



Edição 141 :: R\$ 14,90

 DEVMEDIA

MENTORING: Como criar extensões CDI
Adicione flexibilidade com baixo
acoplamento e fácil manutenção

Micro serviços com Spring e Reactor
Aprenda a criar micro
serviços baseados em RESTful

CONHEÇA O SPRING CLOUD

Sua nova opção
para computação
em nuvem



Java EE 7 na prática – Parte 1
Construa uma aplicação de
Controle de Projetos

Mecanismos de busca com Spring Data Solr
Trazendo a busca textual
às suas aplicações



MVP

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

APlique esse investimento
na sua carreira...

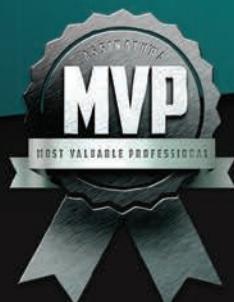
E mostre ao mercado
quanto você vale!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API¹
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 141 • 2015 • ISSN 1676-8361



MVP

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

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Diogo Souza (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa Romulo Araujo

Diagramação Janete Feitosa

Distribuição

FC Comercial e Distribuidora S.A.

Rua Teodoro da Silva, 907, Grajaú - RJ

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

Atendimento ao leitor

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

Publicidade

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

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

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

@eduspinola / @Java_Magazine

CURSOS ONLINE

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



CONHEÇA ALGUNS DOS CURSOS:

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

Para mais informações :

www.devmedia.com.br/curso/javamagazine
(21) 3382-5038



Sumário

Artigo no estilo Curso

06 – Como construir uma aplicação de Controle de Projetos com JavaEE – Parte 1

[Edmar Elias Bregagnoli]

Conteúdo sobre Boas Práticas, Artigo do tipo Mentoring

18 – Como criar extensões CDI

[Adolfo Eloy]

Conteúdo sobre Novidades, Artigo no estilo Solução Completa

34 – Como criar sistemas nas nuvens com Spring Cloud

[Leonardo Gonçalves da Silva]

Conteúdo sobre Boas Práticas, Artigo no estilo Solução Completa

48 – Micro serviços RESTful com Spring Boot e Reactor

[André Luiz Martins Fabbro]

Artigo no estilo Solução Completa, Conteúdo sobre Novidades

62 – Desenvolva mecanismos de busca diferenciados com Spring Data Solr

[Francisco Späth]



Dê seu feedback sobre esta edição!

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

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

www.devmedia.com.br/javamagazine/feedback



CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

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

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

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

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

Na suite ToolsCloud você tem a sua disposição:

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

Acompanhe a Toolscloud:



toolscloud@toolscloud.com



twitter.com/toolscloud



Como construir uma aplicação de Controle de Projetos com JavaEE

– Parte 1

Conheça os recursos da plataforma Java EE 7

ESTE ARTIGO FAZ PARTE DE UM CURSO

Acada nova versão da plataforma Java Enterprise Edition, muitas são as atualizações em especificações, além, é claro, da adição de novas tecnologias. Neste contexto, dentre as principais novidades lançadas na Java EE 7, podemos citar a nova API de aplicações Batch, que se baseou na já consolidada API Spring Batch, a API JSON-P, para manipulação de documentos no formato JSON, e a *Concurrency Utilities*, uma nova especificação que traz uma API para lidar com concorrência em aplicações Java EE. Muito semelhante à API ExecutorService já presente na plataforma Java SE, a *Concurrency Utilities* busca resolver um problema bastante comum: lidar com threads em aplicações corporativas.

Além das novas especificações, várias tecnologias também obtiveram importantes atualizações como, por exemplo, a JPA 2.1, com o suporte transparente a chamadas de procedures, EJB 3.2, que passou a dar suporte a transações dentro dos métodos de *callback* do ciclo de vida dos EJBs, Bean Validation 1.1, cujas anotações podem ser utilizadas em parâmetros de métodos, e JSF 2.2, que possibilitou uma melhor integração com o HTML 5 e trouxe as novas *annotations* @ViewScoped e @FlowScoped, que ajudam na criação de telas semelhantes a telas de *wizards* ou workflow, dando melhor suporte a fluxos de negócios passo a passo.

Fique por dentro

A plataforma Java EE sempre foi considerada um sucesso para o desenvolvimento de aplicações corporativas. Por isso, nesse artigo vamos dar início ao estudo de vários recursos existentes em sua nova versão, a Java EE 7. Juntamente com essa plataforma, também utilizaremos componentes do framework PrimeFaces para criação das telas de um sistema exemplo (neste caso, de controle de projetos). Aqui, criaremos a arquitetura da aplicação, o banco de dados, a primeira tela de cadastro e um serviço RESTful para popular dados no banco de dados de cidades e estados, recursos presentes em praticamente toda aplicação web sendo, portanto, um conteúdo fundamental ao seu dia a dia.

Ademais, tivemos também a nova versão da especificação JAX-RS, a 2.0, que possui uma nova API de cliente, quando até então era necessário utilizar bibliotecas ou APIs de terceiros, e também a API de JMS 2.0, que reduziu consideravelmente a verbosidade de código das versões anteriores. Assim, agora é mais simples e rápido criar componentes para enviar e receber mensagens JMS.

São muitas as novidades presentes na plataforma Java EE 7 e o leitor pode conhecer cada uma delas ao ler as Edições da Java Magazine 117, 119, 121, 122, 123, 124 e 125. Nessas edições, além dos conceitos teóricos da Java EE, o leitor poderá acompanhar vários exemplos práticos reforçando esses conceitos.

Devido à quantidade de mudanças e novidades, se torna inviável demonstrar todas elas em um único artigo. Sendo assim, vamos explorar alguns desses novos conceitos através de um exemplo prático.

A aplicação Controle de Projetos

A nossa aplicação exemplo tem como objetivo controlar os projetos desenvolvidos por uma empresa, possibilitando para isso a alocação de colaboradores, o controle da situação atual dos projetos e o cadastro de departamentos. Para termos uma ideia de como as classes do modelo de negócio desse sistema se relacionam, a **Figura 1** apresenta o diagrama de classes da aplicação.

Nesta primeira parte do artigo construiremos a tela para cadastro de projetos. Nessa tela deve ser informado o nome do projeto, o orçamento, a descrição e a sua situação do projeto (em desenvolvimento, pré-venda, em análise e pós-venda).

Como um dos diferenciais, visando explorar mais recursos da plataforma Java EE, no projeto será construído um web service RESTful com JAX-RS, para que possamos, através da chamada a este serviço, inserir dados referentes a cidades e estados na base de dados.

Ao final desse primeiro artigo o leitor terá toda a arquitetura da aplicação criada, o banco de dados e uma tela de cadastro de projetos construída. Além disso, veremos como criar um componente para popular nosso banco de dados com dados de cidades e estados que serão utilizados na tela de cadastro de colaboradores da empresa.

Visão geral da arquitetura do sistema

As principais tecnologias para desenvolver o sistema estão presentes na Java EE 7. Outras, como o Maven e o PrimeFaces, são amplamente difundidas no mercado e perfeitamente compatíveis com projetos desenvolvidos na plataforma Java EE.

Para configurar o projeto, utilizaremos o gerenciador de dependências Maven. Com essa ferramenta é possível gerenciar projetos Java de forma simplificada, configurando as dependências da aplicação em um único arquivo XML, além de permitir o gerenciamento do ciclo de vida do desenvolvimento (compilação, testes unitários, empacotamento e distribuição).

Além do Maven, a arquitetura do projeto terá o controle das dependências dos componentes feito pelo CDI 1.1, JPA 2.1 para acesso ao banco de dados na camada

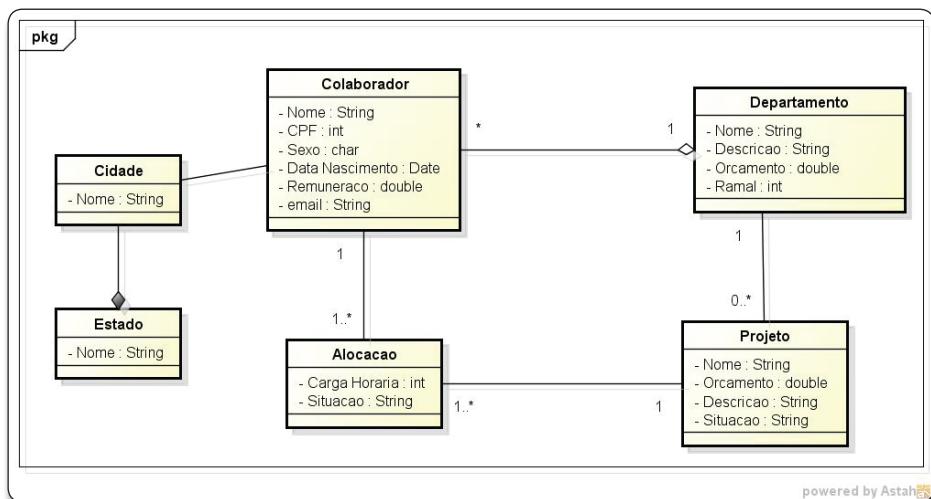


Figura 1. Diagrama de classes do projeto

de persistência, EJB 3.2 para o desenvolvimento da camada de negócios, RESTful web services para popular dados de cidades e estados no banco de dados e JSF 2.2 para criação das telas do sistema juntamente com os componentes de interface rica do PrimeFaces 5. Por fim, para fundamentarmos a base da aplicação, será adotado o Eclipse Kepler como IDE e a versão 1.7 do Java Development Kit (JDK).

Depois de pronto, nosso exemplo será implantado no servidor de aplicações GlassFish 4.0, compatível com a versão do Java EE 7. E o banco de dados escolhido para o desenvolvimento dessa solução foi o MySQL. Apesar disso, o leitor pode adotar o banco de dados que estiver mais familiarizado.

Todos os recursos necessários para implementação do sistema de controle de projetos devem estar instalados na máquina. O caminho para download desses pode ser encontrado na seção **Links**.

Construindo a aplicação Controle de Projetos

Com a arquitetura do projeto definida, podemos dar início à construção da aplicação. Para criar um projeto Maven dentro do Eclipse, clique no menu **File**, selecione **New** e depois **Other**. Feito isso, surgirá uma tela na qual devemos selecionar **Maven** e clicar em **Maven Project**. Na sequência, marque a opção **Create a simple Project** e clique em **Next** para darmos prosseguimento à configuração de nosso projeto.

Nesse momento, vamos definir os parâmetros de configuração do Maven no nosso projeto. Portanto, informe no campo **Group Id** o valor “**br.com.devmedia.controleprojeto**”, em **Artifact Id** defina o valor “**JMControleProjeto**”, em **Package** especifique o valor “**war**” e em seguida clique em **Finish**.

Para completar a configuração inicial do projeto, ainda precisamos determinar algumas propriedades importantes. Assim, clique com o botão direito em cima do projeto **JMControleProjeto** no Eclipse e escolha a opção **Properties**. Na tela que será exibida, selecione a opção **Project Facets**. Em seguida, vamos ajustar as configurações relacionadas às versões do Java, JSF e do módulo web que vamos fazer uso para desenvolver o sistema. Desse modo, selecione a versão 3.1 para **Dynamic Web Module**, o que especifica que vamos trabalhar com a última versão da API de Servlets. Na opção **Java**, escolha a versão 1.7, e por fim, a versão 2.2 para **JavaServer Faces**. Depois clique no botão **Ok**. Após essas configurações a aplicação **JMControleProjeto** será criada no Eclipse com a estrutura de projetos do Maven.

Por fim, para concluir a criação de nosso projeto, vamos configurar o arquivo **web.xml** como mostra a **Listagem 1**.

Nesse arquivo podemos notar que a versão da API de Servlets que estamos utilizando é a 3.1, versão mais recente oferecida pela plataforma Java EE.

Como construir uma aplicação de Controle de Projetos com JavaEE – Parte 1

Listagem 1. Arquivo web.xml da aplicação.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">

  <display-name>Controle Projeto</display-name>

  <welcome-file-list>
    <welcome-file>Home.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>
```

Listagem 2. POM.xml com as configurações das dependências do projeto

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
02   xmlns:xsi="http://www.w3.org/2001/XMLSchema- instance"
03   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
04     http://maven.apache.org/xsd/maven-
05   4.0.0.xsd">
06   <modelVersion>4.0.0</modelVersion>
07   <groupId>br.com.devmedia.controlprojeto</groupId>
08   <artifactId>JMControleProjeto</artifactId>
09   <version>0.0.1-SNAPSHOT</version>
10   <packaging>war</packaging>
11   <name>JMControleProjeto</name>
12
13   <dependencies>
14     <!-- Java EE 7 -->
15     <dependency>
16       <groupId>javax</groupId>
17       <artifactId>javaee-api</artifactId>
18       <version>7.0</version>
19     </dependency>
20
21     <!-- Servlet 3.1 -->
22     <dependency>
23       <groupId>javax.servlet</groupId>
24       <artifactId>javax.servlet-api</artifactId>
25       <version>3.1.0</version>
26       <scope>provided</scope>
27     </dependency>
28
29     <!-- jsf 2 -->
30     <dependency>
31       <groupId>org.glassfish</groupId>
32       <artifactId>javax.faces</artifactId>
33       <version>2.2.8</version>
34       <scope>compile</scope>
35     </dependency>
36
37     <!-- PrimeFaces-->
38     <dependency>
39       <groupId>org.primefaces</groupId>
40       <artifactId>primefaces</artifactId>
41       <version>5.0</version>
42     </dependency>
43     <!-- Hibernate -->
44     <dependency>
45       <groupId>org.hibernate</groupId>
46       <artifactId>hibernate-core</artifactId>
47       <version>4.3.6.Final</version>
48     </dependency>
49     <dependency>
50       <groupId>org.hibernate</groupId>
51       <artifactId>hibernate-entitymanager</artifactId>
52       <version>4.3.6.Final</version>
53     </dependency>
54     <dependency>
55       <groupId>org.hibernate.javax.persistence</groupId>
56       <artifactId>hibernate-jpa-2.0-api</artifactId>
57       <version>1.0.1.Final</version>
58     </dependency>
59     <!-- Junit para Testes Unitarios -->
60     <dependency>
61       <groupId>junit</groupId>
62       <artifactId>junit</artifactId>
63       <version>4.11</version>
64       <scope>test</scope>
65     </dependency>
66   </dependencies>
67
68   <build>
69     <finalName>JMControleProjeto</finalName>
70     <plugins>
71       <plugin>
72         <artifactId>maven-compiler-plugin</artifactId>
73         <version>3.0</version>
74         <configuration>
75           <source>1.7</source>
76           <target>1.7</target>
77         </configuration>
78       </plugin>
79     </plugins>
80   </build>
81
82</project>
```

Além disso, é definido qual será a página inicial de nosso sistema, neste caso a *Home.xhtml*, e também configuramos o FacesServlet, responsável por gerenciar todo o ciclo de vida das requisições web utilizando JavaServer Faces (JSF). Por fim, criamos um mapeamento para que todas as requisições enviadas de páginas com extensão XHTML sejam controladas pelo FacesServlet.

Configurando o arquivo pom.xml

Após configurarmos o *web.xml* é necessário discriminar as bibliotecas e APIs que serão empregadas para a construção do nosso projeto. Deste modo, abra o arquivo *pom.xml* e mantenha as configurações da forma como mostra a Listagem 2.

Nesse arquivo informamos todas as ferramentas que serão utilizadas e suas respectivas versões. Vamos agora entender melhor cada uma das configurações presentes no *pom.xml* que acabamos de definir:

- Linhas 05 a 10: Configuração do projeto Maven. Nessas linhas informamos o **groupId**, que é a identificação do grupo ao qual o projeto pertence (em geral considera-se a estrutura de pacotes da aplicação); o **artifactId**, nome do artefato *.war* gerado no processo

de build (neste caso *JMControleProjeto.war*); **version**, que define a versão atual do projeto; **packaging**, que especifica o tipo de empacotamento utilizado; e a tag **name**, que determina o nome do projeto;

- Linhas 14 a 18: configura a versão da Java EE que iremos utilizar, no caso a versão 7;
- Linhas 21 a 26: informa que iremos trabalhar com a versão 3.1 da API de Servlets;
- Linhas 29 a 34: configura a dependência da versão JSF 2.2, que será fornecida pelas bibliotecas presentes no servidor GlassFish;
- Linhas 37 a 41: informa a dependência do framework PrimeFaces 5.0;
- Linhas 44 a 57: configura a dependência do Hibernate como framework de persistência. A versão do Hibernate escolhida implementa a versão JPA 2.0;
- Linhas 60 a 66: informa a dependência para o framework de teste unitário JUnit;
- Linhas 70 a 79: configura que o projeto será compilado com a versão 7 do Java.

Definidas as dependências do projeto, podemos agora criar o banco de dados e configurar o datasource no servidor de aplicações.

Criando o banco de dados e o datasource

Para criarmos o banco de dados de nossa aplicação é necessário que o MySQL esteja devidamente instalado na máquina e iniciado. Dessa forma, podemos definir uma base de dados com o nome **dbcontroleprojeto**. Com a base de dados especificada, devemos gerar as tabelas que serão utilizadas pela aplicação. Para isso, está disponível no site da Java Magazine o código SQL indicado. Basta rodar esse código dentro da base de dados recém-criada. Feito isso, já teremos nosso banco pronto para ser utilizado.

O próximo passo é especificar o DataSource de nossa aplicação no servidor GlassFish. Para tanto, com o servidor iniciado, acesse seu console administrativo pelo caminho *http://localhost:8484*. Nesse momento uma tela de login será apresentada. Quando instalamos o GlassFish, geralmente o usuário e senha padrões são *admin* e *adminadmin*, respectivamente. Dito isso, basta se autenticar no console para poder iniciar o processo de criação do DataSource.

Deste modo, o primeiro passo será configurar o pool de conexões (*Connection Pool*), responsável por gerenciar as conexões com o banco de dados. Este recurso pode ser criado acessando o menu *Resources > JDBC > Connection Pools*, que fica do lado esquerdo da tela do console administrativo. Na janela que aparece, clique no botão *New*. Nesse momento, uma nova tela será exibida e os parâmetros para criação do pool devem ser preenchidos de acordo com os valores mostrados na **Tabela 1**.

Após informar esses valores, clique no botão *Save* para que o Pool de Conexões seja efetivamente criado.

O próximo passo é configurar o DataSource. Para isso, acesse o menu *Resources > JDBC > JDBC Resources* e clique no botão *New*. Na tela que aparece, informe para o parâmetro *JNDI Name* o

valor “*jdbc/ControleProjeto*” e para *Pool Name*, o valor “*dbControleProjeto*”.

PoolName é o nome do Pool de conexões definido anteriormente e *JNDI Name* é o nome que identifica o DataSource. Esse nome será utilizado na aplicação para realizar a conexão com o banco de dados.

Com essas configurações preenchidas, clique no botão *Save* para que o DataSource de nossa aplicação seja gerado.

| Parâmetro | Valor |
|-----------------------|---|
| Pool Name | dbControleProjeto |
| Resource Type | Javax.sql.ConnectionPoolDataSource |
| Data source Classname | com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource |
| portNumber | 3306 |
| databaseName | dbControleProjeto |
| serverName | localhost |
| User | Root |
| Password | Root |
| url | jdbc:mysql://localhost:3306/dbcontroleprojeto |

Tabela 1. Parâmetros para configuração do pool de conexões com o banco de dados

Criando as entidades de acesso ao banco de dados

Com a estrutura do projeto pronta, juntamente com o banco de dados e o DataSource devidamente configurados no GlassFish, devemos agora criar as entidades JPA. Essas entidades devem ser salvas no pacote **br.com.devmedia.controleprojeto.entidades**. Uma boa prática é organizar nossas classes em pacotes de acordo com suas funcionalidades. Por exemplo, é importante que todas as entidades da aplicação fiquem em um pacote específico, assim como todas as classes responsáveis pelas regras de negócio da aplicação devem ser salvas em um pacote exclusivo para elas.

De acordo com a **Figura 1**, as entidades que devem ser concebidas são: **Cidade**, **Estado**, **Alocacao**, **Colaborador**, **Departamento** e **Projeto**. As **Listagens 3 a 8** apresentam o código relacionado.

Para mapear as entidades da aplicação utilizamos algumas anotações tanto em nível de classe quanto em nível de atributo. Dentro as anotações em nível de classe temos **@Entity** e **@Table**. Ao anotarmos uma classe com **@Entity** marcamos essa classe como uma entidade JPA, representando assim uma tabela do banco de dados. Para determinar qual tabela essa entidade está representando, podemos utilizar a anotação **@Table** e seu **name**. Lembre-se que, se a anotação **@Table** não for informada, o nome da tabela no banco de dados deve ser exatamente igual ao nome da entidade.

Quanto às anotações em nível de atributos, uma das principais opções é **@Id**. Esta anotação marca o atributo como o identificador único da entidade. Geralmente campos que são chaves primárias nas tabelas recebem essa anotação. Se o atributo que receber a anotação **@Id** for uma chave primária, é necessário informar a estratégia de geração de valores para essa chave.

Como construir uma aplicação de Controle de Projetos com JavaEE – Parte 1

Listagem 3. Código da entidade Cidade.

```
package br.com.devmedia.controleprojeto.entidades;  
  
//imports omitidos...  
  
@Entity  
@Table(name = "cidade")  
public class Cidade implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    @Column(name = "ID")  
    private Integer id;  
  
    @Column(name = "NOME")  
    private String nome;  
  
    @JoinColumn(name = "ESTADO_ID", referencedColumnName = "ID")  
    @ManyToOne  
    private Estado estadoid;  
  
    public Cidade(){  
    }  
  
    public Cidade(Integer id) {  
        this.id = id;  
    }  
  
    //Getters e setters omitidos...  
}
```

Listagem 4. Código da entidade Estado.

```
package br.com.devmedia.controleprojeto.entidades;  
  
//imports omitidos...  
  
@Entity  
@Table(name = "estado")  
public class Estado implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    @Column(name = "ID")  
    private Integer id;  
  
    @Column(name = "NOME")  
    private String nome;  
  
    @OneToMany(mappedBy = "estadoid")  
    private List<Cidade> cidadeList;  
  
    public Estado(){  
    }  
  
    public Estado(Integer id) {  
        this.id = id;  
    }  
  
    //Getters e Setters omitidos...  
}
```

Listagem 5. Código da entidade Alocacao.

```
package br.com.devmedia.controleprojeto.entidades;  
  
//imports omitidos...  
  
@Entity  
@Table(name = "alocacao")  
public class Alocacao implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    @Column(name = "ID")  
    private Integer id;  
    @Column(name = "CARGA_HORARIA")  
    private Integer cargaHoraria;  
  
    @Column(name = "SITUACAO")  
    private Integer situacao;  
  
    @JoinColumn(name = "COLABORADOR_ID", referencedColumnName = "ID")  
    @ManyToOne  
    private Colaborador colaboradorid;  
    @JoinColumn(name = "PROJETO_ID", referencedColumnName = "ID")  
    @ManyToOne  
    private Projeto projetoid;  
  
    public Alocacao(){  
    }  
  
    public Alocacao(Integer id) {  
        this.id = id;  
    }  
  
    //Getters e Setters omitidos...  
}
```

Listagem 6. Código da entidade Colaborador.

```
package br.com.devmedia.controleprojeto.entidades;  
//imports omitidos...  
@Entity  
@Table(name = "colaborador")  
public class Colaborador implements Serializable {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    @Column(name = "ID")  
    private Integer id;  
    @Column(name = "NOME")  
    private String nome;  
    @Column(name = "SEXO")  
    private String sexo;  
    @Column(name = "DATA_NASCIMENTO")  
    @Temporal(TemporalType.DATE)  
    private Date dataNascimento;  
    @Column(name = "REMUNERACAO")  
    private Double remuneracao;  
    @Column(name = "EMAIL")  
    private String email;  
    @JoinColumn(name = "DEPARTAMENTO_ID", referencedColumnName = "ID")  
    @ManyToOne  
    private Departamento departamentoid;  
    @OneToMany(mappedBy = "colaboradorid")  
    private List<Alocacao> alocacolist;  
    public Colaborador(){  
    }  
    public Colaborador(Integer id) {  
        this.id = id;  
    }  
    //Getters e Setters omitidos...  
}
```

Listagem 7. Código da entidade Departamento.

```
package br.com.devmedia.controlprojeto.entidades;
//imports omitidos...

@Entity
@Table(name = "departamento")
public class Departamento implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Integer id;
    @Column(name = "NOME")
    private String nome;
    @Column(name = "DESCRICAO")
    private String descricao;
    @Column(name = "ORCAMENTO")
    private Double orcamento;
    @Column(name = "RAMAL")
    private Integer ramal;
    @OneToMany(mappedBy = "departamentold")
    private List<Colaborador> colaboradorList;

    public Departamento() {
    }

    public Departamento(Integer id) {
        this.id = id;
    }

    //Getters e Setters omitidos...
}
```

Listagem 8. Código da entidade Projeto.

```
package br.com.devmedia.controlprojeto.entidades;
//imports omitidos...

@Entity
@Table(name = "projeto")
public class Projeto implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Integer id;
    @Column(name = "NOME")
    private String nome;
    @Column(name = "DESCRICAO")
    private String descricao;
    @Column(name = "SITUACAO")
    private String situacao;
    @Column(name = "ORCAMENTO")
    private Double orcamento;

    @OneToMany(mappedBy = "projetold")
    private List<Alocacao> alocacaoList;

    public Projeto() {
    }

    public Projeto(Integer id) {
        this.id = id;
    }

    //Getters e Setters omitidos...
}
```

Com isso, quando uma operação de inserção for realizada no banco, um valor para a chave primária será gerado.

No nosso exemplo utilizamos a anotação `@.GenerationType` para informar a estratégia que será usada; nesse caso: `GenerationType.IDENTITY`. A estratégia geralmente está relacionada com a forma como o banco gera os valores para as chaves primárias de suas tabelas. O MySQL, por exemplo, (banco de dados utilizado em nosso projeto) gera os valores para as chaves primárias por auto incremento. Sendo assim, a estratégia `GenerationType.IDENTITY` é a recomendada.

Para que os atributos das entidades sejam relacionados com as colunas físicas das tabelas do banco de dados, utilizamos a anotação `@Column`. Por exemplo, na **Listagem 3**, o atributo `nome` recebe a anotação `@Column(name="NOME")`. Com essa anotação, o atributo `nome` da classe **Cidade** está representando a coluna NOME presente na tabela “Cidade” do banco de dados.

Outro tipo de mapeamento presente nas entidades de nossa aplicação é o MUITOS-PARA-UM. Um exemplo desse tipo de mapeamento pode ser observado no atributo `colaboradorId` da entidade **Alocacao**, estabelecendo assim uma ligação com a entidade **Colaborador**.

A semântica desse tipo de relacionamento pode ser entendida da seguinte maneira: um registro presente na tabela *Colaborador* pode estar relacionado a mais de um registro da tabela *Alocacao* e um registro presente na tabela *Alocacao* está relacionado a apenas um registro da tabela *Colaborador*.

Na entidade **Alocacao**, o atributo `colaboradorId` possui duas anotações. A primeira delas é a `@ManyToOne`, que identifica que temos um relacionamento MUITOS-PARA-UM. A outra é a anotação `@JoinColumn`. Esta possui o parâmetro `name` para indicarmos a chave estrangeira da tabela *Alocacao* (`COLABORADOR_ID`) e que recebe o valor da chave primária da tabela *Colaborador*. A anotação `@JoinColumn` possui também o parâmetro `referencedColumnName`, que configura qual é a chave primária da tabela *Colaborador*, neste caso a coluna ID.

No parágrafo anterior analisamos o relacionamento `@ManyToOne` existente na entidade **Alocacao**. Agora, nos vem uma pergunta: Como é o mapeamento do relacionamento da entidade **Colaborador** com a entidade **Alocacao**?

Ao analisar o código da entidade **Colaborador**, encontramos a anotação `@OneToMany` no atributo `alocacaoList`. Nesse tipo de relacionamento, um registro da entidade **Colaborador** pode estar relacionado a muitos registros da entidade **Alocacao**. Por isso o atributo `alocacaoList` é do tipo `List<Alocacao>`.

Ao observar a anotação `@OneToMany`, presente no atributo `alocacaoList`, encontramos o parâmetro `mappedBy` para indicar a qual atributo da entidade **Colaborador** esse relacionamento pertence; neste caso o atributo `colaboradorId`.

Com todas as entidades mapeadas, podemos realizar as consultas necessárias na base de dados, efetuar o cadastro das informações dos projetos, colaboradores e também a alocação dos colaboradores nos projetos.

Já que estamos fazendo uso de JPA para acessar a base de dados, precisamos agora configurar o *persistence.xml*. Neste arquivo é necessário definir uma unidade de persistência, para que durante as operações envolvendo o acesso ao banco de dados (operações de consulta, atualização, inserção e remoção) um contexto de persistência seja criado.

Em nosso projeto nomeamos a *persistence-unit* como **ControleProjetoPU**. Além disso, configuramos o *transaction-type* com o valor **JTA**. Ao especificarmos o tipo de transação como **JTA** estamos definindo que utilizaremos a API **JTA** da Java EE para controlar a transação com o banco de dados. Nesse caso o controle das transações fica por conta do servidor de aplicação, sendo ele o encarregado por realizar as tarefas de *commit* e *rollback* com o banco de dados quando necessário.

Esse arquivo pode ser visualizado na **Listagem 9** e deve ser salvo no diretório *src/main/resources/META-INF*.

Listagem 9. Conteúdo do arquivo *persistence.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  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">

  <persistence-unit name="ControleProjetoPU" transaction-type="JTA">

    <jta-data-source>jdbc/ControleProjeto</jta-data-source>

    <properties/>

  </persistence-unit>

</persistence>
```

Neste momento temos a nossa aplicação configurada no Eclipse, o banco de dados criado, as configurações de datasource realizadas no servidor de aplicação e as entidades que representam as tabelas de nosso banco de dados relacional implementadas. No próximo tópico iremos começar a implementar as funcionalidades necessárias para o sistema de controle de projetos.

Serviço RESTful para popular as tabelas do banco

A arquitetura REST (*Representational State Transfer*, ou *Transferência de Estado Representacional*, em tradução livre) tornou mais simples a construção e manutenção de serviços web, em comparação com o tradicional protocolo **SOAP**.

Na arquitetura REST, qualquer informação disponibilizada por um serviço é chamada de **Resource** (ou **Recurso**). Funcionalidades como busca por clientes ou uma pesquisa de endereço por CEP são exemplos de recursos.

Em REST, cada recurso deve possuir um identificador único. No caso de recursos web, esse identificador é conhecido como **URI** (*Uniform Resource Identifier*). Por exemplo: [www.devmedia.com.br/revista-java-magazine](http://www.devmedia.com.br.revista-java-magazine) é um exemplo de URI que identifica a página da revista Java Magazine.

Até aqui apresentamos algumas características de uma arquitetura REST, porém com certa frequência nos deparamos com o termo **RESTful**. Esse termo é atribuído a aplicações que seguem a arquitetura REST, ou seja, REST é um conceito, um paradigma, e RESTful é uma aplicação ou serviço web que é construído utilizando como base esse paradigma.

Com o intuito de facilitar a criação de serviços RESTful em Java, foi definida uma especificação chamada **JAX-RS** que especifica todos os requisitos para desenvolver web services conforme essa arquitetura. Na Java EE 7 essa especificação foi atualizada para a versão 2.0 e será ela que vamos empregar para construir nosso serviço para popular com dados de cidades e estados as tabelas do banco de dados da aplicação.

Dito isso, primeiramente devemos criar a classe **RESTConfiguration** no pacote **br.com.devmedia.controlprojeto.rest**, com o código indicado na **Listagem 10**.

Listagem 10. Classe de configuração *RESTConfiguration*.

```
package br.com.devmedia.controlprojeto.rest;

//imports omitidos...

@ApplicationPath(value = "recursos")
public class RESTConfiguration extends Application {
```

Essa classe foi criada para determinar que todos os serviços RESTful de nosso sistema terão como path principal: *localhost:8080/JMControleProjeto/recursos/<nome_do_recurso>*. Esta definição é feita através da anotação **@ApplicationPath**, observada sobre a assinatura da classe.

Agora já podemos criar o serviço RESTful responsável por popular nossa base de dados. Para isso, crie a classe **PopulaBancoRest** no pacote **br.com.devmedia.controlprojeto.rest** (mesmo pacote da classe **RESTConfiguration**), com o código da **Listagem 11**.

Observe nas linhas 3 e 4 que definimos **PopulaBancoRest** como um EJB através da anotação **@Stateless** e como uma classe de serviço REST através da anotação **@Path**. Nessa anotação, o parâmetro **value** foi configurado com */populaBancoRest*. Assim, o caminho que identifica esse recurso para que ele possa ser executado será: *http://localhost:8080/JMControleProjeto/recursos/populaBancoRest*, onde *localhost* representa o endereço IP do servidor, 8080 representa a porta padrão para aplicações web do GlassFish, *JMControleProjeto* sinaliza o nome de contexto da aplicação, *recursos* é o identificador presente na classe **RESTConfiguration** definido pela anotação **@ApplicationPath** e *populaBancoRest* é o nome do recurso REST especificado na classe.

Nas linhas 7 e 8 injetamos a classe **EntityManager** da JPA por meio da anotação **@PersistenceContext**. Com essa classe iremos executar as operações de consulta para saber se já existem cadastrados no banco de dados cidades e estados. Caso as tabelas relacionadas estejam vazias, iremos persistir/incluir essas informações com a

execução do método **popularTabelasEstadosCidades()** da classe **PopulaBancoRest**. Esse método recebe duas anotações: **@GET** e **@Produces**. Com **@GET**, determinamos que o método **popularTabelasEstadosCidades()** será acessado a partir da operação GET do protocolo HTTP.

Listagem 11. Classe que implementa o serviço RESTful para popular as tabelas.

```

01 package br.com.devmedia.controlprojeto.rest;
02 //imports omitidos...
03 @Stateless
04 @Path(value = "/populaBancoRest")
05 public class PopulaBancoRest {
06
07     @PersistenceContext(unitName="ControleProjetoPU",
08         type=PersistenceContextType.TRANSACTION)
09     private EntityManager em;
10
11    @GET
12    @Produces(MediaType.APPLICATION_JSON)
13    public Response popularTabelasEstadosCidades() {
14
15        Query query = em.createNamedQuery("Estado.findAll");
16
17        List<Estado> estados = query.getResultList();
18
19        if (estados.isEmpty()) {
20            popularBanco();
21            estados = query.getResultList();
22        }
23
24        return Response.ok().build();
25    }
26
27    private void popularBanco() {
28
29        Estado saoPaulo = new Estado();
30        saoPaulo.setNome("Sao Paulo");
31        em.persist(saoPaulo);
32
33        Estado minasGerais = new Estado();
34        minasGerais.setNome("Minas Gerais");
35        em.persist(minasGerais);
36
37        Estado rioJaneiro = new Estado();
38        rioJaneiro.setNome("Rio de Janeiro");
39        em.persist(rioJaneiro);
40
41        criarCidades();
42    }
43    private criarCidades () {
44        //Código para criar as cidades no banco de dados
45    }

```

Este protocolo, aliás, disponibiliza sete operações: GET, POST, PUT, DELETE, HEAD, OPTIONS e TRACE. Neste artigo, no entanto, não serão discutidos os detalhes de funcionamento de cada uma das operações HTTP existentes.

O serviço REST que estamos construindo utilizará a operação GET para ser executado. Assim, caso esse serviço seja chamado via método POST, por exemplo, um erro de código 405 (método não suportado) será retornado, pois estamos tentando chamar o recurso por um método HTTP diferente do que foi definido.

Além da anotação **@GET**, utilizamos a anotação **@Produces** (**MediaType.APPLICATION_JSON**). Esta solução informa qual será o tipo de retorno do serviço REST. Nesse caso, definimos que o objeto de resposta do serviço (**Response**) será um documento no formato JSON.

Nesse momento vale ressaltar que além do JSON existem outros tipos de retorno, como HTML e documentos no formato texto. Esses tipos de retorno podem ser selecionados por meio do enum **MediaType**.

Nas linhas 14 a 21, relacionadas ao método **popularTabelasEstadosCidades()**, é realizada uma consulta ao banco de dados para saber se já existem estados na base. Caso a consulta retorne uma lista vazia, chamamos o método **popularBanco()**. Este irá inserir os estados e chamar o método **criarCidades()** para também inserir as cidades no banco de dados.

Após a execução de todos esses passos, uma resposta de sucesso é retornada por meio do trecho de código da linha 23: **Response.ok().build()**. Com essa chamada é retornado o código de sucesso 200 do protocolo HTTP.

Tela de cadastro de projetos

Antes de criarmos a página de cadastro de projetos iremos definir uma estrutura de templates, com o intuito de padronizar a interface web e facilitar a construção de todas as páginas da aplicação. Para isso, utilizaremos Facelets, recurso presente no JSF desde a versão 2.0.

Com a estrutura de templates bem definida, conseguimos facilitar as tarefas de manutenção relacionadas às páginas e também aumentar o reuso de código. Com Facelets podemos criar uma estrutura com cabeçalho, menu, conteúdo e rodapé, de modo que ao codificarmos as páginas do sistema, estas possam ser encaixadas em cada uma dessas áreas de forma simples, facilitando o trabalho de design.

Nota

Para que as páginas sigam o padrão Java Facelets, o formato que devem ser salvas é o **.xhtml** (Extensible Hypertext Markup Language). A partir do JSF 2.0 esse é o formato padrão para criação de páginas com essa tecnologia.

Dito isso, a primeira página que iremos criar é a *template.xhtml*, que conterá a estrutura padrão para as demais páginas do sistema. A **Listagem 12** traz o código fonte dessa página.

Algumas características importantes do arquivo *template.xhtml* que podemos destacar são:

- Quando uma requisição AJAX for realizada, uma imagem de “carregando” será exibida na tela até que a requisição seja concluída. Essa funcionalidade encontra-se nas linhas 15 a 22, por meio do componente **ajax:status** do PrimeFaces;
- Das linhas 28 a 41 encontramos o menu com todas as funcionalidades da aplicação. Esse menu também é criado utilizando os componentes do PrimeFaces. Na linha 28 criamos a barra de menus por meio do componente **menuBar**. O acesso às funcionalidades

Listagem 12. Código fonte da página template.xhtml.

