



Edição 153 :: R\$ 14,90

 DEVMEDIA



DESTAQUE:
Introdução a web services RESTful
Aprenda a criar web services
de acordo com esse padrão

**JTA e a demarcação de
transações globais em EJBs**
Dominando os principais conceitos

DataTable no PrimeFaces
Domine os recursos para a criação
de tabelas ricas e dinâmicas

JAVA EE NA NUVEM

Como planejar e
migrar aplicações
para a AWS



MVP

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

APlique esse investimento
na sua carreira...

E mostre ao mercado
quanto você vale!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

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

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

 **DEVMEDIA**

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultora Técnica Anny Caroline (annycarolinegnr@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araújo

Distribuição

FC Comercial e Distribuidora S.A

Rua Teodoro da Silva, 907, Grajaú - RJ

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

Atendimento ao leitor

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

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

Sumário

06 – Explorando o componente DataTable do PrimeFaces

[*Eduardo Felipe Zambom Santana e Luiz Henrique Zambom Santana*]

Destaque – Mentoring

16 – Como migrar seu serviço JMS Java EE para a nuvem da Amazon

[*Daniel Stori*]

28 – Introdução a web services RESTful

[*Miguel Diogenes Matrakas*]

40 – JTA e a demarcação de transações globais em EJBs

[*Michael Henrique R. Pereira*]

Programador Java: Por onde começar?

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



DEVMEDIA

Explorando o componente DataTable do PrimeFaces

Conheça os recursos para a criação de listagens com uma interface rica utilizando o PrimeFaces

JavaServer Faces é a principal especificação para o desenvolvimento de aplicações web na plataforma Java. Com o objetivo de facilitar a implementação de sistemas para a internet com interfaces baseadas em componentes, sua versão 1.0 foi lançada em 2004, substituindo os frameworks baseados em ações, principalmente o Struts, que eram os mais utilizados na época. Atualmente, a especificação está na versão 2.2, lançada em 2013, e existem diversas distribuições importantes, como o Mojarra, implementação de referência, e o MyFaces, criado pela Apache.

Apesar de as interfaces construídas com JSF serem baseadas em componentes, as implementações dessa spec incluem apenas os componentes básicos do HTML, como campos de formulários, botões e links. Devido a isso, foram criados diversos projetos que disponibilizam uma grande variedade de componentes para a construção de interfaces com o usuário, como o RichFaces, ICEfaces e PrimeFaces. Nos últimos anos, no entanto, o PrimeFaces dominou o mercado ao se tornar o mais completo e fácil de utilizar. A **Figura 1** mostra a tela do Google Trends que compara a evolução das buscas por essas três bibliotecas. Nela é possível observar o crescimento do PrimeFaces em relação aos principais concorrentes.

Esse crescimento pode ser justificado, entre outros motivos, pela evolução do framework desde a sua primeira versão, lançada em 2009. Na versão atual, a 5.2.1, o PrimeFaces disponibiliza mais de 100 componentes de interface para diversas funcionalidades, a saber: campos de formulários, listagens, galerias de imagens, captchas, dashboards e gráficos. Entre eles, um dos mais importantes e empregados é o DataTable, que possibilita a criação de tabelas de objetos e que disponibiliza uma

Fique por dentro

Este artigo é útil para conhecer os diversos recursos do componente DataTable do PrimeFaces, como a criação de filtros, ordenação, cabeçalhos e rodapés. O PrimeFaces é um dos principais frameworks para a construção de interfaces gráficas para o JavaServer Faces (JSF). Isso porque ele disponibiliza uma grande quantidade de componentes para construir diversos tipos de telas, como formulários, listagens, gráficos e mapas. Entre esses componentes, o DataTable é um dos mais utilizados, pois a maioria das aplicações necessita de tabelas para, por exemplo, exibir a grade de um curso em um sistema de controle acadêmico, listar os alunos, disciplinas e professores.

grande quantidade de recursos; desde os mais simples, como ordenação e filtragem, até os mais avançados, como *lazy load* e *drag and drop*.

Com base nisso, este artigo demonstrará a utilização do componente DataTable e diversas opções que ele disponibiliza. Para isso, será criada uma pequena aplicação para a listagem de alunos, e a partir disso serão desenvolvidas diversas versões da mesma listagem para demonstrar e explicar os recursos desse componente.

Configuração do JSF e do PrimeFaces

Para a implementação de uma aplicação com o PrimeFaces é necessário configurar o projeto, adicionando as bibliotecas que serão utilizadas e as propriedades do JSF nos arquivos de configuração. Em nosso exemplo, para a gestão das dependências será empregado o Apache Maven, como mostra a **Listagem 1**, onde é apresentado o arquivo *pom.xml* com as dependências do PrimeFaces 5.1 e do JSF 2.2.

Além dessas dependências, ainda é necessário adicionar o framework POI, que manipula planilhas Excel, e o iText, para a

geração de arquivos PDF. Eles são necessários porque em um dos exemplos será criado um DataTable que permite a exportação dos dados da tabela para arquivos XLS e PDF.

Ademais, o projeto apresentado neste artigo foi desenvolvido com a IDE Eclipse e o servidor web Apache Tomcat 8, no entanto, qualquer IDE que suporte o Maven e qualquer servidor web ou de aplicação podem ser adotados.

Visto que a aplicação será um projeto web, é necessário criar o arquivo *web.xml* no diretório *WEB-INF*. O ponto principal desse arquivo é a configuração do JavaServer Faces, que deve ser feita com a inclusão do servlet JSF, implementado pela classe *javax.faces.webapp.FacesServlet*, na tag *<servlet>*, e a configuração do mapeamento das requisições que serão redirecionadas para esse servlet; no caso desse exemplo, todos os endereços com final **.xhtml*. Na **Listagem 2** é mostrado o código do *web.xml*.

Listagem 1. Configuração das dependências do projeto com o Maven.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.devmedia.primefaces</groupId>
  <artifactId>datatable</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>

  <dependencies>
    <dependency>
      <groupId>org.primefaces</groupId>
      <artifactId>primefaces</artifactId>
      <version>5.1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.glassfish</groupId>
      <artifactId>javax.faces</artifactId>
      <version>2.2.0</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.poi</groupId>
      <artifactId>poi</artifactId>
      <version>3.14</version>
    </dependency>
    <dependency>
      <groupId>org.apache.poi</groupId>
      <artifactId>poi-ooxml</artifactId>
      <version>3.14</version>
    </dependency>
    <dependency>
      <groupId>com.lowagie</groupId>
      <artifactId>iText</artifactId>
      <version>2.1.7</version>
    </dependency>
  </dependencies>
</project>
```

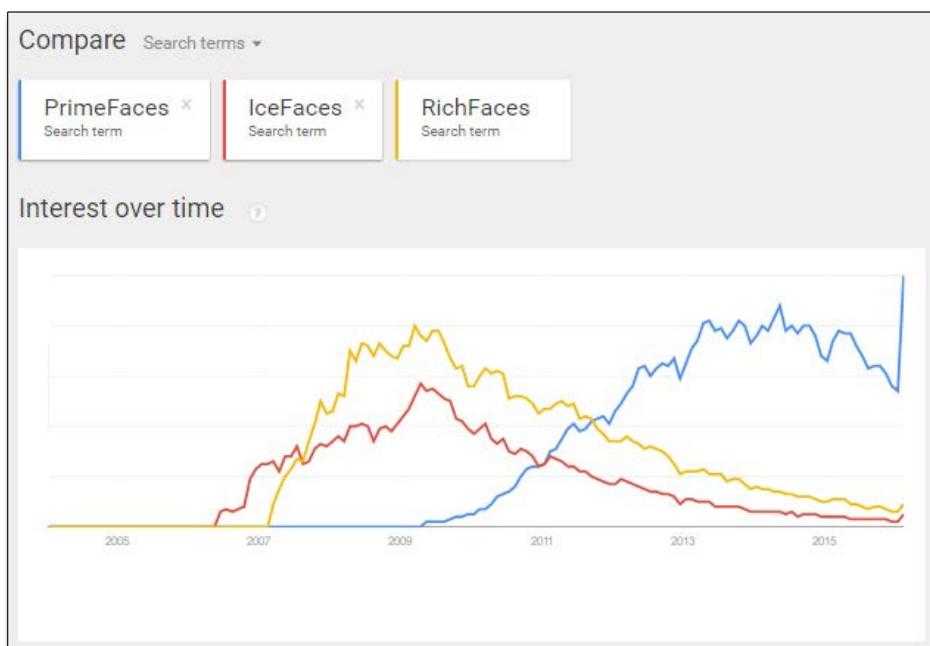


Figura 1. Interesse dos desenvolvedores pelos principais frameworks JSF

Listagem 2. Código do arquivo *web.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <display-name>com.devmedia.primefaces</display-name>
  <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>
  <welcome-file-list>
    <welcome-file>/index.xhtml</welcome-file>
  </welcome-file-list>
  <error-page>
    <exception-type>javax.faces.application.ViewExpiredException
    </exception-type>
    <location>/index.xhtml</location>
  </error-page>
</web-app>
```

Explorando os recursos do DataTable

Com todas as configurações feitas, já podemos iniciar a implementação da aplicação. Visto que os exemplos utilizarão os dados de alunos de uma universidade, é necessário criar a classe **Aluno** com os atributos básicos que servirão para alimentar o componente, a saber: nome, registro acadêmico (ra), CPF, endereço, curso, telefone e data de matrícula. A **Listagem 3** mostra o código dessa classe com seus atributos. Os métodos assessores (get e set) foram omitidos para pouparmos espaço.

Explorando o componente DataTable do PrimeFaces

Listagem 3. Código da classe Aluno.

```
package com.devmedia.primefaces.model;

import java.util.Date;

public class Aluno {

    private String nome;
    private String curso;
    private String ra;
    private String endereco;
    private String telefone;
    private String cpf;
    private Date dataMatricula;

    // métodos get e set omitidos...

}
```

Para mantermos nosso foco nas funcionalidades do DataTable, vamos criar uma classe simples responsável por gerar a massa de dados a ser utilizada nos exemplos. Com esse intuito, implementamos um *managed bean*, classe do JSF que viabiliza a comunicação dos componentes da interface com o servidor. No entanto, caso o leitor tenha interesse, esse código pode ser modificado para que os dados sejam recuperados de outras fontes, como bancos de dados ou arquivos.

A Listagem 4 mostra o código de **AlunoManagedBean**, o qual possui a lista de alunos como atributo e dois métodos: o get do atributo **alunos** e **generateRandomAluno()**, que cria um objeto do tipo **Aluno**. Além disso, há o construtor da classe, que cria cinco alunos ao chamar o método **generateRandomAluno()** repetidas vezes.

Para que os dados que serão mostrados nas tabelas não fiquem repetitivos, criamos um vetor com alguns nomes de pessoas e outro com alguns nomes de cursos, para que os objetos do tipo **Aluno** tenham valores diferentes em seus atributos.

Esse *managed bean* será o suficiente para a maioria das funcionalidades de DataTable que iremos apresentar, porém alguns recursos exigirão novos métodos, que serão implementados assim que necessários. Já podemos, então, desenvolver os primeiros exemplos.

DataTable simples

O primeiro exemplo, apresentado na Listagem 5, demonstra como construir uma tabela simples, que exibe os dados sem qualquer recurso avançado. Como é possível notar, para criar um DataTable deve ser declarada a tag **<p:dataTable>**. Nela, o atributo **var** indica o nome da variável que será utilizada na tabela para acessar os valores do objeto **Aluno**; o atributo **value** indica a fonte dos dados que serão exibidos, no caso, o método **getAlunos()** do MB **alunoMB**; e **style** define apenas o tamanho da tabela na tela.

Note que para cada atributo da classe **Aluno** inserimos em **<p:dataTable>** uma tag **<p:column>**.

Listagem 4. ManagedBean que cria a massa de dados a ser utilizada nas tabelas.

```
package com.devmedia.primefaces.mb;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;

import com.devmedia.primefaces.model.Aluno;

@ManagedBean(name="alunoMB")
@ViewScoped
public class AlunoManagedBean {

    private List<Aluno> alunos = new ArrayList<Aluno>();

    public AlunoManagedBean() {
        for (int i = 0; i < 5; i++) {
            alunos.add(generateRandomAluno());
        }
    }

    public String[] nomes = {"Eduardo", "Luiz", "Henrique", "Felipe", "Bruna", "Brianda", "Sonia", "Carlos"};
    public String[] cursos = {"Ciência da Computação", "Medicina", "Direito", "Engenharia", "Arquitetura"};

    public List<Aluno> getAlunos() {
        return alunos;
    }

    public Aluno generateRandomAluno() {
        int indiceNome = (int) Math.floor(Math.random()*7);
        int indiceCurso = (int) Math.floor(Math.random()*4);
        Aluno aluno = new Aluno();
        aluno.setNome(nomes[indiceNome]);
        aluno.setCurso(cursos[indiceCurso]);
        aluno.setEndereco("Rua " + indiceNome);
        aluno.setCpf("123456");
        aluno.setTelefone(indiceNome * 20 + "123");
        aluno.setDataMatricula(new Date());
        aluno.setRa("4312");

        return aluno;
    }
}
```

Essa tag cria uma nova coluna e declara o atributo **headerText** para especificar o seu título, a ser exibido com destaque na tabela.

Dentro da tag **<p:column>** pode ser inserido qualquer tipo de objeto, como textos, imagens e gráficos. Como nesse exemplo mostramos apenas o valor de cada atributo da classe **Aluno**, é necessário apenas um texto utilizando a tag **<h:outputText>**. Apenas o atributo **dataMatricula** necessita de uma configuração adicional. Visto que esse campo representa uma data, é preciso indicar o formato a ser utilizado na renderização, o que é feito através da tag **<f:convertDateTime>** e seu atributo **pattern**. A Figura 2 mostra a tabela criada com esse código.

Nome	Endereço	RA	CPF	Curso	Telefone	Data Matrícula
Brianda	Rua 5	4312	123456	Medicina	100123	14/02/2016
Luiz	Rua 1	4312	123456	Direito	20123	14/02/2016
Henrique	Rua 2	4312	123456	Medicina	40123	14/02/2016
Henrique	Rua 2	4312	123456	Direito	40123	14/02/2016
Brianda	Rua 5	4312	123456	Engenharia	100123	14/02/2016

Figura 2. DataTable simples listando cinco alunos

Listagem 5. XHTML de um DataTable simples.

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

<h:head>
</h:head>

<h:body>

    <p:dataTable var="aluno" value="#{alunoMB.alunos}" style="width:800px;">
        <p:column headerText="Nome">
            <h:outputText value="#{aluno.nome}" />
        </p:column>

        <p:column headerText="Endereço">
            <h:outputText value="#{aluno.endereco}" />
        </p:column>

        <p:column headerText="RA">
            <h:outputText value="#{aluno.ra}" />
        </p:column>

        <p:column headerText="CPF">
            <h:outputText value="#{aluno.cpf}" />
        </p:column>

        <p:column headerText="Curso">
            <h:outputText value="#{aluno.curso}" />
        </p:column>

        <p:column headerText="Telefone">
            <h:outputText value="#{aluno.telefone}" />
        </p:column>

        <p:column headerText="Data Matrícula">
            <h:outputText value="#{aluno.dataMatricula}">
                <f:convertDateTime pattern="dd/MM/yyyy" />
            </h:outputText>
        </p:column>
    </p:dataTable>
</h:body>
</html>
```

DataTable com filtro e ordenação

Com os conceitos básicos de um DataTable explicados, podemos começar a usar alguns dos seus recursos. Entre eles, dois dos mais interessantes e fáceis de implementar são a ordenação e a filtragem de valores. Para viabilizar a ordenação, a única alteração necessária é adicionar o atributo **sortBy** na tag **<p:column>**, o que permite especificar por qual atributo deve ser feita a ordenação. Por sua vez, para adicionar o filtro, basta declarar o atributo **filterBy**, também na tag **<p:column>**, como indicado na Listagem 6.

Nota

Apesar do exemplo ter usado os mesmos campos da tabela para ordenação e filtro, isso não é obrigatório. É possível, por exemplo, exibir o campo nome, filtrar pelo endereço e fazer a ordenação pelo CPF, tudo em uma mesma coluna.

Listagem 6. DataTable com ordenação e filtros.

```
<h:form>
    <p:dataTable var="aluno" value="#{alunoMB.alunos}"
                  style="width:800px;">
        <p:column headerText="Nome" sortBy="#{aluno.nome}"
                  filterBy="#{aluno.nome}">
            <h:outputText value="#{aluno.nome}" />
        </p:column>
        <p:column headerText="Endereço" sortBy="#{aluno.endereco}"
                  filterBy="#{aluno.endereco}">
            <h:outputText value="#{aluno.endereco}" />
        </p:column>
        <p:column headerText="RA" sortBy="#{aluno.ra}" filterBy="#{aluno.ra}">
            <h:outputText value="#{aluno.ra}" />
        </p:column>
        <p:column headerText="CPF" sortBy="#{aluno.cpf}"
                  filterBy="#{aluno.cpf}">
            <h:outputText value="#{aluno.cpf}" />
        </p:column>
        <p:column headerText="Curso" sortBy="#{aluno.curso}"
                  filterBy="#{aluno.curso}">
            <h:outputText value="#{aluno.curso}" />
        </p:column>
        <p:column headerText="Telefone" sortBy="#{aluno.telefone}"
                  filterBy="#{aluno.telefone}">
            <h:outputText value="#{aluno.telefone}" />
        </p:column>
        <p:column headerText="Data Matrícula" sortBy="#{aluno.dataMatricula}"
                  filterBy="#{aluno.dataMatricula}">
            <h:outputText value="#{aluno.dataMatricula}">
                <f:convertDateTime pattern="dd/MM/yyyy" />
            </h:outputText>
        </p:column>
    </p:dataTable>
</h:form>
```

Além da adição dos atributos **filterBy** e **sortBy**, outra diferença da Listagem 6 para a Listagem 5 é a adição da tag **<h:form>**. Isso não era necessário no exemplo anterior porque não existia nenhum campo de formulário, porém, nesse exemplo, apesar do PrimeFaces fazer todo o trabalho, os campos de filtro que são exibidos no topo da tabela formam um formulário, e por isso é obrigatório adicionar essa tag. A Figura 3 mostra o DataTable com os campos de filtro e as setas para ordenação. Ao clicar nelas, o componente realizará a ordenação ascendente ou descendente.

Nome	Endereço	RA	CPF	Curso	Telefone	Data Matrícula
Henrique	Rua 2	4312	123456	Engenharia	40123	14/02/2016
Brianda	Rua 5	4312	123456	Ciência da Computação	100123	14/02/2016
Bruna	Rua 4	4312	123456	Engenharia	80123	14/02/2016
Luiz	Rua 1	4312	123456	Medicina	20123	14/02/2016
Luiz	Rua 1	4312	123456	Medicina	20123	14/02/2016

Figura 3. DataTable com ordenação e filtro

Explorando o componente DataTable do PrimeFaces

DataTable com cabeçalho e rodapé

É possível adicionar mais informações à tabela com a inclusão de itens como cabeçalho e rodapé. Para inserir o cabeçalho é necessário declarar, dentro da tag `<p:DataTable>`, a tag `<f:facet>` com o atributo `name` especificado como `header`. Feito isso, o texto que for informado dentro dessa tag será exibido no topo da tabela. Quase da mesma forma, para inserir o rodapé também é utilizada a tag `<f:facet>`, mas com o atributo `name` definido como `footer`.

Assim, o texto que for informado será exibido na parte de baixo da tabela.

A **Listagem 7** mostra o código da tabela com cabeçalho e rodapé. Como complemento, note que é utilizada, no rodapé, a função `fn:length` — da biblioteca de funções do JSTL — para contar o número de alunos cadastrados. A **Figura 4** mostra a tabela resultante.

Listagem 7. DataTable com cabeçalho e rodapé.

```
<h:form>
<p:DataTable var="aluno" value="#{alunoMB.alunos}" style="width:800px;">
  <f:facet name="header">
    Lista de Alunos Cadastrados
  </f:facet>

  <p:column headerText="Nome" sortBy="#{aluno.nome}"
    filterBy="#{aluno.nome}">
    <h:outputText value="#{aluno.nome}" />
  </p:column>

  <p:column headerText="Endereço" sortBy="#{aluno.endereco}"
    filterBy="#{aluno.endereco}">
    <h:outputText value="#{aluno.endereco}" />
  </p:column>

  <p:column headerText="RA" sortBy="#{aluno.ra}" filterBy="#{aluno.ra}">
    <h:outputText value="#{aluno.ra}" />
  </p:column>

  <p:column headerText="CPF" sortBy="#{aluno.cpf}" filterBy="#{aluno.cpf}">
    <h:outputText value="#{aluno.cpf}" />
  </p:column>

  <p:column headerText="Curso" sortBy="#{aluno.curso}"
    filterBy="#{aluno.curso}">
    <h:outputText value="#{aluno.curso}" />
  </p:column>

  <p:column headerText="Telefone" sortBy="#{aluno.telefone}"
    filterBy="#{aluno.telefone}">
    <h:outputText value="#{aluno.telefone}" />
  </p:column>

  <p:column headerText="Data Matrícula" sortBy="#{aluno.dataMatricula}"
    filterBy="#{aluno.dataMatricula}">
    <h:outputText value="#{aluno.dataMatricula}">
      <f:convertDateTime pattern="dd/MM/yyyy"/>
    </h:outputText>
  </p:column>

  <f:facet name="footer">
    <h:outputText value="Existem " />
    <h:outputText value="#{fn:length(alunoMB.alunos)}" />
    <h:outputText value=" alunos cadastrados." />
  </f:facet>
</p:DataTable>
</h:form>
```

Lista de Alunos Cadastrados						
Nome	Endereço	RA	CPF	Curso	Telefone	Data Matrícula
Eduardo	Rua 0	4312	123456	Direito	0123	02/03/2016
Éduardo	Rua 0	4312	123456	Direito	0123	02/03/2016
Bruna	Rua 4	4312	123456	Direito	80123	02/03/2016
Luiz	Rua 1	4312	123456	Engenharia	20123	02/03/2016
Sonia	Rua 6	4312	123456	Direito	120123	02/03/2016

Existem 5 alunos cadastrados.

Figura 4. DataTable com cabeçalho e rodapé

DataTable com paginação

Outro recurso bastante interessante é a paginação. Essa opção é importante quando as tabelas podem ter muitas linhas e precisamos evitar que o usuário tenha que utilizar a barra de rolagem da página para encontrar o registro desejado.

Adicionar a paginação em uma tabela também é bastante simples. Basta incluir no `<p:DataTable>` os seguintes atributos: `rows`, que especifica o número de linhas que cada página do DataTable deve conter; e `paginator`, com o valor `true`, habilitando assim a paginação. A **Listagem 8** mostra o código do componente com esses atributos declarados.

A **Figura 5** mostra a tabela construída com o código dessa listagem. Note que para termos a paginação com oito páginas foi necessário aumentar o número de alunos criados no *managed bean* de cinco para 40.

Lista de Alunos Cadastrados						
Nome	Endereço	RA	CPF	Curso	Telefone	Data Matrícula
Eduardo	Rua 0	4312	123456	Engenharia	0123	02/03/2016
Luiz	Rua 1	4312	123456	Direito	20123	02/03/2016
Luiz	Rua 1	4312	123456	Engenharia	20123	02/03/2016
Eduardo	Rua 0	4312	123456	Engenharia	0123	02/03/2016
Brianda	Rua 5	4312	123456	Medicina	100123	02/03/2016

Existem 40 alunos cadastrados.

Figura 5. DataTable com paginação, cabeçalho e rodapé

DataTable com seleção e pop-up

Muitas vezes, principalmente quando os objetos a serem exibidos em uma listagem têm muitos atributos, pode ser melhor apresentar apenas alguns desses atributos na tabela. Nesse cenário, caso seja importante viabilizar o acesso a todos os dados, podemos utilizar um recurso que nos possibilita selecionar um item na tabela e, então, através de um pop-up, exibir os dados que tinham sido omitidos.

A **Listagem 9** mostra o código da tabela. Note que pouca coisa mudou. Apenas foram retiradas algumas colunas que serão exibidas posteriormente no pop-up. Para abrir o pop-up, definido pela tag `<p:dialog>`, o usuário deve clicar no botão que foi adicionado na tabela em uma coluna com um ícone de busca. Nesse exemplo, os atributos `nome`, `endereço` e `ra` continuam sendo exibidos na tabela, mas o atributo `curso`, por sua vez, só é exibido quando o usuário pressiona o ícone para ver os detalhes do aluno.

Listagem 8. DataTable com paginação.

