também registrar a implementação dessa interface (a classe **HelloServiceImpl**) no contexto do *bundle*, pois assim será possível executar a implementação especificada quando o serviço for acionado.

## Registro do serviço

Sendo assim, iremos primeiro realizar o registro do serviço no contexto do *bundle*. Tal registro ocorrerá no momento da inicialização do módulo. Para isso, usaremos a mesma estratégia do primeiro *bundle* criado no tópico anterior, ou seja, implementar o método **start()** e **stop()** em uma nova classe que esteja na hierarquia de **BundleActivator**. Portanto, crie uma nova classe no projeto **devmedia-hello-service**, denominada **HelloServiceActivator**, no pacote **br.com.devmedia.hello.service.impl**, e copie o conteúdo da **Listagem 6**.

**Listagem 6.** Código da classe HelloServiceActivator, que implementa um novo **BundleActivator** para o módulo “Hello Service”.

```
01 package br.com.devmedia.hello.service.impl;
02
03 import org.osgi.framework.BundleActivator;
04 import org.osgi.framework.BundleContext;
05 import org.osgi.framework.ServiceRegistration;
06
07 import br.com.devmedia.hello.service.HelloService;
08
09 public class HelloServiceActivator implements BundleActivator {
10
11     private ServiceRegistration helloServiceRegistration;
12
13     @Override
14     public void start(BundleContext context) throws Exception {
15         HelloService helloService = new HelloServiceImpl();
16         helloServiceRegistration = context.registerService(
17             HelloService.class.getName(), helloService, null);
18     }
19
20     @Override
21     public void stop(BundleContext context) throws Exception {
22         helloServiceRegistration.unregister();
23     }
24 }
```

Ao analisar o código dessa listagem, é possível notar que ela implementa a interface **BundleActivator** do próprio *framework*, conforme a linha 09, para que os métodos **start()** e **stop()** sejam executados no início e na parada do *bundle*, como já dissemos anteriormente. Além disso, criamos um atributo do tipo **ServiceRegistration**, na linha 11, que será usado para armazenar a referência do registro desse serviço. No método **start()**, especificamente na linha 15, note que é criada uma nova instância de **HelloService**, escolhendo pela implementação do tipo **HelloServiceImpl**. Em seguida, na linha 16, essa instância é registrada e sua referência permanecerá ligada ao atributo **helloServiceRegistration**. Finalmente, no método **stop()**, é realizada a limpeza do registro do serviço, pois será quando o *bundle* receberá o sinal de parada.

## Exportando o pacote do serviço

Após a implementação do **BundleActivator** para registrar o serviço, inclua o conteúdo da **Listagem 7** no arquivo **build.gradle** desse projeto. O ponto mais importante a ser observado nessa listagem é a linha 23, onde se encontra a instrução para adicionar o parâmetro **Export-Package** com o valor “**br.com.devmedia.hello.service**”.

**Listagem 7.** Arquivo build.gradle para o projeto “Hello Service” com as informações para o MANIFEST.MF.

```
01 apply plugin:'java'
02 apply plugin:'osgi'
03 apply plugin:'maven'
04
05 group = 'br.com.devmedia'
06 version ='1.0.0'
07
08 repositories {
09   jcenter()
10 }
11
12 dependencies {
13   compile'org.osgi:org.osgi.core:6.0.0'
14 }
15
16 jar {
17   manifest {
18     name'Hello Service'
19     version'1.0.0'
20     vendor'DEVMEDIA'
21     instruction'Import-Package','org.osgi.
framework;version="[1.8,2)"
22     instruction'Bundle-Activator',
23     'br.com.devmedia.hello.service.impl
.HelloServiceActivator'
24     instructionReplace'Export-Package',
25     'br.com.devmedia.hello.service'
26 }
27 archiveName'devmedia-hello-service.jar'
28 }
```

Isso incluirá tal diretiva no arquivo *MANIFEST.MF* do *bundle*. Essa é justamente a instrução que fará com que todas as classes pertencentes ao pacote indicado sejam disponibilizadas para importação por outros módulos do *container*. Além disso, na linha 22, é incluída a instrução para adicionar o parâmetro “*Bundle-Activator*” contendo a classe **HelloServiceActivator**, responsável pelo registro do serviço no momento da inicialização, como demonstra a **Listagem 6**.