```
01 <!DOCTYPE html>
02 <html xmlns="http://www.w3.org/1999/xhtml"
03   xmlns:h="http://java.sun.com/jsf/html"
04   xmlns:f="http://java.sun.com/jsf/core"
05   xmlns:ui="http://java.sun.com/jsf/facelets"
06   xmlns:p="http://primefaces.org/ui">
07
08 <h:head>
09   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
10  <title><ui:insert name="titulo">Controle Projeto</ui:insert></title>
11  <h:outputStylesheet library="css" name="sistema.css"/>
12 </h:head>
13
14 <h:body>
15 <p:ajaxStatus styleClass="ajax-status">
16   <f:facet name="start">
17     <h:graphicImage library="imagens" name="carregando.gif"/>
18   </f:facet>
19   <f:facet name="complete">
20     <h:outputText value="" />
21   </f:facet>
22 </p:ajaxStatus>
23
24 <header>
25
26 <div style="float: left; margin-right: 110px">
27   <h:form style="display: inline-block">
28     <p:menubar styleClass="menu-sistema">
29
30       <p:submenu label="Cadastro">
31         <p:menuitem value="Colaborador"
32           outcome="/colaborador/CadastroColaborador.xhtml"/>
33         <p:menuitem value="Projeto"
34           outcome="/projeto/CadastroProjeto.xhtml"/>
35
36       <p:submenu label="Projeto">
37         <p:menuitem value="Alocacao"
38           outcome="/projeto/allocacao.xhtml"/>
39         <p:menuitem value="Alterar Projeto"
40           outcome="/projeto/alterarProjeto.xhtml"/>
41
42     </p:menubar>
43   </h:form>
44 <div style="clear: both"></div>
45 </header>
46
47 <div id="conteudo">
48   <ui:insert name="corpo"/>
49 </div>
50
51 <p:separator style="margin-top: 20px"/>
52
53 <footer>
54   Sistema de Controle de Projetos - versão 1.0.0
55 </footer>
56 </h:body>
57 </html>
```

da aplicação é feito pelos “itens” presentes nesse menu. Para criar esses “itens” utilizamos o componente **menuItem**. Na linha 32, por exemplo, informamos no parâmetro **outcome** de **menuItem** o caminho para a página de cadastro de projetos;

- Na linha 48 inserimos o componente **<ui:insert>**. Esse componente define o local onde as páginas da aplicação serão apresentadas quando forem acessadas. Para que uma página seja inserida nesse ponto ela deve definir uma estrutura chamada **corpo**. Um exemplo de como isso é realizado encontra-se na página de cadastro de projetos, que veremos a seguir.

Na **Listagem 13** apresentamos o código responsável pela tela de cadastro de projetos. Na segunda linha desse código temos o componente **<ui:composition template="/WEB-INF/template/template.xhtml">**, que determina que estamos utilizando o template *template.xhtml*.

Na linha 9 encontra-se o componente **<ui:define name="titulo">**. Com ele configuramos que o título da nossa página será *Cadastro de Projetos* e o *template.xhtml* se encarrega de renderizá-lo. Já na linha 11 encontra-se o componente **<ui:define name="corpo">**, dentro do qual estará presente todo o conteúdo da página de cadastro de projetos. Todo o código escrito dentro dessa tag será renderizado no “corpo” de *template.xhtml*.

Na linha 14, por sua vez, abrimos o **form** de cadastro e em seguida utilizamos o componente de mensagens do PrimeFaces **<p:messages>** para que todas as mensagens de erro e sucesso sejam renderizadas nesse componente. Já na linha 17 temos o componente **panelGrid**, que contém os campos que receberão as informações a serem preenchidas para o cadastro de projetos. Por exemplo, na linha 20, através do componente **h:inputText** do JSF, utilizamos a EL **#{projetoBean.projeto.nome}** para definir que o valor informado nesse *input text* seja atribuído ao atributo **nome** do objeto **projeto** do bean **ProjetoBean**.

Nessa mesma linha verificamos que este é um campo obrigatório, visto que o parâmetro **required** está setado como **true**. Caso não seja informado nenhum valor para esse campo, a mensagem de erro definida no parâmetro **requiredMessage** será exibida no componente de mensagem **p:messages**, que se encontra na linha 15. Essa mesma análise é válida para os demais campos.

Por fim, na linha 37, temos o código do botão **Salvar**, que ao ser selecionado irá enviar todas as informações preenchidas para o bean **ProjetoBean** executando o método **salvar()**.

A **Listagem 14** apresenta o código da página inicial da aplicação, chamada *Home.xhtml*. Na primeira linha desse arquivo encontramos o componente **<ui:composition>**, definindo que será utilizado o template apresentado na **Listagem 12**, e nas linhas seguintes são configurados os namespaces do JSF e do PrimeFaces.

Dando sequência à análise da página *Home.xhtml*, temos na linha 8 a definição do título da página como “Controle de Projetos”, através do componente **<ui:define>**, que define que esse título será inserido no *template.xhtml* no ponto onde se encontra o componente **<ui:insert name="titulo">**. Seguindo essa mesma

proposta, na linha 10 é especificado o corpo da página inicial, através da tag <ui:define name="corpo">, com uma mensagem de boas-vindas.

Listagem 13. Código fonte da página cadastroProjetos.xhtml.

```
01 <!DOCTYPE html>
02 <ui:composition template="/WEB-INF/template/template.xhtml"
03   xmlns="http://www.w3.org/1999/xhtml"
04   xmlns:h="http://java.sun.com/jsf/html"
05   xmlns:f="http://java.sun.com/jsf/core"
06   xmlns:ui="http://java.sun.com/jsf/facelets"
07   xmlns:p="http://primefaces.org/ui">
08
09   <ui:define name="titulo">Cadastro de Projetos</ui:define>
10
11   <ui:define name="corpo">
12     <h1>Cadastro de Projetos</h1>
13
14     <h:form id="frmCadastro">
15       <p:messages id="messages" showDetail="false" autoUpdate="true"
16         closable="true"/>
17
18       <p:panelGrid columns="2" id="painel" style="width: 100%; margin-top: 20px" columnClasses="rotulo, campo">
19
20         <p:outputLabel value="Nome" for="nome"/>
21         <h:inputText id="nome" value="#{projetoBean.projeto.nome}"
22           required="true" requiredMessage="Campo Nome obrigatorio"/>
23
24         <p:outputLabel value="Descrição" for="descricao"/>
25         <h:inputText id="descricao" value="#{projetoBean.projeto.descricao}"
26           required="true" requiredMessage="Campo Descrição obrigatorio"/>
27
28         <p:outputLabel value="Orçamento" for="orcamento"/>
29         <h:inputText id="orcamento" value="#{projetoBean.projeto.orcamento}"
30           required="true" requiredMessage="Campo Orçamento obrigatorio"/>
31
32         <p:outputLabel value="Situacao" for="situacao"/>
33         <p:selectOneMenu id="situacao" value="#{projetoBean.projeto.situacao}"
34           required="true" requiredMessage="Campo Situação obrigatorio">
35           <f:selectItem itemLabel="Selecione a situação do projeto"/>
36           <f:selectItems value="#{projetoBean.listaSituacao}" var="situacao"
37             itemLabel="#{situacao}" itemValue="#{situacao}"/>
38         </p:selectOneMenu>
39     </p:panelGrid>
40
41   </ui:define>
42 </ui:composition>
```

Listagem 14. Código fonte da página Home.xhtml.

```
01 <ui:composition template="/WEB-INF/template/template.xhtml"
02   xmlns="http://www.w3.org/1999/xhtml"
03   xmlns:h="http://java.sun.com/jsf/html"
04   xmlns:f="http://java.sun.com/jsf/core"
05   xmlns:ui="http://java.sun.com/jsf/facelets"
06   xmlns:p="http://primefaces.org/ui">
07
08   <ui:define name="titulo">Controle de Projetos</ui:define>
09
10   <ui:define name="corpo"> Bem-Vindo a Aplicação Controle de projetos
11 </ui:define>
12
13 </ui:composition>
```

Criadas as páginas da aplicação, vamos desenvolver o managed bean responsável por cadastrar os projetos. Esse componente receberá o nome de **ProjetoBean** e seu código fonte é apresentado na **Listagem 15**. Note que essa classe (**ProjetoBean**) está anotada com **@Named**, anotação CDI que marca uma classe como um bean gerenciável. Essa anotação foi utilizada porque a partir da Java EE 7 é recomendado que os managed beans do JSF sejam anotados com **@Named**, em vez de **@ManagedBean**. Além disso, utilizamos a anotação **@RequestScoped**, pertencente ao CDI, que marca um bean como sendo do escopo *request*. Apesar disso, essa anotação não é obrigatória, pois caso seja omitida o valor *default* para o escopo também será *request*.

Listagem 15. Código do bean ProjetoBean.

```
01 package br.com.devmedia.controleprojeto.mb;
02
03 //imports omitidos...
04
05 @Named
06 @RequestScoped
07 public class ProjetoBean implements Serializable {
08
09   private static final long serialVersionUID = 1L;
10   private Projeto projeto = new Projeto();
11   private List<String> listaSituacao = new ArrayList<>();
12
13   @Inject
14   private ProjetoService projetoService;
15
16   public void salvar() {
17
18     projetoService.salvarProjeto(this.projeto);
19     this.init();
20     retornarMsgSucesso("Projeto Cadastrado com sucesso");
21   }
22
23
24   @PostConstruct
25   public void init() {
26     projeto = new Projeto();
27     listaSituacao.add("EM DESENVOLVIMENTO");
28     listaSituacao.add("PRE-VENDA");
29     listaSituacao.add("EM ANALISE");
30     listaSituacao.add("POS-VENDA");
31   }
32
33   private void retornarMsgSucesso(String msg) {
34     FacesContext.getCurrentInstance().addMessage(null,
35       new FacesMessage(FacesMessage.SEVERITY_INFO, msg, msg));
36   }
37
38   /* Métodos getters e setters omitidos*/
39 }
```

Nas linhas 10 e 11 são instanciados um objeto do tipo **Projeto**, para armazenar as informações de projeto preenchidas na tela, e uma lista do tipo **String**, que contém várias situações para definir o status do projeto.

Já na linha 13 é realizada a injeção de dependência do EJB **ProjetoServiceBean** com a anotação **@Inject** do CDI. Esse EJB será responsável por executar as regras de negócio.

Como construir uma aplicação de Controle de Projetos com JavaEE – Parte 1

Dentre elas, cadastrar os projetos no banco de dados.

Nessa listagem também encontramos anotação `@PostConstruct`, sobre o método `init()` (vide linha 24). Com essa marcação esse método será executado assim que o bean `ProjetoBean` for instanciado. Dentro de `init()` é carregada a lista de situações do projeto (`listaSituacao`), que é percorrida para popular o combo utilizado para determinar a situação do projeto (em desenvolvimento, pré-venda, em análise e pós-venda).

Já o método `salvar()`, presente nas linhas 16 a 22, é executado ao clicar no botão `Salvar` da página de cadastro. Dentro dele é invocado o método `salvarProjeto()` do EJB `ProjetoServiceBean` passando o objeto `projeto` que contém as informações fornecidas na tela.

Após o retorno de `salvarProjeto()` o método `init()` é executado novamente para recarregar o combo que contém as opções relacionadas à situação do projeto. Por fim, o método `retornarMsgSucesso()` mostra na tela uma mensagem de sucesso.

Esse método, codificado entre linhas 32 e 35, passa a mensagem recebida como

parâmetro para o objeto `FacesMessage` como uma mensagem do tipo `FacesMessage.SEVERITY_INFO`, ou seja, como uma mensagem informativa. Com o objeto `FacesMessage` instanciado é necessário adicioná-lo ao contexto do JSF, o que é feito por meio do objeto `FacesContext`, que representa o contexto de execução do JSF. Deste modo as mensagens podem ser renderizadas na tela da aplicação utilizando um componente de mensagens do JSF ou mesmo do PrimeFaces, como veremos mais à frente.

As regras de negócio relativas ao cadastro de projetos estão presentes no EJB `ProjetoServiceBean`, descrito na **Listagem 16**. Nesse código, a classe está anotada com `@Stateless`, que define a classe como um EJB, e com `@LocalBean`, que a define como um EJB local. Repare que não foi necessário implementar nenhuma interface, visto que a partir do EJB 3.1 deixa de ser obrigatório o uso de interfaces para EJBs desse tipo.

Quando precisamos realizar operações com o banco de dados dentro de algum EJB, é necessário injetar um objeto do

tipo `EntityManager` por meio da anotação `@PersistenceContext` (veja as linhas 9 e 10). Com essa anotação fazemos a injeção do contexto de persistência configurado no arquivo `persistence.xml`.

Para encerrar, na linha 13 temos o método `salvarProjeto()`, que recebe como parâmetro um objeto `projeto` a ser salvo no banco de dados através do método `persist()`.

Chegou a hora de executar a aplicação no GlassFish. Para isso, abra o prompt de comando no diretório raiz do projeto `JMControleProjeto` e execute o comando do Maven: `mvn clean install`.

Feito isso, será criado o diretório `target` com todos os artefatos compilados do projeto. Dentro desse diretório você encontrará também o arquivo `JMControleProjeto.war`, a ser utilizado para fazer deploy no servidor.

Para colocar a aplicação no servidor, acesse o console administrativo do GlassFish pelo endereço `http://localhost:4848`. Em seguida, clique na opção `Applications`, localizada do lado esquerdo da tela (veja a **Figura 2**), e depois no botão `Deploy...` e em `Escolher Arquivo`. Por fim, selecione o arquivo `JMControleProjeto.war` no diretório `target` e clique em `OK`. Após esses passos a aplicação deve ser carregada com sucesso.

Agora, acesse a página inicial através do endereço `http://localhost:8080/JMControleProjeto`. O resultado deve ser semelhante à tela exibida na **Figura 3**, que traz uma mensagem de boas-vindas ao sistema de

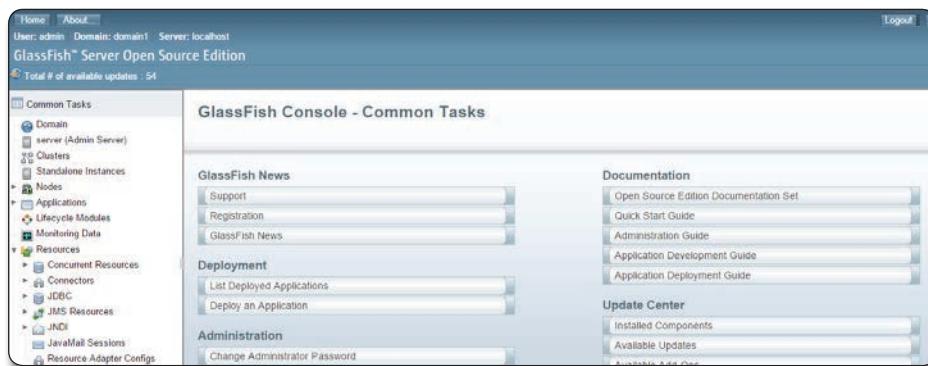


Figura 2. Console Administrativo do servidor GlassFish

A screenshot of the 'Cadastro de Projetos' (Project Registration) form. It has four input fields: 'Nome' (Name), 'Descrição' (Description), 'Orçamento' (Budget), and 'Situacao' (Situation). The 'Situacao' field has a dropdown menu with the placeholder 'Selecione a situação do projeto' (Select project status). Below the form is a 'Salvar' (Save) button. At the bottom of the page, there is a footer with the text 'Sistema de Controle de Projetos - versão 1.0.0'.

Figura 3. Tela inicial do sistema de Controle de Projetos

Listagem 16. Código fonte do EJB Projeto-ServiceBean.

```
01 package br.com.devmedia.controlprojeto.component;
02
03 //imports omitidos...
04
05 @Stateless
06 @LocalBean
07 public class ProjetoService {
08
09     @PersistenceContext
10    private EntityManager em;
11
12    @Override
13    public void salvarProjeto(Projeto projeto) {
14        em.persist(projeto);
15    }
16
17}
```

controle de projetos e o menu de opções. Como nesse primeiro artigo desenvolvemos apenas o cadastro de projetos, acesse o menu *Cadastro > Projetos*. Feito isso, dever ser exibida a tela da **Figura 4**.

Nessa tela vemos os campos que deverão ser preenchidos para o cadastro completo de um projeto, além do botão **Salvar**, que chama o bean **ProjetoBean** responsável por fornecer a funcionalidade de cadastro. Como especificado, todos esses campos são de preenchimento obrigatório e caso seja selecionado o botão *Salvar* sem informar seus respectivos valores, uma mensagem de erro deve ser apresentada, como demonstra a **Figura 5**.

Observe que todas as mensagens de erro dos campos obrigatórios ficaram concentradas no componente de mensagens do PrimeFaces **p:messages**, localizado no topo da página, como era esperado.

Nota

As mensagens que apareceram nesse componente estão definidas no parâmetro **requiredMessage** dos componentes **h:inputText** definidos na página.

Ao informar os valores corretamente para esses campos, o cadastro deve ser executado com sucesso e deve ser exibida uma mensagem conforme a exposta na **Figura 6**.

A nova versão da plataforma Java EE 7 traz novidades importantes para o desenvolvimento de aplicações corporativas, fornecendo novas especificações e novas APIs, além de inúmeras atualizações para outras especificações já existentes.

Pensando na camada de visão, para auxiliar no desenvolvimento web com Java EE 7, frameworks de interfaces ricas como o PrimeFaces podem ser uma opção interessante para incrementar as telas de uma aplicação, viabilizando um ganho muito grande de produtividade. Além desta, outras bibliotecas com o mesmo objetivo estão disponíveis no mercado, como ICEfaces e RichFaces, que possuem componentes visuais bem interessantes e que valem a pena ser estudados.

Figura 4. Tela de cadastro de Projetos

Figura 5. Validação dos campos obrigatórios na tela

Figura 6. Mensagem de cadastro realizado com sucesso

Autor



Edmar Elias Bregagnoli
edmareliasb@gmail.com.br
É Bacharel em Ciência da Computação pela FUNVIC de Mococa e Pós-graduado em Desenvolvimento de Software para Web pela UFSCAR de São Carlos. Possui MBA em Gestão de Projetos PMI pelas Faculdades Veris/IBTA de Campinas. Trabalha com Java há sete anos e atualmente é Analista de Sistemas desenvolvendo projetos na área financeira pela MATERA Systems em Campinas. Possui a certificação OCJP

Links:

Endereço para download do Java 7.
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Novas especificações da plataforma Java EE 7.
https://blogs.oracle.com/arungupta/entry/java_ee_7_key_features

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Como criar extensões CDI

Adicione flexibilidade a seus sistemas

ESTE ARTIGO É DO TIPO MENTORING

SAIBA MAIS: WWW.DEVMEDIA.COM.BR/MENTORING-SAIBAMAI

Existem situações em que é preciso que o sistema seja parametrizado para permitir que seu funcionamento seja alterado sem a necessidade de compilação ou realização de um novo deploy. Para isso, muitas vezes são criados arquivos de configuração que definem o comportamento do sistema, regras de acesso a funcionalidades, log, etc. Geralmente, esses arquivos possuem conteúdo diferente para ambientes de desenvolvimento e produção, ficando a cargo do desenvolvedor tomar o devido cuidado para não confundir os arquivos e não realizar o deploy da aplicação com o arquivo errado. Observe que, dependendo da situação, podemos diminuir a complexidade do processo de deploy criando scripts ou soluções mais dinâmicas que determinem o comportamento do sistema em tempo de execução.

Considerando uma aplicação que possua acesso a dados ou parâmetros que possam ser usados para definir o comportamento da mesma de forma dinâmica, ainda assim é necessário que tomadas de decisão sejam definidas no código fonte. Estas tomadas de decisão podem ser escritas através de simples ifs ou utilizando polimorfismo em conjunto com boas práticas de orientação a objetos. Já com o uso da injeção de dependências, simplificamos a construção de um bom design orientado a objetos, o que pode ser viabilizado ao adicionar CDI ao projeto. Através do CDI, mais precisamente a partir de extensões, é possível implementar uma solução para o cenário citado sem a necessidade de escrever vários ifs no código.

Deste modo, para exemplificar a criação de extensões CDI, este artigo apresentará uma aplicação web envolvida em um contexto que traz como importante requisito a necessidade de implementação de uma solução dinâmica que não faça uso de arquivos de configuração. Esta aplicação encontra-se disponível no repositório online Bitbucket com o nome artigo-cdi (veja o endereço na seção **Links**). O Bitbucket trata-se de um serviço de hospedagem de projetos que podem ser gerenciados através de sistemas de controle de versão como Git e Mercurial.

A aplicação criada para o artigo trata-se de uma loja de

Cenário

Este artigo visa mostrar de forma prática o uso e a criação de extensões CDI. Como exemplo, será refatorada uma simples aplicação web que deve se comportar de uma maneira diferente de acordo com as características específicas do servidor em que for distribuída. A característica que será considerada no artigo é a localização geográfica, ou região. Note que é possível atender a esse requisito sem utilizarmos o poder da injeção de dependências e da inversão de controle, porém, deste modo, podem ser criadas soluções que não fazem um bom uso da orientação a objetos, perdendo em flexibilidade, manutenibilidade, adaptabilidade e até mesmo em legibilidade. Em contrapartida, com o uso de extensões CDI conseguimos sanar o problema viabilizando uma aplicação com baixo acoplamento e de fácil manutenção.

livros virtual fictícia que deve atender a usuários em diferentes regiões geográficas, sendo necessário aplicar regras distintas para cálculo de impostos, ou seja, é de grande importância que a aplicação seja flexível. Mais detalhes sobre o seu funcionamento serão explicados no próximo tópico.

Dito isso, para conseguir alcançar a flexibilidade necessária e atender as diferentes regiões do mundo, suponha que foi sugerida a utilização da computação em nuvem, de modo que o serviço contratado ofereça meios para identificação da região através de uma API própria. Como exemplo de empresa que oferece serviços de computação em nuvem, vamos considerar a Amazon, empresa que disponibiliza servidores virtuais distribuídos em diferentes regiões geográficas. Estes servidores são oferecidos como instâncias EC2 (*Elastic Compute Cloud*). Dessa forma, o desenvolvedor pode criar sistemas de alta disponibilidade e aumentar a escalabilidade contratando mais instâncias de acordo com a demanda. Esse é um exemplo de computação em nuvem entregue como IaaS, ou *Infrastructure as a Service* (Infraestrutura como serviço), e que também é oferecido por outras empresas como a DigitalOcean e a Microsoft, com o Microsoft Azure. Muitos desses serviços oferecidos na nuvem permitem recuperar dados sobre a região da instância sendo utilizada através de API própria.

Na Amazon, por exemplo, quando se cria uma instância EC2, é necessário informar em qual região se deseja hospedar a aplicação. Para permitir esse tipo de funcionalidade, a Amazon define dois conceitos importantes, conhecidos como *Region* and *Availability Zone*. Baseando-se neles é fornecida uma API que permite ao desenvolvedor descobrir, em tempo de execução, qual região está sendo atendida. No exemplo adotado para este artigo, serão atendidas duas regiões: América do Norte e a região de São Paulo,

lembra que para atender diferentes regiões, não basta realizar apenas a internacionalização de textos. Algumas vezes é necessário também atender a requisitos específicos, como diferentes regras para cálculo de impostos, serviços de postagem, regras para compra e venda, etc.

Já na versão inicial, o sistema utiliza uma API que simula o serviço oferecido pela Amazon para recuperação dos dados da região corrente. Porém, aplica as regras específicas para cada região em tempo de execução através de `if/else`, dentro de uma mesma unidade de código (uma classe Java, por exemplo). Os problemas trazidos por essa abordagem incluem alto acoplamento, código de difícil manutenção, baixa legibilidade e alto índice de complexidade ciclomática.

Para resolver os problemas descritos serão apresentadas duas possíveis soluções utilizando CDI. A primeira delas separa as implementações das regras específicas em classes diferentes e as anota com `@Alternative`, permitindo ao desenvolvedor escolher, em tempo de desenvolvimento, qual implementação deve ser utilizada através do arquivo XML `beans.xml`, que, por sua vez, deve ser criado dentro do diretório `WEB-INF` da aplicação (por tratar-se de uma aplicação web). A partir desse arquivo é possível habilitar quais beans estarão disponíveis para injetão durante a inicialização da aplicação. Já a segunda solução demonstra uma implementação mais dinâmica, que faz uso da extensão CDI que será criada neste artigo. Com essa extensão o desenvolvedor não precisa definir no arquivo `beans.xml` quais beans devem estar disponíveis para cada região, pouparia a equipe da necessidade de manter diferentes versões do arquivo.

Descrição inicial do projeto

A loja virtual de livros utilizada como exemplo possui os seguintes requisitos:

- O usuário pode selecionar um ou mais livros para compra;
- O usuário deve confirmar um pedido após solicitar a compra de livros;
- O sistema deve apresentar o valor total do pedido antes que o usuário confirme a compra;
- O valor total do pedido é calculado somando-se o valor dos itens mais o valor do imposto, que pode variar de acordo com a região atendida. Para o Oeste dos EUA, será cobrada uma taxa de 4% e para São Paulo, será cobrada uma taxa de 15%. Todos esses detalhes da aplicação podem ser verificados no código fonte do projeto.

Em cada etapa da evolução da aplicação, o problema da região será resolvido de uma forma diferente, permitindo que o leitor perceba as vantagens da utilização de extensões CDI. Para que o leitor possa baixar os exemplos considerando cada cenário ou solução, foram criadas tags através do Git (disponíveis no Bitbucket) específicas para cada abordagem, como pode ser visto na Tabela 1.

O container web utilizado para executar a aplicação foi o Tomcat versão 7. A Figura 1 exibe uma visão macro do fluxo básico de navegação da aplicação, onde o usuário seleciona dois livros e, ao confirmar a compra, é levado para a segunda tela, que contém o

valor total do pedido com os impostos devidamente calculados. Neste exemplo, ao comprar dois livros, nos valores de 50,00 e 158,27, o usuário terá que pagar 208,27 mais 4% referente a impostos, totalizando o valor de 216,6008. Note que não utilizamos unidade monetária para simplificar os exemplos de código disponíveis no artigo.



Figura 1. Compra de um produto mostrando o cálculo do valor total mais impostos

| Tag | Descrição |
|--------------------|---|
| versao-simples | Abordagem utilizando estruturas simples de controle (<code>if/else</code>). |
| versao-alternative | Solução utilizando <code>@Alternative</code> para injetar a calculadora correta para cada região. A implementação da calculadora a ser utilizada é definida no arquivo <code>beans.xml</code> . |
| versao-extensions | Solução utilizando extensões CDI. |

Tabela 1. Abordagens utilizadas para adicionar flexibilidade ao sistema.

O código fonte do projeto possui a estrutura de diretórios apresentada na Figura 2. Como se trata de uma aplicação web configurada com o Maven, esta estrutura segue algumas convenções, de modo que os arquivos JSP devem ficar na pasta `src/main/java/webapp` e as classes Java na pasta `src/main/java`. Além disso, as classes do projeto foram agrupadas nos pacotes `model`, `web` e `region`, conforme descrito com mais detalhes a seguir:

- **model:** define as classes de domínio que possuem as regras de negócios;
- **web:** agrupa os servlets que controlam a navegação nas telas do sistema e utilizam as funcionalidades fornecidas pelo model;
- **region:** esse pacote possui as classes que simulam o serviço de identificação da região onde a aplicação está sendo executada.

Solução inicial e um design de código a ser melhorado

A primeira tela da aplicação exibe os livros disponíveis para compra na livraria fictícia. Essa tela é gerada a partir do acesso ao servlet `IndexServlet`, mapeado como `/index`, que recupera a lista de livros para venda a partir de um repositório (neste caso, uma instância de `RepositorioDeLivros`) e realiza um `forward` para o arquivo `livros.jsp`, conforme a linha 10 da Listagem 1. O arquivo `livros.jsp`, cujo código fonte encontra-se na Listagem 2, possui o código HTML necessário para apresentar a lista de livros e o botão comprar.

A Figura 3 demonstra como deve ficar a primeira tela, com os livros disponíveis para compra. Ao pressionar o botão comprar será realizado um `submit` com os dados do formulário, ou seja,

Como criar extensões CDI

com os códigos que identificam os livros selecionados. Os dados do formulário serão recebidos pelo servlet **CheckoutServlet**, que é responsável por criar um pedido e adicionar os livros selecionados. A linha 16 da **Listagem 2** mostra a definição dos atributos **method** e **action** do formulário com os valores **POST** e **/checkout**, respectivamente. A **Figura 4** exibe as classes utilizadas por **CheckoutServlet**, cujo código é exposto na **Listagem 3**.

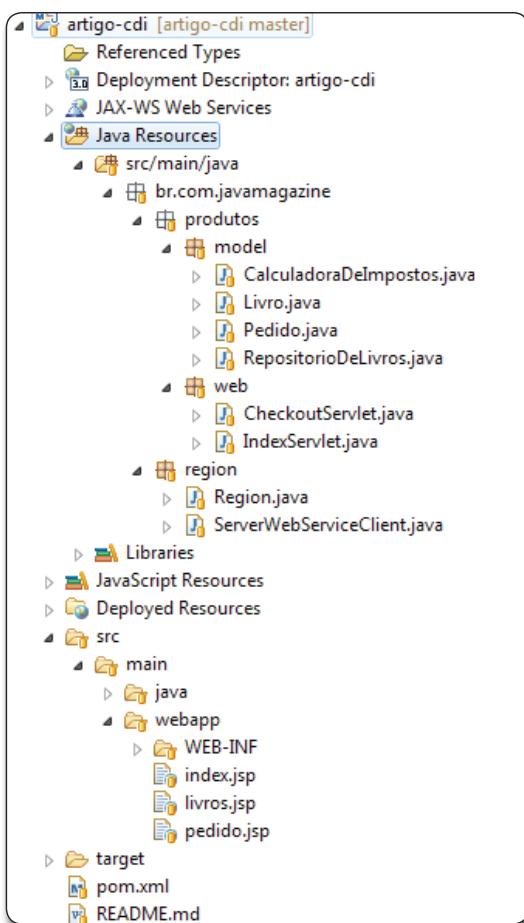


Figura 2. Estrutura de pacotes e diretórios do projeto

Listagem 1. Servlet que recupera os livros a serem exibidos na página inicial.

```
01 @WebServlet("/index")
02 public class IndexServlet extends HttpServlet {
03     private static final long serialVersionUID = 1L;
04
05     private RepositorioDeLivros repositorio = new RepositorioDeLivros();
06
07     protected void doGet(HttpServletRequest request,
08             HttpServletResponse response)
09             throws ServletException, IOException {
10
11     request.setAttribute("livros", repositorio.todosLivros());
12     request.getRequestDispatcher("/livros.jsp").forward(request, response);
13 }
14
15 }
```

Livraria

- Building Microservices - 96.00
- Java EE Patterns - 50.00
- OCM Java EE 6 - 158.27

Comprar

Figura 3. Tela inicial com os livros disponíveis para compra

Listagem 2. Tela inicial da aplicação. Exibe a lista de livros preenchida a partir de **IndexServlet**.

```
01 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
02 pageEncoding="ISO-8859-1"%>
03 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
04
05 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
06 "http://www.w3.org/TR/html4/loose.dtd">
07 <html>
08 <head>
09 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
10 <title>Livros</title>
11 </head>
12 <body>
13 <h1>Livraria</h1>
14 <hr/>
15
16 <form action="" method="post">
17 <ul>
18 <c:forEach items="${livros}" var="livro">
19 <li><input type="checkbox" name="livro" value="${livro.id}">
20 ${livro.nome}
21 ${livro.preco}</li>
22 </c:forEach>
23 <button type="submit">Comprar</button>
24 </form>
25
26 </body>
27 </html>
```

Listagem 3. Servlet para finalização do pedido de compra de um livro.

```
01 @WebServlet("/checkout")
02 public class CheckoutServlet extends HttpServlet {
03     private static final long serialVersionUID = 1L;
04
05     private RepositorioDeLivros repositorio = new RepositorioDeLivros();
06
07     protected void doPost(HttpServletRequest request,
08             HttpServletResponse response)
09             throws ServletException, IOException {
10
11     String[] ids = request.getParameterValues("livro");
12     Pedido pedido = new Pedido(new CalculadoraDeImpostos());
13
14     for (String idLivroSelecionado : ids) {
15
16         try {
17             pedido.adicionar(repositorio.getLivro(idLivroSelecionado));
18         } catch (Exception e) {
19             System.out.println("Enviado um livro que nao existe no banco de dados");
20         }
21     }
22
23     request.setAttribute("pedido", pedido);
24     request.getRequestDispatcher("/pedido.jsp").forward(request, response);
25 }
26
27 }
```

Os códigos dos livros recebidos pelo Servlet **CheckoutServlet** são utilizados como parâmetro para a busca dos livros no repositório definido pela classe **RepositorioDeLivros**, que tem o código fonte apresentado na [Listagem 4](#).

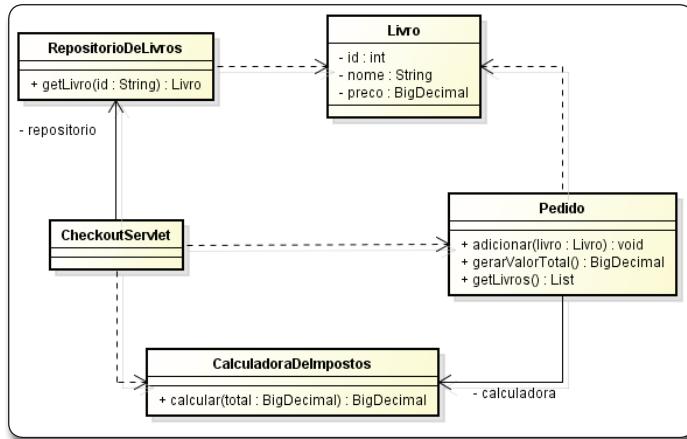


Figura 4. Classes utilizadas pela servlet CheckoutServlet

Listagem 4. Re却itorio de livros com dados estáticos.

```

01 // Simula a implementação de um re却itorio.
02 // Não será utilizado um banco de dados para simplificar o exemplo
03 public class Re却itorioDeLivros {
04
05     private static final List<Livro> livros = new ArrayList<>();
06
07     static {
08
09         livros.add(new Livro(1, "Building Microservices", new BigDecimal("96.00")));
10        livros.add(new Livro(2, "Java EE Patterns", new BigDecimal("50.00")));
11        livros.add(new Livro(3, "OCM Java EE 6", new BigDecimal("158.27")));
12
13    }
14
15    public List<Livro> todosLivros() {
16        return livros;
17    }
18
19    public Livro getLivro(String id) throws Exception {
20
21        for (Livro livro : todosLivros()) {
22            if (livro.getId() == Integer.parseInt(id)) {
23                return livro;
24            }
25        }
26
27        throw new Exception("Livro não encontrado");
28    }
29}
  
```

Este re却itorio permite obter uma lista de livros pré-definidos com valores fixos. Em um código de produção, esse re却itorio poderia recuperar os dados de um banco de dados ou a partir de um web service. Para representar um livro foi criada a classe **Livro** (veja a [Listagem 5](#)). Esta classe possui como atributos: código de identificação, nome e preço. Uma vez recuperados do re却itorio, os livros são adicionados em um **Pedido**, como indica a linha 16 da [Listagem 3](#).

Listagem 5. Bean para representação dos livros.

```

01 public class Livro {
02     private int id;
03     private String nome;
04     private BigDecimal preco;
05
06     public Livro(int id, String nome, BigDecimal preco) {
07         this.id = id;
08         this.nome = nome;
09         this.preco = preco;
09     }
10
11     // getters e setters omitidos para facilitar a leitura
11 }
  
```

Como podemos observar na linha 11, para que o pedido seja criado, é necessário passar uma calculadora do tipo **CalculadoraDeImpostos** para o método construtor de **Pedido**. A calculadora informada é responsável por realizar o cálculo do valor total do pedido aplicando a devida taxa de imposto, determinada dinamicamente de acordo com a região sendo atendida.

Já a [Listagem 6](#) apresenta o código da classe **CalculadoraDeImpostos**. Após a criação do pedido, o mesmo é associado a um atributo da requisição para que possa ser utilizado na JSP *pedido.jsp*, que é responsável por exibir a situação final do pedido e solicitar a confirmação por parte do usuário. O código fonte para a página *pedido.jsp* encontra-se na [Listagem 7](#) e, ao ser renderizada, deve ficar semelhante à [Figura 5](#).



Figura 5. Tela de confirmação do pedido com o valor da compra já calculado

A classe **Pedido**, mencionada anteriormente e cujo código é apresentado na [Listagem 8](#), possui o método **getValorTotal()**. Este retorna a soma dos preços de todos os livros com o valor do imposto devidamente calculado por **CalculadoraDeImpostos** logo após invocar o método **calcular()**.

Observando o funcionamento do sistema, nota-se que os requisitos funcionais explicados na descrição do projeto são atendidos, ou seja, o usuário seleciona livros, efetua uma compra e recebe uma tela para confirmação do pedido com o valor total calculado considerando as devidas taxas de impostos. Porém, estudando o código da classe **CalculadoraDeImpostos**, é possível verificar alguns problemas de design que podem não atender a determinados requisitos não funcionais importantes, a saber:

Como criar extensões CDI

manutenibilidade, extensibilidade, facilidade para distribuição/*deployment* e legibilidade.

Listagem 6. Classe com a regra de cálculo de imposto sobre o valor total.