```

<h:form>
  <p:dataProvider var="aluno" value="#{alunoMB.alunos}"
    style="width:800px;" rows="5" paginator="true">
    <f:facet name="header">
      Lista de Alunos Cadastrados
    </f:facet>
    <p:column headerText="Nome" sortBy="#{aluno.nome}"
      filterBy="#{aluno.nome}">
      <h:outputText value="#{aluno.nome}" />
    </p:column>
    <p:column headerText="Endereço" sortBy="#{aluno.endereco}"
      filterBy="#{aluno.endereco}">
      <h:outputText value="#{aluno.endereco}" />
    </p:column>
    <p:column headerText="RA" sortBy="#{aluno.ra}" filterBy="#{aluno.ra}">
      <h:outputText value="#{aluno.ra}" />
    </p:column>
    <p:column headerText="CPF" sortBy="#{aluno.cpf}"
      filterBy="#{aluno.cpf}">
      <h:outputText value="#{aluno.cpf}" />
    </p:column>
    <p:column headerText="Curso" sortBy="#{aluno.curso}"
      filterBy="#{aluno.curso}">
      <h:outputText value="#{aluno.curso}" />
    </p:column>
    <p:column headerText="Telefone" sortBy="#{aluno.telefone}"
      filterBy="#{aluno.telefone}">
      <h:outputText value="#{aluno.telefone}" />
    </p:column>
    <p:column headerText="Data Matrícula" sortBy="#{aluno.dataMatricula}"
      filterBy="#{aluno.dataMatricula}">
      <h:outputText value="#{aluno.dataMatricula}">
        <f:convertDateTime pattern="dd/MM/yyyy" />
      </h:outputText>
    </p:column>
    <f:facet name="footer">
      <h:outputText value="Existem " />
      <h:outputText value="#{fn:length(alunoMB.alunos)}" />
      <h:outputText value="alunos cadastrados. ">
    </f:facet>
  </p:dataProvider>
</h:form>

```

Como pode ser verificado, o código do pop-up é bastante simples. Dentro dele é criado um painel de duas colunas com o componente **<p:panelGrid>**, e dentro desse painel são colocados os atributos de aluno. A **Figura 6** mostra a tabela com as colunas removidas, a nova coluna com o ícone de busca e o pop-up aberto com os dados detalhados de um aluno.

A tabela principal ficou escurecida por causa do atributo **modal** da tag **<p:dialog>**, que indica que a tela fica congelada e só volta ao normal quando o pop-up for fechado.

DataTable com edição

Um recurso interessante do DataTable é a edição dos objetos diretamente na tabela, o que evita a necessidade de criar uma nova tela apenas para isso. Assim como as demais, a implementação dessa funcionalidade é bastante simples. Para isso, primeiro, devemos adicionar à tag **<p:dataProvider>** o atributo **editable** com o valor **true**, o que sinaliza que a tabela permite a edição de valores, e o atributo **editMode** com o valor **cell**, que possibilita que ao clicar em uma célula o campo seja habilitado para edição.

Listagem 9. DataTable que possibilita a seleção de um objeto.

```

<h:form id="form">
  <p:dataProvider var="aluno" value="#{alunoMB.alunos}"
    style="width:800px;" rows="5" paginator="true">
    <p:column headerText="Nome" sortBy="#{aluno.nome}"
      filterBy="#{aluno.nome}">
      <h:outputText value="#{aluno.nome}" />
    </p:column>
    <p:column headerText="Endereço" sortBy="#{aluno.endereco}"
      filterBy="#{aluno.endereco}">
      <h:outputText value="#{aluno.endereco}" />
    </p:column>
    <p:column headerText="RA" sortBy="#{aluno.ra}" filterBy="#{aluno.ra}">
      <h:outputText value="#{aluno.ra}" />
    </p:column>
    <p:column headerText="CPF" sortBy="#{aluno.cpf}"
      filterBy="#{aluno.cpf}">
      <h:outputText value="#{aluno.cpf}" />
    </p:column>
    <p:column headerText="Curso" sortBy="#{aluno.curso}"
      filterBy="#{aluno.curso}">
      <h:outputText value="#{aluno.curso}" />
    </p:column>
    <p:column headerText="Telefone" sortBy="#{aluno.telefone}"
      filterBy="#{aluno.telefone}">
      <h:outputText value="#{aluno.telefone}" />
    </p:column>
    <p:column headerText="Data Matrícula" sortBy="#{aluno.dataMatricula}"
      filterBy="#{aluno.dataMatricula}">
      <h:outputText value="#{aluno.dataMatricula}">
        <f:convertDateTime pattern="dd/MM/yyyy" />
      </h:outputText>
    </p:column>
    <f:facet name="footer">
      <h:outputText value="Existem " />
      <h:outputText value="#{fn:length(alunoMB.alunos)}" />
      <h:outputText value="alunos cadastrados. ">
    </f:facet>
  </p:dataProvider>
</h:form>

```



Figura 6. DataTable com pop-up

Em seguida, devemos mudar o código presente na tag **<p:column>**, declarando a tag **<p:cellEditor>** para especificar que os campos da coluna serão editáveis, e, além da tag **<h:outputText>** com o valor do atributo a ser exibido na tela, incluir um **<p:inputText>**, a ser renderizado quando o usuário selecionar uma célula para edição. A **Listagem 10** mostra as alterações feitas no código.

Explorando o componente DataTable do PrimeFaces

Listagem 10. DataTable que possibilita a edição de um objeto.

```
<h:form>
<p: dataTable var="aluno" value="#{alunoMB.alunos}"
style="width:800px;" editable="true" editMode="cell">
<p:column headerText="Nome">
<p:cellEditor>
<f:facet name="output"><h:outputText value="#{aluno.nome}" /> </f:facet>
<f:facet name="input"><p:inputText value="#{aluno.nome}" /> </f:facet>
</p:cellEditor>
</p:column>
<p:column headerText="Endereço">
<p:cellEditor>
<f:facet name="output"><h:outputText value="#{aluno.endereco}" />
</f:facet>
<f:facet name="input"><p:inputText value="#{aluno.endereco}" />
</f:facet>
</p:cellEditor>
</p:column>
<p:column headerText="RA">
<p:cellEditor>
<f:facet name="output"><h:outputText value="#{aluno.ra}" />
</f:facet>
<f:facet name="input"><p:inputText value="#{aluno.ra}" />
</f:facet>
</p:cellEditor>
</p:column>
<p:column headerText="CPF">
<p:cellEditor>
<f:facet name="output"><h:outputText value="#{aluno.cpf}" />
</f:facet>
<f:facet name="input"><p:inputText id="modelInput" value="#{aluno.cpf}" />
</f:facet>
</p:cellEditor>
</p:column>
<p:column headerText="Curso">
<h:outputText value="#{aluno.curso}" />
</p:column>
<p:column headerText="Telefone">
<h:outputText value="#{aluno.telefone}" />
</p:column>
<p:column headerText="Data Matrícula">
<h:outputText value="#{aluno.dataMatricula}" />
<f:convertDateTime pattern="dd/MM/yyyy" />
</h:outputText>
</p:column>
</p: dataTable>
</h:form>
```

Já a **Figura 7** mostra esse código em execução. Repare que o campo RA da tabela está permitindo que o usuário digite um novo valor. Como nesse exemplo os dados estão em uma lista, nenhum código adicional é necessário para atualizar os atributos dos objetos que forem modificados. Por outro lado, se os dados fossem recuperados de um banco de dados, seria necessário criar um método para atualizar os valores no banco depois de qualquer mudança na tabela.

Exportando os dados do DataTable

Outra funcionalidade bastante útil do DataTable é a exportação dos dados em diversos formatos. Para isso, devemos utilizar a tag **<p:dataExporter>**, que suporta as opções PDF, XML, XLS e CSV. No desenvolvimento dessa funcionalidade nenhuma alteração é necessária na tabela, bastando adicionar um **<p:commandLink>** para cada formato desejado.

Nome	Endereço	RA	CPF	Curso	Telefone	Data Matrícula
Sonia	Rua 6	4312	123456	Ciência da Computação	120123	16/02/2016
Felipe	Rua 3	4312	123456	Engenharia	60123	16/02/2016
Bruna	Rua 4	4312	123456	Ciência da Computação	80123	16/02/2016
Branda	Rua 5	4312	123456	Ciência da Computação	100123	16/02/2016
Sonia	Rua 6	4312	123456	Engenharia	120123	16/02/2016

Figura 7. DataTable que possibilita a edição de um objeto

Dentro dessa tag devem ser inseridos um texto ou uma figura para indicar ao usuário o tipo de arquivo que será exportado, e a tag **<p:dataExporter>** com os atributos **type**, que especifica o tipo do arquivo a ser exportado, **target**, que define o nome da tabela de onde os dados serão exportados, **filename**, sugerindo o nome do arquivo que será salvo, e **pageOnly**, onde, se o valor for **true**, exporta apenas os dados da página selecionada, e, se o valor for **false**, exporta os dados da tabela inteira. Note que esse último atributo só faz sentido para tabelas com paginação. A **Listagem 11** mostra o código da tabela com essa funcionalidade.

Nesse exemplo é viabilizada a exportação dos dados nos quatro formatos citados anteriormente. A **Figura 8** mostra uma parte do arquivo PDF e do XML gerados a partir de uma lista de alunos.

DataTable com lazy load

Quando lidamos com grandes listagens, a quantidade de memória consumida para a criação de tabelas como as que estamos apresentando pode ser um problema. Lembre-se que, diferentemente do HTML puro, que apenas renderiza os textos mostrados na tabela, no JSF são manipulados objetos inteiros, que ocupam muito mais memória. Além disso, há também o problema da quantidade de dados trafegados pela rede, o que pode fazer com que a renderização da página fique bastante lenta, por vezes dando até a impressão de que a aplicação travou.

Para resolver esse problema, o DataTable disponibiliza um recurso chamado de Lazy Load. Esse permite que os dados sejam carregados apenas quando forem solicitados para exibição na listagem, utilizando para isso a paginação. Assim, com o AJAX, são carregados apenas os registros da página que está sendo exibida. O problema do lazy load é que o código acaba ficando um pouco mais complicado, principalmente quando forem adotadas opções como ordenação e filtros; porém, esse é um recurso necessário, pois possibilita a visualização de listagens que consomem muita memória e pacotes de dados.

Para a utilização do lazy load é preciso criar duas classes auxiliares, que analisaremos a seguir, e realizar algumas mudanças no *managed bean* e no XHTML, que quase não terá alterações, como mostra a **Listagem 12**. Os únicos ajustes nesse arquivo ficam por conta da adição do atributo **lazy** com valor **true** na tag **<p: dataTable>**, explicitando que a tabela fará uso desse recurso, e a alteração do valor do atributo **value** para recuperar os dados de **AlunoLazyDataModel**, classe que ainda vamos criar.

Como será necessária a adição de código para cada atributo da classe **Aluno** que viabilizará a ordenação e o filtro, implementaremos essas funcionalidades apenas nos campos nome e curso.

Listagem 11. Exportando dados do DataTable.

```

<h:form>
  <p:dataTable id="alunos" var="aluno" value="#{alunoMB.alunos}"
    style="width:800px;" rows="5" paginator="true">
    <f:facet name="header">Lista de Alunos Cadastrados</f:facet>
    <p:column headerText="Nome" sortBy="#{aluno.nome}"
      filterBy="#{aluno.nome}">
      <h:outputText value="#{aluno.nome}" />
    </p:column>
    <p:column headerText="Endereço" sortBy="#{aluno.endereco}"
      filterBy="#{aluno.endereco}">
      <h:outputText value="#{aluno.endereco}" />
    </p:column>
    <p:column headerText="RA" sortBy="#{aluno.ra}" filterBy="#{aluno.ra}">
      <h:outputText value="#{aluno.ra}" />
    </p:column>
    <p:column headerText="CPF" sortBy="#{aluno.cpf}"
      filterBy="#{aluno.cpf}">
      <h:outputText value="#{aluno.cpf}" />
    </p:column>
    <p:column headerText="Curso" sortBy="#{aluno.curso}"
      filterBy="#{aluno.curso}">
      <h:outputText value="#{aluno.curso}" />
    </p:column>
    <p:column headerText="Telefone" sortBy="#{aluno.telefone}"
      filterBy="#{aluno.telefone}">
      <h:outputText value="#{aluno.telefone}" />
    </p:column>
    <p:column headerText="Data Matrícula" sortBy="#{aluno.dataMatricula}"
      filterBy="#{aluno.dataMatricula}">
      <h:outputText value="#{aluno.dataMatricula}">

```

```

        <f:convertDateTime pattern="dd/MM/yyyy" />
      </h:outputText>
    </p:column>
    <f:facet name="footer">
      <h:outputText value="Existem " />
      <h:outputText value="#{fn:length(alunoMB.alunos)}" />
      <h:outputText value=" alunos cadastrados. ">
    </f:facet>
  </p:dataTable>
  <h:commandLink>
    <h:outputText value="XLS" />
    <p:dataExporter type="xls" target="alunos" fileName="alunos"
      pageOnly="false" />
  </h:commandLink>
  <h:commandLink>
    <h:outputText value="PDF" />
    <p:dataExporter type="pdf" target="alunos" fileName="alunos"
      pageOnly="false" />
  </h:commandLink>
  <h:commandLink>
    <h:outputText value="CSV" />
    <p:dataExporter type="csv" target="alunos" fileName="alunos"
      pageOnly="false" />
  </h:commandLink>
  <h:commandLink>
    <h:outputText value="XML" />
    <p:dataExporter type="xml" target="alunos" fileName="alunos"
      pageOnly="false" />
  </h:commandLink>
</h:form>

```

Listagem 12. DataTable com lazy load.

```

<h:form>
  <p:dataTable var="aluno" value="#{alunoMB.lazyModel}"
    style="width:800px;" rows="5" paginator="true" lazy="true">
    <p:column headerText="Nome" sortBy="#{aluno.nome}"
      filterBy="#{aluno.nome}">
      <h:outputText value="#{aluno.nome}" />
    </p:column>
    <p:column headerText="Endereço">
      <h:outputText value="#{aluno.endereco}" />
    </p:column>
    <p:column headerText="RA">
      <h:outputText value="#{aluno.ra}" />
    </p:column>
    <p:column headerText="CPF">
      <h:outputText value="#{aluno.cpf}" />

```

```

    </p:column>
    <p:column headerText="Curso" sortBy="#{aluno.curso}"
      filterBy="#{aluno.curso}">
      <h:outputText value="#{aluno.curso}" />
    </p:column>
    <p:column headerText="Telefone">
      <h:outputText value="#{aluno.telefone}" />
    </p:column>
    <p:column headerText="Data Matrícula">
      <h:outputText value="#{aluno.dataMatricula}">
        <f:convertDateTime pattern="dd/MM/yyyy" />
      </h:outputText>
    </p:column>
  </p:dataTable>
</h:form>

```

Contudo, caso o leitor deseje fornecer essas opções nos outros atributos, basta seguir o padrão utilizado nos campos mencionados. Para facilitar a explicação, retiramos as funcionalidades apresentadas nos exemplos anteriores, como os cabeçalhos, o pop-up e a exportação dos dados da tabela.

Feito isso, para viabilizar o lazy load é necessário, ainda, implementar uma nova classe, chamada **AlunoLazyDataModel**, que será responsável pela busca

Nome	Endereço	RA	CPF	Curso
Eduardo	Rua 0	4312	123456	Medicina
Luiz	Rua 1	4312	123456	Medicina
Luiz	Rua 1	4312	123456	Cincia da Computao
Brianda	Rua 5	4312	123456	Engenharia
Eduardo	Rua 0	4312	123456	Engenharia
Brianda	Rua 5	4312	123456	Direito
Felipe	Rua 3	4312	123456	Cincia da Computao
Felipe	Rua 3	4312	123456	Direito
Henrique	Rua 2	4312	123456	Medicina
Eduardo	Rua 0	4312	123456	Direito

Figura 8. PDF e XML gerados a partir de uma lista de alunos

```

<?xml version="1.0"?>
- <alunos>
  - <aluno>
    <nome>Eduardo</nome>
    <endereco>Rua 0</endereco>
    <ra>4312</ra>
    <cpf>123456</cpf>
    <curso>Medicina</curso>
    <telefone>0123</telefone>
    <data_matricula>20/03/2016</data_matricula>
  </aluno>
  - <aluno>
    <nome>Luiz</nome>
    <endereco>Rua 1</endereco>
    <ra>4312</ra>
    <cpf>123456</cpf>
    <curso>Medicina</curso>
    <telefone>0123</telefone>

```

Explorando o componente DataTable do PrimeFaces

dos dados em um data source; no caso desse exemplo, a lista de alunos. A **Listagem 13** mostra o código dessa classe, que possui dois métodos: o construtor, que recebe como parâmetro a lista de alunos que será a fonte de dados da tabela; e o método **load()**, que faz toda a manipulação da lista, como a ordenação, os filtros e a paginação, para possibilitar o lazy load.

A primeira ação do método **load()** é fazer uma iteração sobre todos os objetos **Aluno** que serão exibidos na tabela para verificar se eles respeitam os filtros passados pelo usuário da aplicação. Como o usuário pode utilizar mais de um filtro ao mesmo tempo, o PrimeFaces cria um **Map**, onde o nome do filtro (sempre o nome da coluna na tabela) é a chave, e o dado que o usuário digitou no filtro é o valor. Com esse **Map**, é feita uma iteração sobre todos os filtros comparando o nome da coluna em que foi feito o filtro para encontrar onde o usuário digitou um valor; no caso desse exemplo, podem ser digitados dados nas colunas nome e curso.

Logo após, com o método **startsWith()** da classe **String**, é verificado se o atributo que será filtrado de um objeto **Aluno** coincide com o valor passado pelo usuário no filtro. Caso o valor coincida, é analisado o próximo filtro, caso contrário, não é necessário verificar-lo, pois para ser mostrado na tabela o objeto deve respeitar todos os filtros. Assim, com o comando **break**, a iteração é interrompida e passa-se para a verificação do próximo objeto.

A segunda parte do método faz a ordenação dos dados da tabela

apenas utilizando o método **sort()** da classe **Collections**. No entanto, ainda será necessário implementar a classe **LazySorter** para fazer a ordenação dos objetos, pois é obrigatório definir quais os campos da classe **aluno** serão utilizados para isso.

Ainda no método **load()**, é recuperado o tamanho total da lista, necessário para saber quantas páginas serão requeridas para mostrar todos os dados da tabela.

Por último, é selecionado um subconjunto da lista, com o método **subList()** da classe **List**, para retornar apenas o que será exibido na página selecionada da tabela. O parâmetro **first** do método **subList()** indica qual é o primeiro valor que deve ser exibido, e o **pageSize** indica a quantidade de objetos que devem ser mostrados na tabela em cada página. Conforme mencionado anteriormente, adotando o lazy load o código fica mais complexo, mas sem essa funcionalidade a manipulação de uma tabela com uma quantidade muito grande de registros fica prejudicada, pois a página pode demorar para ser renderizada e a quantidade de memória utilizada pode ser muito alta.

E como já mencionado, é necessário, ainda, criar a classe **LazySorter** para possibilitar a organização dos objetos na tabela. Essa classe é uma implementação da interface **Comparator**, que deve ser desenvolvida para o método **sort()** funcionar. Nessa classe são definidos o construtor, que recebe como parâmetros o nome do atributo da classe **Aluno** que deve ser usado para a

Listagem 13. Código da classe AlunoLazyDataModel.

```
package com.devmedia.primefaces.lazyload;

// imports omitidos...

public class AlunoLazyDataModel extends LazyDataModel<Aluno> {

    private List<Aluno> datasource;

    public AlunoLazyDataModel(List<Aluno> datasource) {
        this.datasource = datasource;
    }

    @Override
    public List<Aluno> load(int first, int pageSize, String sortField, SortOrder sortOrder,
                           Map<String, Object> filters) {
        List<Aluno> data = new ArrayList<Aluno>();

        // filtro dos objetos do DataTable
        for (Aluno aluno : datasource) {
            boolean match = true;
            if (filters != null) {
                for (Iterator<String> it = filters.keySet().iterator(); it.hasNext();) {
                    String filterProperty = it.next();
                    Object filterValue = filters.get(filterProperty);

                    String fieldValue = "";
                    if (filterProperty.equals("nome")) {
                        fieldValue = aluno.getNome();
                    }

                    if (filterProperty.equals("curso")) {
                        fieldValue = aluno.getCurso();
                    }

                    if (filterValue == null || fieldValue.startsWith(filterValue.toString())) {
                        match = true;
                    } else {
                        match = false;
                        break;
                    }
                }
            }

            if (match) {
                data.add(aluno);
            }
        }

        // ordenação dos objetos do DataTable
        if (sortField != null) {
            Collections.sort(data, new LazySorter(sortField, sortOrder));
        }

        // calcula o número de registros da tabela para fazer a paginação
        int dataSize = data.size();
        this.setRowCount(dataSize);

        // paginação do DataTable
        if (dataSize > pageSize) {
            try {
                return data.subList(first, first + pageSize);
            } catch (IndexOutOfBoundsException e) {
                return data.subList(first, first + (dataSize % pageSize));
            }
        } else {
            return data;
        }
    }
}
```

ordenação e a ordem da lista, se ascendente ou descendente, e o método **compare()**, que é um método da interface **Comparator** no qual é verificado se a comparação será pelo nome do aluno ou pelo curso, pois é por esses dois campos que o usuário poderá fazer a ordenação na tabela. A **Listagem 14** mostra o código da classe **LazySorter**.

Por fim, para a implementação do Lazy Load também é necessária uma pequena alteração no *managed bean* com o objetivo de incluir o atributo **lazyModel**, a ser acessado no XHTML da tabela. A **Listagem 15** mostra o código com a alteração que deve ser feita.

O PrimeFaces é um excelente framework para a criação de interfaces ricas com o JSF, principalmente por causa de seu grande conjunto de componentes com diferentes opções para atender a

Listagem 14. Classe que faz a ordenação dos objetos da classe Aluno.

```
package com.devmedia.primefaces.lazyload;

import java.util.Comparator;
import org.primefaces.model.SortOrder;

import com.devmedia.primefaces.model.Aluno;

public class LazySorter implements Comparator<Aluno> {

    private String sortField;

    private SortOrder sortOrder;

    public LazySorter(String sortField, SortOrder sortOrder) {
        this.sortField = sortField;
        this.sortOrder = sortOrder;
    }

    public int compare(Aluno aluno1, Aluno aluno2) {

        String value1 = "";
        String value2 = "";
        if (this.sortField.equals("nome")) {
            value1 = aluno1.getNome();
            value2 = aluno2.getNome();
        }

        if (this.sortField.equals("curso")) {
            value1 = aluno1.getCurso();
            value2 = aluno2.getCurso();
        }

        int value = ((Comparable) value1).compareTo(value2);

        return sortOrder.ASCENDING.equals(sortOrder) ? value : -1 * value;
    }
}
```

Listagem 15. Alteração necessária no managed bean.