Desse modo, fica claro perceber que somente a exportação de classes de um módulo para outro não disponibiliza, necessariamente, o serviço para ser consumido no *bundle* destino, apenas indica que um módulo poderá usar as classes do outro (que estejam no pacote indicado). Para que um serviço seja compartilhado entre os módulos é necessário registrá-lo explicitamente no contexto do *bundle*, conforme observado na **Listagem 6**.

### Construção e instalação do *bundle*

Sendo assim, o projeto **devmedia-hello-service** já está pronto para construção e instalação no *container OSGi*. Entretanto,

```
Andres-MacBook-Pro:felix-framework-5.4.0 andrefabbro$ java -jar bin/felix.jar
-----
Welcome to Apache Felix Gogo

g! install /Users/andrefabbro/DEV/workspaces/devmedia-osgi/devmedia-hello-service/bui
ld/libs/devmedia-hello-service.jar
Bundle ID: 6
g! [
```

**Figura 11.** Output do comando *install* para o módulo devmedia-hello-service

```
Welcome to Apache Felix Gogo

g! install /Users/andrefabbro/DEV/workspaces/devmedia-osgi/devmedia-hello-service/build/libs/devmedia-hello-service.jar
Bundle ID: 6
g! bundle 6
Location      file:/Users/andrefabbro/DEV/workspaces/devmedia-osgi/devmedia-hello-service/build/libs/devmedia-hello-service.jar
State         2
BundleContext null
BundleId       6
SymbolicName  br.com.devmedia.hello-service
RegisteredServices null
ServicesInUse  null
Bundle          6! Installed  1 lib:br.com.devmedia.hello-service (1.0.0)
Revisions      [br.com.devmedia.hello-service [6](R 6.0)]
Version        1.0.0
LastModified   1474140001486
Headers        [Created-By=1.8.0.40 (Oracle Corporation), Manifest-Version=1.0, Bnd-LastModified=1474138899000, Priv
ote-Package=br.com.devmedia.hello.service.impl, Bundle-Name=Hello Service, Bundle-Vendor=DEVMEDIA, Import-Package=org.osgi
.framework;version="[1.8,2)", Export-Package=br.com.devmedia.hello.service, Bundle-ManifestVersion=2, Bundle-SymbolicName=
br.com.devmedia.hello-service, Bundle-Version=1.0.0, Bundle-Activator=br.com.devmedia.hello.service.impl.HelloServiceActiv
ator, Require-Capability=osgi.ee;filter="(&(osgi.ee=JavaSE)(version=1.8))", Tool=Bnd-3.2.0.201605172007]
g! [
```

**Figura 12.** Output do comando *bundle <<ID>>* para o módulo “Hello Service”

para construí-lo devemos executar o comando *gradle install*, em vez de *gradle build* (como feito no primeiro projeto). Isso fará com que o projeto seja compilado, empacotado em um arquivo JAR (na pasta *build/libs*) e instalado no seu repositório local. É importante que o módulo seja instalado no seu repositório local porque você deverá importá-lo como uma dependência para o primeiro projeto, e assim utilizar classes desse módulo onde for necessário. Veremos como incluí-lo como dependência para o outro projeto mais adiante, quando voltarmos ao projeto **devmedia-osgi-exemplo** para realizar mais algumas alterações.

Agora, após executar o comando *gradle install*, basta abrir o console Gogo e executar o comando para instalação no *container*, por exemplo: *install /caminho-completo-do-seu-projeto/build/libs/devmedia-hello-service.jar*, conforme o exemplo da **Figura 11**. Veja que o *output* será o ID do novo *bundle* instalado.

Para verificar se o *bundle* está mesmo exportando o pacote conforme incluímos nos parâmetros do arquivo de construção do Gradle, você deve executar o comando *bundle <<ID>>*, onde <<ID>> é o ID do *bundle* que se deseja obter informações.

Ao executar esse comando para o módulo **devmedia-hello-service**, o *output* será conforme o exemplo da **Figura 12**. Observe no trecho grifado em amarelo que a diretiva **Export-Package** está conforme definimos anteriormente no arquivo de construção do Gradle.

Feito isso, basta executar o comando *start <<ID>>* (substituindo o <<ID>> pelo identificador do *bundle*) para o **devmedia-hello-service**, e seu serviço **HelloService** já estará disponível para ser utilizado por outros *bundles*.

### Consumindo o serviço pelo outro módulo

Uma vez que o módulo **Hello Service** tenha sido construído, copiado para o repositório local e instalado no *container OSGi*, podemos iniciar as alterações no projeto **devmedia-osgi-exemplo** a fim de alcançar duas coisas: importar os pacotes exportados pelo outro módulo e consumir o serviço registrado por ele.