```
01 public class CalculadoraDeImpostos {  
02     public BigDecimal calcular(BigDecimal valorTotal) {  
03         // guarda o valor do percentual  
04         BigDecimal percentual = BigDecimal.ZERO;  
05         // recupera a regiao através da api fornecida pelo servidor na nuvem.  
06         ServerWebServiceClient swsClient = new ServerWebServiceClient();  
07         Region bucketLocation = swsClient.getBucketLocation();  
08         // imposto de 15% para a região de sao paulo e 4% para US_West  
09         if(bucketLocation.equals(Region.SA_SaoPaulo)) {  
10             percentual = new BigDecimal("0.15");  
11         } else if(bucketLocation.equals(Region.US_West)) {  
12             percentual = new BigDecimal("0.04");  
13         }  
14         return valorTotal.add(valorTotal.multiply(percentual));  
15     }  
16 }
```

Listagem 7. JSP que renderiza a tela de confirmação do pedido.

```
01 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"  
02  pageEncoding="ISO-8859-1"%>  
03 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>  
04 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
"http://www.w3.org/TR/html4/loose.dtd">  
05 <html>  
06 <head>  
07     <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">  
08     <title>Livros - pedido</title>  
09 </head>  
10 <body>  
11     <h1>Livraria</h1>  
12     <hr/>  
13     <form action="" method="post">  
14         <h3>Produtos Selecionados</h3>  
15         <ul>  
16             <c:forEach items="${pedido.livros}" var="livro">  
17                 <li>${livro.nome} - ${livro.preco}</li>  
18             </c:forEach>  
19         </ul>  
20         <h3>Valor Total do Pedido</h3>  
21         <label>${pedido.valorTotal}</label>  
22         <hr/>  
23         <button type="submit">Finalizar Pedido</button>  
24     </form>  
25 </body>  
26 </html>
```

Dentre os fatores que levam a um design pouco flexível, podemos considerar o alto acoplamento entre as classes e a baixa coesão. A linha 11 da **Listagem 3**, por exemplo, mostra que uma instância da classe **CalculadoraDeImpostos** está sendo criada manualmente, ou seja, a classe **CheckoutServlet** e a classe **Pedido** tornam-se dependentes da implementação de **CalculadoraDeImpostos**.

Uma das maneiras de diminuir o acoplamento é projetar um design orientado a interfaces trazendo, assim, maior flexibilidade e extensibilidade para a calculadora. Além do problema de acoplamento, a regra para decidir qual valor de imposto deve ser utilizado está definida na classe **CalculadoraDeImpostos**, tornando o código pouco flexível e violando o princípio da responsabilidade única, pois o método realiza cálculos diferentes para cada região.

Vejamos o código da calculadora de impostos. Na linha 10 da **Listagem 6**, o método **calcular()** recupera a região da instância do servidor corrente a partir do método **getBucketLocation()** da classe **ServerWebServiceClient**, cujo código fonte encontra-se na **Listagem 9**.

Listagem 8. Bean com responsabilidades relacionadas ao pedido.

```
01 public class Pedido {  
02     private List<Livro> livros = new ArrayList<>();  
03     private CalculadoraDeImpostos calculadora;  
04     public Pedido(CalculadoraDeImpostos calculadora) {  
05         this.calculadora = calculadora;  
06     }  
07     public void adicionar(Livro livro) {  
08         livros.add(livro);  
09     }  
10     public List<Livro> getLivros() {  
11         return Collections.unmodifiableList(livros);  
12     }  
13     public BigDecimal getValorTotal() {  
14         BigDecimal valorTotal = BigDecimal.ZERO;  
15         for (Livro livro : livros) {  
16             valorTotal = valorTotal.add(livro.getPreco());  
17         }  
18         return calculadora.calcular(valorTotal);  
19     }  
20 }
```

Listagem 9. Classe que recupera a localização do nosso servidor.

```
01 public class ServerWebServiceClient {  
02     public Region getBucketLocation() {  
03         return Region.US_West;  
04     }  
05 }
```

Esta classe foi criada para simular um serviço fornecido pela Amazon e retorna apenas uma região, porém um serviço real deve retornar uma região de forma dinâmica. As regiões disponíveis para o nosso exemplo estão definidas no enum **Region**, que possui as opções **SA_SaoPaulo** e **US_West**, como indica a **Listagem 10**.

Para determinar o valor de imposto a ser utilizado foi criado um **if** na linha 13 da **Listagem 6** que trata cada caso de forma independente. Deste modo, se for necessário adicionar suporte a mais uma região, o desenvolvedor será obrigado a adicionar mais um elemento no enum **Region** e adicionar mais um **if/else** no código da calculadora.

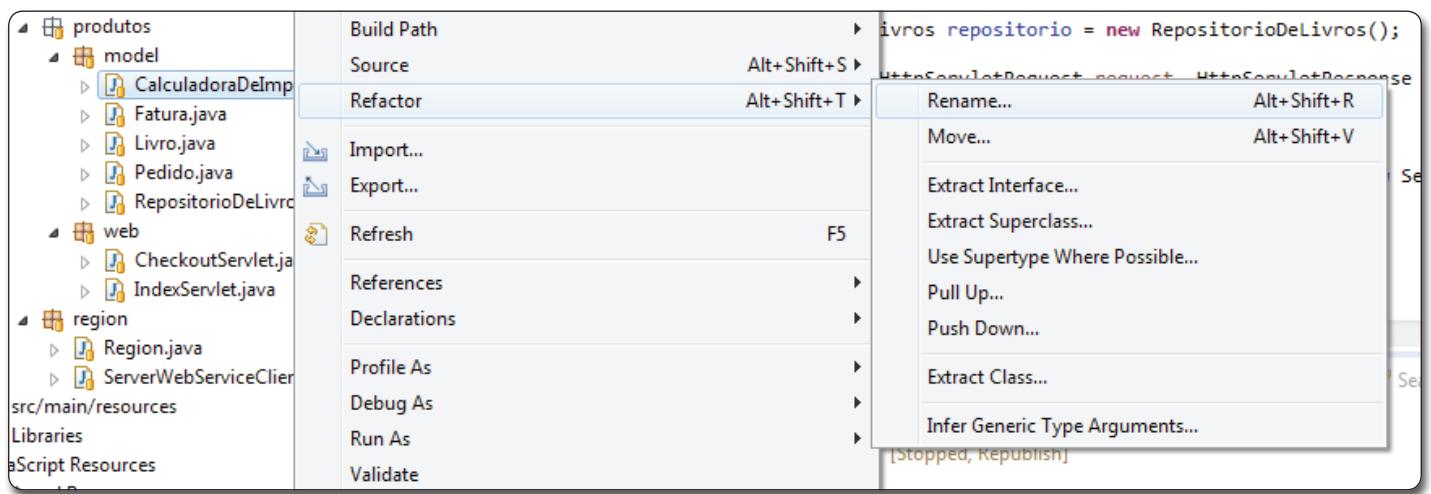


Figura 6. Refatoração para renomear a classe CalculadoraDeImpostos

Listagem 10. Enum com as regiões fornecidas pelo servidor fictício.

```
01 public enum Region {
02
03     SA_SaoPaulo,
04     US_West
05
06 }
```

Refatoração do código e um breve histórico sobre CDI

Analisando a solução apresentada até aqui, pode-se identificar uma série de problemas, conforme os pontos enumerados a seguir:

1. Alto acoplamento entre a classe **CalculadoraDeImpostos** e as classes **CheckoutServlet** e **Pedido**;
2. Violação do princípio da responsabilidade única dentro do método **calcular()**, da classe **CalculadoraDeImpostos**;
3. Alta complexidade na classe **CalculadoraDeImpostos**, com a utilização de **if/else** para tratar cada região.

Sendo assim, inicialmente vamos extrair uma interface e criar implementações específicas para atender as regras de cada região.

A refatoração do código foi realizada através de alguns recursos da IDE Eclipse, versão Luna. Antes de extrair uma interface, no entanto, vamos alterar o nome da calculadora atual para **CalculadoraDeImpostosPadrao**. Para isso, primeiro clique com o botão direito na classe **CalculadoraDeImpostos**, selecione a opção **Refactor** e, em seguida, selecione **Rename**, conforme a **Figura 6**.

Ao selecionar a opção **Rename**, será apresentada a caixa de diálogo exposta na **Figura 7**. Nela, deve-se informar o novo nome para a classe, neste caso: **CalculadoraDeImpostosPadrao**.

Agora que a classe foi renomeada, podemos extrair uma interface a partir de **CalculadoraDeImpostosPadrao**. Para tanto, clique com o botão direito na mesma e selecione a opção **Extract Interface**. Logo após, selecione a opção **Refactor** (vide **Figura 8**).

Ao selecionar a opção **Extract Interface** será exibida uma caixa de diálogo solicitando o nome da interface a ser criada, como demonstra a **Figura 9**. Digite o nome da interface (**Calculadora-**

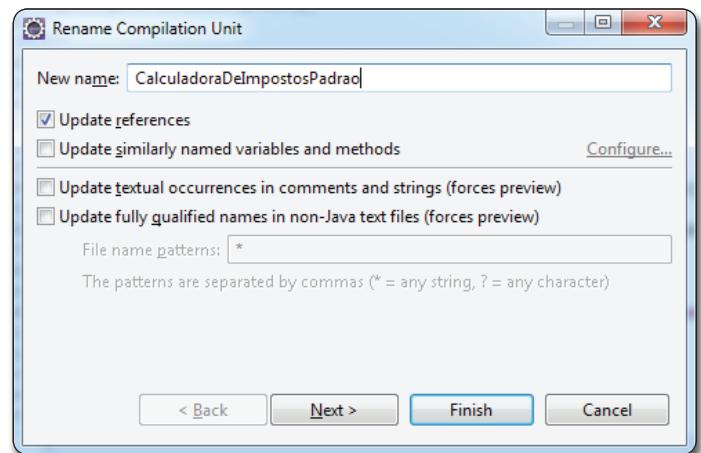


Figura 7. Renomeando a classe para CalculadoraDeImpostosPadrao

DeImpostos) e, em *Members to declare in the interface*, marque o método **calcular(BigDecimal)**.

Ao utilizar a interface me vez da implementação já é possível minimizar o acoplamento entre a classe **Pedido** e a implementação de **CalculadoraDeImpostos**. Portanto, após realizar estas refatorações, a classe **Pedido** passa a referenciar uma calculadora através da interface e a classe **CheckoutServlet** terá sido alterada para criar uma instância de **CalculadoraDeImpostosPadrao** a ser passada como parâmetro para o método construtor de **Pedido**. Ao extrair uma interface para a calculadora, as classes serão modeladas conforme a estrutura apresentada no diagrama de classes da **Figura 10**.

Como podemos observar no trecho de código extraído da classe **CheckoutServlet**, exibido na **Listagem 11**, a calculadora de impostos está sendo criada através do operador **new**, ou seja, ainda existe um forte acoplamento entre **CheckoutServlet** com a implementação da calculadora de impostos. Para realmente diminuir o acoplamento entre **CheckoutServlet** e a calculadora, serão apresentados os conceitos de injeção de dependências e inversão de controle de modo que a referência para a calculadora seja realizada através de interface.

Como criar extensões CDI

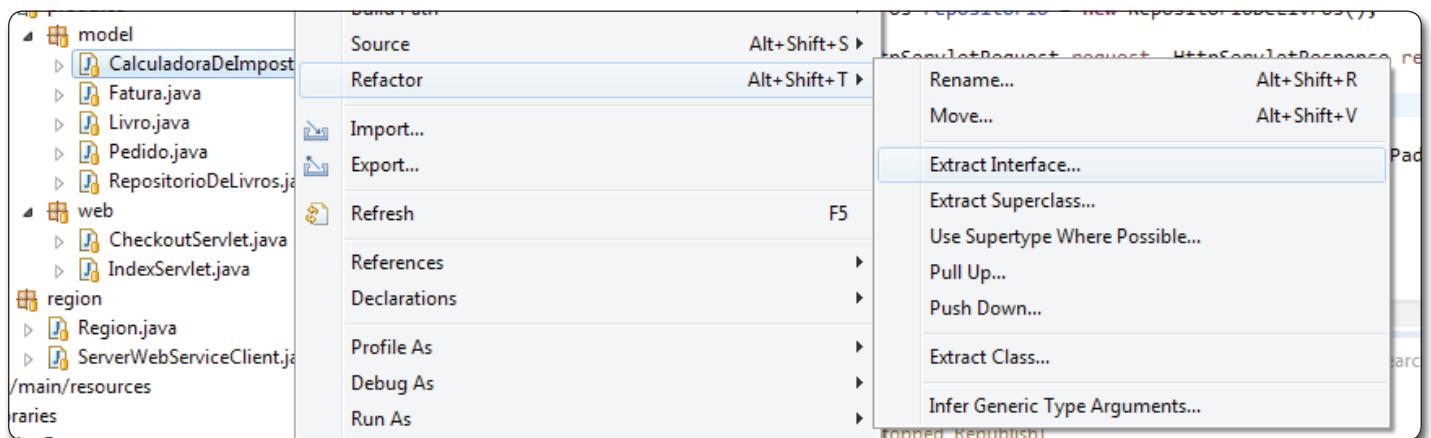


Figura 8. Opção de refatoração para extração de interface

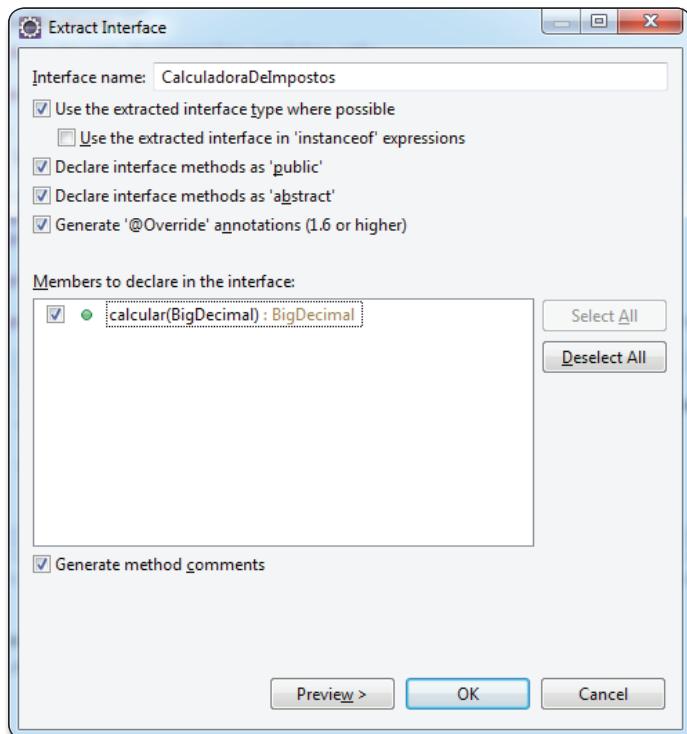


Figura 9. Caixa de diálogo solicitando o nome da interface a ser extraída

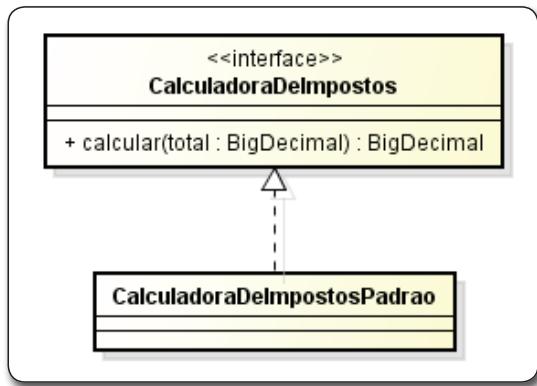


Figura 10. Refatoração da calculadora de impostos

Listagem 11. Trecho de código de CheckoutServlet.

```
01 protected void doPost(HttpServletRequest request, HttpServletResponse response)
02     throws ServletException, IOException {
03
04     String[] ids = request.getParameterValues("livro");
05     Pedido pedido = new Pedido(new CalculadoraDeImpostosPadrao());
```

Injeção de Dependências

O conceito de Injeção de Dependências permite desacoplar a responsabilidade de um determinado objeto de ter que criar as dependências das quais o mesmo necessita para funcionar. Assim, em vez de uma classe cliente utilizar o operador `new` para criar os objetos que precisa, a mesma pode simplesmente recebê-los através de algum método ou construtor. Os objetos que são recebidos via injeção devem, preferencialmente, ser referenciados através de interfaces, para minimizar o acoplamento no código e aumentar a flexibilidade do sistema.

Como exemplo de utilização de interface em vez da implementação diretamente, a classe `Pedido` faz referência à implementação da calculadora através da interface `CalculadoraDeImpostos`. Porém, ao observar a linha 5 da **Listagem 11**, nota-se que `CheckoutServlet` tem a responsabilidade de criar uma instância de `CalculadoraDeImpostosPadrao` e realizar a injeção em `Pedido`. Deste modo, apesar de a classe `Pedido` referenciar calculadora através da interface, ainda existe um forte acoplamento entre `CheckoutServlet` e `CalculadoraDeImpostosPadrao`. Por fim, esse acoplamento também pode ser reduzido ao transferir a responsabilidade de criar uma instância de calculadora para o container ou framework. A inversão da responsabilidade de criar objetos e injetar dependências pode ser concretizada ao adotar o conceito de inversão de controle com o uso de CDI.

Inversão de Controle

Um artigo publicado em 1988, na revista científica *"Journal Of Object Oriented Programming"*, cujo título é *Designing Reusable Classes*, é provavelmente o primeiro a citar o conceito de inversão de controle. Nesse artigo, os autores Ralph Johnson e Brian Foote

falam sobre frameworks como uma maneira de reutilizar código. Através de frameworks é possível fornecer funcionalidades comuns que se apliquem a um determinado contexto, de modo que o desenvolvedor precisa implementar apenas o que não é disponibilizado pelo framework. Desta forma, o framework funciona como um template que executa o código escrito pelo programador, similar ao padrão de projeto *Template Method*, descrito pelo GoF (*Gang of Four*).

Ao adotar esse padrão, o desenvolvedor geralmente estende uma classe que define um template de código e implementa um ou mais métodos que são invocados dentro do template. Ao fazer isso, o desenvolvedor não precisa criar chamadas para os métodos que ele implementou. Em vez disso, o próprio template realiza as chamadas. Devido a essa inversão de responsabilidades sobre quem faz as chamadas ao código implementado é que surge o conceito de Inversão de Controle.

Mas o que a Inversão de Controle tem em comum com a Injeção de Dependências? Graças à inversão de controle, o programador não precisa criar as dependências e nem mesmo realizar a injeção manualmente. Toda responsabilidade de criação e injeção fica a cargo do próprio container ou framework utilizado. Alguns frameworks famosos, como o Spring, realizam a injeção de dependências através da inversão de controle de forma declarativa, seja via XML ou anotações.

Considerando a loja fictícia que criamos para o artigo, vamos melhorar ainda mais o uso da **CalculadoraDeImpostos** deixando a realização da injeção de dependências a cargo do CDI.

A especificação CDI

A especificação CDI foi definida inicialmente através da JSR 299, para permitir que algumas ideias que vinham sendo trabalhadas no framework JBoss Seam fossem levadas para dentro da plataforma Java EE 6.

Segundo uma entrevista dada por Gavin King (criador do projeto JBoss Seam), para a equipe do DZone, originalmente os problemas que a equipe que trabalhou na JSR 299 gostaria de resolver estavam relacionados em como integrar facilmente aplicativos que usavam JSF com tecnologias como EJB. Inicialmente, a especificação havia sido nomeada como *Web Beans*, devido aos propósitos mais focados na web, porém, com a evolução das ideias e percebendo que a tecnologia ia muito além da web, o nome foi trocado para *Context and Dependency Injection* (CDI). A JSR 299 especifica como frameworks de injeção de dependência devem funcionar dentro da plataforma Java como um todo, ou seja, é possível utilizar CDI tanto em aplicações Java EE como Java SE.

Além da injeção de dependências, a especificação CDI também oferece suporte à criação de escopos customizados, *interceptors*, *decorators*, notificação por eventos e extensibilidade, e a partir de todos esses recursos é possível melhorar a integração entre os elementos que compõem a plataforma Java EE, como EJB, JSF, JPA, JMS, JSP, Servlet, entre outros.

Uma grande diferença entre CDI e as demais especificações presentes na plataforma Java EE é que a CDI, através da criação

de extensões, permite aumentar os recursos da plataforma. Como exemplo de implementações da especificação temos a OpenWebBeans da Apache e o Weld, da JBoss, sendo esta última a implementação de referência.

Como o assunto CDI é muito abrangente para ser detalhado em um único artigo, uma boa introdução a respeito pode ser encontrada na edição 116 da revista Java Magazine. Nesse artigo, nosso foco será mantido em como criar extensões CDI.

Aprimorando o design do código com @Alternative

Voltando ao projeto da loja de livros, vamos utilizar injeção de dependências para favorecer o baixo acoplamento da calculadora com o servlet **CheckoutServlet**. Para suportar a injeção de dependências e a inversão de controle, foram adicionadas algumas bibliotecas ao projeto (entre elas o Weld).

Como o aplicativo da loja de livros foi criado utilizando o Maven, as dependências foram declaradas no arquivo *pom.xml*, que se encontra no diretório raiz do projeto. A **Listagem 12** mostra apenas o trecho deste arquivo com as declarações das dependências. Além de adicionar as dependências no *pom.xml*, é necessário garantir que o arquivo *beans.xml* tenha sido criado dentro do diretório *WEB-INF* do projeto, pois trata-se de uma exigência da especificação CDI. Apesar disso, não é necessário que o *beans.xml* tenha conteúdo nesse momento. O código fonte com a estrutura básica deste arquivo pode ser visualizado na **Listagem 13**.

Listagem 12. Dependências necessárias para utilização do CDI.

```
01 <dependencies>
02   <dependency>
03     <groupId>javax.enterprise</groupId>
04     <artifactId>cdi-api</artifactId>
05     <version>1.2</version>
06   </dependency>
07   <dependency>
08     <groupId>org.jboss.weld.servlet</groupId>
09     <artifactId>weld-servlet</artifactId>
10    <version>2.2.0.Final</version>
11  </dependency>
12  <dependency>
13    <groupId>javax.servlet</groupId>
14    <artifactId>javax.servlet-api</artifactId>
15    <version>3.1.0</version>
16  </dependency>
17  <dependency>
18    <groupId>jstl</groupId>
19    <artifactId>jstl</artifactId>
20    <version>1.2</version>
21  </dependency>
22 </dependencies>
```

Listagem 13. Exemplo de estrutura básica para o arquivo *beans.xml*

```
01 <?xml version="1.0"?>
02 <beans xmlns="http://java.sun.com/xml/ns/javaee"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                        http://jboss.org/schema/cdi/beans_1_0.xsd">
05 </beans>
```

Como criar extensões CDI

Com o projeto devidamente configurado para utilizar CDI, vamos alterar o servlet **CheckoutServlet** conforme o código da **Listagem 14** para que o mesmo não precise criar uma instância da calculadora e nem do repositório, e sim que as dependências sejam injetadas automaticamente pelo container. As linhas 5 e 8 demonstram o uso da anotação **@Inject** para que sejam injetadas as instâncias de **RepositorioDeLivros** e **CalculadoraDeImpostos**, respectivamente. Esta anotação indica o ponto de injeção onde o container deve de fato adicionar as dependências. O CDI permite a utilização de três pontos de injeção, a saber:

- Injeção via construtor;
- Injeção via método;
- Injeção via propriedade.

A utilização da injeção por construtor e por método oferecem a vantagem de tornar a classe mais fácil de ser testada através de ferramentas como JUnit ou TestNG, pois durante a escrita de um teste unitário é possível criar e injetar manualmente uma instância fake ou mock com comportamentos pré-definidos para o teste.

Ao utilizar a injeção de dependência via propriedades, a escrita de testes unitários pode ser comprometida, pois pode ser necessário configurar atributos de uma dependência antes de realizar a injeção na classe a ser testada.

Listagem 14. CheckoutServlet utilizando @Inject com CDI.

```
01 @WebServlet("/checkout")
02 public class CheckoutServlet extends HttpServlet {
03     private static final long serialVersionUID = 1L;
04
05     @Inject
06     private RepositorioDeLivros repositorio;
07
08     @Inject
09     private CalculadoraDeImpostos calculadora;
10
11     protected void doPost(HttpServletRequest request,
12                           HttpServletResponse response)
13             throws ServletException, IOException {
14
15         String[] ids = request.getParameterValues("livro");
16         Pedido pedido = new Pedido(calculadora);
17
18         for (String idLivroSelecionado : ids) {
19
20             try {
21                 pedido.adicionar(repositorio.getLivro(idLivroSelecionado));
22             } catch (Exception e) {
23                 System.out.println("Enviado um livro que não existe no banco de dados");
24             }
25         }
26
27         request.setAttribute("pedido", pedido);
28         request.getRequestDispatcher("/pedido.jsp").forward(request, response);
29     }
30
31 }
```

A classe **CalculadoraDeImpostosPadrao** também possui uma dependência com a classe **ServiceWebServerClient**, utilizada para identificar a região em que o sistema está executando. Sendo assim, vamos alterar a classe **CalculadoraDeImpostosPadrao** para receber um **ServiceWebServerClient** através de injeção de dependência pelo método construtor, conforme expõe a **Listagem 15**.

Listagem 15. CalculadoraDeImpostosPadrao utilizando CDI.

```
01 public class CalculadoraDeImpostosPadrao implements CalculadoraDeImpostos {
02
03     private ServerWebServiceClient swsClient;
04
05     @Inject
06     public CalculadoraDeImpostosPadrao(ServerWebServiceClient swsClient) {
07         this.swsClient = swsClient;
08     }
09
10    @Override
11    public BigDecimal calcular(BigDecimal valorTotal) {
12
13        // guarda o valor do percentual
14        BigDecimal percentual = BigDecimal.ZERO;
15
16        // recupera a região através da API fornecida pelo servidor na nuvem.
17        Region bucketLocation = swsClient.getBucketLocation();
18
19        // imposto de 15% para a região de São Paulo e 4% para US_West
20        if (bucketLocation.equals(Region.SA_SaoPaulo)) {
21            percentual = new BigDecimal("0.15");
22            System.out.println("utilizando o imposto de sao paulo");
23        } else if (bucketLocation.equals(Region.US_West)) {
24            percentual = new BigDecimal("0.04");
25            System.out.println("utilizando o imposto dos EUA");
26        }
27
28        return valorTotal.add(valorTotal.multiply(percentual));
29    }
30
31 }
```

Deste modo, conseguimos minimizar o acoplamento entre algumas classes do sistema. Porém, ainda é preciso solucionar o problema de violação da responsabilidade única e o problema da alta complexidade.

O princípio da responsabilidade única está sendo violado porque a calculadora possui duas responsabilidades: calcular o valor do imposto para a região de São Paulo e calcular o valor do imposto para a região do Oeste dos Estados Unidos. Já o fato de codificar regras com **if/else** aumentam a complexidade ciclomática do código.

Para resolver estes problemas vamos separar as lógicas de cálculo do imposto para São Paulo e para o Oeste dos Estados Unidos criando duas classes, de acordo com o diagrama de classes da **Figura 11**.

Deste modo, criamos as classes **CalculadoraDeImpostosSA** e **CalculadoraDeImpostosUS**. O código da classe **CalculadoraDeImpostosSA**, mostrado na **Listagem 16**, possui apenas uma responsabilidade: calcular a taxa de imposto sobre o valor total para a região de São Paulo. Como podemos verificar, o novo código fica bem mais simples e segue o princípio da responsabilidade única.

Por sua vez, a calculadora de impostos específica para a região dos Estados Unidos também possui apenas uma responsabilidade: calcular o imposto sobre o valor total considerando as regras da região dos Estados Unidos (veja a **Listagem 17**).

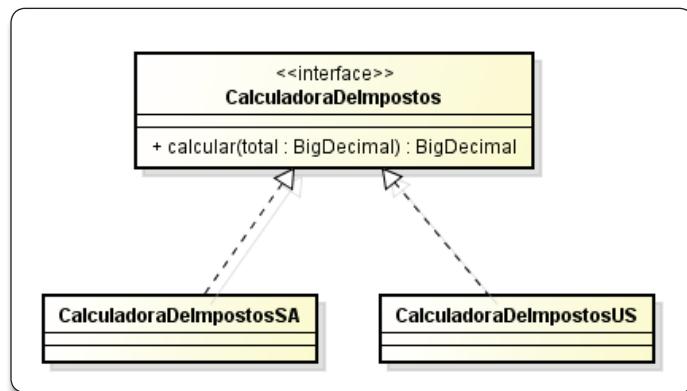


Figura 11. Diagrama de classes para o modelo da calculadora de impostos

Listagem 16. Implementação específica da calculadora de impostos para a região de São Paulo.

```

01 public class CalculadoraDeImpostosSA implements CalculadoraDeImpostos {
02
03     @Override
04     public BigDecimal calcular(BigDecimal valorTotal) {
05         System.out.println("utilizando a calculadora de sao paulo");
06         return valorTotal.add(valorTotal.multiply(new BigDecimal("0.15")));
07     }
08
09 }
  
```

Listagem 17. Implementação específica da calculadora de impostos para a região dos Estados Unidos.

```

01 public class CalculadoraDeImpostosUS implements CalculadoraDeImpostos {
02
03     @Override
04     public BigDecimal calcular(BigDecimal valorTotal) {
05         System.out.println("utilizando a calculadora dos EUA");
06         return valorTotal.add(valorTotal.multiply(new BigDecimal("0.04")));
07     }
08
09 }
  
```

Agora, com o novo modelo de classes para a calculadora, não precisamos mais da classe **CalculadoraDeImpostosPadrao**. Portanto, podemos removê-la do projeto de modo que a estrutura de classes e pacotes fique semelhante à exibida na **Figura 12**.

Ao tentar realizar o deploy do projeto no estado atual, o container exibirá uma mensagem semelhante à mostrada na **Listagem 18**, pois existe ambiguidade de classes implementando a interface **CalculadoraDeImpostos**. Isso acontece porque o container não tem condições de identificar qual bean adicionar no contexto do CDI. Observe que nas linhas 5 e 6 o container indica quais são as possíveis dependências (nesse caso, **CalculadoraDeImpostosUS** e **CalculadoraDeImpostosSA**). Para resolver esse problema é necessário que o desenvolvedor indique qual implementação utilizar

excluindo as que não interessam, o que deve ser feito através da anotação **@Alternative**.

Inicialmente, vamos permitir a injeção da calculadora de São Paulo apenas para exemplificar o uso da anotação **@Alternative**. Vamos, então, adicionar **@Alternative** na classe **CalculadoraDeImpostosUS**, conforme mostrado na **Listagem 19**, para que a mesma não seja injetada nesse momento.

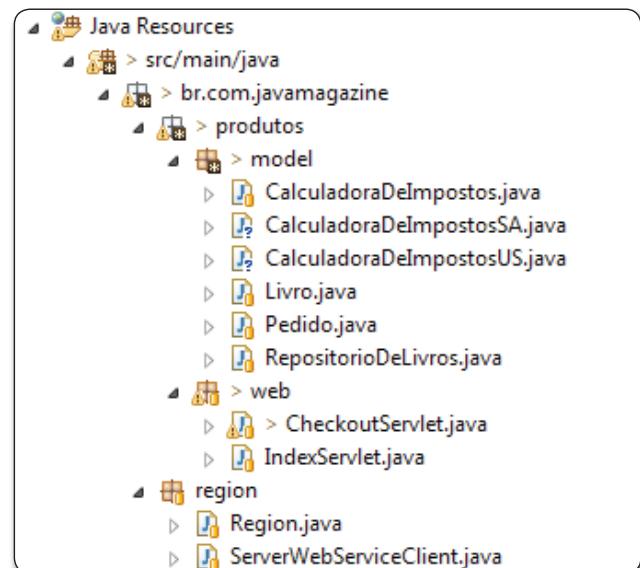


Figura 12. Estrutura de classes e pacotes após a separação das responsabilidades

Listagem 18. Log de erro gerado durante o deploy da aplicação.

```

01 Caused by: org.jboss.weld.exceptions.DeploymentException:
           WELD-001409: Ambiguous dependencies for type CalculadoraDeImpostos
           with qualifiers @Default
02 at injection point [BackedAnnotatedField] @Inject private br.com
           .javamagazine.produtos.web.CheckoutServlet.calculadora
03 at br.com.javamagazine.produtos.web.CheckoutServlet.calculadora
           (CheckoutServlet.java:0)
04 Possible dependencies:
05 - Managed Bean [class br.com.javamagazine.produtos.model.Calculadora
           DeImpostosUS] with qualifiers [@Any @Default],
06 - Managed Bean [class br.com.javamagazine.produtos.model.Calculadora
           DeImpostosSA] with qualifiers [@Any @Default]
  
```

Listagem 19. Adicionando **@Alternative** na classe **CalculadoraDeImpostosUS**.

```

01 @Alternative
02 public class CalculadoraDeImpostosUS implements CalculadoraDeImpostos {
03
04     @Override
05     public BigDecimal calcular(BigDecimal valorTotal) {
06         System.out.println("utilizando a calculadora dos EUA");
07         return valorTotal.add(valorTotal.multiply(new BigDecimal("0.04")));
08     }
09
10 }
  
```

Como criar extensões CDI

Feito isso, ao realizar o deploy do projeto novamente, a aplicação voltará a funcionar, porém irá atender apenas à região de São Paulo. Ao adicionar a anotação `@Alternative`, estamos tornando opcional a adição do bean `CalculadoraDeImpostosUS` no contexto CDI.

Além disso, também podemos tornar opcional a calculadora de impostos para São Paulo, utilizando a anotação `@Alternative` mais uma vez, conforme o código da [Listagem 20](#).

Porém, agora que as duas possíveis dependências estão anotadas com `@Alternative`, o contêiner não poderá encontrar uma dependência válida, pois ambas se tornaram opcionais. Portanto, ao realizar o deploy, o desenvolvedor ainda precisa identificar qual região será atendida. Para isso, ele deve indicar qual bean injetar através de uma entrada no arquivo `beans.xml`. A [Listagem 21](#) mostra um exemplo no qual é especificado que a aplicação deve injetar a calculadora de São Paulo onde houver uma dependência da `CalculadoraDeImpostos`.

Podemos notar que foram obtidas as seguintes vantagens:

- **Baixo acoplamento:** conseguimos minimizar o acoplamento utilizando interfaces e injeção de dependências através da inversão de controle fornecida pelo contexto do CDI. Dessa forma, as classes recebem as dependências sem a necessidade de conhecer a implementação;
- **Alta coesão:** ao separar as responsabilidades em diferentes classes foi possível construir classes que possuem uma única responsabilidade;
- **Flexibilidade:** a flexibilidade foi alcançada utilizando a anotação `@Alternative`, de modo que o desenvolvedor pode trocar a implementação a ser injetada em tempo de desenvolvimento no arquivo `beans.xml`.

Listagem 20. Adicionando `@Alternative` na classe `CalculadoraDeImpostosSA`.

```
01 @Alternative
02 public class CalculadoraDeImpostosSA implements CalculadoraDeImpostos {
03
04     @Override
05     public BigDecimal calcular(BigDecimal valorTotal) {
06         System.out.println("utilizando a calculadora de sao paulo");
07         return valorTotal.add(valorTotal.multiply(new BigDecimal("0.15")));
08     }
09
10 }
```

Listagem 21. Habilitando a classe `CalculadoraDeImpostosSA` para injeção.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
05   version="1.1" bean-discovery-mode="all">
06
07     <alternatives>
08       <class>br.com.javamagazine.produtos.model.CalculadoraDeImpostosSA
09         </class>
10     </alternatives>
11
12 </beans>
```

Apesar destas vantagens, a solução que faz uso da anotação `@Alternative` ainda traz o seguinte problema: toda vez que for realizado um deploy deve ser disponibilizado um arquivo `beans.xml` que habilita uma implementação específica para uma determinada região. O trabalho de ter que trocar os arquivos de configuração durante o deploy torna a distribuição da aplicação um processo propenso a falhas, pois é possível que



o desenvolvedor deixe de enviar os arquivos corretos para a instância apropriada.

Dante disso, uma possível solução para aumentar a flexibilidade do projeto seria a própria aplicação identificar em qual região ela está sendo executada. Para isso, o CDI fornece um mecanismo importante que permite a criação de extensões através da arquitetura *Service Provider Interface* (SPI), disponível no Java SE. Graças à SPI e ao suporte a extensões da especificação CDI, bibliotecas podem ser criadas para adicionar o suporte a transações, validações e integração com o JSF, como é o caso do projeto DeltaSpike.

O DeltaSpike é um conjunto de extensões CDI com várias funcionalidades agrupadas em módulos específicos para segurança, abstração de banco de dados, módulos para JSF, validações, scheduler e etc. Mais detalhes sobre este projeto podem ser encontrados no artigo “Apache DeltaSpike: CDI Programável”, publicado na edição 135 da revista Java Magazine. Além das extensões fornecidas pelo DeltaSpike e outros projetos, os desenvolvedores também podem criar suas próprias extensões para atender necessidades específicas de cada projeto.

Portanto, nosso próximo passo será criar uma extensão CDI que deverá identificar a região durante o processo de inicialização da aplicação, onde a mesma deve indicar quais beans devem ficar disponíveis para injecão. Dessa maneira, implementações de beans específicos para a região de São Paulo estarão habilitados para injecão quando a aplicação for distribuída em uma instância de São Paulo. O mesmo acontecerá quando a aplicação for distribuída em instâncias de outras regiões, de modo que o desenvolvedor não precisa mais ficar preocupado se adicionou um código específico ou se configurou um arquivo XML corretamente.