```
@ManagedBean(name="alunoMB")
@ViewScoped
public class AlunoManagedBean {

    private LazyDataModel<Aluno> lazyModel;
    // get e set do atributo lazyModel
    ...
}
```

diversos tipos de requisitos. Este artigo explorou essa variedade através de exemplos utilizando o DataTable.

Além dele, alguns dos principais componentes do PrimeFaces são: o **AutoComplete**, que exibe um campo de formulário com sugestões, recuperadas de uma fonte de dados, para completar o texto sendo informado pelo usuário; o **Editor**, para a criação de uma caixa de texto com a opção de adicionar formatação, como verificamos em editores de texto; o **Tree**, que organiza uma coleção de dados em formato de árvore; o **Schedule**, para mostrar um calendário, onde é possível fazer anotações em datas ou agendar eventos; e o **BarCode**, para a criação de códigos de barras em diversos formatos.

Como o framework é constantemente atualizado com novos componentes e novas funcionalidades, recomendamos que o leitor acesse constantemente seu blog oficial e, em caso de dúvidas sobre os componentes ou erros no framework, entre em contato com sua comunidade, bastante ativa, ou ainda com alguns dos líderes do projeto, que respondem a comentários e problemas diretamente no fórum oficial do projeto.

Autor



Eduardo Felipe Zambom Santana

ezambomsantana@gmail.com

É bacharel e mestre em Ciência da Computação pela UFSCar.

Possui mais de 10 anos de experiência em programação.

Atualmente é aluno de doutorado na USP e professor na Universidade Anhembi Morumbi.



Autor



Luiz Henrique Zambom Santana

lhzsantana@gmail.com

É bacharel e mestre em Ciência da Computação. Atualmente cursa doutorado também em Ciência da Computação na UFSC.

Possui mais de 10 anos de experiência em programação Java e há dois anos atua com tecnologias de Big Data. Trabalhou em projetos para grandes empresas no Brasil, Argentina e Alemanha. Atualmente é consultor de Elasticsearch.



Links:

Implementação de referência do GlassFish.

<https://glassfish.java.net/downloads/ri/>

Site oficial do PrimeFaces.

<http://primefaces.org/>

Fórum oficial do PrimeFaces.

<http://forum.primefaces.org/>

Show case com exemplos de todos os componentes do PrimeFaces.

<http://www.primefaces.org/showcase/>

Guia de usuário do PrimeFaces.

http://www.primefaces.org/docs/guide/primefaces_user_guide_5_1.pdf

Como migrar seu serviço JMS Java EE para a nuvem da Amazon

Aprenda neste artigo a utilizar serviços da nuvem pública da Amazon Web Services para construir aplicações mais leves, rápidas e com menor custo

ESTE ARTIGO É DO TIPO MENTORING

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

Definitivamente, Cloud Computing deixou de ser apenas uma tendência e tornou-se um padrão de uso de recursos computacionais de escala mundial. Diretamente relacionado ao crescimento exponencial dessa tecnologia, temos o crescimento das startups nos últimos anos, o que seria mais difícil de alcançar sem as facilidades e o baixo custo da computação em nuvem, que reduz consideravelmente o “time-to-market” de um software e permite que empresas com poucos recursos escalem suas aplicações conforme a demanda por elas aumente.

No entanto, para desenvolver um software que seja escalável e eficiente para esse ambiente, precisamos atentar para a forma como a nuvem opera, os serviços que ela oferece, o custo dos mesmos e nos livrar de paradigmas antigos de desenvolvimento, que nos serviram muito bem até agora, mas que precisam ser deixados de lado para que nossas aplicações sejam realmente competitivas em um ambiente cloud.

Para quem iniciou no desenvolvimento de software recentemente, já entrou no mercado na “era” cloud. Consequentemente, já adota (ou deve adotar), naturalmente, práticas que são mais adequadas ao deploy na nuvem, algumas vezes até mesmo sem saber disso, pois apenas são direcionados ou condicionados pelo serviço de nuvem que estão utilizando. Porém, desenvolvedores que fizeram carreira nas empresas criando aplicações corporativas devem se atentar a esse novo modelo,

Cenário

Atualmente é um fato que a Cloud Computing representa o presente e o futuro dos ambientes de execução de software, porém, para explorar todos os seus diferenciais, precisamos rever alguns conceitos relacionados à forma como implementamos nossas aplicações e assim aumentar a eficiência das mesmas nesse ambiente. Com base nisso, neste artigo apresentaremos uma aplicação corporativa que não foi planejada nem desenvolvida para a nuvem e ensinaremos como migrar o serviço JMS dessa aplicação para a nuvem pública da Amazon, a AWS (Amazon Web Services), quando aprenderemos, também, a utilizar o serviço de mensageria SQS (Simple Queue Service).

principalmente porque estamos vivendo um momento no qual as empresas estão iniciando sua jornada (termo utilizado para migração) para a nuvem. Dentro desse contexto de mudanças de paradigma, aplicações legadas são um grande desafio para a cloud, pois são menos eficientes do que aplicações que foram criadas para ela, já que essas últimas se aproveitam dos benefícios oferecidos pelos provedores de computação na nuvem.

Neste artigo iremos falar sobre computação em nuvem, seus benefícios e como construir aplicações mais aderentes ao seu modelo, principalmente nos concentrando no mundo das aplicações Java Enterprise Edition. Como exemplo, iremos implementar uma aplicação que utiliza um serviço de fila local de um container Java EE, demonstrar como substituí-lo por um serviço de fila de um provedor de nuvem e abordar os ganhos que temos ao realizar essa atualização.

Antes disso, no entanto, vamos nos aprofundar sobre os modelos de deploy e as diferenças entre eles, iniciando com uma breve introdução sobre computação em nuvem.

Computação em Nuvem

O Gartner define computação em nuvem como um *estilo de computação onde a TI escalável e elástica é entregue como serviço usando tecnologias da internet*. Em outras palavras, computação em nuvem pode ser definida como um conjunto de serviços de tecnologia, inclusive computação propriamente dita, oferecida de uma forma que você paga pelo que usar, quando usar e se usar. Fazendo uma analogia com nosso cotidiano, podemos comparar a oferta de computação em nuvem com a energia elétrica. Não precisamos ter em nossas casas ou empresas uma usina de geração de energia elétrica. Basta conectar os aparelhos na tomada e consumir a energia no período que desejarmos. A computação em nuvem é similar. Ao invés de manter um parque de máquinas e pessoas qualificadas para manter esse parque em nossas casas ou empresas, basta conectarmos nossas máquinas à internet e aos serviços de nuvem e utilizar a computação, rede, banco de dados, filas, caches e tantos outros serviços que os diversos provedores oferecem, pagando apenas por aquilo que utilizarmos. Como esperado, isso reduz consideravelmente o custo de infraestrutura de software e em alguns casos até mesmo possibilita o que antes era inviável financeiramente.

Provedores de computação em nuvem

A AWS (*Amazon Web Services*) é a pioneira nos serviços de computação em nuvem. Em consequência disso, é a mais madura e com maior quantidade de serviços oferecidos. No entanto, nos últimos anos diversas empresas investiram pesado nesse segmento e estão abraçando uma boa fatia do mercado. O Google, por exemplo, vem ganhando bastante força, assim como a IBM, Microsoft, Rackspace, entre outras. Isso sem contar os diversos fornecedores que utilizam esses provedores como base e oferecem uma abstração de alto nível (PaaS), como veremos a seguir.

Tipos de oferta — IaaS, PaaS, SaaS

Termos comuns dentro do mundo da computação em nuvem, IaaS (*Infrastructure as a Service*), PaaS (*Platform as a Service*) e SaaS (*Software as a Service*) referem-se aos

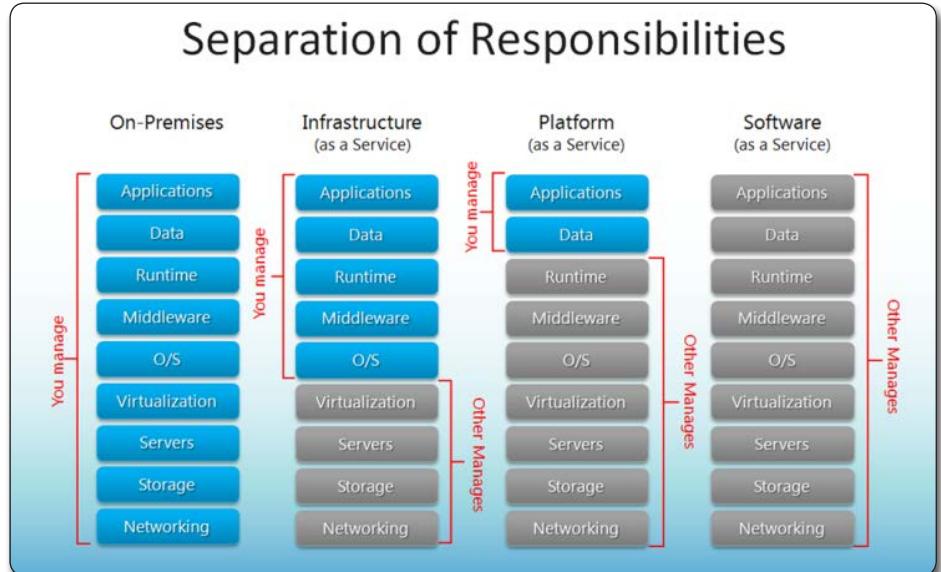


Figura 1. Da esquerda para a direita, do modelo tradicional de infraestrutura até o mais alto nível de oferta de computação em nuvem (SaaS) — Fonte: Technet

tipos de oferta que estão relacionados ao quanto de autonomia você terá sobre os recursos contratados da nuvem.

Quando usamos um serviço de IaaS estamos consumindo o nível mais baixo de serviço da nuvem, que é a computação propriamente dita. Isto é, estamos pagando para utilizar uma máquina que nos é fornecida pela internet, com os diversos recursos contidos na mesma, como CPU, memória, rede, etc. Nesse modelo, somos responsáveis por praticamente toda a gestão da máquina, como:

- Instalação e manutenção do sistema operacional e tudo que for relacionado a ele, como patches de segurança, updates, etc.;
- Instalação e configuração do ambiente necessário para executar nossa aplicação, como JVM, servidor de aplicação, etc.;
- Instalação e configuração da própria aplicação.

O IaaS é o modelo mais próximo do modelo tradicional, sendo que se difere por não termos que cuidar fisicamente da máquina, pois alguém (o provedor de serviço na nuvem) está fazendo isso por nós, mantendo a máquina ligada, resfriada e funcionando. Além disso, quando desligamos a máquina não pagamos por ela, apenas pelo espaço que a sua imagem ocupa no datacenter do provedor.

Por sua vez, quando falamos de PaaS, nossas responsabilidades diminuem e as do provedor aumentam. Nessa opção, a máquina, sistema operacional e muitas vezes até o servidor de aplicação nos são abstraídos. Assim, devemos nos preocupar exclusivamente com a nossa aplicação. Nesse tipo de oferta a quantidade de fornecedores aumenta consideravelmente e podemos citar players consolidados, como Heroku, CloudBees, Jelastic, além dos serviços de PaaS disponibilizados por grandes empresas, como IBM Bluemix, AWS Elastic Beanstalk, entre outros.

O PaaS, como esperado, é mais interessante para desenvolvedores, pois toda a operação fica por conta da nuvem. Desse modo, em muitos casos basta uma atualização no repositório de código fonte e a nova versão da aplicação já estará em produção, após um ciclo automático e integrado de build. Por outro lado, como consequência de toda essa facilidade, abre-se mão de controle, pois qualquer configuração específica de SO, de servidor de aplicação, ou mesmo de JVM, muitas vezes não é permitida. Nos casos em que precisamos disso, ir para o IaaS pode ser o melhor caminho.

A Figura 1 apresenta as responsabilidades que temos em relação ao provedor da nuvem nos diversos modelos.

No modelo SaaS, que é a oferta de mais alto nível de tecnologia na nuvem, utilizamos um software sem saber como ele foi implementado, em qual máquina, sob qual sistema operacional, servidor de aplicação ou linguagem de programação. Consumimos esse software como um serviço, pagando pelo que utilizamos. Além disso, podemos fazer uso desse software a partir de outro.

Nos enganamos quando achamos que quem consome SaaS é somente o usuário final. Como desenvolvedores, também utilizamos SaaS, por exemplo quando fazemos uso de um software através de uma API. Para que isso fique mais claro, imagine que estamos desenvolvendo um sistema que precisa enviar e-mails. Ao invés de reinventar a roda, ou “redeployar” a roda, podemos optar por um serviço como o madmimi.com, que é um software de envio de e-mail que você paga por e-mail enviado, um típico caso de SaaS que é consumido por outro software.

Além da solução adotada no exemplo, as nuvens oferecem outros tipos de serviços, facilitando bastante o ciclo de desenvolvimento e deploy. Alguns exemplos são o banco de dados (relacional ou não), filas, cache, mensageria, inteligência artificial, busca, monitoramento, entre outros.

A concorrência entre as nuvens fomenta a criação de novos serviços continuamente, aumentando o leque de opções para o desenvolvedor, que ao implementar soluções na nuvem utilizando toda essa gama de serviços e infraestrutura gerenciada, traz consigo as principais diferenças de paradigma entre o desenvolvimento tradicional e o cloud, conforme veremos no próximo tópico.

Desenvolvendo para a nuvem: adeus ao monolito

Muito tem se falado em arquitetura de microserviços, que é a antítese de uma arquitetura monolítica. Porém, esse assunto não será tratado neste artigo. O que tentaremos evidenciar aqui é que o modelo tradicional de desenvolvimento de software sempre tendeu a gerar aplicações monolíticas, em dois aspectos:

- Funcionalmente, como um grande bloco com diversas funcionalidades e sem uma separação clara entre elas;
- Não funcionalmente, como um grande deploy único em containers com o modelo “all-in-one”, onde diversas tecnologias são oferecidas dentro de uma mesma instalação, como os containers Java EE.

Isso, de forma alguma, é uma crítica aos containers Java EE, que sempre atenderam e atendem ao desenvolvimento enterprise de maneira plena. No entanto, temos que repensar a forma de utilizá-los em um ambiente cloud. Para facilitar esse processo, vamos esclarecer por que uma aplicação monolítica não se comporta bem em cloud.

Em cloud, você paga pelo que utiliza

Diferentemente do modelo tradicional, em cloud computing você paga pelo que utiliza. Sendo assim, quanto maior for a aplicação, mais recursos ela tende a consumir, seja de CPU, memória, rede e/ou disco. Como consequência disso, uma aplicação muito pesada pode ser inviável financeiramente em cloud.

O padrão de deploy de uma aplicação em cloud, por questões de custo, é iniciar com uma estrutura mais simples e escalar horizontalmente de acordo com a demanda de usuários. Por exemplo, iniciamos a execução de uma aplicação com apenas uma máquina e, conforme a quantidade de usuários aumenta, adicionamos (de maneira automática, através de serviços da nuvem) mais máquinas ao cluster. Ao contrário do modelo tradicional, no qual temos que dimensionar normalmente pela carga máxima, em cloud começamos pequenos e crescemos conforme a necessidade, o que torna nosso deploy “elástico”, ou seja, ora incrementando o número de máquinas, ora diminuindo esse número.

Mas afinal, em uma aplicação Java EE, o que podemos fazer para deixá-la mais aderente ao modelo cloud, ou até deixá-la apenas viável para esse modelo? Esse não é um desafio simples devido à característica dos containers Java EE, que oferecem, em um único deploy, tudo o que a aplicação precisa: cache, fila, mensageria, entre outros serviços. Por um lado, isso é extremamente prático, mas, por outro, faz com que cada deploy seja muito grande e caro. Uma das soluções para isso é substituir os módulos do container por serviços da nuvem, conforme veremos logo mais.

Como otimizar o Java EE na nuvem

Containers Java EE são o paraíso para muitos desenvolvedores, pois em um único pacote são oferecidos diversos serviços completamente integrados à aplicação. O problema é que tanta oferta de serviço em um mesmo container acaba induzindo o desenvolvedor a criar aplicações grandes e monolíticas, pela facilidade e agilidade de entrega. No entanto, como já discutimos, essa forma de desenvolver não é aderente à cloud, pelo elevado custo de um deploy desse porte e pela tendência de cloud ser mais eficiente com aplicações mais leves, escaláveis horizontalmente em um cluster elástico.

Conforme adiantamos na seção anterior, uma das saídas é substituir os serviços oferecidos pelos containers Java EE por serviços da nuvem, reduzindo o peso local da aplicação e delegando parte do processamento para o provedor. Em um mundo ideal, todo serviço de infraestrutura da sua aplicação que o container oferece, como cache, busca, fila e mensageria, seria substituído por um serviço da nuvem. Assim, sua aplicação, ou seu deploy, teria apenas as regras de negócio, e todo o “boiler-plate” ficaria por conta da nuvem.

Nota

O desenvolvedor deve entender que é possível criar um modelo distribuído utilizando apenas os containers Java EE, ao contrário do que iremos propor usando serviços na nuvem. Porém, a especificação Java EE não define padrões para esse tipo de implementação. Sendo assim, cada implementação de container opta por um caminho, o que torna o processo de configuração de um ambiente distribuído extremamente complexo.

Usando o serviço Amazon SQS

A partir de agora apresentaremos uma seção prática que irá explorar um exemplo no qual deixaremos de utilizar o serviço JMS do

container Java EE para adotar um serviço de mensagens da AWS, no caso, o SQS (*Simple Queue Service*). Neste momento é válido fazer uma ressalva: o JMS é uma especificação excelente e de fácil utilização através dos Message Driven Beans ou através da sua API. O que estamos propondo aqui é outra forma de desenvolver utilizando um paradigma distribuído, com o objetivo de deixar o conjunto da aplicação mais leve, ao abrir mão de um módulo inteiro de mensageria para consumir um serviço da nuvem.

Neste artigo vamos evitar detalhes sobre a AWS, assumindo que existem várias referências sobre o tema que explicam como criar uma conta, realizar o primeiro deploy, etc., a exemplo do artigo “AWS: Leve sua aplicação para a Nuvem em minutos”, publicado na Java Magazine 115. Sendo assim, poderemos nos concentrar em como substituir um serviço do container por um serviço oferecido pela nuvem, o SQS.

SQS

O *Simple Queue Service* é um serviço de fila distribuído altamente disponível e escalável que é oferecido pela nuvem da Amazon. Ele é classificado como altamente disponível porque cada mensagem enviada para uma fila é copiada para múltiplos datacenters, reduzindo consideravelmente a possibilidade de perda. E por escalável, devemos entender que não existe limite para a quantidade de mensagens ou, ao menos, não é uma preocupação nossa, pois a AWS garante o funcionamento independentemente da quantidade que utilizarmos.

Custo do serviço

De acordo com a documentação da AWS, o primeiro milhão de requisições de publicação enviadas por mês é gratuito. Após isso, é cobrado USD 0,50 por milhão de requisições, resultando em um custo médio de USD 0,00000050 por requisição. Adicionalmente, existe um custo de transferência de dados, que depende do tamanho de cada mensagem.

VotaMais - Aplicação de votação que usa filas

Neste artigo iremos desenvolver uma aplicação de votação altamente escalável, que chamaremos de VotaMais. A solução é bastante simples e possui apenas duas funcionalidades:

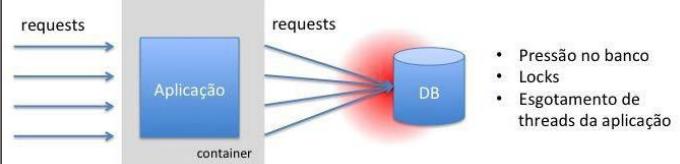
1. Votar em um candidato;
2. Ver os resultados parciais da votação.

Para suportar um alto volume de votações, soluções desse tipo devem utilizar um modelo assíncrono de atualização de dados, pois atualizações síncronas tendem a pressionar o banco com locks, consequentemente exaurindo as threads disponíveis, sejam elas do banco, ou da aplicação.

Nesse cenário, o modelo de fila se encaixa como uma luva, pois a publicação de uma mensagem em uma fila é muito mais rápida do que a atualização em banco de dados e não envolve concorrência. Além disso, o consumo da mensagem que efetivamente atualiza o banco pode ser feito de maneira serial, ou com uma estratégia que não o pressione tanto. A **Figura 2** elucida a diferença entre

os modelos, além de expor as vantagens de utilizar o modelo assíncrono, o qual será adotado neste artigo.

Modelo Síncrono de Publicação



Modelo Assíncrono com Fila

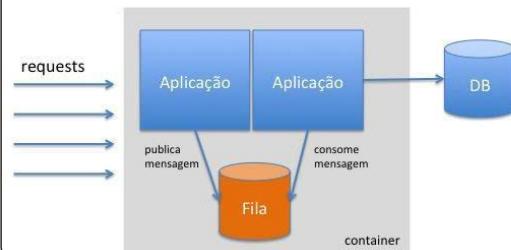


Figura 2. Diferença entre o modelo síncrono de publicação e o modelo assíncrono utilizando fila

VotaMais com WildFly e Active MQ

Antes de utilizar o SQS iremos implementar o VotaMais no modelo tradicional do Java EE, isto é, com o serviço de fila local do container. Aqui, optamos por utilizar o servidor de aplicação WildFly 10, que implementa o modelo full Java EE 7 e traz consigo o servidor de filas ActiveMQ Artemis.

A aplicação será construída utilizando uma página HTML simples, que irá realizar requisições para serviços REST JAX-RS. Esses serviços, por sua vez, irão consumir beans CDI para publicar mensagens de votação em uma fila local do WildFly. Essa fila, então, terá suas mensagens lidas por componentes que irão atualizar o banco de dados com a votação. Adicionalmente, existirá um serviço para buscar os resultados parciais da votação.

Como de costume na maior parte dos projetos, o Maven será utilizado para gestão das dependências e build. Dito isso, começamos apresentando o arquivo *pom.xml*, que deve ser criado na raiz da aplicação e ter o conteúdo apresentado na **Listagem 1**.

Como dependências, esse arquivo declara apenas a API do Java EE 7 e o conector do HSQLDB, que é o banco de dados que estamos adotando. Além disso, dois plug-ins estão configurados: o plugin de compilação padrão do Maven, para informar que desejamos que a aplicação seja compilada em Java 8, e o plugin do WildFly, chamado *wildfly-maven-plugin*, que, como veremos adiante, facilitará a execução do exemplo e os testes.

Logo após, nas **Listagens 2, 3 e 4** apresentamos os três arquivos XML necessários para rodar a aplicação a contento. Todos devem estar localizados na pasta *WEB-INF*: o *web.xml*, padrão do Java EE, o *jboss-web.xml*, para informarmos o contexto web, e o *beans.xml*, mesmo que vazio, para informar ao container que estamos utilizando o CDI.

Como migrar seu serviço JMS Java EE para a nuvem da Amazon

Listagem 1. Arquivo pom.xml da aplicação VotaMais com JMS.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javamagazine.cloud</groupId>
  <artifactId>votamais</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Vota Mais</name>
  <description>Aplicação para votação escalável</description>

  <packaging>war</packaging>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <groupId>org.wildfly.plugins</groupId>
```

```
  <artifactId>wildfly-maven-plugin</artifactId>
  <configuration>
    <server-config>standalone-full.xml</server-config>
  </configuration>
  </plugin>
</plugins>
</build>

<dependencies>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>7.0</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.3.3</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
</project>
```

Listagem 2. Arquivo web.xml da aplicação VotaMais com JMS.

```
<web-app 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"
  version="3.1">
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Listagem 3. Arquivo jboss-web.xml da aplicação VotaMais com JMS.

```
<jboss-web>
  <context-root>votamais</context-root>
</jboss-web>
```

Listagem 4. Arquivo beans.xml da aplicação VotaMais com JMS.

```
<><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>
```

Evidentemente não podemos esquecer do último XML, o *persistence.xml*, localizado na pasta *META-INF*, para configurarmos a persistência, conforme demonstra a **Listagem 5**. Esse arquivo, padrão do JPA, possui a configuração mínima para uma persistência utilizando HSQL.

Configurados esses arquivos, podemos seguir com o código propriamente dito. Por uma questão de simplicidade, iremos iniciar do front-end para o back-end. Sendo assim, precisamos criar a classe **VoteApplication**, responsável pela configuração básica do JAX-RS e onde informamos o path principal das APIs REST, conforme a **Listagem 6**.

Listagem 5. Arquivo persistence.xml da aplicação VotaMais com JMS.

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
  version="2.1">
  <persistence-unit name="votePU">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="hibernate.archive.autodetection" value="class" />
      <property name="hibernate.connection.driver_class"
        value="org.hsqldb.jdbcDriver" />
      <property name="hibernate.connection.url"
        value="jdbc:hsqldb:file:target/testdb;shutdown=true" />
      <property name="hibernate.connection.user" value="sa" />
      <property name="hibernate.flushMode" value="FLUSH_AUTO" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
  </persistence-unit>
</persistence>
```

Listagem 6. Código da classe de configuração do JAX-RS

```
@ApplicationPath("/api")
public class VoteApplication extends Application {
}
```

Note que essa classe não tem código e serve apenas para informar ao container que as APIs REST devem ser prefixadas por / *api*. Na **Listagem 7**, apresentamos o código do único controller da aplicação, o **VoteController**.

Observe que o controller é bastante simples e aceita apenas duas chamadas:

- **GET /api/vote:** que retorna o resultado parcial da votação usando um bean do CDI chamado **VoteService**, injetado no topo da classe;

- **POST /api/vote:** que publica um voto utilizando um parâmetro de URL chamado **candidate**. Esse serviço consome um bean do CDI chamado **VotePublisher** para publicar o voto.

O leitor pode observar que o resultado da votação, retornado pelo método **partial()**, é uma lista do modelo **Candidate**, que é uma entidade JPA implementada de acordo com a **Listagem 8** e que armazena os votos do candidato.

Listagem 7. Controller responsável pelas chamadas REST da solução VotaMais.

```
@Path("/vote")
public class VoteController {

    @Inject
    VoteService voteService;

    @Inject
    VotePublisher votePublisher;

    @POST
    public void vote(@QueryParam("candidate") String candidate) {
        votePublisher.publishVote(candidate);
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Candidate> partial() {
        return voteService.partial();
    }
}
```

Listagem 8. Código da entidade Candidate.

```
@Entity
public class Candidate {

    @Id
    private String name;

    @Column
    private int votes;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getVotes() { return votes; }
    public void setVotes(int votes) { this.votes = votes; }
}
```

Com as classes de front-end criadas, vamos ao desenvolvimento dos serviços de back-end, que lidam diretamente com o banco de dados, tanto na busca do resultado quanto na publicação do voto. A classe **VoteService** implementa essas funcionalidades e é apresentada na **Listagem 9**.

Repare que não existe nada de extraordinário nessa classe. Declaramos apenas um Singleton do CDI com um método transacional que publica o voto propriamente dito (**saveVote()**) e um método não transacional que busca a lista de candidatos com seus respectivos votos para exibir o resultado parcial (**partial()**).

Em seguida, implementaremos a **VotePublisher**, que faz a publicação e o consumo nas filas. Como podemos notar, **VotePublisher** deve ser criada como uma interface, pois teremos duas

implementações concretas dela: uma para o sistema de filas local do WildFly e outra para o AWS SQS. Assim, para evitar alterações no controller, utilizamos a abstração da interface, demonstrada na **Listagem 10**.

Essa interface define apenas um método, o **publishVote()**, que irá realizar a publicação do voto. A partir disso, criamos a primeira classe concreta derivada dela, a qual irá implementar esse método para publicação das mensagens em uma fila local do container. Seu código é apresentado na **Listagem 11**.

Listagem 9. Código do serviço **VoteService**, responsável pelas transações com o banco de dados.

```
@Singleton
public class VoteService {

    @PersistenceContext(name = "votePU")
    EntityManager em;

    public List<Candidate> partial() {
        return em.createQuery("SELECT c FROM Candidate c", Candidate.class)
            .getResultList();
    }
}
```

```
@Transactional
public void saveVote(String candidateName) {
    Candidate candidate = em.find(Candidate.class, candidateName);
    if (candidate == null) {
        candidate = new Candidate();
        candidate.setName(candidateName);
    }
    candidate.setVotes(candidate.getVotes() + 1);
    em.persist(candidate);
}
```

Listagem 10. Código da interface **VotePublisher**, que abstrai a implementação da fila da aplicação.

```
public interface VotePublisher {
    void publishVote(String candidateName);
}
```

Listagem 11. Código da classe **VotePublisherLocal**, que publica o voto usando a fila local do WildFly.

```
@Singleton
public class VotePublisherLocal implements VotePublisher {

    @Resource(name = "java:/ConnectionFactory")
    ConnectionFactory connectionFactory;

    @Resource(name = "java:/jms/queue/ExpiryQueue")
    Destination queue;

    @Override
    public void publishVote(String candidateName) {
        try (JMSContext context = connectionFactory.createContext();) {
            context.createProducer().send(queue, candidateName);
        }
    }
}
```

Como migrar seu serviço JMS Java EE para a nuvem da Amazon

Note, através da anotação `@Singleton`, que essa classe também é um Singleton do CDI, pois não precisamos de mais do que uma instância da mesma. Em seguida, para utilizar a API do JMS de publicação em fila, devemos injetar uma `connection factory` e uma `queue` do container. Um facilitador aqui é que tanto a `ConnectionFactory` quanto a `ExpiryQueue` são recursos que já vêm por padrão na distribuição do WildFly 10.

Analizando agora o método `publishVote()`, note que ele utiliza uma nova API de cliente JMS — lançada junto com o Java EE 7 — para enviar mensagens. Se o leitor estiver familiarizado com a API de cliente JMS do Java EE 6 ou anteriores, deve ter percebido que a nova versão está muito mais simples.

Lembre-se que o método `publishVote()` é o método chamado pelo controller `VoteController` no momento da votação (veja a [Listagem 7](#)).

Por fim, temos que implementar a classe que consome as mensagens publicadas por `publishVote()` e efetivamente realiza o voto, representada pelo Message Driven Bean `VotePublisherLocalMDB` — veja a [Listagem 12](#).