Para isso, abra o arquivo *build.gradle* do projeto **devmedia-osgi-exemplo**. Note que esse arquivo está idêntico ao conteúdo da **Listagem 3**. Então, sobrescreva-o com o conteúdo da **Listagem 8**. A primeira diferença entre as duas versões é que, na **Listagem 8**, estamos incluindo a

dependência para o projeto **devmedia-hello-service**, conforme mostra a linha 11 (por isso que foi necessário executar *gradle install* ao invés de *gradle build* para esse projeto). A segunda diferença é a inclusão do repositório Maven local, conforme mostra a linha 05, pois será de lá que o Gradle buscará o componente **devmedia-hello-service** para adicioná-lo ao *classpath* do projeto. Finalmente, nota-se também que foi adicionada a diretiva para importação do pacote **br.com.devmedia.hello.service**, na linha 20. Veja que também estamos importando pacotes do *framework OSGi* (**org.osgi.framework**), já que também queremos ter acesso às suas classes/interfaces, como a interface **BundleActivator**.

**Listagem 8.** Conteúdo do arquivo *build.gradle* do projeto *devmedia osgi exemplo*.

```
01 apply plugin:'java'
02 apply plugin:'osgi'
03
04 repositories {
05   mavenLocal()
06   jcenter()
07 }
08
09 dependencies{
10   compile'org.osgi:org.osgi.core:6.0.0'
11   compile'br.com.devmedia:devmedia-hello-service:1.0.0'
12 }
13
14 jar{
15   manifest {
16     name 'Devmedia OSGi Exemplo'
17     version '1.0.0'
18     vendor 'DEVMEDIA'
19     instruction 'Bundle-Activator','br.com.devmedia.osgi.exemplo.Activator'
20     instruction 'Import-Package','org.osgi.framework;version="[1.8,2)"'
21     br.com.devmedia.hello.service
21 }
22 }
```

Em seguida, devemos alterar a implementação dos métodos **start()** e **stop()** da classe **Activator** do projeto **devmedia osgi exemplo** (atualmente igual à **Listagem 1**). Tal alteração consiste apenas em incluir a chamada ao serviço do módulo *Hello Service* para evidenciar a execução do serviço de um módulo em outro. Portanto, copie o conteúdo da **Listagem 9** para a classe **Activator**.

Como pode ser observado, as únicas diferenças são a inclusão do atributo **helloServiceReference**, do tipo **ServiceReference**, e a implementação dos métodos **start()** (linhas 14 a 20) e **stop()** (linhas 23 a 26). Veja que, no método **start()**, estamos ligando a referência do serviço ao atributo **helloServiceReference**, conforme mostra a linha 15. Em seguida, declaramos uma variável **helloService** do tipo **HelloService**, e na linha 19 imprimimos o retorno do método **sayHello()** do serviço, para comprovar que o serviço do outro módulo está sendo realmente consumido. Quanto ao método **stop()**, apenas incluímos a linha 25, responsável por liberar a referência do serviço no momento de parada do *bundle*.

## Reconstruindo e atualizando o primeiro módulo

Após tais alterações no projeto, podemos realizar a construção do mesmo através do comando *gradle build*.

Feito isso, acesse o console Gogo e execute os comandos *stop*, *update* e *start* do *bundle*, conforme mostra o *output* da **Figura 13**.

Como podemos perceber, após o *start* do *bundle* é impresso no console exatamente o que foi instruído pelo método **sayHello()** do **HelloService**, pertencente ao outro módulo, “Hello Service”. Isso demonstra que o serviço de um *bundle* foi definitivamente consumido pelo outro.