Criando uma extensão CDI

A especificação CDI permite a extensão de suas funcionalidades através da CDI *Service Provider Interface* (SPI). Para iniciar a criação de uma extensão, é necessário que a mesma seja declarada dentro do arquivo `javax.enterprise.inject.spi.Extension`, que deve ser criado dentro do diretório `META-INF/services`, na pasta `resources` do projeto, como mostra a **Figura 13**.

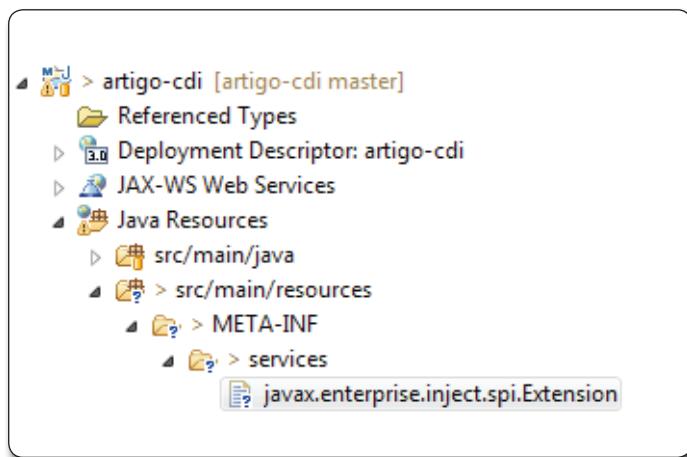


Figura 13. Estrutura de diretórios para o arquivo de definição da extensão CDI

Dentro desse arquivo precisamos declarar o nome totalmente qualificado da classe que implementa a interface `Extension`, conforme a linha apresentada a seguir:

```
br.com.javamagazine.extension.RegionExtension
```

De acordo com esse conteúdo, deve ser criada uma classe com o nome `RegionExtension` dentro do pacote `br.com.javamagazine.extension` e a mesma deve implementar a interface `javax.enterprise.inject.spi.Extension` (veja o código na **Listagem 22**). A extensão CDI `RegionExtension`, assim como qualquer outra, é carregada durante a inicialização do container e é mantida no contexto CDI por todo o ciclo de vida da aplicação.

A classe `RegionExtension`, através do método `loadClassesToBeIncludedByCDI()`, tem por objetivo observar o evento `ProcessAnnotatedType`, que é disparado para cada classe que é descoberta pelo contêiner durante o processo de varredura do projeto. Dessa forma, é possível recuperar metadados presentes na definição de beans como, por exemplo, anotações. Em nossa aplicação, a linha 6 da **Listagem 22** verifica se o bean descoberto possui a anotação `@IncludeRegion` (definida **Listagem 23**) e, posteriormente, na linha 15, é realizada uma validação através do método `unsupported(region)`, da classe `RegionValidator` (vide **Listagem 24**), para saber se este bean deve ou não ser adicionado ao contexto CDI. Para indicar que o bean não deve ser adicionado ao contexto, na linha 16 da **Listagem 22** é invocado o método `veto()`, que força o container a ignorar o bean atual.

A anotação `@IncludeRegion`, codificada na **Listagem 23**, foi criada para indicar a região suportada por determinada implementação de calculadora de impostos. Note que a mesma possui atributo `value`, que permite indicar a região através do `enum Region`. Deste modo é possível recuperar a região indicada na anotação, comparar com a região da instância do servidor sendo utilizado e decidir qual bean deve ou não ficar disponível para injecão.



Como criar extensões CDI

Listagem 22. Classe que implementa a extensão para inclusão de beans por região.

```
01 public class RegionExtension implements Extension {  
02  
03     public void loadClassesToBeIncludedByCDI  
04         (@Observes ProcessAnnotatedType<?> pat) {  
05             AnnotatedType<?> annotatedType = pat.getAnnotatedType();  
06  
07             boolean isIncludeAnnotationPresent = annotatedType.isAnnotationPresent  
08                 (IncludeRegion.class);  
09  
10             if (isIncludeAnnotationPresent) {  
11                 IncludeRegion annotation = annotatedType.getAnnotation  
12                     (IncludeRegion.class);  
13                 Region region = annotation.value();  
14  
15                 if (validator.unsupported(region)) {  
16                     pat.veto();  
17                 }  
18             }  
19         }  
20     }  
21 }  
22  
23 }
```

A classe **RegionValidator** é responsável por comparar o valor da região indicada em **@IncludeRegion** com a região da instância do servidor corrente. A região do servidor é recuperada através do método **getBucketLocation()** da classe **ServerWebServiceClient**, conforme exibido na linha 4.

A extensão definida através da classe **RegionExtension** auxiliou na criação de uma solução dinâmica que identifica se um bean deve ou não ser adicionado ao contexto CDI no momento em que os mesmos estão sendo carregados. Graças ao mecanismo

de notificação por eventos disponível na especificação CDI, é possível observar tudo o que acontece durante o ciclo de vida de um bean. No caso da extensão que criamos, o método **loadClassesToBeIncludedByCDI()** está aguardando notificações do evento **ProcessAnnotatedType**, que é recebido através do parâmetro anotado com **@Observes**, conforme mostrado na linha 3 da **Listagem 22**. Além do evento **ProcessAnnotatedType**, também existem outros eventos que podem ser observados durante o ciclo de vida de um bean.

Listagem 23. Anotação para identificação dos beans a serem adicionados ao contexto CDI.

```
01 package br.com.javamagazine.extension;  
02  
03 import java.lang.annotation.ElementType;  
04 import java.lang.annotation.Retention;  
05 import java.lang.annotation.RetentionPolicy;  
06 import java.lang.annotation.Target;  
07  
08 import br.com.javamagazine.region.Region;  
09  
10 @Retention(RetentionPolicy.RUNTIME)  
11 @Target({ElementType.TYPE, ElementType.FIELD})  
12 public @interface IncludeRegion {  
13     Region value();  
14 }
```

Listagem 24. Classe que verifica se uma região está sendo suportada pela instância do servidor corrente.

```
01 public class RegionValidator {  
02     public boolean unsupported(Region region) {  
03         ServerWebServiceClient swsClient = new ServerWebServiceClient();  
04         Region bucketLocation = swsClient.getBucketLocation();  
05         return !bucketLocation.equals(region);  
06     }  
07 }
```



Uma lista com alguns dos eventos mais usados para criar extensões CDI é exibida a seguir (a descrição de cada evento foi traduzida do Javadoc da API `javax.enterprise.inject.spi`):

- **BeforeBeanDiscovery:** Evento disparado pelo container antes que o processo de descoberta dos beans seja iniciado. Caso uma exceção seja lançada dentro de algum método que observe esse evento, um erro de definição será detectado. Erros de definição são erros relacionados à declaração de escopo, eventos, interceptors e outras possíveis configurações para identificação de um bean por parte do contexto CDI;
- **AfterBeanDiscovery:** Evento lançado pelo container quando o processo de descoberta dos beans termina de ser executado e validado. A validação do bean garante que não existem erros de definição;
- **ProcessAnnotatedType:** Evento disparado pelo container para cada classe que o mesmo descobre durante a inicialização do contexto CDI. Ao observar esse evento é possível modificar, adicionar ou decorar uma anotação;
- **ProcessProducer:** O container dispara esse evento para cada método ou atributo declarado como produtor em cada bean que tenha sido habilitado. Métodos ou atributos produtores são membros anotados com `@Produces` e atuam como fábricas para os beans. Dessa forma, beans mais complexos de serem criados podem ser instanciados dentro de métodos produtores, que serão invocados quando for solicitada a injeção de um bean que seja do mesmo tipo que o definido no retorno do método produtor. Métodos que observam o evento **ProcessProducer** podem decorar um produtor adicionando comportamento para o mesmo;
- **ProcessObserverMethod:** Dentre as funcionalidades disponíveis a partir da especificação CDI, existe também a possibilidade de notificar através de eventos. Para que um método possa ser notificado de um determinado evento, o mesmo deve ter um ou mais argumentos anotados com `@Observes`. Para que o evento seja enviado para os métodos observadores, basta utilizar o método `fire()` da classe **Event**, onde o parâmetro informado para `fire()` deve ser um objeto que pode ser recuperado pelo *observer*. Os métodos observadores, ou seja, aqueles que são notificados por eventos, podem ser acessados ou manipulados através da interface **ObserverMethod** dentro do contexto CDI. Esta interface define tudo o que o container precisa saber sobre o método observador. Quando um **ObserverMethod** é registrado no contexto CDI, o evento **ProcessObserverMethod** é disparado;
- **ProcessInjectionPoint:** Evento disparado pelo container para cada componente Java EE que suporte injeção e que pode ser instanciado pelo container, o que inclui managed beans, EJBs, message driven beans ou interceptors e decorators.

Agora que a extensão CDI foi criada, é necessário anotar as classes que implementam as regras da calculadora com `@IncludeRegion` para poder identificar qual tipo de calculadora será utilizada na região em que a aplicação for implantada. As Listagens 25 e 26 exibem o novo código fonte das classes **CalculadoraDeImpostosSA** e **CalculadoraDeImpostosUS**. Perceba que a anotação `@Alterna-`

tive

foi removida, pois agora passaremos a utilizar a solução mais dinâmica com extensões CDI.

Listagem 25. Calculadora para a região de São Paulo utilizando a extensão `@IncludeRegion`.

```
01 @IncludeRegion(Region.SA_SaoPaulo)
02 public class CalculadoraDeImpostosSA implements CalculadoraDeImpostos {
03
04     @Override
05     public BigDecimal calcular(BigDecimal valorTotal) {
06         System.out.println("Utilizando a calculadora de impostos de São Paulo");
07         BigDecimal percentual = new BigDecimal("0.15");
08         return valorTotal.add(valorTotal.multiply(percentual));
09     }
10
11 }
```

Listagem 26. Calculadora para a região dos EUA utilizando a extensão `@IncludeRegion`

```
01 @IncludeRegion(Region.US_West)
02 public class CalculadoraDeImpostosUS implements CalculadoraDeImpostos {
03
04     @Override
05     public BigDecimal calcular(BigDecimal valorTotal) {
06         System.out.println("Utilizando a calculadora de impostos dos EUA");
07         BigDecimal percentual = new BigDecimal("0.04");
08         return valorTotal.add(valorTotal.multiply(percentual));
09     }
10
11 }
```

Listagem 27. Versão final do arquivo `beans.xml` – sem as tags `<alternatives>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
</beans>
```

Antes de realizar o teste da aplicação com a extensão, remova a entrada `<alternatives>` e seu conteúdo do arquivo `beans.xml`, atualizando o mesmo para que fique de acordo com o código da Listagem 27.

Com a extensão CDI que foi criada, o desenvolvedor não precisa mais se preocupar com arquivos de configuração, pois a própria aplicação tornou-se capaz de realizar, de forma automática, a identificação da região em que está sendo distribuída. Essa dinâmica também foi alcançada porque no exemplo utilizado contamos com o auxílio de um serviço fornecido pelo servidor.

Por que a solução que utiliza a extensão CDI é melhor?

Estudando algumas formas para deixar mais dinâmica a aplicação apresentada no artigo, foi possível implementar uma extensão e conhecer mais detalhes da especificação CDI. Além dos benefícios de se utilizar a injeção de dependências de maneira

padronizada, o CDI trouxe como inovação a possibilidade de criar extensões. Ao fazer isso, o desenvolvedor não fica preso apenas ao que existe na especificação, podendo aumentar as capacidades da plataforma Java EE. Como mais um diferencial, extensões podem ser criadas de maneira portável para que sejam empregadas em qualquer implementação da especificação CDI.

A aplicação criada para o artigo, por exemplo, faz uso de uma extensão portável que pode ser utilizada tanto com Weld como com OpenWebBeans. O uso da extensão CDI implementada no artigo permite minimizar o uso de arquivos de configuração, além de favorecer um design que segue boas práticas de orientação a objetos.

No entanto, apesar da flexibilidade e facilidade para isso, antes de começar a criar extensões é importante que o desenvolvedor pesquise por extensões já existentes, como as disponíveis em projetos como o DeltaSpike. Este projeto, por sinal, já fornece uma gama enorme de extensões úteis para resolver problemas comuns a muitas soluções através de módulos individuais que facilitam o acesso a bancos de dados, construção de validações, agendamento de tarefas, segurança, implementação e execução de testes, além de adicionar funcionalidades extras ao JSF, como novos escopos, gerenciamento de exceções e mensagens e etc.

Enfim, a especificação CDI traz inúmeras possibilidades tanto para desenvolvedores quanto para a própria plataforma Java EE, o que o torna um assunto que vale a pena ser estudado, explorado e empregado em muitos projetos.

Autor



Adolfo Eloy

adolfo.eloy@gmail.com

Pós-graduado em Desenvolvimento de Soluções Corporativas com Java pela FIAP e graduado em Ciência da Computação pela FAC-FITO. Trabalha a mais de 10 anos com desenvolvimento de sistemas, tendo utilizado diversas tecnologias e linguagens como Java, PHP, JavaScript, Objective-C, Ruby, Python, Basic e C. Entusiasta de tecnologias Open Source, Agile, Orientação a Objetos, DDD e computação em geral. Atualmente trabalha como Arquiteto de Sistemas utilizando Java EE 6. Possui a certificação OCPJP 6.



Links:

Código fonte do projeto.

<https://bitbucket.org/astronauta/artigo-cdi>

Region and Availability Zones.

<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html>

Relação entre JSR 299 e 330.

http://www.adam-bien.com/roller/abien/entry/what_is_the_relation_between

Service Provider Interface.

<http://docs.oracle.com/javase/tutorial/sound/SPI-intro.html>

Página do projeto DeltaSpike.

<https://deltaspike.apache.org/index.html>

Javadoc da API de Injeção de Dependências.

<https://docs.oracle.com/javaee/6/api/javax/enterprise/inject/spi/package-summary.html>

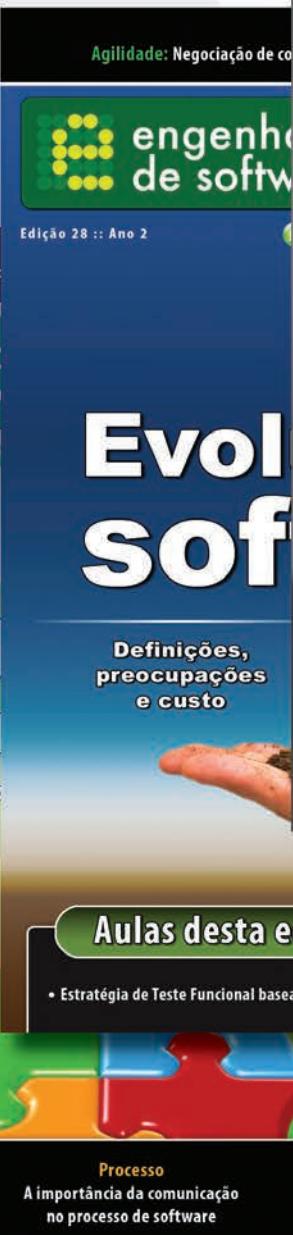
Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



**Conhecimento
faz diferença!**



+ de 290 vídeos
para assinantes

Faça já sua assinatura digital! | www.devmedia.com.br/es

Faça um upgrade em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional.

Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software.

Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

 **DEV MEDIA**

Como criar sistemas nas nuvens com Spring Cloud

Desenvolva aplicações orientadas a serviços nas nuvens

Computação em Nuvem, ou *Cloud Computing*, já se tornou um termo amplamente conhecido na indústria de tecnologia da informação. Muito já foi especificado e debatido a respeito do assunto, que recentemente deixou de ser uma “utopia” tecnológica para se tornar uma realidade de grandes empresas de TI, como Google, Facebook, Amazon, Salesforce, Twitter, Netflix, entre outras.

De acordo com o NIST (*National Institute of Standards and Technology*), agência do departamento de comércio dos Estados Unidos, cloud computing é definido como “um modelo para permitir acesso ubíquo, conveniente e sob demanda através da rede a um “pool” configurável de recursos computacionais (por exemplo, redes de computadores, servidores, armazenamento em disco, aplicações e serviços) que podem ser rapidamente aprovisionados e liberados para uso com um esforço de gerenciamento mínimo e de preferência sem interferência direta do provedor de serviço”.

A ideia é permitir que qualquer empresa ou pessoa acesse e utilize serviços computacionais oferecidos por um fornecedor, da maneira mais simples possível, usando o quanto quiser (ou puder pagar) e, se for o caso, se desfaça desses recursos com a mesma facilidade.

Nicholas Carr, escritor americano especializado em Tecnologia da Informação, em seu livro *The Big Switch*, afirma: “assim como as empresas e pessoas a cem ou mais anos atrás deixaram de produzir sua própria energia elétrica através de geradores próprios para utilizar a rede elétrica pública, hoje algo semelhante ocorre com a migração de sistemas e hardware das empresas locais para os grandes datacenters, tornando a computação uma utilidade, assim como é a energia”.

Fique por dentro

Neste artigo vamos aprender o que é o Spring Cloud e seu relacionamento com o framework Spring, além de entendermos as principais características para o desenvolvimento de soluções na nuvem. Para isso, abordaremos o que é PaaS (Platform as a Service) e analisaremos na prática como são utilizadas as APIs e abstrações do Spring Cloud e o quanto ele facilita a integração das nossas aplicações com as diversas plataformas e ambientes de computação em nuvem.

Hoje em dia, com um tablet ou smartphone, compramos, conversamos, vendemos, falamos, trabalhamos e nos divertimos, tudo funcionando porque em algum lugar está hospedado um serviço, seja na sua cidade ou do outro lado do mundo, facilmente acessível e a um preço atraente, quando não, gratuito.

Desenvolver estas soluções para funcionar em um ambiente na nuvem exige uma série de cuidados e tecnologias que antes desconhecíamos ou ignorávamos justamente por termos uma falsa impressão de previsibilidade e controle. Durante as fases preliminares de um projeto, estimávamos uma certa quantidade de usuários que iriam utilizar a aplicação e sabíamos que ela seria instalada nos “nossos” servidores e até mesmo contávamos com as famosas “paradas” para fazer manutenção.

Quando criamos um serviço, seja para um ambiente Cloud Computing ou não, devemos nos preparar para eventuais falhas na execução de nossas aplicações, pois elas vão acontecer. Além disso, e principalmente na nuvem, devemos estar preparados para corrigir os defeitos em ciclos de desenvolvimento mais curtos, evitando assim maiores prejuízos econômicos; otimizar ao máximo a performance, pois estamos pagando por cada ciclo de CPU gasto inutilmente; e ainda manter uma arquitetura evolutiva, que permita escalabilidade horizontal e vertical, monitoramento, maior agilidade em recuperação de falhas e administração.

Neste contexto, o Spring Cloud surge para compatibilizar o já mundialmente famoso framework Spring e suas APIs e padrões (como injeção de dependências e inversão de controle) com as plataformas de aplicações nas nuvens, também chamadas de PaaS, tais como CloudFoundry e Heroku.

O que é IaaS, PaaS e SaaS?

A computação nas nuvens é normalmente dividida em três principais modelos de serviço: IaaS (Infraestrutura como serviço), PaaS (Plataforma como Serviço) e SaaS (Software como Serviço).

Em IaaS, o provedor de nuvem disponibiliza uma série de recursos computacionais virtualizados, como servidores, memória, storage, elementos de rede e um ambiente de administração e é o cliente o responsável por instalar e configurar qualquer aplicação que deseja, inclusive sistemas operacionais. Um exemplo de fornecedor de IaaS é o Amazon Web Services.

Já no modelo PaaS, é oferecido ao cliente uma plataforma composta de uma série de serviços e frameworks já instalados, sendo possível ao usuário desenvolver e configurar seus sistemas de acordo com as regras e suporte oferecidos pela plataforma. O Heroku (veja a seção **Links**) é um exemplo claro desse modelo, onde apenas algumas linguagens de programação específicas são suportadas.

Já no modelo SaaS, é oferecido ao consumidor uma aplicação totalmente instalada e configurada em ambiente de nuvem e não é permitido ao usuário ter contato com nenhum aspecto como instalações ou novos desenvolvimentos na plataforma. Gmail, Office Online e Facebook são exemplos simples desse tipo de modelo.

O Spring Cloud pode ser utilizado tanto em IaaS quanto em PaaS. Neste contexto, possibilita uma integração perfeita com plataformas de diversos fornecedores, bem como é compatível com vários projetos open source que normalmente fazem parte de arquiteturas de alto desempenho, como平衡adores de carga, caches, servidores de proxy, controladores de versão e sistemas de mensageria.

Spring Cloud

Para ser capaz de se integrar a uma grande variedade de componentes e fornecedores de plataformas, os engenheiros da empresa Pivotal resolveram separar o Spring Cloud em uma série de componentes menores, a saber:

- **Spring Cloud Config:** é responsável por permitir que as aplicações mantenham suas configurações (propriedades e parâmetros) armazenadas no gerenciador de versões Git. Com configurações e arquivos de propriedades armazenados e versionados remotamente, fica mais fácil reconfigurar aplicações sem a necessidade de acessar cada servidor da nuvem onde as mesmas estão instaladas;
- **Spring Cloud Netflix:** integra aplicações construídas com Spring Cloud a diversos componentes open source da Netflix, tais como Eureka (registrator e balanceador de serviços), Hystrix (biblioteca com algoritmos para construção de sistemas tolerantes a falhas), Zuul (serviço de borda para roteamento, monitoramento

e segurança) e Archaius (biblioteca para gerenciamento de configurações dos componentes da aplicação);

- **Spring Cloud Bus:** simplifica o acesso e uso de servidores de mensageria como RabbitMQ;

- **Spring Cloud for Cloud Foundry:** integra a aplicação aos serviços da plataforma PaaS conhecida como Cloud Foundry;

- **Spring Cloud For Amazon Web Services:** facilitador para construção e manutenção de soluções integradas à plataforma de IaaS conhecida como Amazon Web Services. A partir do Spring Cloud o desenvolvedor pode fazer uso de serviços como Cache distribuído, Mensageria, Storage e outros;

- **Spring Cloud Connectors:** facilitador de integração entre a aplicação e sistemas de backend, principalmente Message Brokers e Bancos de Dados nas nuvens;

- **Spring Cloud CLI:** é um plugin para Spring Boot que traz grande rapidez na criação de aplicações Groovy baseadas em Spring Cloud.

Além disso, o Spring Cloud se integra facilmente a várias outras APIs do framework Spring, tais como Spring Boot, Spring Data, Spring Web e Spring Security. Dessa forma, equipes já treinadas nos componentes citados terão mais facilidade de portar e manter suas aplicações em ambiente de Cloud Computing.

Não podemos deixar de citar que o Spring Cloud é extremamente extensível. Sendo assim, é possível integrá-lo a qualquer serviço ou fornecedor de plataforma IaaS, PaaS ou servidor Middleware, além das tecnologias que já são suportadas. Assim como o próprio framework Spring, o Spring Cloud é apoiado e mantido por uma comunidade engajada e possui uma documentação completa e bastante atualizada.

É importante deixar claro que criar soluções para a nuvem não é uma questão de se adicionar simplesmente os componentes do Spring Cloud ou utilizar o Spring Boot como muitos podem achar. O desenvolvedor que se aventura por esse novo mundo terá que entender, de fato, em que modelo irá construir sua aplicação (IaaS ou PaaS) e que ferramentas e tecnologias ele terá em mãos para fazer sua aplicação escalar com performance sem perder de vista os custos com o uso de hardware e serviços de terceiros.

PaaS na prática: Cloud Foundry

O Cloud Foundry é uma plataforma totalmente open source para criação de soluções em nuvem que está categorizada dentro do modelo PaaS (vide **Figura 1**). De propriedade da Pivotal Software, foi amplamente adotada pela comunidade de desenvolvedores que tem contribuído fortemente para a criação de novos serviços e compatibilização desse produto com os grandes fornecedores de IaaS, como Amazon Web Services, VMware e OpenStack.

A arquitetura do Cloud Foundry comprehende uma série de componentes e servidores que fornecem ao desenvolvedor um ambiente propício para elaboração e instalação em produção de soluções de alta performance, com segurança, mecanismos de tolerância a falhas e monitoramento integrados.

Como criar sistemas nas nuvens com Spring Cloud

Além de sua arquitetura escalável, é simples e fácil integrar o Cloud Foundry a outros sistemas e ferramentas de automação e integração contínua, seja através da chamada ao seu aplicativo cliente chamado cf, seja pelo uso da API REST que dá acesso às funcionalidades da plataforma.

Nesse artigo utilizaremos uma versão de testes online da plataforma Cloud Foundry para hospedarmos a nossa aplicação desenvolvida com Spring Cloud.

Spring Cloud na prática

A partir do que foi apresentado até aqui desenvolveremos uma pequena solução composta de uma aplicação web e um serviço REST, ambos hospedados em um ambiente na nuvem. Para isso, construiremos três componentes: o primeiro será a aplicação web, chamada de "Markers Web", responsável por exibir dados geográficos ao usuário final; o segundo, chamado de "Markers Client",

será responsável por simplificar e concentrar toda a funcionalidade de acesso aos serviços HTTP, atuando como fachada ao serviço remoto de informações geográficas; e o terceiro componente é o serviço REST que fornece os dados geográficos. A Figura 2 mostra a visão geral dessa arquitetura.

Vejamos uma breve descrição a respeito de cada um deles:

- **Aplicação Web (HTML + JavaScript):** o browser interage com a aplicação Markers Web através de uma interface HTML + JavaScript. Ela estará acessível aos usuários finais por meio de um endereço fornecido pela plataforma Cloud Foundry e não acessa os serviços da nuvem diretamente, mas sim a partir do componente Markers Client;
- **Markers Client:** interage com o serviço REST hospedado na nuvem e atua como fachada, isolando a aplicação Markers Web de detalhes do protocolo HTTP;

- **Markers Service:** implementação simples de serviço REST instalado na nuvem que fornece coordenadas geográficas.

O ambiente PaaS

O ambiente PaaS que utilizaremos é o Pivotal Web Services (veja a seção [Links](#)), que é uma instalação do Cloud Foundry oferecido gratuitamente por um período de 60 dias e com uma restrição da quantidade de aplicações que podem ser instaladas.

Para se cadastrar é simples: basta acessar a opção *SIGN UP FOR FREE* (Figura 3) e fornecer seu endereço de e-mail. Feito isso, você receberá uma mensagem para confirmar a criação da conta e em instantes já poderá acessar o ambiente de administração web de onde será possível gerenciar as aplicações e serviços em execução na nuvem.

A maneira mais fácil de instalar aplicações e realizar outras atividades administrativas neste ambiente é através do aplicativo "cf". Ele pode ser obtido a partir dos links disponibilizados dentro do ambiente do Pivotal Web Services (Figura 4), de acordo com o seu ambiente de desenvolvimento. Deste modo, se você vai desenvolver a aplicação para nuvem no Linux, deve adotar a versão do aplicativo CF para Linux e assim sucessivamente.

Caso o aplicativo cf seja corretamente instalado, basta executar o comando *cf -version* na linha de comando. O resultado deve ser semelhante a: *cf version 6.4.0-952cc94-2014-08-13T23:33:07+00:00*.

Orgs e Spaces

Ao acessar a administração Web do Pivotal Web Services (veja a seção [Links](#)) utilizando seu browser, você deve perceber que existem dois nomes no mínimo incomuns: *org* e *space*. Se o seu nome de usuário é, por exemplo, "xx", o nome da sua "org" será "xx-org" e o space será "xx-space".

Na plataforma Cloud Foundry, uma Org é simplesmente uma conta no ambiente de nuvem que pode conter um ou mais usuários e spaces. Um space ou espaço, por sua vez, é uma área onde você pode instalar suas aplicações e serviços. É válido ressaltar que é possível instalar várias aplicações em um mesmo space.

Figura 1. Visão Geral do Cloud Foundry - Fonte: www.cloudfoundry.org



Figura 2. Visão geral da arquitetura da aplicação

Normalmente criamos espaços para organizar os estágios das nossas aplicações. Por exemplo: um espaço para desenvolvimento, outro para homologação, outro para produção, de acordo com as definições do seu projeto. Além disso, é possível criar usuários diferentes com níveis de acesso diferentes para cada space.

Primeiro serviço na nuvem: Markers Service

A maneira mais fácil de utilizar o Spring Cloud em qualquer projeto, seja de serviço REST ou Web, é através de alguma ferramenta de gerenciamento de dependências, tais como o Apache Maven e o Gradle. Para os exemplos desse artigo, faremos uso do Maven e do JDK 7. Portanto, é necessário que o ambiente de desenvolvimento tenha instaladas essas ferramentas (veja a seção [Links](#)).

Visto que adotaremos o Maven para gerenciar o build, começamos criando o arquivo de projeto *pom.xml*, descrito na [Listagem 1](#), para o componente Markers Service. Este será responsável, dentre outras coisas, por fazer o download de todos os pacotes básicos para construção e execução do serviço.

Observando atentamente o *pom.xml*, percebemos que as dependências declaradas na seção `<dependencies>` estão sem versão (veja a tag XML `<version>`). Isso ocorre porque estamos utilizando um projeto pai (`<parent>`, vide linha 13) chamado `spring-cloud-starter-parent` que contém todas as versões dos componentes que normalmente uma aplicação baseada em Spring Cloud utiliza em seu funcionamento básico.

Dando continuidade à análise do *pom.xml*, observamos na linha 9 uma referência para uma classe chamada `br.com.springcloud.Application`. Esta será implementada a seguir e é responsável pela inicialização da aplicação quando a mesma for instalada no ambiente de nuvem.

Ainda na [Listagem 1](#), linha 47, vemos o uso de um plugin do Maven chamado `spring-boot-maven-plugin`. Este plugin faz parte do Spring Boot e permite que o processo de build gere artefatos do tipo JAR ou WAR autocontidos, isto é, que podem

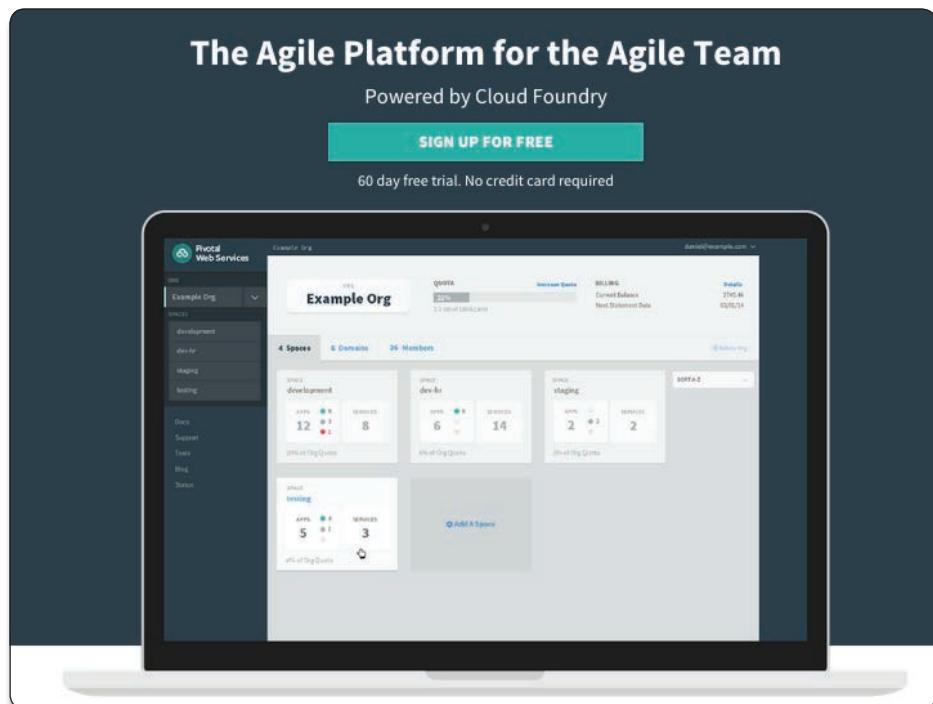


Figura 3. Criação de conta na aplicação Pivotal Web Services

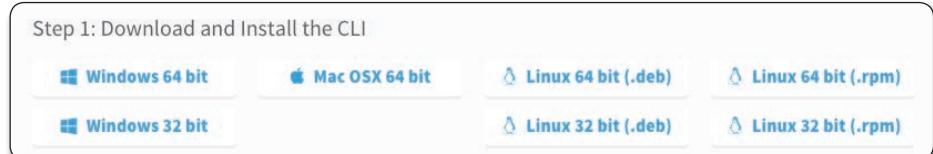


Figura 4. Download do aplicativo cf

ser chamados diretamente pelo comando `java -jar meuarquivo.jar`, por exemplo. Em ambientes de nuvem, principalmente na plataforma Cloud Foundry, essa é a forma usual para execução de aplicações.

Com o *pom.xml* pronto, podemos utilizá-lo para gerar o projeto no Eclipse executando o comando: `mvn eclipse:eclipse`. A versão sugerida da IDE Eclipse para esse projeto é a Luna.

Finalizada a etapa de configuração do projeto, podemos dar início ao desenvolvimento das classes que compõem esse serviço.

A primeira classe a ser criada é aquela responsável por inicializar a aplicação (vide [Listagem 2](#)), chamada `br.com.springcloud.Application`. Como se pode notar, `Application` tem poucas linhas de código, já que faz uso de uma anotação do Spring Boot chamada `@SpringBootApplication`. Esta anotação fará toda a configura-

ração e inicialização dos componentes do framework Spring, como a criação de beans e a injeção automática de dependência.

Quando utilizamos o Spring Boot juntamente com o Spring Security (como é o caso das aplicações que dependem do Spring Cloud), um dos beans criados é o `AuthenticationManager`. Este configura, automaticamente, um usuário default para a aplicação chamado `user`, cuja senha é alterada toda vez que a aplicação é iniciada, sendo exibida no log de inicialização.

Para desativarmos o mecanismo de troca de senha, utilizamos a propriedade `security.user.password`. Em nosso caso, configuramos seu valor como "javajava" (linha 10). Assim, esta passa a ser a senha do usuário default.

A lógica do serviço REST está contida em uma classe de nome `br.com.springcloud.controller.MarkersController` (vide [Listagem 3](#)).

Como criar sistemas nas nuvens com Spring Cloud

Listagem 1. Arquivo pom.xml para setup do projeto Maven.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <project>
03   <modelVersion>4.0.0</modelVersion>
04   <groupId>br.com.springcloud</groupId>
05   <artifactId>markers-service</artifactId>
06   <version>0.0.1</version>
07   <properties>
08     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
09   <start-class>br.com.springcloud.Application</start-class>
10   <java.version>1.7</java.version>
11 </properties>
12 <parent>
13   <groupId>org.springframework.cloud</groupId>
14   <artifactId>spring-cloud-starter-parent</artifactId>
15   <version>1.0.0.RELEASE</version>
16   <relativePath />!-- lookup parent from repository -->
17 </parent>
18 <dependencies>
19   <dependency>
20     <groupId>org.springframework.cloud</groupId>
21     <artifactId>spring-cloud-config-client</artifactId>
22   </dependency>
23   <dependency>
24     <groupId>org.springframework.cloud</groupId>
25     <artifactId>spring-cloud-config-server</artifactId>
26 </dependency>
```



```
27   <dependency>
28     <groupId>org.springframework.boot</groupId>
29     <artifactId>spring-boot-starter-security</artifactId>
30   </dependency>
31   <dependency>
32     <groupId>org.springframework.cloud</groupId>
33     <artifactId>spring-cloud-cloudfoundry-connector</artifactId>
34   </dependency>
35   <dependency>
36     <groupId>org.springframework.cloud</groupId>
37     <artifactId>spring-cloud-spring-service-connector</artifactId>
38   </dependency>
39   <dependency>
40     <groupId>org.springframework.boot</groupId>
41     <artifactId>spring-boot-starter-actuator</artifactId>
42   </dependency>
43 </dependencies>
44 <build>
45   <plugins>
46     <plugin>
47       <groupId>org.springframework.boot</groupId>
48       <artifactId>spring-boot-maven-plugin</artifactId>
49     </plugin>
50   </plugins>
51 </build>
52 </project>
```

Listagem 2. Classe de inicialização do componente Markers Service.

```
01 package br.com.springcloud;
02
03 import org.springframework.boot.SpringApplication;
04 import org.springframework.boot.autoconfigure.SpringBootApplication;
05
06 @SpringBootApplication
07 public class Application {
08
09   public static void main(String[] args) {
10     System.setProperty("security.user.password", "javajava");
11     SpringApplication.run(Application.class, args);
12   }
13 }
```

Note que ela também faz uso de anotações de outra API do framework Spring, o Spring MVC, para mapear as requisições feitas a esse serviço ao contexto `/markers` (vejas as anotações `@Controller` e `@RequestMapping` nas linhas 9 e 10). Por simplicidade, foi criado apenas um método, chamado `getMarkers()` (linha 14), para retornar em formato JSON objetos do tipo `Position`. O código desta classe, que representa uma coordenada geográfica, é apresentado na [Listagem 4](#).

Se você é um desenvolvedor com certa experiência em Spring, notará que até o momento não criamos nada específico para o ambiente em nuvem. Apenas estamos reaproveitando funcionalidades já oferecidas pelo framework.

Com as nossas classes Java prontas, basta criarmos um arquivo descriptor (chamado `manifest.yml` – veja a [Listagem 5](#)) no diretório raiz do projeto, seguindo o padrão YAML (BOX 1), que é reconhecido pela plataforma Cloud Foundry para definirmos que recursos computacionais serão necessários para

Listagem 3. Código da classe para o Serviço REST.

```
01 package br.com.springcloud.controller;
02
03 import java.util.ArrayList;
04 import java.util.List;
05
06 import org.springframework.stereotype.Controller;
07 import org.springframework.web.bind.annotation.*;
08
09 @Controller
10 @RequestMapping("/markers")
11 public class MarkersController {
12
13   @RequestMapping(method=RequestMethod.GET)
14   public @ResponseBody List<Position> getMarkers() {
15     List<Position> positions = new ArrayList<Position>();
16     Position torreEiffel = new Position(48.858292, 2.294347);
17     positions.add(torreEiffel);
18   return positions;
19 }
20 }
```

Listagem 4. Código da classe Position.

```
01 package br.com.springcloud.controller;
02
03 public final class Position {
04   private Double lat;
05   private Double lng;
06
07   Position(double lat, double lng) {
08     this.lat = lat;
09     this.lng = lng;
10   }
11
12   public Double getLat() {
13     return lat;
14   }
15
16   public Double getLng() {
17     return lng;
18   }
19 }
```

que a aplicação seja configurada e executada corretamente no ambiente de nuvem.