Listagem 12. VotePublisherLocalMDB, código da classe consumidora das mensagens de voto.

```
@MessageDriven(mappedName = "ExpiryQueue", activationConfig = {  
    @ActivationConfigProperty(propertyName = "acknowledgeMode",  
        propertyValue = "Auto-acknowledge"),  
    @ActivationConfigProperty(propertyName = "destination",  
        propertyValue = "java:/jms/queue/ExpiryQueue"),  
    @ActivationConfigProperty(propertyName = "destinationType",  
        propertyValue = "javax.jms.Queue")})  
public class VotePublisherLocalMDB implements MessageListener {  
  
    @Inject  
    VoteService voteService;  
  
    @Override  
    public void onMessage(Message message) {  
        try {  
            voteService.saveVote(message.getBody(String.class));  
        } catch (JMSException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Apesar do volume de configurações por anotação do MDB, basicamente estamos informando que esse bean irá consumir mensagens da fila `ExpiryQueue`, e para cada mensagem irá chamar o serviço de votação e publicar o voto.

Como em todo MDB, devemos implementar a interface `MessageListener` e como consequência disso, codificar o método `onMessage()`, que tem como parâmetro o objeto `Message`, ou seja, a mensagem que iremos receber. Dentro desse método implementamos a chamada que efetivamente persiste o voto baseado no conteúdo do corpo da mensagem.

Por último, mas não menos importante, criamos a página HTML para interagir com o usuário. Como podemos notar na [Listagem 13](#), temos uma página simples, apenas para demonstrar as funcionalidades do VotaMais.

Listagem 13. Página HTML do VotaMais.

```
<html>  
  <head>  
    <script type="text/javascript"  
      src="https://code.jquery.com/jquery-2.2.0.min.js">  
    </script>  
  </head>  
  <body>  
    <h3>VotaMais - Escolha o presidente do Brasil</h3>  
    <div id="votePanel">  
      <input type="radio" name="candidate" value="James">James<br>  
      <input type="radio" name="candidate" value="Dennis">Dennis<br>  
      <input type="radio" name="candidate" value="Ada">Ada<br>  
      <input type="radio" name="candidate" value="Martin">Martin<br><br>  
      <input type="button" id="vote" value="Votar"/>  
    </div>  
    <div id="votingPanel">  
      <h4>Votando...</h4>  
    </div>  
    <div id="resultPanel">  
      <ul id="resultList">  
      </ul>  
    </div>  
    <script>  
      $("#votingPanel").hide();  
      $("#resultPanel").hide();  
      $("#vote").click(function() {  
        $("#votePanel").hide();  
        $("#votingPanel").show();  
        $.post("/votamais/api/vote?candidate=" + $("input:checked").val(), function() {  
          $.get("/votamais/api/vote", function(result) {  
            result.forEach(function(candidate) {  
              $("#resultList").append("<li>" + candidate.name +  
                " (" + candidate.votes + " votos)</li>");  
            });  
            $("#votingPanel").hide();  
            $("#resultPanel").show();  
          })  
        });  
      });  
    </script>  
  </body>  
</html>
```

Talvez o que tenha de mais sofisticado nesse HTML sejam as chamadas AJAX implementadas com jQuery e os callbacks que escondem e exibem os divs de acordo com a ação do usuário.

Concluída essa etapa, a estrutura do projeto com seus devidos arquivos deve ficar parecida com a [Figura 3](#).

Executando a aplicação

Para executar e testar a aplicação, o leitor pode optar por dois caminhos:

- Gerar um pacote WAR através do build padrão do Maven e realizar o deploy desse pacote no WildFly 10 já instalado na máquina;
- Utilizar o plugin do WildFly do Maven, que já baixa automaticamente o WildFly 10 e realiza o deploy da aplicação. Como essa opção é mais simples e menos suscetível a falhas, a adotaremos neste artigo.

Assim, para executar a aplicação basta acessar a raiz do projeto, onde está localizado o `pom.xml`, e digitar o comando:

```
mvn org.wildfly.plugins:wildfly-maven-plugin:1.1.0.Alpha6:run
```

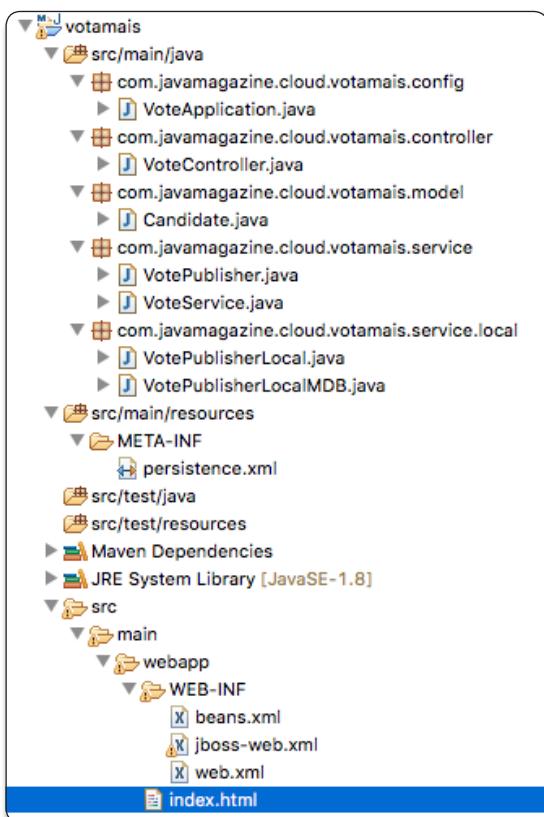


Figura 3. Estrutura do projeto VotaMais utilizando fila local do WildFly

Nota

A primeira execução pode demorar um pouco, pois o Maven irá baixar todo o container para o repositório local, mas as execuções subsequentes serão mais rápidas.

Superada essa etapa, o WildFly será inicializado e automaticamente realizará o deploy da aplicação, que deve ficar acessível através da URL <http://localhost:8080/votamais> (veja a **Figura 4**).

Nessa página, ao selecionar um candidato e clicar em *Votar*, o bloco com o resultado parcial é exibido, como demonstra a **Figura 5**.

Vejamos um resumo do que ocorre no momento em que votamos em um candidato:

1. Ao clicar em *Votar* o usuário executa uma chamada AJAX para o serviço REST implementado na classe **VoteController**. Assim, temos a chamada ao método **vote()**;

2. Esse método, através do método **publishVote()** de um bean CDI denominado **VotePublisher**, envia uma mensagem para uma fila do ActiveMQ do WildFly, fila essa chamada de **ExpiryQueue**. Essa mensagem contém o nome do candidato;

3. O **VotePublisherLocalMDB**, que é um EJB do tipo Message Driven Bean e está “escutando” a fila **ExpiryQueue**, recebe a mensagem e chama o serviço de atualização do banco, implementado pela classe **VoteService**, que utiliza funções básicas do JPA para persistir o voto no banco de dados. Nesse momento o voto é publicado;

4. Após o sucesso da chamada AJAX da publicação do voto, a página executa outra chamada para o serviço REST **VoteController**, dessa vez para o método **partial()**;

5. Esse método, então, chama o serviço **VoteService**, que apenas faz uma leitura dos votos e retorna uma lista, convertida em JSON, de acordo com anotações do JAX-RS.

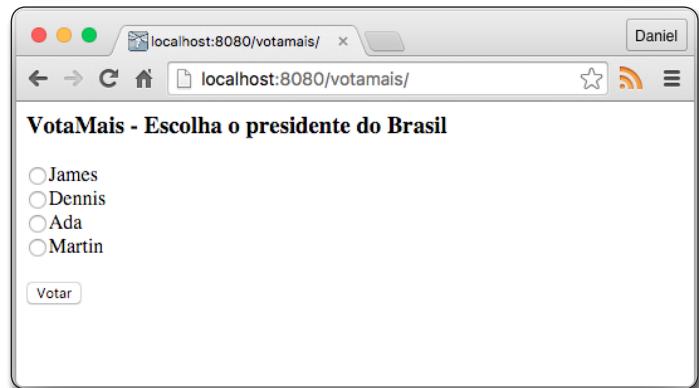


Figura 4. Página (modesta) da aplicação VotaMais

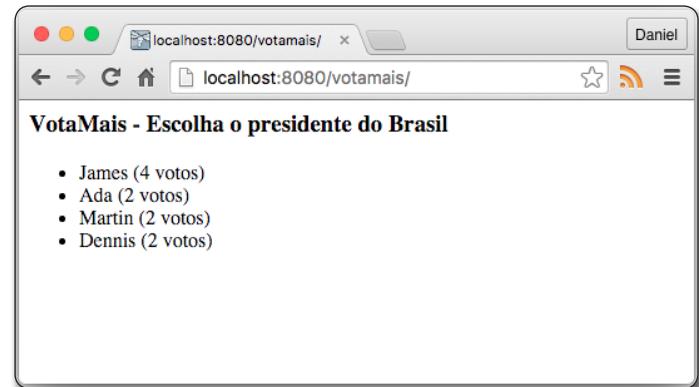


Figura 5. Página de resultado parcial do VotaMais

Neste ponto temos uma aplicação projetada e construída sobre um modelo não distribuído — como exibido na **Figura 2** — com todas as implicações explicadas anteriormente de um modelo monolítico de desenvolvimento. A partir disso, a seguir vamos implementar as modificações necessárias para alterar o serviço de fila e utilizar o SQS da Amazon Web Services.

Criando uma fila no SQS

Para a continuidade do artigo, o leitor precisa ter uma conta da AWS para que possa desfrutar dos recursos e facilidades oferecidos por essa plataforma. Sendo assim, o leitor deve criar e acessar sua conta na AWS para configurar a infraestrutura necessária; neste caso, apenas uma fila no serviço SQS, o que pode ser feito através da opção de menu *SQS* do console da Amazon, como ilustra a **Figura 6**.

Caso o leitor nunca tenha criado uma fila, a tela que é exibida pela Amazon após clicar na opção *SQS* da figura anterior é a de apresentação do SQS, com o botão *Get Started Now*.

Como migrar seu serviço JMS Java EE para a nuvem da Amazon

Ao clicar nesse botão, o leitor será conduzido para a tela de criação da fila, apresentada na **Figura 7**.

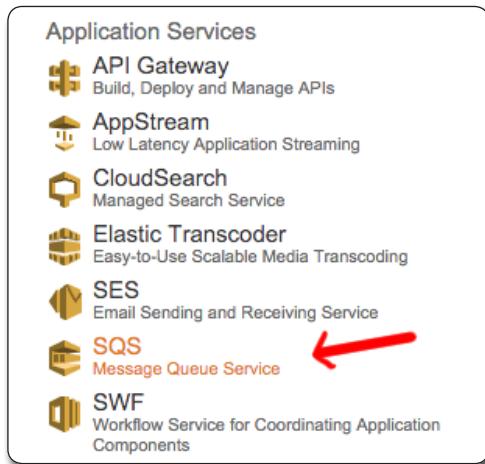


Figura 6. Trecho do console da AWS que exibe a opção SQS

A screenshot of the 'Configure votamais' dialog box. It has two main sections: 'Queue Settings' and 'Dead Letter Queue Settings'. In 'Queue Settings', fields include 'Default Visibility Timeout' (30 seconds), 'Message Retention Period' (4 days), 'Maximum Message Size' (256 KB), 'Delivery Delay' (0 seconds), and 'Receive Message Wait Time' (20 seconds). In 'Dead Letter Queue Settings', fields include 'Use Redrive Policy' (unchecked), 'Dead Letter Queue' (empty), and 'Maximum Receives' (1000). At the bottom are 'Cancel' and 'Save Changes' buttons.

Figura 7. Tela de criação de uma fila no SQS

Os campos mais importantes nessa tela são os que definem o nome da fila (no caso do exemplo, "votamais") e o "Wait Time", que será explicado adiante. Os demais parâmetros podem ser deixados com seus valores padrão, mas para conhecimento do leitor, apresentaremos uma breve descrição sobre cada um:

- **Default Visibility Timeout:** Quando um consumidor de mensagens "retira" uma mensagem da fila para processá-la, se não deletar essa mensagem depois de um determinado período, que é o *Visibility Timeout*, a mensagem volta para a fila para ser consumida por outro cliente. Por isso, é importante, como veremos a seguir, após processar a mensagem, removê-la da fila. É importante, também, alinhar esse tempo de acordo com o tempo máximo que imaginamos que nossa mensagem pode ser processada após a retirada da fila, para evitar que dois clientes processem a mesma mensagem;

- **Message Retention Period:** É o período em que uma mensagem fica na fila sem ser retirada. Após esse período o SQS remove a mensagem da fila em que foi publicada e a coloca na fila de mensagens mortas (*Dead Letter Queue*);

- **Maximum Message Size:** É o tamanho máximo de uma mensagem em bytes que desejamos receber. Mensagens que ultrapassam esse limite serão bloqueadas e uma exceção será lançada na chamada da API;

- **Delivery Delay:** Podemos escolher um tempo entre a publicação da mensagem e a disponibilidade dela para eventuais consumidores. Por padrão é zero, ou seja, ao ser pulicada a mensagem já pode ser consumida. No entanto, podem existir casos de uso nos quais desejamos "esperar um pouco" para disponibilizar a mensagem, dando, por exemplo, a opção do publicador "se arrepender";

- **Receive Message Wait Time:** É o tempo máximo que desejamos que o cliente espere por uma mensagem quando se conecta na fila. Quanto maior esse valor, mais tempo a thread que tenta fazer a leitura pode ficar ociosa. Quando criamos um cliente para se conectar na fila, ele poderá esperar pelo surgimento de mensagens pelo tempo definido nesse atributo.

Após informar os parâmetros o leitor deve clicar em *Create Queue*. Assim, a tela com a lista de filas deve ser exibida já com a fila criada, como expõe a **Figura 8**.

Obviamente a tabela apresenta 0 mensagens, pois a fila está vazia. Para integrarmos essa fila ao nosso código, o leitor deve clicar sobre a linha da fila e observar os detalhes que são exibidos na página (veja a **Figura 9**). Entre eles, o mais importante é o campo URL. Neste momento, saiba que o que chamaremos de endpoint no código que analisaremos a seguir é a parte da URL até o nome da fila.

Filter by Prefix: <input type="text" value="Enter Text..."/>			
	Name	Messages Available	Messages in Flight
	votamais	0	0

Figura 8. Lista de filas criadas no SQS

1 SQS Queue selected			
Details	Permissions	Redrive Policy	Monitoring
Name: votamais			
URL: https://sqs.us-east-1.amazonaws.com/326433034059/votamais			
ARN: arn:aws:sqs:us-east-1:326433034059:votamais			
Created: 2016-02-17 18:29:16 GMT-02:00			
Last Updated: 2016-02-17 18:29:16 GMT-02:00			
Delivery Delay 0 seconds			

Figura 9. Detalhamento de uma fila no SQS

Listagem 14. Arquivo pom.xml da aplicação VotaMais com SQS.

```

<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javamagazine.cloud</groupId>
  <artifactId>votamais-sqs</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Vota Mais</name>
  <description>Aplicação para votação escalável em cloud</description>

  <packaging>war</packaging>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-maven-plugin</artifactId>
        <configuration>
          <server-config>standalone.xml</server-config>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-java-sdk-sqs</artifactId>
      <version>1.10.52</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>2.3.3</version>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</project>
```

VotaMais – Substituindo a fila local pelo SQS

Conforme explicado anteriormente, o SQS é um serviço de fila com alta disponibilidade oferecido pela AWS. Assim como os demais serviços da empresa, possui um SDK para desenvolvimento em Java produzido e mantido pela própria Amazon.

Inicialmente, temos que fazer algumas alterações no *pom.xml* para adicionar a dependência do SDK do SQS e configurar o plugin do WildFly, como ilustra a **Listagem 14**.

Nota

Para as alterações que iremos realizar para implementar o uso do SQS, sugerimos que o leitor crie um novo projeto a partir do anterior e, então, faça as devidas modificações que serão citadas no artigo.

O leitor pode observar que a dependência em destaque é justamente o artefato do SDK do SQS, chamado **aws-java-sdk-sqs**. Além disso, na configuração do plugin do WildFly alteramos o valor do parâmetro **server-config** de **standalone-full** para **standalone**, com o objetivo de evitar o carregamento do serviço de fila local do container, pois enquanto na versão anterior precisávamos subir o container com todos os serviços (inclusive o JMS), nesta versão não precisamos, pois utilizaremos o serviço de fila da nuvem. Posteriormente, o leitor poderá comparar o tempo de subida das duas versões e perceber que a segunda é obviamente mais rápida, o que remete à proposta deste artigo: usar menos container local e mais nuvem.

Adicionalmente, o leitor deve excluir as classes **VotePublisherLocal** e **VotePublisherLocalMDB**, já que criaremos uma implementação específica de publicação para o SQS, chamada **VotePublisherSQS** (veja a **Listagem 15**).

Diferentemente dos demais beans criados até agora, como podemos notar esse é um Singleton EJB. A seguir apresentamos uma análise mais detalhada a respeito desse código.

Criando o cliente SQS

Para começar, precisamos criar um cliente para nos comunicarmos com a AWS, o que está sendo feito no método **initSQSClient()**, executado na inicialização do bean, em consequência da anotação **@PostConstruct**. Nesse método, a criação do cliente é realizada através da instanciação da classe **AmazonSQSAsyncClient**, que recebe como parâmetro no construtor as credenciais de acesso à conta da AWS. Saiba que existem vários construtores com várias formas de informar as credenciais.

A mais simples é através da criação de um usuário no serviço IAM (*Identity and Access Management*) da AWS. Ao fazer isso, a AWS associa ao mesmo um par de chaves. Lembre-se que é necessário dar as permissões de acesso à fila ao usuário criado no IAM.

Com esse par de chaves em mãos, informe na classe Java as constantes **AWS_ACCESS_KEY** e **AWS_SECRET_KEY**, que compõem o par.

Ajustando o endpoint

Após a criação do objeto **client**, precisamos informar um endpoint para o SQS através do método **setEndpoint()**. Isso porque as filas do serviço SQS são identificadas por um endpoint mais o nome da fila. O texto que identifica o endpoint deve ser especificado no código na constante **AWS_SQS_ENDPOINT**, definida na classe **AmazonSQSAsyncClient**.

Como migrar seu serviço JMS Java EE para a nuvem da Amazon

Listagem 15. Código da classe VotePublisherSQS, implementação de serviço de fila utilizando o SQS da AWS.

```
@Singleton
public class VotePublisherSQS implements VotePublisher {

    private static final String AWS_ACCESS_KEY = "AKIAJNOMRCLGYPGZMMNA";
    private static final String AWS_SECRET_KEY = "P2Tces3eGhKVZWD2O31Cs
    OYUDqCzzXt9mh/hmDbk";
    private static final String AWS_SQS_ENDPOINT =
    "https://sqs.us-east-1.amazonaws.com/326433034059";
    private static final String AWS_SQS_QUEUE = "votamais";

    @Inject
    VoteService voteService;

    AmazonSQSAsyncClient client;

    @PostConstruct
    public void initSQSClient() {
        client = new AmazonSQSAsyncClient(new BasicAWSCredentials
        (AWS_ACCESS_KEY, AWS_SECRET_KEY));
        client.setEndpoint(AWS_SQS_ENDPOINT);
    }

    @Override
    public void publishVote(String candidateName) {
        client.sendMessage(new SendMessageRequest(AWS_SQS_QUEUE,
        candidateName));
    }

    @Schedule(hour = "*", minute = "*", second = "*20")
    void startReceive() {
        client.receiveMessageAsync(AWS_SQS_QUEUE, new AsyncHandler
        <ReceiveMessageRequest, ReceiveMessageResult>() {

            @Override
            public void onSuccess(ReceiveMessageRequest req,
            ReceiveMessageResult res) {
                for (Message message : res.getMessages()) {
                    voteService.saveVote(message.getBody());
                    client.deleteMessageAsync(new DeleteMessageRequest
                    (AWS_SQS_QUEUE, message.getReceiptHandle()));
                }
            }

            @Override
            public void onError(Exception e) {
            }
        });
    }
}
```

Enviando uma mensagem

Repare a simplicidade de uso da API no método `publishVote()`. Nele, para o envio das mensagens utilizamos apenas o `sendMessage()`, o qual pode ser encontrado com várias assinaturas, mas no nosso exemplo optamos pela mais simples, que recebe um objeto do tipo `SendMessageRequest`, objeto esse que recebe como parâmetros em seu construtor o nome da fila criada anteriormente, e informada na constante `AWS_SQS_QUEUE`, mais a mensagem.

Recebendo mensagens

No lado responsável pelo recebimento das mensagens, o método `startReceive()` é executado constantemente pelo serviço de sche-

duling do EJB. Nesse momento ele faz uma chamada ao método `receiveMessageAsync()` e passa como parâmetros o nome da fila e um callback. Repare que, via annotation, informamos ao scheduler para executar esse método a cada 20 segundos, mesmo valor configurado para o atributo “Wait Time”, definido na criação da fila e que representa o tempo que o cliente da fila espera por uma mensagem durante uma conexão.

Em seguida, iteramos por uma lista de mensagens no método `onSuccess()`, pois a AWS envia várias mensagens a cada requisição. Por questões de performance, a AWS decidiu enviar em cada chamada à API a quantidade máxima de mensagens que estão disponíveis para leitura na fila, mensagens essas que são retornadas através da chamada a `getMessages()`. Sendo assim, devemos iterar por essas mensagens e tratar o recebimento de cada uma.

Adicionalmente, para cada mensagem processada devemos apagá-la da fila, através do método `deleteMessageAsync()`. Isso porque mensagens não apagadas voltam para a fila depois de um período em processamento e, assim, podem ser processadas novamente, o que não é desejado em nosso exemplo.

Realizadas as devidas alterações no código, a estrutura do projeto deve ficar semelhante à **Figura 10**.

Observe que as únicas alterações foram a remoção das classes que implementavam a fila local e a criação da classe `VotePublisherSQS`.

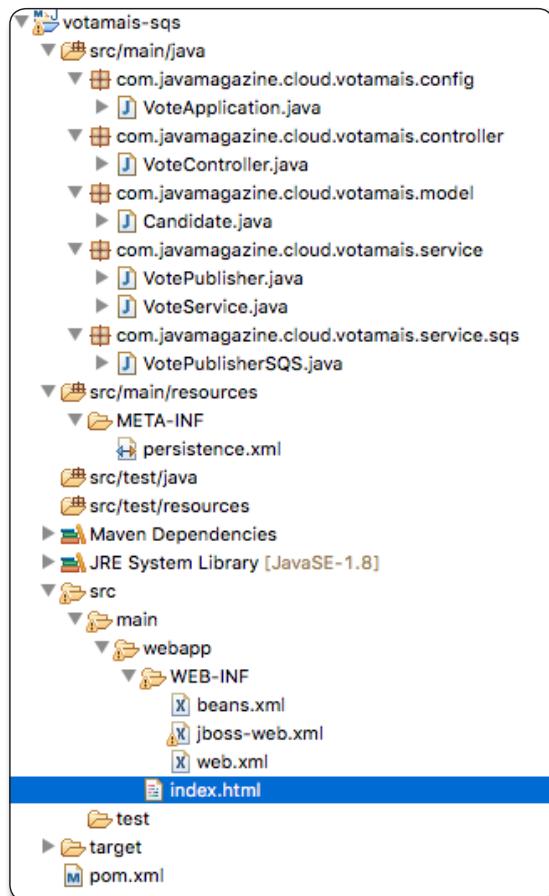


Figura 10. Estrutura do projeto VotaMais usando o serviço de fila da nuvem

Agora, para executar o projeto o leitor deve utilizar o mesmo comando do Maven citado anteriormente:

```
mvn org.wildfly.plugins:wildfly-maven-plugin:1.1.0.Alpha6:run
```

O comportamento da aplicação deve ser exatamente o mesmo, com a única diferença de que ela publicará as mensagens em um serviço de fila da AWS, ao invés de utilizar o serviço do container local.

Nota

Algumas exceções podem ser lançadas pelo SDK do SQS. As mais comuns são: QueueDoesNotExistException, que significa que o endpoint ou o nome da fila são inválidos, e AmazonServiceException (Caused By InvalidClientTokenId), que nos informa que as credenciais informadas são inválidas. Caso essa última aconteça, verifique no serviço IAM a validade das credenciais informadas e, se for o caso, gere um novo par de chaves para acesso ao serviço.

Aplicações mais aderentes à Cloud - Próximos passos

Em uma aplicação simples, como o VotaMais, os ganhos em consumir um único serviço de nuvem podem parecer ínfimos, mas devemos lembrar que esse é apenas um exemplo de como remover o peso da aplicação para torná-la mais aderente ao modelo cloud. No dia a dia lidamos com aplicações muito mais complexas e que utilizam vários serviços que podem ser delegados para a nuvem.

Usando essa abordagem, o impacto na redução do custo não é algo fácil de medir, pois não se trata apenas do custo do serviço propriamente dito. Entendemos que se deixarmos nossa aplicação mais leve, ela vai consumir menos recursos computacionais e, em consequência disso, escalar horizontalmente de maneira mais granular. Por outro lado, teremos os custos dos serviços que vamos utilizar. Ainda assim, como esses custos são tarifados por uso, tendem a ser menores do que se utilizássemos uma infraestrutura de serviços própria.

Neste momento é interessante uma reflexão. Quanto custa, por exemplo, manter um cluster de banco de dados com alta disponibilidade? Ou um serviço de fila com alta de disponibilidade? Enfim, todos esses pontos devem ser levados em consideração ao realizarmos a comparação entre os modelos de desenvolvimento, além, é claro, de ponderar sobre o ganho mais óbvio, que é fazer com que a aplicação lide apenas com o negócio que lhe compete e todo o resto seja delegado para a nuvem. A Figura 11 ilustra a redução do “peso” da aplicação quando utilizados os serviços da nuvem.

Entendemos que o modelo tradicional de construção e deploy de uma aplicação, mais focado no padrão de software monolítico, pode não ser eficiente para o novo paradigma de cloud computing. Diante disso, as novas aplicações tendem a ser criadas de forma mais aderente à cloud, até por força da necessidade de iniciar com baixo custo de infraestrutura e operação.

Dentro dessa linha, aplicações que utilizam a plataforma Java EE, solução completa e que oferece ao desenvolvedor diversos tipos de serviços em um único container, devem ser repensadas para um melhor funcionamento na nuvem.

Evolução da Aderência à Cloud

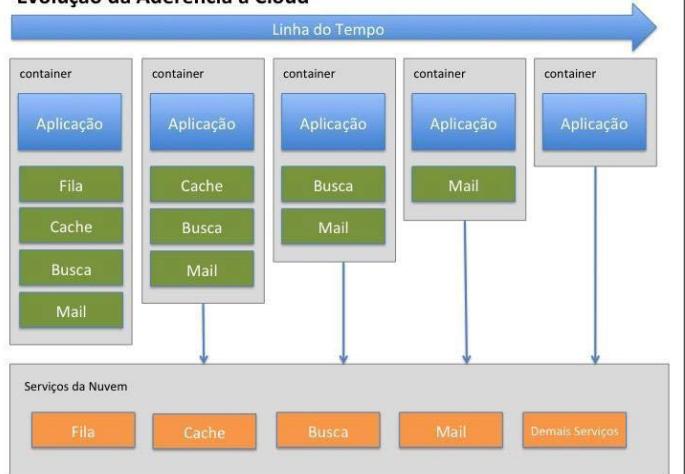


Figura 11. Modelo de evolução de uma aplicação em relação à aderência à cloud

Com base nisso, este artigo mostrou que é possível substituir os serviços do container por serviços da nuvem quando viável e vantajoso tecnologicamente e financeiramente.

Em caráter de exemplo, apresentamos a utilização de um serviço de fila de uma nuvem pública, mas se ampliarmos essa ideia, podemos substituir praticamente tudo o que não for regra de negócio por serviços da nuvem, deixando a aplicação mais leve, com baixo custo, mais fácil de manter e simples de escalar.

Autor



Daniel Stori

dstori@gmail.com

é Físico pelo Centro Universitário Fundação Santo André. Atuou na última década como Arquiteto Java e nos últimos dois anos trabalha como Arquiteto Cloud na TOTVS. Desenha nas horas vagas e mantém o site de humor geek <http://turnoff.us>.



Links:

Swarm, ferramenta para empacotamento de aplicações Java EE.

<http://wildfly-swarm.io/>

Apresentação do SQS.

<https://aws.amazon.com/pt/sqs/>

Exemplo de uso do SDK Java do SQS.

<https://github.com/aws/aws-sdk-java/tree/master/src/samples/AmazonSimpleQueueService>

WildFly, servidor Java EE utilizado neste artigo.

<http://wildfly.org/>

ActiveMQ Artemis, componente de mensageria do WildFly 10.

<http://activemq.apache.org/artemis/>

Documentação do SDK da AWS para Java.

<https://aws.amazon.com/pt/sdk-for-java/>

API do SQS.

<http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html>

Introdução a web services RESTful

Aprenda neste artigo quais são as tecnologias de base e como criar seus primeiros web services no padrão RESTful

As aplicações e sistemas desenvolvidos hoje em dia precisam atender a certas condições do cenário atual, tanto no caso de usuários individuais quanto para o caso de organizações, ou usuários corporativos. Essas condições dizem respeito às características de execução dos softwares de forma distribuída, rápida, segura, portável e confiável.

Considerando que as empresas precisam que seus colaboradores possam acessar informações para desempenhar as suas atividades de maneira rápida, segura e de qualquer lugar, seja de dentro ou fora de suas dependências, os sistemas precisam levar em consideração essas características e, portanto, estarem preparados para ambientes de uso e execução distribuídos e heterogêneos. Além de prover acesso a dados e serviços aos seus colaboradores, existe o caso de empresas que formam parcerias, necessitando realizar a troca de informações entre os seus sistemas de gerenciamento e os sistemas de seus parceiros. Uma terceira situação diz respeito a empresas ou entidades que fornecem serviços direto a seus clientes. Nesse caso, eles precisam ter acesso tanto às informações quanto aos serviços disponibilizados, seja para que eles possam realizar uma compra ou mesmo tirar dúvidas a respeito da utilização de um produto adquirido.

Esta breve exposição a respeito de alguns cenários de interação entre colaboradores, clientes, parceiros, empresas e os seus respectivos sistemas de software serve para caracterizar as necessidades de execução já elencadas e que são exigidas das aplicações. Uma das características mais importantes entre as que fazem parte dos requisitos dos sistemas atualmente é a sua disponibilidade de operação em um ambiente distribuído, que é caracterizado pelo fato de que uma solução não é mais pensada para ser executada em um único computador, mas sim em diversas máquinas simultaneamente, atendendo a diferentes tipos de consultas, atividades ou serviços, e,

Fique por dentro

Para utilizar bem uma determinada tecnologia é preciso conhecer os seus fundamentos, os conceitos nos quais ela é construída e, principalmente, como os diversos elementos que a compõem se relacionam. Neste artigo serão apresentados os fundamentos de como os web services RESTful funcionam e quais os elementos necessários para desenvolver e executar esses serviços, tudo isso de maneira objetiva e utilizando exemplos simples, de modo que seja fácil compreender o seu processo de criação e desenvolvimento.

na maioria dos casos, a partir de equipamentos completamente distintos uns dos outros, como notebooks, celulares e computadores desktop. Portanto, além do acesso distribuído, o sistema de software deve ser portável para diferentes plataformas, tanto de hardware quanto de software, pois além das diferenças já enumeradas, as máquinas podem estar utilizando sistemas operacionais diferentes, como Android, MacOS, Windows e Linux.

As condições de uso das aplicações, levando em consideração as situações apresentadas, requerem o uso de soluções tecnológicas que permitam aos desenvolvedores entregar as soluções de forma rápida, confiável e com baixo custo. Uma das tecnologias que permite isso são os web services, que podem ser definidos como aplicações cliente/servidor que se utilizam da comunicação através da internet, por meio do protocolo HTTP (*HyperText Transfer Protocol*), para prover serviços entre softwares que estejam executando em diferentes plataformas. Essas aplicações fazem uso do padrão XML (*eXtensible Markup Language*) para prover descritores que são interpretados automaticamente por outros sistemas, permitindo assim que diversos programas simples possam interagir uns com os outros para, em conjunto, fornecer soluções mais complexas e sofisticadas.

É importante entender que web service é um tipo de arquitetura de desenvolvimento cujo objetivo é a troca de informações entre duas entidades de software através da internet, utilizando os protocolos de comunicação disponíveis e tornando, assim, o envio

e recepção dos dados um processo já consolidado e conhecido. As entidades de software envolvidas nesse processo de comunicação são aplicações consideradas como servidor quando seu objetivo é prestar um serviço ou fornecer um mecanismo de acesso a um conjunto de dados. Já as aplicações consideradas como cliente têm como objetivo consumir um ou mais serviços disponibilizados pelos servidores.

Portanto, agora é possível compreender melhor o significado do termo web service, que pode ser entendido como um “serviço” correspondente a um componente de software acessível através de um endereço ou ponto de rede. Para isso, tanto o consumidor quanto o provedor do serviço utilizam mensagens com um formato próprio para enviar requisições e receber as respostas sem fazer qualquer suposição quanto às capacidades ou tipo de tecnologia empregada tanto numa ponta quanto na outra do processo de comunicação.

A característica que diz respeito a não fazer distinção nem exigir uma determinada tecnologia para qualquer um dos envolvidos no processo de troca de mensagens é que confere aos web services a grande vantagem e aceitação como base no desenvolvimento de sistemas a serem utilizados em ambientes distribuídos e heterogêneos, como os encontrados em sistemas das mais diferentes áreas hoje em dia.

Os web services podem ser implementados de muitas maneiras diferentes, cada uma com vantagens e desvantagens próprias da solução técnica adotada. Em sistemas desenvolvidos se utilizando a linguagem Java existem duas soluções, que na documentação da Oracle são denominadas como **Big Web Services** e **RESTful Web Services**. Tanto os web services Big quanto os RESTful fazem parte da API Java para XML (Java API for XML — JAX), que foi introduzida ao Java SE na versão 5.

A solução denominada Big Web Services é baseada na troca de mensagens codificadas em XML e utiliza o Protocolo de Acesso Simples a Objetos (*Simple Object Access Protocol* — SOAP), cujo padrão é mantido pela W3C. Esses serviços normalmente provêm um arquivo contendo a descrição padronizada das operações oferecidas, escrito seguindo a Linguagem de Descrição de Web Services (*Web Services Description Language* — WSDL), que, por sua vez, é um padrão definido em XML e utilizado para especificar interfaces de software que podem ser interpretadas pelos possíveis clientes. O propósito dessas declarações é que seja possível verificar se determinado serviço ou recurso é fornecido pelo web service em questão, além de descrever como deve ser realizado o acesso a esse serviço. Por utilizar o protocolo SOAP, os Big Web Services também são conhecidos como **Web Services SOAP**.

Já os web services RESTful (*Representational State Transfer*) são mais adequados para a utilização em cenários mais básicos, e também são melhor adaptados ao uso do protocolo HTTP do que os serviços SOAP. Os serviços RESTful são mais leves, o que significa que podem ser desenvolvidos com menor esforço, tornando-os mais fáceis de serem adotados como parte da implementação de um sistema.

O desenvolvimento de um web service RESTful é apropriado quando não é necessário manter informações de estado entre

chamadas do serviço. Além disso, se as respostas fornecidas podem ser armazenadas em um sistema de cache para melhorar o desempenho global do sistema e como não há uma descrição formal do funcionamento do serviço, tanto a aplicação que corresponde ao servidor do serviço quanto a que corresponde ao cliente ou o consumidor do serviço precisam entender e concordar com o contexto da troca de informações que ocorre quando são realizados acessos aos recursos disponíveis no web service.

Ao levar essas características em consideração, é possível entender uma distinção na aplicabilidade e uso dessas duas maneiras de implementar web services. Os Big Web Services são utilizados em aplicações corporativas que possuem requisitos tanto de qualidade quanto de segurança e performance mais avançados. Já os web services RESTful são mais indicados para a integração de sistemas através da internet, conferindo às soluções qualidades como escalabilidade, simplicidade e fraco acoplamento entre as partes.

No restante deste artigo serão fornecidos mais detalhes de como os web services RESTful são estruturados e implementados, além de serem abordados também tópicos a respeito das dependências de recursos externos, como bibliotecas e APIs necessárias durante o desenvolvimento e implantação dos serviços.

Padrão RESTful

O padrão REST determina como deve ser realizada a Transfência de Estado Representacional (*Representational State Transfer* — REST), ou seja, a representação que corresponde ao conjunto de valores que representa uma determinada entidade em um dado momento. Essa transmissão de estados se dá a partir da especificação de parâmetros, em alguns casos chamados de restrições, que podem ser aplicados a web services. Quando isso ocorre, o que se obtém são serviços escaláveis, de fácil modificação e manutenção, e que apresentam boa performance, tornando esses serviços adequados a serem utilizados através da internet. Outra característica do padrão REST é que a aplicação fica limitada a uma arquitetura cliente/servidor na qual um protocolo de comunicação que não mantenha o estado das transações entre uma solicitação e outra deve ser utilizado.

Nos serviços RESTful, tanto os dados quanto as funcionalidades são considerados recursos e ficam acessíveis aos clientes através da utilização de URIs (*Uniform Resource Identifiers*), que normalmente são endereços na web que identificam tanto o servidor no qual a aplicação está hospedada quanto a própria aplicação e qual dos recursos oferecidos pela mesma está sendo solicitado.

Os recursos disponíveis em um serviço RESTful podem ser acessados ou manipulados a partir de um conjunto de operações predefinido pelo padrão. As operações possibilitam criar (PUT), ler (GET), alterar (POST) e apagar (DELETE) recursos, e estão disponíveis a partir de mensagens utilizando o protocolo HTTP. Ao receber uma solicitação, ou mensagem HTTP, todas as informações necessárias para a realização da transação estão inclusas, não sendo necessário salvar informações para requisições de serviço futuras, o que resulta em aplicações simples e leves do ponto de vista do desenvolvedor. Já para os usuários que utilizam sistemas

baseados em web services RESTful, são serviços com um bom desempenho e transparentes, considerando sua utilização. Isso quer dizer que além de as solicitações aos serviços apresentarem boa responsividade, de acordo com a infraestrutura disponível, os usuários não fazem muita distinção entre um sistema executando localmente ou em um servidor de aplicação. Além dessas qualidades, a adoção do RESTful também tem impacto sobre outras métricas de qualidade de software, por exemplo: a facilidade de uso da aplicação e o tempo necessário pelo usuário para aprender a utilizá-la.

Outra característica importante dos recursos no padrão RESTful é que esses são dissociados de sua representação, ou seja, a maneira como os dados são manipulados na implementação do serviço não está vinculada ao formato da resposta a ser fornecida a uma solicitação, o que permite que os clientes peçam os dados em uma grande variedade de formatos, como HTML, XML, texto puro, PDF, JPG, entre outros.

Todas as características discutidas até aqui, que são parte do padrão RESTful, são adicionadas ao serviço utilizando-se anotações no código, de maneira que, a partir delas, o servidor de aplicações possa interpretar as URIs que recebe e direcionar a mensagem ao método correspondente do web service que o cliente deseja acessar. Essas anotações devem seguir as definições da API JAX-RS para determinar quais ações podem ser realizadas em cada recurso disponibilizado. A API JAX-RS especifica um conjunto de anotações de tempo de execução, portanto a ligação do servidor de aplicações com os recursos do web service acontece quando o pacote com as classes que o implementam é carregado no servidor. A hospedagem, ou publicação, de web services RESTful pode ser realizada em diferentes servidores de aplicação, como o GlassFish, Apache Tomcat e Oracle WebLogic.

Para o servidor de aplicações possa prover serviços RESTful, além da implementação do próprio serviço, é necessário também que esteja disponível a implementação da API JAX-RS, na qual estão disponíveis os serviços e as classes que definem as anotações e os seus correspondentes comportamentos, tornando possível a ligação entre as solicitações dos clientes e os recursos do web service publicado. A implementação de referência da API JAX-RS é o Jersey, um projeto de código aberto, portável, com qualidade, que está agora em sua versão 2.23 e que contempla as JSRs 311 e 339, que dizem respeito aos web services RESTful.

Passos para criar um web service RESTful

Para criar um web service RESTful é necessário implementar uma classe para servir de base para os recursos a serem acessados pelos clientes. Essa classe é um POJO (*Plain Old Java Object* — veja o **BOX 1**) que possui a anotação (veja o **BOX 2**) `@Path` em nível de classe ou pelo menos em um de seus métodos, ou ainda um método anotado com uma designação de método de pedido (*request method designator*), que pode ser `@GET`, `@PUT`, `@POST` ou `@DELETE`.

O padrão RESTful faz uso de anotações para facilitar o desenvolvimento dos web services, de modo que a declaração dos recursos e ações que poderão ser realizadas sejam especificadas utilizando

esses metadados nos membros da classe. Algumas das anotações mais utilizadas, definidas pela API JAX-RS, estão listadas na **Tabela 1**, juntamente com uma breve explicação a respeito do seu significado e de como são inseridas no código.

BOX 1. ORM

Quando se diz que uma classe Java é um POJO, significa que ela segue o conceito de "quanto mais simples melhor", ou seja, não possui uma regra específica para a sua codificação e também não utiliza os recursos de herança da Orientação a Objetos para estender uma classe já existente. Dessa maneira, o objeto resultante dessa classe deve possuir apenas os atributos e funcionalidades mínimos para atender aos requisitos especificados para a classe, sem a necessidade de atender a regras de nomenclatura de métodos e atributos, como no caso dos JavaBeans.

De forma simplificada, o termo POJO é utilizado para designar as classes Java que não seguem as regras de algum padrão ou framework, ou seja, as classes puras e simples. O nome foi criado por Martin Fowler, Rebecca Parsons e Josh MacKenzie em setembro de 2000 para facilitar o uso de objetos simples e regulares pelos programadores e analistas em seus projetos.

Estritamente falando, um POJO não deve estender uma classe, não deve implementar interfaces e não deve utilizar anotações em nível da classe, de atributos ou de métodos. Porém, em alguns casos, existem padrões de desenvolvimento que especificam o uso de anotações e declaram que as classes são POJOs, o que pode ser considerado como aceitável se a classe anotada foi um POJO antes da inclusão das anotações e volta a ser caso as anotações sejam retiradas. Nesse caso, a classe pode ser aceita como sendo um POJO, pois não possui outras características que a definam como sendo uma classe ou objeto especializado (SJO — Specialized Java Object).

BOX 2. Anotações

Uma anotação é uma informação a respeito de um dos componentes de uma classe ou da própria classe, ou seja, é um metadado (dados a respeito dos dados, ou seja, dados que descrevem ou caracterizam outros dados). As anotações são um recurso incluído na especificação da linguagem Java a partir de sua versão 5 com o objetivo de facilitar a documentação do código, rastrear dependências entre entidades do sistema e também possibilitar a execução de verificações em tempo de compilação.

A inclusão de uma anotação no código Java utiliza o símbolo `@` seguido do nome da anotação e, quando for o caso, de um conjunto de parâmetros no formato nome=valor separados por vírgula.

Um exemplo simples, que faz parte da API Java, é a anotação `@Override`, que informa ao compilador que o método anotado é uma sobrescrita de outro método. A partir dessa informação o compilador pode efetuar uma verificação adicional para determinar se a declaração do método está correta, o que não seria possível sem a indicação de que o mesmo está sobrepondo uma funcionalidade de uma de suas superclasses.

As anotações também são utilizadas por frameworks para adicionar características ao código que serão utilizadas em tempo de compilação ou de execução para produzirem, por exemplo, as informações de configuração necessárias ou para criar elementos externos, como tabelas em uma base de dados, entre muitas outras possibilidades.

Nas próximas seções serão apresentados e explicados um conjunto de exemplos com o objetivo de criar um web service RESTful simples. O primeiro exemplo envolve a criação do projeto, sua configuração e a codificação de um recurso que apenas responde a uma requisição HTTP GET com a frase "Olá mundo!", com o objetivo de entender a estrutura do projeto, seus componentes e as ligações entre os mesmos. Os exemplos seguintes acrescentarão novos recursos e funcionalidades ao POJO inicial para demonstrar diferentes maneiras de receber e enviar dados para o cliente.

Anotação	Descrição
@Path	Informa o valor de uma URI relativa, que determina o caminho no qual a classe ou método será hospedado no servidor. Além da especificação do caminho do recurso é possível também incluir a definição de variáveis nas URIs, como o nome do usuário manuseando o sistema ou os parâmetros a serem utilizados durante a execução da requisição enviada pelo cliente.
@GET	Essa anotação é um designador de método de pedido (Request Method Designator), que corresponde ao método HTTP de mesmo nome, e determina que o método da classe anotado processe e responda às solicitações GET recebidas. O resultado esperado é que o serviço retorne o valor correspondente ao recurso solicitado, que pode ser o conteúdo de um atributo, os registros de uma tabela em uma base de dados, o resultado de algum cálculo, entre outras possibilidades.
@POST	Essa anotação também é um designador de método de pedido e determina que o método da classe anotado processe e responda às solicitações POST recebidas. Como resultado de uma requisição POST é esperada a alteração do valor ou estado de algum recurso disponibilizado pelo serviço, que pode ser um registro em uma base de dados, o conteúdo de um arquivo, entre outros.
@PUT	Essa anotação também é um designador de método de pedido e determina que o método da classe anotado processe e responda às solicitações PUT recebidas. O uso de uma requisição PUT objetiva a criação de um recurso pelo serviço, que pode ser um arquivo, um registro em um arquivo ou em uma base de dados, entre tantas outras situações possíveis.
@DELETE	Esta anotação também é um designador de método de pedido e determina que o método da classe anotado processe e responda às solicitações DELETE recebidas. O resultado esperado para uma requisição DELETE é a eliminação ou destruição do recurso correspondente, ou seja, corresponde à eliminação do conteúdo de um registro em uma base de dados ou à remoção de um arquivo de dados, conforme a especificação utilizada na implementação do serviço.
@PathParam	Essa anotação designa um parâmetro que está na URI de requisição do serviço. O nome e posição dos parâmetros na URI são determinados pelo modelo definido pela anotação @Path.
@QueryParam	Essa anotação designa um parâmetro que está na parte de parâmetros de busca da URI enviada pelo cliente. Os parâmetros definidos dessa forma não necessitam estar especificados no modelo da URI definido pela anotação @Path.
@Consumes	É uma anotação para especificar o tipo de dado que um recurso pode consumir, ou seja, que o cliente pode enviar ao serviço. Os tipos de dados dessa anotação são especificados usando-se os tipos do padrão MIME.
@Produces	Essa é uma anotação para especificar o tipo de dados que um recurso pode produzir e enviar para o cliente em resposta a uma solicitação. Os tipos de dados dessa anotação também são especificados usando-se os tipos do padrão MIME.

Tabela 1. Descrição das anotações da API JAX-RS

O ambiente utilizado para o desenvolvimento dos exemplos a seguir envolve a IDE Eclipse Mars 2 (4.5.2), o servidor Apache Tomcat v7.0 e o framework Jersey versão 1.19.1. Ao selecionar a versão do Eclipse, verifique se a mesma é destinada à aplicação Java EE. Caso não seja, será necessário instalar os plugins e componentes correspondentes ao desenvolvimento de aplicações nessa plataforma para que seja possível trabalhar com o desenvolvimento de web services RESTful no mesmo. Ademais, durante o processo de implementação dos exemplos, o servidor Tomcat será iniciado a partir do Eclipse, portanto, ao instalá-lo, não informe que deseja iniciar o serviço automaticamente. Por último, os arquivos que compõem o framework Jersey devem ser vinculados a cada projeto individualmente, para garantir que o mesmo esteja disponível em tempo de execução quando o serviço for publicado no servidor de aplicações.

Implementando um web service RESTful básico

Para iniciar o desenvolvimento de um web service no Eclipse é necessária a criação de um Projeto Web Dinâmico (*Dynamic Web Project*). Feito isso, devem ser informados os dados a respeito do servidor no qual a aplicação será hospedada. Conforme já mencionado, nesse exemplo será utilizado o Apache Tomcat (vide **Figura 1**). Essas informações podem ser alteradas no futuro para verificar a compatibilidade com outros servidores, como o GlassFish. Os dados de configuração são utilizados pelo ambiente para publicar o web service, assim como para iniciar e parar o servidor propriamente dito.

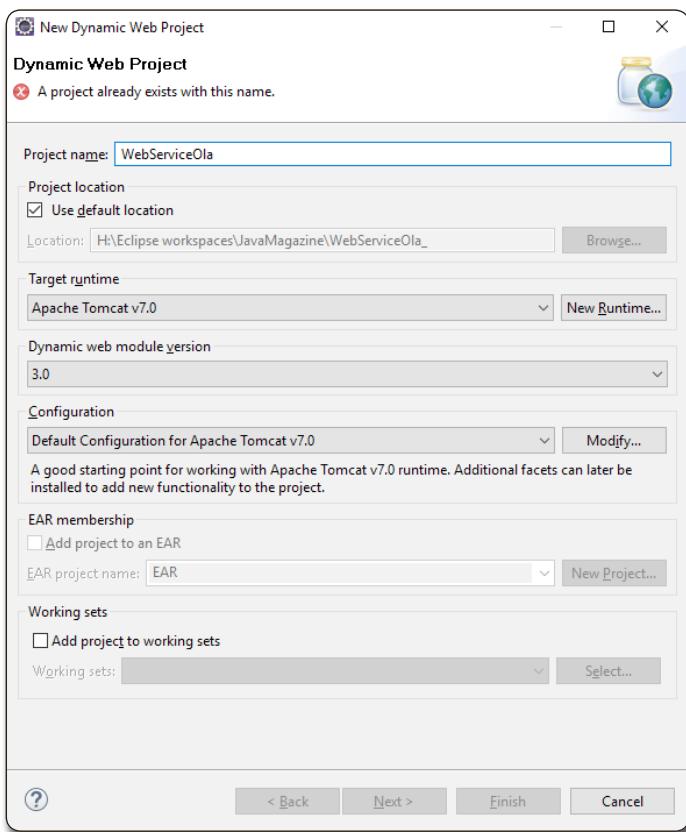


Figura 1. Configuração de um projeto Web Dinâmico para implementar um web service no Eclipse

Introdução a web services RESTful

Além das configurações a respeito do servidor, é necessária a criação de um arquivo XML com as informações a respeito do web service que será publicado, contendo o nome do web service, o nome do pacote no qual as classes Java estão codificadas, bem como o nome da aplicação, que será a base do caminho nas URIs enviadas nas solicitações dos clientes. Na última etapa de criação do projeto no Eclipse existe a opção correspondente à criação desse arquivo, (*Generate web.xml deployment descriptor*).

Uma vez criado o projeto, pode-se então incluir recursos Java, ou seja, o POJO no qual as funcionalidades do web service serão implementadas. Para isso, é necessário antes criar um pacote para abrigar as classes do web service, cujo nome o servidor Tomcat utiliza para buscar os recursos do serviço ao responder às requisições HTTP dos clientes. É preciso também adicionar ao projeto os arquivos JAR do framework Jersey para que tanto as anotações quanto as demais funcionalidades do mesmo possam ser utilizadas.

Na Figura 2 é exibida a estrutura lógica do projeto, na qual estão destacados o pacote a ser utilizado e os arquivos correspondentes à API Jersey. No caso do Jersey, os arquivos foram copiados para a pasta *WebContent/WEB-INF/lib*, onde devem ser colocados todos os arquivos correspondentes às bibliotecas e APIs externas das quais o serviço fará uso durante a sua execução no ambiente do servidor de aplicações.

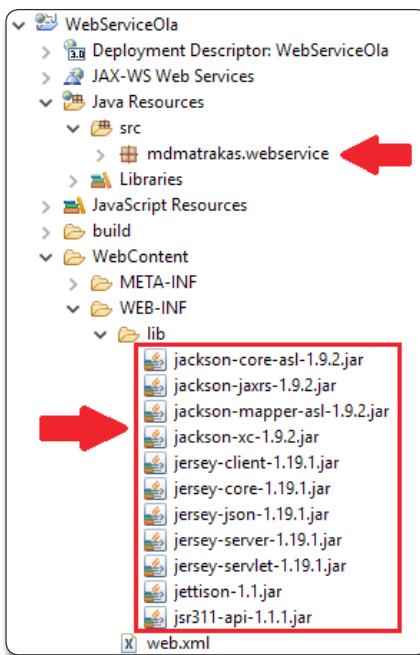


Figura 2. Estrutura lógica dos componentes de um Dynamic Web Project

Entre os serviços executados pelos componentes do framework Jersey está o tratamento de identificação das requisições HTTP recebidas pelo servidor, ou seja, a partir da URI e da identificação do tipo de requisição (GET, PUT, etc.) o framework identifica qual é a aplicação (web service) para a qual o cliente direcionou a requisição, e, entre os diferentes recursos disponibilizados pelo web service, qual é o que deve ser executado.

A identificação dos recursos do web service no código é feita com o uso das anotações apresentadas na Tabela 1, entre outras mais, que estão disponíveis no framework. Na requisição do cliente os recursos são identificados a partir dos nomes existentes no caminho da URI utilizada na requisição do serviço.

Para esse exemplo, a classe do web service deve conter apenas um recurso, que irá responder a requisições GET enviadas pelos clientes, devolvendo a frase “Olá Mundo!”. Na Listagem 1 é apresentado o código para o POJO que implementa esse recurso. Os pontos importantes para o web service nesse código são os seguintes: o nome do pacote no qual a classe é definida, pois esse nome será informado ao servidor de aplicação para que ele possa localizar corretamente os recursos; e a anotação `@Path("servico")` em nível de classe, que informa que todos os recursos implementados nessa classe serão acessados usando o nome fornecido como parâmetro à anotação, na URI. Isso quer dizer que quando o cliente desejar acessar algo que está implementado nessa classe, ele deve fazer isso utilizando uma URI que contenha o nome de caminho ao nível de classe seguido do nome do recurso, por exemplo: /servico/[nome do recurso].

O nome da classe não é relevante para o acesso ao web service, ao contrário dos valores passados às anotações para configurar os caminhos parciais de cada recurso, pois o servidor de aplicação não conhece os nomes das entidades declaradas no código Java, mas sim as informações armazenadas a partir dos parâmetros enviados às anotações do padrão RESTful.

Como base para todos os recursos disponíveis no web service, é utilizado o nome do projeto. Em seguida, considerando que em um projeto Web Dinâmico pode existir mais de um POJO, da mesma maneira que ocorre em projetos Java EE, eles são diferenciados um do outro na URI fornecida ao servidor de aplicação através do nome informado na anotação `@Path`, em nível de classe. Essa característica pode ser utilizada, por exemplo, para criar conjuntos de recursos, como o acesso a diferentes tabelas em uma base de dados, sendo o acesso a cada uma delas implementado por um POJO diferente, dado que todos fazem parte do mesmo projeto Java.

Ao desenvolver um web service RESTful, uma parte importante do processo é a elaboração das URIs de acesso aos recursos. Alguns pontos importantes a serem considerados neste processo são elencados a seguir:

- **Usar substantivos no plural:** A opção pelo plural facilita a identificação dos recursos nas URIs, evitando erros e simplificando a depuração da comunicação entre o cliente e o servidor.
- **Evitar o uso de espaços:** Utilizar o caractere sublinhado (_) ou o hífen (-) ao utilizar nomes longos compostos por mais de uma palavra. Apesar da sequência “%20” poder ser utilizada nas URIs para designar a utilização de espaços, essa substituição nem sempre é realizada da maneira correta, gerando erros por falha na especificação da URI;
- **Utilizar letras minúsculas:** Apesar das URIs serem sensíveis a maiúsculas e minúsculas, é uma boa prática manter todas as suas letras minúsculas, evitando erros de formação nas URIs, como no caso do uso do espaço;

- **Manter a retrocompatibilidade:** Como os web services são serviços públicos, uma vez que uma URI tenha sido divulgada ela deveria estar sempre disponível. Caso seja necessário atualizar uma URI, a URI em desuso deveria ser redirecionada ao novo endereço utilizando o código de retorno 300 do protocolo HTTP;
- **Utilizar os comandos HTTP:** Ao realizar operações nos recursos, utilizar as mensagens GET, PUT, POST e DELETE do protocolo HTTP. Não é uma boa prática utilizar nomes de operações nas URIs.

Listagem 1. Web service simples para atender a uma requisição HTTP GET.