**Listagem 9.** Nova classe Activator para o projeto *devmedia osgi exemplo*.

```
01 package br.com.devmedia.osgi.exemplo;
02
03 import org.osgi.framework.BundleActivator;
04 import org.osgi.framework.BundleContext;
05 import org.osgi.framework.ServiceReference;
06
07 import br.com.devmedia.hello.service.HelloService;
08
09 public class Activator implements BundleActivator {
10
11   private ServiceReference helloServiceReference;
12
13   @Override
14   public void start(BundleContext context) throws Exception {
15     helloServiceReference =
16       context.getServiceReference(HelloService.class.getName());
17     HelloService helloService =
18       (HelloService) context.getService(helloServiceReference);
19     System.out.println(helloService.sayHello());
20   }
21
22   @Override
23   public void stop(BundleContext context) throws Exception {
24     System.out.println("Goodbye World");
25     context.ungetService(helloServiceReference);
26   }
27 }
```

```
g! lb
START LEVEL 1
ID|State    |Level|Name
0|Active   | 0|System Bundle (5.4.0)|5.4.0
1|Active   | 1|Apache Felix Bundle Repository (2.0.6)|2.0.6
2|Active   | 1|Apache Felix Gogo Command (0.16.0)|0.16.0
3|Active   | 1|Apache Felix Gogo Runtime (0.16.2)|0.16.2
4|Active   | 1|Apache Felix Gogo Shell (0.10.0)|0.10.0
5|Active   | 1|Devmedia OSGi Exemplo (1.0.0)|1.0.0
6|Active   | 1|Hello Service (1.0.0)|1.0.0
g! stop 5
Goodbye World
g! update 5
g! start 5
Executando o metodo sayHello() ←
Say Hello
g! |
```

**Figura 13.** Output para parada, atualização e inicialização do módulo DevMedia OSGi Exemplo

## Criando uma fábrica de serviços

Como foi visto no tópico anterior, usamos o *framework OSGi* para criar um objeto Java e registrá-lo como um serviço para ser consumido por quaisquer outros módulos. Entretanto, ao observarmos o método **HelloServiceActivator.start()** (**Listagem 6**), é possível notar que será criada uma única instância do tipo **HelloServiceImpl**, a qual será registrada no contexto do *bundle*. Consequentemente, sempre que outro *bundle* requisitar pelo serviço **HelloService**, o *container* entregará essa mesma instância.

Tal abordagem funciona perfeitamente para a maioria dos casos. Entretanto, suponhamos que se deseja retornar uma instância diferente de **HelloServiceImpl** para cada chamada ao serviço por outro *bundle*, ou então que o objeto de serviço necessite abrir uma conexão com o banco de dados e que tal conexão deva ser criada por demanda a fim de evitar conexões abertas sem necessidade.

Para esse tipo de situação, a solução é criar uma classe que implementa uma interface denominada **ServiceFactory**, do próprio *framework OSGi*, e registrar uma instância dessa no contexto do *bundle* em vez do objeto de serviço. Desse modo, tal instância controlará a criação dos objetos de serviço ao passo que forem solicitados por outros *bundles*. Ou seja, ela criará um novo objeto de serviço para cada *bundle*, e somente a partir da primeira requisição, nunca antes.

Para alcançar esse objetivo, vamos criar uma classe denominada **HelloServiceFactory** (no pacote `br.com.devmedia.hello.service`) com o conteúdo da **Listagem 10**.

**Listagem 10.** Código da classe HelloServiceFactory, responsável por fabricar objetos do tipo HelloServiceImpl.

```
01 package br.com.devmedia.hello.service;
02
03 import org.osgi.framework.Bundle;
04 import org.osgi.framework.ServiceFactory;
05 import org.osgi.framework.ServiceRegistration;
06
07 import br.com.devmedia.hello.service.impl.HelloServiceImpl;
08
09 public class HelloServiceFactory implements ServiceFactory {
10
11     private int contadorDeUso = 0;
12
13     @Override
14     public Object getService(Bundle bundle, ServiceRegistration registration) {
15         System.out.println(
16             "Cirando objeto do tipo HelloService para " +
17             bundle.getSymbolicName());
18         contadorDeUso++;
19         System.out.println(
20             "Quantidade de bundles usando o serviço: " + contadorDeUso);
21
22         HelloService helloService = new HelloServiceImpl();
23         return helloService;
24     }
25
26     @Override
27     public void ungetService(
28         Bundle bundle, ServiceRegistration registration, Object service) {
29         System.out.println(
30             "Removendo objeto do tipo HelloService para " +
31             bundle.getSymbolicName());
32         contadorDeUso--;
33         System.out.println(
34             "Quantidade de bundles usando o serviço: " + contadorDeUso);
35     }
36 }
```