Na linha 3 definimos o nome da aplicação como **markers-service**. Na linha 4 o atributo **host** determina o endereço externo do serviço. Podemos adicionar a esse atributo a marcação **{random-word}** para que a cada nova instalação seja gerado um nome aleatório diferente, garantindo dessa forma que não haverá conflito da URL da nossa aplicação com outra já instalada por outro desenvolvedor no ambiente. Assim, quando disponibilizarmos **markers-service** na plataforma Pivotal Web Services, seu endereço terá o seguinte formato: <http://markers-service-XXX.cfapps.io>, onde XXX é o nome randômico gerado no momento da instalação.

Em seguida, nas linhas 5 e 6 determinamos que serão alocados 512 megabytes de memória RAM para a instanciação de apenas uma JVM para conter a aplicação, isto é, não haverá balanceamento de carga, já que para isso devem existir duas ou mais instâncias ativas.

É interessante saber que o PaaS se encarrega de manter sempre uma instância da aplicação “no ar”, de forma que, em caso de algum erro que cause a finalização da JVM, **markers-service** será reiniciado, fazendo com que o serviço continue disponível.

Listagem 5. Arquivo de manifesto em formato YAML.

```
01 ---
02 applications:
03 - name: markers-service
04   host: markers-service-${random-word}
05   memory: 512M
06   instances: 1
07   path: target\markers-service-0.0.1.jar
08   timeout: 180
09
10 env:
11   SPRING_PROFILES_ACTIVE: cloud
```

BOX 1. YAML

É um formato de serialização inspirado em formatos como JSON e linguagens de programação como Python e Perl. É amplamente utilizado por ferramentas e frameworks de nuvem, como o Cloud Foundry e OpenStack, para armazenar configurações de forma concisa e organizada, facilmente interpretada por máquinas e seres humanos. A sintaxe é totalmente diferente do formato XML e muitas vezes o conteúdo deve ser identado (sempre utilizando espaços em vez do “caractere” tab). Essa formatação serve para estruturar o conteúdo de forma hierárquica, assim como um arquivo XML.

Agora, antes de fazer a instalação da aplicação na nuvem é necessário gerar o pacote que está configurado no *manifest.yml*. Para isso, utilizaremos o comando do *mvn clean package*, que em caso de sucesso deve gerar, na pasta *target*, o artefato *markers-service-0.0.1.jar*.

Realizar a instalação do JAR em um ambiente Cloud Foundry depende do uso do aplicativo cf, além das credenciais da sua conta de usuário no serviço Pivotal Web Services. Dito isso, inicialmente é necessário fazer o login através do comando *cf api https://api.run.pivotal.io*. A execução deste iniciará o processo de logon, exigindo que sejam fornecidos seu e-mail e senha.

Em seguida, será solicitado que você escolha um nome de espaço (*Space*) em uma lista apresentada pelo aplicativo. Qualquer espaço pode ser selecionado. Ele representa sua área de trabalho, isto é, o local onde devem ser publicadas as aplicações.

Terminada essa etapa, estamos prontos para instalar o pacote contendo o **Markers Service** no ambiente remoto. Para isso, devemos executar, no diretório raiz do projeto, no prompt de comando, o comando *cf push markers-service -f manifest.yml*, que gerará uma série de informações na saída padrão e emitirá o status **OK** caso a aplicação tenha sido “deployada” com sucesso (**Figura 5**).

```
Utilizando arquivo de manifesto manifest.yml
Atualizando app markers-service na org xxx-org / espaço development como xxx@gmail.com...
OK
Criando rota markers-service-powerful-nil.cfapps.io...
OK
Vinculando markers-service-powerful-nil.cfapps.io com markers-service...
OK
Enviando markers-service...
Enviando app com arquivos do caminho: target/markers-service-0.0.1.jar
Enviando 782.5K, 106 arquivos
Done uploading
OK
Parando app markers-service na org xxx-org / espaço development como xxx@gmail.com...
OK
Atenção: falta ao tentar exibir logs continuadamente
Terminal loggercat ausente em arquivo de configuração
Iniciando app markers-service na org xxx-org / espaço development como xxx@gmail.com...
0 de 1 instâncias em execução, 1 iniciando
1 de 1 instâncias em execução

Aplicativo iniciado
OK
App markers-service was started using this command `SERVER_PORT=$PORT $PWD/java-buildpack/open_jdk_jre/bin/java -cp $PWD/:$PWD/java-buildpack/spring_auto_reconfiguration/spring_auto_reconfiguration-1.7.0.RELEASE.jar -Djava.io.tmpdir=$TMPDIR -XX:OnOutOfMemoryError=$PWD/java-buildpack/open_jdk_jre/bin/killjava.sh -Xmx382293K -Xms382293K -XX:MaxMetaspaceSize=64M -XX:MetaspaceSize=64M -Xss95K org.springframework.boot.loader.JarLauncher` 
Mostrando status da app markers-service na org xxx-org / espaço development como xxx@gmail.com...
OK
estado requerido: started
instâncias: 1/1
uso: 512M x 1 instâncias
urls: markers-service.cfapps.io, markers-service-powerful-nil.cfapps.io
package uploaded: Thu Apr 16 20:13:39 UTC 2015
stack: lucid64

  estado      desde      cpu    memória      disco rígido      details
#0  executando  2015-04-16 05:14:30 PM  0.0%  363.7M de 512M  147.5M de 1G
```

Figura 5. Processo de instalação do serviço **Markers Service**

Ainda resta uma última tarefa: tornar o serviço REST disponível para uso pelas aplicações. Para isso, executaremos o seguinte comando: *cf create-user-provided-service marker-ws*, indicando que estamos habilitando um serviço fornecido por um usuário (*user-provided service*) e não pelos engenheiros da plataforma Cloud Foundry (**BOX 2**).

BOX 2. Cloud Foundry

Nele, qualquer aplicação reutilizável pode ser um serviço (Service), seja ela um Banco de Dados relacional ou NoSQL, um Middleware de Mensageria ou, no nosso caso, o **Markers Service**. Assim, aplicações que queiram fazer uso de **Markers Service** devem fazê-lo através do seu apelido: *marker-ws*. Veremos como isso se aplica na prática mais adiante.

Conectando-se ao serviço na nuvem: **Markers Client**

O próximo componente que examinaremos será o **Markers Client**, que é responsável por concentrar toda a lógica de acesso ao serviço **Markers Service**. Esse projeto também utiliza o Maven para gerenciar suas dependências e processo de build e, portanto, a criação de um arquivo *pom.xml* se faz necessária (vide **Listagem 6**).

Como criar sistemas nas nuvens com Spring Cloud

Listagem 6. pom.xml para o projeto Markers Client.

```
01 <project>
02   <modelVersion>4.0.0</modelVersion>
03   <groupId>br.com.springcloud</groupId>
04   <artifactId>markers-client</artifactId>
05   <version>0.0.1</version>
06   <properties>
07     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
08     <start-class>br.com.springcloud.ConfigServer</start-class>
09     <java.version>1.7</java.version>
10     <feignVersion>6.1.3</feignVersion>
11   </properties>
12   <parent>
13     <groupId>org.springframework.cloud</groupId>
14     <artifactId>spring-cloud-starter-parent</artifactId>
15     <version>1.0.0.RELEASE</version>
16     <relativePath /> <!-- lookup parent from repository -->
17   </parent>
18   <dependencies>
19     <dependency>
20       <groupId>org.springframework.cloud</groupId>
21       <artifactId>spring-cloud-config-client</artifactId>
22     </dependency>
23     <dependency>
24       <groupId>org.springframework.cloud</groupId>
25       <artifactId>spring-cloud-config-server</artifactId>
26     </dependency>
27     <dependency>
28       <groupId>org.springframework.boot</groupId>
29       <artifactId>spring-boot-starter-security</artifactId>
30     </dependency>
31   </dependencies>
32   <dependency>
33     <groupId>org.springframework.boot</groupId>
34     <artifactId>spring-boot-starter-test</artifactId>
35     <scope>test</scope>
36   </dependency>
37   <dependency>
38     <groupId>org.springframework.cloud</groupId>
39     <artifactId>spring-cloud-cloudfoundry-connector</artifactId>
40   </dependency>
41   <dependency>
42     <groupId>org.springframework.cloud</groupId>
43     <artifactId>spring-cloud-spring-service-connector</artifactId>
44   </dependency>
45   <dependency>
46     <groupId>org.springframework.boot</groupId>
47     <artifactId>spring-boot-starter-actuator</artifactId>
48   </dependency>
49   <dependency>
50     <groupId>com.netflix.feign</groupId>
51     <artifactId>feign-core</artifactId>
52     <version>${feignVersion}</version>
53   </dependency>
54   <dependency>
55     <groupId>com.netflix.feign</groupId>
56     <artifactId>feign-jackson</artifactId>
57     <version>${feignVersion}</version>
58   </dependency>
59 </dependencies>
60 </project>
```

Como podemos notar, o *pom.xml* desse componente é muito parecido com o que criamos para Markers Service, entretanto, adicionamos a ele as dependências *feign-core* e *feign-jackson* (linhas 50 a 58), já que utilizaremos o cliente HTTP Feign (BOX 3) para realizar as requisições ao serviço REST.

BOX 3. Feign

É um cliente HTTP em Java construído pela empresa Netflix com um design fortemente inspirado por outras bibliotecas como Retrofit (comumente utilizada por aplicações Android), JAX-RS 2.0 e WebSockets. A ideia dos engenheiros que elaboraram essa biblioteca é que ela permitisse a integração com outros frameworks de mercado, reutilizando ao máximo o que já existe e ainda assim possibilitar ao desenvolvedor fazer chamadas HTTP/HTTPS com o mínimo de código possível. O Feign será utilizado no nosso projeto de teste para fazer a integração com o serviço REST Markers Service.

Estabelecer o acesso a qualquer recurso provido pela plataforma PaaS, seja esse recurso um serviço REST ou um banco de dados, exige a utilização de mecanismos e APIs fornecidos pelo ambiente no qual nossa aplicação está inserida. Em nosso caso, nossas soluções fazem uso dessas APIs por meio de uma abstração do Spring Cloud, que possui um componente especializado chamado Spring Cloud for Cloud Foundry. Este concentra todas as especificidades e complexidades da plataforma em questão, permitindo que nosso código fique mais coeso e simples.

Um bom exemplo de como o Spring Cloud simplifica e provê a integração perfeita com a nuvem pode ser verificado com a im-

plementação do componente Markers Client. Este utiliza classes e interfaces do Spring Cloud para criar instâncias da interface **MarkersRepository**, cujo código é apresentado na Listagem 7, e serve como fachada para as operações fornecidas por Markers Service, como demonstra a Figura 6.

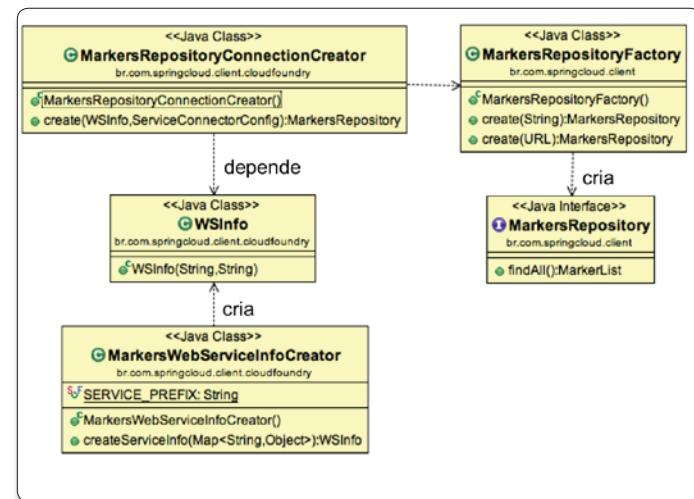


Figura 6. Classes do componente Markers Client

Rapidamente podemos notar que a interface **MarkersRepository** está anotada com **@Repository** (linha 8) e sua única operação, **findAll()**, que recebe a anotação **@RequestLine** (linha 11), retorna uma lista de objetos **Marker** (Listagem 8). Assim, quando o

método **findAll()** for chamado será feita uma requisição HTTP GET ao contexto `/markers`, exposto por Markers Service através de seu controlador.

Listagem 7. Interface de Repositório MarkersRepository.

```
01 package br.com.springcloud.client;
02
03 import org.springframework.stereotype.Repository;
04
05 import feign.RequestLine;
06 import br.com.springcloud.client.model.MarkerList;
07
08 @Repository
09 public interface MarkersRepository {
10
11     @RequestLine("GET /markers")
12     public MarkerList findAll();
13 }
```

Listagem 8. Objeto Marker, para representar pontos geográficos.

```
01 package br.com.springcloud.client.model;
02
03 import java.io.Serializable;
04 import java.util.ArrayList;
05
06 public class Marker implements Serializable {
07     private double lat;
08     private double lng;
09
10     public double getLat() {
11         return lat;
12     }
13     public void setLat(double lat) {
14         this.lat = lat;
15     }
16     public double getLng() {
17         return lng;
18     }
19     public void setLng(double lng) {
20         this.lng = lng;
21     }
22     public static class MarkerList extends ArrayList<Marker> {}
23 }
```

A classe responsável por criar instâncias de **MarkersRepository** é chamada **MarkersRepositoryService**, como expõe o código da **Listagem 9**. Ela possui um método **create()** (linha 12) que recebe por parâmetro a URL do serviço REST que queremos consultar.

No método **create()**, inicialmente criamos uma instância de **ObjectMapper**, que é responsável pela transformação das mensagens JSON vindas do serviço em objetos da aplicação. Logo após, utilizamos o **Feign** para processar as anotações da classe **MarkersRepository** (linha 20), gerando em seguida uma instância dessa interface. Note que com apenas uma linha de código adicionamos a autenticação básica, recurso necessário para acessar markers-service, o que demonstra claramente que o Feign cumpre sua promessa de ser extremamente produtivo.

É possível criar uma instância do repositório **MarkersRepository** passando por parâmetro a URL do serviço Markers Service. Isso é feito pela classe **MarkersRepositoryConnectionCreator**, apresentada na **Listagem 10**.

Nota

Toda classe anotada com `@Repository` representa um estereótipo de Repositório que, de acordo com a documentação do framework Spring, é "um mecanismo para encapsular armazenamento, recuperação e comportamentos de busca que emitem uma coleção de objetos. De maneira geral, desenvolvedores vêm utilizando essa anotação em classes que fazem acesso ao banco de dados como DAO (Data Access Object), mas por ser de livre uso, `@Repository` pode ser aplicada em outros contextos, de acordo com as definições de cada projeto."

Listagem 9. Factory para MarkersRepository.

```
01 package br.com.springcloud.client;
02
03 import java.net.URL;
04
05 import com.fasterxml.jackson.databind.*;
06 import feign.Feign;
07 import feign.auth.BasicAuthRequestInterceptor;
08 import feign.jackson.JacksonDecoder;
09
10 public class MarkersRepositoryFactory {
11
12     public MarkersRepository create(String url) {
13         ObjectMapper mapper = new ObjectMapper()
14             .configure(
15                 DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
16         return Feign.builder()
17             .requestInterceptor(
18                 new BasicAuthRequestInterceptor("user","javajava"))
19             .decoder(new JacksonDecoder(mapper))
20             .target(MarkersRepository.class, url);
21     }
22 }
23
24 public MarkersRepository create(URL url) {
25     return create(url.toString());
26 }
27 }
```

Listagem 10. Integrando Spring Cloud com MarkersRepository.

```
01 package br.com.springcloud.client.cloudfoundry;
02
03 import org.springframework.cloud.service.*;
04 import br.com.springcloud.client.*;
05
06 public class MarkersRepositoryConnectionCreator
07     extends AbstractServiceConnectorCreator<MarkersRepository, WSInfo> {
08
09     @Override
10     public MarkersRepository create(WSInfo i, ServiceConnectorConfig c) {
11         return new MarkersRepositoryFactory().create(i.getUri());
12     }
13 }
```

Se você imaginou que iria se envolver com várias particularidades e complexidades da plataforma a fim de interpretar a URL que representa o endereço de Markers Service, se enganou: o Spring Cloud já fez isso por nós. A única coisa que precisamos é sobrecrever o método **create()**, herdado da classe **org.springframework.cloud.service.AbstractServiceConnectorCreator**, fazendo com que ele delegue a lógica de criação ao **create()** da factory **MarkersRepositoryFactory**.

Desde a versão 1.6 a plataforma Java possui um mecanismo de carregamento de serviços conhecido como ServiceLoader. Este é utilizado pelo Spring Cloud para instanciar objetos do tipo **AbstractServiceConnectorCreator**, como é o caso da classe **MarkersRepositoryConnectionCreator**. Para o correto funcionamento dessa instanciação, devemos criar um arquivo descriptor chamado *org.springframework.cloud.service.ServiceConnectorCreator* na pasta / *META-INF/services* do nosso componente Markers Client. O código desse arquivo é apresentado a seguir:

```
br.com.springcloud.client.cloudfoundry.MarkersRepositoryConnectionCreator
```

Quando esse componente for carregado no classpath, o Java carregará o descriptor e o Spring Cloud saberá que para criar instâncias de **MarkersRepository**, deverá fazer uso da classe **MarkersRepositoryConnectionCreator**.

Para finalizar a lista de classes de Markers Client, veremos agora **MarkersWebServiceInfoCreator**, apresentada na **Listagem 11**. Esta classe interage com a plataforma PaaS através da extensão de *org.springframework.cloud.cloudfoundry.CloudFoundryServiceInfoCreator*.

Listagem 11. Obtendo a URL do serviço Markers Service.

```
01 package br.com.springcloud.client.cloudfoundry;
02
03 import java.util.Map;
04 import org.springframework.cloud.cloudfoundry.*;
05
06 public class MarkersWebServiceInfoCreator
07 extends CloudFoundryServiceInfoCreator<WSInfo> {
08 public static final String SERVICE_PREFIX = "markers";
09
10 public MarkersWebServiceInfoCreator() {
11 super(new Tags(), SERVICE_PREFIX);
12 }
13
14 @Override
15 public WSInfo createServiceInfo(Map<String, Object> serviceData) {
16 String id = (String) serviceData.get("name");
17 Map<String, Object> credentials = getCredentials(serviceData);
18 String uri = getUriFromCredentials(credentials);
19 return new WSInfo(id, uri);
20 }
21 }
```

Para isso, é necessário criar um construtor para **MarkersWebServiceInfoCreator** (linha 10) que identificará e filtrará na lista de serviços que estão disponíveis para a aplicação, apenas o desejado. No nosso caso, estamos interessados em referenciar o serviço cujo nome se inicia com a palavra *markers* (linha 8).

Em seguida, devemos sobrescrever o método **createServiceInfo()** (linha 15), responsável por criar objetos do tipo **WSInfo**, que é aquele que contém toda e qualquer informação relativa a um serviço associado à aplicação na plataforma Cloud Foundry. Veremos adiante como associar um ou mais serviços a uma aplicação dentro da plataforma na nuvem.

Para que **MarkersWebServiceInfoCreator** seja reconhecido e ativado pelo Spring Cloud, devemos criar um arquivo chamado *org.springframework.cloud.cloudfoundry.CloudFoundryServiceInfoCreator* na pasta / *META-INF/services* do componente Markers Client. O conteúdo desse arquivo é apresentado a seguir:

```
br.com.springcloud.client.cloudfoundry.MarkersWebServiceInfoCreator
```

Finalmente, temos um cliente pronto para consultar coordenadas geográficas e expor essas informações por meio de um repositório. Faremos uso desse cliente na aplicação Spring Web, que exibirá para o usuário final um mapa integrado ao Google Maps.

Aplicação HTML: Markers UI

A aplicação Markers UI se assemelha muito a uma aplicação Spring MVC convencional, mas com um tempero especial para a nuvem. Nela utilizaremos o componente Markers Client para consultar o serviço REST e exibir, por intermédio de uma interface HTML, a coordenada geográfica a partir da integração com o Google Maps.

Assim como nos demais componentes construídos até o momento, novamente faremos uso do Maven e, portanto, precisaremos desenvolver mais um descriptor *pom.xml* (vide **Listagem 12**). Nesse arquivo, vemos a referência para a dependência *markers-client* (linha 49).

Como toda aplicação que utiliza o Spring Cloud e, consequentemente, o Spring Boot, precisamos criar uma classe inicial que será responsável por iniciar a configuração dos “beans” do framework. Neste componente, essa classe será chamada de **Application** (veja o código na **Listagem 13**).

As anotações utilizadas na classe **Application** são: **@SpringBootApplication**, que equivale, na prática, a adicionarmos as anotações **@Configuration**, **@EnableAutoConfiguration** e **@ComponentScan**; e **@EnableWebSecurity**, que dá suporte à configuração da segurança dos recursos da aplicação, como páginas, contextos, etc.

Para permitirmos que usuários anônimos acessem as páginas HTML de Markers UI, reconfiguramos suas políticas de acesso e segurança através da criação de um bean do tipo **WebSecurityConfigurerAdapter** (linha 10). No Spring, classes que estendem *org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter* têm a capacidade de habilitar ou desabilitar métodos de autenticação, permissões para acesso a contextos web e a páginas.

Ao analisar a classe que estende **WebSecurityConfigurerAdapter**, vemos que toda a lógica de segurança está contida no método **configure()**, que executa duas ações: a primeira (linha 12), desabilita a autenticação HTTP básica, padrão em aplicações Spring Cloud; e a segunda é permitir que um usuário sem privilégios acesse toda a aplicação (linha 13).

Seguindo o padrão de uma aplicação Spring convencional, criamos uma classe com a anotação **@Configuration** chamada **CloudConfiguration** (vide **Listagem 14**), que será responsável

Listagem 12. Arquivo pom.xml para Markers UI.

```
01 <project>
02   <modelVersion>4.0.0</modelVersion>
03   <groupId>br.com.springcloud</groupId>
04   <artifactId>markers-ui</artifactId>
05   <version>0.0.1</version>
06   <properties>
07     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
08     <start-class>br.com.springcloud.ui.Application</start-class>
09     <java.version>1.7</java.version>
10   </properties>
11   <parent>
12     <groupId>org.springframework.cloud</groupId>
13     <artifactId>spring-cloud-starter-parent</artifactId>
14     <version>1.0.0.RELEASE</version>
15     <relativePath /> <!-- lookup parent from repository -->
16   </parent>
17   <dependencies>
18     <dependency>
19       <groupId>org.springframework.cloud</groupId>
20       <artifactId>spring-cloud-config-client</artifactId>
21     </dependency>
22     <dependency>
23       <groupId>org.springframework.cloud</groupId>
24       <artifactId>spring-cloud-config-server</artifactId>
25     </dependency>
26     <dependency>
27       <groupId>org.springframework.cloud</groupId>
28       <artifactId>spring-cloud-cloudfoundry-connector</artifactId>
29     </dependency>
30     <dependency>
31       <groupId>org.springframework.cloud</groupId>
32       <artifactId>spring-cloud-service-connector</artifactId>
33     </dependency>
34     <dependency>
35       <groupId>org.springframework.boot</groupId>
36       <artifactId>spring-boot-starter-actuator</artifactId>
37     </dependency>
38     <dependency>
39       <groupId>org.springframework.boot</groupId>
40       <artifactId>spring-boot-starter-web</artifactId>
41     </dependency>
42     <dependency>
43       <groupId>org.springframework.data</groupId>
44       <artifactId>spring-data-commons</artifactId>
45     </dependency>
46     <dependency>
47       <groupId>br.com.springcloud</groupId>
48       <artifactId>markers-client</artifactId>
49       <version>0.0.1</version>
50     </dependency>
51   </dependencies>
52   <build>
53     <plugins>
54       <plugin>
55         <groupId>org.springframework.boot</groupId>
56         <artifactId>spring-boot-maven-plugin</artifactId>
57       </plugin>
58     </plugins>
59   </build>
60 </project>
```

Listagem 13. Classe de Aplicação para Markers UI.

```
01 package br.com.springcloud.ui;
02
03 import org.springframework.boot.SpringApplication;
04 import org.springframework.boot.autoconfigure.SpringBootApplication;
05 import org.springframework.security.config.annotation.web.configuration.*;
06 @SpringBootApplication
07 @EnableWebSecurity
08 public class Application {
09   public WebSecurityConfigurerAdapter webSecurityConfigurerAdapter() {
10     return new WebSecurityConfigurerAdapter {
11       protected void configure(HttpSecurity http) throws Exception {
12         http.httpBasic().disable();
13         http.authorizeRequests().antMatchers("/**").anonymous();
14     }
15   };
16 }
17
18 public static void main(String[] args) {
19   SpringApplication.run(Application.class, args);
20 }
21 }
```

por configurar beans de repositório **MarkersRepository**. Note ainda que utilizamos a anotação **@Profile**, do Spring Boot, para determinar que essa configuração será ativada apenas quando a variável de ambiente **spring.profiles.active** possuir seu valor igual à String “cloud”. Essa variável possui o valor padrão “default”.

O ponto principal de **CloudConfiguration** é o fato dela ser uma extensão de **org.springframework.cloud.config.java.AbstractCloudConfig**. Por isso, por meio do método **connectionFactory()** (linha 13) essa classe é capaz de obter, no contexto Spring da aplicação, instâncias de **MarkersRepository**.

Listagem 14. Classe de Configuração para serviços de Markers UI.

```
01 package br.com.springcloud.ui.config;
02
03 import org.springframework.cloud.config.java.AbstractCloudConfig;
04 import org.springframework.context.annotation.*;
05
06 import br.com.springcloud.client.MarkersRepository;
07
08 @Configuration
09 @Profile({"cloud"})
10 public class CloudConfiguration extends AbstractCloudConfig {
11   @Bean
12   public MarkersRepository markersRepository() {
13     return connectionFactory().service(MarkersRepository.class);
14   }
15 }
```

Com o repositório pronto para uso, podemos construir um controlador que será responsável por atender às requisições feitas pelo usuário por intermédio de seu navegador. O nosso controlador principal será a classe **MainController**, cujo código é exposto na

Como criar sistemas nas nuvens com Spring Cloud

Listagem 15. e atende a requisições feitas no endereço /locations.

Note que uma instância do repositório é injetada através do seu construtor (linha 14). Essa instância é utilizada pelo método **findAll()** – linha 20 – para retornar as informações de coordenadas geográficas.

O componente Markers UI, a partir do uso de Repository e Spring Cloud, cumpriu seu papel de facilitador para acesso ao serviço REST, pois a aplicação web não precisou conhecer nenhum detalhe sobre o protocolo HTTP ou JSON para interagir com a plataforma Cloud Foundry ou com Markers Service.

Uma única página HTML, apresentada na **Listagem 16**, é responsável por fazer a chamada à URL /locations (linha 65). Esta chamada, por sua vez, é processada pelo controlador, que logo após fará a chamada ao serviço REST, como já explicado.

Para viabilizarmos a integração com o Google Maps de maneira simples, foi utilizado o framework JavaScript *gmaps.js* (veja a seção **Links**). A partir dele podemos interpretar as coordenadas geográficas obtidas remotamente (linha 10) e adicionar essa informação

Listagem 15. Controlador principal.

```
01 package br.com.springcloud.ui.controller;
02
03 import org.springframework.beans.factory.annotation.Autowired;
04 import org.springframework.web.bind.annotation.*;
05 import br.com.springcloud.client.MarkersRepository;
06 import br.com.springcloud.client.model.Marker.MarkerList;
07
08 @RestController
09 @RequestMapping("/locations")
10 public class MainController {
11     private MarkersRepository repository;
12
13     @Autowired
14     public MainController(MarkersRepository rep) {
15         this.repository = rep;
16     }
17
18     @RequestMapping(method=RequestMethod.GET)
19     public MarkerList findAll() {
20         return repository.findAll();
21     }
22 }
```

Listagem 16. Página HTML para interface com o usuário

```
01 <!DOCTYPE html>
02 <html>
03 <head>
04     <title>gmaps.js &mdash; the easiest way to use Google Maps</title>
05     <script type="text/javascript" src="jquery.min.js"/>
06     <script type="text/javascript" src="gmaps.js"></script>
07     <script type="text/javascript">
08         var map;
09
10         function loadResults (data) {
11             var items, markers_data = [];
12             if (data.length > 0) {
13                 items = data;
14
15                 for (var i = 0; i < items.length; i++) {
16                     var item = items[i];
17
18                     if (item.lat != undefined && item.lng != undefined) {
19                         var icon = 'https://foursquare.com/img/categories/food/default.png';
20
21                         markers_data.push({
22                             lat : item.lat,
23                             lng : item.lng
24                             icon : {
25                                 size : new google.maps.Size(32, 32),
26                                 url : icon
27                             }
28                         });
29                     }
30                 }
31             }
32             map.addMarkers(markers_data);
33         }
34
35         function printResults(data) {
36             $('#foursquare-results').text(JSON.stringify(data));
37         }
38
39         $(document).on('click','.pan-to-marker',function(e) {
40             e.preventDefault();
41             var position, lat, lng, $index;
42             $index = $(this).data('marker-index');
43             position = map.markers[$index].getPosition();
44             lat = position.lat();
45             lng = position.lng();
46             map.setCenter(lat, lng);
47         });
48
49
50         $(document).ready(function(){
51             map = new GMaps({
52                 div:'#map',
53                 lat: -12.043333,
54                 lng: -77.028333
55             });
56
57             map.on('marker_added', function (marker) {
58                 var index = map.markers.indexOf(marker);
59                 $('#results').append('<li><a href="#" class="pan-to-marker" data-marker-index="'+ index +'">' + marker.title + '</a></li>');
60             }
61             if (index == map.markers.length - 1) {
62                 map.fitZoom();
63             }
64         });
65         var xhr = $.getJSON('/locations');
66         xhr.done(loadResults);
67     });
68     </script>
69 </head>
70 <body>
71     <div id="body">
72         <h3>Working with JSON</h3>
73         <div class="row">
74             <div class="span11">
75                 <div class="popin">
76                     <div id="map"></div>
77                 </div>
78             </div>
79         </div>
80     </div>
81 </body>
82 </html>
```

à lista de pontos existentes (linha 21). O resultado será um mapa com as coordenadas plotadas, como demonstra a **Figura 7**.

Por ser uma aplicação web, para fazer sua instalação no Pivotal Web Services, será necessário criar um arquivo de deployment chamado *manifest.yml*, de acordo com a **Listagem 17**. Nesse arquivo devemos colocar todas as informações necessárias para o funcionamento de Markers UI, tais como quantidade de memória (linha 4) e artefato a ser instalado (linha 6). Em seguida observamos uma referência para o serviço *markers-ws* (linha 7), que é o nome dado ao serviço Markers Service quando realizamos sua instalação na plataforma Pivotal Web Services.

Listagem 17. Configuração do arquivo manifest.yml.

```
1 ---
2 applications:
3 - name: markers-ui-$random-word
4 memory: 512M
5 instances: 1
6 path: target/markers-ui-0.0.1.jar
7 services: [ markers-ws ]
8 env:
9 SPRING_PROFILES_ACTIVE: cloud
```

Working with JSON

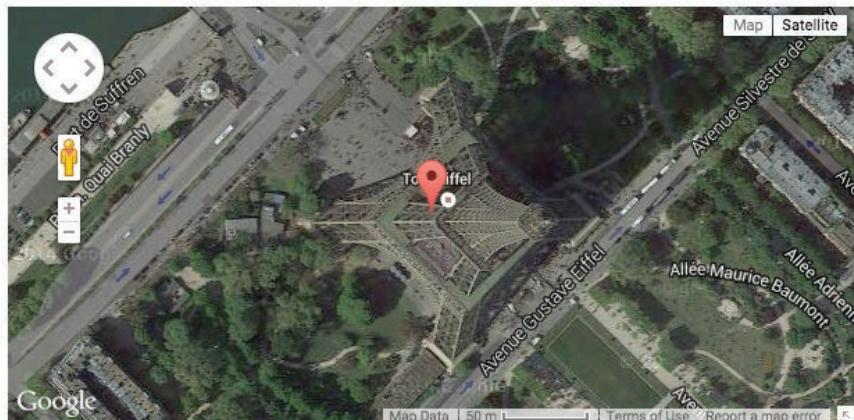


Figura 7. Aplicação web Markers UI

Finalizando a análise do manifesto, note que adicionamos uma variável de ambiente através da tag **env** (linha 8) que serve para ativar o profile Spring chamado “cloud”, referenciado pela anotação **@Profile** na definição da classe de configuração **CloudConfiguration** (veja novamente a **Listagem 14**). Com isso, ga-

rantimos que os beans criados pela classe **CloudConfiguration** estarão disponíveis para uso pela aplicação.

A instalação de Markers UI segue os mesmos passos básicos utilizados na instalação do componente Markers Service. Sendo assim, primeiro geramos o pacote utilizando o Maven através do comando

RENOVE JÁ!

Sua assinatura pode estar acabando

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

Brinde!

DEV MEDIA

Como criar sistemas nas nuvens com Spring Cloud

`mvn clean package`. Em seguida, executamos o comando `cf push` para instalar o artefato gerado na plataforma Cloud Foundry.

Pela administração web do Pivotal Web Services é possível verificar se os serviços e aplicações estão disponíveis, como expõe a Figura 8. Se o status estiver em 100%, basta acessar a aplicação Markers UI através de qualquer navegador Internet pelo endereço exposto – neste caso: <http://markers-ui-xxx.cfapps.io>.

| APPLICATIONS Learn More | | | |
|---|--|-----------|--------|
| STATUS | APP | INSTANCES | MEMORY |
| | markers-service markers-service.cfapps.io | 1 | 512MB |
| | markers-ui markers-ui.cfapps.io | 1 | 512MB |

| SERVICE INSTANCE | SERVICE PLAN | BOUND APPS |
|----------------------|---------------|------------|
| markers-ws Delete | User Provided | 1 |

Figura 8. Aplicações em funcionamento no Cloud Foundry

Neste artigo apenas começamos a explorar o Spring Cloud e a descobrir como ele é um grande facilitador para o desenvolvimento de soluções Java nas nuvens. Com o objetivo de manter o foco nesta nova solução Spring, mantemos nossos exemplos e testes apenas no Pivotal Web Services, mas com ela podemos rodar essa mesma aplicação com pouquíssimas alterações em ambientes como Heroku ou mesmo Amazon Web Services, que é um IaaS.

Em plataformas na nuvem, está à nossa disposição uma infinidade de outros serviços, como bancos de dados, servidores de cache e de mensageria. Todos utilizáveis facilmente a partir de integrações que os engenheiros do Spring já implementaram. Ademais, é possível utilizar essas integrações para experimentar uma série de componentes do próprio Cloud Foundry, disponíveis no Marketplace do Pivotal Web Services (veja a Figura 9).

E se você gostou da ideia de desenvolver para a plataforma PaaS com Spring Cloud, saiba que é possível instalar a plataforma Cloud Foundry em qualquer ambiente, seja no seu desktop, no datacenter da sua empresa ou até mesmo na sua conta da Amazon Web Services.

Services Marketplace

Get started with our free marketplace services. Upgrade to gain access to premium service plans.

| | |
|--|--|
| Searchify Custom search you control | BlazeMeter The JMeter Load Testing Cloud |
| Redis Cloud Enterprise-Class Redis for Developers | ClearDB MySQL Database Highly available MySQL for your Apps. |
| CloudAMQP Managed HA RabbitMQ servers in the cloud | ElephantSQL PostgreSQL as a Service |
| SendGrid Email Delivery. Simplified. | MongoLab Fully-managed MongoDB-as-a-Service |
| New Relic Manage and monitor your apps | Load Impact Automated and on-demand performance testing |

Figura 9. Marketplace do Pivotal Web Services

Autor



Leonardo Gonçalves da Silva

leogilva@ok.de

É pós-graduado em Qualidade de Software e atua com arquitetura de sistemas e desenvolvimento na plataforma Java desde 1998. Atualmente trabalha com desenvolvimento Mobile e soluções para Cloud Computing.



Links:

Página do Pivotal Web Services.

<http://run.pivotals.io/>

Site da plataforma Cloud Foundry.

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

Site da plataforma Heroku.

<http://www.heroku.com/>

Site do projeto Apache Maven.

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

Página da Google Maps API – Gmaps.js.

<http://hpneo.github.io/gmaps/>

Página do projeto Feign, cliente HTTP.

<https://github.com/Netflix/feign>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!





DEVMEDIA

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior
portal para
desenvolvedores
da América
Latina!



20
mil
posts

430
mil
cadastrados

10
milhões de
page-views
por mês

Micro serviços RESTful com Spring Boot e Reactor

Como criar um micro serviço baseado em RESTful

Após uma série de escândalos financeiros em meados de 2001, quando grandes empresas americanas fraudaram demonstrações financeiras para encobrir prejuízos – o que fez com que muitos investidores tivessem grandes prejuízos – o congresso americano aprovou uma lei denominada Sarbanes-Oxley. Também conhecida como SOX, esta lei penaliza, criminalmente, executivos de empresas com ações na bolsa de Nova York por fraudes ou desvios em demonstrações financeiras.

Diante deste novo contexto de responsabilidade financeira, houve um significativo aumento na pressão por uma maior transparência nos relacionamentos envolvendo acionistas, conselho, diretoria, auditoria independente e conselho fiscal. Para alcançar tal transparência, as empresas buscaram os princípios de uma metodologia conhecida como Governança Corporativa.