```
// Pacote no qual a classe que implementa o web service está localizada
package mdmatrakas.webservice;

// Importação das classes da API JAX-RS em uso no web service
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

// Caminho relativo para o serviço representado por esta classe
@Path("servico")
public class WebServiceOla {
    // A anotação @GET indica que o método a seguir processa requisições HTTP GET
    @GET
    // A anotação @Path determina o caminho relativo do método
    // Como o método faz parte da classe WebServiceOla, que está hospedada no
    // caminho "/servico",
    // o caminho relativo para acessar este método é "/servico/oi"
    @Path("oi")
    // A anotação @Produces informa o tipo de dado produzido pelo método e que
    // será enviado ao
    // cliente que realizou a solicitação HTTP GET.
    // Neste caso o retorno será do tipo "Texto Puro"
    @Produces(MediaType.TEXT_PLAIN)
    // O método ola() implementa um recurso do web service
    public String ola() {
        // A String retornada por este método será enviada ao cliente que solicitou o
        // recurso deste web service
        // a partir da URI ".../servico/oi" com uma requisição HTTP GET
        return "Olá mundo!";
    }
}
```

A seguir são listados alguns pontos importantes a serem considerados durante a modelagem ou representação dos recursos em um web service RESTful:

- **Compreensibilidade:** Tanto o servidor quanto o cliente devem ser capazes de entender e utilizar a forma de representação do recurso;
- **Integralidade:** O formato do dado a ser utilizado deve ser capaz de representar um recurso completamente, mesmo nos casos em que um recurso possa ser composto por outros recursos. Dessa forma, a representação adotada deve conseguir representar tanto recursos com estruturas simples quanto complexas, ou seja, caracterizar corretamente casos em que os dados que representam o recurso possuem um conjunto simples de valores e também os casos nos quais os mesmos apresentam, por exemplo, dados

em mais de um nível, com dependências entre os valores de cada nível;

- **Acoplamento:** Um recurso pode conter ligações com outros recursos. Sendo assim, a sua representação deve poder lidar com tais situações.

Nos web services disponíveis atualmente, os recursos são representados utilizando formatação tanto XML quanto JSON. A descrição de como representar recursos utilizando esses formatos não faz parte do escopo deste artigo, apesar de alguns dos exemplos implementados utilizarem a representação de recursos no formato XML.

A última etapa no processo de criação de web services RESTful é a alteração do arquivo *web.xml* (ou a sua criação, dependendo das opções no momento de criar o projeto no Eclipse). Esse arquivo possui as configurações de publicação do web service, sendo o seu conteúdo mostrado na **Listagem 2**.

As informações mais importantes desse arquivo são as seguintes:

- Linha 11: Especificação da classe que implementa o **servlet** a ser utilizado na execução do web service. Essa classe faz parte do framework Jersey e é responsável por interpretar o conteúdo da solicitação HTTP e realizar a ligação com o método correspondente do web service;
- Linha 14: Especificação do nome do pacote que contém a(s) classe(s) com os recursos do web service;
- Linha 20: Especificação do caminho relativo para acessar os recursos desse web service. Nesse exemplo o caminho foi definido como “*/**”, não acrescentando nomes na URI do web service. Portanto, o único recurso disponível é acessado a partir do nome do serviço, isto é, o nome do projeto (*/WebServiceOla*), seguido do caminho da classe (*/servico/*) e do nome do recurso (*oi*), formando a URI */WebService/servico/oi*.

A classe da **Listagem 1** e o arquivo de configurações da **Listagem 2** caracterizam um web service simples, que já pode ser executado e testado. Entretanto, para que isso ocorra, é necessário que o mesmo seja publicado no servidor de aplicações, no caso o Apache Tomcat. As configurações fornecidas no momento da criação do projeto no Eclipse permitem que o mesmo realize os procedimentos exigidos pelo processo de publicação de forma automatizada, bastando então solicitar a execução da classe utilizando a opção executar no servidor (*Run on Server*).

Uma vez publicado o exemplo, e considerando que o servidor Apache está executando na mesma máquina na qual o web service está sendo implementado, ao fornecer a URI *http://localhost:8080/WebServiceOla/servico/oi* a um navegador web, o texto “Olá Mundo!” será exibido como resposta, como mostrado na **Figura 3**.

Melhorando a formatação da resposta

Na **Figura 3** o texto exibido pelo navegador não corresponde ao que foi produzido pela função que implementa o recurso *oi* do **WebServiceOla**, porque a codificação padrão dos navegadores é para texto em inglês, que não prevê o uso de caracteres acentuados.

Introdução a web services RESTful

Uma maneira de melhorar a apresentação da resposta no navegador é criar um recurso que retorne uma página HTML com as informações de codificação corretas para o idioma utilizado nos textos, no caso o português brasileiro. O código correspondente à mesma funcionalidade, porém com o texto fazendo parte da página HTML, está na **Listagem 3**.

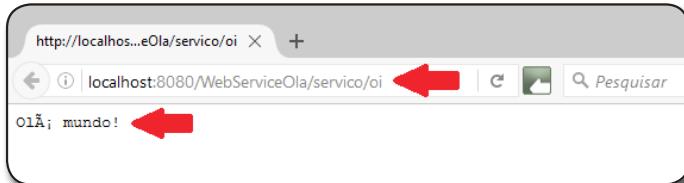


Figura 3. Resposta do recurso "oi" disponível no WebServiceOla

Listagem 2. Conteúdo do arquivo de configuração para o web service RESTful chamado WebServiceOla.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <web-app>
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xmlns="http://java.sun.com/xml/ns/javaee"
05   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
06   id="WebApp_ID"
07   version="3.0">
08   <display-name>WebServiceOla</display-name>
09   <servlet>
10     <servlet-name>RESTful Service Ola</servlet-name>
11     <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer
</servlet-class>
12     <init-param>
13       <param-name>com.sun.jersey.config.property.packages</param-name>
14       <param-value>mdmatrakas/webservice</param-value>
15     </init-param>
16     <load-on-startup>1</load-on-startup>
17   </servlet>
18   <servlet-mapping>
19     <servlet-name>RESTful Service Ola</servlet-name>
20     <url-pattern>/*</url-pattern>
21   </servlet-mapping>
22 </web-app>
```

Caso os dois recursos sejam mantidos, o caminho de acesso a cada um deve ser único nas suas respectivas anotações `@Path`. Além disso, é preciso modificar o tipo de dado produzido pelo recurso na anotação `@Produces`, indicando "TEXT_HTML". O resultado desse recurso é apresentado na **Figura 4**.

Na string retornada pelo recurso deve estar codificada a página HTML com todas as informações relevantes e necessárias para a correta exibição do texto pelo navegador. Assim, as metainformações de conteúdo de texto e conjunto de caracteres do tipo UTF-8, no cabeçalho da página, alteram a codificação utilizada pelo navegador ao exibir esse conteúdo. Outra vantagem de utilizar uma página HTML para exibir as respostas em um navegador é a possibilidade de alterar a formatação do texto para enfatizar as informações mais importantes, ou para formatar um relatório dinâmico. Esses recursos são particularmente interessantes ao produzir relatórios a partir de buscas em bases de dados cujo

acesso seja realizado por web services, ou para compor diferentes tipos de conteúdo em uma mesma resposta, por exemplo quando for necessário criar um texto com imagens.

Listagem 3. Produzindo uma página HTML no web service.

```
public class WebServiceOla
{
  // código repetido omitido...

  @GET
  @Path("oi-html")
  @Produces(MediaType.TEXT_HTML)
  public String olaHtml()
  {
    return "<html lang=\"pt-br\">
      <head><META http-equiv=\"Content-Type\" content=\"text/html; charset=UTF-8\">
      <title>Informações</title>
    </head>
    <body><h1>
      Olá mundo!
    </h1></body>
  </html>";
  }
}
```

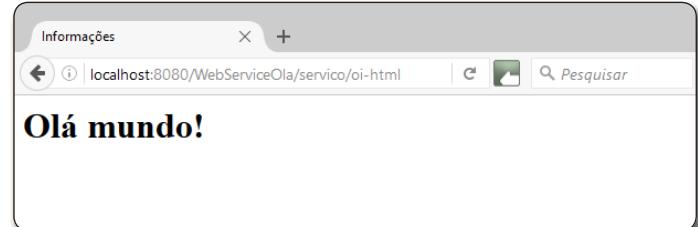


Figura 4. Resposta do WebServiceOla utilizando formatação HTML

Neste exemplo, as respostas são exibidas em um navegador, mas na maioria das vezes os resultados serão processados por uma aplicação cliente, de forma automatizada. Quando isso acontece, é mais importante a estrutura e codificação dos dados do que sua formatação para exibição. Para esses casos, a utilização de um arquivo XML como resposta tem mais vantagens, pois facilita a interpretação das informações, além de ser de fácil tratamento em qualquer plataforma, tornando o web service mais robusto.

Enviar dados do cliente ao web service

Nas seções anteriores foram apresentados exemplos nos quais o web service produz informações para o cliente. A questão agora é como o cliente pode enviar dados a um web service para serem utilizados como parâmetros em um recurso. Para exemplificar essa situação, a seguir será demonstrada uma situação na qual a requisição irá realizar uma operação de soma entre dois valores informados pelo cliente.

Na **Listagem 4** podemos ver o código com a implementação dessa nova funcionalidade no WebServiceOla. Os valores que serão somados devem ser enviados na URI de acesso após o nome do recurso, separados por "/". Isso é possível ao adicionar os campos a serem interpretados como parâmetros na string passada para

a anotação `@Path`, e depois utilizar esses campos em anotações `@PathParam` nas respectivas variáveis do método, conforme exposto nos comentários do código da **Listagem 4**.

Como esse método também implementa uma resposta a uma requisição HTTP GET, é possível testar a sua execução utilizando um navegador web normalmente, como apresentado na **Figura 5**.

Porém, como já mencionado, quando se deseja enviar dados para o servidor com o objetivo de atualizar um registro, por exemplo, o recomendado é utilizar uma mensagem HTTP PUT ou HTTP POST.



Figura 5. Resposta do recurso "soma"

Listagem 4. Enviando dados ao web service através da URI.

```
public class WebServiceOla
{
    // código repetido omitido...

    // Anotação @GET indica que o método a seguir processa requisições HTTP GET
    @GET
    // A anotação @Path determina o caminho relativo do método, adicionando
    // dois nomes de parâmetros ao caminho: "a" e "b".
    @Path("soma/{a}/{b}")
    // A anotação @Produces informa que o retorno será do tipo "Texto Puro".
    @Produces(MediaType.TEXT_PLAIN)
    // O método soma() implementa um recurso do web service.
    // As anotações @PathParam indicam que os parâmetros serão fornecidos a
    // partir da URI do
    // recurso.
    public String soma(@PathParam("a") int a, @PathParam("b") int b)
    {
        return String.valueOf(a + b);
    }
}
```

Na **Listagem 5** é exibido o código para o tratamento de uma mensagem PUT. Nesse exemplo é enviado apenas um parâmetro ao recurso do web service, também através da URI. A única diferença entre os métodos `soma()` (**Listagem 4**) e `criaRegistro()` (**Listagem 5**) é a utilização da anotação `@PUT` para indicar que o segundo será utilizado através da mensagem correspondente no protocolo HTTP. Como não é possível enviar uma mensagem HTTP PUT a partir dos navegadores web, é necessário utilizar uma ferramenta externa para testar o funcionamento do novo recurso do web service *Ola*.

Existem alguns aplicativos capazes de enviar diferentes mensagens HTTP, como PUT, POST DELETE, entre outras. Na Chrome Web Store, dois apps de uso bastante simples e que podem ser instalados como extensões para o navegador Google Chrome são

o *Advanced Rest Client* e o *POSTMAN*. Eles podem ser utilizados para testar as mensagens HTTP para os exemplos deste artigo.

Na **Figura 6** é apresentado o resultado ao enviar o comando HTTP PUT para a URI do recurso implementado na **Listagem 5**, utilizando o Advanced Rest Client. Estão destacados nessa imagem a URI do recurso solicitado, o tipo de mensagem HTTP e a resposta enviada pelo servidor para essa solicitação. Não faz parte do escopo deste artigo descrever o funcionamento dessas aplicações clientes, que podem ser utilizadas para testar as mensagens HTTP e depurar os web services e seus recursos, até porque seu uso é bastante simples e intuitivo.

Listagem 5. Enviando dados ao web service através da URI utilizando uma requisição PUT.

```
public class WebServiceOla
{
    // código repetido omitido...

    // A anotação @PUT indica que o método a seguir processa requisições HTTP PUT.
    @PUT
    // A anotação @Path determina o caminho relativo do método, adicionando
    // o parâmetro "registro" ao caminho.
    @Path("enviar/{registro}")
    @Produces(MediaType.TEXT_PLAIN)
    public String criaRegistro(@PathParam("registro") String registro)
    {
        // Aqui deve ser incluído o código para criar o registro solicitado.
        return "Registro: " + registro + " criado com sucesso!";
    }
}
```

De acordo com as descrições fornecidas na **Tabela 1**, as mensagens HTTP PUT são utilizadas para criar ou armazenar um novo valor no servidor. Já as mensagens do tipo POST são utilizadas para atualizar valores. Com relação a isso, aqui cabe uma nota: a operação de atualização também pode ser interpretada de maneira que seja possível realizar a criação de um novo valor, que pode ser, por exemplo, um novo registro em uma base de dados ou qualquer tipo de objeto que o web service venha a criar, isto é, uma requisição POST pode ser utilizada tanto para atualizar quanto para criar um novo registro/valor. Portanto, cabe ao desenvolvedor determinar se uma mensagem POST pode apenas atualizar um valor já existente, ou, no caso de ser solicitado a alterar um valor que ainda não existe, realizar a sua criação. Qualquer uma das duas soluções está correta e deve ser documentada na descrição dos recursos expostos pelo web service.

Na **Listagem 6** está a implementação do método que responde à mensagem POST simulando a atualização do registro solicitado. Novamente é interessante notar que não há muitas diferenças no código, apenas a alteração da anotação `@PUT` para `@POST`, inclusive mantendo o mesmo caminho na anotação `@Path`, o que não causa problemas, pois os caminhos, apesar de serem os mesmos, são utilizados para atender a requisições diferentes. Neste caso, o uso do mesmo caminho facilita a interação com o serviço, pois o significado de "enviar" dados é complementado pelo tipo de requisição HTTP, sendo uma para criar um novo registro (PUT) e outra para atualizar um registro já existente (POST).

Introdução a web services RESTful

Segundo a documentação do padrão JAX-RS e da linguagem Java, tanto métodos assinados com `@PUT` quanto com `@POST` podem ser utilizados para criar ou alterar valores. Como o tratamento de mensagens POST pode significar qualquer coisa, cabe à aplicação determinar o seu significado. Já uma mensagem PUT possui uma semântica bem definida para criar e atualizar um recurso.

Listagem 6. Enviando dados ao web service através da URI utilizando uma requisição POST.

```
public class WebServiceOla
{
    // código repetido omitido...

    // A anotação @POST indica que o método a seguir processa requisições HTTP POST
    @POST
    @Path("enviar/{registro}")
    @Produces(MediaType.TEXT_PLAIN)
    public String atualizaRegistro(@PathParam("registro") String registro)
    {
        // Aqui deve ser incluído o código para atualizar o registro solicitado.
        return "Registro: " + registro + " atualizado com sucesso!";
    }
}
```

Assim, a representação que um cliente envia em uma requisição PUT deve ser a mesma representação obtida utilizando uma requisição GET para o mesmo tipo de mídia (*MediaType*), o que não é obrigatório em uma requisição POST.

Devido a essas restrições, uma requisição PUT não permite que um recurso seja alterado parcialmente, ou seja, não é possível enviar dados parciais para um determinado elemento. O cliente deve enviar o objeto completo, com a mesma representação da requisição GET.

Enviando parâmetros como valores de busca

Quando um cliente envia ao servidor uma requisição que necessita de parâmetros, esses, além de serem posicionados na URI como variáveis, também podem ser especificados como parâmetros de busca. Ao definir uma URI, os parâmetros de busca são posicionados ao seu final após um caractere de interrogação (?), utilizando-se pares chave-valor para cada parâmetro. Com esse tipo de construção não é necessário especificar os nomes dos parâmetros na anotação `@Path` da maneira como foi feito nos exemplos das **Listagens 4, 5 e 6**, mas o cliente deve especificar o nome de cada um dos parâmetros de busca utilizando exatamente o mesmo nome que o web service está esperando.

Para receber esse tipo de parâmetro é utilizada a anotação `@QueryParam`, que informa que os valores estão codificados como parâmetros de busca na URI e não como parte do caminho. Para ilustrar o uso de um parâmetro de busca, o exemplo da requisição PUT foi reescrito na **Listagem 7**.

Para enviar mais de um parâmetro de busca, os mesmos devem ser separados uns dos outros utilizando-se o caractere “&”. Portanto, se os valores para “idade” e “nome” fossem esperados pelo servidor, a URI de busca seria: *bservice/recurso?idade=15&nome=Fulano*. Nesse caso a ordem dos valores não é importante, pois a busca é feita pelo seu nome. Ademais, para cada parâmetro de busca é possível especificar um valor padrão com a anotação `@DefaultValue("valor")`. Esse será o valor atribuído à variável correspondente caso o parâmetro não esteja especificado na string enviada pelo cliente.

Outro ponto importante a ser levado em consideração ao trabalhar com parâmetros de busca, da mesma forma que valores codificados no caminho da URI, é o uso de caracteres especiais nos valores. Assim, se, por exemplo, o valor for uma frase ou o nome completo de uma

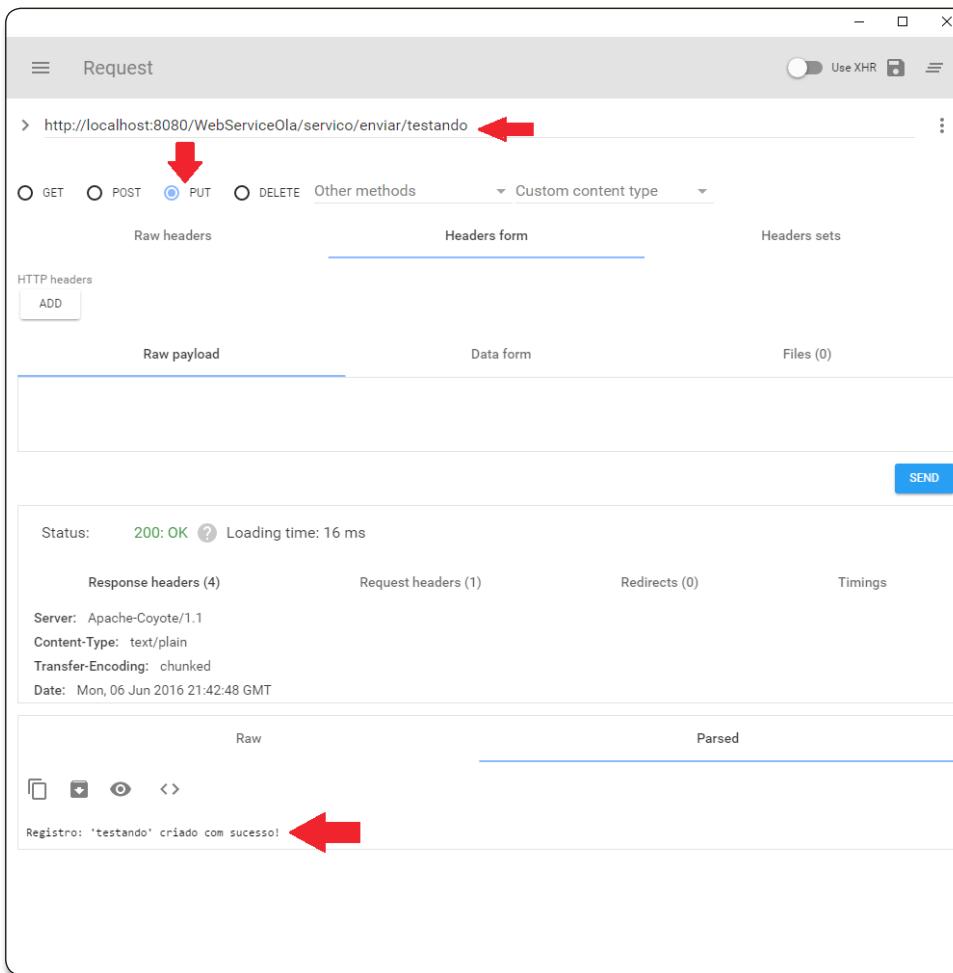


Figura 6. Resposta a uma requisição PUT

pessoa, como “Fulano de Tal”, essa sequência de caracteres não pode ser enviada diretamente, nem como parte do caminho da URI nem como um parâmetro de busca, pois possui dois espaços, e eles não podem fazer parte da URI, devendo ser substituídos por “%20”. Outros caracteres especiais também devem ser substituídos por seus respectivos códigos de formatação usando “%cod”.

Listagem 7. Enviando dados ao web service através de um parâmetro de busca.

```
public class WebServiceOla
{
// código repetido omitido...

    @PUT
    // A anotação @Path determina o caminho relativo do método
    @Path("enviar")
    @Produces(MediaType.TEXT_PLAIN)
    // A anotação @QueryParam informa que o parâmetro esperado por este
    //método será recebido
    // como um parâmetro de busca na URI: "servico/enviar?registro=valor"
    public String criaRegistroQuery(@QueryParam("registro") String registro)
    {
        // Aqui deve ser incluído o código para criar o registro solicitado.
        return "Registro: " + registro + "\\' criado com sucesso!";
    }
}
```

Outra característica importante dos valores enviados ao web service, tanto com **@PathParam** quanto com **@QueryParam**, é que essas anotações podem ser utilizadas apenas com os seguintes tipos de dados:

- Todos os tipos primitivos, exceto **char**;
- Todas as classes *wrapper* para tipos primitivos, exceto **Character**;
- Qualquer classe que tenha um construtor que aceite uma única **String** como argumento;
- Qualquer classe que tenha um método estático chamado **valueOf** que receba como argumento uma única **String** para realizar a sua inicialização;
- **List<T>**, **Set<T>** ou **SortedSet<T>**, nos quais T corresponda aos critérios já listados nos marcadores anteriores.