Ao analisar esse código, o primeiro ponto a ser observado é que essa classe implementa a interface **ServiceFactory** do *framework OSGi*. Tal interface define dois métodos a serem implementados: **getService()** e **ungetService()**. O primeiro é invocado na primeira vez que um *bundle* qualquer solicita um objeto de serviço através

da instrução **BundleContext.getService(ServiceReference)**, como a linha 18 da **Listagem 9**. No caso da **Listagem 10**, estamos retornando uma instância diferente, do tipo **HelloServiceImpl**, para cada chamada proveniente de outro *bundle*, como é observado nas linhas 22 e 23. Além disso, estamos imprimindo uma mensagem para indicar que o método está sendo executado, na linha 15, e outra mostrando o valor do atributo **contadorDeUso**, que é um simples controle para sabermos quantos *bundles* estão usando o serviço. Note que há um incremento desse atributo a cada chamada do método, na linha 18. Vale ressaltar aqui que o *container OSGi* cria um cache na primeira chamada do serviço por um outro *bundle*, e por isso ele garante que retornará sempre a mesma instância de **HelloServiceFactory**, mesmo em futuras chamadas de **BundleContext.getService(ServiceReference)** provenientes do mesmo *bundle*. Quanto ao método **ungetService()**, o *container* irá invocá-lo quando o serviço for liberado por um outro *bundle*. Assim, o objeto de serviço deve ser destruído, evitando, por exemplo, vazamentos de memória, conexões desnecessárias com um banco de dados, etc. Na **Listagem 10**, utilizamos esse método para decrementar o valor do atributo **contadorDeUso** (linha 32) e imprimir a quantidade de clientes que está usando o serviço (linha 33).

Uma vez criada a classe **HelloServiceFactory**, devemos atualizar o **BundleActivator** do módulo *Hello Service*, trocando o registro do objeto de serviço pelo objeto fábrica. Para isso, abra a classe **HelloServiceActivator** (note que o conteúdo deve estar idêntico ao da **Listagem 6**) e substitua o código do método **start()** pelo conteúdo da **Listagem 11**.

**Listagem 11.** Novo método start() da classe HelloServiceActivator, trocando o objeto de serviço pelo objeto fábrica.

```
01     @Override
02     public void start(BundleContext context) throws Exception {
03         HelloServiceFactory helloServiceFactory = new HelloServiceFactory();
04         helloServiceRegistration = context.registerService(
05             HelloService.class.getName(), helloServiceFactory, null);
06     }
```

Note que o novo método **start()** cria uma instância do tipo **HelloServiceFactory** na linha 03 e registra tal instância no contexto do *bundle*, conforme as linhas 04 e 05. Na versão anterior, o objeto registrado no contexto era do tipo **HelloService**. A partir de agora, o objeto fábrica entregará instâncias diferentes do tipo **HelloServiceImpl** para cada módulo consumidor que executar **BundleContext.getService(ServiceReference)**. Efetuada essa alteração, compile do projeto usando o comando *gradle install* e acesse o console Gogo para atualização dos módulos.

É importante interromper a execução dos *bundles* para que suas classes de importação e exportação sejam atualizadas. Para isso, execute o comando *stop* para os dois *bundles*, e, em seguida, o comando *update*, também para os dois *bundles*. Depois, execute o comando *start*: primeiro para o *bundle* “Hello Service”; e depois para o *bundle* “Devmedia OSGi Exemplo”.

# Modularização dinâmica em Java com OSGi

Saiba que é importante iniciar primeiro o “Hello Service” para que o *container* OSGi registre o serviço a ser consumido pelo “Devmedia OSGi Exemplo”, caso contrário, o segundo *bundle* não encontrará nenhum serviço registrado pelo *container*, o que acarretará em uma exceção.

O *output* desse procedimento é apresentado na **Figura 14**. Veja que, após realizar o início do *bundle Devmedia OSGi Exemplo*, as mensagens incluídas na classe **HelloServiceFactory** são impressas no *output*, evidenciando que o objeto fábrica está entregando corretamente os objetos de serviço.

Realizada a inicialização do *bundle Devmedia OSGi Exemplo*, execute agora o comando *stop* para ele. Observe que no *output* encontraremos as mensagens definidas na **HelloServiceFactory** para o momento de liberação da instância, como demonstrado na **Figura 15**.