Esta busca alavancou também a adoção dos princípios de outra metodologia, denominada Governança de TI. De fato, a Governança de TI está muito ligada à gestão corporativa, porque sistemas de informação constituem uma peça fundamental na gestão de toda a complexidade encontrada nos processos de uma organização. Deste modo, os holofotes voltaram-se para os departamentos de TI das grandes empresas, gerando uma reflexão sobre como os sistemas controlavam a informação e, principalmente, como trocavam tais informações entre si. A partir daí guias como o COBIT (*Control objectives for information and related technology*) estabeleceram-se dentro das empresas e, analisando sob o âmbito do desenvolvimento de software, conceitos como BPM (*Business Process Management*) e SOA (*Service-Oriented Architecture*) se fortaleceram, ganhando cada vez mais espaço dentro das grandes corporações.

Fique por dentro

A Arquitetura Orientada a Serviços, ou SOA, tornou-se um padrão nas empresas quando existe a necessidade de integração entre sistemas. Sem dúvidas, esse padrão tem sido a base para oferecer às organizações o poder de gerenciar o fluxo de informações e escolher a melhor maneira de tratá-las. Porém, com o advento da Computação em Nuvem, houve um significativo aumento na flexibilidade com que recursos computacionais são disponibilizados às aplicações, facilitando bastante a criação de aplicações distribuídas. Diante disso, surgiu uma nova forma de oferecer serviços dentro do padrão SOA. Conhecido como micro serviços, este novo conceito oferece boas vantagens frente às aplicações monolíticas encontradas nas empresas, principalmente com relação à escalabilidade. Pensando nisso, este artigo tem como objetivo apresentar conceitos e demonstrar a criação de um micro serviço usando ferramentas que têm revolucionado o desenvolvimento ágil contemporâneo: Spring Boot, Reactor e Gradle.

Nesta época, a prática de SOA mostrou-se como uma solução de baixo custo e com maior flexibilidade para integrar os sistemas monolíticos encontrados nas empresas. Além disso, o SOA proporcionava maior proximidade aos requisitos relacionadas à transparência demandados pelas novas políticas corporativas. Outro fato relevante, que contribuiu ainda mais para o desenvolvimento deste estilo arquitetural, foi o progresso de tecnologias relacionadas a web services na primeira década deste século, ocorrido a partir do esforço em conjunto de empresas como Microsoft, Sun, IBM e Oracle.

SOA – Arquitetura Orientada a Serviços

Por definição, Arquitetura Orientada a Serviços é um *design pattern* no qual componentes oferecem serviços a outros componentes através de um protocolo de comunicação. Geralmente, o método de integração usado para esta comunicação é conhecido

como “Integração Horizontal”, ou *Enterprise Service Bus* (ESB), onde há um barramento central responsável por controlar o fluxo de eventos e informações entre os componentes.

O princípio básico da orientação a serviços é ser independente de qualquer software ou tecnologia específica. Sendo assim, SOA não é um padrão de tecnologia enterprise, que depende de um único protocolo de comunicação, como IIOP ou SOAP, ou de uma única linguagem de programação, como Java ou .NET. Pelo contrário, muitas vezes incorpora diferentes tecnologias independente do protocolo de comunicação ou linguagem de programação. O foco da arquitetura SOA é a definição clara e objetiva de contratos de serviços orientados ao negócio, muito mais do que a tecnologias.

Por esse motivo, SOA tem sido, há alguns anos, um padrão de arquitetura predominante em grandes companhias. Isto porque é possível, de forma estruturada, realizar integrações entre múltiplos sistemas heterogêneos, sejam eles de Business Intelligence, BPMs, portais corporativos, RH, ERP, CRM, entre outros.

Em meio a todas essas inovações “arquiteturais” que nasceram dentro das grandes corporações, surgiu um novo elemento que, além de transformar todo o universo da computação contemporânea, também contribuiu para a evolução da arquitetura SOA. Tal elemento é a Computação em Nuvem, ou *Cloud Computing*. A Computação em Nuvem proporcionou o provisionamento dinâmico de recursos computacionais, o que trouxe uma nova perspectiva para a indústria da tecnologia da informação e também alavancou a criação de um novo padrão dentro do conceito de arquitetura orientada a serviços, denominado micro serviços. De acordo com as definições de Anne Thomas, vice-presidente do Gartner Group: “Micro serviços é a prática de aplicar princípios da arquitetura orientada a serviços em um nível de granularidade menor”. Pierre Fricke, diretor de marketing de produto da Red Hat JBoss Middleware, define micro serviços como: “Um nome que denota uma forma mais leve e mais rápida de desenvolver e distribuir do que o SOA tradicional”.

Micro serviços e o princípio da responsabilidade única

De um modo geral, pode-se dizer que micro serviços são softwares independentes com responsabilidades muito bem definidas e que expõem seus comportamentos através de uma API. Em conjunto, podem compor uma ou mais soluções tecnológicas. Uma vez que o princípio básico de um micro serviço é o da Responsabilidade Única (*Single Responsible Principle – SRP*), cada um deve limitar-se a um único escopo de trabalho.

Por exemplo, em uma loja virtual pode-se definir um micro serviço para consulta de CEPs, outro para calcular o frete com base naquele CEP e no peso dos produtos do carrinho de compras, um para processar o pedido e outro para emitir a Nota Fiscal eletrônica. Todos esses micro serviços trabalhando em conjunto determinam o sistema de E-commerce da loja virtual.

Contudo, entre as inúmeras vantagens dessa arquitetura, existe uma que foi especialmente alavancada pela Computação Elástica: a possibilidade de escalar cada micro serviço individualmente. Isto é possível porque um micro serviço possui um contexto indi-

vidual de execução, que pode ser sua própria JVM, ou até mesmo seu próprio servidor virtual dedicado. A Netflix, por exemplo, possui o seu sistema distribuído em micro serviços instalados em múltiplas instâncias da Amazon EC2, e por isso pode ser escalado muito facilmente.

Tal benefício também chamou a atenção de arquitetos Java EE. Isto porque o consumo de CPU do GC (*Garbage Collector*) é proporcional ao tamanho da memória Heap disponibilizada para a JVM. Isto é, o GC irá percorrer toda a memória Heap com o objetivo de eliminar objetos que não estão mais sendo usados. Diante disso, quanto maior a memória Heap, maior o tempo que ele levará para percorrê-la, e com isso, maior será a quantidade de instruções enviadas para a CPU. Sendo assim, dependendo do algoritmo escolhido para o GC e de sua frequência de execução, a opção por memórias Heap muito grandes pode causar contenção de CPU e comprometer todo o sistema. Então, mesmo com servidores com capacidades de memória RAM cada vez maiores, é prudente ter cautela na alocação de mais do que 4GB para a memória Heap da JVM.

Diante deste dilema, a utilização de *load balancers* entre servidores de aplicação tornou-se uma prática cada vez mais comum. Os arquitetos preferiam montar múltiplas instâncias, alocando até 4GB por JVM, do que um único servidor com uma alocação de memória maior. Por esse motivo, em abril de 2012, na *release* do JDK 7 update 4, foi introduzido um novo algoritmo para o GC, desenhado especialmente para situações em que a memória Heap é maior do que 4GB. Este algoritmo foi batizado de *Garbage First Collector*, mas também é conhecido como G1 Collector. Para utilizá-lo, basta incluir a flag `-XX:+UseG1GC` nos parâmetros de inicialização da JVM.

Recentemente, em agosto do ano passado, o Java 8 update 20 trouxe uma boa melhoria para o G1 Collector, chamada G1 Collector String deduplication. Esta identifica todas as *strings* duplicadas na memória Heap e as ajusta para que apontem para o mesmo *array* de `char[]`. Para usar tal recurso, deve-se incluir a flag `-XX:+UseStringDeduplicationJVM` como parâmetro de inicialização da JVM.

Benefícios e inconvenientes dos micro serviços

Entretanto, mesmo com tais melhorias no algoritmo de GC da JVM, muitos arquitetos têm preferido granularizar serviços a fim de oferecer, em conjunto, um poder computacional maior. Ou seja, vêm adotando os princípios de micro serviços. E isto ocorre não apenas pelo motivo que acabamos de citar. Há muitas outras vantagens na escolha pela arquitetura baseada em micro serviços, a saber:

- Cada serviço é desenvolvido e instalado de forma independente, e preferencialmente, deve ser executado sem a necessidade de um servidor de aplicação ou web-container;
- Uma vez que os micro serviços são relativamente pequenos, há uma maior facilidade no entendimento do escopo de cada um. Além disso, não há a sobrecarga na IDE de desenvolvimento por causa de projetos gigantescos e que também aumentam a

complexidade no setup. Isso tudo melhora consideravelmente a produtividade dos desenvolvedores, além de fazer com que novos desenvolvedores se tornem produtivos em menos tempo;

- Já que são instalados de forma independente, há um benefício quando a frequência de instalação de novas versões dos serviços é maior, pois haverá uma redução do tempo de deploy e, consequentemente, do tempo das pausas por indisponibilidade;
- Haverá uma maior facilidade para escalar o desenvolvimento. Por exemplo, é possível distribuir o esforço de desenvolvimento para múltiplos times, onde cada time pode ser responsável por um único micro serviço, construindo e escalando sem qualquer dependência com os demais;
- O isolamento de possíveis falhas que possam ocorrer em um determinado serviço. Por exemplo, se existe algum vazamento de memória em um serviço, apenas este será afetado, diferentemente das aplicações monolíticas, onde um único serviço pode comprometer todo o sistema. Além disso, fica muito mais fácil detectar onde está o problema, já que cada micro serviço pode ser monitorado de forma independente;
- Elimina o compromisso a longo prazo com uma determinada tecnologia ou framework. Em aplicações monolíticas o esforço para troca, ou mesmo upgrade, de um determinado framework, é muito grande, enquanto para um micro serviço esta tarefa é bem menos onerosa.

Em contrapartida, existem alguns inconvenientes que a decisão por uma arquitetura baseada em micro serviços pode trazer, conforme listado a seguir:

- Desenvolvedores devem lidar com itens de maior complexidade ao criar um sistema distribuído:
 - As IDEs foram construídas pensando no desenvolvimento de aplicações monolíticas e não oferecem suporte nativo a aplicações distribuídas;
 - Os testes também podem ser mais difíceis, já que aumenta a quantidade de *mocks* e *fake objects* que devem ser produzidos e mantidos;
 - A implementação de casos de uso que usam vários serviços pode ser mais difícil no ponto de vista transacional. Isto porque é mais complexo montar um contexto transacional em aplicações distribuídas;
 - Deve haver uma coordenação entre times que desenvolvem serviços relacionados a um mesmo caso de uso.
- Aumento da complexidade na instalação do ambiente de produção, já que a operação de instalação do ambiente produtivo dependente de múltiplos serviços exige um maior cuidado do que em ambientes com aplicações monolíticas;
- Há também uma maior alocação de recursos computacionais. Isto porque uma arquitetura baseada em micro serviços substitui N instâncias de um sistema monolítico por NxM instâncias de serviços. Se cada um rodar em sua própria JVM (ou equivalente), é necessária uma quantidade M de runtimes da JVM para cada N instâncias. Ou ainda, se cada micro serviço executar em seu próprio servidor virtual dedicado, por exemplo em uma instância

da Amazon EC2 (como é o caso da Netflix), então o consumo de recursos será ainda maior.

Além disso, ainda existem alguns desafios na adoção da arquitetura baseada em micro serviços. Um deles está relacionado à decisão de qual metodologia empregar para partitionar o sistema em múltiplos serviços. Neste contexto, existem algumas estratégias que podem ajudar. Uma delas é dividir os micro serviços por verbos. Por exemplo, em uma aplicação de e-commerce, pode-se ter um serviço denominado Shipping Service, responsável pela entrega dos pedidos; ou então um serviço chamado Login Service, responsável por implementar o caso de uso de autenticação.

Outra estratégia para divisão dos micro serviços é partitionar os serviços pelos nomes dos recursos e/ou entidades do sistema. Por exemplo, pode-se ter um serviço chamado Inventory Service, que será responsável por todas as operações relacionadas ao estoque dos produtos.

O mais importante é garantir que cada micro serviço possua responsabilidade somente sobre uma pequena parte do sistema, seguindo o conceito de Responsabilidade Única. Uma analogia muito utilizada para exemplificar o conceito de micro serviços é a forma como os aplicativos Unix são desenhados. O Unix provê um grande número de aplicativos utilitários, como *grep*, *cat* e *find*. Cada utilitário faz muito bem uma única coisa, mas quando combinados em um script shell, podem realizar diferentes tarefas e mais complexas. Assim devem ser também os micro serviços. Cada um é responsável por uma tarefa, e quando combinados são capazes de realizar grandes objetivos.

O projeto exemplo

Para apresentar a construção de um micro serviço, será demonstrado neste artigo o desenvolvimento de um serviço responsável por receber uma imagem e gerar a sua versão em miniatura. É fato que processos que envolvem manipulação de imagens costumam ser bem onerosos quanto a recursos computacionais. Deste modo, seria pertinente criar um micro serviço somente para esse tipo de tarefa, assim, em vez de escalar todo o sistema, será possível escalar apenas aquele determinado serviço responsável pela geração da miniatura. Esse projeto exemplo foi inspirado em um dos tutoriais disponíveis no site do projeto Spring IO, e sua versão original pode ser importada diretamente na IDE Spring Tool Suite (STS), através da opção *Import Getting Started Content*.

Para batizar o projeto exemplo será utilizada uma referência a um termo muito comum para denotar miniaturas de imagens, chamado “*thumbnail*”. Traduzindo ao pé da letra, *thumbnail* significa a unha do polegar (*thumb*: polegar, *nail*: unha). Este termo tem sido usado no universo da computação desde os anos 80 para fazer referência a representações gráficas pequenas de imagens maiores ou layouts de página. É uma alusão ao pequeno tamanho da unha do dedão. Sendo assim, o serviço responsável para gerar a miniatura (ou *thumbnail*) da imagem será batizado de *Thumbnailer*, que, em português, poderia ser chamado de “*Miniaturizador*”.

Porém, a escolha do termo em inglês justifica-se pela grande popularidade de *thumbnail* no meio digital.

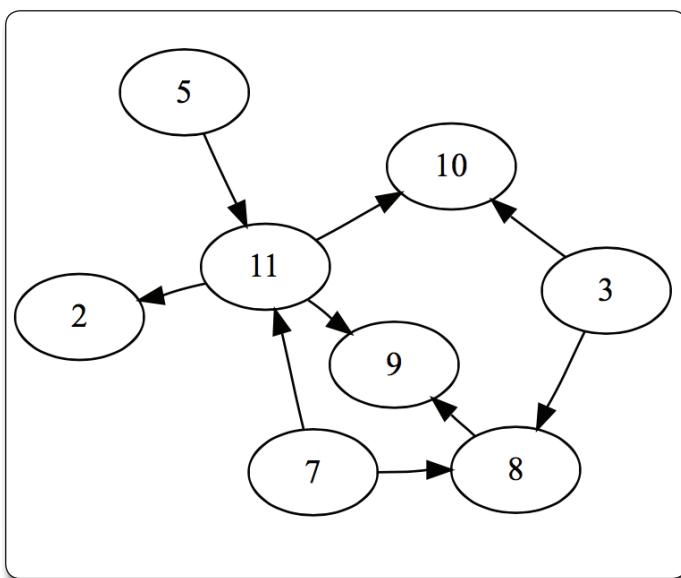
A fim de dar início ao projeto exemplo, vamos primeiro criar a estrutura de pastas. Para isso, será adotada a convenção de projetos definida pelo Maven, ou seja, todo o código dentro de uma pasta *src*, separando tudo o que é código Java na pasta *src/main/java* e tudo o que não é código Java na pasta *src/main/resources*. O nome da pasta raiz do projeto será “*thumbnailer-service*” e todas as classes serão colocadas em um pacote denominado **br.com.devmedia**. Definidos estes pontos, pode-se utilizar os comandos da **Listagem 1** para criação da estrutura do projeto.

Listagem 1. Criação das pastas do projeto.

```
01 mkdir -p thumbnailer-service/src/main/java;br/com/devmedia  
02 mkdir -p thumbnailer-service/src/main/resources/
```

Gradle: gerenciando dependências de forma ágil

A ferramenta de automação escolhida para construção do projeto será o Gradle. Tal ferramenta foi concebida a partir dos princípios do Apache Ant e do Apache Maven e introduz um conceito baseado em Groovy, denominado *Domain Specific Language* (DSL), para a forma como são declaradas as configurações do projeto, em contraste ao tradicional estilo baseado em XML. Diferente do Maven, que define ciclos de vida (*lifecycles*) dos projetos, e do Ant, onde as tarefas são chamadas ordenadamente a partir de uma árvore de dependências, o Gradle utiliza o conceito de *Directed Acyclic Graph* (DAG) para determinar a ordem na qual as tarefas são executadas. Este conceito é baseado em grafos, onde trechos de informações são organizadas em nós ligados por vértices, sendo que para qualquer vértice v , não há como percorrer nenhum caminho dirigido que comece e acabe em v , ou seja, trata-se de um grafo sem ciclo, como mostra a **Figura 1**.



Entre as vantagens apresentadas pelo Gradle está o suporte para *build incremental*, que inclui uma inteligência para determinar quais partes da árvore de compilação estão atualizadas, para que cada tarefa dependente de tais partes não seja executada novamente. Até por isso o Gradle tem sido muito adotado pelos adeptos do desenvolvimento ágil. Tal ferramenta foi desenvolvida focada inicialmente para Java, Groovy e Scala, porém outras linguagens de programação estão no *roadmap* da comunidade. Entre as empresas pioneiras no uso do Gradle, estão a Netflix e o LinkedIn.

Para instalar o Gradle, basta ir ao site oficial (veja a seção **Links**) e baixar a versão mais atual. Feito isso, é só adicionar a pasta *bin* ao *path* do sistema operacional. Deste modo o comando principal, denominado *gradle*, já estará disponível via *shell*. Caso queira usar o Gradle a partir de uma IDE, existe um plugin para o Eclipse disponível no *Eclipse Marketplace*. Outra alternativa seria usar o *Spring Tool Suite* (STS), baseado no Eclipse, que também oferece suporte a esta ferramenta.

Uma vez determinada a IDE e criada a estrutura de pastas, deve-se definir as configurações do projeto. Para isto, crie um arquivo chamado *build.gradle* na pasta raiz. O conteúdo deste arquivo é apresentado na **Listagem 2**.

Listagem 2. Conteúdo do arquivo build.gradle.

```
01 ext {  
02   reactorVersion = '1.1.0.M3'  
03 }  
04  
05 apply plugin:'java'  
06 apply plugin:'eclipse'  
07 apply plugin:'idea'  
08 apply plugin:'spring-boot'  
09  
10 sourceCompatibility = 1.8  
11 targetCompatibility = 1.8  
12  
13 buildscript {  
14   ext {  
15     springBootVersion = '1.2.3.RELEASE'  
16   }  
17   repositories {  
18     mavenCentral()  
19   }  
20   dependencies {  
21     classpath("org.springframework.boot:spring-boot-gradle-plugin:$  
22     {springBootVersion}")  
23   }  
24  
25 jar {  
26   baseUrl = 'thumbnailer'  
27   version = '0.0.1-SNAPSHOT'  
28 }  
29  
30 repositories {  
31   mavenCentral()  
32   maven { url "https://repo.spring.io/libs-milestone/" }  
33 }  
34  
35 dependencies {  
36   compile "org.springframework.boot:spring-boot-starter"  
37   compile "org.projectreactor:reactor-net:$reactorVersion"  
38   compile "org.projectreactor.spring:reactor-spring-context:$reactorVersion"  
39 }
```

Ao analisar este código, nota-se que as configurações são organizadas em trechos bem definidos entre chaves ({}). As primeiras três linhas mostram a declaração do que são chamadas de “extra properties”, ou propriedades extras, identificadas pela área cujo *namespace* é chamado de “ext”. Essas propriedades são usadas para especificação de atributos customizados. Neste caso, estamos definindo uma variável para armazenar a versão de uma das dependências que serão usadas no projeto, o reactor-spring-context (mais adiante falaremos sobre essa dependência).

Em seguida, nas linhas 05 a 08, são declarados os plug-ins a serem usados pelo Gradle, cujo conceito é o mesmo dos plug-ins existentes para o Maven. Na linha 05 é declarado o plugin “java”, que define o compilador Java para construção do projeto. Já nas linhas 06 e 07, são declarados os plugin “eclipse” e “idea”, responsáveis por formatar o projeto de uma maneira que possam ser importados pelas IDEs Eclipse e IntelliJ IDEA, respectivamente. Com o plugin do Eclipse, por exemplo, é possível executar o comando *gradle eclipse* na raiz do projeto. Isso irá criar os arquivos *.classpath* e *.project* e a pasta *.settings*, ou seja, os artefatos específicos de configuração do Eclipse para que o projeto esteja pronto para importação.

Aplicações “stand-alone” com Spring Boot

A linha 08, por sua vez, declara o plug-in “spring-boot”, que é responsável por oferecer suporte à construção de projetos que fazem uso do Spring Boot. O Spring Boot é um projeto da SpringSource que se baseia no conceito de “convenção sobre configuração” (*convention-over-configuration*), usado principalmente em metodologias de desenvolvimento ágil. Uma das vantagens na utilização do Spring Boot é oferecer a opção para criação de aplicações “stand-alone”, isto é, um pacote executável que não precisa necessariamente de um container específico ou de um servidor de aplicação para ser executado. Por exemplo, é possível gerar um JAR que pode inicializar a aplicação apenas executando o comando *java -jar*. Isso vai ao encontro dos princípios básicos de uma aplicação criada para ser um micro serviço.

Para usar o plug-in para Spring Boot no Gradle, além da definição do mesmo conforme a linha 08, deve-se ainda declarar a dependência à biblioteca “spring-boot-gradle-plugin”, como mostra a linha 21. Sobre essa dependência especificamente, vale notar que ela é declarada em outro trecho do arquivo de configuração, que não é o mesmo trecho onde são declaradas as outras dependências do projeto (*namespace dependencies*, entre as linhas 35 a 39). Esse trecho é denominado “buildscript” (linha 13) e a dependência foi colocada ali porque não há necessidade de que seja incluída no pacote final, ou seja, trata-se de uma biblioteca usada somente para a construção do projeto e não para a execução do mesmo. Este *namespace*, *buildscript*, tem justamente a função de definir configurações específicas apenas para o processo de *build*. Note ainda que também é possível definir propriedades extras para cada *namespace*, como ocorre na linha 15, onde a versão do plug-in Spring Boot é incluída em uma variável para ser usada na linha 21.

Outra vantagem do plug-in do Spring Boot é a customização do recurso *ResolutionStrategy* do Gradle. Com essa customização, não é necessário incluir as versões dos artefatos considerados como “abençoados” (*blessed*) pelo Spring Boot. Assim, o plug-in se encarregará de trazer sempre a versão mais atualizada compatível. Por exemplo, não está sendo informada a versão para a biblioteca “spring-boot-starter”, declarada como dependência na linha 36.

Ainda sobre a “spring-boot-starter”, ela refere-se a um conceito muito interessante introduzido pelo Spring Boot: a ideia dos “starters”. Estes são bibliotecas de apoio que já oferecem suporte “on-the-fly” a diversas tecnologias e frameworks, ou seja, resolvem dinamicamente as dependências e versões que devem ser incorporadas ao projeto. Por exemplo, caso seja necessário usar Spring e JPA para acesso ao banco de dados, basta incluir a dependência “spring-boot-starter-data-jpa”; se for necessário suporte ao *javax.mail*, basta incluir “spring-boot-starter-mail”. A dependência será sempre no formato “spring-boot-starter-***”, onde “***” é um tipo particular de starter que deseja-se utilizar. No caso da dependência “spring-boot-starter”, ela inclui o núcleo do Spring Boot starter, com o suporte para a autoconfiguração e *logging*.

Ao final da sessão de dependências do projeto, as linhas 37 e 38 adicionam duas dependências referentes à biblioteca Reactor para JVM. A primeira, “reactor-net”, contém a implementação principal para uso do Reactor, e a segunda, “reactor-spring-context”, é a integração da implementação principal com o contexto de aplicação do Spring Framework.

Reactor: framework ou design pattern?

Antes de falar sobre a biblioteca Reactor, é preciso entender o conceito do *design pattern* Reactor (*Reactor Pattern*). Este padrão define um modelo para gerenciar eventos assíncronos entregues por requisições simultâneas através de múltiplas entradas, ou *inputs*. De acordo com o padrão, o serviço responsável por manipular tais requisições deve de-multiplexar as informações e então dispará-las de forma sincronizada para o seu respectivo serviço associado. Inspirado neste padrão, a Spring IO Platform concebeu o projeto para construção da biblioteca Reactor, a qual implementa uma especificação denominada Reactive Streams. Tal especificação foi criada a fim de definir uma melhor maneira de gerenciar o fluxo de cadeias de dados de forma assíncrona.

O objetivo do Reactor é oferecer suporte ao desenvolvimento de aplicações que necessitam de uma alta taxa de transferência, com baixa latência e que devem funcionar, conforme descrito no blog do Spring IO: “com milhares, dezenas de milhares e até milhões de requisições concorrentes por segundo”. De fato, os relatos no blog indicam que o Reactor pode processar cerca de 10 a 15 milhões de eventos por segundo em um laptop convencional para desenvolvimento. Em um processador de alta performance, é possível encadear até 100 milhões de eventos por segundo. É claro que uma aplicação não dependerá apenas do processamento de eventos pelo Reactor, mas também das tarefas realizadas sobre os dados trafegados. O que torna relevante o uso do Reactor em um projeto de micro serviços é o fato de que este framework

permite escalar o processamento dos eventos, os quais são a base para a comunicação entre as múltiplas interfaces de aplicações distribuídas.

Para fechar a explicação sobre o arquivo *build.gradle*, nas linhas 10 e 11 há a definição da versão do JDK que será usado para compilar e validar o código fonte. Já nas linhas 25 a 28 são definidos o nome e a versão do JAR que será criado após a compilação. E nas linhas 30 a 33, são especificados os repositórios nos quais serão feitas as buscas pelas bibliotecas dependentes.

Implementação

Configurado este arquivo, já é possível criar a primeira classe do micro serviço, que será chamada de **BufferedImageThumbnailer**. Esta classe será responsável por receber o caminho da imagem que foi enviada através de um POST e redimensioná-la usando a classe **Graphics2D** da biblioteca AWT. Sendo assim, crie uma classe chamada **BufferedImageThumbnailer** na pasta */src/main/java/br/com/devmedia/*. O seu código é mostrado na **Listagem 3**.

Perceba que esta classe é uma implementação da interface **reactor.function.Function<T, R>**. Tal interface é responsável por definir a tarefa a ser executada após o Reactor enviar a requisição. A vantagem de construir essa integração usando interfaces é que o código ficará totalmente desacoplado, ou seja, pode ser possível, em tempo de execução, alternar entre as implementações de **Function**. Por exemplo, será possível trocar para outra implementação que faça o redimensionamento de imagens usando qualquer outra biblioteca.

As implementações da interface **Function<T, R>** devem executar uma tarefa através do método **apply()**, que recebe um parâmetro **T** e retorna um parâmetro **R**, que pode ou não ser um tipo diferente de **T**. Para essa implementação (**BufferedImageThumbnailer**), os modificadores genéricos para a interface **Function<T, R>** são **Event** e **Path**, respectivamente, como mostra a linha 5. A classe **Path (java.nio.file.Path)** representa a localização de um arquivo no sistema de arquivos do SO e **Event (reactor.event.Event<T>)** é usada pelo Reactor para empacotar informações que serão processadas pelos consumidores dentro do próprio framework, de modo similar ao *design pattern* Observer.

No caso deste projeto, o **Event** também possui um modificador genérico do tipo **Path**, que será o caminho físico da imagem que foi enviada através da requisição. Portanto, o método **apply()** irá receber um **Event<Path>**, que é um evento com o caminho do arquivo, e retornar um **Path** com o caminho da imagem já redimensionada apropriadamente. Nota-se também que o construtor da classe, iniciado na linha 14, recebe um parâmetro do tipo inteiro, denominado **maxLongSide**. Este é usado para definir para qual tamanho o lado maior da figura será redimensionado, e então o lado menor será calculado proporcionalmente com base neste tamanho. Isso flexibiliza o miniaturizador para manter a proporção em caso de imagens no formato retrato (*portrait*) ou paisagem (*landscape*).

Note que não há muito mais a ser dito sobre a classe **BufferedImageThumbnailer**. Seu escopo de trabalho é bem simples e a

forma como a imagem é redimensionada usando a Java Graphics 2D, através da implementação no método **apply()**, não tem nada de especial.

Listagem 3. Código da classe **BufferedImageThumbnailer**.

```
01 package br.com.devmedia;
02
03 // imports omitidos...
04
05 public class BufferedImageThumbnailer implements Function<Event<Path>, Path> {
06
07     private static final ImageObserver DUMMY_OBSERVER = (img, infoflags, x, y,
08             width, height) -> true;
09
10    private final Logger log = LoggerFactory.getLogger(getClass());
11
12    private final int maxLongSide;
13
14    public BufferedImageThumbnailer(int maxLongSide) {
15        this.maxLongSide = maxLongSide;
16    }
17
18    @Override
19    public Path apply(Event<Path> ev) {
20        try {
21            Path srcPath = ev.getData();
22            Path thumbnailPath = Files.createTempFile("thumbnail", ".jpg")
23                .toAbsolutePath();
24            BufferedImage imgIn = ImageIO.read(srcPath.toFile());
25
26            double scale;
27            if (imgIn.getWidth() >= imgIn.getHeight()) {
28                scale = Math.min(maxLongSide, imgIn.getWidth())
29                    / (double) imgIn.getWidth();
30            } else {
31                scale = Math.min(maxLongSide, imgIn.getHeight())
32                    / (double) imgIn.getHeight();
33            }
34
35            BufferedImage thumbnailOut = new BufferedImage(
36                (int) (scale * imgIn.getWidth()),
37                (int) (scale * imgIn.getHeight()), imgIn.getType());
38            Graphics2D g = thumbnailOut.createGraphics();
39
40            AffineTransform transform = AffineTransform.getScaleInstance(scale,
41                scale);
42            g.drawImage(imgIn, transform, DUMMY_OBSERVER);
43            ImageIO.write(thumbnailOut, "jpeg", thumbnailPath.toFile());
44
45            log.info("Thumbnail da imagem agora em: {}", thumbnailPath);
46
47            return thumbnailPath;
48        } catch (Exception e) {
49            throw new IllegalStateException(e.getMessage(), e);
50        }
51    }
52}
```

Para prosseguir com o projeto, podemos construir a classe que será responsável por fazer a camada REST do serviço. Esta será batizada de **ImageThumbnailerRestApi**. Na verdade, esta é uma classe Helper, ou seja, nunca será instanciada, e seus métodos serão usados de forma estática. Tais métodos serão responsáveis pelo algoritmo a ser executado sobre as requisições HTTP.

Micro serviços RESTful com Spring Boot e Reactor

Sendo assim, crie esta classe no caminho `src/main/java/br/com/devmedia`. O seu código é mostrado na **Listagem 4** e 5.

Para analisar o código da classe **ImageThumbnailerRestApi**, começaremos pelo método `thumbnailImage()`, iniciado na linha 11.

Este método é o responsável por publicar o evento da “miniatização” da imagem no barramento de eventos do Reactor. Ele trata a requisição HTTP e retorna uma instância do tipo **Consumer (reactor.function.Consumer<T>)**. Como o próprio

Listagem 4. Código da classe `ImageThumbnailerRestApi`, responsável por tratar as requisições HTTP – Parte 1.

```
01 package br.com.devmedia;
02
03 //imports omitidos...
04
05 public class ImageThumbnailerRestApi {
06
07     public static final String IMG_THUMBNAIL_URI = "/image/thumbnail.jpg";
08
09     public static final String THUMBNAIL_REQ_URI = "/thumbnail";
10
11     public static Consumer<FullHttpRequest> thumbnailImage(
12         NetChannel<FullHttpRequest, FullHttpResponse> channel,
13         AtomicReference<Path> thumbnail, Reactor reactor) {
14
15         return req -> {
16             if (req.getMethod() != HttpMethod.POST) {
17                 channel.send(badRequest(req.getMethod())
18                     + " não suportado para esta URI");
19             }
20         }
21
22         Path imgIn = null;
23
24         try {
25             imgIn = readUpload(req.content());
26         } catch (IOException e) {
27             throw new IllegalStateException(e.getMessage(), e);
28     }
29
30     reactor.sendAndReceive("thumbnail", Event.wrap(imgIn), ev -> {
31         thumbnail.set(ev.getData());
32         channel.send(redirect());
33     });
34 }
35 }
36
37 public static Consumer<FullHttpRequest> serveThumbnailImage(
38     NetChannel<FullHttpRequest, FullHttpResponse> channel,
39     AtomicReference<Path> thumbnail) {
40
41     return req -> {
42         if (req.getMethod() != HttpMethod.GET) {
43             channel.send(badRequest(req.getMethod()
44                     + " não suportado para esta URI"));
45         } else {
46             try {
47                 channel.send(servelImage(thumbnail.get()));
48             } catch (IOException e) {
49                 throw new IllegalStateException(e.getMessage(), e);
50             }
51         }
52     };
53 }
```

Listagem 5. Código da classe `ImageThumbnailerRestApi`, responsável por tratar as requisições HTTP – Parte 2.

```
55     public static Consumer<Throwable> errorHandler(
56         NetChannel<FullHttpRequest, FullHttpResponse> channel) {
57
58         return ev -> {
59             DefaultFullHttpResponse resp = new DefaultFullHttpResponse(
60                 HttpVersion.HTTP_1_1,
61                 HttpResponseStatus.INTERNAL_SERVER_ERROR);
62             resp.content().writeBytes(ev.getMessage().getBytes());
63             resp.headers().set(HttpHeaders.Names.CONTENT_TYPE, "text/plain");
64             resp.headers().set(HttpHeaders.Names.CONTENT_LENGTH,
65                 resp.content().readableBytes());
66             channel.send(resp);
67     };
68 }
69
70     private static Path readUpload(ByteBuf content) throws IOException {
71         byte[] bytes = new byte[content.readableBytes()];
72         content.readBytes(bytes);
73         content.release();
74
75         Path imgIn = Files.createTempFile("upload", ".jpg");
76         Files.write(imgIn, bytes);
77
78         imgIn.toFile().deleteOnExit();
79
80         return imgIn;
81     }
82
83     public static FullHttpResponse badRequest(String msg) {
84
85         DefaultFullHttpResponse resp = new DefaultFullHttpResponse(HTTP_1_1,
86             BAD_REQUEST);
87         resp.content().writeBytes(msg.getBytes());
88         resp.headers().set(CONTENT_TYPE, "text/plain");
89         resp.headers().set(CONTENT_LENGTH, resp.content().readableBytes());
90     }
91
92     public static FullHttpResponse redirect() {
93         DefaultFullHttpResponse resp = new DefaultFullHttpResponse(HTTP_1_1,
94             MOVED_PERMANENTLY);
95         resp.headers().set(CONTENT_LENGTH, 0);
96         resp.headers().set(LOCATION, IMG_THUMBNAIL_URI);
97     }
98
99
100    public static FullHttpResponse servelImage(Path path) throws IOException {
101        DefaultFullHttpResponse resp = new DefaultFullHttpResponse(HTTP_1_1, OK);
102
103        RandomAccessFile file = new RandomAccessFile(path.toString(), "r");
104        resp.headers().set(CONTENT_TYPE, "image/jpeg");
105        resp.headers().set(CONTENT_LENGTH, file.length());
106
107        byte[] bytes = Files.readAllBytes(path);
108        resp.content().writeBytes(bytes);
109
110        return resp;
111    }
112}
```

nome da classe indica, esta instância de **Consumer** será a consumidora do evento enviado pelo barramento. Pode-se notar que o modificador genérico definido para essa instância de **Consumer** é do tipo `io.netty.handler.codec.http.FullHttpRequest`, que é uma classe pertencente ao framework Netty.

O Netty é uma solução que implementa um conector HTTP embutido (entre outros protocolos), assim como o Jetty ou o Tomcat, porém voltado a eventos assíncronos e desenhado especificamente para desenvolvimento ágil e alta performance. O que justifica a adoção do Netty em um micro serviço é que ele faz uso extensivo do Java NIO (*Java Non Blocking IO*), evitando o problema de espera pela resposta nas tradicionais conexões socket, e ainda usa multiplexação (*multiplexing*) para permitir o atendimento de múltiplas requisições por um número reduzido de threads.

Além do retorno `Consumer<FullHttpRequest>`, o método `thumbnailImage()` recebe como parâmetro objetos do tipo `NetChannel<FullHttpRequest, FullHttpResponse>`, `AtomicReference<Path>` e `Reactor`. O objeto `channel`, do tipo `NetChannel`, é responsável por manter um canal de comunicação com o socket. Ele será usado para enviar mensagens HTTP direto para o cliente. Já o objeto `thumbnail`, do tipo `AtomicReference`, contém uma referência atômica para o caminho da imagem que foi enviada no request via POST. E por último, o objeto do tipo `Reactor`, que é o barramento de eventos do framework.

Ao analisar os detalhes da implementação do método `thumbnailImage()`, especificamente entre as linhas 16 e 20, nota-se o teste para verificar se o método enviado pelo *request* é mesmo do tipo POST. Caso não seja, a execução continua na linha 17 e chama o método `badRequest()` (declarado na linha 83), que é responsável por escrever uma mensagem HTTP do tipo "BAD REQUEST" (código 400) no objeto `response (FullHttpResponse)`. Tal objeto é enviado para o cliente através do `channel (NetChannel)`, incluindo a informação de que o método não é suportado (linhas 43 e 44). Portanto, se o cliente enviar a imagem via outro método que não seja POST, por exemplo um PUT, ele receberá a mensagem de *bad request* (400).