Caso a anotação **@DefaultValue()** não tenha sido especificada para uma variável e o parâmetro não estiver presente na requisição, o valor atribuído será o valor padrão para tipos primitivos, uma coleção vazia quando for o caso ou *nulo* para outros tipos de objetos.

Por fim, além dos parâmetros de caminho e de busca, também estão disponíveis os seguintes tipos de parâmetros: Form, Cookie, Header e Matrix, que não serão tratados neste artigo. Todos eles obedecem às mesmas regras citadas.

Enviando parâmetros no corpo de uma requisição

O leitor mais atento deve ter notado que os exemplos apresentados até o momento recebem seus parâmetros através da URI de acesso ao web service. No entanto, é válido ressaltar que essa não é a única maneira de enviar dados utilizando o protocolo HTTP.

Para entender melhor como esse processo de envio de dados acontece, é preciso conhecer um pouco do funcionamento desse protocolo, no qual um cliente envia mensagens no formato de uma requisição HTTP (**Figura 7**) e o servidor responde com outra no formato de uma resposta HTTP (**Figura 8**). Em uma requisição HTTP, as mensagens possuem as seguintes partes:

1. O **Verbo** identifica qual é o tipo de requisição que está sendo enviada, podendo ser GET, PUT, etc.;
2. A **URI (Uniform Resource Identifier)** especifica o endereço do recurso que se deseja acessar no servidor;
3. **Versão HTTP** indica qual a versão do protocolo;
4. O **cabeçalho da requisição** especifica os metadados da requisição no formato de pares chave-valor, por exemplo: tipo de cliente ou navegador, formatos suportados pelo cliente, formato do corpo da mensagem, configurações de cache, etc.
5. O **corpo da requisição** é o conteúdo da mensagem ou a representação de um recurso.

Já as mensagens de resposta HTTP possuem as seguintes partes:

1. O **código de resposta** indica o estado do servidor para o recurso solicitado, por exemplo: 404 significa recurso não encontrado e 200 significa sucesso;
2. **Versão HTTP** indica qual a versão do protocolo;
3. O **cabeçalho da resposta** especifica os metadados a respeito da mensagem de resposta no formato de pares chave-valor, por exemplo: tamanho do conteúdo, tipo do conteúdo, data, tipo do servidor, etc.;
4. O **corpo da requisição** é o conteúdo da mensagem de resposta ou a representação do recurso solicitado.

Verbo	URI	Versão HTTP
	Cabeçalho da requisição	
	Corpo da requisição	

Figura 7. Formato de uma requisição HTTP

Código de resposta	Versão HTTP
Cabeçalho da resposta	
Corpo da resposta	

Figura 8. Formato de uma resposta HTTP

Note que todos os exemplos apresentados até o momento não utilizaram o campo da mensagem HTTP, que corresponde ao conteúdo, o **corpo da requisição**. Utilizando esse espaço é possível informar que os dados necessários para completar uma requisição do web service sejam transferidos no corpo da mensagem de requisição do serviço e não na URI.

Introdução a web services RESTful

Como exemplo, as requisições tratadas tanto na **Listagem 5** quanto na **Listagem 6** podem ser modificadas para receber os seus parâmetros dessa forma. A maneira de especificar isso no código é utilizando a anotação `@Consumes()`, passando como argumento o tipo de dado esperado. Como nos exemplos anteriores são esperadas sequências de caracteres sem nenhuma formatação específica, o tipo de dado a ser utilizado é o `MediaType.TEXT_PLAIN`.

O código correspondente às requisições PUT e POST modificadas está na **Listagem 8**, na qual pode-se perceber que os caminhos para os recursos continuam os mesmos, mas sem a especificação dos parâmetros na URI como nos exemplos anteriores, modificando-se apenas os nomes dos métodos Java para que não ocorra um erro de compilação por existirem dois métodos com a mesma assinatura. Nesses casos as requisições são diferenciadas umas das outras pelo número de elementos na URI, ou seja, se existir algum valor especificado após o caminho `/enviar` na URI, o método que recebe os parâmetros pela URI será invocado, caso contrário, o método que recebe o parâmetro pelo corpo da mensagem será chamado.

Listagem 8. Enviando dados ao web service através do corpo da requisição HTTP.

```
public class WebServiceOla
{
    // código repetido omitido...

    @PUT
    @Path("/enviar")
    @Produces(MediaType.TEXT_PLAIN)
    // A anotação @Consumes informa o tipo de dado consumido/recebido pelo
    //método
    @Consumes(MediaType.TEXT_PLAIN)
    public String criaRegistroCorpo(String registro)
    {
        // Aqui deve ser incluído o código para criar o registro solicitado.
        return "Registro: " + registro + "\'criado com sucesso!'";
    }

    @POST
    @Path("enviar")
    @Produces(MediaType.TEXT_PLAIN)
    // A anotação @Consumes informa o tipo de dado consumido/recebido pelo
    //método
    @Consumes(MediaType.TEXT_PLAIN)
    public String atualizaRegistroCorpo(String registro)
    {
        // Aqui deve ser incluído o código para atualizar o registro solicitado.
        return "Registro: " + registro + "\'atualizado com sucesso!'";
    }
}
```

No exemplo da **Listagem 8**, o tipo de dado enviado ao servidor foi um conjunto de caracteres sem nenhuma formatação, o que, para esse caso simples ou quando o dado a ser enviado é formado por uma única sequência de caracteres, funciona bem. Em aplicações reais, no entanto, é necessário utilizar uma estrutura padronizada de caracteres para representar os dados.

Para esses casos, como já mencionado, um dos padrões mais utilizados é o XML, que permite que uma aplicação defina a estrutura dos dados que devem ser transmitidos. Além disso, existem APIs

para tratar arquivos XML, o que facilita muito a adoção dessa opção. O mesmo pode ser dito do JSON, que também pode ser empregado para formatar os dados no corpo das mensagens HTTP.

De acordo com a arquitetura REST, os web services não devem manter informações a respeito do estado do cliente no servidor, ou seja, é responsabilidade da aplicação cliente enviar o seu contexto ao servidor, que pode ou não o armazenar para processar futuras requisições do cliente.

Nos exemplos estudados, os métodos do web service não armazenam qualquer informação a respeito do cliente que realiza a requisição. Sendo assim, se a URI `localhost:8080/servico/enviar/teste` for enviada utilizando um navegador web, um cliente Java ou uma aplicação como a Advanced Rest Client, a resposta será exatamente a mesma, sem nenhuma distinção nos dados que serão armazenados no arquivo ou banco de dados correspondente à aplicação, cujos recursos estão disponíveis aos clientes através do web service. A isso se dá o nome de “sem monitoração de estado” ou *stateless*, em inglês.

Os Big Web Services, ou web service SOAP, diferentemente dos web services RESTful, realizam o monitoramento de estado das solicitações, ou seja, mantêm, entre outras informações, um histórico das requisições já enviadas por cada cliente.

Levando isso em consideração, pode-se elencar algumas vantagens para web services sem monitoramento de estado:

- Cada requisição pode ser tratada independentemente;
- O projeto da aplicação é simplificado já que não é necessário manter as interações anteriores do cliente;
- O protocolo HTTP em si é sem monitoramento de estado. Portanto, web services RESTful trabalham perfeitamente com o HTTP.

Por outro lado, também existem desvantagens ao utilizar web services sem monitoramento de estado:

- Cada requisição necessita de informações extras, que precisam ser interpretadas para determinar o estado do cliente, caso as interações do cliente devam ser levadas em consideração;
- Informações a respeito da seção de um usuário devem ser mantidas no cliente, e a cada nova requisição devem ser enviadas ao web service para validação do nível de acesso correspondente, quando for o caso.

O uso de web services como parte de aplicações, sejam elas corporativas ou não, já se tornou um padrão no desenvolvimento de software. A necessidade de conhecer os conceitos e características das tecnologias empregadas nesse tipo de sistema, portanto, é uma necessidade para os desenvolvedores.

Os exemplos abordados durante o artigo apresentam algumas das características fundamentais para a criação de web services RESTful, no entanto esses exemplos não estão vinculados a uma aplicação e, portanto, não definem um serviço específico, como manter um cadastro de pessoas ou permitir o acesso a um conjunto de métodos específicos para processar um certo tipo de dado. Cabe a cada analista ou desenvolvedor, a partir dos conhecimentos aqui abordados, identificar situações nas quais os sistemas em que

trabalham possam se beneficiar do uso dessa tecnologia, como: a geração de páginas de conteúdo dinâmico a partir de requisições de serviço informando alguns parâmetros de busca.

É importante ressaltar que uma das grandes vantagens do uso de web services é fornecer uma forma de acesso segura a recursos, disponíveis em uma máquina ou em uma rede, que não devem estar expostos na internet, como no caso de servidores de bancos de dados. Nesses casos é possível escrever um conjunto de web services que publique um conjunto de ferramentas de acesso à base de dados para possibilitar que usuários consigam realizar consultas ou até mesmo algumas alterações, sem correr o risco de acessos indevidos. Isso ocorre porque todas as interações com a base de dados serão feitas através do web service, que disponibiliza uma interface padronizada sem a possibilidade de que algum tipo de operação não autorizada seja realizada, visto que o acesso à base de dados, ou a qualquer recurso a ser protegido, estará nele codificado, evitando assim que requisições com erros ou com objetivos escusos causem danos aos dados.

Para exemplificar as questões de segurança mencionadas, pode-se pensar em uma indústria cujos equipamentos podem ser controlados e configurados remotamente, caso os canais de controle estejam acessíveis diretamente pela internet. Neste caso, seria possível, através de tentativa e erro, descobrir as senhas e alterar as configurações, causando desde problemas na produção a defeitos nos equipamentos. Ao utilizar um conjunto de web services, para realizar o acesso a esses equipamentos, apenas as requisições implementadas poderão ser utilizadas, além de ser viável identificar padrões de ataque nas requisições, evitando acessos indevidos.

Outra situação na qual o uso desse tipo de sistema pode trazer vantagens é na disponibilização de recursos computacionais exclusivos de determinada máquina para um público mais abrangente. Por exemplo, no caso de uma instituição que possua um supercomputador, alguns de seus sistemas podem ser disponibilizados para acesso externo através de web services, possibilitando a usuários que não se encontrem próximos à máquina acesso ao poder computacional dessa para resolver problemas.

Autor



Miguel Diogenes Matrakas

mdmatrakas@yahoo.com.br

É Mestre em Informática pela PUC-PR e doutorando em Métodos Numéricos em Engenharia, pela UFPR (Universidade Federal do Paraná). Trabalha como professor de Java há quatro anos nas Faculdades Anglo-Americanas de Foz do Iguaçu.



Links:

Especificação do padrão SOAP - Simple Object Access Protocol.

<https://www.w3.org/TR/soap/>

Descrição da API Java para XML - JAX.

<https://jax-ws.java.net/>

Tutorial do Java Enterprise Edition versão 6, com capítulos dedicados aos Big Web Services e Web Services RESTful.

<http://docs.oracle.com/javaee/6/tutorial/doc/index.html>

Página do projeto do Framework Jersey.

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

Página do servidor Apache Tomcat.

<https://tomcat.apache.org/>

Link para a instalação do plugin Advanced Rest Client no navegador Chrome.

https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffhphgfcellkdfbfbjeloo?hl=en-US&utm_source=ARC

Link para a instalação do plugin Postman no navegador Chrome.

<https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbnccccdomop/related>

JTA e a demarcação de transações globais em EJBs

Veja neste artigo qual o propósito das especificações JTA e EJB e como elas viabilizam a demarcação de transações globais em EJBs

Na computação, uma transação atômica representa um conjunto de uma ou mais operações efetuadas como se fossem uma unidade única e indivisível de execução. Assim, operações executadas em uma transação são todas efetivadas ou descartadas por inteiro, de forma atômica. Se ocorrer falha em pelo menos uma operação da transação, todas as operações, mesmo as que foram executadas com sucesso, são descartadas, permitindo que o estado dos dados permaneça o mesmo de antes da criação da transação.

Muitos tipos de sistemas computacionais precisam utilizar transações para que funcionem corretamente. Um caixa eletrônico, por exemplo, é um tipo de dispositivo que utiliza transações. Imagine uma transferência bancária entre duas contas, efetuada utilizando um caixa eletrônico. Uma pessoa, ao solicitar uma transferência de uma conta A para uma conta B, induz o sistema do caixa eletrônico a criar uma transação que conterá duas operações: a operação de debitar o saldo de uma conta e a operação para creditar esse saldo em outra conta.

Digamos que, hipoteticamente, a operação de debitar saldo da conta A seja efetuada com sucesso, mas a operação de creditar saldo na conta B seja negada, por restrições quaisquer, por exemplo conta inoperante. Nesse cenário, mesmo que a primeira operação tenha sido concluída com sucesso, como ela faz parte de uma transação atômica, toda a transação é abortada, e a conta corrente original A não terá seu saldo alterado. Como sabemos, esse comportamento é primordial para que o sistema funcione corretamente.

Fique por dentro

Este artigo é útil por apresentar o que é uma transação atômica, conceito existente na maioria dos sistemas enterprise implementados utilizando a plataforma Java EE. Para isso, serão analisadas as principais diferenças entre transações locais, globais e as formas de demarcação de cada tipo, além de entendermos como o servidor de aplicação se comporta ao identificar essas demarcações nos EJBs de uma aplicação.

Por esse motivo, uma transação é um conceito essencial a ser utilizado em todos os sistemas que desejam integridade de dados. Com base nisso, neste artigo iremos abordar a diferença entre transações locais, globais, a demarcação de transações globais em EJBs e como o servidor de aplicação se comporta ao identificar tais demarcações, comportamento esse descrito nas especificações JTA e EJB.

Recursos transacionais e transações locais

São considerados recursos transacionais aqueles que viabilizam o suporte a transações, isto é, aqueles que possibilitam que transações sejam criadas e finalizadas corretamente. Os principais recursos transacionais empregados em aplicações corporativas — de conhecimento inerente a qualquer profissional da área de desenvolvimento de software — são os bancos de dados relacionais, utilizados para armazenar os dados da aplicação. Podemos citar como exemplos de bancos relacionais o MySQL, o SQL Server, o PostgreSQL, entre outros.

Nesses bancos de dados, podemos iniciar uma transação, executar uma série de inserções, atualizações e exclusões de dados

e efetivar ou cancelar todas as operações contidas dentro dessa transação de uma só vez, de forma atômica.

Para que seja possível nos comunicarmos com esse tipo de recurso transacional na aplicação, utilizamos drivers JDBC. A especificação JDBC define as interfaces e comportamentos que todo driver JDBC deve implementar. Utilizando essa API, a aplicação é capaz de se conectar ao banco, iniciar uma transação, executar as operações necessárias, como inserir dados, por exemplo, e finalmente concluir a transação com sucesso (efetuando commit) ou com falha (efetuando rollback).

Se a aplicação utiliza apenas um banco de dados, mesmo que esse banco esteja em outro servidor, a transação criada será do tipo local. Isso porque toda transação que envolve apenas um recurso transacional — um único banco de dados relacional, por exemplo — é considerada local. Podemos verificar na **Listagem 1** um exemplo de aplicação que cria uma conexão com um único recurso transacional (MySQL) e efetua a atualização de dados.

Listagem 1. Aplicação standalone – atualização de dados no banco de dados MySQL.

```
public class Application {  
  
    public static void main(String[] args) throws Exception {  
        Class.forName("com.mysql.jdbc.Driver");  
        Connection conexao = DriverManager.getConnection  
            ("jdbc:mysql://localhost/test","usuarioTeste","12345");  
        PreparedStatement pst = conexao.prepareStatement  
            ("UPDATE tb_usuario SET ativo=1");  
        int totalLinhasAfetadas = pst.executeUpdate();  
        conexao.close();  
    }  
}
```

Nesse exemplo, quando a aplicação cria uma transação, executa as operações necessárias e então finaliza a transação. Todas essas operações são enviadas ao MySQL por meio do driver JDBC. O MySQL, por sua vez, torna-se responsável por gerenciar esses comandos. Perceba, nessa listagem, que não há vários recursos transacionais diferentes envolvidos, apenas um único banco MySQL. Desse modo, dizemos que a transação é local para esse recurso.

Ainda nessa listagem, observe que a transação é criada de forma implícita assim que efetuamos a atualização dos dados, portanto não foi necessária a demarcação explícita do início e fim da transação pois a especificação JDBC declara que toda nova conexão criada por um driver JDBC será configurada, por padrão, para efetuar o commit automaticamente dos comandos enviados ao banco de dados. No entanto, esse comportamento pode ser alterado, modificando a configuração de **auto-commit** da conexão para **false**. Isso indica que o driver não deverá mais efetuar commit automaticamente, apenas quando explicitamente solicitado. Nesses casos, a demarcação de início e fim da transação deve ser efetuada de forma explícita no código. Podemos verificar esse comportamento na **Listagem 2**.

Listagem 2. Demarcando a transação local controlada de forma explícita.

```
public class Application {  
    public static void main(String[] args) throws Exception {  
        Class.forName("com.mysql.jdbc.Driver");  
  
        Connection conexao = DriverManager.getConnection  
            ("jdbc:mysql://localhost/test","usuarioTeste","12345");  
  
        try{  
            conexao.setAutoCommit(false);  
            PreparedStatement pst = conexao.prepareStatement  
                ("UPDATE tb_usuario SET ativo=1");  
            int totalLinhasAfetadas = pst.executeUpdate();  
            conexao.commit();  
        }catch(Exception e){  
            conexao.rollback();  
            throw e;  
        }finally{  
            if(conexao != null){  
                conexao.close();  
            }  
        }  
    }  
}
```

Nesse código, como o commit automático foi desativado pela declaração **conexao.setAutoCommit(false)**, é necessário demarcar o fim da transação efetuando commit ou rollback, ou seja, é necessário chamar os métodos **conexao.commit()** e **conexao.rollback()**.

O que é uma transação global?

Foi discutido que transação local é aquela que possui apenas um recurso transacional. Agora, imagine o seguinte cenário: um usuário acessa um sistema de ordens de serviço (OS) e, ao cadastrar uma nova OS, algumas informações são gravadas no Oracle, a ser



acessado por uma equipe, e outras no SQL Server, a ser acessado por outra equipe. Com isso, ambas as equipes podem promover as atividades necessárias para concluir a OS.

Entretanto, imagine que, no momento de salvar os dados no SQL Server, uma falha ocorra. Como os dados não foram devidamente persistidos no SQL Server, as informações do Oracle também não poderiam ser persistidas, pois levaria parte da equipe a iniciar um trabalho não conhecido pela outra, violando os procedimentais operacionais da empresa.

Para viabilizar o comportamento correto, obviamente precisamos de uma transação, de forma a criar uma unidade indivisível de execução, ou seja, ou tudo falha ou tudo é completado com sucesso. Mas o principal detalhe a se notar aqui é que, para o cenário específico apresentado, há mais de um recurso transacional na transação, nesse caso mais de um banco de dados relacional (Oracle e SQL Server). Esse é um cenário real no qual necessitamos de uma transação global (ou distribuída, como também é chamada), isto é, aquela transação que envolve mais de um recurso transacional.

Esse tipo de transação é suportado por um servidor de aplicação Java EE, pois ele permite que mais de um recurso transacional seja associado e coordenado pela mesma transação.

Como demarcar transações globais em EJBs

De acordo com a especificação EJB, antes de demarcarmos o início e o fim de uma transação global, devemos determinar o tipo de gerenciamento de transação dos EJBs da aplicação. Essa configuração é feita por meio de uma importante anotação, a `@javax.ejb.TransactionManagement`, que deve ser declarada na classe e pode assumir dois valores: **BEAN** e **CONTAINER**. Se for

definida como **BEAN**, indicará ao servidor de aplicação que ele não deve se intrometer no gerenciamento das transações globais, pois o próprio bean (o EJB, neste caso) irá efetuar o controle dessas transações, ou seja, demarcar o início e o fim das mesmas. Veja um exemplo na **Listagem 3**.

Por outro lado, se o valor da anotação `@TransactionManagement` for **CONTAINER**, significa que o servidor de aplicação é que deve gerenciar o início e o fim das transações globais do EJB. A **Listagem 4** demonstra essa configuração.

Listagem 3. EJB – gerenciamento de transações do tipo BEAN (Bean Management Transaction, BMT).

```
import javax.ejb.Stateless;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
```

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class OSService {
```

```
}
```

Listagem 4. EJB – gerenciamento de transações do tipo CONTAINER (Container Management Transaction, CMT).

```
import javax.ejb.Stateless;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
```

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class OSService {
```

```
}
```

BMT – Gerenciamento de Transações pelo Bean

Na **Listagem 3** demonstramos a declaração de um EJB de nome **OSService** representando nosso serviço de criação de OS. Perceba que o gerenciamento da transação deve ser efetuado pelo próprio EJB, uma vez que `@TransactionManagement` possui o valor **BEAN**. Nesse caso, como acabamos de verificar, a demarcação do início e fim da transação global deve ser efetuada de forma explícita.

De acordo com a especificação JTA, para que isso seja possível, precisamos utilizar um objeto que implemente a interface `javax.transaction.UserTransaction`, objeto esse que poderá ser acessado por meio do método `getUserTransaction()` da sessão do EJB. A sessão do EJB em execução pode ser injetada por meio da anotação `@Resource`, para o tipo de variável de instância `javax.ejb.SessionContext`.

Assim, podemos utilizar o **UserTransaction** disponibilizado para demarcar o início e o fim da transação global em qualquer método que se fizer necessário do respectivo EJB. Um exemplo disso pode ser verificado na **Listagem 5**.

Perceba no código que injetamos outro EJB, que representa a camada DAO, e invocamos os respectivos métodos para criar

a ordem de serviço, o que refletirá em dados em cada banco de dados (Oracle e SQL Server). Nesse caso, para que tenhamos uma transação global, foi necessário demarcar o início e o fim da transação utilizando o **UserTransaction**, uma vez que trouxemos a responsabilidade de gerenciar as transações para o EJB — note a anotação **@TransactionManagement** com o valor **BEAN**. Ao invocar o método **begin()**, uma nova transação global é iniciada, já os métodos **commit()** e **rollback()** finalizam a transação efetuando o commit ou o rollback, respectivamente.

Listagem 5. Demarcação de transação de forma explícita pelo EJB

```
import javax.annotation.Resource;
import javax.ejb.EJB;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;

import app.dao.OSRepository;

@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class OSService {

    @EJB
    private OSRepository dao;

    @Resource
    private javax.ejb.SessionContext ejbContext;

    public String criarOrdemServico(String descricao) throws Exception {
        // implementação qualquer para obter um novo número de ordem de serviço
        String nro = "1XYZ2";

        try{
            ejbContext.getUserTransaction().begin();
            dao.criarOrdemServicoEquipeEquipamento(nro, descricao);
            dao.criarOrdemServicoEquipeLogistica(nro, descricao);
            ejbContext.getUserTransaction().commit();
        }catch(Exception e){
            ejbContext.getUserTransaction().rollback();
            throw e;
        }
        return nro;
    }
}
```

Assim, caso tudo ocorra com sucesso, efetuamos o commit da transação. Como os métodos do DAO são invocados dentro da transação, ambos sofreram commit ou rollback de forma atômica. Em breve analisaremos como isso acontece na camada DAO.

Apesar da demonstração, o tipo de gerenciamento **BEAN** raramente é utilizado. É considerada boa prática deixar que o servidor de aplicação gerencie as transações para que o EJB possa focar apenas na implementação das regras de negócio.

Vamos, então, analisar como o gerenciamento de transações — mais especificamente a demarcação — aconteceria caso o servidor de aplicação fosse o responsável por gerenciá-las.

CMT – Gerenciamento de Transações pelo Container (servidor Java EE)

Na **Listagem 4** demonstramos a declaração de um EJB de nome **OSService**, que representa, na verdade, a refatoração do EJB da **Listagem 3**. Na **Listagem 4**, **@TransactionManagement** está configurada para **CONTAINER**, sinalizando que o servidor de aplicação iniciará e finalizará as transações globais automaticamente.

Dessa forma o desenvolvimento é muito mais simples, pois precisamos apenas demarcar a criação e propagação de transações nos métodos do EJB, o que é feito utilizando a anotação **@javax.ejb.TransactionAttribute** (pertencente à especificação EJB). A **Listagem 6** demonstra o serviço **OSService** refatorado.

Listagem 6. Demarcação de transação apenas pela anotação **@TransactionAttribute**.

```
import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;

import app.dao.OSRepository;

@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class OSService {

    @EJB
    private OSRepository dao;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public String criarOrdemServico(String descricao) throws Exception {
        // implementação qualquer para obter um novo número de ordem de serviço
        String nro = "1XYZ2";

        dao.criarOrdemServicoEquipeEquipamento(nro, descricao);
        dao.criarOrdemServicoEquipeLogistica(nro, descricao);

        return nro;
    }
}
```



Observe que o método `criarOrdemServico()` está menos verbo. Quando `@TransactionManagement` é configurada para **CONTAINER**, a anotação `@TransactionAttribute` de cada método do EJB indica para o servidor de aplicação se ele deverá criar ou não uma transação global assim que o método for invocado.

Essa anotação foi configurada com o valor **REQUIRED**, indicando uma transação requerida. Portanto, o servidor de aplicação irá criar (e posteriormente finalizar) uma transação global assim que esse método for invocado. Se o método completar com sucesso, a transação recebe commit, mas se o servidor identificar lançamento de exceções para fora do método demarcado pela anotação, a transação recebe rollback, de forma totalmente automática.

Aqui, analisamos os tipos de gerenciamento de transação de um EJB (**BEAN** e **CONTAINER**) e como demarcar o início e o fim da transação em cada caso. Precisamos agora analisar como os métodos do DAO, invocados pelo EJB, viabilizam a participação de cada banco de dados na transação global criada para o EJB.

Nota

Se não declaradas, as anotações `@TransactionAttribute` e `@TransactionManagement` recebem os valores padrões **REQUIRED** e **CONTAINER**, respectivamente.

Como o servidor orquestra os recursos (bancos de dados) da transação global

Quando uma transação global é criada, seja utilizando um gerenciamento do tipo **BEAN** ou **CONTAINER**, a mesma é propagada pelo call stack, ou seja, para métodos de outros EJBs invocados ao longo do processamento da thread. Isso significa que a mesma transação criada no método `criarOrdemServico()` é propagada tanto para o método `criarOrdemServicoEquipeEquipamento()` quanto para o método `criarOrdemServicoEquipeLogistica()` do DAO. Vejamos o código do DAO na *Listagem 7*.



Nota

Em Ciência da Computação, call stack representa a pilha de chamada (pilha de execução) que armazena as rotinas invocadas em uma thread.

Listagem 7. OSRepository – injeção de fontes de dados representando cada banco de dados relacional.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.sql.DataSource;

@Stateless
public class OSRepository {

    @Resource(lookup="jdbc/oracle")
    private DataSource oracle;

    @Resource(lookup="jdbc/sqlserver")
    private DataSource sqlserver;

    public boolean criarOrdemServicoEquipeEquipamento(
        String nro, String descricao) throws SQLException{
        Connection conexao = oracle.getConnection();
        PreparedStatement st = conexao.prepareStatement
            ("INSERT INTO tbordemservicoeq VALUES (?, ?)");
        st.setString(1, nro);
        st.setString(2, descricao);
        st.executeUpdate();
        return true;
    }

    public boolean criarOrdemServicoEquipeLogistica(String nro, String descricao)
        throws SQLException{
        Connection conexao = oracle.getConnection();
        PreparedStatement st = conexao.prepareStatement
            ("INSERT INTO tbordemservicologis VALUES (?, ?)");
        st.setString(1, nro);
        st.setString(2, descricao);
        st.executeUpdate();
        return true;
    }
}
```

Perceba que o DAO efetua a injeção de dois Data Sources, cada um representando um banco de dados relacional — SQL Server e Oracle —, registrados no servidor de aplicação por nomes JNDI específicos.

Nesse momento, assim que os métodos `criarOrdemServicoEquipeEquipamento()` e `criarOrdemServicoEquipeLogistica()` forem invocados, o servidor de aplicação identifica a injeção de recursos (por meio da anotação `@Resource`) que ele próprio gerencia e tenta fazer com que os Data Sources se unam à transação que está sendo propagada do EJB anterior. Isso é chamado de **Resource Enlistment**, ou seja, envolver um recurso em uma transação global.

Transaction Manager – JTA

Quando uma transação global existir, deverá existir também um componente externo responsável por controlá-la, efetuando a comunicação necessária com todos os recursos nela envolvidos. Esse componente é chamado de Transaction Manager, sendo provido pelo servidor de aplicação Java EE e definido na especificação JTA.

A especificação JTA também define a forma como o Transaction Manager se comunicará com os recursos envolvidos: por meio da utilização do protocolo *Two Phase Commit Protocol*. Esse protocolo é herdado da especificação padrão da indústria para transações distribuídas, chamada X/Open XA e mantida pelo *The Open Group*.

Two Phase Commit Protocol

O protocolo recebe esse nome porque realmente possui duas fases. Como podemos perceber na **Listagem 7**, cada método utiliza seu respectivo Data Source para efetuar um INSERT (hipotético) no banco de dados. Assim que os métodos retornam (call stack), o Transaction Manager entra em ação, separando a comunicação com os recursos (Data Sources, neste caso) em duas fases.

Na primeira fase, o Transaction Manager pergunta para cada recurso da transação se ele é capaz de efetuar o commit das operações efetuadas sobre o mesmo, no nosso exemplo, um INSERT. Entretanto, note que apenas é questionado se o recurso pode efetuar o commit, não é solicitado para que o faça imediatamente. Assim, cada recurso responde de forma positiva ou negativa, ou seja, sim ou não.

Na segunda fase, o Transaction Manager analisa as respostas de cada recurso. Se ele perceber que todas as respostas foram favoráveis, ou seja, não há eventuais erros que gerariam exceções, ele solicita a cada recurso que efetue o commit de fato, e a operação é efetuada atomicamente. Entretanto, se pelo menos um recurso responder negativamente na primeira fase, o Transaction Manager entende que a atomicidade de commit de todos os recursos não será mais possível e solicita a cada recurso que efetue o rollback da transação, deixando os dados da forma como estavam quando a transação global foi criada.

Desse modo, esse protocolo garante a comunicação do Transaction Manager com os recursos envolvidos na transação. Para que o recurso suporte esse protocolo de comunicação — *Two Phase Commit Protocol* — ele deverá ser considerado um recurso XA, comportamento esse descrito na especificação JTA.

Recurso XA

Para que o Transaction Manager consiga se comunicar com um dado recurso por meio do *Two Phase Commit Protocol*, o recurso deverá suportar essa comunicação. Isso significa, para a especificação JTA, que o recurso é XA, isto é, que ele implementa a interface `javax.transaction.xa.XAResource`. Como o servidor de aplicação utiliza drivers JDBC para se comunicar com os bancos de dados relacionais, o driver deverá implementar essa interface.

A boa notícia é que a maioria dos drivers JDBC atuais, como os drivers para o MySQL, SQL Server, Oracle, PostgreSQL, entre outros, viabilizam a implementação dessa interface. Apenas para

exemplificar, podemos ver na **Listagem 8** que o driver do SQL Server (`sqljdbc4.jar`) realmente contém uma classe que provê a implementação da interface `XAResource`.

Listagem 8. SQLServerXAResource, driver JDBC versão 4 do SQL Server.

```
package com.microsoft.sqlserver.jdbc;

import java.sql.CallableStatement;
import java.sql.SQLException;
import java.text.MessageFormat;
import java.util.Properties;
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.transaction.xa.XAException;
import javax.transaction.xa.XAResource;
import javax.transaction.xa.Xid;

public final class SQLServerXAResource implements XAResource
{
    private int timeoutSeconds;
    static final int XA_START = 0;
    static final int XA_END = 1;
    static final int XA_PREPARE = 2;

    // ... demais conteúdo da classe
}
```

Mas, por que essa interface? Basta analisarmos alguns métodos presentes nela e logo identificaremos que, por meio dela, o Transaction Manager é capaz de se comunicar com um recurso e pedir para iniciar, finalizar, suspender ou efetuar commit ou rollback de uma transação, exatamente como explicamos no protocolo Two Phase Commit.



Método da Interface	Descrição
void start(Xid xid, int flags)	Esse método permite ao Transaction Manager solicitar a inicialização de uma nova transação para o recurso em que foi invocado.
void end(Xid xid, int flags)	Esse método permite ao Transaction Manager solicitar a finalização de uma transação para o recurso em que foi invocado.
int prepare(Xid xid)	Esse método permite ao Transaction Manager questionar ao recurso em específico se o mesmo é capaz de comitar a transação identificada pelo ID informado (Xid), ou seja, o que realmente acontece na fase 1.
void commit(Xid xid, boolean onePhase)	Esse método permite ao Transaction Manager solicitar ao recurso que, de fato, efetue o commit da transação identificada pelo ID informado (Xid).
void rollback(Xid xid)	Esse método permite ao Transaction Manager solicitar ao recurso que, de fato, efetue o rollback da transação identificada pelo ID informado (Xid).

Tabela 1. Alguns métodos da interface javax.transaction.xa.XAResource

Nota

Além de o driver JDBC suportar uma implementação XA, o próprio SGBD também deve suportar transações distribuídas, suporte esse que deve estar ativado. A maneira de fazer isso é específica a cada banco e foge do escopo deste artigo, portanto consulte a documentação do SGBD que estiver utilizando.

A **Tabela 1** demonstra alguns métodos dessa interface, confirmindo o que mencionamos.

Ademais, lembre-se que um Data Source ser um recurso XA viabilizado pelo driver JDBC é apenas parte da solução. Precisamos, ainda, informar ao servidor de aplicação que o mesmo é do tipo XA. Isso é feito no momento da declaração do Data Source, e como declarar isso difere de um servidor Java EE para outro. Podemos ver na **Listagem 9**, por exemplo, como declarar um XA Data Source no WildFly 8, mais precisamente no arquivo *standalone.xml*.

No WebSphere Liberty 8.5, por sua vez, declaramos de outra maneira, conforme o exemplo demonstrado na **Listagem 10**.



Listagem 9. WildFly 8 – Declaração de XA Data Source (standalone.xml).

```
<xa-datasource jndi-name="jdbc/NOME_JNDI" pool-name="NOME DATA SOURCE">
<driver>h2</driver>
<xa-datasource-property name="URL">URL JDBC</xa-datasource-property>
<xa-pool>
<min-pool-size>QUANTIDADE MINIMA POOL</min-pool-size>
<max-pool-size>QUANTIDADE MAXIMA POOL</max-pool-size>
<prefill>true</prefill>
</xa-pool>
<security>
<user-name>USUARIO</user-name>
<password>SENHA</password>
</security>
</xa-datasource>
```

Listagem 10. WebSphere Liberty 8.5 – Declaração de XA Data Source (server.xml).

```
<dataSource id="DefaultDataSource" jndiName="NOME JNDI"
type="javax.sql.XADatasource">
<jdbcDriver>
<library>
<fileset dir="/drivers/" includes="sqljdbc4.jar"/>
</library>
</jdbcDriver>

<connectionManager maxPoolSize="QUANTIDADE MAXIMA POOL"
minPoolSize="QUANTIDADE MINIMA POOL"/>

<properties.microsoft.sqlserver
databaseName="NOME DO BANCO DE DADOS"
user="USUARIO DO BANCO"
password="SENHA DO BANCO"
serverName="SERVIDOR DO BANCO"
portNumber="PORTA"/>
</dataSource>
```

No caso do WebSphere Liberty 8.5, um Data Source do tipo `javax.sql.XADataSource` é capaz de criar conexões XA representadas pela interface `javax.sql.XAConnection`, as quais irão fornecer `javax.transaction.xa.XAResource`.

Portanto, os recursos do DAO são envolvidos na transação global sendo propagada e o Transaction Manager se comunicará com eles por meio da interface `javax.transaction.xa.XAResource`, orquestrando a transação como um todo.

Nota

É importante consultar a documentação do servidor de aplicação Java EE que você está utilizando para entender como declarar um XA Data Source.

Resource Enlistment

Já mencionamos que o Transaction Manager consegue conceber o Two Phase Commit comunicando-se com os recursos por meio da interface `javax.transaction.xa.XAResource`. Mas como o recurso foi envolvido na transação global pelo servidor?

Podemos analisar na **Listagem 11** uma implementação hipotética do servidor de aplicação para envolver o recurso transacional em uma respectiva transação global da aplicação, processo chamado de *Resource Enlistment*.

Listagem 11. Implementação hipotética (mas válida) para viabilizar o mecanismo de Resource Enlistment.

```
// por meio do Transaction Manager, obtém uma nova transação para o contexto  
// da thread atual  
javax.transaction.Transaction transaction = ...  
XAConnection xaConnection = ((javax.sql.XADataSource) ds).getXAConnection();  
XAResource xaRes = xaConnection.getXAResource();  
transaction.enlistResource(xaRes);  
Connection con = xaConnection.getConnection();
```

Primeiramente, a transação é criada (ou propagada, dependendo do call stack). Assim que o servidor de aplicação identifica a transação, ele analisa se há recursos que devem ser envolvidos na mesma, como instâncias do tipo `javax.sql.DataSource`, injetadas em um DAO por meio da anotação `@Resource`. Na **Listagem 11**, ao identificar o Data Source, o servidor o transforma em uma fonte de dados XA, obtém uma conexão XA e, finalmente, o XAResource, que logo é envolvido na transação. Desse momento em diante, o Transaction Manager gerencia a transação global comunicando-se com os recursos, nesse caso XA Data Sources, por meio do *Two Phase Commit Protocol*.

Apenas um recurso na transação global

Caso a aplicação utilize apenas um Data Source, ainda assim é possível termos uma transação global, visto que a própria especificação EJB viabiliza a existência de transações. Entretanto, se o servidor envolver apenas um recurso na transação global, o Transaction Manager poderá, eventualmente, otimizar a comuni-

cação com esse recurso ignorando o protocolo Two Phase Commit. Uma vez que há apenas um recurso, commit ou rollback podem ser solicitados diretamente para o recurso.

Nota

A transação global é um mecanismo para orquestrar os recursos transacionais envolvidos na mesma, por meio do Transaction Manager, mas, obviamente, cada recurso terá suas próprias transações locais criadas e gerenciadas por si próprio.

Non-XA Data Source

Mesmo se o driver JDBC viabilizar a implementação de um `XAResource`, se o Data Source for declarado no servidor de aplicação como sendo comum, ou seja, um Non-XA Data Source, esse recurso não poderá ser envolvido em transações globais que tenham mais de um recurso envolvido, pois não representaria um `XAResource` e não poderia se comunicar com o Transaction Manager.

JPA – Java Persistence API

Caso a aplicação utilize JPA, o `persistence.xml` também deverá conter as declarações de Data Sources corretamente, utilizando a tag `<jta-data-source>` ou `<non-jta-data-source>`, para os respectivos lookups JNDI de cada Data Source registrado no servidor de aplicação.

Outros tipos de recursos XA

Além disso, o Java EE possibilita outros tipos de recursos XA que podem ser envolvidos em transações globais, além dos bancos de dados relacionais. Filas JMS, por exemplo, também podem ser transacionadas. Nesse caso, mensagens enviadas por uma sessão JMS transacionada serão recebidas pelo respectivo consumidor apenas se a transação global na qual foram envolvidas receber commit com sucesso. Caso a transação receba rollback, as mensagens poderão ser descartadas antes de serem consumidas.

Nota

Perceba que as tecnologias da plataforma Java EE trabalham de forma integrada. Diversas interfaces de diversas especificações trabalham em conjunto e de forma harmoniosa, como as especificações JDBC e JTA.

Como controlar a propagação de transações em EJBs

Explicamos que, por padrão, quando um EJB invoca outro, uma transação global criada no primeiro EJB é propagada para o segundo. Entretanto, a anotação `@TransactionAttribute` permite controlar de forma mais refinada se uma transação é propagada, suspensa, entre outros comportamentos. É exatamente isso que analisaremos neste tópico.

A anotação `@TransactionAttribute` pode receber um dos seis valores a seguir: `REQUIRED`, `REQUIRES_NEW`, `MANDATORY`, `NOT_SUPPORTED`, `SUPPORTS` e `NEVER`. Para que possamos explicar cada um, vamos nos basear no código da **Listagem 6**,

onde o primeiro EJB, o **OSService**, declara `@TransactionAttribute` como **REQUIRED** no método `criarOrdemServico()`.

Como sabemos, assim que esse método for invocado, uma transação global será criada. Se alterarmos o valor da anotação `@TransactionAttribute` do método `criarOrdemServicoEquipeEquipamento` de **OSRepository**, que é invocado pelo primeiro EJB, o comportamento de criação e propagação da transação global será alterado. Vamos, portanto, analisar cada possível valor para essa anotação. Podemos chamar o primeiro EJB, **OSService**, de **Caller**, pois ele é o chamador de outro EJB, nesse caso de **OSRepository**.

@TransactionAttribute** como **REQUIRED

Um método com o valor **REQUIRED** para a anotação `@TransactionAttribute` tem a garantia de executar em uma transação global. Na **Listagem 12**, como o método `criarOrdemServicoEquipeEquipamento()` está configurado como **REQUIRED**, se ao ser invocado já existir uma transação global sendo propagada, o mesmo se unirá a ela e executará dentro da mesma transação (o que de fato acontecerá no exemplo). Se uma transação não estiver sendo propagada, o servidor de aplicação irá criar uma nova para garantir que esse método execute dentro de uma transação.

@TransactionAttribute** como **REQUIRES_NEW

Um método com o valor **REQUIRES_NEW** para a anotação `@TransactionAttribute` tem a garantia de executar em uma transação global. A diferença entre esse e o **REQUIRED** é que nesse uma nova transação sempre será criada, independentemente se há transação global sendo propagada para o método ou não. Nesse caso, a transação do **OSService** sendo propagada é temporariamente suspensa até que o método do DAO termine de executar. Podemos ver essa configuração na **Listagem 13**.



Listagem 12. `OSRepository` – método `criarOrdemServicoEquipeEquipamento()` como **REQUIRED**.

```
@Stateless  
public class OSRepository {  
  
    @Resource(lookup="jdbc/oracle")  
    private DataSource oracle;  
  
    @Resource(lookup="jdbc/sqlserver")  
    private DataSource sqlserver;  
  
    @TransactionAttribute(TransactionAttributeType.REQUIRED)  
    public boolean criarOrdemServicoEquipeEquipamento(  
        String nro, String descricao) throws SQLException{  
        Connection conexao = oracle.getConnection();  
        PreparedStatement st = conexao.prepareStatement(  
            ("INSERT INTO tb_ordem_servico_eq VALUES (?, ?)");  
        st.setString(1, nro);  
        st.setString(2, descricao);  
        st.executeUpdate();  
        return true;  
    }  
  
    // ... restante da implementação  
}
```

Listagem 13. `OSRepository` – método `criarOrdemServicoEquipeEquipamento()` como **REQUIRES_NEW**.

```
@Stateless  
public class OSRepository {  
  
    // ... trecho idêntico ao mesmo bloco de código da Listagem 12  
  
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)  
    public boolean criarOrdemServicoEquipeEquipamento(  
        String nro, String descricao) throws SQLException{  
        // ... trecho idêntico ao mesmo bloco de código da Listagem 12  
    }  
  
    // ... restante da implementação  
}
```

@TransactionAttribute** como **MANDATORY

Um método com o valor **MANDATORY** para a anotação `@TransactionAttribute` também tem a garantia de executar em uma transação global. Entretanto, como a transação é mandatória, deve obrigatoriamente existir uma transação sendo propagada. Se não existir, uma exceção é lançada pelo servidor de aplicação. Um exemplo dessa configuração é apresentado na **Listagem 14**.

@TransactionAttribute** como **NOT_SUPPORTED

Um método com o valor **NOT_SUPPORTED** para a anotação `@TransactionAttribute` não suporta transação global e, portanto, executará sem participar de qualquer transação. Se uma transação global estiver sendo propagada, a mesma é suspensa até que o método termine de executar. A **Listagem 15** mostra um exemplo dessa configuração.

Listagem 14. OSRepository – método criarOrdemServicoEquipeEquipamento() como MANDATORY.

```
@Stateless  
public class OSRepository {  
  
    // ... trecho idêntico ao mesmo bloco de código da Listagem 12  
  
    @TransactionAttribute(TransactionAttributeType.MANDATORY)  
    public boolean criarOrdemServicoEquipeEquipamento  
(String nro, String descricao) throws SQLException{  
        // ... trecho idêntico ao mesmo bloco de código da Listagem 12  
    }  
  
    // ... restante da implementação  
}
```

Listagem 15. OSRepository – método criarOrdemServicoEquipeEquipamento() como NOT_SUPPORTED.

```
@Stateless  
public class OSRepository {  
  
    // ... trecho idêntico ao mesmo bloco de código da Listagem 12  
  
    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)  
    public boolean criarOrdemServicoEquipeEquipamento  
(String nro, String descricao) throws SQLException{  
        // ... trecho idêntico ao mesmo bloco de código da Listagem 12  
    }  
  
    // ... restante da implementação  
}
```

tipo de gerenciamento de transações seja **CONTAINER**, ou seja, caso a anotação **@TransactionManagement** seja declarada com o valor **CONTAINER**.

Listagem 16. OSRepository – método criarOrdemServicoEquipeEquipamento() como SUPPORTS.

```
@Stateless  
public class OSRepository {  
  
    // ... trecho idêntico ao mesmo bloco de código da Listagem 12  
  
    @TransactionAttribute(TransactionAttributeType.SUPPORTS)  
    public boolean criarOrdemServicoEquipeEquipamento  
(String nro, String descricao) throws SQLException{  
        // ... trecho idêntico ao mesmo bloco de código da Listagem 12  
    }  
  
    // ... restante da implementação  
}
```

Listagem 17. OSRepository – método criarOrdemServicoEquipeEquipamento() como NEVER.

```
@Stateless  
public class OSRepository {  
  
    // ... trecho idêntico ao mesmo bloco de código da Listagem 12  
  
    @TransactionAttribute(TransactionAttributeType.NEVER)  
    public boolean criarOrdemServicoEquipeEquipamento  
(String nro, String descricao) throws SQLException{  
        // ... trecho idêntico ao mesmo bloco de código da Listagem 12  
    }  
  
    // ... restante da implementação  
}
```

@TransactionAttribute como SUPPORTS

Um método com o valor **SUPPORTS** para a anotação **@TransactionAttribute** indica ao servidor de aplicação que suporta executar em uma transação caso uma esteja sendo propagada, mas essa transação não é obrigatória. Se uma transação global estiver sendo propagada, o método executa na mesma transação, caso contrário, executa normalmente fora dela. Vejamos um exemplo para essa configuração na **Listagem 16**.

@TransactionAttribute como NEVER

Por fim, um método com o valor **NEVER** para a anotação **@TransactionAttribute** indica que ele executará, obrigatoriamente, fora de uma transação global. O método, definitivamente, não suporta transação global, portanto, se uma transação estiver sendo propagada para o mesmo, ela não será suspensa porque nesse caso é considerada um erro, fazendo com que uma exceção seja lançada pelo servidor de aplicação. Podemos ver essa configuração na **Listagem 17**.

Perceba como a especificação EJB define uma variedade de mecanismos de controle de propagação de transação global, viabilizados por meio da anotação **@TransactionAttribute**. É importante mencionar que a anotação apenas terá efeito caso o



Nota

Este artigo tem como intuito explicar os conceitos de transações locais e globais e parte da especificação JTA, não sendo propósito do mesmo definir as melhores práticas de desenvolvimento de cada camada, como utilização de JPA, declaração de view remote e local e implementação de interfaces nos EJBs. O detalhamento dessas práticas está fora de escopo deste artigo.

Transações distribuídas não são apenas úteis, mas muitas vezes necessárias. Portanto, faça uso dos conceitos que você aprendeu por meio deste artigo para que seja capaz de viabilizar requisitos não funcionais, tais como a existência de transações distribuídas, caso isso seja necessário para os sistemas corporativos nos quais você esteja envolvido.

Autor



Michael Henrique R. Pereira

michaelhenrique182@gmail.com

<https://br.linkedin.com/in/michaelhenriquepublic>

Arquiteto de software, possui conhecimento de diversas tecnologias de front-end e back-end e dá cursos de tecnologias da plataforma Java EE e JavaScript. Possui as certificações SCJP 6, SCBCD 5 e Open Span Certified Developer. É autor do livro AngularJS – Uma abordagem prática e objetiva, pela editora Novatec.



Links:

Endereço para download da especificação JTA.

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

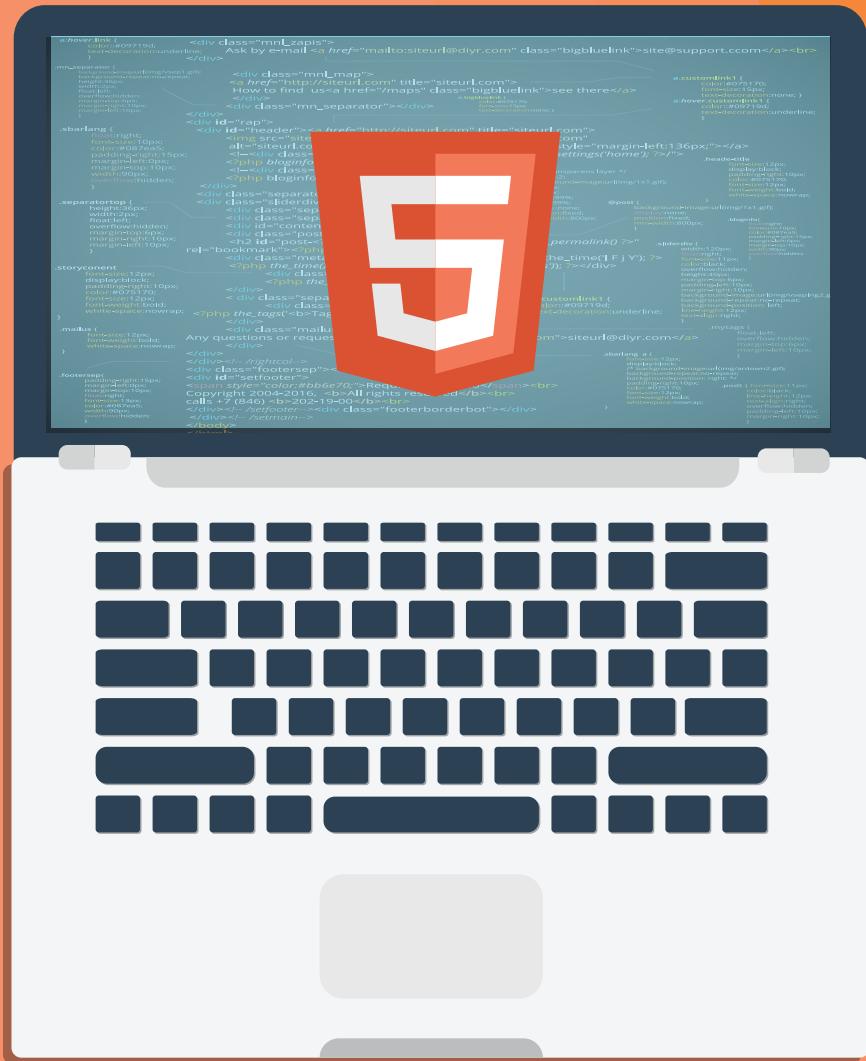
Endereço para download da especificação EJB.

<http://www.oracle.com/technetwork/java/docs-135218.html>



Guia HTML 5

Um verdadeiro manual de referência com tudo que você precisa sobre HTML!



DEVMEDIA

<http://www.devmedia.com.br/guias/guia-html/3>

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