A partir dos exemplos apresentados neste artigo, foram introduzidos os conceitos básicos para o desenvolvimento de uma aplicação modular usando OSGi, como o ciclo de vida dos módulos, o compartilhamento de classes entre eles e duas abordagens para o registro de serviços no *container*.

```
g! lb
START LEVEL 1
ID|State |Level|Name
0|Active | 0|System Bundle (5.4.0)|5.4.0
1|Active | 1|Apache Felix Bundle Repository (2.0.6)|2.0.6
2|Active | 1|Apache Felix Gogo Command (0.16.0)|0.16.0
3|Active | 1|Apache Felix Gogo Runtime (0.16.2)|0.16.2
4|Active | 1|Apache Felix Gogo Shell (0.10.0)|0.10.0
5|Resolved | 1|Devmedia OSGi Exemplo (1.0.0)|1.0.0
6|Resolved | 1|Hello Service (1.0.0)|1.0.0
g! update 5
g! update 6
g! start 6
g! start 5
Cirando objeto do tipo HelloService para devmedia osgi exemplo
Quantidade de bundles usando o serviço: 1
Executando o metodo sayHello()
Say Hello
g!
```

**Figura 14.** Output demonstrando a execução do código implementado pela classe **HelloServiceFactory**

```
g! lb
START LEVEL 1
ID|State |Level|Name
0|Active | 0|System Bundle (5.4.0)|5.4.0
1|Active | 1|Apache Felix Bundle Repository (2.0.6)|2.0.6
2|Active | 1|Apache Felix Gogo Command (0.16.0)|0.16.0
3|Active | 1|Apache Felix Gogo Runtime (0.16.2)|0.16.2
4|Active | 1|Apache Felix Gogo Shell (0.10.0)|0.10.0
5|Active | 1|Devmedia OSGi Exemplo (1.0.0)|1.0.0
6|Active | 1|Hello Service (1.0.0)|1.0.0
g! stop 5
Goodbye World
Removendo objeto do tipo HelloService para devmedia osgi exemplo
Quantidade de bundles usando o serviço: 0
g!
```

**Figura 15.** Output do comando *stop* no *bundle Devmedia OSGi Exemplo*. Demonstra o código implementado na classe **HelloServiceFactory**

Além disso, vimos um pouco do histórico dessa tecnologia, que, mesmo sendo baseada em uma especificação fora do JCP, tornou-se muito popular.

Ao passo que você se aprofundar mais sobre essa tecnologia, verá que existem muitas ferramentas que facilitam bastante o desenvolvimento, como o **Bnd** (BuNDle), criado por Peter Kriens (da própria OSGi Alliance), que ajuda a criar módulos a partir da análise das classes do pacote JAR, simplificando a adaptação de componentes existentes em *bundles* OSGi (esse processo é geralmente chamado “OSGify”). Como exemplo, podemos dizer que vamos “OSGify um componente”, ou seja, transformá-lo em um *bundle* OSGi). Vale a pena analisar também os outros *containers*, como o recomendadíssimo Equinox, que vem por *default* no Eclipse e se trata da implementação de referência da especificação OSGi.

Por fim, o que se pode agregar de maior valor ao seguir com um desenvolvimento na linha do OSGi é a abstração de toda a aplicação em múltiplos módulos, cada um com uma responsabilidade bem definida e, de preferência, com um escopo bem restrito. Isso certamente eleva a qualidade de abstração do código, levando a um alto grau de reutilização. Ademais, aproveitando o *hype* acerca dos microsserviços que presenciamos nos últimos anos, muitas pessoas têm dito que OSGi é o caminho natural para uma solução baseada nesse conceito, já que muitos dos fundamentos são exatamente os mesmos. Assim, a transformação de um *bundle* OSGi em um microsserviço escalável e independente é uma tarefa extremamente simples. Sobre esse tema, vale assistir a uma apresentação de Milen Dyankov no Devoxx, ocorrido em novembro de 2015, cujo endereço encontra-se na seção **Links**.

## Autor



**André Luiz Martins Fabbro**

[andre.fabbro@gmail.com](mailto:andre.fabbro@gmail.com)

Graduado em Tecnologia em Informática pela UNICAMP. Há mais de 10 anos atuando como consultor Java EE. Hoje trabalha como Consultor na Liferay Inc.



## Links:

**Site do OSGi Alliance.**

<http://www.osgi.org>

**Site do Apache Felix.**

<http://felix.apache.org>

**Página do Gradle.**

<http://www.gradle.org>

**Apresentação de Milen Dyankov sobre microsserviços e OSGi, no Devoxx 2015.**

[https://youtu.be/077777Zy\\_HE](https://youtu.be/077777Zy_HE)

# Programador Java: Por onde começar?

Descubra nesse vídeo como entrar na carreira Java com o pé direito!



DEVMEDIA

# 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