Seguindo com a implementação, na linha 25 existe a tratativa para ler o conteúdo do *request*, que será uma cadeia de bytes da imagem que desejassem miniaturizar. Através do método `readUpload()` (linhas 70 a 81) a imagem é gravada em disco, especificamente na pasta temporária usada pela JVM, conforme mostram as linhas 75 e 76. Logo após apontar o caminho da imagem na variável `imgIn`, pode-se então publicar o evento no barramento do Reactor. Conforme mostram as linhas 30 a 33, a publicação do evento é feita através do uso do método `sendAndReceive()` de `reactor`.

Esse método recebe como parâmetro a chave de identificação do evento, que estamos chamando de "`thumbnail`". Em seguida, recebe um objeto do tipo `Event`, que empacotará os bytes da imagem como sua própria carga. E por fim, recebe um objeto do tipo `Consumer`, que é quem será notificado quando o pro-

cessamento da imagem estiver finalizado. Na implementação de **Consumer**, conforme as linhas 31 e 32, é definido o path da miniatura na referência atômica `thumbnail` e, em seguida, é enviado ao canal um objeto `response (FullHttpResponse)` criado pelo método `redirect()` (linha 92). Tal método é responsável por instanciar um objeto do tipo `FullHttpResponse`, escrever uma mensagem HTTP do tipo 301 (*Moved Permanently*) e informar a URI da miniatura em seu cabeçalho.

Continuando com os outros métodos desta classe, na linha 37 encontra-se o método `serveThumbnailImage()`, que possui uma assinatura semelhante ao `thumbnailImage()`, porém não possui o parâmetro do tipo `Reactor`, uma vez que não publicará nenhum evento para o barramento. O objetivo desta implementação é retornar para o canal de ligação com o socket (`channel`) a imagem miniaturizada. Nota-se que na linha 42 é realizado um teste semelhante ao da linha 16, só que desta vez, ele só aceitará requisições do tipo GET, caso contrário enviará também o "*bad request*" para o canal.

Na linha 47, após a implementação ter se assegurado que se trata de um método GET, é enviado ao canal o `response (FullHttpResponse)` contendo os bytes da imagem miniaturizada. Neste ponto é possível perceber a utilização do método `serveImage()` para buscar a imagem no disco e escrever seus bytes no conteúdo do `response (FullHttpResponse)`. A implementação deste método está presente entre as linhas 100 e 111.

Após a análise da classe `Helper` responsável pelo tratamento das requisições REST, podemos agora prosseguir para a classe principal do micro serviço, denominada `ThumbnailerServiceApplication`. Para isso, crie esta classe na pasta do pacote principal da aplicação (`src/main/java/br/com/devmedia/`). O seu código é apresentado na [Listagem 6](#).

A primeira observação a fazer sobre a classe `ThumbnailerServiceApplication` está relacionada às suas anotações. A primeira, presente na linha 5, denominada `@EnableAutoConfiguration`, é responsável por habilitar um recurso do Spring que tenta descobrir e configurar, a partir do seu `classpath`, os `beans` que poderão ser usados pela aplicação. Por exemplo, se incluirmos um artefato `tomcat-embedded.jar` no `classpath`, então é muito provável que será usado um `bean` do tipo `TomcatEmbeddedServletContainerFactory`, já que esta classe é responsável por construir uma instância do Tomcat embutido e não teria sentido incluir tal artefato que não fosse o de usar tal instância. Por isso o Spring, através de seu container IOC, já "autoconfigura" tal `bean`.

A segunda anotação, denominada `@Configuration` (linha 6), indica que a `ThumbnailerServiceApplication` (linha 9) é uma classe que poderá declarar `beans` do Spring Framework, através de métodos anotados com `@Bean`. Prosseguindo com a terceira anotação, denominada `@ComponentScan` (linha 07), ela é responsável por varrer o código da aplicação em busca de outros `beans`. Trata-se da versão em anotação da tag `<context:component-scan>`. Com relação à quarta anotação, chamada `@EnableReactor` (linha 08), ela pertence ao framework

Micro serviços RESTful com Spring Boot e Reactor

Listagem 6. Código da classe ThumbnailerServiceApplication.

```
01 package br.com.devmedia;
02
03 //imports omitidos...
04
05 @EnableAutoConfiguration
06 @Configuration
07 @ComponentScan
08 @EnableReactor
09 public class ThumbnailerServiceApplication {
10
11     @Bean
12     public Reactor reactor(Environment env) {
13         Reactor reactor = Reactors.reactor(env, Environment.THREAD_POOL);
14         reactor.receive("${'thumbnail"}", new BufferedImageThumbnailer(250));
15         return reactor;
16     }
17
18     @Bean
19     public ServerSocketOptions serverSocketOptions() {
20         return new NettyServerSocketOptions()
21             .pipelineConfigurer(pipeline -> pipeline.addLast(
22                 new HttpServerCodec()).addLast(
23                 new HttpObjectAggregator(16 * 1024 * 1024)));
24     }
25
26     @Bean
27     public NetServer<FullHttpRequest, FullHttpResponse> restApi(
28         Environment env, ServerSocketOptions opts, Reactor reactor,
29         CountDownLatch closeLatch) throws InterruptedException {
30         AtomicReference<Path> thumbnail = new AtomicReference<>();
31
32         NetServer<FullHttpRequest, FullHttpResponse>
33         server = new TcpServerSpec<FullHttpRequest, FullHttpResponse>(
34             NettyTcpServer.class)
35             .env(env)
36             .dispatcher("sync")
37             .options(opts)
38             .listen(tcpPort)
39             .consume(
40                 ch -> {
41                     Stream<FullHttpRequest> in = ch.in();
42                     in.filter(
43                         (FullHttpRequest req) -> ImageThumbnailerRestApi
44                             .equals(req.getUri())))
45                         .when(Throwable.class,
46                             ImageThumbnailerRestApi
47                             .errorHandler(ch))
48                         .consume(
49                             ImageThumbnailerRestApi
50                             .serveThumbnailImage(ch,
51                             thumbnail));
52                     in.filter(
53                         (FullHttpRequest req) -> ImageThumbnailerRestApi
54                             .THUMBNAIL_REQ_URI
55                             .equals(req.getUri())))
56                         .when(Throwable.class,
57                             ImageThumbnailerRestApi
58                             .errorHandler(ch))
59                         .consume(
60                             ImageThumbnailerRestApi
61                             .thumbnailImage(ch,
62                             thumbnail, reactor));
63                     in.filter(
64                         (FullHttpRequest req) -> "/shutdown"
65                             .equals(req.getUri())).consume(
66                             req -> closeLatch.countDown());
67                     }).get();
68                 server.start().await();
69             return server;
70     }
71     @Bean
72     public CountDownLatch closeLatch() {
73         return new CountDownLatch(1);
74     }
75     public static void main(String... args) throws InterruptedException {
76         ApplicationContext ctx = SpringApplication.run(
77             ThumbnailerServiceApplication.class, args);
78         CountDownLatch closeLatch = ctx.getBean(CountDownLatch.class);
79         closeLatch.await();
80     }
81
82     @Value("${netty.listen.port}")
83     private Integer tcpPort;
84 }
```

Reactor é responsável por criar o contexto deste framework, bem como os *beans* necessários para seu funcionamento. Uma observação importante é que esta anotação só deve ser usada em classes anotadas com **@Configuration**.

Após a descrição das anotações presentes na classe **ThumbnailerServiceApplication**, é possível prosseguir para uma análise geral da sua estrutura. Veja que são declarados quatro *beans*, que serão detalhados mais adiante neste artigo, dos seguintes tipos: **Reactor** (linha 12), **ServerSocketOptions** (linha 19), **NetServer** (linha 27) e **CountDownLatch** (linha 71). Em

seguida, na linha 75, encontramos o tradicional método **main()**, que será o primeiro a ser executado no início da aplicação (mais adiante também falaremos sobre ele). Finalmente, é declarada a propriedade **tcpPort** na linha 83. Esta propriedade será responsável por definir qual porta TCP o Netty irá escutar. Veja que ela está parametrizada, ou seja, seu valor será carregado a partir do arquivo *application.properties*.

A vantagem de parametrizar essa propriedade é que será possível executar múltiplas instâncias da aplicação no mesmo servidor, sendo que cada instância escuta em uma porta diferente.

Para isso, crie o arquivo *application.properties* na pasta */src/main/resources* e adicione o conteúdo exposto a seguir:

```
netty.listen.port=9090
```

Note que a única propriedade declarada é a *netty.listen.port*, com o valor **9090**, ou seja, o servidor irá escutar na porta 9090. Essa propriedade é carregada a partir da anotação *@Value* na linha 82 da **Listagem 6**, passando apenas o nome da propriedade como parâmetro. Este é mais um benefício da convenção sobre configuração do Spring Boot.

Com relação ao *bean* do tipo **Reactor**, criado pelo método *reactor()* (vide [linha 12](#)), trata-se do barramento de eventos do framework Reactor. Veja que, após a construção do objeto, na linha 13, é registrada uma nova instância do tipo **Function** no barramento, cujo identificador é denominado **thumbnail**. No caso deste projeto, a implementação escolhida é a da classe **BufferedImageThumbnailer**. Note que o parâmetro para construção desta é o inteiro **250**, que define o valor em pixels do maior lado da miniatura que desejasse criar.

Logo em seguida, na linha 19, é declarado o método *serverSocketOptions()*, que cria um *bean* do tipo **ServerSocket-**

Options, responsável por configurar o canal de comunicação via socket. Neste caso, é adicionado ao pipeline de configuração um codec padrão para o tratamento de mensagens HTTP, através de um objeto **HttpServerCodec** (linha 22). Além disso, também é incluído um *handler* do tipo **HttpObjectAggregator** (linha 23) que define o tamanho máximo das mensagens, no caso 16 MB ($16 * 1024 * 1024$), ou seja, não será possível enviar uma imagem maior do que 16 megabytes, senão será lançada uma exceção do tipo **TooLongFrameException**.

Uma vez construído o barramento de eventos, provido pelo *bean* **Reactor**, e definidas as configurações padrões do *socket*, através do *bean* **ServerSocketOptions**, podemos construir o *bean* **NetServer**. Este será responsável por servir as conexões TCP e é criado através do método *restApi()*, na linha 27. Ao analisar a implementação deste método, note que na linha 30 é declarada uma variável denominada **thumbnail**, responsável por armazenar uma referência atômica para o caminho da imagem e que será usada mais adiante nesta implementação. Na linha 32, nota-se a criação do objeto **server** do tipo **TcpServerSpec**, que implementa a interface **NetServer**. Veja que o construtor de **TcpServerSpec** recebe um parâmetro do tipo **class** (linhas 32 e 33) e neste caso está recebendo a classe **NettyTcpServer**,

CURSOS ONLINE

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



CONHEÇA OS CURSOS MAIS RECENTES:

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

Para mais informações :

<http://www.devmedia.com.br/cursos/banco-de-dados>

(21) 3382-5038

Micro serviços RESTful com Spring Boot e Reactor

que também faz parte do framework Reactor e é usada para construir uma instância de um servidor do tipo Netty. Em seguida, nas linhas 34 a 36, são especificadas as propriedades gerais do objeto `server`, e na linha 37, é definida a porta na qual o servidor receberá as requisições; neste caso, está sendo usada a propriedade `tcpPort`, já citada anteriormente, que é definida no arquivo de propriedades da aplicação.

Finalmente, na linha 38, é chamado o método `consume()`, que define o algoritmo a ser executado quando a requisição chegar através do canal, da mesma forma que uma função `callback`. É possível observar na linha 40 que a variável `in` armazena as requisições TCP em forma de cadeia de dados ou *stream*. E nas linhas 41, 51 e 61, é realizado um filtro nesta variável, testando a URI de cada requisição e chamando o método correspondente do helper `ImageThumbnailerRestApi`, seja para obter a URL da miniatura, gerar a miniatura e, no último teste, realizar a parada da aplicação, caso o usuário faça uma requisição a `/shutdown`.

```
:compileJava  
:processResources  
:classes  
:jar  
:findMainClass  
:startScripts  
:distTar  
:distZip  
:bootRepackage  
:assemble  
:compileTestJava UP-TO-DATE  
:processTestResources UP-TO-DATE  
:testClasses UP-TO-DATE  
:test UP-TO-DATE  
:check UP-TO-DATE  
:build  
  
BUILD SUCCESSFUL
```

Figura 2. Output da execução da construção do projeto

Para concluir a análise da classe `ThumbnailerServiceApplication`, observamos na linha 75 o método `main()`. Basicamente, ele cria um novo contexto do Spring, como visto nas linhas 76 e 77, e nas linhas 78 e 79, percebemos o uso da classe `CountDownLatch`, do pacote `java.util.concurrent`. Esta classe é usada quando queremos segurar uma thread ativa, que pode receber notificações de outras threads de forma assíncrona. É possível controlar quando esta thread será finalizada a partir de um contador. Quando este chegar a zero, a thread é finalizada. O uso deste recurso se faz necessário para manter ativa a thread principal, já que servidor TCP do Reactor é non-blocking, ou seja, irá liberar as threads que o chamaram. Então, é criado um *bean* do tipo `CountDownLatch`, que recebe um contador com valor “1”, conforme o método `closeLatch()` na linha 71.

De acordo com a linha 79, esse *bean* é colocado em espera no método `main()` para que segure a thread principal. A partir daí, toda vez que é chamado o método `countDown()` nesse *bean*, o contador é decrementado, e quando chegar a zero, a thread é finalizada. O motivo de atribuirmos o valor “1” para este contador é que desejamos executar apenas uma vez o comando para finalizar a aplicação, conforme a linha 64, quando a URI da requisição HTTP for igual a “/shutdown”.

Compilando o projeto e executando a aplicação

A partir de agora iremos compilar e empacotar a aplicação, para em seguida testar o micro serviço. Para isso, basta acessar a pasta raiz do projeto e executar o comando `gradle build`. O output gerado é apresentado na Figura 2.

Após a compilação, o Gradle cria uma pasta denominada `build` na raiz do projeto. É nela que são colocados os artefatos compilados. Entre eles, o JAR principal da aplicação, que foi batizado de `thumbnailer-0.0.1-SNAPSHOT.jar`. Por se tratar de um JAR executável, pode-se usar o comando `java -jar` para iniciar a aplicação. Portanto, a partir da pasta raiz do projeto, execute o código apresentado a seguir para que o micro serviço seja iniciado:

```
java -jar build/libs/thumbnailer-0.0.1-SNAPSHOT.jar
```



```
java -jar build/libs/thumbnailer-0.0.1-SNAPSHOT.jar
```

```
2015-04-26 13:09:11.793 INFO 39713 — [           main] b.c.d.ThumbnailerServiceApplication : Starting ThumbnailerServiceApplication  
on Andres-MacBook-Pro.local with PID 39713 (/Users/andrefabbro/DEV/workspace/spring-tool-suite/thumbnailer-service/build/libs/thumbnailer-0.0.1-SNAPSHOT.jar started by andrefabbro in /Users/andrefabbro/DEV/workspace/spring-tool-suite/thumbnailer-service)  
2015-04-26 13:09:11.824 INFO 39713 — [           main] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@14d2f524: startup date [Sun Apr 26 13:09:11 BRT 2015]; root of context hierarchy  
2015-04-26 13:09:12.444 INFO 39713 — [entExecutor-1-1] reactor.net.netty.tcp.NettyTcpServer : BIND /0:0:0:0:0:0:0:9090  
2015-04-26 13:09:12.533 INFO 39713 — [           main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup  
2015-04-26 13:09:12.541 INFO 39713 — [           main] b.c.d.ThumbnailerServiceApplication : Started ThumbnailerServiceApplication in 0.928 seconds (JVM running for 1.223)
```

Figura 3. Output da execução do micro serviço

FÓRUM DEVMEDIA

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



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

ACESSE AGORA
www.devmedia.com.br/forum



O output resultante é exposto na **Figura 3**. Com a aplicação em execução, escutando na porta 9090, pode-se realizar o teste para miniaturizar uma imagem. Para isso, navegue para qualquer pasta que contenha uma imagem grande. Supondo que o nome do arquivo da imagem seja *imagem-exemplo.jpg*, execute o comando *curl* indicado a seguir:

```
curl -v -XPOST -H "Content-Type: image/jpeg" --data-binary @imagem-exemplo.jpg  
http://localhost:9090/thumbnail
```

Note que esse comando envia os dados binários da imagem para o endereço local `http://localhost:9090/thumbnail`. Justamente a URI responsável por miniaturizar a imagem, como foi visto na

```
* Connected to localhost (::1) port 9090 (#0)
> POST /thumbnail HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9090
> Accept: */*
> Content-Type: image/jpeg
> Content-Length: 4167118
> Expect: 100-continue
>
< HTTP/1.1 100 Continue
< HTTP/1.1 301 Moved Permanently
< Content-Length: 0
< Location: /image/thumbnail.jpg
* HTTP error before end of send, stop sending
<
* Closing connection 0
```

Figura 4. Output do upload da imagem via REST para o micro serviço

classe **ThumbnailerServiceApplication**. O output da execução desse comando é mostrado na **Figura 4**.

Podemos notar através do output que, após o envio dos bytes, o servidor retorna a mensagem 301 (*Moved Permanently*) indicando o redirecionamento para a URI que contém a imagem gerada, como é visto através do texto “< Location: /image/thumbnail.jpg”. Para baixar a miniatura, basta abrir qualquer navegador e digitar o endereço <http://localhost:9090/image/thumbnail.jpg> ou executar o comando `wget`, exposito a seguir:

```
wget http://localhost:9090/image/thumbnail.jpg
```

O output da execução desse comando é mostrado na Figura 5.

É possível notar ainda, através da **Figura 6**, o log da aplicação, que sinaliza o salvamento da imagem na pasta temporária da JVM, conforme explícita a última linha.

Possibilidades no universo de micro serviços

Deste modo, percebemos o quanto é simples o desenvolvimento de um micro serviço responsável por uma tarefa bem definida. A partir de agora, o desenvolvedor pode começar a pensar nas formas de escalar a aplicação. Por exemplo, incluir um Apache HTTP Server com um *mod_proxy* para fazer o *load balance*, direcionando requisições igualmente para as múltiplas instâncias do micro serviço, que podem rodar em servidores virtuais distintos. Além disso, é possível ainda criar scripts usando softwares como Vagrant ou Puppet para a criação de

```
--2015-04-26 15:00:14--  http://localhost:9090/image/thumbnail.jpg
Resolving localhost... ::1, 127.0.0.1
Connecting to localhost|::1|:9090... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7346 (7.2K) [image/jpeg]
Saving to: 'thumbnail.jpg'

thumbnail.jpg          100%[=====]  7.17K  --.-KB/s   in 0s

2015-04-26 15:00:14 (637 MB/s) - 'thumbnail.jpg' saved [7346/7346]
```

Figura 5. Output do comando wget para salvar a imagem miniaturizada em disco

Figura 6. Log do upload da imagem via REST para o micro serviço

servidores virtuais sob demanda, já com esta aplicação instalada e pronta para rodar. Ou então, usar a própria estrutura a Amazon EC2 para monitorar e aumentar o número de instâncias conforme a demanda por requisições. Estas são algumas das vantagens que só a computação elástica pode nos proporcionar.

Além destas vantagens no âmbito de infraestrutura, a separação de responsabilidades pode tornar o micro serviço tão independente que, não necessariamente, só poderá ser usado por um único sistema ou cliente. Imagine, por exemplo, um micro serviço para geração de códigos de barras. Ele pode ser usado por diferentes sistemas em departamentos distintos de uma empresa.

Enfim, a ideia é deixar os micro serviços tão desacoplados que múltiplos sistemas possam ser construídos a partir da combinação de um conjunto de micro serviços disponíveis dentro ou fora das fronteiras de uma organização.

Autor



André Luiz Martins Fabbro

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:

Página do Gradle.

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

Descrição do padrão de Arquitetura Micro serviços.

<http://microservices.io/patterns/microservices.html>

Princípio da Responsabilidade Única.

<http://www.objectmentor.com/resources/articles/srp.pdf>

Repositório do Reactor no GitHub.

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

Página do projeto Spring IO.

<http://spring.io>

Especificação Reactive Streams.

<http://www.reactive-streams.org/>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Desenvolva mecanismos de busca diferenciados com Spring Data Solr

Como desenvolver busca textual às suas aplicações Spring

O Spring Framework é uma plataforma que provê uma série de abstrações que auxiliam o desenvolvedor na construção de aplicações Java. O intuito é deixar o desenvolvedor focado nas regras de negócio da aplicação, enquanto toda a infraestrutura básica é fornecida pelo Spring. Por exemplo: transações passam a ser criadas e aplicadas por métodos anotados com `@Transactional`; operações gerenciadas podem ser criadas sem ter que lidar com a API da JMX; etc.

O Spring, em virtude do tamanho, tem sua estrutura separada em módulos e tais módulos devem ser adicionados ao projeto em desenvolvimento de acordo com a necessidade. Cada módulo (sendo Spring ou não) requerido pela aplicação é considerado uma dependência e estas podem ser administradas por um gerenciador de dependências, como é o caso do Maven.

Além do gerenciamento de dependências (e a incorporação das mesmas ao projeto), é necessária, a aplicações web, a configuração em um padrão que seja entendido pelo servidor de aplicação (por exemplo, a configuração de Servlets ou filtros). Este trabalho pode ser mitigado com a utilização do Spring Boot, que parte do princípio de que toda aplicação tem uma configuração padrão, e que pode ser modificada se necessário. Assim, com o Spring Boot, evita-se que 99% das aplicações Spring precisem ser configuradas do zero.

A estrutura proposta pelo Spring Framework promove a utilização de abstrações para fornecer uma interface padrão. Esta característica permite que o desenvolvedor comece com alguns testes preliminares sem que se aprofunde na tecnologia em questão. O módulo/projeto Spring Data é um exemplo. Alguns padrões e interfaces

Fique por dentro

A incorporação de funcionalidades relacionadas à busca textual em aplicações vem sendo um requisito cada vez mais comum. Neste cenário, uma das opções disponíveis e que fornece um leque bastante amplo de funções é o Solr. Aproveitando a bem-sucedida abstração de acesso a repositórios de dados persistentes, Spring Data, uma implementação desta, nomeada Spring Data Solr, foi introduzida com o objetivo de facilitar o acesso a dados armazenados em servidores Solr. Assim, pensando em favorecer o entendimento desta tecnologia, serão apresentados neste artigo os seus principais recursos e depois, com uma fundamentação sólida de conceitos e convenções, será desenvolvido um exemplo para demonstrar o uso do Spring Data Solr já integrado ao Spring Boot, que permite uma ágil construção de projetos e sem grandes custos de setup, comuns a projetos Spring.

são estipulados em um projeto comum e diferentes implementações utilizam esta base comum para acesso a repositórios específicos. Assim, tem-se implementações do Spring Data para acesso a bancos de dados relacionais por meio da JPA, ElasticSearch, Neo4J, MongoDB, Solr, entre outros, todos utilizando a mesma metodologia. Portanto, ter conhecimento de como funciona o Spring Data pela utilização de uma das implementações fornece uma base para que outras sejam utilizadas pelo desenvolvedor.

O Spring Data facilita o desenvolvimento de aplicações com acesso a repositórios de dados pela disponibilização de um conjunto de operações e interfaces padrão. Estas compreendem desde operações simples como criação, carga, modificação e exclusão de registros, bem como operações mais complexas, que envolvem paginação e ordenação de registros.

Outra funcionalidade marcante do Spring Data é a capacidade de interpretação de nomes de métodos para gerar ações (geralmente queries). Por exemplo: ao escrever uma interface de acesso a dados contendo o seguinte método: `Page<Product> findByCategory(String category, Pageable pageable)`, quando invocada, a query resultante da interpretação do nome deste método é executada junto ao repositório de dados considerando questões de paginação providas como parâmetro. Este método em específico selecionará os produtos utilizando como cláusula de busca o campo `category`, que deve coincidir com o valor informado no primeiro parâmetro.

Para demonstrar as vantagens na utilização do Spring Data Solr, revisaremos alguns conceitos iniciais desta tecnologia. Em um segundo momento, a biblioteca nativa de acesso ao servidor Solr para Java (SolrJ) é apresentada. E para finalizar, um protótipo aplicando os conceitos apresentados é desenvolvido.

Solr, Lucene e Spring Data Solr

O Lucene é uma biblioteca que fornece ao desenvolvedor suporte a operações de busca textual. Como funcionalidade principal, viabiliza operações de busca em texto considerando similaridade, ordem das palavras, relevância e outros fatores. Já o Solr é uma aplicação construída utilizando o Lucene para prover ainda mais funcionalidades, como agrupamentos, informações estatísticas sobre campos, enfatização de termos de busca, clustering e outras mais.

De forma comparativa, pode-se dizer que documentos em Solr são análogos a registros em um banco de dados relacionais. Estes documentos, por sua vez, são armazenados em um núcleo (core), que pode ser visto como uma tabela. Além disso, cada núcleo possui um esquema (*schema*), que é um modelo para regulamentar a estrutura dos documentos que ali serão armazenados.

No entanto, não se encontra aqui (Solr) integridade referencial, isolamento, consistência ou outras características comuns a um banco de dados relacional. Por outro lado, temos suporte a operações de busca textual e a possibilidade de efetuar escalabilidade horizontal, onde a robustez de um serviço é aumentada com a adição de mais nós (máquinas) ao cluster.

O acesso às funcionalidades oferecidas pelo Solr pode ser feito utilizando uma interface REST, bastando desta forma uma requisição HTTP ao servidor. Como resposta, vários formatos podem ser provados, sendo XML o formato padrão. Outros formatos incluem, mas não se limitam a: JSON, Javabin, PHP, Ruby, Python e CSV. Quando utilizado a partir de uma aplicação Java, o formato Javabin pode ser o escolhido, sendo assim possível a desserialização do resultado diretamente para objetos Java, em vez de efetuar toda a interpretação do documento XML para então fornecer um grafo de objetos.

De modo a facilitar a integração com o Java, uma biblioteca nomeada SolrJ é disponibilizada. Com isso, a execução de funcionalidades em um servidor Solr se torna transparente para programadores Java.

Assim como o SolrJ, o Spring Data Solr é uma interface de acesso a um servidor Solr. A diferença básica é a adoção de padrões e

conceitos definidos pelo Spring Data e a pronta integração com aplicações Spring. Esta abordagem traz duas vantagens para o desenvolvedor:

1. Convenções e automatismos facilitam o desenvolvimento de métodos de acesso a dados. Para isso, métodos definidos em uma interface, por exemplo, têm o seu nome analisado junto a uma série de convenções de nomes para definir de forma dinâmica a query a ser executada. O automatismo é verificado quando uma interface de acesso a dados precisa ser instanciada. Neste ponto, um proxy é criado. E quando um método deste proxy é invocado, convenções e anotações são utilizadas, definindo assim ações que serão executadas junto ao servidor Solr;
2. Pouco conhecimento de bibliotecas para uso de repositório de dados é necessário para que o desenvolvedor possa fazer uso do mesmo. Também se enquadram aqui os conceitos introduzidos pelo Spring Data, que são compartilhados por todos os projetos (Spring Data JPA, Spring Data Solr, Spring Data MongoDB e outros). Assim, fica mais fácil a adoção de outros módulos Spring Data.

Solr

Independentemente da biblioteca utilizada para acessar um servidor Solr, é importante ter conhecimento das funcionalidades que esta ferramenta pode oferecer. Assim sendo, uma listagem com as funcionalidades básicas, seguidas de sua descrição e tipo de retorno, é apresentada a seguir:

- **Faceting** – podemos comparar esta funcionalidade, de forma simplista, com o select-count-where-group-by de um banco de dados relacional. O resultado conterá a listagem de documentos encontrados pela query e uma sumarização para cada campo pelo qual os documentos selecionados devem ser agrupados. Desta forma é possível, com uma única requisição ao servidor Solr, por exemplo, selecionar documentos utilizando alguns critérios (que seria a query em si) e agrupar tais documentos por uma lista de campos (um agrupamento por campo);
- **Grouping** – o agrupamento em Solr tem um escopo um pouco mais abrangente. Aqui não são considerados os documentos selecionados, mas sim todos os documentos presentes em um determinado núcleo. Utilizando *group* (ou *field collapsing*) pode-se agrupar os documentos por um determinado campo, função ou query;
- **Stats** – fornece informações estatísticas simples para um campo numérico, texto ou data. Os resultados apresentados na resposta a uma requisição de Stats englobam: valor máximo, mínimo, média, mediana, contagem de vazios, desvio padrão e valores distintos presentes no campo;
- **Enfatização dos Termos de Busca (Highlighting)** – esta funcionalidade pode ser empregada para grifar os termos de busca em determinados campos dos documentos selecionados. Imagine que o resultado da busca seja exibido por uma página web. Desta forma, seria interessante ressaltar os termos encontrados com '**' e '**'. Vale ressaltar que a sequência de caracteres antes e depois do termo a ser grifado é parametrizável. A partir disso,

o resultado pode ser visto como uma substituição de texto para cada termo de busca, onde é realizada a concatenação da cadeia de caracteres inicial, o próprio termo e a cadeia final de caracteres;

- **Funções** – funções podem ser utilizadas em queries a um servidor Solr. Algumas das funções disponíveis incluem, mas não se limitam a: **if**, **max**, **not**, **geodist** e **geohash**;

- **Real Time Get** – documentos que são armazenados em Solr podem demorar alguns instantes para que sejam persistidos devido à natureza assíncrona do processo de persistência. Suponha que alguns milhares de documentos estejam sendo persistidos em uma importação em lote. Mesmo que o documento já esteja salvo, não significa que está pronto para que buscas sejam efetuadas sobre os seus campos. Como alternativa, o Solr oferece uma seleção em tempo real baseada na identificação única do documento.

As funcionalidades supracitadas podem ser utilizadas a partir de uma aplicação Java fazendo uso da biblioteca SolrJ. Agora, antes de começar a implementação de alguns exemplos de manipulação de dados, um servidor Solr deve ser configurado e iniciado, o que pode ser feito executando os seguintes passos:

1. Efetue o download do servidor Solr 4. Para este artigo foi adotada a versão 4.7.2, que pode ser baixada do endereço indicado na seção **Links**;

2. Descompacte o arquivo baixado utilizando seu descompactador favorito;

3. Inicie o servidor executando o arquivo *start.jar* presente no diretório *solr-4.7.2/example*, descompactado no passo anterior. Para executar o arquivo *start.jar*, um terminal pode ser utilizado, navegando primeiramente para o diretório onde o arquivo se encontra (*cd \$DIR_SOLR/solr-4.7.2/example/*, onde *\$DIR_SOLR* é o diretório onde o Solr fora descompactado) e logo após executar o JAR utilizando *java -jar start.jar*. Ou, dependendo da instalação Java presente no seu sistema, simplesmente clicando duas vezes sobre o arquivo *solr-4.7.2/example/start.jar*. O mesmo princípio se aplica a máquinas Linux, onde um terminal pode ser aberto, depois navegue até a pasta onde o servidor Solr foi descompactado, mude o diretório corrente para *./solr-4.7.2/example* e execute *start.jar* com o comando *java -jar start.jar*.

Feito isso, temos um servidor Solr operando na porta 8983 e que pode ser acessado através do endereço *http://localhost:8983/solr/* a partir de um navegador.

Para criarmos um repositório próprio para este artigo (o que seria análogo a uma tabela em um banco de dados relacional), executaremos os seguintes passos:

1. Com o servidor iniciado, copie a estrutura de exemplo provida junto com o arquivo baixado, nomeada *collection1* para *products* (o nome de cada núcleo aqui pode ser comparado ao nome de tabelas em um banco de dados relacional). Desta forma, teremos um núcleo separado para este artigo. Para fazer isto, copie a pasta *\$SOLR_DIR/solr-4.7.2/example/solr/collection1* para *\$SOLR_DIR/solr-4.7.2/example/solr/products*;

2. Remova o arquivo *\$SOLR_DIR/solr-4.7.2/example/solr/products/core.properties*;
3. Acesse a interface de gerenciamento do Solr pelo browser: *http://localhost:8983/solr/*;
4. Clique em *Core Admin* e, logo em seguida, em *Add Core*;
5. Agora, informe o novo núcleo que criamos no primeiro passo deste processo. Seguem os campos e seus respectivos valores: **name**: *products* e **instanceDir**: *products*.

Com isso, temos um servidor preparado para os exemplos que apresentaremos neste artigo. O *schema* para nosso índice (*products*) pode ser encontrado em *\$SOLR_DIR/solr-4.7.2/example/solr/products/conf/schema.xml* e já possui uma configuração padrão que será adotada. Para iniciarmos a utilização do servidor a partir de uma aplicação Java, crie uma aplicação Maven com a dependência conforme exposto na **Listagem 1**.

Listagem 1. Dependência Maven para SolrJ.

```
<dependency>
  <groupId>org.apache.solr</groupId>
  <artifactId>solr-solrj</artifactId>
  <version>4.7.2</version>
</dependency>
```

Com o código aplicado ao projeto, classes da biblioteca de acesso (SolrJ) podem ser utilizadas para acessar o servidor Solr configurado e iniciado localmente. A classe básica para acesso ao servidor Solr, presente na biblioteca SolrJ, é a **HttpSolrServer**. Um exemplo de como instanciar um objeto a partir desta classe é apresentado a seguir:

```
SolrServer solr = new HttpSolrServer("http://localhost:8983/solr/products");
```

Como verificado, a URL completa do servidor é a única coisa que precisa ser informada. A partir deste momento podemos utilizar a instância criada (*solr*) para acesso e manipulação de dados. Como um exemplo inicial, tem-se o armazenamento de um documento exposto a **Listagem 2**.

Listagem 2. Armazenando um documento no Solr.

```
SolrInputDocument document = new SolrInputDocument();
document.addField("id", "id_001");
document.addField("name", "Gouda");
document.addField("price", 49.99F);

UpdateResponse response = solr.add(document);

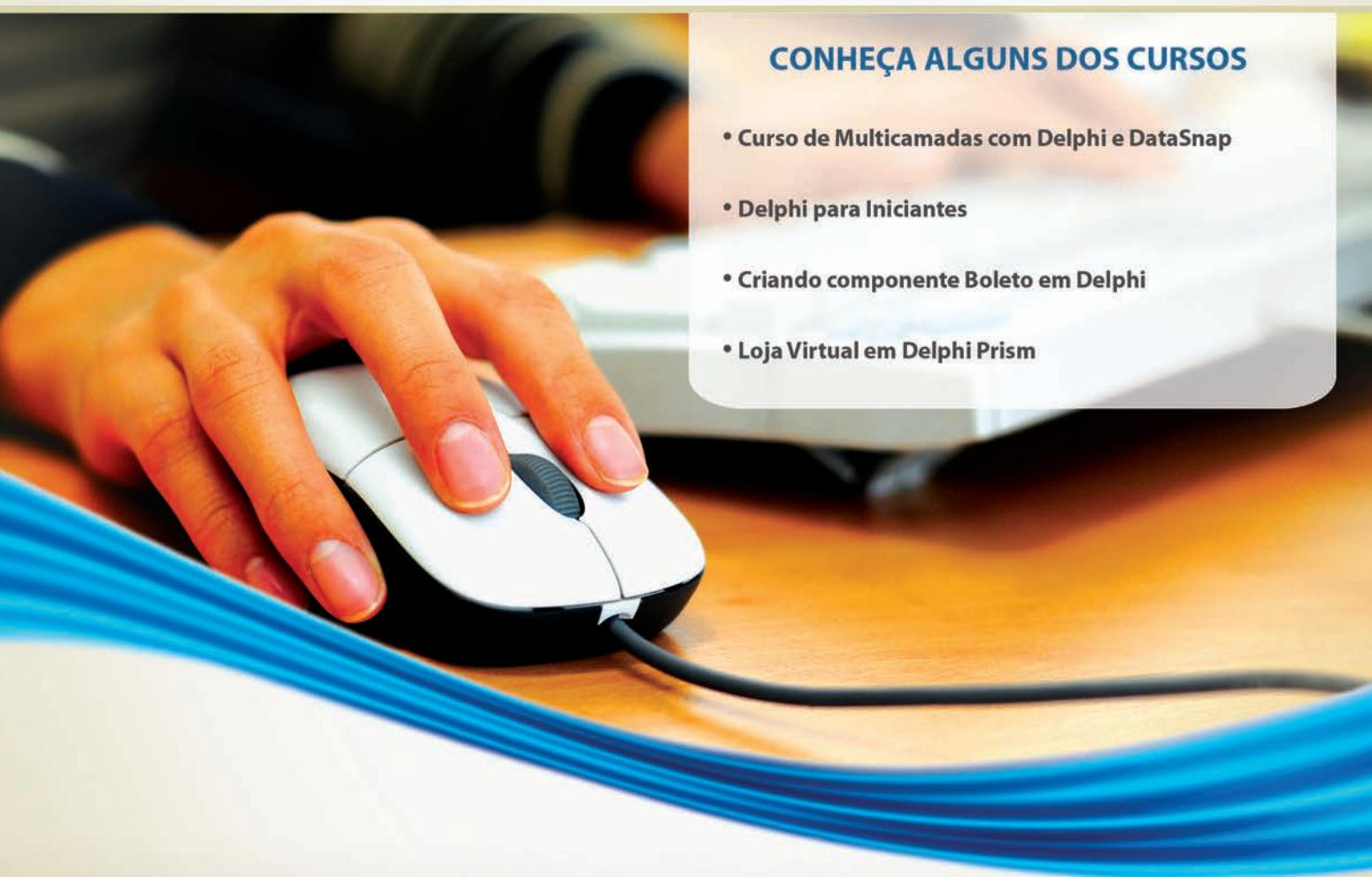
solr.commit();
```

O documento é criado utilizando a classe **SolrInputDocument**, que tem seus campos (**id**, **name** e **price**) configurados e logo após o documento é persistido com o método **SolrServer.add(SolrInputDocument)**. Para que a persistência dos documentos seja perpetuada, um **commit()** é requisitado após o armazenamento.

CURSOS ONLINE



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



CONHEÇA ALGUNS DOS CURSOS

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

Para mais informações :

www.devmedia.com.br/cursos/delphi

(21) 3382-5038

Desenvolva mecanismos de busca diferenciados com Spring Data Solr

Agora veremos um exemplo de exclusão de documentos. Esta operação pode ser executada de duas formas: por id (utilizando a identificação única do documento), bem como por seleção (onde uma query é fornecida e todos os documentos resultantes desta query são excluídos), conforme demonstrado na **Listagem 3**.

Listagem 3. Exclusão de documentos no Solr.

```
// exclusão por id (somente o documento com o id informado será excluído)
solr.deleteByQuery("id_001");
solr.commit();

// exclusão por query (todos os documentos serão excluídos)
solr.deleteByQuery("*:*");
solr.commit();
```

Para seleção de documentos, o método **SolrServer.query(SolrParams)** pode ser utilizado fornecendo uma query. Como resposta, obtém-se um objeto do tipo **QueryResponse**, de onde pode-se acessar as mais diversas informações requisitadas ao servidor Solr (o resultado da busca propriamente dito, faceting, estatísticas, agrupamentos, etc.). Na **Listagem 4** é apresentada uma seleção exibindo o número de documentos encontrados e o campo de identificação dos mesmos.

Listagem 4. Seleção e listagem de documentos.

```
// exclusão por id (somente o documento com o id informado será excluído)
solr.deleteByQuery("id_001");
solr.commit();

// exclusão por query (todos os documentos serão excluídos)
solr.deleteByQuery("*:*");
solr.commit();
```

Ainda como uma forma de facilitar o gerenciamento de documentos, existe a possibilidade de anotar classes pertencentes ao modelo da aplicação e utilizar instâncias destas classes diretamente junto à instância da classe **HttpSolrServer** (mais especificamente, **SolrServer.saveBean(Object bean)**), que fará a conversão do objeto para um documento Solr e armazenará o mesmo. Para que isto seja possível, a classe deve ter os campos a serem persistidos anotados com **@Field**, indicando desta forma como compor um documento para que o mesmo seja armazenado e restaurado. Um exemplo de uma classe de modelo anotada pode ser visto na **Listagem 5**.

Para salvar e carregar objetos com o modelo proposto, pode-se implementar algo como o exposto na **Listagem 6**.

Como podemos verificar, a utilização de SolrJ é extremamente simples e consiste em um mapeamento básico das funções presentes em um servidor Solr para uma interface Java. Contudo, o SolrJ também apresenta algumas limitações:

- Tipos personalizados (*custom types*) – A utilização de outros tipos de dados que não sejam suportados nativamente por Solr (como,

por exemplo, **Point** e **DateTime**, da conhecida biblioteca Joda-Time, ou mesmo classes próprias) é um tanto quanto complexa. Uma solução é a implementação de um tipo de campo (**FieldType**) para o Solr e adaptar o schema para utilizar tal implementação. Como alternativa, tem-se uma estratégia parecida com a adotada por Spring Data Solr, que consiste na introdução de uma camada responsável pela tradução dos tipos de dados personalizados para tipos conhecidos pelo Solr;

- Query API – O SolrJ não disponibiliza nenhuma interface para criação de queries. Deste modo, o desenvolvedor precisa partir para a criação manual das mesmas, concatenando texto e efetuando *escaping* manualmente.

Listagem 5. Classe de modelo anotada para SolrJ.

```
public class Product {

    @Field private String id;
    @Field private String name;
    @Field private float price;

    // setters, getters
}
```

Listagem 6. Armazenamento e carga de documentos utilizando classes de modelo anotadas com SolrJ.

```
// salvar
Product prod = new Product();
prod.setId("id_001");
prod.setName("Gouda");
prod.setPrice(49.99F);

solr.saveBean(prod);
solr.commit();

// carregar
QueryResponse qr = solr.query(new SolrQuery("id:id_001"));
List<Produto> produtos = qr.getBeans(Produto.class);
System.out.println("found: " + produtos.toString());
```

Spring Data Solr

O módulo Spring Data Solr foi desenvolvido com o intuito de abstrair detalhes de configuração, assim como de manipulação e acesso a dados a um servidor Solr. Para demonstrar a simplicidade de acesso ao servidor Solr empregando o Spring Data Solr, uma aplicação Spring Boot será criada para disponibilizar uma interface web para listar os documentos existentes. Para tanto, comece criando o arquivo *pom.xml*. De início, teremos algo parecido com o conteúdo da **Listagem 7**.

Um detalhe que merece atenção nesse arquivo é a sobreescrita da propriedade **spring-data-releasetrain.version** (vide linhas 19 a 21). Isto porque a versão do Spring Boot utilizada (1.2.3) faz referência a uma versão anterior do Spring Data. Ao sobreescriver tal propriedade, poderemos utilizar a última versão disponível do Spring Data Solr (1.4.0.RELEASE).

Listagem 7. Configuração inicial do pom.xml.

```
01. <project xmlns="http://maven.apache.org/POM/4.0.0"
02.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
04.     http://maven.apache.org/xsd/maven-4.0.0.xsd">
05.
06.   <modelVersion>4.0.0</modelVersion>
07.   <groupId>spring.data.solr.example</groupId>
08.   <artifactId>spring-data-solr-application</artifactId>
09.   <version>0.0.1-SNAPSHOT</version>
10.   <packaging>war</packaging>
11.
12.   <parent>
13.     <groupId>org.springframework.boot</groupId>
14.     <artifactId>spring-boot-starter-parent</artifactId>
15.     <version>1.2.3.RELEASE</version>
16.   </parent>
17.
18.   <properties>
19.     <spring-data-releasetrain.version>
20.       Fowler-RELEASE
21.     </spring-data-releasetrain.version>
22.     <java.version>1.7</java.version>
23.   </properties>
24.
25.   <dependencies>
26.     <dependency>
27.       <groupId>org.springframework.boot</groupId>
28.       <artifactId>spring-boot-starter-web</artifactId>
29.     </dependency>
30.     <dependency>
31.       <groupId>org.springframework.boot</groupId>
32.       <artifactId>spring-boot-starter-data-solr</artifactId>
33.     </dependency>
34.     <dependency>
35.       <groupId>org.springframework.boot</groupId>
36.       <artifactId>spring-boot-starter-tomcat</artifactId>
37.       <scope>provided</scope>
38.     </dependency>
39.   </dependencies>
40.
41.</project>
```

Outro ponto importante é a configuração da dependência **spring-boot-starter-tomcat** como **provided** (linhas 34 a 38). Isto permitirá a execução da aplicação de forma embarcada durante a fase de desenvolvimento, mas não adicionará nenhuma biblioteca do Tomcat ao pacote war resultante do processo de empacotamento da aplicação, evitando conflitos com o container ao qual a aplicação será adicionada.

Tendo as dependências necessárias configuradas, parte-se para a configuração da aplicação, que se resume ao código da **Listagem 8**.

A configuração da aplicação deve conter duas coisas: a anotação **@SpringBootApplication**, que configura um contexto do Spring de forma automática; e a anotação **@EnableSolrRepositories**, para que objetos de conexão e interfaces de acesso a dados Solr sejam criados.

A propriedade **multicoreSupport**, configurada na anotação **@EnableSolrRepositories**, indica a utilização de múltiplos núcleos

para armazenar os documentos gerados pela aplicação. Por padrão, **multicoreSupport** é falso. Desta forma todos os documentos são armazenados em um único núcleo, independentemente de seu tipo. Esta estratégia pode elevar a complexidade da aplicação, pois o desenvolvedor deve considerar este fato também nas queries executadas, onde os tipos de documentos não relevantes à query em questão devem ser filtrados. Uma alternativa para este problema é a utilização de núcleos diferentes para tipos diferentes de documentos. Nesses casos, **multicoreSupport** é verdadeiro e o nome do núcleo para cada tipo de documento é definido utilizando a anotação **@SolrDocument** junto à classe de modelo, como será visto mais adiante.

Listagem 8. Classe de configuração da aplicação.

```
01. @SpringBootApplication
02. @EnableSolrRepositories(multicoreSupport = true)
03. public class ApplicationConfiguration extends SpringBootServletInitializer {
04.
05.   public static void main(String[] args) {
06.     SpringApplication.run(ApplicationConfiguration.class);
07.   }
08.
09.   @Override
10.   protected SpringApplicationBuilder configure(
11.     SpringApplicationBuilder builder) {
12.     return builder.sources(ApplicationConfiguration.class);
13.   }
14.
15. }
```

O método **SpringApplication.run()** é utilizado para configurar e executar uma aplicação Spring. Assim que invocado, o contexto e todas as suas dependências são criadas. Deste modo, não é necessário que a aplicação, mesmo como uma aplicação web, seja adicionada a um servidor de aplicação. Quando executado, o Spring Boot criará um servidor de aplicação embarcado para rodar a aplicação local.

Para que a aplicação possa ser adicionada a um servidor de aplicação, como o Tomcat, a classe **SpringBootServletInitializer** deve ser estendida e configurada. Isto pode ser feito em dois passos:

1. Estender a classe **SpringBootServletInitializer**, como visto na linha 3 da **Listagem 8**. Com isto Spring Boot registrará servlets, filtros e listeners, tradicionalmente configurados utilizando **web.xml** (em aplicações seguindo a especificação Servlet 2) ou **@WebServlet**, **@WebFilter** e **@WebListener** (em aplicações que adotam a especificação Servlet 3);
2. Sobrescrever o método **configure()** para indicar as classes de configuração necessárias para a composição do contexto da aplicação. Isto é feito invocando o método **sources()** do objeto **builder** passado como parâmetro, como pode-se ver na linha 12 da **Listagem 8**.

O Spring Boot adota também um arquivo padrão para a configuração de seus módulos. Este arquivo é nomeado como *application*.

Desenvolva mecanismos de busca diferenciados com Spring Data Solr

properties e é adicionado ao projeto no diretório `/src/main/resources/`. Neste arquivo podemos adicionar o endereço do servidor Solr a ser utilizado pelo projeto, como demonstra o código a seguir:

```
spring.data.solr=http://localhost:8983/solr
```

Uma vez que o endereço do servidor Solr esteja configurado, parte-se para a implementação do modelo, que nada mais é do que a estrutura básica de um documento que será armazenado no servidor Solr. Junto à classe de modelo, deve-se ainda especificar o núcleo em que os documentos deste tipo devem ser armazenados. Isto é feito adicionando a anotação `@SolrDocument` e definindo a propriedade `solrCoreName`. Para este artigo utiliza-se o núcleo nomeado como `products`. Desta forma, tem-se a definição da classe de modelo conforme o código da [Listagem 9](#).

Listagem 9. Modelo para a aplicação exemplo.

```
@SolrDocument(solrCoreName = "products")
public class Product {

    @Id @Field private String id = UUID.randomUUID().toString();
    @Field private String name;
    @Field private float price;
    @Field private String category;

    // setters, getters e toString()

}
```

Neste ponto, com o modelo já definido, inicia-se a implementação dos métodos de acesso a dados. A forma mais simples e produtiva para tanto é a extensão das interfaces `SolrRepository` ou `SolrCrudRepository`, onde a primeira fornece as operações básicas de acesso e a segunda uma versão completa para manipulação de documentos (com métodos como: `List<T> findAll()`, `T findOne(ID)`, `<S extends T> S save(S)`, `void deleteAll()`, `void delete(ID)`, entre outros).

Para a nossa interface de acesso a documentos do tipo `Product`, vamos estender a interface `SolrCrudRepository`, como exposto na [Listagem 10](#). Pode-se perceber que a nova interface (`ProductDao`) não adiciona nenhum método novo, mas tem sua importância na definição dos tipos genéricos utilizados juntos à extensão da interface `SolrCrudRepository`, no caso `<Product, String>` (que são, respectivamente, o tipo de documentos retornados pelo repositório e o tipo da propriedade utilizada para identificação única do documento).

É preciso ressaltar ainda que a implementação propriamente dita desta interface não é necessária. Os métodos declarados pelas interfaces (`SolrCrudRepository` ou `SolrRepository`) já possuem implementação e os métodos adicionais declarados nesta interface terão sua funcionalidade derivada da interpretação do nome do método ou de meta-informações como anotações anexas ao mesmo.

Para verificar a implementação do modelo e da interface de acesso a dados, crie um REST Controller (vide [Listagem 11](#)) que disponibiliza uma URI de acesso `(/)`. Esta tem por função a listagem de todos os documentos armazenados no repositório de produtos.

Listagem 10. Interface de acesso a dados para Product.

```
public interface ProductDao extends SolrCrudRepository<Product, String> {
```

```
}
```

Listagem 11. REST Controller como façade para o repositório de produto.

```
@RestController
public class ProductController {

    @Autowired private ProductDao productDao;

    @RequestMapping("/")
    public Iterable<Product> get() {
        return productDao.findAll();
    }

}
```

A partir deste momento, podemos acessar a listagem de documentos através da URL: `http://localhost:8080/`. Caso o repositório esteja vazio, pode-se acessar a interface de administração Solr e adicionar um documento selecionando inicialmente a coleção `products`.

Queries em classes de acesso a dados

A forma mais simples pela qual queries são derivadas da declaração de métodos em uma interface é pelo nome do mesmo. Para que os repositórios possam ser instanciados junto ao contexto do Spring, um proxy é criado. No momento da construção dos proxies, queries são derivadas dos nomes dos métodos ou anotações presentes nos mesmos, definindo assim a query que cada método executará. Como exemplo tem-se a [Listagem 12](#). A query resultante para o método declarado nesta interface será: `name?:0`, onde `?0` é a `String` passada como parâmetro para a invocação do método.

Listagem 12. Derivando a query do nome do método definido na interface.

```
public interface ProductDao extends SolrRepository<Product, String> {

    Product findByName(String name);

}
```

O método declarado na interface exposta nesta listagem obedece a algumas convenções para ter uma query derivada do nome do método. Assim, queries relativamente complexas podem ser geradas, por exemplo: `Product findByNameAndPriceLessThanEqual(String name, float price);`, que resulta na query: `name?:0 AND popularity:[* TO ?1]`, onde `?0` é o nome e `?1` é o preço passados como parâmetro.

| Termo | Exemplo | Query de Retorno |
|--------------------|---|---|
| And | findByNameAndPopularity | q=name:?0 AND popularity:?1 |
| Or | findByNameOrPopularity | q=name:?0 OR popularity:?1 |
| Is | findByName | q=name:?0 |
| Not | findByNameNot | q=-name:?0 |
| IsNull | findByNameIsNull | q=-name:[* TO *] |
| IsNotNull | findByNameIsNotNull | q=name:[* TO *] |
| Between | findByPopularityBetween | q=popularity:[?0 TO ?1] |
| LessThan | findByPopularityLessThan | q=popularity:[* TO ?0} |
| LessThanEqual | findByPopularityLessThanEqual | q=popularity:[* TO ?0] |
| GreaterThan | findByPopularityGreaterThan | q=popularity:[?0 TO *] |
| GreaterThanOrEqual | findByPopularityGreaterThanOrEqual | q=popularity:[?0 TO *] |
| Before | findByLastModifiedBefore | q=last_modified:[* TO ?0} |
| After | findByLastModifiedAfter | q=last_modified:[?0 TO *] |
| Like | findByNameLike | q=name:?0* |
| NotLike | findByNameNotLike | q=-name:?0* |
| StartingWith | findByNameStartingWith | q=name:?0* |
| EndingWith | findByNameEndingWith | q=name:?:?0 |
| Containing | findByNameContaining | q=name:?:?0* |
| Matches | findByNameMatches | q=name:?0 |
| In | findByNameIn(Collection<String> names) | q=name:(?0...) |
| NotIn | findByNameNotIn(Collection<String> names) | q=-name:(?0...) |
| Within | findByStoreWithin(Point, Distance) | q={!geofilt pt=?0.latitude,?0.longitude sfield=store d=?1} |
| Near | findByStoreNear(Point, Distance) | q={!bbox pt=?0.latitude,?0.longitude sfield=store d=?1} |
| Near | findByStoreNear(Box) | q=store[?0.start.latitude,?0.start.longitude TO ?0.end.latitude,?0.end.longitude] |
| True | findByAvailableTrue | q=inStock:true |
| False | findByAvailableFalse | q=inStock:false |

Tabela 1. Palavras reservadas para derivação de queries a partir de nomes de métodos

A **Tabela 1** apresenta as palavras interpretadas pelo Spring Data Solr quando a derivação do nome de um método é feita para uma query.

Para queries mais complexas, a utilização do nome do método é um tanto quanto desconfortável, exigindo nomes excessivamente longos. Pensando nisso, uma segunda possibilidade é fornecida através do uso de anotações. Um pequeno exemplo é disposto na **Listagem 13** para ilustrar a diferença.

Os dois métodos dessa interface apresentarão o mesmo resultado quando invocados com o mesmo parâmetro **Pageable (page)** e o texto de procura para os parâmetros restantes. A diferença é que

Listagem 13. Derivando a query a partir de anotações do método.

```

01. public interface ProdutoDao extends SolrRepository {
02.
03.     @Query("title:?0 OR name:?0 OR description:?0")
04.     Page<Product> findByText(String text, Pageable page);
05.
06.     Page<Product> findByTitleOrNameOrDescription(String title, String name,
07.         String description, Pageable page);
08.
09. }
```

um (`findByText()`) utiliza a anotação `@Query` para gerar a query a ser executada e o outro (`findByTitleOrNameOrDescription()`) o nome do método. Pode-se ver também a utilização de uma notação específica junto à anotação `@Query`, utilizada para indicar onde os parâmetros devem ser injetados, por exemplo: `?0` (visto na linha 03) significa neste caso o primeiro parâmetro que o método recebe em sua invocação.

Outro ponto que a ser observado na **Listagem 13** é o gerenciamento automático da paginação de dados. `Pageable page` é uma representação da página de documentos sendo solicitada, tipicamente composta pela posição do primeiro documento (`offset`) e número máximo de documentos na página resultante (`size`). Como resultado da invocação dos métodos obtém-se uma instância de `Page` que, dentre outras informações, expõe o número total de documentos encontrados e os documentos pertencentes somente à página solicitada. `Page` (como utilizado na declaração do método na linhas 04 e 06) é a interface mais simples para resultados de documentos paginados.

Especializações do tipo `Page` são apresentadas com sua utilização a seguir:

- **ScoredPage<T>** - fornece uma interface de acesso a informações de ranking. Por padrão, os documentos resultantes de uma busca são ordenados por relevância, o que forma o ranking de documentos. Para ter acesso aos valores de relevância dos documentos presentes em uma query efetuada, basta que o modelo tenha um campo `Float` anotado com `@Score`. Feito isso, o score de cada documento é carregado para este campo. Em casos onde o maior score presente no resultado de uma query é necessário, o método `ScoredPage.getMaxScore()` pode ser utilizado;
- **FacetPage<T>** - tem por função disponibilizar métodos de acesso a informações de faceting que tenham sido requisitadas. Detalhes de uma query que envolve faceting, como campos de agrupamento, podem ser definidos anotando o método de acesso a dados com `@Facet`;
- **StatsPage<T>** - disponibiliza informações adicionais referentes a estatísticas de campos existentes em um determinado índice. Detalhes da query a ser executada como, por exemplo, os campos que devem ter suas estatísticas retornadas, podem ser definidos na anotação `@Stats` utilizada para anotar o método de acesso a dados que retorna uma `StatsPage<T>`;
- **HighlightPage<T>** - permite o recebimento de documentos retornados por uma query com parâmetros para grifar determinados termos. A sua utilização direta em interfaces de repositórios pode ser feita com a aplicação da anotação `@Highlight` ao método;
- **GroupPage<T>** - viabiliza a utilização de métodos de acesso a informações de agrupamento de documentos. Este tipo de resultado não possui suporte a métodos de interfaces de repositório, mas pode ser adquirido invocando o método `SolrOperations.queryForGroupPage(Query query, Class<T> clazz)` (`SolrOperations` pode ser obtido a partir do contexto do Spring).

Na **Listagem 14** apresentamos um exemplo de repositório com seleções diversas, esperando diferentes tipos de resultados.

Listagem 14. Diferentes tipos de retorno para seleções diversas.

```
public interface ProductDao extends SolrRepository<Product, String> {  
  
    ScoredPage<Product> findByNameLike(String name, Pageable page);  
  
    @Facet(fields = {"category"}, limit = 5)  
    FacetPage<Product> findAllFacetOnCategory(Pageable page);  
  
    @Stats(value = "category", distinct = true)  
    StatsPage<Product> findAllStatsOnCategory(Pageable page);  
  
    @Highlight(prefix = "<b>", postfix = "</b>")  
    HighlightPage<Product> findByName(String name, Pageable page);  
  
}
```

SolrOperations

Uma das classes fundamentais no Spring Data Solr é a `SolrTemplate`. Esta classe implementa a interface `SolrOperations` e tem por objetivo o interfaceamento de operações que devem ser executadas no servidor Solr.

Desta forma pode-se utilizar `SolrTemplate` para a execução de operações em um servidor Solr. Isto permite ao desenvolvedor utilizar as funcionalidades e interfaces introduzidas pelo Spring Data Solr para gerenciar documentos em um servidor Solr. Por exemplo, o código da **Listagem 15** executa uma query que seleciona todos os produtos cujo campo nome iniciem com a sequência de caracteres “sp” e exibe os resultados.

Listagem 15. Selecionando e iterando dados provenientes do Solr utilizando SolrOperations.

```
Pageable pr = new PageRequest(1,20);  
Query query = new SimpleQuery(new Criteria("name").startsWith("sp"));  
query.setPageRequest(pr);  
Page<Product> page = solrOperations.queryForPage(query, Product.class);  
  
for (Product product : page.getContent()) {  
    System.out.println(product.name);  
}
```

Paginação (`Pageable`), Criteria API (`Criteria`) e mapeamento (`Mapping`) de documentos para objetos do tipo `Product` são as funcionalidades providas pelo Spring Data Solr no pequeno exemplo da **Listagem 15**.

A seguir apresentamos uma sumarização das operações suportadas por `SolrOperations`:

- `ping()` – verifica se o servidor está acessível;
- `count(SolrDataQuery)` – conta os elementos que sejam encontrados por uma determinada query;

- **saveBean(Object)** – persiste o bean passado como parâmetro no servidor. Uma variação deste método (**saveBeans(Collection<?>)**) pode ser utilizada para persistir múltiplos beans ao servidor;
- **saveBean(Object, int)** – persiste o bean passado como parâmetro no servidor e informa o tempo máximo, em milissegundos, para que a operação de armazenamento seja perpetuada (commit). Uma variação deste método (**saveBean(Collection<?>, int)**) pode ser utilizada para persistir múltiplos beans;
- **delete(Object)** – efetua a exclusão do bean provido. Variações deste método também permitem efetuar a exclusão utilizando a identificação única do bean, a saber: **deleteById(String)** para remover um elemento em específico ou **deleteById(Collection)** para remover múltiplos elementos;
- **queryForObject(Query, Class)** – executa a seleção junto ao servidor Solr e retorna a conversão do primeiro documento encontrado para um objeto da classe passada como parâmetro;
- **queryForPage(Query, Class)** – executa a seleção junto ao servidor Solr e converte os documentos selecionados para objetos do tipo da classe passada como parâmetro, apresentando os mesmos em uma estrutura paginada;
- **queryForFacetPage(FacetQuery, Class)** – executa a seleção junto ao servidor Solr e converte os documentos resultantes como feito em **queryForPage()**. Adicionalmente, dados de faceting requisitados na query passada como parâmetro são agregados à página resultante;
- **queryForHighlightPage(HighlightQuery, Class)** – executa a seleção junto ao servidor Solr e converte os documentos resultantes como feito em **queryForPage()**. Além dos documentos em suas versões originais, é agregada a este tipo de resultado uma versão dos campos de textos de cada documento com os termos de busca ressaltados;
- **queryForTermsPage(TermsQuery)** – executa uma seleção por termos junto ao servidor Solr. Porém, o resultado não são documentos convertidos, e sim a contagem de documentos que são selecionados por cada termo informado no objeto **TermsQuery** fornecido;
- **queryForCursor(Query, Class)** – executa uma seleção junto ao servidor Solr retornando um **Cursor** que, diferentemente de um objeto do tipo **Page**, é otimizado para percorrer uma grande quantidade de documentos;
- **queryForGroupPage(Query, Class)** – executa a seleção junto ao servidor Solr e converte os documentos resultantes como feito em **queryForPage()**. Adicionalmente, dados de agrupamento requisitados na query passada como parâmetro são agregados à página resultante;
- **queryForStatsPage(Query, Class)** – executa a seleção junto ao servidor Solr e converte os documentos resultantes como feito em **queryForPage()**. Adicionalmente, dados estatísticos de colunas requisitados na query passada como parâmetro são agregados à página resultante;
- **getById(Serializable, Class)** – executa uma operação de busca pelo identificador único de um documento em tempo real. Este método retornará documentos mesmo que estes não estejam persistidos de forma durável;
- **commit()** – aplica todas as alterações no servidor Solr de forma definitiva. Após este processo os dados estarão armazenados de forma durável. Uma variação deste método é o **softCommit()**, que não garante que os documentos em questão sejam persistidos de forma durável, mas que se satisfaz a partir do momento em que seja possível a utilização destes dados em queries que o servidor venha a executar. Neste caso, a persistência dos dados para uma forma durável é feita de forma assíncrona;
- **rollback()** – tem por intuito desfazer alterações que por ventura tenham sido feitas em documentos, mas que ainda não foram persistidos utilizando **commit()** ou **softCommit()**;
- **execute(SolrCallback)** – permite que o desenvolvedor implemente a classe **SolrCallback**, que provê acesso ao servidor Solr (**SolrClient**). Desta forma, operações que ainda não sejam suportadas pelo Spring Data Solr, mas que sejam possíveis utilizando SolrJ, podem ser executadas mesmo através do Spring Data Solr.

A aplicação exemplo

A fim de demonstrar uma aplicação com Spring Data Solr, realizaremos a implementação de uma interface REST que servirá como back-end para uma listagem de produtos.

Requisitos

Para esta prática, os seguintes requisitos serão levados em consideração:

1. Um produto tem as seguintes propriedades: nome, descrição, categoria, preço, popularidade e disponibilidade em estoque;
2. A interface de acesso deve permitir a inserção, alteração, exclusão por id e a recuperação de produtos também por id (a intenção aqui é a demonstração dos métodos mais comuns, necessários a qualquer aplicação com acesso a um repositório de dados);
3. A interface de acesso deve permitir uma listagem de produtos por categoria e disponíveis em estoque (demonstramos assim como queries podem ser executadas em Solr derivadas do nome do método);
4. A interface de acesso deve permitir uma busca textual (considerando título e descrição) por produtos. O resultado deve ser categorizado, apresentando um total por categorias (busca textual e faceting são os pontos demonstrados neste ponto).

Implementação

Para a concretização dos requisitos elencados no tópico anterior, utiliza-se o projeto configurado anteriormente (no tópico “Spring Data Solr”) como ponto de partida. Desta forma, a implementação pode ser resumida aos seguintes passos:

1. **Configuração geral da aplicação** – a configuração necessária para este passo já fora feita anteriormente (no tópico “Spring Data Solr”) e será reutilizada;

2. Adição de novas propriedades ao modelo de dados – campos adicionais precisam ser inseridos, como descrição, popularidade e disponibilidade em estoque;
3. Adição dos novos métodos requeridos à camada de acesso a dados **ProductDao** – implementação dos métodos adicionais que incluem a busca textual e a listagem de produtos disponíveis em estoque por categoria;
4. Adição dos novos mapeamentos ao controle REST para o serviço web **ProductController** – com o intuito de acessar algumas funcionalidades implementadas na interface de acesso a dados, alguns métodos serão adicionados.

Modelo

A implementação da classe de modelo (**Product**) necessária para a realização dos requisitos elencados pode ser resumida a uma classe conforme o código exposto na **Listagem 16**. A implementação dos setters e getters para as propriedades do produto devem ser adicionadas no local com o comentário designado para os mesmos. Ainda nesta listagem, repara-se a presença da anotação **@SolrDocument**, que pode ser utilizada para definir o núcleo Solr no qual os documentos serão armazenados.

Listagem 16. Extensão do modelo com campos adicionais introduzidos nos requisitos.

```
@SolrDocument(solrCoreName = "products")
public class Product {

    @Id @Field private String id = UUID.randomUUID().toString();
    @Field private String name;
    @Field private String description;
    @Field private String category;
    @Field private float price;
    @Field private int popularity;
    @Field("inStock") private boolean available;

    // setters and getters

}
```

Quando um objeto é convertido para um documento Solr, os nomes nos documentos serão os mesmos dos campos na classe sendo convertida. Assim sendo, tem-se em um documento os campos **name**, **description**, **category**, **price** e **popularity**. Para o campo **available**, o mapeamento refere-se ao campo **inStock** em um documento Solr (como pode ser verificado pelo nome indicado na anotação **@Field**).

Data Access Object (DAO)

A implementação da lógica de acesso a dados se resume à escrita da interface para acesso a documentos do tipo **Product**, uma vez que as queries requeridas podem ser derivadas do nome do método e anotações do mesmo. Assim sendo, tem-se o código apresentado na **Listagem 17**.

Listagem 17. Implementação da classe de acesso a dados conforme os requisitos da aplicação.

```
public interface ProductDao extends SolrCrudRepository<Product, String> {

    @Query("*,*")
    Page<Product> findAllPaged(Pageable pageRequest);

    Page<Product> findByCategoryAndAvailableTrue(String category,
                                                Pageable pageRequest);

    @Facet(fields = "category", minCount = 1)
    @Query("title?:0 description?:0")
    FacetPage<Product> find(String searchString, Pageable pageRequest);

}
```

Nota-se que tudo o que é preciso para que uma interface seja qualificada como uma interface de acesso a dados Solr é a extensão da interface **SolrRepository** ou da interface **SolrCrudRepository**.

A seguir temos uma descrição dos métodos declarados nessa listagem:

- **findByCategoryAndAvailableTrue(String, Pageable)**: método que efetua uma busca no campo **category** que deve coincidir com o parâmetro informado (instruído por **findByCategory**) e deve ter o campo **available** configurado como verdadeiro (instruído por **AndAvailableTrue**), obedecendo aos parâmetros de paginação (**Pageable**);
- **find(String, Pageable)**: tem por função efetuar uma busca do texto dado como parâmetro nos campos **title** e **description** (como declarado na query utilizando **@Query**), obedecendo aos parâmetros de paginação (**Pageable**). Ainda como retorno desta query, é desejada uma contagem de produtos utilizando a categoria do mesmo (**category**) como parâmetro de distinção, o que se dá pelo uso de **@Facet**. Para que a contagem de documentos em cada categoria não retorne também as categorias com 0 documentos, é configurado o **minCount** para 1;
- **findAllPaged(Pageable)**: método que retornará todos os documentos (**@Query("*,*")**), também obedecendo aos parâmetros de paginação informados (**Pageable**).

Controller

A última etapa necessária para que possamos constatar o funcionamento da implementação do nosso repositório de forma prática é a implementação de uma interface de acesso. Considerando que um **@RestController**, nomeado **ProductController** e exposto na **Listagem 11**, já foi previamente escrito, o incrementaremos com as operações adicionais aqui requeridas.

Para a inserção de documentos (**Product**), tem-se um método implementado como exposto na **Listagem 18** e seu uso exemplificado na **Listagem 19**, que espera uma requisição do tipo PUT ao recurso raiz (/).

No caso da exclusão, espera-se uma requisição utilizando o verbo DELETE para o recurso **/id/**, onde **{id}** é a identificação única do documento (**Product**) que desejamos excluir.

A implementação desse método é exposta na **Listagem 20** e seu uso pode ser verificado a seguir:

```
curl -XDELETE http://localhost:8080/5f5fe376-c596-11e4-8c80-f0defa6c59f4/
```

Listagem 18. Implementação do método save() no Controller.

```
@RequestMapping(value = "", method = RequestMethod.PUT)
public void save(@RequestBody Product product) {
    productDao.save(product);
}
```

Listagem 19. Exemplo de requisição ao método save().

```
curl -XPUT http://localhost:8080/ -H "Content-Type: application/json" -d'{
    "inStock": true,
    "popularity": 1,
    "id": "5f5fe376-c596-11e4-8c80-f0defa6c59f4",
    "price": 1250,
    "price_c": "1250.0,USD",
    "category": "Periféricos",
    "name": "Monitor 17\"",
    "description": "Monitor com caixas de som incluídas, definição UHD."
}'
```

Listagem 20. Implementação do método delete() no Controller.

```
@RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
public void delete(@PathVariable("id") String id) {
    productDao.delete(id);
}
```

Além da listagem e da exclusão, uma operação para obter um produto em específico também deve ser oferecida. E isto será disponibilizado a partir da requisição do tipo GET ao recurso /*{id}*/, onde *{id}* é a identificação única do documento (**Product**) que desejamos carregar. A implementação desse método é exposta na **Listagem 21** e seu uso pode ser verificado a seguir:

```
curl -XGET http://localhost:8080/5f5fe376-c596-11e4-8c80-f0defa6c59f4/
```

Listagem 21. Implementação do método get() no Controller.

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public Product get(@PathVariable("id") String id) {
    return productDao.findOne(id);
}
```

Outro requisito existente para o protótipo sendo desenvolvido é a necessidade de um método de seleção paginada de produtos de uma determinada categoria que estejam em estoque, e isto pode ser feito através de requisições do tipo GET ao recurso /list/{category}/{page}/{size}/, onde {category} refere-se ao nome da categoria desejada, {page} à página sendo requisitada e {size} ao número máximo de documentos (**Product**) esperados na resposta. A implementação desse método é exposta na **Listagem 22** e seu uso é apresentado a seguir:

```
curl -XGET http://localhost:8080/list/Periféricos/0/20/
```

Listagem 22. Implementação do método listByCategoryAndAvailability() no Controller.

```
@RequestMapping(value = "/list/{category}/{page}/{size}/",
    method = RequestMethod.GET)
public Page<Product> listByCategoryAndAvailability( //
    @PathVariable("category") String category, //
    @PathVariable("page") int page, //
    @PathVariable("size") int size //
) {
    return productDao.findByCategoryAndAvailableTrue(category,
        new SolrPageRequest(page, size));
}
```

E para a implementação do último requerimento a ser contemplado pelo nosso protótipo, tem-se a busca textual, que pode ser feita utilizando uma requisição do tipo GET para o recurso /search/{searchString}/{page}/{size}/, onde {searchString} é o texto que desejamos buscar nos produtos armazenados, {page} representa a página sendo requisitada e {size} o número máximo de documentos (**Product**) esperados na resposta (vide **Listagem 23**). Um exemplo de uso desse método pode ser verificado a seguir:

```
curl -XGET http://localhost:8080/search/caixas%20monitor/0/20/
```

Listagem 23. Implementação do método search() no Controller.

```
@RequestMapping(value = "/search/{searchString}/{page}/{size}/",
    method = RequestMethod.GET)
public FacetPage<Product> search( //
    @PathVariable("searchString") String searchString, //
    @PathVariable("page") int page, //
    @PathVariable("size") int size //
) {
    return productDao.find(searchString, new SolrPageRequest(page, size));
}
```

Uma versão completa do exemplo pode ser baixada do endereço indicado na seção [Links](#).

A utilização de módulos Spring Data para acesso a dados facilita o desenvolvimento de aplicações ao abstrair o desenvolvedor da tecnologia que tem por função armazenar os dados sendo geridos. Neste contexto, o Spring Data Solr provê a integração do Spring Data com servidores Solr fornecendo uma base sólida de funcionalidades prontas para uso em aplicações Spring.

A utilização de convenções e automatismos, vistos inicialmente em SolrJ e amplificados pelo Spring Data Solr, ajuda a diminuir a quantidade de código repetitivo, diminuindo assim a complexidade da aplicação.

O Solr, como qualquer outra tecnologia, possui seu foco e é uma boa opção para projetos onde a busca textual e o desempenho na seleção de dados são essenciais. Ainda assim, a substituição de um sistema de gerenciamento de banco de dados relacional

por um servidor Solr pode não ser uma boa ideia. Isto é notado, principalmente, em projetos onde a integridade referencial, a atomicidade e o isolamento são requisitos primários, funções que não são fornecidas ou são parcialmente fornecidas por um servidor Solr.

Um modelo híbrido pode ser uma boa solução para casos onde a busca textual deva ser provida, mas requisitos indiquem a utilização de um banco de dados relacional. Neste tipo de solução, o banco de dados relacional continua como uma fonte de dados primária, mas determinadas entidades são armazenadas também em um servidor Solr. A aplicação passa então a utilizar o Solr para executar queries envolvendo busca textual e também para obter pequenas estatísticas. Para o gerenciamento das duas fontes de dados, pode-se adotar o Spring Data JPA para operações em bancos de dados relacionais e o Spring Data Solr para operações junto ao servidor Solr.

Por fim, a integração Spring Boot e Spring Data Solr permite a criação de uma aplicação utilizando a abstração de acesso a dados de forma rápida e fácil, resultando em uma aplicação enxuta e com uma configuração limitada ao estritamente necessário para o domínio da aplicação.

Autor



Francisco Späth

Arquiteto de software na "Net-M, a NTT-DoCoMo Group Company", contribui com o desenvolvimento de open source software e aprecia de discutir sobre arquitetura e design de sistemas.



Links:

Endereço para download do Apache Solr.

<http://archive.apache.org/dist/lucene/solr/4.7.2/>

Endereço para download do exemplo.

<https://github.com/franciscospaeth/samples/tree/minimal-spring-data-solr-setup>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



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

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

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

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



Estrutura

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



Suporte diferenciado

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



Serviços

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



1-to-1

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



Porta 80
WEB HOSTING

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

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

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

DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

FORMAÇÃO DESENVOLVEDOR JAVA

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

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer

