



Edição 142 :: R\$ 14,90

DESTAQUE:
Primeiros passos na programação reativa com Vert.x
Conheça a plataforma para desenvolvimento
de aplicações web e serviços

Relatórios com Hibernate e JasperReports
Desenvolva relatórios avançados com
virtualização e paginação

Conhecendo o PrimeFaces 5.1
Aprenda a utilizar os novos componentes

PREPARE-SE PARA O FAST DATA

**Big Data em tempo real
com Storm e Spark**



Java EE 7 na prática
Como criar a lógica de negócios
com EJB e o controle de acesso com filtros

Definindo a interface para a web
Como escolher a opção mais adequada
para a interface de suas aplicações Java EE

DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

FORMAÇÃO DESENVOLVEDOR JAVA

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

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer





Edição 142 • 2015 • ISSN 1676-8361



MVP

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

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Diogo Souza (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa 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

CURSOS ONLINE

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



CONHEÇA ALGUNS DOS CURSOS:

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

Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Sumário

Artigo no estilo Curso

06 – Como construir uma aplicação de Controle de Projetos com Java EE – Parte 2

[Edmar Elias Bregagnoli]

Conteúdo sobre Novidades, Artigo no estilo Solução Completa

18 – Storm vs. Spark: uma introdução à Fast Data

[Luiz Henrique Zambom Santana e Eduardo Felipe Zambom Santana]

Artigo no estilo Curso

28 – Relatórios avançados com Hibernate, JasperReports e PrimeFaces – Parte 1

[Marcos Vinicios Turisco Dória]

Conteúdo sobre Boas Práticas, Conteúdo sobre Novidades

42 – Primeiros passos na programação reativa com Vert.x

[Michel Graciano]

Conteúdo sobre Novidades, Artigo no estilo Solução Completa

52 – PrimeFaces 5.1 na prática

[Eduardo Felipe Zambom Santana e Luiz Henrique Zambom Santana]

Conteúdo sobre Boas Práticas

62 – Como definir a tecnologia para o desenvolvimento da interface web

[Emerson Sachio Saito]



Dê seu feedback sobre esta edição!

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

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

www.devmedia.com.br/javamagazine/feedback



CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

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

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

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

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

Na suite ToolsCloud você tem a sua disposição:

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

Acompanhe a Toolscloud:



toolscloud@toolscloud.com



twitter.com/toolscloud



Como construir uma aplicação de Controle de Projetos com Java EE

– Parte 2

Veja nesse artigo como criar a lógica de negócios com EJB e um controle de acesso utilizando filtros

ESTE ARTIGO FAZ PARTE DE UM CURSO

Na primeira parte desta série de artigos apresentamos as principais novidades introduzidas pelo Java EE 7 a partir do desenvolvimento de uma aplicação corporativa de Controle de Projetos. No artigo da edição anterior criamos toda a estrutura necessária para o desenvolvimento da aplicação, começando pelo banco de dados, configuração do Maven para o controle de dependências e criação da arquitetura da aplicação.

Ainda na primeira parte, criamos um serviço REST fazendo uso da versão 2 da especificação JAX-RS para popular as tabelas de cidades e estados do banco de dados e, por fim, foi desenvolvida a primeira tela da aplicação.

A partir de agora, continuaremos o desenvolvimento de nossa aplicação exemplo dando maior destaque à implementação de uma lógica de negócio para alocação de colaboradores aos projetos. Para isso, demonstraremos a criação de duas páginas web: uma para o cadastro de colaboradores e outra para a alocação dos colaboradores nos projetos.

Por fim, vamos construir uma página de login para realizarmos o controle de acesso ao sistema. A imple-

Fique por dentro

Com a plataforma Java EE é possível criar sistemas que atendam praticamente a todas as necessidades de uma corporação. Por isso, nesse artigo vamos continuar o desenvolvimento de um sistema de Controle de Projetos demonstrando a criação de uma funcionalidade para alocação de colaboradores nos projetos da empresa. Vamos mostrar também como criar um sistema de controle de acesso utilizando o recurso de filtros, presente na API de Servlets. E para auxiliar no desenvolvimento destas funcionalidades, utilizaremos os componentes visuais do PrimeFaces para criação das telas do sistema.

mentação dessa funcionalidade será realizada através de filtros, demonstrando como podemos utilizar esse recurso da API de Servlets para garantir a segurança de nossa aplicação.

Criando a página de cadastro de colaboradores

Vamos iniciar a segunda parte do artigo criando a página de cadastro de colaboradores. Para tanto, crie o arquivo *Cadastro-Colaborador.xhtml* dentro do diretório *src/main/webapp/colaborador*. O código dessa página encontra-se na **Listagem 1**. Como podemos verificar, temos vários componentes do PrimeFaces, porém neste tópico vamos destacar aqueles que ainda não foram apresentados nesta série de artigos.

Com isso, começaremos pelo componente `<p:selectOneRadio>`, presente na linha 24. Este gera dois *radio buttons* na tela para que

o usuário possa escolher entre a opção *Masculino* e *Feminino* para o campo *Sexo* no cadastro de colaborador. As opções que podem ser escolhidas para este campo são definidas por meio do componente **<f:selectItem>**, onde cada componente **<f:selectItem>**, definido em **<p:selectOneRadio>**, acaba se tornando um *radio button* em específico.

Outro componente bastante útil em um formulário de cadastro é o **<p:calendar>**, presente na linha 30. Esse componente fornece um calendário para que o usuário escolha uma data auxiliando-o no preenchimento podendo, inclusive, receber um *pattern* de data (neste caso, no formato *dd/MM/yyyy*). Além disso, o componente de calendário do PrimeFaces permite que seja definida uma máscara para datas através da propriedade **mask**.

Para realizar o cadastro de colaboradores é necessário que seja informado o departamento onde o mesmo está alocado. Uma das formas possíveis de realizar essa tarefa é disponibilizando um componente *dropdown* onde apareçam todos os departamentos da empresa para que o usuário possa selecionar um deles. O componente **<p:selectOneMenu>** atende perfeitamente a essa demanda e pode ser encontrado na linha 40. Para carregar esse componente com os departamentos da empresa é necessário utilizar o componente **<f:selectItems>**, que itera sobre uma lista de objetos exibindo dentro do componente **<p:selectOneMenu>** os valores dessa lista. Esse componente é responsável por receber o atributo **listaDepartamentos** do bean **ColaboradorBean**, como mostra a linha 43.

Listagem 1. Código fonte da página CadastroColaborador.xhtml.

```

01. <ui:composition template="/WEB-INF/template/template.xhtml"
02.   xmlns="http://www.w3.org/1999/xhtml"
03.   xmlns:h="http://java.sun.com/jsf/html"
04.   xmlns:f="http://java.sun.com/jsf/core"
05.   xmlns:ui="http://java.sun.com/jsf/facelets"
06.   xmlns:p="http://primefaces.org/ui"
07.   xmlns:o="http://omnifaces.org/ui">
08.
09. <ui:define name="titulo">Cadastro de Colaborador</ui:define>
10.
11. <ui:define name="corpo">
12.
13. <h1>Cadastro de Colaborador</h1>
14.
15. <h:form id="frmCadastro">
16.
17. <p:messages id="messages" autoUpdate="true" closable="true"/>
18. <p:panelGrid columns="2" id="painel" style="width: 100%; margin-top: 20px" columnClasses="rotulo, campo">
19.
20. <p:outputLabel value="Nome" for="nome"/>
21. <p:inputText id="nome" value="#{colaboradorBean.colaborador.nome}" required="true" requiredMessage="Campo Nome obrigatorio"/>
22.
23. <p:outputLabel value="Sexo" for="sexo"/>
24. <p:selectOneRadio id="sexo" value="#{colaboradorBean.colaborador.sexo}" required="true" requiredMessage="Campo Sexo obrigatorio"/>
25.
26. <f:selectItem itemLabel="Masculino" itemValue="M"/>
27. <f:selectItem itemLabel="Feminimo" itemValue="F"/>
28. </p:selectOneRadio>
29. <p:outputLabel value="Data de Nascimento" for="dataNascimento"/>
30. <p:calendar id="dataNascimento" value="#{colaboradorBean.colaborador.dataNascimento}" pattern="dd/MM/yyyy" mask="true" required="true" requiredMessage="Campo Data de Nascimento obrigatorio"/>
31.
32. <p:outputLabel value="Remuneracao" for="remuneracao"/>
33. <p:inputText id="remuneracao" value="#{colaboradorBean.colaborador.remuneracao}" required="true" requiredMessage="Campo Remuneracao obrigatorio"/>
34.
35. <p:outputLabel value="E-mail" for="email"/>
36. <p:inputText id="email" value="#{colaboradorBean.colaborador.email}"/>
37.
38. <p:outputLabel value="Departamento" for="departamento"/>
39. <p:selectOneMenu id="departamento" value="#{colaboradorBean.colaborador.departamento}" converter="departamentoConverter" required="true" requiredMessage="Campo Departamento obrigatorio"/>
40. <f:selectItems value="#{colaboradorBean.listaDepartamentos}" var="departamento" itemLabel="#{departamento.nome}" itemValue="#{departamento}"/>
41. </p:selectOneMenu>
42.
43. <p:outputLabel value="Estado" for="estado"/>
44. <p:selectOneMenu id="estado" value="#{colaboradorBean.estadoSelecionado}" converter="estadoConverter" required="true" requiredMessage="Campo Estado obrigatorio">
45. <f:selectItem itemLabel="Selecione um Estado"/>
46. <f:selectItems value="#{colaboradorBean.listaEstados}" var="estado" itemLabel="#{estado.nome}" itemValue="#{estado}"/>
47. <p:ajax listener="#{colaboradorBean.carregarCidades}" update="cidade"/>
48. </p:selectOneMenu>
49.
50. <p:outputLabel value="Cidade" for="cidade"/>
51. <p:selectOneMenu id="cidade" value="#{colaboradorBean.colaborador.cidade}" converter="cidadeConverter" required="true" requiredMessage="Campo Cidade obrigatorio">
52. <f:selectItem itemLabel="Selecione uma Cidade"/>
53. <f:selectItems value="#{colaboradorBean.listaCidades}" var="cidade" itemLabel="#{cidade.nome}" itemValue="#{cidade}"/>
54. </p:selectOneMenu>
55.
56. <p:outputLabel value="Cidade" for="cidade"/>
57. <p:selectOneMenu id="cidade" value="#{colaboradorBean.colaborador.cidade}" converter="cidadeConverter" required="true" requiredMessage="Campo Cidade obrigatorio">
58. <f:selectItem itemLabel="Selecione uma Cidade"/>
59. <f:selectItems value="#{colaboradorBean.listaCidades}" var="cidade" itemLabel="#{cidade.nome}" itemValue="#{cidade}"/>
60. </p:selectOneMenu>
61.
62. </p:panelGrid>
63.
64. <p:toolbar style="margin-top: 20px">
65.
66. <ff:facet name="left">
67. <p:commandButton value="Salvar" action="#{colaboradorBean.salvar}" update="@form"/>
68. </ff:facet>
69.
70. </p:toolbar>
71. </p:toolbar>
72.
73. </p:toolbar>
74.
75. </h:form>
76.
77. </ui:define>
78.
79. </ui:composition>

```

Como podemos observar, o componente `<f:selectItems>` possui quatro propriedades principais que merecem destaque:

- **value:** Lista de objetos que contém as informações a serem renderizadas pelo componente;
- **var:** Referência ao objeto da lista que está sendo iterada para que possamos acessar os atributos desse objeto;
- **itemLabel:** Informação a ser renderizada na tela. Neste caso, o nome do departamento;
- **itemValue:** Informação que será passada para o bean quando a página for submetida. Neste caso, um objeto do tipo **Departamento**.

Continuando a análise do componente `<p:selectOneMenu>`, presente na linha 40 vemos que o mesmo utiliza o converter **DepartamentoConverter**. O código desse converter pode ser analisado na **Listagem 2**.

Listagem 2. Código da classe DepartamentoConverter.

```
01. @Named
02. @FacesConverter(value="departamentoConverter")
03. public class DepartamentoConverter implements Converter {
04.
05.     @Inject
06.     private ColaboradorBean colaboradorBean;
07.
08.     public Object getAsObject(FacesContext arg0, UIComponent uiComponent,
09.                               String value) {
10.
11.         if (value != null && !value.isEmpty()) {
12.
13.             Departamento departamento = colaboradorBean.
14.                 buscarDepartamentoById(Integer.valueOf(value));
15.             return departamento;
16.         }
17.
18.         return null;
19.     }
20.
21.     public String getAsString(FacesContext arg0, UIComponent uiComponent,
22.                               Object value) {
23.
24.         if (value instanceof Departamento) {
25.             Departamento depto = (Departamento) value;
26.             return String.valueOf(depto.getId());
27.         }
28.         return "";
29.     }
30. }
```

Quando a página de cadastro de colaborador é renderizada, o método `getAsString()` de **DepartamentoConverter** é executado para cada departamento e é retornado o id do departamento como uma **String**. No momento em que o formulário for submetido, o método `getAsObject()` será executado. Dentro desse método é recuperado o id do departamento que veio do componente **SelectOneMenu** para que possamos consultar as informações do departamento selecionado e setar essa informação na propriedade **value** de **SelectOneMenu** que, nesse exemplo, tem o valor `colaboradorBean.colaborador.departamento`.

Podemos notar que o converter **DepartamentoConverter** é um bean gerenciado pelo CDI, já que está anotado com `@Named` e,

por isso, pode realizar a injeção de dependência do bean **ColaboradorBean** através da anotação `@Inject`.

Essa injeção de dependência se faz necessária porque dentro do método `getAsObject()` do **DepartamentoConverter** é realizada uma chamada para o método `buscarDepartamentoById()` do **ColaboradorBean** e para isso é necessário que esse bean tenha sua dependência “injetada” para que seus métodos possam ser executados.

Prosseguindo com a construção de nossa página de cadastro, temos como exemplo mais dois componentes **SelectOneMenu**: um para estados e outro para cidades.

Esse exemplo é interessante porque são componentes dependentes, ou seja, quando é selecionado um estado, é realizada uma requisição AJAX passando o estado selecionado para o método `carregarCidades()` do bean **ColaboradorBean** e que retorna as cidades pertencentes a este estado para o **SelectOneMenu**, responsável por mostrar as cidades na tela. Ambos os componentes encontram-se nas linhas 49 a 62 e possuem o mesmo funcionamento do **SelectOneMenu** de departamento. O único detalhe está no **SelectOneMenu** que carrega os estados. Após carregar todos os estados por meio de `<f:selectItems>` na linha 52, é definida uma operação AJAX na página por meio do componente `<p:ajax>` (vide linha 53).

Com esse componente, toda vez que um usuário escolher um estado, o mesmo será passado para o bean **ColaboradorBean** no momento exato da seleção, via AJAX. A propriedade `listener` de `<p:ajax>` indica qual método do bean será executado após um estado ser selecionado. Neste caso, o método será `carregarCidades()`.

Após o método `carregarCidades()` de **ColaboradorBean** ser executado, o **SelectOneMenu** de cidades será carregado com as cidades pertencentes ao estado escolhido. Esse comportamento acontece porque o componente `<p:ajax>` indica, por meio do parâmetro `update`, o id do componente da página que deve ser atualizado sendo, neste exemplo, o componente `cidade`.

O método `carregarCidades()` pode ser analisado na linha 50 da **Listagem 3**, que mostra o código da classe **ColaboradorBean**. Note que esse método recebe como parâmetro uma instância da classe **AjaxBehaviorEvent**, visto que ele é executado a cada seleção de um estado por meio de um evento do tipo AJAX. Por fim, na linha 51, é realizada a busca das cidades pertencentes ao estado selecionado.

O último passo é salvar as informações do colaborador por meio do botão *Salvar*, presente na linha 70 da **Listagem 1**. Este chama o método de mesmo nome da classe **ColaboradorBean**. O método `salvar()`, como podemos observar nas linhas 43 a 48 da **Listagem 3**, salva as informações de **Colaborador** e em seguida retorna uma mensagem de sucesso.

A **Figura 1** mostra a tela de Cadastro de Colaboradores com todos os componentes apresentados e pronta para ser utilizada pelo nosso sistema de Controle de Projetos.

Listagem 3. Código da classe ColaboradorBean.

```
01. package br.com.devmedia.controleprojeto.mb;
02.
03. //imports omitidos...
04.
05. @Named
06.
07. @ViewScoped
08.
09. public class ColaboradorBean implements Serializable {
10.
11. private static final long serialVersionUID = 1L;
12.
13. @Inject
14. private DepartamentoService depService;
15.
16. @Inject
17. private EstadoService estadoService;
18.
19. @Inject
20. private CidadeService cidadeService;
21.
22. @Inject
23. private ColaboradorService colaboradorService;
24.
25. private Colaborador colaborador = new Colaborador();
26.
27. private List<Departamento> listaDepartamentos;
28.
29. private List<Estado> listaEstados;
30.
31. private List<Cidade> listaCidades;
32.
33. private Estado estadoSelecionado;
34.
35. @PostConstruct
36. public void init() {
37.
38.     this.colaborador = new Colaborador();
39.     listaDepartamentos = depService.buscarTodosDepartamentos();
40.     listaEstados = estadoService.buscarTodosEstados();
41. }
42.
43. public void salvar() {
44.     colaboradorService.salvarColaborador(this.colaborador);
45.
46.
47.     retornarMsgSucesso("Colaborador Cadastrado com sucesso");
48. }
49.
50. public void carregarCidades(AjaxBehaviorEvent ev) {
51.     listaCidades = cidadeService.buscarCidadesPorEstado(estadoSelecionado);
52. }
53.
54. public Departamento buscarDespartamentoById(Integer id) {
55.     return depService.buscarDeptobyId(id);
56. }
57.
58. public Estado buscarEstadoById(Integer id) {
59.     return estadoService.buscarEstadoById(id);
60. }
61.
62. public Cidade buscarCidadeById(Integer id) {
63.     return cidadeService.buscarCidadeById(id);
64. }
65.
66. private void retornarMsgSucesso(String msg) {
67.     FacesContext.getCurrentInstance().addMessage(null,
68.         new FacesMessage(FacesMessage.SEVERITY_INFO, msg, msg));
69. }
70.
71. /*métodos getters e setters omitidos*/
73.}
```

Figura 1. Tela de Cadastro de Colaboradores

Criando a página de alocação de colaboradores em projetos

A próxima página a ser criada é a de Alocação de Colaboradores aos projetos. Com essa funcionalidade será possível escolher um colaborador cadastrado e alocá-lo em um projeto, lembrando que um colaborador pode estar relacionado em mais de um projeto.

Para criar a tela de alocação, crie o arquivo *alocacao.xhtml* dentro do diretório *src/main/webapp/projeto* e deixe o código dessa página semelhante ao apresentado na **Listagem 4**.

O resultado desse código pode ser visto na **Figura 2**.

Figura 2. Tela de Alocação de Colaboradores.

Listagem 4. Código da página alocação.xhtml.

```
01. <!DOCTYPE html>
02. <ui:composition template="/WEB-INF/template/template.xhtml"
03.   xmlns="http://www.w3.org/1999/xhtml"
04.   xmlns:h="http://java.sun.com/jsf/html"
05.   xmlns:f="http://java.sun.com/jsf/core"
06.   xmlns:ui="http://java.sun.com/jsf/facelets"
07.   xmlns:p="http://primefaces.org/ui">
08.
09. <ui:define name="titulo">Alocação de Colaboradores</ui:define>
10.
11. <ui:define name="corpo">
12.   <h1>Alocação de Colaboradores</h1>
13.
14.   <h:form id="frmCadastro">
15.     <p:messages id="messages" autoUpdate="true" closable="true"/>
16.
17.     <p:panelGrid columns="2" id="painel" style="width: 100%; margin-top: 20px" columnClasses="rotulo, campo">
18.
19.       <p:outputLabel value="Projeto" for="projeto"/>
20.
21.       <p:selectOneMenu id="projeto" value="#{alocacaoBean.alocacao.projeto}" converter="projetoConverter" required="true" requiredMessage="Campo Projeto obrigatório">
22.         <f:selectItem itemLabel="Selecione um Projeto"/>
23.         <f:selectItems items="#{alocacaoBean.listaProjetos}" var="projeto" itemLabel="#{projeto.nome}" itemValue="#{projeto}"/>
24.       </p:selectOneMenu>
25.
26.       <p:outputLabel value="Colaborador" for="colaborador"/>
27.
28.       <h:panelGroup>
29.
30.         <h:panelGroup>
```

31. <p:inputText id="colaborador" value="#{alocacaoBean.nomeColaborador}" required="true" requiredMessage="Campo Colaborador obrigatório" readonly="#{facesContext.currentPhaseId.name eq 'RENDER_RESPONSE'}"/>
32.
33. <p:commandButton icon="ui-icon-search" title="Pesquisa" action="#{pesquisaColaboradorDialogBean.configurarComponentePesquisa()}" process="@this" update="@none">
34. <p:ajax event="dialogReturn" listener="#{alocacaoBean.colaboradorSelecionadoEvento}" process="@this" update="colaborador"/>
35. <p:resetInput target="colaborador"/>
36.
37. </p:commandButton>
38.
39. </h:panelGroup>
40.
41. </p:panelGrid>
42.
43. <p:toolbar style="margin-top: 20px">
44. <f:facet name="left">
45. <p:commandButton value="Realiza Alocação" action="#{alocacaoBean.salvar}" update="@form"/>
46. </f:facet>
47. </p:toolbar>
48. </h:form>
49. </ui:define>
50. </ui:composition>

Na página de alocação teremos um componente **SelectOneMenu** (vide linhas 21 a 26) responsável por exibir os projetos cadastrados no banco de dados. Esse componente tem comportamento semelhante aos demonstrados na página de Cadastro de Colaboradores. Para exibir os projetos cadastrados, utilizamos a classe converter **ProjetoConverter**. Com essa classe apresentamos no **SelectOneMenu** os dados da **listaProjetos** do bean **AlocacaoBean** (vide Listagem 5).

Além do componente **SelectOneMenu**, utilizamos o recurso Dialog Framework do PrimeFaces para pesquisar os colaboradores cadastrados no banco de dados da aplicação. Esse recurso possibilita que, em vez de termos um componente **SelectOneMenu** com todos os colaboradores cadastrados, tenhamos um componente de pesquisa com uma interface bem mais amigável ao usuário. Imagine a situação em que a base de dados tenha por volta de duzentos colaboradores. Certamente com o componente **SelectOneMenu** a usabilidade da página seria prejudicada. Por isso, é recomendável aplicarmos um componente visual mais elaborado e o PrimeFaces nos oferece esse recurso.

Para criarmos esse componente começamos definindo um **<h:panelGroup>** na página de alocação – linhas 30 a 44 da Listagem 4. Dentro deste **PanelGroup** definimos um **<h:inputText>**, na linha 31, de preenchimento obrigatório, pois para criar uma alocação é preciso que seja informado um colaborador. Além disso, definimos no parâmetro **value** a expressão **#{alocacaoBean.nomeColaborador}**, para fazer o **binding** do nome

do colaborador informado na tela de Alocação com a propriedade **nomeColaborador** do bean **AlocacaoBean**.

Como estamos criando um componente de pesquisa para buscar os colaboradores, é interessante que o campo Colaborador da tela seja **readOnly**, e neste modo seja preenchido apenas com o retorno do componente de pesquisa. Sendo assim, ainda na linha 31, na definição do **<h:inputText>** que irá receber o nome do colaborador, configuramos a propriedade **readonly** com o valor **#{facesContext.currentPhaseId.name eq 'RENDER_RESPONSE'}**.

Para abrir o componente de pesquisa de colaboradores, adicionamos na tela o botão **<p:commandButton>**, indicado na linha 34. Ao ser selecionado, o componente correspondente é aberto, como mostra a Figura 3. Nessa imagem podemos ver o componente de pesquisa de colaboradores. Este possui, basicamente, um campo texto, no qual o usuário irá informar o nome do colaborador que será pesquisado, um botão **Pesquisar**, que irá chamar uma consulta no banco de dados para retornar os colaboradores, uma grid com o resultado dessa consulta e, na frente de cada nome de colaborador, um botão de seleção que irá retornar para a tela de Alocação o colaborador selecionado.

No entanto, para que esse componente seja renderizado na tela, quando o usuário seleciona o botão de pesquisa da tela de Alocação, o método **configurarComponentePesquisa()**, do bean **PesquisaColaboradorDialogBean**, apresentado na Listagem 6, é executado.

Listagem 5. Código da classe AlocacaoBean.

```
01. package br.com.devmedia.controleprojeto.mb;
02. /*imports omitidos*/
03. @Named
04. @ViewScoped
05. public class AlocacaoBean implements Serializable{
06.
07.     private static final long serialVersionUID = 1L;
08.
09.     @Inject
10.     private ProjetoService projetoService;
11.
12.     @Inject
13.     private AlocacaoService alocacaoService;
14.
15.     private Alocacao alocacao = new Alocacao();
16.
17.     private List<Projeto> listaProjetos;
18.
19.     @PostConstruct
20.     public void init() {
21.         listaProjetos = projetoService.buscarTodosProjetos();
22.     }
23.
24.     public void colaboradorSelecionado(Evento event) {
25.         Colaborador colaborador = (Colaborador) event.getObject();
26.         this.alocacao.setColaborador(colaborador);
27.     }
28.
29.     public void salvar() {
30.         alocacaoService.salvarProjeto(alocacao);
31.
32.         retornarMsgSucesso("Alocacao realizada com sucesso");
33.     }
34.
35.     private void retornarMsgSucesso(String msg) {
36.         FacesContext.getCurrentInstance().addMessage(null,
37.             new FacesMessage(FacesMessage.SEVERITY_INFO, msg, msg));
38.     }
39.
40.     public Alocacao getAlocacao() {
41.         return alocacao;
42.     }
43.
44.     public void setAlocacao(Alocacao alocacao) {
45.         this.alocacao = alocacao;
46.     }
47.
48.     public List<Projeto> getListaProjetos() {
49.         return listaProjetos;
50.     }
51.
52.     public void setListaProjetos(List<Projeto> listaProjetos) {
53.         this.listaProjetos = listaProjetos;
54.     }
55.
56.     public String getNomeColaborador() {
57.         return alocacao.getColaborador() == null
58.             ? null : alocacao.getColaborador().getNome();
59.     }
60.
61.     public void setNomeColaborador(String nomeColaborador){
62.
63.     }
64. }
```



Figura 3. Tela de pesquisa de colaboradores

Nesse método são configuradas algumas propriedades do componente de pesquisa em um mapa, a saber: **modal** igual a **true**, para que o componente seja apresentado na tela como uma janela modal; **resizable** igual **false**, para que a janela onde o componente de pesquisa será apresentado não possa ser redimensionada; e, por fim, definimos a propriedade **contentHeight**, que ajusta a altura do componente para 470. Realizadas estas configurações no *map opcoes*, executamos o método **openDialog()**, na linha 29, responsável por renderizar o componente de pesquisa na tela. Nesse método passamos como parâmetro uma **String**, que representa o nome da página XHTML onde o componente de pesquisa foi criado (neste caso, "pesquisaColaborador") e um **map** com as configurações do componente.

Entretanto, para que esse método funcione e o componente de pesquisa de colaboradores seja renderizado, devemos criar uma

Listagem 6. Código da classe PesquisaColaboradorDialogBean.

```
01. package br.com.devmedia.controleprojeto.mb;
02.
03. /*imports omitidos*/
04.
05. @Named
06. @ViewScoped
07. public class PesquisaColaboradorDialogBean implements Serializable {
08.
09.     @Inject
10.     private ColaboradorService colaboradorService;
11.
12.     private String nome;
13.     private List<Colaborador> listaColaboradores;
14.
15.     public void pesquisar() {
16.         listaColaboradores = colaboradorService.buscarColaboradorPorNome(nome);
17.     }
18.
19.     public void selecionar(Colaborador colaborador) {
20.         RequestContext.getCurrentInstance().closeDialog(colaborador);
21.     }
22.
23.     public void configurarComponentePesquisa() {
24.         Map<String, Object> opcoes = new HashMap<>();
25.         opcoes.put("modal", true);
26.         opcoes.put("resizable", false);
27.         opcoes.put("contentHeight", 470);
28.
29.         RequestContext.getCurrentInstance().openDialog(
30.             "pesquisaColaborador", opcoes, null);
31.     }
32. }
```

nova página *.xhtml* para esse componente. Sendo assim, dentro do diretório *src/main/webapp/projeto*, crie a página *pesquisaColaborador.xhtml* com o código semelhante ao apresentado na **Listagem 7**.

Listagem 7. Código do componente de pesquisa *pesquisaColaborador.xhtml*.

```
01. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
02. <html xmlns="http://www.w3.org/1999/xhtml"
03.   xmlns:h="http://java.sun.com/jsf/html"
04.   xmlns:f="http://java.sun.com/jsf/core"
05.   xmlns:ui="http://java.sun.com/jsf/facelets"
06.   xmlns:p="http://primefaces.org/ui">
07.
08. <h:head>
09.   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
10.   <title>Pesquisa de Colaboradores</title>
11.   <h:outputStylesheet library="css" name="sistema.css"/>
12. </h:head>
13.
14. <h:body>
15.   <h:form>
16.     <div style="margin-top: 5px; margin-bottom: 20px">
17.       <p:inputText id="nome" size="40" value="#{pesquisaColaborador
18.           DialogBean.nome}" />
19.
20.       <p:commandButton value="Pesquisar"
21.         action="#{pesquisaColaboradorDialogBean.pesquisar}"
22.         update="@form"/>
23.     </div>
24.
25.     <p:dataTable value="#{pesquisaColaboradorDialogBean
26.         .listaColaboradores}" var="colaborador"
27.         emptyMessage="Nenhum Colaborador encontrado.">
28.
29.       <p:column headerText="Código" style="text-align: center; width: 40px">
30.         <h:outputText value="#{colaborador.id}" />
31.       </p:column>
32.       <p:column headerText="Nome">
33.         <h:outputText value="#{colaborador.nome}" />
34.       </p:column>
35.       <p:column style="width: 30px; text-align: center">
36.         <p:commandButton icon="ui-icon-check" title="Selecionar"
37.           action="#{pesquisaColaboradorDialogBean.
38.               selecionar(colaborador)}"
39.           process="@this"/>
40.       </p:column>
41.     </p:dataTable>
42.   </h:form>
43. </h:body>
44.
45. </html>
```

Como podemos observar nessa listagem, nosso componente é representado por uma nova página na aplicação. Essa abordagem nos dá a vantagem de poder reutilizar esse componente em vários outros pontos do sistema caso seja necessário buscar por colaboradores cadastrados no banco de dados. Para isso, basta inserir o componente nas páginas que necessitem dessa funcionalidade.

Analisando o código da página *pesquisaColaborador.xhtml*, temos na linha 17 um componente *<p:inputText>* que recebe o nome do colaborador a ser pesquisado. Esse nome é passado para o atributo *nome* do bean *PesquisaColaboradorDialogBean* através da

propriedade *value*. Em seguida, na linha 19, temos o botão *Pesquisar*, que executa o método *pesquisar()* para buscar os colaboradores. Como podemos verificar, esse botão possui a propriedade *update* com o valor *@form*. Deste modo, informamos ao PrimeFaces para atualizar, via AJAX, o componente *form* de nossa página *pesquisa-Colaborador.xhtml* após a busca ser realizada.

O método *pesquisar()* do bean *PesquisaColaboradorDialogBean* pode ser analisado na **Listagem 6**, a partir da linha 16. Neste método, chamamos o EJB *ColaboradorService* para executar o método *buscarColaboradorPorNome()*, que realizará uma consulta pelos colaboradores cadastrados. O resultado dessa consulta é passado ao atributo *listaColaboradores* na linha 17.

Após a execução dessa consulta, o componente *<p:datatable>*, da página *pesquisaColaborador.xhtml*, é carregado com a lista de colaboradores pesquisados.

O componente **DataTable** do PrimeFaces recebe a lista de colaboradores pesquisados por meio da sua propriedade *value*. Nessa tabela, exibimos três colunas com o componente *<p:column>*. Na primeira delas, mostraremos o ID do colaborador; na segunda, o nome do colaborador; e na última coluna, um botão para o usuário selecionar um dos colaboradores da lista, como podemos observar na linha 32 da **Listagem 7**.

Quando o usuário clicar no botão *Selecionar*, o método *selecionar()* do bean *PesquisaColaboradorDialogBean* será executado, recebendo como parâmetro o colaborador selecionado, como pode ser visto na linha 22 da **Listagem 6**. Dentro desse método, a única operação que realizamos é fechar o componente de pesquisa por meio do método *closeDialog()* passando o colaborador selecionado como parâmetro para que o mesmo possa ser apresentado na página *alocacao.xhtml*, como demonstra a **Figura 4**.

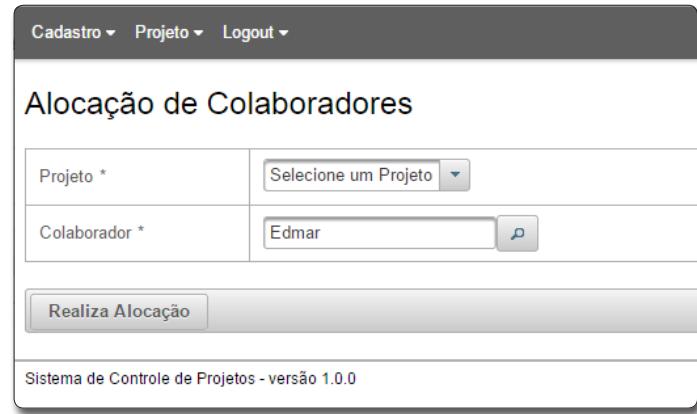


Figura 4. Tela de alocação de colaboradores com colaborador selecionado

Entretanto, apenas essa operação não é suficiente para que o colaborador seja mostrado na tela de alocação. Alguns ajustes ainda precisam ser realizados. Sendo assim, vamos voltar à linha 38 da **Listagem 4**. Nessa linha, note que inserimos dentro do botão responsável por executar o componente de pesquisa de colaboradores o componente *<p:ajax>*.

**Conhecimento
faz diferença!**



Agilidade: Acompanhamento de projetos ágeis distribuído através do Daily Meeting

Projeto
Diagrama de sequência na prática

Projeto
Como inserir padrões de projeto através de refatorações – Parte 2

SOA
Processo e levantamento de requisitos de negócios – Parte 2

Qualidade de Software
Definição, características e importância

Automação de Testes

Cuidados a serem tomados na implantação | Processo e automação de testes de Software

Aulas desta edição:
• Atividades da Gerência de Projetos – Partes 10 a 14

ISSN 1983127-7

+ de 290 vídeos para assinantes

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

Faça um upgrade em sua carreira

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

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

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

 **DEVMEDIA**

Como denota seu nome, esse componente é responsável por manipular eventos AJAX. Em nosso exemplo, vamos manipular o evento **dialogReturn**, que irá recuperar o retorno do componente de pesquisa após ser fechado. Para isso, dentro da propriedade **listener** chamamos o método **colaboradorSelecionadoEvento()** do bean **AlocacaoBean**.

O código desse método encontra-se nas linhas 27 a 30 da **Listagem 5**. O primeiro detalhe que devemos observar é que o mesmo recebe um parâmetro do tipo **SelectEvent**. Como esse é um evento de seleção, esse método irá receber como parâmetro o objeto selecionado no nosso componente de pesquisa. Dito isso, na linha 28 realizamos um *cast* do objeto selecionado pela pesquisa para o tipo **Colaborador** e na linha 29 passamos o colaborador para o objeto **Alocacao** do bean **AlocacaoBean**.

Dessa forma, quando o método **colaboradorSelecionadoEvento()** terminar de ser executado, o componente **<p:ajax>** da página *alocacao.xhtml* executará um **update** para atualizar o componente **<p:inputText>** com o nome do colaborador selecionado, como mostra a **Figura 4**.

Por fim, após o usuário selecionar um colaborador e um projeto, basta clicar no botão *Realiza Alocação*, da página *alocacao.xhtml*, para que o método **salvar()** do bean **AlocacaoBean** seja executado e a alocação do colaborador no projeto seja realizada.

Criando o controle de acesso ao sistema

A última funcionalidade que iremos desenvolver para nosso Sistema de Controle de Projetos será o controle de acesso. Com essa funcionalidade garantiremos que apenas os usuários devidamente autenticados poderão acessar as páginas do sistema.

Neste artigo, como o objetivo é focar no funcionamento de Web Filters do Java EE, vamos criar um sistema de controle de acesso simples.

O sistema de controle de acesso que iremos desenvolver não irá buscar os usuários válidos para autenticação no sistema na base de dados. Esses usuários estarão definidos em um array **usuariosValidos** no código fonte do bean **LoginBean**, responsável pelas operações de login e logout. Assim, quando ocorrer uma tentativa de login na aplicação, iremos comparar se as credenciais enviadas na tela de login estão presentes nesse array de usuários.

Dito isso, para criarmos o controle de acesso, a primeira coisa que faremos é construir a página de *login*, onde o usuário poderá informar suas credenciais (nome de usuário e senha). Portanto, crie o arquivo *login.xhtml* no diretório *src/main/webapp*. O código desta página é apresentado na **Listagem 8** e sua renderização pode ser vista na **Figura 5**.

Nesta página, ao clicar no botão *Efetuar Login*, linha 30, executamos o método **login()** do bean **LoginBean**, cujo código pode ser visto na **Listagem 9**. Este método tem a responsabilidade de verificar se as informações de nome de usuário e senha são válidas.

Ainda sobre **LoginBean**, na linha 8, injetamos o bean **Usuario**. Este é responsável por gerenciar a sessão do usuário na aplicação e seu código fonte pode ser visto na **Listagem 10**. O bean **Usuario** contém apenas dois atributos: **nome** e um valor booleano utiliza-

Listagem 8. Código da página de login *login.xhtml*.

```
01. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
02. <html xmlns="http://www.w3.org/1999/xhtml"
03.   xmlns:ui="http://java.sun.com/jsf/faces"
04.   xmlns:f="http://java.sun.com/jsf/core"
05.   xmlns:h="http://java.sun.com/jsf/html"
06.   xmlns:p="http://primefaces.org/ui">
07.
08. <h:head>
09.   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
10.  <title><ui:insert name="titulo">Controle Projeto</ui:insert></title>
11.  <h:outputStylesheet library="css" name="sistema.css" />
12. </h:head>
13.
14. <h:body>
15.
16. <h:form id="formLogin">
17.   <p:panel header="Faça o Login para Entrar no Sistema"
18.     style="width: 400px; margin: auto;">
19.     <p:messages id="messages" autoUpdate="true" closable="true" />
20.
21.     <h:panelGrid columns="2" styleClass="grid-login">
22.
23.       <p:outputLabel value="Usuário" for="usuario" />
24.       <p:inputText id="usuario" size="25" value="#{loginBean.usuario}"
25.         required="true" requiredMessage="Campo Usuario obrigatorio"/>
26.
27.       <p:outputLabel value="Senha" for="senha" />
28.       <p:password id="senha" size="25" value="#{loginBean.senha}"
29.         required="true" requiredMessage="Campo Senha obrigatorio"/>
30.       <p:outputLabel />
31.       <p:commandButton value="Efetuar Login"
32.         action="#{loginBean.login}" />
33.     </h:panelGrid>
34.   </p:panel>
35. </h:form>
36. </h:body>
37. </html>
```

A interface de usuário para o login consiste em um formulário com dois campos: "Usuário" e "Senha", ambos rotulados com asteriscos para indicar que são campos obrigatórios. Abaixo dos campos, há um botão cinza com o texto "Efetuar Login". O formulário está inserido em uma caixa com uma barra superior cinza contendo o texto "Faça o Login para Entrar no Sistema".

Figura 5. Tela de Login da aplicação

do para armazenar **true** se o usuário estiver logado ou **false** se o usuário não estiver logado. Outro detalhe é que este bean recebe a anotação **@SessionScope**. Deste modo, enquanto durar a sessão do usuário com a aplicação, esse bean estará ativo.

Voltando ao código de **LoginBean**, vamos analisar o método **login()**. Realizamos um loop **for**, entre as linhas 19 e 30, no array **usuariosValidos**, para verificar se o nome de usuário e senha informados pelo usuário na tela de login estão presentes nesse array. Caso as credenciais coincidam com algum usuário válido, passamos para o bean **Usuario** o nome do usuário logado e

setamos como **true** o atributo **usuarioLogado** (linhas 26 e 27). Por fim, na linha 29 fazemos um *redirect* para a página *Home.xhtml* mostrando para o usuário autenticado os menus da aplicação.

Listagem 9. Código do bean LoginBean.

```
01. package br.com.devmedia.controleprojeto.mb;
02.
03. /*imports omitidos*/
04.
05. @Named
06. @RequestScoped
07. public class LoginBean implements Serializable {
08.
09.     @Inject
10.     private Usuario usuarioSistema;
11.
12.     private String usuario;
13.     private String senha;
14.     private String[] usuarioValidos = {"admin:123","java:magazine"};
15.
16.     public String login() {
17.         FacesContext context = FacesContext.getCurrentInstance();
18.
19.         for (String usuarioValido : usuarioValidos) {
20.
21.             String login = usuarioValido.split(":")[0];
22.             String password = usuarioValido.split(":")[1];
23.
24.             if (login.equals(usuario) && password.equals(senha)) {
25.
26.                 this.usuarioSistema.setNome(usuario);
27.                 this.usuarioSistema.setLogado(true);
28.
29.                 return "/Home.xhtml?faces-redirect=true";
30.             }
31.         }
32.         FacesMessage mensagem = new FacesMessage("Login Incorreto: Verifique o nome de usuário ou senha utilizados!");
33.         mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
34.         context.addMessage(null, mensagem);
35.         return null;
36.     }
37.
38.     public String logout() {
39.         FacesContext.getCurrentInstance().getExternalContext().invalidateSession();
40.         return "/login.xhtml?faces-redirect=true";
41.     }
42.
43.     /*métodos getter e setter omitidos*/
44. }
```

Listagem 10. Código do bean Usuario, responsável por guardar as informações do usuário na sessão.

```
01. package br.com.devmedia.controleprojeto.mb
02. /*imports omitidos*/
03.
04. @Named
05. @SessionScoped
06. public class Usuario implements Serializable{
07.
08.     private String nome;
09.     private boolean logado;
10.
11.     public boolean isLogado() {
12.         return logado;
13.     }
14. }
```

Outra situação ocorre quando alguém tenta acessar a aplicação informando dados incorretos. Neste caso, uma mensagem de erro é apresentada na tela de login (**Figura 6**). Esse erro acontece quando as credencias informadas na tela de login não são encontradas no array **usuariosValidos**. Assim, uma mensagem de erro é apresentada para o usuário, conforme a linhas 32 a 35 de **LoginBean**.

O segundo método presente em **LoginBean** é **logout()**. Esse método é importante porque quando o usuário terminar de realizar suas tarefas na aplicação é necessário que ele encerre a sessão com o sistema. Para isso, basta acessar o menu **Logout** e depois clicar na opção **Sair**, como mostra a **Figura 7**. Ao fazer isso, o método **logout()** será executado. Como demonstrado entre as linhas 38 e 41 da **Listagem 9**, esse método possui apenas duas linhas efetivas de código: a primeira encerra a sessão do usuário e a segunda redireciona o usuário do sistema de volta para a tela de *login.xhtml*, para que um novo login possa ser realizado.

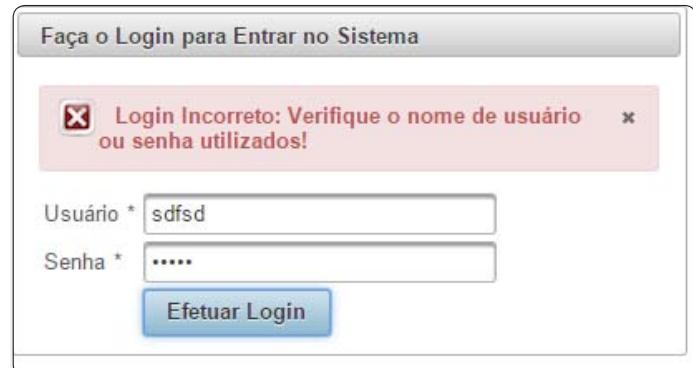


Figura 6. Tela de Login da aplicação com mensagem de erro



Figura 7. Menu de Logout da aplicação com a opção Sair

Criando o filtro de login da aplicação

Neste tópico, iremos desenvolver o controle de autorização utilizando o recurso de Web Filters, presente na Java EE 7. Com este recurso é possível verificar se o usuário autenticado tem permissão para acessar determinada página ou funcionalidade do sistema.

Com o controle de autorização, quando o usuário fizer uma requisição por uma página, teremos um filtro que irá analisar se o usuário possui permissão para acessá-la. Para isso, criamos a classe **AutenticacaoFilter**, que implementa a interface **Filter** e possui a anotação **@WebFilter**, como mostra a **Listagem 11**.

Nesse código, observamos na linha 5 que a anotação **@WebFilter** recebe como parâmetro a **String ".xhtml"**. Com essa informação,

quando o usuário tentar acessar qualquer página da aplicação, o método **doFilter()** de **AutenticacaoFilter** será executado.

Nesse método, realizamos uma validação para saber se o usuário está logado e se a página que está sendo acessada é diferente de *login.xhtml* (vide linhas 18 a 24). Se o usuário não estiver logado e a página que está sendo requisitada não for a de login, o usuário será imediatamente redirecionado para *login.xhtml*, através da chamada ao método **sendRedirect()**, da classe **HttpServletResponse**. Se o usuário estiver logado, o método **doFilter()** da classe **FilterChain**, invocado na linha 23, será executado e nossa aplicação seguirá seu fluxo normal, mostrando a página requisitada pelo usuário, já que o mesmo está autenticado e possui permissão de acesso.

Listagem 11. Código do filtro AutenticacaoFilter.

```
01. package br.com.devmedia.controleprojeto.filter;
02.
03. /* imports omitidos */
04.
05. @WebFilter("*.xhtml")
06. public class AutenticacaoFilter implements Filter {
07.
08.     @Inject
09.     private Usuario usuario;
10.
11.    @Override
12.    public void doFilter(ServletRequest req, ServletResponse res,
13.                         FilterChain chain) throws IOException, ServletException {
14.
15.        HttpServletResponse response = (HttpServletResponse) res;
16.        HttpServletRequest request = (HttpServletRequest) req;
17.
18.        if (!usuario.isLogado())
19.            && !request.getRequestURI().endsWith("/login.xhtml")) {
20.
21.            response.sendRedirect(request.getContextPath() + "/login.xhtml");
22.        } else {
23.            chain.doFilter(req, res);
24.        }
25.
26.    }
27.
28.    @Override
29.    public void destroy() {
30.
31.    }
32.
33.    @Override
34.    public void init(FilterConfig conf) throws ServletException {
35.
36.    }
37.
38. }
```

Após criarmos o filtro **AutenticacaoFilter** concluímos o desenvolvimento do controle de autenticação e autorização da aplicação. Repare que nenhuma configuração adicional foi necessária. Em versões anteriores da Java EE, no entanto, era preciso configurar o **Filter** no arquivo *web.xml*, mas com a anotação **@WebFilter** essa configuração passa a ser opcional.

Com isso, encerramos também o desenvolvimento do sistema de Controle de Projetos, utilizado nesta série de artigos para demonstrar recursos da versão 7 da plataforma Java EE.

Neste artigo optamos por utilizar o recurso de filtros da API de Servlets, porém existem várias outras formas e frameworks disponíveis para realizar a mesma tarefa. Dentro da Java EE, por exemplo, também temos o JAAS, que é uma solução bastante robusta para o desenvolvimento de controle de autenticação e autorização. Caso o leitor se interesse, a Java Magazine já publicou dois excelentes artigos relacionados (Criando aplicações web seguras – partes 1 e 2), publicados nas edições 114 e 115, que mostram em detalhes os recursos desta API.

Além do JAAS, temos o Spring Security, framework da plataforma Spring que é bastante difundido no mercado e abrange todas as necessidades para criação de um controle de acesso robusto e, principalmente, seguro. Ademais, apresenta uma curva de aprendizado pequena para quem já está familiarizado com a plataforma Spring.

Observamos ainda que a plataforma Java EE possui ótimas soluções para o desenvolvimento de aplicações corporativas não apenas pelas várias especificações presentes, que atendem a qualquer tipo de projeto, mas também pelas constantes atualizações, que mantêm a plataforma atualizada com as necessidades do mercado.

Essas características reforçam a ideia de adotar a plataforma Java EE para o desenvolvimento de novos projetos corporativos, haja vista a quantidade de recursos à disposição dos desenvolvedores, além da facilidade de manutenção dos sistemas construídos com este importante elemento do ecossistema Java.

Autor



Edmar Elias Bregagnoli
edmareliasb@gmail.com.br

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



Links:

Endereço para download do Java 7.

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

Novas especificações da plataforma Java EE 7.

https://blogs.oracle.com/arungupta/entry/java_ee_7_key_features

Endereço para download do servidor de aplicações GlassFish 4.

<https://glassfish.java.net/download.html>

Endereço com vários exemplos de componentes do PrimeFaces.

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

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



RENOVE JÁ!

Sua assinatura pode estar acabando



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

Faça a renovação de sua assinatura agora mesmo.



www.devmedia.com.br/renovacao ou (21)3382-5038

 DEV MEDIA

Storm vs. Spark: uma introdução à Fast Data

Aprenda neste artigo como processar Big Data em tempo real

Muitos analistas apontam que o Big Data não pode ser visto como a solução para todos os problemas computacionais atuais, mesmo para os que envolvem dados. Isso porque a análise de dados das aplicações modernas envolve muitas facetas, como análise de estatísticas, aprendizado de máquina e soluções em tempo real.

Do ponto de vista do desenvolvimento de software, o principal fator que permitiu a construção de aplicações Big Data foi o advento do paradigma de MapReduce. Esse paradigma, publicado em 2004, foi desenvolvido pelo Google para seu mecanismo de busca e tem como objetivo processar grandes quantidades de dados através da execução de Jobs em lote (tradução do termo em inglês *batch*), podendo assim computar um incrível volume de dados com enorme vazão (tradução do termo em inglês *throughput*). Durante a última década, o MapReduce revolucionou a TI, principalmente através do Apache Hadoop, sua implementação mais difundida. Contudo, ao longo do tempo descobriu-se que esse paradigma não é suficiente para lidar com aplicações Big Data nas quais a latência é tão ou mais importante que o throughput.

Para entender a diferença entre esses dois conceitos, imagine uma aplicação de controle de servidores responsável por enviar e-mails aos administradores de sistemas em caso de falhas. Nesses casos, ter uma grande vazão (alcançada, por exemplo, com o uso do Hadoop) representaria avisar uma quantidade enorme de usuários ao mesmo tempo. Por outro lado, uma latência pequena representaria avisar tais usuários em poucos segundos após a falha ocorrer. No cenário dessa aplicação, claramente uma latência pequena é mais importante que uma vazão grande, pois um retardamento de horas não seria interessante mesmo que a quantidade enorme de administradores seja avisada após a falha ocorrer.

Em vista disso, passou a ser natural o desenvolvimento de ferramentas que garantam baixa latência no cenário de Big Data. Ao conceito que engloba estas novas ferra-

Fique por dentro

Esse artigo é útil para estudantes e profissionais que tenham interesse em conhecer dois frameworks para Fast Data, conceito derivado de Big Data com enfoque em soluções cujo tempo de processamento seja crucial. Um caso de uso comum desta tecnologia pode ser encontrado nas redes sociais, onde um evento só tem sentido se pode ser compartilhado (acessado, visualizado, comentado) por vários usuários logo após sua publicação, ou seja, em tempo real. Dito isso, ao longo do texto analisaremos o Storm e o Spark, dois novos frameworks da Apache que já são empregados por importantes players do mercado de Big Data, como Twitter, Hortonworks, Groupon e Databricks.

mentas foi dado o nome de Fast Data (dados rápidos, em inglês). Segundo a InfoWorld, representam o próximo passo na evolução do Big Data, pois podem ser entendidos como uma contraposição à limitação do Hadoop em prover baixa latência.

Nesse contexto, tempo real pode ser entendido como um sinônimo de *streaming*, pois aplicações de Fast Data são desafiadas a examinar em poucos segundos uma onda constante de dados recebida de forma incessante, sendo esse exame fundamental para o modelo de negócio no qual estão inseridas. Esse problema vem sendo estudado em paralelo e de diferentes maneiras pelos grandes players de redes sociais, sendo criados, portanto, distintos tipos de soluções para streaming como, por exemplo: baseada no modelo publisher/subscriber, como a proposta pelo LinkedIn; em troca de mensagens, como a adotada pelo Twitter; e no uso intensivo de agregações, como a projetada pelo Facebook.

Com base nesses conceitos, este artigo apresentará dois frameworks que propõem soluções para análise em tempo real de informações Big Data: Apache Storm e Apache Spark. Ainda que possam ser usados por um grande número de linguagens, Storm e Spark – desenvolvidos respectivamente em Clojure e Scala, duas linguagens executadas pela JVM – guardam como semelhança o fato de Java ser comum a ambos. Por isso, este artigo apresenta o desenvolvimento de aplicações usando as APIs Java destas

tecnologias com foco principal na criação de uma aplicação de monitoramento em tempo real de páginas web.

Além desta, outras similaridades importantes entre estes frameworks são: baseiam-se em clusters e no uso intensivo de memória principal, e têm a baixa latência como principal requisito de projeto. Contudo, essa aparente sobreposição é apenas superficial, pois a forma de implementação e os casos de uso variam bastante de uma ferramenta para outra, como poderemos constatar nos próximos tópicos.

Apache Storm

O Storm é um framework projetado para ser escalável, tolerante a falhas, com garantia de resposta e ainda pensado para prover facilidade de configuração e operação. Esses objetivos de projeto são atingidos através de quatro abstrações básicas, analisadas a seguir e ilustradas na **Figura 1**:

- **Tuple** (em português, Tupla): representa uma mensagem que flui através da arquitetura do Storm. As tuplas carregam informações que vão sendo computadas, transformadas ou persistidas pelos componentes dessa arquitetura. Essas informações podem ser, por exemplo, um twitter postado pouco tempo antes, um documento recentemente cadastrado ou uma nova leitura do GPS feita por um sensor;
- **Spout** (em português, torneira): são elementos de código (mais propriamente classes) que estão conectados a uma fonte de dados para transformar estes dados em tuplas, que por sua vez passam a ser processadas pelo Storm;
- **Bolt** (em português, raio): são classes responsáveis unicamente por fazer computações (como transformações, cálculos, filtros, agregações, persistência) sobre as tuplas enviadas pelos spouts ou por outros bolts;
- **Topologia**: define como Spouts e Bolts são combinados em certa aplicação.

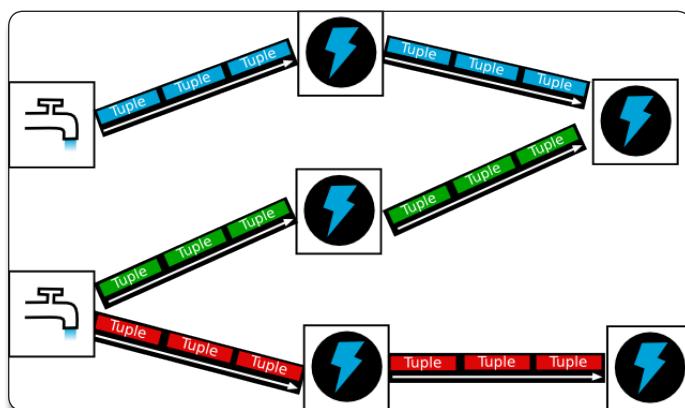


Figura 1. Visão geral de uma topologia do Storm

As abstrações apresentadas são suficientes para que possamos projetar e codificar aplicações usando o Storm. Assim, na sequência apresentaremos como desenvolver uma solução para monitorar um site em tempo real. Para isso, imagine que estamos acompanhando

a primeira página do site da Globo.com a fim de que, quando uma notícia for adicionada a essa página, possamos extrair dela o texto através de um framework como o Jsoup (vide [Links](#)) e indexá-lo em uma ferramenta de busca como o Elasticsearch (tecnologia já apresentada em edições anteriores da Java Magazine).

Para o desenvolvimento desta aplicação, assume-se que utilizaremos o Eclipse como IDE e o Apache Maven como gerenciador de pacotes. Então, após baixar e instalar o Eclipse, crie um projeto Maven. Feito isso, automaticamente, será criado um arquivo *pom.xml*, no qual podemos definir as dependências do projeto. Como apresentado na **Listagem 1**, a única dependência do Storm é a *storm-core*, versão 0.9.3. Além disso, foram adicionadas as dependências do *elasticsearch* e *jsoup*, também importantes para o exemplo.

Listagem 1. Arquivo pom.xml com a configuração das dependências para a aplicação exemplo.

```
<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</groupId>
  <artifactId>storm-spark</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>storm-spark</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.storm</groupId>
      <artifactId>storm-core</artifactId>
      <version>0.9.3</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.elasticsearch</groupId>
      <artifactId>elasticsearch</artifactId>
      <version>1.5.1</version>
    </dependency>
    <dependency>
      <groupId>org.jsoup</groupId>
      <artifactId>jsoup</artifactId>
      <version>1.8.2</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <mainClass>fully.qualified.MainClass</mainClass>
            </manifest>
          </archive>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Storm vs. Spark: uma introdução à Fast Data

Listagem 2. Spout para leitura de uma página HTML.

```
package com.devmedia.storm_spark.storm;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;

@SuppressWarnings("serial")

public class HtmlSpout extends BaseRichSpout {

    private final String page="www.globo.com";

    SpoutOutputCollector _collector;

    public void nextTuple() {

        Document doc = Jsoup.parse(page);
        _collector.emit(new Values(doc));
    }

    public void open(Map arg0, TopologyContext arg1, SpoutOutputCollector collector) {
        _collector = collector;
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("page"));
    }

    @Override
    public void ack(Object id) {

    }

    @Override
    public void fail(Object id) {

    }
}
```

Com o ambiente de desenvolvimento pronto, podemos enfim iniciar a implementação. Em um projeto Storm, a primeira classe a ser criada é um **Spout** que, como mencionado anteriormente, tem o papel de traduzir uma fonte de dados em tuplas que possam ser tratadas por uma topologia do Storm. Na aplicação exemplo, a fonte de dados será a página inicial da Globo.com e as tuplas conterão as URLs dos links presentes nessa página.

A **Listagem 2** apresenta a classe **HtmlSpout**, responsável por ler uma página HTML para obter uma sequência de links, transformar esses links em tuplas e, finalmente, emitir – palavra usada pela documentação, com o sentido de transmitir em *broadcast* – essa tupla para o resto da topologia.

Neste código podemos notar que **HtmlSpout** estende **BaseRichSpout**, a forma mais simples de se desenvolver um spout. Os métodos de **HtmlSpout**, herdados de **BaseRichSpout**, representam o ciclo de vida de um Spout no cluster do Storm:

- **open()**: executado quando uma instância desse spout é criado. Pode ser utilizado para iniciar conexões e variáveis que serão empregadas durante o ciclo de vida desse objeto;
- **nextTuple()**: método que faz efetivamente o trabalho do spout, emitindo uma tupla para a topologia;
- **ack()**: confirma que a tupla enviada pelo Spout foi recebida por um Bolt e vai ser processada;
- **fail()**: recebe uma mensagem caso haja alguma falha no envio de uma tupla;
- **declareOutputFields()**: método que declara quais campos conterão a tupla enviada à topologia.

É importante que o desenvolvedor saiba que existem diferentes tipos de spout, como: **BasePartitionedTransactionalSpout**, que permite implementar a execução com transações, facilitando o controle da consistência através de commits – como os realizados em bancos de dados; e **BasePartitionedSpout**, que permite dividir o processamento de uma mesma entrada em distintos spouts.

Cada tipo de spout deve ser usado para uma situação específica, contudo, todas estas opções compartilham a mesma estrutura básica do **BaseRichSpout**, explicado anteriormente.

Como em nosso exemplo existe apenas uma fonte de dados, a topologia para esse exemplo vai conter apenas um spout. Entretanto, vamos ter três bolts, a saber:

- **RecuperaHtmlBolt**: responsável por recuperar a página do link recebido na tupla e comparar a versão da página recuperada com as páginas já processadas para verificar se ela é repetida;
- **AnalisaHtmlBolt**: responsável por categorizar a página de acordo com a análise do seu texto;
- **IndexaHtmlBolt**: responsável por indexar a página no mecanismo de busca.

A **Listagem 3** apresenta o código de **RecuperaHtmlBolt**. Como podemos verificar, ele estende **BaseRichBolt**, interface que declara o método **declareOutputFields()**, que é responsável por declarar quais campos conterão a tupla enviada à topologia, e o método **execute()**, que realiza o processamento definido para esse bolt. No caso de **RecuperarHtmlBolt**, **execute()** utilizará o Jsoup para recuperar a página relacionada ao link recebido na tupla (note que a tupla é um dos parâmetros do método), calcular o hash do texto dessa página e comparar com o hash retornado pelo Elasticsearch. Caso haja uma diferença de hash ou se a página não for encontrada no Elasticsearch, **RecuperaHtmlBolt** irá emitir essa página para os próximos bolts, a fim de que a mesma seja processada. Caso contrário, a execução dessa tupla na topologia é encerrada.

O próximo bolt que devemos desenvolver é o **AnalisaHtmlBolt**. Este irá receber a página emitida por **RecuperaHtmlBolt**, extrair informações importantes e criar um documento que será enviado ao próximo bolt da topologia. Como pode-se verificar na **Listagem 4**, o texto da página recebida pelo bolt será analisado a fim de contar as palavras mais frequentes. Em seguida, a palavra mais frequente será escolhida para representar a categoria

Listagem 3. Bolt para recuperação de uma página HTML – RecuperaHtmlBolt.

```
package com.devmedia.storm_spark.storm;

import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseBasicBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

@SuppressWarnings("serial")

public class RecuperaHtmlBolt extends BaseBasicBolt {

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        Document doc = (Document) tuple.getValue(0);
        if(checkHashCode(doc.location())==doc.hashCode()){
            Elements links = doc.select("a[href*=event-details]");
            for(Element link: links){
                collector.emit(new Values(link));
            }
        }
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("link"));
    }

    private int checkHashCode(String url){
        return 0;
    }
}
```

da página. Com esses dados é criado um documento JSON para ser armazenado no Elasticsearch. Além disso, metadados como data da última atualização, URL, tamanho em bytes e número de caracteres também terão campos específicos no JSON, assim como o hash do texto da página. Após essa análise, uma tupla contendo um documento com todos esses campos é emitida para a topologia.

É importante ressaltar que esse exemplo de análise foi desenvolvido apenas para ilustrar a capacidade do Storm. Em uma aplicação real, muito mais poderia ser extraído dessa página.

Finalmente, a **Listagem 5** apresenta o **IndexaHtmlBolt**, classe responsável pela indexação do resultado da análise de uma página no Elasticsearch, ou seja, este bolt recebe os dados emitidos por **AnalisaHtmlBolt**, os transforma em um documento JSON e os envia para o índice do Elasticsearch.

A última classe a ser desenvolvida para esse exemplo é a responsável pela criação da topologia que envolve o spout e os três bolts. Como apresentado na **Listagem 6**, uma topologia nada mais é do que uma classe principal que instancia cada uma das classes apresentadas anteriormente.

Como em todo ambiente com alto poder de processamento o paralelismo é uma característica fundamental, o Storm possibilita ao desenvolvedor informar o número de *threads* que cada spout e bolt precisarão. O cluster de Storm então irá iniciar a sua execução com esse número de *threads* e, em caso de alguma necessidade da aplicação, poderá aumentar ou diminuir essa quantidade.

Listagem 4. Bolt para análise de uma página HTML – AnalisaHtmlBolt.

```
package com.devmedia.storm_spark.storm;

import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseBasicBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

@SuppressWarnings("serial")
public class AnalisaHtmlBolt extends BaseBasicBolt {

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String link = ((Element) tuple.getValue(0)).attr("abs:href");
        Document doc = Jsoup.parse(link);
        try {
            XContentBuilder json = jsonBuilder().startObject();
            json.field("page").value(doc.text());
            json.field("date").value(new Date());
            json.field("category").value(doc.text());
            json.field("hash").value(json.hashCode());
            json.endObject();

            collector.emit(new Values(json));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("json"));
    }
}
```

```
}
```

```
private String findCategory(String text){
    Map<String,Integer> words_count = new HashMap<String,Integer>();
    String[] words = text.split("\\s+");

    for(int i=0;i<words.length;i++) {
        String s = words[i];
        if(words_count.keySet().contains(s)) {
            Integer count = words_count.get(s) + 1;
            words_count.put(s, count);
        } else{
            words_count.put(s, 1);
        }
    }
    Integer frequency = null;
    String mostFrequent = null;
    for(String s : words_count.keySet()) {
        Integer i = words_count.get(s);
        if(frequency == null)
            frequency = i;
        if(i > frequency) {
            frequency = i;
            mostFrequent = s;
        }
    }
    return mostFrequent;
}
```

Storm vs. Spark: uma introdução à Fast Data

Listagem 5. Bolt para indexação de uma página HTML.

```
package com.devmedia.storm_spark.storm;

@SuppressWarnings("serial")
public class IndexaHtmlBolt extends BaseBasicBolt {

    private static Client client;
    private static TransportClient transportClient;

    private final static String server = "xxx.xxx.xxx.xx";
    private final static String port = "9300";
    private final static String cluster = "elasticsearch";

    public static synchronized Client getClient() throws Exception {
        if (client == null) {
            Settings settings = ImmutableSettings.settingsBuilder()
                .put("cluster.name", cluster).build();
            transportClient = new TransportClient(settings);
            client = transportClient
                .addTransportAddress(new InetSocketAddress(server,
                    Integer.parseInt(port)));
        }
        return client;
    }

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        XContentBuilder json = (XContentBuilder) tuple.getValue(0);

        try {
            getClient().prepareIndex().setSource(json).execute();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            client.close();
            transportClient.close();
        }
    }

    public void declareOutputFields(OutputFieldsDeclarer arg0) {
    }
}
```

Listagem 6. Topologia para o exemplo de monitoramento de páginas.

```
package com.devmedia.storm_spark.storm;

public class MonitoraPaginaTopology {

    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("htmlSpout", new HtmlSpout(), 5);
        builder.setBolt("recuperaHtmlBolt", new RecuperaHtmlBolt(),
            8).shuffleGrouping("spout");
        builder.setBolt("analisaHtmlBolt", new AnalisaHtmlBolt(),
            12).fieldsGrouping("split", new Fields("word"));
        builder.setBolt("indexaHtmlBolt", new IndexaHtmlBolt(),
            8).shuffleGrouping("spout");
        Config conf = new Config();
        conf.setDebug(true);

        if (args != null && args.length > 0) {
            conf.setNumWorkers(3);
            StormSubmitter.submitTopologyWithProgressBar
                ("JMTopology", conf,
                builder.createTopology());
        } else {
            conf.setMaxTaskParallelism(3);
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("word-count", conf,
                builder.createTopology());
            Thread.sleep(10000);
            cluster.shutdown();
        }
    }
}
```

Neste exemplo, a topologia será definida da seguinte forma:

- **HtmlSpout:** chamado htmlSpout, com o valor 5 na “dica” de paralelismo;
- **RecuperaHtmlBolt:** chamado recuperaHtmlBolt, com o valor 8 na “dica” de paralelismo;
- **AnalisaHtmlBolt:** chamado analisaHtmlBolt, com o valor “12” na dica de paralelismo;
- **IndexaHtmlBolt:** chamado indexaHtmlBolt, com o valor “8” na dica de paralelismo.

Antes de passar para a execução da aplicação é importante entender como um cluster Storm funciona. Como ilustrado na **Figura 2**, o cluster Storm trabalha de acordo com o paradigma Gerente/Trabalhadores (ou em uma nomenclatura ultrapassada, também chamado de Mestre/Escravo). O Gerente Storm é chamado de Nimbus e tem como responsabilidade organizar a execução da aplicação e comunicar-se com o Zookeeper (que pode ter várias instâncias), outro framework da Apache (veja mais informações em [Links](#)) que tem a função de registrar e gerenciar os Trabalhadores, que no Storm são chamados de Supervisores. São nos Supervisores que os Spouts e Bolts são executados e, conforme a configuração de cada topologia, trocam tuplas entre si.

Com base nisso, para executar a aplicação é necessário um cluster Storm em execução. Esse cluster, que pode ser instalado em uma máquina local ou espalhado em muitas máquinas, irá gerenciar o ciclo de vida da aplicação Storm. Além disso, vale ressaltar que um mesmo cluster pode ter várias topologias ativas – em execução –, onde os spouts estarão recebendo

informações exteriores e as bombeando para dentro da arquitetura do cluster.

Para instalar um cluster no seu ambiente de desenvolvimento, faça o download da última versão do Storm (veja o endereço na seção **Links**) e execute o comando `storm` da pasta `bin`.

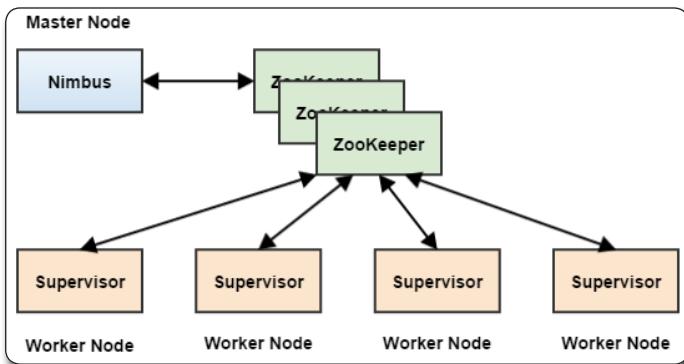


Figura 2. Visão geral de um cluster do Storm

Com o cluster já disponível, deve-se empacotar a aplicação usando o Maven. Para tal, execute `mvn assembly:assembly` no mesmo diretório do arquivo `pom.xml` do projeto. Este comando irá criar uma pasta de nome `target` que conterá o código empacotado em um arquivo JAR chamado `storm-spark0.0.1-SNAPSHOT.jar`. Feito isso, esse arquivo deve ser enviado ao cluster através do comando: `storm jar storm-spark0.0.1-SNAPSHOT.jar com.devmedia.MonitoraPaginaTopology`.

Para parar a execução dessa topologia, execute: `storm kill {stormname}`. No caso da aplicação desenvolvida: `storm kill {JMTopology}`.

Apache Spark

Assim como o Storm, o Apache Spark também é uma ferramenta voltada para Fast Data, compartilhando o foco em baixa latência nas respostas e oferecendo facilidade de uso. No entanto, de forma distinta, não possui um conjunto de abstrações para definir toda a arquitetura da aplicação. Como diferencial, o Spark possui interessantes ferramentas de análise estatística e aprendizado de máquina pré-implementadas.

Com base na arquitetura do Spark (**Figura 3**), cada aplicação deverá instanciar um contexto do Spark (Spark Context) que contém as informações necessárias para acessar o cluster onde a mesma será executada. O Spark Context comunica-se com o Cluster Manager, um componente responsável por distribuir a execução de jobs em Workers disponíveis nos nós do cluster. A principal vantagem desta arquitetura é que o processamento dos dados é feito todo em uma memória compartilhada entre os nós, sendo que o acesso ao disco é evitado sempre que possível.

Além da arquitetura geral, é importante conhecer os seguintes conceitos do Spark:

- **Resilient Distributed Datasets (RDD):** abstraem um conjunto de objetos distribuídos no cluster, geralmente executados em memória principal. Esses objetos podem estar armazenados em

sistemas de arquivo tradicional, no HDFS e em alguns bancos de dados NoSQL como Cassandra e HBase;

- **Operações:** representam transformações (como agrupamentos, filtros e mapeamentos entre os dados) ou ações (como contagens e persistências);

- **Contexto Spark (Spark Context):** como ilustrado anteriormente, representa o principal ponto de entrada ao cluster Spark para uma aplicação que o está utilizando;

- **Discretized Stream (DStream):** abstrai conjuntos de RDDs que recebem um fluxo contínuo de dados. O DStream é o mais importante conceito quando se utiliza o Spark para streaming, pois facilita o tratamento dos dados que chegam de distintas fontes através de mecanismos simples, consistentes e tolerantes a falhas.

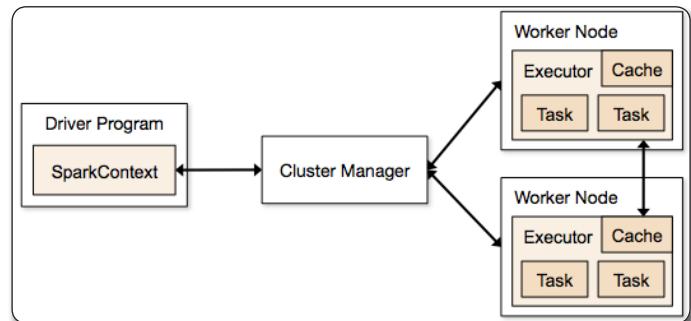


Figura 3. Visão geral da arquitetura do Spark

Conhecidos estes conceitos, para o desenvolvimento da aplicação exemplo (igual à apresentada anteriormente), também será utilizado o Eclipse como IDE e o Apache Maven como gerenciador de pacotes. Além disso, usaremos o Spark versão 1.4.0. Assim, crie um novo projeto Maven no Eclipse e altere o `pom.xml` para que fique igual ao da **Listagem 7**. Novamente, utilizaremos o Elasticsearch e o Jsoup apenas para a implementação do exemplo, e a dependência `spark-streaming_2.10` será responsável por adicionar os pacotes do Spark ao projeto.

A principal diferença entre a implementação em Storm e em Spark é que em vez de uma página ser processada através do fluxo de uma tupla por meio de diferentes etapas (spout e bolts), no Spark são definidas diversas operações (transformações e ações) sobre os dados. Isto é, o Spark não oferece uma abstração de alto nível como a topologia do Storm. Cada operação será uma atividade independente no cluster, cabendo ao programador organizar o fluxo desejado.

Assim como ocorreu na implementação da aplicação Storm, a primeira tarefa deste novo projeto é desenvolver uma classe responsável por recuperar uma página e extrair seus links. Na **Listagem 8** nota-se que a classe `RecuperaHtml` limita-se a recuperar estes links (no exemplo, da Globo.com) e enviá-los por um socket, que será explicado a seguir.

O próximo passo apresentado na **Listagem 9** é usar o Jsoup para recuperar a página relacionada a cada link, calcular o hash do texto dessa página e comparar com o hash retornado pelo

Storm vs. Spark: uma introdução à Fast Data

Listagem 7. Arquivo pom.xml com a configuração de dependências Maven para a aplicação exemplo com Spark

```
<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</groupId>
  <artifactId>storm-spark</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>storm-spark</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.elasticsearch</groupId>
      <artifactId>elasticsearch</artifactId>
      <version>1.5.1</version>
    </dependency>
    <dependency>
      <groupId>org.jsoup</groupId>
      <artifactId>jsoup</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-streaming_2.10</artifactId>
      <version>1.4</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <mainClass>fully.qualified.MainClass</mainClass>
            </manifest>
          </archive>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Listagem 8. Classe para extrair os links da página – RecuperaHtml.

```
package com.devmedia.storm_spark.spark;

public class RecuperaHtml {

    private final String page = "www.globo.com";

    public void main(String[] args) {
        Document doc = Jsoup.parse(page);
        try {
            Socket socket = new Socket("localhost", 9999);
            if(checkHashCode(doc.location())==doc.hashCode()){
                Elements links = doc.select("a[href*=event-details]");
                OutputStream outstream = socket.getOutputStream();
                PrintWriter out = new PrintWriter(outstream);
                for(Element link: links){
                    out.print(link.attr("abs:href"));
                }
            }
            socket.close();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private int checkHashCode(String url){
        return 0;
    }
}
```

Elasticsearch. Como apresentado nas linhas iniciais da classe **MonitoraPagina**, deve-se criar um **SparkContext** ao instanciar as classes **SparkConf** e **JavaStreamContext**. **SparkConf** é meramente uma classe auxiliar, que unifica as configurações que serão usadas para conexão ao cluster Spark.

Já a classe **JavaStreamContext** é a implementação do **SparkContext** para aplicações que tratam streams e são desenvolvidas em Java, como a aplicação apresentada ao longo do artigo. No caso da **Listagem 9**, as configurações informadas ao **SparkConf** são o nome da aplicação (no exemplo, “MonitoraPagina”) e de quanto em quanto tempo serão buscados novos dados, com o objetivo de eliminar informações antigas automaticamente em um ambiente de streaming. No caso da aplicação exemplo, a cada segundo, novos dados serão lidos e as informações antigas descartadas.

Ainda nessa listagem, cria-se um **DStream** que espera informações vindas através de um socket na porta 9999. Como explicado anteriormente, o **DStream** é uma abstração para um fluxo contínuo de dados, facilitando o gerenciamento de falhas e de consistência (por exemplo, duplicação, valores inválidos) no recebimento desses dados. Um exemplo interessante de **DStream** é o implementado para o Twitter, que encapsula as atividades relacionadas à conexão, ao monitoramento e ao tratamento de informações vindas da rede social.

Ainda na classe **MonitoraPagina** apresentam-se os códigos para extrair informações de contexto da página e criar um documento JSON com informações como data da última atualização, URL, tamanho em bytes, número de caracteres e o hashcode do texto da página. Ao final, temos o código responsável pela indexação do resultado da análise de uma página, isto é, o código que

envia os dados da página ao índice do Elasticsearch.

Antes de executar o nosso código é interessante ter uma visão geral de como o cluster Spark funciona. Como podemos verificar na **Figura 4**, este cluster também trabalha com o paradigma Gerente/Trabalhador. Assim, para cada cluster haverá um Cluster Manager, que pode ser chamado de Spark Master (Mestre), e diversos Spark Workers (Trabalhadores). Em resumo, o Gerente é responsável por registrar os Trabalhadores, garantir que completem seu trabalho e, no caso de uma falha, redirecionar esse trabalho para outro trabalhador. O trabalhador, por sua vez, é responsável por realizar os processamentos da aplicação. Neste caso, podemos ver que os mesmos vão manipular os RDDs e executar as transformações, como apresentado anteriormente.

Para executar a aplicação Spark o primeiro passo é baixar o cluster compilado (veja

o endereço na seção **Links**). Na página de downloads, escolha o release do Spark (atualmente na versão 1.4.0) e o tipo do pacote. Como o Spark pode ser integrado ao Hadoop, o pacote define com qual versão do Hadoop será feita a integração.

Visto que neste artigo não temos nada relacionado ao Hadoop, pode ser escolhida qualquer versão. Ainda assim, recomendamos que utilizem a última versão (*Hadoop 2.6 or later*). Feito isso, descompacte o arquivo baixado e execute: *bin\spark-class.cmd*

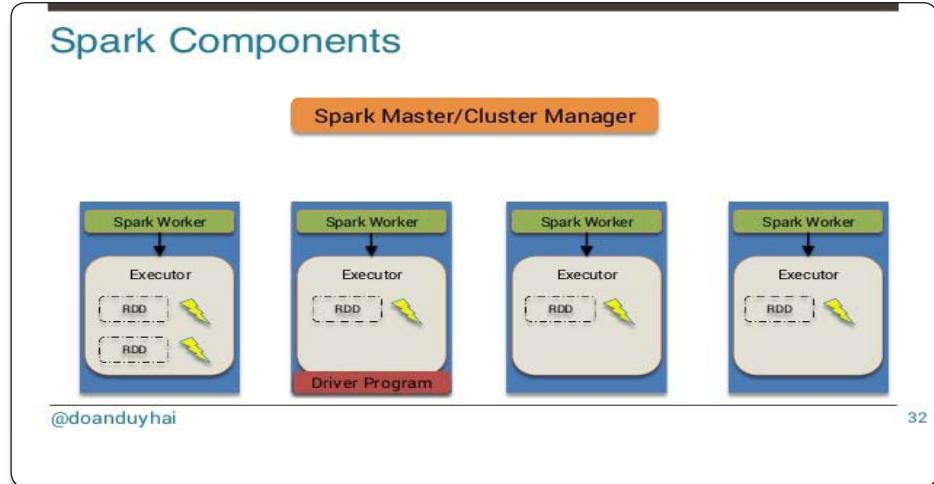


Figura 4. Master e workers no cluster Spark

Listagem 9. Código da classe MonitoraPagina.

```
package com.devmedia.storm_spark.spark;

public class MonitoraPagina {

    private static Client client;
    private static TransportClient transportClient;

    private final static String server = "198.23.188.82";
    private final static String port = "9301";
    private final static String cluster = "elasticsearch";

    public static void main(String[] args) {

        SparkConf conf = new SparkConf().setMaster("local").setAppName(
            "MonitoraPagina");
        JavaStreamingContext jssc = new JavaStreamingContext(conf,
            Durations.seconds(1));
        JavaReceiverInputDStream<String> pages = jssc.socketTextStream(
            "localhost", 9999);

        pages.map(new Function<String, String>(){

            public String call(String arg0) throws Exception {
                Document doc = Jsoup.parse(arg0);

                try {
                    XContentBuilder json = jsonBuilder().startObject();
                    json.field("page").value(doc.text());
                    json.field("date").value(new Date());
                    json.field("category").value(doc.text());
                    json.field("hash").value(json.hashCode());
                    json.endObject();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                return null;
            }
        });
    }

    public String call(String arg0, String arg1) throws Exception {
        try {
            getClient().prepareIndex().setSource(arg0).execute();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            client.close();
            transportClient.close();
        }
        return null;
    }

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        XContentBuilder json = (XContentBuilder) tuple.getValue(0);
        try {
            getClient().prepareIndex().setSource(json).execute();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            client.close();
            transportClient.close();
        }
    }
}
```

Storm vs. Spark: uma introdução à Fast Data

`org.apache.spark.deploy.master.Master`, o que irá iniciar o mestre do cluster. Na sequência, acesse `http://localhost:8080/` (vide **Figura 5**) e veja qual é o endereço de rede desse mestre (geralmente `spark://<IP>:7077`). Com o mestre rodando, execute `bin\spark-class .cmd org.apache.spark.deploy.worker.Worker spark://<IP>:7077`, substituindo `<IP>` pelo endereço do mestre. Ao recarregar a página acessando o endereço `http://localhost:8080/`, você verá que o primeiro worker foi adicionado.

The screenshot shows the Spark Master interface at `localhost:8080`. It displays the following information:

- Spark 1.4.0** logo and title "Spark Master at spark://192.168.56.1:7077".
- Workers:** One worker listed: "worker-20150620175534-192.168.56.1-53247" (Address: 192.168.56.1:53247, State: ALIVE, Cores: 2 (0 Used), Memory: 2.9 GB (0.0 B Used)).
- Running Applications:** No applications currently running.
- Completed Applications:** No completed applications listed.

Figura 5. Painel de controle do Spark

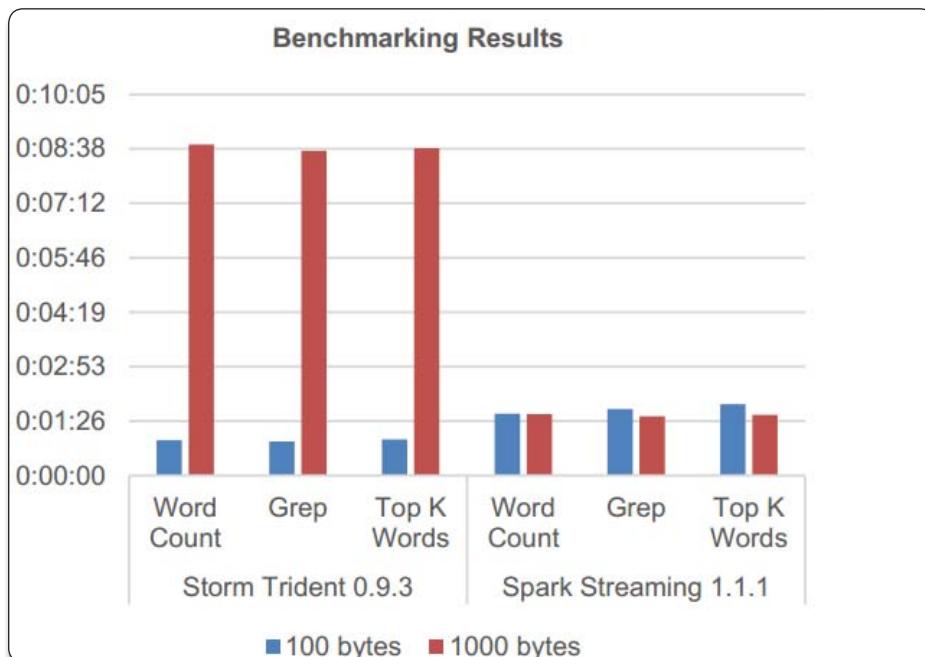


Figura 6. Comparação entre Storm e Spark

Comparação entre Storm e Spark

Em uma aplicação em tempo real, o tempo de resposta é obviamente o fator mais importante a ser considerado. Pensando nisso, foram estabelecidas algumas comparações com base no tempo de resposta de algumas aplicações desenvolvidas com Storm e Spark. O gráfico da **Figura 6** traz o resultado a partir da execução de três funções: contagem de palavras (word count); busca por uma palavra (grep); e encontrar as palavras mais frequentes

(top k words). A primeira metade expõe os tempos do Storm e a segunda os tempos do Spark, também dividido em requisições pequenas e grandes.

Ao analisar este resultado é possível constatar que o Storm é cerca de 40% mais rápido para tuplas pequenas (do tamanho de um tweet, aproximadamente). Contudo, quando o tamanho da tupla aumenta, o Spark tem o melhor desempenho, mantendo praticamente o mesmo tempo de processamento.

Apesar da importância do tempo de resposta para a comparação dos frameworks, também é importante avaliar outras características. Por exemplo: um arquiteto deve levar em consideração questões como as ilustradas pela **Tabela 1** para tomar a decisão entre um e outro. Note que esta tabela sumariza características como as linguagens de programação suportadas, modelo de tratamento de streaming, garantia de entrega e ordem de grandeza da latência esperada (veja a seção **Links** para mais informações).

Ainda que o Storm suporte mais linguagens de programação, vale frisar que a tendência natural é que ambos suportem o mesmo conjunto de linguagens, já que se tratam de ferramentas de código aberto, com uma comunidade que cria suas próprias ferramentas. Com relação ao modelo de tratamento de streaming, podemos constatar que o Storm analisa os dados conforme os mesmos chegam a sua arquitetura, enquanto o Spark usa um modelo chamado de micro-batching, cuja ideia é empacotar uma pequena quantidade de dados antes de processá-los. O objetivo disso é otimizar o uso de recursos e diminuir o tráfego na rede interna ao cluster. Porém, causa certo atraso quando existem mensagens muito pequenas ou muito frequentes (o que explica em partes os resultados da **Figura 6**).

Sobre a garantia de processamento dos dados recebidos pelas arquiteturas, ou seja, a certeza de que quando certa informação chega à arquitetura ela será processada, existem três modelos: processar pelo menos uma vez, no máximo uma vez ou exatamente uma vez. No Spark, os dados são sempre processados pelo menos uma vez, ou seja, o Spark garante que não haverá dados perdidos.

O Storm, por outro lado, permite ao desenvolvedor escolher como esses dados serão processados. A garantia de processamento pelo menos uma vez é a mais barata do ponto de vista computacional, pois não necessita de supervisão entre as execuções. Por sua vez, garantir que seja executado apenas uma vez exige que a arquitetura do Storm controle os vários componentes em execução.

Finalmente, deve-se ter em mente que o Spark foi projetado para que cada uma das tarefas seja executada em milissegundos, enquanto o Storm foi pensado para executar em segundos.

O Storm oferece ainda uma interessante abstração para o desenvolvimento de aplicações complexas, pois dispõe de uma abstração que ajuda o arquiteto a projetar o fluxo de análises. Por outro lado, o Spark é muito interessante pelo fato do tratamento dos dados ocorrer na memória primária, sem que seja necessário acesso ao disco, o que possibilita a esse framework o alcance de velocidades muito maiores em tarefas pontuais, como se confirmou na análise de tempo da aplicação.

É válido frisar que o mercado de Big Data está extremamente aquecido e que as ferramentas de desenvolvimento (como Storm e Spark) continuam evoluindo muito rapidamente. Por isso, os resultados apresentados neste tópico servem apenas como um parâmetro de comparação para as versões de ambos os frameworks que foram utilizadas. Deste modo, no momento de escolher a tecnologia para um projeto, seus responsáveis devem refazer essa pesquisa a fim de garantir a melhor escolha.

Apesar do Storm e do Spark terem o mesmo objetivo, cada um dos frameworks apresenta características que os fazem mais interessantes para certos tipos de aplicações. Um exemplo disso é o tamanho médio da informação analisada pela aplicação. O Storm lida melhor com dados pequenos, enquanto o Spark se sobressai com dados maiores.

Vale acrescentar que tanto o Storm quanto o Spark são apoiados pela Fundação Apache. Portanto, pode-se confiar na qualidade dos projetos e, principalmente, assegurar-se que os desenvolvedores serão apoiados pela comunidade e ter acesso ao código fonte.

Como em toda decisão arquitetural, antes de optar por um dos frameworks, deve-se considerar primeiramente o objetivo do software que está sendo desenvolvido. Ademais, apesar de terem características parecidas, esses frameworks podem eventualmente cooperar em uma arquitetura complexa. Por exemplo, uma aplicação pode se beneficiar dos componentes de streaming do Storm, mas utilizar as ferramentas de estatística e de aprendizado de máquina do Spark.

Por fim, deve-se ressaltar que existem outros frameworks com propostas similares, dentre os quais podemos citar o Apache Kafka, desenvolvido e usado pelo LinkedIn, e o Apache Flume, usado pelo SoundCloud.

Para os leitores interessados em utilizar o Storm ou o Spark como parte de uma solução, fica a sugestão de conhecer a arquitetura Lambda – proposta por Nathan Marz (também criador do Storm) –, que organiza em uma única visão o processamento em tempo real com o processamento batch.

	Storm	Spark
Linguagens de programação suportadas	Java, Clojure, Scala, Python e Ruby	Java, Scala e Python
Modelo de tratamento de streaming	Streaming de eventos	Micro-batching
Garantia de processamento	Pelo menos uma vez / No máximo uma vez	Exatamente uma vez
Ordem de grandeza da latência esperada	Milissegundos	Segundos

Tabela 1. Comparação entre Storm e Spark

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 e startupper na 33!.

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 é professor na Universidade Anhembi Morumbi.

Links:

Endereço para download do Storm.

<https://github.com/apache/storm/releases>

Endereço para download do Spark.

<http://spark.apache.org/downloads.html>

GitHub do Apache Storm.

<https://github.com/apache/storm>

GitHub do Apache Spark.

<https://github.com/apache/spark>

GitHub do autor, com os códigos apresentados no artigo.

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

Página do projeto Apache Zookeeper.

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

Comparação entre Storm e Spark.

http://www.cs.utoronto.ca/~patricia/docs/Analysis_of_Real_Time_Stream_Processing_Systems_Considering_Latency.pdf

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Relatórios avançados com Hibernate, JasperReports e PrimeFaces – Parte 1

Desenvolva relatórios com virtualização e paginação utilizando Hibernate e JasperReports

ESTE ARTIGO FAZ PARTE DE UM CURSO

A tomada de decisões nas empresas de hoje está completamente atrelada à disponibilização de relatórios por seus sistemas. No entanto, como a linguagem Java não possui suporte nativo a tal recurso, precisamos de mecanismos externos que supram essa carência.

Surgiu assim o JasperReports, uma biblioteca open source escrita em Java focada exatamente na simplificação da construção de relatórios para aplicações desenvolvidas com a mesma linguagem. Apesar de seu sucesso hoje em dia, as primeiras versões do JasperReports eram bastante improdutivas, pois exigiam o domínio do seu DTD (*Document Type Definition*), um documento descritor XML utilizado pelo JasperReports para gerar os relatórios. O desenvolvedor tinha que escrever seu código num editor de textos qualquer, pois não existia uma ferramenta que abstráisse esse trabalho. Na **Listagem 1** apresentamos um exemplo de como declarar um label no DTD. Como pode ser observado, não é nada trivial e de fácil leitura.

O intervalo entre as linhas 1 e 8 define um texto estático, um label. Na linha 2 os atributos **x** e **y** configuram as coordenadas do posicionamento do elemento no relatório e **width** e **height** definem sua área delimitadora. Em seguida, na linha 3, definimos a margem esquerda

Fique por dentro

Segundo estudos elaborados pela IDC, uma empresa de consultoria para indústrias de TI, a quantidade de informação dobra a cada dois anos no mundo e com esse crescimento, os sistemas são submetidos a processar quantidades cada vez maiores de dados. Para lidar com essa questão, foram criadas diversas tecnologias/técnicas focadas no aumento da eficiência do gerenciamento de memória. Com base nisso, abordaremos neste artigo algumas dessas técnicas, como a virtualização e a paginação da memória heap em aplicações escritas em Java, com ênfase na geração de relatórios que acessam grandes quantidades de dados. Dessa forma o leitor terá uma referência do que fazer ao se deparar com situações semelhantes.

interna do texto dentro de sua região. Das linhas 4 a 6 especificamos a formatação do elemento de texto, que no caso será exibido em negrito. Finalmente, na linha 7 adicionamos o conteúdo que será apresentado no relatório, pela label: o texto “Descrição”.

Listagem 1. Exemplo de código do arquivo DTD.

```
01 <staticText>
02   <reportElement x="0" y="0" width="117" height="35"/>
03   <box leftPadding="2"/>
04     <textElement verticalAlignment="Middle">
05       <font isBold="true"/>
06     </textElement>
07     <text><![CDATA[Descrição]]></text>
08   </staticText>
```

Constatada essa dificuldade, o iReport foi criado. Como já esperado, seu objetivo é facilitar o desenvolvimento de relatórios, tornando transparente o conhecimento sobre como estruturar o arquivo DTD, o que se dá com a automatização do processo de escrita desse arquivo.

Com o iReport podemos projetar relatórios através de uma interface gráfica que disponibiliza diversas opções para criar layouts complexos e de maneira simples. Esta ferramenta possibilita também utilizar recursos avançados do JasperReports que podem auxiliar em situações específicas como, por exemplo, a virtualização da memória heap, que permite fazer swap em disco. Esta técnica faz com que pequenas porções da memória heap sejam escritas diretamente no disco, tornando possível seu gerenciamento. A virtualização é recomendada quando processamos uma grande quantidade de informações e temos pouca memória heap disponível, o que pode ocasionar erros do tipo **OutOfMemoryException**.

Outro recurso muito interessante que pode ser implementado com o auxílio do JasperReports e que ajuda no gerenciamento da memória é a paginação. No entanto, a paginação não é um recurso nativo desta biblioteca, tendo que ser implementada à parte. Veremos como fazer isso mais adiante.

Uma vez que o nosso foco será a demonstração de recursos mais avançados do iReport e do JasperReports, este artigo pressupõe que o leitor já tenha noções básicas destas ferramentas. Caso queira adquirir tais conhecimentos, recomendamos a leitura do artigo “Gerando Relatórios com JasperReports”, publicado na Edição 19 da Easy Java Magazine.

Dito isso, para demonstrar a virtualização e a técnica de paginação, apresentaremos um exemplo desses recursos na prática a partir da construção de um pequeno simulador genérico de perguntas e respostas que utiliza o Tomcat como container web, o Hibernate para acessar a base de dados MySQL e o PrimeFaces para construir a interface gráfica.

O PrimeFaces

O PrimeFaces é considerado a biblioteca mais completa de componentes ricos para aplicações web ou mobile escritas em Java. Devendo ser utilizada junto com o JavaServer Faces (JSF), foi uma das primeiras soluções RIA a estar totalmente convertida para funcionar com o JSF na versão 2.0.

Lançada em 2008 pela PrimeTek, o PrimeFaces traz como uma das suas principais vantagens uma comunidade bastante ativa, onde problemas comuns são relatados e solucionados de forma interativa. A biblioteca tem seu código aberto e é muito bem documentada, disponibilizando versões gratuitas e com várias opções de componentes de interface.

As aplicações RIA têm como particularidade oferecer recursos ricos de interface, como formas de arrastar e soltar objetos na tela, atualização de dados sem recarregamento da página, animações e um visual diferenciado. Devido a isso, construir uma solução RIA era extremamente complexo, pois além de exigir conhecimentos em diferentes tecnologias, tornava o desenvolvimento bastante

improdutivo, ao obrigar o desenvolvedor a implementar todas as rotinas necessárias para o seu funcionamento.

A proposta do PrimeFaces é auxiliar na resolução dos problemas proporcionados pela utilização dos componentes ricos de interface, resolvendo questões como a necessidade de adquirir conhecimentos em várias linguagens de programação e a alta complexidade na construção das rotinas de funcionamento de tais componentes. Essa facilidade é provida pelo framework através da disponibilização de componentes que englobam as funcionalidades dos recursos RIA, de modo que o desenvolvedor não precisa se preocupar com detalhes da sua implementação. Com isso, ganhamos na produtividade, pois ao utilizá-lo diminuímos o tempo no desenvolvimento das funcionalidades e melhoramos também a qualidade do software, por conta da adoção de códigos padronizados e já testados pela equipe do PrimeFaces.

Com base nisso, demonstraremos como utilizar alguns dos seus componentes no decorrer da construção do simulador aqui proposto e, à medida que adicionarmos tais componentes, notaremos a facilidade com que ele lida com os recursos característicos do RIA comentados anteriormente.

O Hibernate

Ao utilizar bancos de dados relacionais em aplicações Java, nos deparamos com um problema conhecido como impedância, que é a incompatibilidade entre o modelo relacional e o modelo orientado a objetos. Esta incompatibilidade ocorre devido ao paradigma orientado a objetos ser baseado nos princípios da engenharia de software, ao passo que o paradigma relacional é fundamentado nos princípios matemáticos da álgebra relacional.

Uma solução para o problema da impedância é a utilização de bibliotecas ORM (*Object Relational Mapping*), que são responsáveis pelo mapeamento entre as classes Java e as tabelas do banco, deixando esse trabalho totalmente transparente para o desenvolvedor.

O Hibernate é a solução ORM para aplicações Java mais utilizada do mercado e traz como principais vantagens: ser um framework não intrusivo, ou seja, a arquitetura da aplicação permanece independente; ter uma comunidade bastante ativa; e ser muito bem documentado. Para seu correto funcionamento, ele precisa de metadados, isto é, informações inseridas nas classes (ou em arquivos de configuração) que são necessárias ao mapeamento para interligar elementos do modelo orientado a objetos a elementos do modelo relacional. Na versão 3.5.0, versão adotada no simulador, utilizamos anotações para mapear os relacionamentos do sistema como, por exemplo, `@OneToMany`, que define um relacionamento de um para muitos entre duas entidades, e `@ManyToMany`, que define um relacionamento de muitos para muitos. Em versões anteriores à 3, os mapeamentos eram realizados através de um arquivo XML, conhecido como `hibernate.cfg`.

Ainda neste artigo, demonstraremos como configurar um recurso do Hibernate responsável por exportar todo o esquema de criação do banco de dados de forma automática. Ao utilizar esse recurso, nos preocupamos apenas em anotar corretamente

as propriedades das classes para que o próprio Hibernate se encarregue com a administração da estrutura do banco de dados. Desta forma, nos concentraremos apenas com as regras de negócio e o layout da aplicação.

Criando a aplicação

Como dito no início deste artigo, criaremos um pequeno sistema simulador de perguntas e respostas para demonstrar o mecanismo de virtualização e a técnica de paginação com JasperReports. Ao término dos simulados o sistema disponibilizará uma opção para gerar um relatório com o histórico de resultados das avaliações realizadas, apresentando dados estatísticos de desempenho dos testes. Também analisaremos na prática alguns recursos RIA do PrimeFaces, e ainda, adotaremos o Hibernate para acessar os dados da aplicação. As páginas do projeto, por sua vez, serão baseadas em *templates* construídos com Facelets, do JSF. Além disso, o banco de dados utilizado será o MySQL e o acesso se dará através da JPA 2.0, com o controle de transações efetuado pelo container.

O ambiente requerido para implementar o sistema envolverá os softwares listados a seguir, que precisam estar instalados na máquina em que o desenvolvimento dos exemplos ocorrerá (se necessário, confira a seção **Links** para saber onde baixá-los):

- Container web Tomcat 7.0;
- Eclipse Luna para desenvolvedores Java EE;
- Java Development Kit versão 1.7 configurado no Eclipse;
- SGBD MySQL 5.1.17 ou superior;
- iReport 5.5.0;
- JasperReports 5.5.0;
- Apache Maven 2.0.2.

Uma vez que você tenha todos os softwares configurados na máquina, vamos criar o projeto Maven. Para isto, na view *Project Explorer* do Eclipse, clique com o botão direito, selecione *New* e clique em *Other*. Na tela que surge, selecione *Maven* e depois clique em *Maven Project*. Feito isso, marque a opção *Create a simple Project (skip archetype selection)* e clique em *Next*. Neste passo, informe no campo *Group Id* o valor “*br.com.javamagazine*”, em *Artifact Id* o valor “*simulador*”, em *Packaging* o valor “*war*” e no campo *Name* o valor “*simulador*”. Em seguida, clique em *Finish*.

Executados estes passos iniciais, vamos configurar alguns parâmetros adicionais para a aplicação rodar corretamente. Sendo assim, clique com o botão direito em cima da raiz do projeto e depois na opção *Properties*. Na tela que surge, clique em *Project Facets*. No item *Dynamic Web Module*, selecione a versão 3.0, no item *Java*, selecione a opção 1.7, no item JavaServer Faces escolha a opção 2.2 e clique em *OK*.

Após finalizarmos as alterações nas propriedades do projeto, nosso próximo passo será alterar o arquivo *web.xml*, que se encontra no diretório *src/main/webapp/WEB-INF*. Essas modificações são necessárias para que o JSF funcione corretamente. Entre elas, definiremos também a página inicial do sistema. Portanto, deixe o código igual ao da **Listagem 2**.

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

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
04   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
05   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
06   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
07   version="3.0">
08   <display-name>simulador</display-name>
09   <welcome-file-list>
10     <welcome-file>pagina_inicial.xhtml</welcome-file>
11   </welcome-file-list>
12   <servlet>
13     <servlet-name>Faces Servlet</servlet-name>
14     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
15     <load-on-startup>1</load-on-startup>
16   </servlet>
17   <servlet-mapping>
18     <servlet-name>Faces Servlet</servlet-name>
19     <url-pattern>*.xhtml</url-pattern>
20   </servlet-mapping>
21   <context-param>
22     <param-name>qtde_dados_carga_base</param-name>
23     <param-value>5</param-value>
24   </context-param>
25   <context-param>
26     <param-name>porcentual_acerto_avaliacao</param-name>
27     <param-value>50</param-value>
28   </context-param>
29 </web-app>
```

As principais configurações apresentadas são:

- **Linhas 08 a 10:** Informa a página inicial do sistema como *pagina_inicial.xhtml*;
- **Linhas 11 a 15:** Registra a classe **FacesServlet** como *Front Controller* da nossa aplicação e informa que ela deve ser carregada em tempo de deployment;
- **Linhas 16 a 19:** Informa que as requisições a serem processadas pelo *Front Controller* **FacesServlet** devem seguir o *pattern* “*.xhtml”;
- **Linhas 20 a 27:** Definimos dois atributos de contexto. Desta forma teremos acesso aos seus conteúdos em qualquer parte do sistema e ainda podemos alterá-los sem a necessidade de mudar o código fonte da aplicação.

Modificando o *pom.xml*

Como utilizaremos várias bibliotecas não fornecidas pelo JDK, para que elas fiquem disponíveis no projeto precisamos configurá-las no arquivo *pom.xml*. Deste modo, clique duas vezes sobre este arquivo para editar seu código fonte, que deve ficar semelhante ao da **Listagem 3**.

As principais configurações apresentadas no código são as seguintes:

- **Linhas 04 a 07:** Especificações gerais do projeto informadas no início do artigo, ao criarmos a aplicação. O **groupId** identifica o grupo de projetos de uma organização e normalmente obedece à estrutura de pacotes do sistema. Já o **artifactId** é o nome utilizado para gerar o *war*;
- **Linha 08:** Informa o nome do projeto;

Listagem 3. Código do arquivo pom.xml.

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
02   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
04   http://maven.apache.org/xsd/maven-4.0.0.xsd">
05   <modelVersion>4.0.0</modelVersion>
06   <groupId>br.com.javamagazine</groupId>
07   <artifactId>simulador</artifactId>
08   <version>0.0.1-SNAPSHOT</version>
09   <packaging>war</packaging>
10   <name>simulador</name>
11   <build>
12     <plugins>
13       <plugin>
14         <artifactId>maven-compiler-plugin</artifactId>
15         <version>2.0.2</version>
16         <configuration>
17           <source>1.7</source>
18           <target>1.7</target>
19         </configuration>
20       </plugin>
21     </plugins>
22   </build>
23   <repositories>
24     <repository>
25       <id>JBoss Repo</id>
26       <url>http://repository.jboss.com/maven2</url>
27     </repository>
28     <repository>
29       <id>maven2-repository.dev.java.net</id>
30       <name>Java.net Repository for Maven</name>
31       <url>http://download.java.net/maven/2</url>
32     </repository>
33     <repository>
34       <id>prime-repo</id>
35       <name>PrimeFaces Maven Repository</name>
36       <url>http://repository.primefaces.org</url>
37       <layout>default</layout>
38     </repository>
39     <repository>
40       <id>central</id>
41       <url>http://repo1.maven.org/maven2</url>
42     </repository>
43   </repositories>
44   <dependencies>
45     <dependency>
46       <groupId>org.hibernate</groupId>
47       <artifactId>hibernate-entitymanager</artifactId>
48       <version>3.5.0-Beta-2</version>
49     </dependency>
50     <dependency>
51       <groupId>org.slf4j</groupId>
52       <artifactId>slf4j-api</artifactId>
53       <version>1.7.1</version>
54     </dependency>
55     <dependency>
56       <groupId>commons-collections</groupId>
57       <artifactId>commons-collections</artifactId>
58       <version>3.0.</version>
59     </dependency>
60     <dependency>
61       <groupId>com.sun.faces</groupId>
62       <artifactId>jsf-api</artifactId>
63       <version>2.0.0-b13</version>
64     </dependency>
65     <dependency>
66       <groupId>com.sun.faces</groupId>
67       <artifactId>jsf-impl</artifactId>
68       <version>2.0.0-b13</version>
69     </dependency>
70     <dependency>
71       <groupId>javax.servlet</groupId>
72       <artifactId>javax.servlet-api</artifactId>
73       <version>3.0.1</version>
74     </dependency>
75     <dependency>
76       <groupId>org.primefaces</groupId>
77       <artifactId>primefaces</artifactId>
78       <version>5.1</version>
79     </dependency>
80     <dependency>
81       <groupId>net.sf.jasperreports</groupId>
82       <artifactId>jasperreports</artifactId>
83       <version>5.5.0</version>
84     </dependency>
85     <dependency>
86       <groupId>mysql</groupId>
87       <artifactId>mysql-connector-java</artifactId>
88       <version>5.1.17</version>
89     </dependency>
90     <dependency>
91       <groupId>net.vidageek</groupId>
92       <artifactId>mirror</artifactId>
93       <version>1.6.1</version>
94     </dependency>
95   </dependencies>
96 </project>
```

- **Linhas 09 a 20:** Configura a versão 1.7 para o *plugin* de compilação do código fonte do sistema. Desta forma, informamos ao Eclipse que o JDK 1.7 deve ser usado e que ele precisa ser colocado no *classpath* da aplicação;
- **Linhas 21 a 41:** Define todos os repositórios necessários para baixar as dependências que serão utilizadas na aplicação;
- **Linhas 42 a 93:** Define todas as dependências utilizadas no projeto;
- **Linhas 43 a 47:** Declara as dependências do Hibernate;
- **Linhas 48 a 67:** Declara as dependências necessárias para o funcionamento do JSF;
- **Linhas 68 a 72:** Define as dependências necessárias para utilizar a anotação `@WebFilter("/*")`, como veremos adiante. A partir

desta anotação conseguimos interceptar todas as requisições efetuadas pelo usuário e deixamos o controle das transações a cargo do container;

- **Linhas 73 a 77:** Declara as dependências do PrimeFaces;
- **Linhas 78 a 82:** Declara as dependências do JasperReports;
- **Linhas 83 a 87:** Define o *driver* do MySQL como dependência para podermos acessar o SGBD;
- **Linhas 88 a 92:** Define o *driver* da biblioteca Mirror como dependência para uso de reflexão na funcionalidade de paginação com JasperReports.

Após modificar o *pom.xml*, vamos atualizar as dependências do Maven no sistema. Para realizar esse procedimento, clique com o

botão direito em cima da raiz do projeto, selecione *Maven* e clique em *Update Project*. Na tela que surge, clique em *OK* e aguarde alguns segundos para que o Eclipse realize o processo.

Configurando a JPA e o banco de dados

Para inserir a JPA em nosso projeto, inicialmente precisamos criar o arquivo *persistence.xml*, que é responsável pela configuração desta API. Deste modo, crie uma pasta chamada *META-INF* dentro de *src/main/resources*. Após realizar este passo, crie o *persistence.xml* dentro do novo diretório para que o container consiga acessá-lo e modifique o código gerado automaticamente para que fique igual ao da **Listagem 4**.

Listagem 4. Código do arquivo *persistence.xml*.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
05   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
06   version="2.0">
07   <persistence-unit name="simuladorPU" transaction-type="RESOURCE_LOCAL">
08     <class>br.com.javamagazine.simulador.domain.Tema</class>
09     <class>br.com.javamagazine.simulador.domain.Pergunta</class>
10     <class>br.com.javamagazine.simulador.domain.Resposta</class>
11     <class>br.com.javamagazine.simulador.domain.Avaliacao</class>
12     <class>br.com.javamagazine.simulador.domain.ResultadoAvaliacao
13       <properties>
14         <property name="hibernate.dialect"
15           value="org.hibernate.dialect.MySQLDialect"/>
16         <property name="hibernate.connection.driver_class"
17           value="com.mysql.jdbc.Driver"/>
18         <property name="hibernate.connection.username" value="root"/>
19         <property name="hibernate.connection.password" value="root"/>
20         <property name="hibernate.connection.url"
21           value="jdbc:mysql://localhost:3306/simulador_db"/>
22         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
23       </properties>
24     </persistence-unit>
25   </persistence>
```

ResultadoAvaliacao. Para criar a primeira delas (**Tema**), clique com o botão direito em cima do projeto, selecione a opção *New* e depois clique em *Class*. Na tela que surge, informe em *Package* o valor “*br.com.javamagazine.simulador.domain*”, em *Name* informe “**Tema**” e clique no botão *Finish*. O código inicial que o Eclipse gera automaticamente deve ser modificado para ficar semelhante ao da **Listagem 5** (lembre-se de inserir os *gets*, *sets*, *equals()* e *hashCode()* que foram omitidos).

Listagem 5. Código da classe **Tema**.

```
01 package br.com.javamagazine.simulador.domain;
02
03 import java.io.Serializable;
04 import javax.persistence.*;
05
06 @NamedQueries({
07   @NamedQuery(name=Tema.LISTAR_TEMAS, query="select t from Tema t")
08 })
09 @Entity
10 public class Tema implements Serializable {
11
12   private static final long serialVersionUID = 1012607822199188869L;
13
14   public static final String LISTAR_TEMAS = "Tema.buscarTemas";
15
16   @Id
17   @GeneratedValue(strategy=GenerationType.IDENTITY)
18   private Long id;
19
20   @Column
21   private String descricao;
22
23   public Tema() {
24     super();
25   }
26
27   public Tema(String descricao) {
28     this.descricao = descricao;
29   }
30
31   // gets e sets omitidos
32   // equals() e hashCode() omitidos
33 }
```

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

- **Linha 6:** Definimos o nome **simuladorPU** para nossa **PersistenceUnit** e indicamos que o tipo de transação é **RESOURCE_LOCAL**, o que significa que teremos que obter o **EntityManager** através da **EntityManagerFactory** e que o controle das transações será manual;
- **Linhas 7 a 11:** Lista as entidades que nosso projeto terá;
- **Linhas 12 a 19:** Configura o dialeto a ser usado pelo Hibernate (do MySQL), o *driver* e os parâmetros da conexão com o banco;
- **Linha 18:** Recurso comentado na introdução do artigo que exporta automaticamente para o banco de dados, em tempo de deployment, o esquema DDL configurado nas anotações das entidades da aplicação, evitando assim ter que realizar este trabalho através de scripts SQL.

Criando as entidades e os enums

Como vimos, tivemos que declarar cinco entidades no arquivo *persistence.xml*. São elas: **Tema**, **Pergunta**, **Resposta**, **Avaliacao** e

Os principais trechos do código apresentado podem ser explicados da seguinte forma:

- **Linha 6, 7 e 8:** O uso da anotação **@NamedQueries** serve para definir consultas dentro de uma entidade. Nela, declaramos uma pesquisa que listará todos os temas cadastrados no sistema;
- **Linha 9:** A anotação **@Entity** indica que esta classe é uma entidade mapeada para uma tabela no banco de dados (por padrão, o nome da tabela é o mesmo nome da classe, mas com a primeira letra também em minúsculo);
- **Linha 14:** Definimos um atributo estático, cujo valor servirá como chave para que o Hibernate seja capaz de disparar a consulta declarada na linha 7, através do atributo **name**;
- **Linha 16:** Seta o atributo **id** como chave primária da tabela;
- **Linha 17:** Define que o valor do atributo **id** será gerado automaticamente através do auto incremento;

- Linhas 20 e 21:** Define que a classe **Tema** terá um atributo mapeado e que ele será gravado no banco de dados no formato **String**;
- Linhas 27, 28 e 29:** Utilizamos um construtor parametrizado para inicializar a propriedade **descricao** no momento da instanciação de um objeto do tipo **Tema**. Ao utilizá-lo somos obrigados a implementar um construtor padrão, como pode ser observado nas linhas 23, 24 e 25. Isto se faz necessário para o correto funcionamento do Hibernate.

Repetindo o mesmo procedimento para geração de uma classe, crie a entidade **Pergunta** no pacote informado anteriormente e substitua o código gerado pelo Eclipse para ficar igual ao da **Listagem 6**. Mais uma vez, lembre-se de inserir os gets, sets, **equals()** e **hashCode()**.

Por enquanto a classe ficará com erro, pois precisamos criar os enums **NivelEnum** e **TipoQuestaoEnum** e a classe **Resposta**, o que será realizado mais à frente. Na linha 10 declaramos uma consulta parametrizada que é caracterizada pela utilização do caractere dois pontos imediatamente antes de um nome dentro de uma HQL. Desta forma o Hibernate entende que a consulta necessita de parâmetro para funcionar. Na linha 12 definimos que a classe **Pergunta** será uma entidade no banco de dados ao anotá-la com **@Entity**. Nas linhas 19 e 20 informamos que a propriedade **id** será chave primária da tabela e que o SGBD utilizará

como estratégia de geração de chaves o auto incremento. Já na linha 23, com o uso da anotação **@Lob**, do Hibernate, indicamos que a propriedade anotada poderá armazenar dados de tamanho superior ao que uma **String** pode suportar. Isso se deve ao fato que podemos ter enunciados de perguntas que ultrapassem a capacidade que um **varchar** pode suportar.

Na linha 27 aplicamos a anotação **@OneToMany** para determinar o tipo de relacionamento entre as entidades **Pergunta** e **Resposta**. Com o uso desta anotação sinalizamos que para cada pergunta podemos ter várias respostas e que teremos um relacionamento bidirecional, informado através do atributo **mappedBy**. Nas linhas 30 e 34 temos a anotação **@Enumerated**, que utilizamos para mapear os enums e definir se trabalharemos com valores inteiros ou alfanuméricos, neste caso usaremos valores inteiros. Usamos a anotação **@OneToOne** na linha 38 para estabelecer o tipo de relacionamento entre as entidades **Pergunta** e **Tema**, informando assim que para cada pergunta teremos apenas um tema. Finalmente, a anotação **@Transient**, utilizada nas linhas 41, 44 e 47, informa ao Hibernate que estas propriedades não serão persistidas e que ele deve desconsiderá-las.

Nosso próximo passo será criar a próxima entidade, **Resposta**, no mesmo pacote das outras classes. Após esse procedimento, substitua o código gerado para ficar igual ao da **Listagem 7** (lembre-se novamente de inserir os gets, sets, **equals()** e **hashCode()**).

Listagem 6. Código da classe Pergunta.

```

01 package br.com.javamagazine.simulador.domain;
02
03 import java.io.Serializable;
04 import java.util.*;
05 import javax.persistence.*;
06 import br.com.javamagazine.simulador.enums.NivelEnum;
07 import br.com.javamagazine.simulador.enums.TipoQuestaoEnum;
08
09 @NamedQueries({
10     @NamedQuery(name=Pergunta.LISTAR_TEMA_PERGUNTAS,
11         query="select p from Pergunta p where p.tema = :tema")
12 })
13 @Entity
14 public class Pergunta implements Serializable {
15
16     private static final long serialVersionUID = -187649131345659471L;
17
18     public static final String LISTAR_TEMA_PERGUNTAS =
19         "Pergunta.buscarTemaPerguntas";
20
21     @Id
22     @GeneratedValue(strategy=GenerationType.IDENTITY)
23     private Long id;
24
25     @Lob
26     @Column
27     private String descricao;
28
29     @OneToMany(mappedBy="pergunta", fetch=FetchType.EAGER)
30     private List<Resposta> listaResposta;
31
32     private NivelEnum nivel;
33
34     @Enumerated(EnumType.ORDINAL)
35     @Column
36     private TipoQuestaoEnum tipoQuestao;
37
38     @OneToOne
39     private Tema tema;
40
41     @Transient
42     private List<Resposta> respostasSelecionadas;
43
44     @Transient
45     private Resposta respostaSimples;
46
47     @Transient
48     private boolean contemRespostasErradas;
49
50     public Pergunta() {
51         super();
52     }
53
54     public Pergunta(String descricao, NivelEnum nivel,
55         TipoQuestaoEnum tipoQuestao, Tema tema) {
56         super();
57         this.descricao = descricao;
58         this.nivel = nivel;
59         this.tipoQuestao = tipoQuestao;
60         this.tema = tema;
61     }
62
63     // gets e sets omitidos
64     // equals() e hashCode() omitidos
65 }
```

Listagem 7. Código da classe Resposta.

```
01 package br.com.javamagazine.simulador.domain;
02
03 import java.io.Serializable;
04 import javax.persistence.*;
05
06 @Entity
07 public class Resposta implements Serializable {
08
09     private static final long serialVersionUID = 1012607822199188869L;
10
11     @Id
12     @GeneratedValue(strategy=GenerationType.IDENTITY)
13     private Long id;
14
15     @Lob
16     @Column
17     private String descricao;
18
19     @ManyToOne
20     private Pergunta pergunta;
21
22     @Column
23     private boolean correta;
24
25     public Resposta() {
26         super();
27     }
28
29     public Resposta(String descricao, Pergunta pergunta, boolean correta) {
30         super();
31         this.descricao = descricao;
32         this.pergunta = pergunta;
33         this.correta = correta;
34     }
35
36     // gets e sets omitidos
37     // equals() e hashCode() omitidos
38 }
```

Listagem 8. Código da classe Avaliacao.

```
01 package br.com.javamagazine.simulador.domain;
02
03 import java.io.Serializable;
04 import java.util.List;
05 import javax.persistence.*;
06 import br.com.javamagazine.simulador.util.Cronometro;
07
08 @NamedQueries({
09     @NamedQuery(name=Avaliacao.LISTAR_TODOS,
10     query="select a from Avaliacao a")
11 })
12 @Entity
13 public class Avaliacao implements Serializable {
14
15     private static final long serialVersionUID = 1012607822199188869L;
16
17     public static final String LISTAR_TODOS = "Avaliacao.buscarAvaliacoes";
18
19     @Id
20     @GeneratedValue(strategy=GenerationType.IDENTITY)
21     private Long id;
22
23     @Column
24     private String descricao;
25
26     @ManyToMany
27     @JoinTable(name="AVA_PER",
28     joinColumns=@JoinColumn(name="AVA_ID"),
29     inverseJoinColumns=@JoinColumn(name="PER_ID"))
30     private List<Pergunta> listaPergunta;
31
32     @Column
33     private Long tempo;
34
35     @Column
36     private Integer quantidadeQuestoes;
37
38     @Column
39     private Double porcentagemAcerto;
40
41     @OneToOne
42     private Tema tema;
43
44     public Avaliacao() {
45         super();
46     }
47
48     public Avaliacao(String descricao, List<Pergunta> listaPergunta,
49     Integer quantidadeQuestoes, Long tempo,
50     Double porcentagemAcerto, Tema tema) {
51         super();
52         this.descricao = descricao;
53         this.listaPergunta = listaPergunta;
54         this.quantidadeQuestoes = quantidadeQuestoes;
55         this.tempo = tempo;
56         this.porcentagemAcerto = porcentagemAcerto;
57         this.tema = tema;
58     }
59
60     public String getTempoFormatado() {
61         return Cronometro.formatarTempoAvaliacao(getTempo());
62     }
63 }
```

Ao verificar o código constatamos que muitas das anotações utilizadas já foram analisadas. Portanto, manteremos o foco apenas na que ainda não foi explicada; neste caso, **@ManyToOne**, definida na linha 19. Esta concretiza um relacionamento onde podemos ter uma, nenhuma ou muitas respostas para determinada pergunta.

Criaremos agora a classe **Avaliacao**, no mesmo pacote das classes definidas até aqui (**br.com.javamagazine.simulador.domain**). Feito isso, substitua o código gerado para ficar igual ao da **Listagem 8** e como de costume, lembre-se de inserir os gets, sets, **equals()** e **hashCode()**.

Novamente podemos observar que temos um erro de compilação, pois precisamos da classe **Cronometro**, ainda não implementada. Desconsiderando esse erro e continuando com a análise das anotações, temos nas linhas 8, 9 e 10 a declaração da anotação **@NamedQueries**, que serve para definir consultas dentro de uma entidade. Nela, declaramos uma pesquisa que listará todas as avaliações cadastradas no sistema. Essa definição é considerada por muitos uma boa prática, pois ao centralizar consultas em um mesmo lugar, tornamos o código mais organizado e facilitamos futuras alterações. Na linha 16, criamos um atributo estático que será

Listagem 9. Código da classe ResultadoAvaliacao.

```
01 package br.com.javamagazine.simulador.domain;
02
03 import java.io.Serializable;
04 import java.util.Date;
05 import javax.persistence.*;
06
07 @NamedQueries({
08     @NamedQuery(name=ResultadoAvaliacao.LISTAR_TODOS,
09         query="select ra from ResultadoAvaliacao ra")
10 })
11 public class ResultadoAvaliacao implements Serializable {
12
13     private static final long serialVersionUID = 1012607822199188869L;
14
15     public static final String LISTAR_TODOS =
16         "ResultadoAvaliacao.buscarResultadosAvaliacoes";
17
18     @Id
19     @GeneratedValue(strategy=GenerationType.IDENTITY)
20     private Long id;
21
22     @OneToOne
23     private Avaliacao avaliacao;
24
25     @Temporal(TemporalType.DATE)
26     @Column
```

```
27     private Date dataAvaliacao;
28
29     @Column
30     private Double pontuacao;
31
32     @Column
33     private Integer totalCorretas;
34
35     @Column
36     private Integer totalErradas;
37
38     public ResultadoAvaliacao() {
39     }
40
41     public ResultadoAvaliacao(Avaliacao avaliacao, Date dataAvaliacao,
42         double pontuacao, Integer totalCorretas, Integer totalErradas) {
43         this.avaliacao = avaliacao;
44         this.dataAvaliacao = dataAvaliacao;
45         this.pontuacao = pontuacao;
46         this.totalCorretas = totalCorretas;
47         this.totalErradas = totalErradas;
48     }
49
50     // gets e sets omitidos
51     // equals() e hashCode() omitidos
```

utilizado pelo Hibernate como chave para disparar a consulta “listar todos”. Na linha 25 temos a anotação **@ManyToMany**, que define um relacionamento de muitos para muitos, ou seja, será possível ter muitas perguntas em uma avaliação e uma pergunta poderá ser usada em diversas avaliações.

Com a ajuda da anotação **@JoinTable**, definida na linha 26, podemos personalizar as características da tabela auxiliar que será gerada automaticamente pelo Hibernate por conta do relacionamento de N para N. Neste caso, mudamos o nome da tabela e suas respectivas chaves estrangeiras. Com o atributo **name** da anotação, nomeamos a tabela para **AVA_PER** e nas linhas 27 e 28 nomeamos suas chaves estrangeiras. Finalmente, as linhas 57, 58 e 59 declaram um método que retornará o tempo restante da avaliação em formato definido pela classe **Cronometro**.

Criaremos agora a última classe do sistema, **ResultadoAvaliacao**, no mesmo pacote das demais. Após esse procedimento, substitua o código gerado para ficar igual ao da **Listagem 9** e de modo semelhante, insira os gets, sets, **equals()** e **hashCode()**.

Como diferencial em relação às listagens analisadas anteriormente, na linha 24 usamos a anotação **@Temporal**, que define que o tipo de dado deste campo na tabela será do tipo **Date**. Da linha 40 a 46, declaramos um construtor com parâmetros. Desta forma, inicializaremos os valores das propriedades na instanciação do objeto quando necessário, evitando assim ter que chamar seus respectivos métodos sets. E para atender às exigências do Hibernate e a aplicação funcionar corretamente, declaramos nas linhas 37 e 38 um construtor padrão.

Ao término da implementação destas classes, daremos início à construção dos enums. Dito isso, vamos criar o enum **NivelEnum**, que poderia ser uma entidade no sistema, mas para simplificar, optamos por o codificar dessa forma. Deste modo, clique com botão direito em cima do projeto, selecione a opção *New* e depois clique em *Enum*. Na tela que surge, informe em *Package* o valor “**br.com.javamagazine.simulador.enums**”, em *Name* informe “**NivelEnum**” e clique no botão *Finish*. Logo após, modifique o código para ficar semelhante ao da **Listagem 10**. Este enum será utilizado para classificar o grau de dificuldade de uma pergunta na avaliação.

Listagem 10. Código do enum NivelEnum.

```
01 package br.com.javamagazine.simulador.enums;
02
03 public enum NivelEnum {
04
05     MUITO_FACIL("Muito Fácil"),
06     FACIL("Fácil"),
07     MEDIO("Médio"),
08     DIFICIL("Difícil"),
09     MUITO_DIFICIL("Muito Difícil");
10
11     private String descricao;
12
13     private NivelEnum(String descricao) {
14         this.descricao = descricao;
15     }
16
17     // gets e sets omitidos
18 }
```

Criaremos agora o enum **TipoQuestaoEnum**, no mesmo pacote do enum criado anteriormente. Após esse procedimento, substitua o código gerado para ficar igual ao da **Listagem 11** (lembre-se novamente de inserir os gets e sets). Este enum configura a maneira como as respostas serão exibidas no decorrer da avaliação. Ao cadastrar uma resposta para determinada pergunta do simulado, escolhemos também seu tipo. Cada opção renderiza determinado componente do PrimeFaces. Esta foi a solução adotada para determinar se a questão é de múltiplas escolhas com apenas uma alternativa correta, mais de uma ou de arrastar e soltar.

Listagem 11. Código do enum TipoQuestaoEnum.

```
01 package br.com.javamagazine.simulador.enums;
02
03 public enum TipoQuestaoEnum {
04
05     MARCAR_SIMPLES("MS", "Marcar Simples"),
06     MARCAR_MULTIPLAS("MM", "Marcar Múltiplas"),
07     ARRASTAR_SOLTAR("AS", "Arrastar e Soltar");
08
09     private String codigo;
10     private String descricao;
11
12     private TipoQuestaoEnum(String codigo, String descricao) {
13         this.codigo = codigo;
14         this.descricao = descricao;
15     }
16
17     // gets e sets omitidos
18 }
```

Finalmente, vamos criar o último enum da aplicação, que se chamará **ConstantesEnum**. Este será responsável por armazenar as principais constantes utilizadas no sistema. Para manter o padrão, o colocaremos no mesmo pacote dos outros enums. Feito isso, substitua o código gerado para ficar igual ao da **Listagem 12**.

Criando as classes utilitárias e o VO do projeto

Concluída a etapa anterior, chegou a hora de implementar as classes utilitárias. Começaremos pela classe **Cronometro**, responsável pelo funcionamento do cronômetro regressivo que será utilizado pelo simulador. Para isso, execute os mesmos passos utilizados na criação das entidades, alterando apenas o nome do *Package* para “*br.com.javamagazine.simulador.util*”. Feito isso, modifique o código para ficar semelhante ao da **Listagem 13**.

Reunimos nesta classe todos os métodos encarregados pelo funcionamento do cronômetro, adotando, mais uma vez, boas práticas em nosso projeto. Ao centralizar ações comuns em um só lugar, podemos acessá-las de diferentes pontos do sistema sem que seja necessário repetir código. Da linha 12 à linha 17, temos o método **obterDataTempo()**, responsável por converter um dado do tipo **Long** em **Calendar**. Esta conversão se faz necessária porque ao cadastrar uma avaliação, gravamos sua duração em milissegundos.

Entre as linhas 19 e 22 declaramos o método **formatarTempoAvaliacao()**, que retornará o tempo formatado no padrão “**hh:mm:ss**”. O próximo método, seguindo a ordem de declaração, da linha 24 à linha 34, é o **decrementarTempoAvaliacao()**, que representa o motor do cronômetro regressivo, ou seja, tem a responsabilidade de decrementar o tempo em um segundo a cada segundo.

Já nas linhas 36, 37 e 38 temos o método **fimTeste()**, que verifica se o tempo já expirou. Por sua vez, o método **configurarTempoAvaliacao()**, definido no intervalo das linhas 40 e 44, configura o tempo de duração do teste.

No intervalo entre as linhas 46 e 49 temos o método **obterDataAtualAvaliacao()**, com a função de formatar a data de avaliação no padrão “**dd/MM/yyyy**”. Finalmente, entre as linhas 51 e 53, criamos o método privado **obterCalendario()**, que como o próprio nome indica, retorna um objeto do tipo calendário. Este método foi criado porque sua lógica é utilizada em diferentes pontos desta classe, evitando assim a repetição de código.

Chegou a hora de implementar a classe encarregada pela configuração dos relatórios na aplicação. Para isso, criaremos a classe **Relatorio** no mesmo pacote informado anteriormente e substituiremos o código gerado pelo Eclipse para ficar igual ao da **Listagem 14**. Para ser possível imprimir um relatório na aplicação, temos que prover determinadas informações que o JasperReports precisa para funcionar, tais como: o local do arquivo de extensão Jasper, que é a compilação do arquivo criado no iReport para definir o layout do relatório; um mapa de parâmetros, onde depositamos informações que personalizam o relatório, como o título e o cabeçalho; e a fonte de dados que será exibida nos resultados. Nesta classe configuramos também o mecanismo de virtualização proposto no artigo.



Note que efetuamos todas as configurações dentro do construtor da classe. Optamos por esse caminho pela praticidade.

Nas linhas 24, 25 e 26, setamos, respectivamente, uma referência ao objeto **response**, que será utilizado para escrever os bytes do pdf gerado pelo JasperReports na resposta, a fonte de dados que contém os registros retornados diretamente da base de dados e serão listados no relatório; e finalmente, atribuímos um mapa de parâmetros que carregam informações que personalizam os relatórios.

Na linha 28 informamos o caminho do arquivo Jasper, que nada mais é do que o DTD compilado pelo iReport. Na linha 31 chamamos um método que configura algumas informações genéricas a todos os relatórios, como a data atual da geração e o nome do sistema. Já as linhas 32 e 33 configuraram de fato o mecanismo de virtualização. Nelas definimos o tamanho máximo da porção de memória heap que será cacheada em disco e o diretório onde essa troca acontecerá. Por fim, na linha 36 preenchemos todos os campos do relatório com as informações manipuladas pela aplicação e guardamos sua referência para utilizar no momento de exportá-lo para o browser.

Para desenvolver o recurso de paginação, criaremos uma classe Java que implemente a interface **JRDataSource** da biblioteca JasperReports. Deste modo, teremos que implementar os métodos **getFieldValue()** e **next()**, que se encarregarão do correto funcionamento deste recurso. Como vantagem, esta funcionalidade nos traz o controle no gerenciamento do uso da memória heap.

Listagem 12. Código do enum ConstantesEnum.

```
01 package br.com.javamagazine.simulador.enum;
02
03 public enum ConstantesEnum {
04
05     CHAVE_FLASH("flash", "resultadoAvaliacao"),
06     CAMINHO_RESULTADO("resultado", "/avaliacao/resultado.xhtml"),
07     CAMINHO_TIMEOUT("timeout", "/simulador/avaliacao/timeout.xhtml"),
08     CHAVE_TOTAL_CORRETAS("corretas", "totalCorretas"),
09     CHAVE_TOTAL_ERRADAS("erradas", "totalErradas"),
10     CHAVE_CABECALHO_RELATORIO("cabecalho", "cabecalho"),
11     PRIMEIRA_PERGUNTA("indice", 0),
12     TAMANHO_MAXIMO("tamanho", 1),
13     TAMANHO_VIRTUALIZACAO("virtualizacao", 1),
14     EXTENSAO_JASPER("extensao", "jasper"),
15     CAMINHO_JASPER("jasper", "/relatorio/"),
16     CAMINHO_VIRTUAL("virtual", "/relatorio/virtual/"),
17     NOME_RELATORIO("avaliacao", "RelatorioAvaliacao"),
18     TITULO_RELATORIO("titulo", "Relatório de Avaliações"),
19     DATA_ATUAL("data", new Date()),
20     NOME_SISTEMA("sistema", "Simulador - Java Magazine");
21
22     private String nome;
23     private Object valor;
24
25     private ConstantesEnum(String nome, Object valor) {
26         this.setNome(nome);
27         this.setValor(valor);
28     }
29
30     // gets e sets omitidos
31 }
```

Listagem 13. Código da classe Cronometro.

```
01 package br.com.javamagazine.simulador.util;
02
03 import java.util.List;
04 import java.io.Serializable;
05 import java.text.SimpleDateFormat;
06 import java.util.*;
07
08 public class Cronometro implements Serializable {
09
10     private static final long serialVersionUID = -2043309898587041724L;
11
12     public static Calendar obterDataTempo(Long tempo) {
13         Calendar cal = Cronometro.obterCalendario();
14         cal.clear();
15         cal.setInMillis(tempo);
16         return cal;
17     }
18
19     public static String formatarTempoAvaliacao(Long tempo) {
20         Calendar cal = Cronometro.obterDataTempo(tempo);
21         return cal.get(Calendar.HOUR) == 0 ? new SimpleDateFormat("00:mm:ss")
22             .format(cal.getTime()) : new SimpleDateFormat("hh:mm:ss")
23             .format(cal.getTime());
24     }
25
26     public static Calendar decrementarTempoAvaliacao(Long tempo) {
27         Calendar cal = Cronometro.obterDataTempo(tempo);
28         if (cal.get(Calendar.SECOND) == 0) {
29             cal.roll(Calendar.MINUTE, false);
30         }
31         if (cal.get(Calendar.HOUR) != 0 && cal.get(Calendar.MINUTE) == 0) {
32             cal.roll(Calendar.HOUR, false);
33         }
34         return cal;
35     }
36
37     public static boolean fimTeste(Calendar cal) {
38         return cal.get(Calendar.HOUR) == 0 && cal.get(Calendar.MINUTE) ==
39             0 && cal.get(Calendar.SECOND) == 0 ? true : false;
40     }
41
42     public static Long configurarTempoAvaliacao(Integer horas, Integer minutos,
43         Integer segundos) {
44         Calendar c = Cronometro.obterCalendario();
45         c.set(0, 0, horas, minutos, segundos);
46         return c.getTimeInMillis();
47     }
48
49     public static String obterDataAtualAvaliacao() {
50         Calendar cal = Cronometro.obterCalendario();
51         return new SimpleDateFormat("dd/MM/yyyy").format(cal.getTime());
52     }
53
54     private static Calendar obterCalendario() {
55         return Calendar.getInstance(new Locale("pt", "BR"));
56     }
57 }
```

Listagem 14. Código da classe Relatorio.

```

01 package br.com.javamagazine.simulador.util;
02
03 import java.io.File;
04 import java.util.*;
05 import javax.faces.context.FacesContext;
06 import javax.servlet.http.HttpServletResponse;
07 import br.com.javamagazine.simulador.enums.ConstantesEnum;
08 import net.sf.jasperreports.engine.*;
09 import net.sf.jasperreports.engine.fill.JRFileVirtualizer;
10
11 public class Relatorio {
12
13     private String caminhoJasper;
14     private String caminhoVirtualizacao;
15     private String nomeRelatorio;
16     private String tituloRelatorio;
17     private Map<String, Object> parametros;
18     private JasperReport relatorio;
19     private HttpServletResponse response;
20     private JasperPrint impressao;
21     private JRDataSource dataSource;
22
23     public Relatorio(JRDataSource dataSource, Map<String, Object>
parametros, String nomeRelatorio) {
24         setResponse((HttpServletResponse) FacesContext.getCurrentInstance()
.getExternalContext().getResponse());
25         setDataSource(dataSource);
26         setParametros(parametros);
27         String caminhoRelatorio = FacesContext.getCurrentInstance()
.getExternalContext().getRealPath(ConstantesEnum.CAMINHO_JASPER
.getValor().toString());
28         setCaminhoJasper(caminhoRelatorio.concat(File.separator).concat(
(nomeRelatorio).concat(ConstantesEnum.EXTENSAO_JASPER.getValor())
.toString()));
29         setCaminhoVirtualizacao(FacesContext.getCurrentInstance()
.getExternalContext().getRealPath(ConstantesEnum.CAMINHO_VIRTUAL
.getValor().toString()).concat(File.separator));
30         setNomeRelatorio(nomeRelatorio);
31         montarParametrosGenericosRelatorio();
32         JRFileVirtualizer fileVirtualizer = new JRFileVirtualizer(Integer.valueOf
(ConstantesEnum.TAMANHO_MAXIMO.getValor().toString()),
getCaminhoVirtualizacao());
33         getParametros().put(JRParameter.REPORT_VIRTUALIZER, fileVirtualizer);
34
35         try {
36             setImpressao(JasperFillManager.fillReport(getCaminhoJasper().toString(),
getParametros(), getDataSource()));
37         } catch(Exception e) {
38             e.printStackTrace();
39         }
40     }
41
42     public void montarParametrosGenericosRelatorio() {
43         try {
44             getParametros().put(ConstantesEnum.DATA_ATUAL.getNome(),
(Date)ConstantesEnum.DATA_ATUAL.getValor());
45             getParametros().put(ConstantesEnum.NOME_SISTEMA.getNome(),
ConstantesEnum.NOME_SISTEMA.getValor().toString());
46         } catch(Exception e) {
47             e.printStackTrace();
48         }
49     }
50
51     // gets e sets omitidos
52 }
```



Isto ocorre porque podemos configurar a quantidade de registros consultados por vez no banco de dados, e desta maneira, conseguimos também determinar o número de acessos ao disco.

A escolha do número de registros consultados por vez no banco de dados pode melhorar a gerência da memória. Por outro lado, é provável que aumente o processamento computacional. Isto é um exemplo típico de trade-off na computação, que é verificado quando há um conflito de escolhas/interesses. Por isso, devemos avaliar com cautela o uso desse mecanismo para tentar equilibrar o uso da memória e do processamento.

A paginação será implementada com o uso do JasperReports, do Hibernate e da biblioteca Mirror. Esta última se faz necessária porque usaremos reflexão para tornar genérico o acesso aos métodos gets das propriedades dos objetos envolvidos no processamento do JasperReports. Dito isso, crie a classe **JRDataSourcePaginacao** no mesmo pacote informado anteriormente e substitua o código gerado pelo Eclipse para ficar igual ao da **Listagem 15**.

Os métodos implementados da interface **JRDataSource** são chamados pelo JasperReports em tempo de execução, no momento em que ele inicia a interpretação do arquivo compilado e encontra referências às propriedades de um determinado objeto. Neste código, das linhas 18 a 22 criamos um construtor parametrizado. Nele, passamos a consulta HQL que será utilizada pelo Hibernate e seus respectivos limites inicial e final, que servirão para

Listagem 15. Código da classe JRDataSourcePaginacao.

```
01 package br.com.javamagazine.simulador.util;
02
03 import java.util.*;
04 import javax.persistence.TypedQuery;
05 import net.sf.jasperreports.engine.*;
06 import net.vidageek.mirror.dsl.*;
07 import br.com.javamagazine.simulador.enums.ConstantesEnum;
08 import br.com.javamagazine.simulador.persistence.HibernateUtil;
09
10 public class JRDataSourcePaginacao implements JRDataSource {
11
12     private Iterator<Object> iterator;
13     private Object cursor;
14     private final int paginacao;
15     private int inicio;
16     private String namedQuery;
17
18     public JRDataSourcePaginacao(String namedQuery) {
19         this.paginacao = Integer.valueOf(ConstantesEnum.
20             TAMANHO_VIRTUALIZACAO.getValor().toString());
21         this.inicio = -this.paginacao;
22         this.namedQuery = namedQuery;
23     }
24
25     @Override
26     public Object getFieldValue(JRField campo) throws JRException {
27         try {
28             String[] campos = campo.getName().split("\\.");
29
30             if (campos.length > 1) {
31                 AccessorsController instancia = new Mirror().on(this.cursor);
32
33                 for (int i=0; i<campos.length; i++) {
34                     if (campos.length-1 == i) {
35                         return instancia.invoke().getterFor(campos[i]);
36                     } else {
37                         instancia = new Mirror().on(instancia.invoke().getterFor(campos[i]));
38                     }
39
40             return null;
41         } else {
42             return new Mirror().on(this.cursor).invoke().getterFor(campo.getName());
43         }
44     } catch (Exception e) {
45         e.printStackTrace();
46         return null;
47     }
48 }
49
50     @Override
51     public boolean next() {
52         List<Object> list = null;
53
54         if (this.iterator != null && this.iterator.hasNext()) {
55             this.cursor = this.iterator.next();
56         } else {
57             this.inicio += this.paginacao;
58             list = paginar();
59
60             if (list == null || list.isEmpty()) {
61                 return false;
62             } else {
63                 this.iterator = list.iterator();
64             }
65
66             if (this.iterator != null && this.iterator.hasNext()) {
67                 this.cursor = this.iterator.next();
68             } else {
69                 return false;
70             }
71         }
72
73         return true;
74     }
75
76     public List<Object> paginar() {
77         TypedQuery<Object> query = HibernateUtil.getEntityManager().
78             createNamedQuery(this.namedQuery, Object.class);
79
80         query.setFirstResult(this.inicio);
81         query.setMaxResults(this.paginacao);
82
83         return query.getResultList();
84     }

```

determinar a quantidade máxima de registros recuperados de uma determinada consulta. Entre as linhas 24 e 48 implementamos o método **getFieldValue()**, cujo objetivo principal é obter informações das propriedades de um objeto qualquer. Acessamos os gets das propriedades através de reflexão. Desta forma não precisamos saber seus tipos, deixando este trabalho a cargo da aplicação.

Na linha 58 efetuamos uma chamada ao método **paginar()**, que, por sua vez, é encarregado de realizar a consulta dos dados que serão apresentados no relatório. Aqui determinaremos a quantidade máxima de registros retornados do banco de dados e os aloçamos na memória heap.

No intervalo entre as linhas 50 e 74 configuramos o método **next()**, que é o primeiro a ser executado pelo framework. Sua principal atribuição é identificar se existe pelo menos um objeto recuperado na consulta, para que seja possível obter o conteúdo de suas propriedades.

Com isso, podemos concluir que as funcionalidades implementadas nos métodos desta classe lidam com todas as questões referentes ao mecanismo de paginação do sistema.

Agora, falta criar a classe responsável por imprimir o relatório em formato pdf e exportá-lo para o browser. Portanto, dentro do pacote **util** do sistema, crie a classe **GeradorRelatorio** e altere seu código conforme o conteúdo da **Listagem 16**.

Na linha 23 desta classe obtemos os bytes provenientes do relatório gerado pelo JasperReports. Antes disso, na linha 19, configuramos o cabeçalho da resposta HTTP, para que o browser entenda que está recebendo um arquivo em formato pdf e saiba como tratá-lo. Finalmente, das linhas 27 a 31, efetuamos a exportação do arquivo para o navegador.

Para encerrar esta parte da implementação do simulador, criaremos uma classe que terá as características de um VO, que é apenas um objeto comum, onde agrupamos diferentes informações com o objetivo de trafegá-las entre os componentes da aplicação. Desta forma, ao invés de enviarmos esses dados todos segmentados, os enviamos dentro de uma estrutura única. No nosso simulador, armazenaremos neste objeto as respostas selecionadas pelo usuário durante a avaliação e o seu respectivo tempo de duração.

Listagem 16. Código da classe GeradorRelatorio.

```
01 package br.com.javamagazine.simulador.util;
02
03 import java.io.*;
04 import javax.faces.context.FacesContext;
05 import net.sf.jasperreports.engine.*;
06
07 public class GeradorRelatorio {
08
09     public void gerarRelatorio(Relatorio relatorio) {
10         gerarPDF(relatorio);
11     }
12
13     private void gerarPDF(Relatorio relatorioGenerico) {
14         try {
15             relatorioGenerico.getResponse().setHeader("Expires", "0");
16             relatorioGenerico.getResponse().setHeader("Cache-Control",
17                 "must-revalidate, post-check=0, pre-check=0");
18             relatorioGenerico.getResponse().setHeader("Pragma", "public");
19             relatorioGenerico.getResponse().setHeader("Content-Disposition",
20                 "attachment; filename=" + relatorioGenerico.getNomeRelatorio()
21                 .toUpperCase().concat(".pdf"));
22             relatorioGenerico.getResponse().setContentType("application/pdf");
23
24             OutputStream out = relatorioGenerico.getResponse().getOutputStream();
25
26             byte[] bytes = JasperExportManager.exportReportToPdf(
27                 relatorioGenerico.getImpressao());
28
29             relatorioGenerico.getResponse().setContentLength(bytes.length);
30
31             out.write(bytes, 0, bytes.length);
32             out.flush();
33             out.close();
34
35             FacesContext.getCurrentInstance().responseComplete();
36         } catch (IOException ioe) {
37             ioe.printStackTrace();
38         } catch (JRException jre) {
39             jre.printStackTrace();
40         }
41     }
42 }
```

Nossa intenção ao utilizá-lo é repassar essas informações para o módulo **ResultadoAvaliacaoMB**, onde serão avaliadas.

Dito isso, crie a classe **AvaliacaoVO** e altere o sufixo do package para **vo**. Concluído este processo, modifique o código para ficar igual ao da **Listagem 17**.

Note que na linha 16 inicializamos a data atual da avaliação. Desta forma não precisamos nos preocupar em configurá-la via método auxiliar. Outro detalhe importante é o método **getTempoFormatado()**, que servirá para retornar o tempo gasto na realização do simulado, em formato apropriado a ser exibido pela página de resultados.

O gerenciamento de memória em aplicações Java fica a cargo da JVM e normalmente não nos preocupamos com ele durante o desenvolvimento. Apenas em situações específicas nos deparamos com tal necessidade e quando isso acontece, devemos ter muita cautela antes de optar por interferir na gerência da memória. Visando facilitar esses momentos, vale analisar a adoção da virtualização e da paginação para casos em que processamos grandes quantidades de informações e não temos memória heap suficiente para armazená-las.

Listagem 17. Código da classe AvaliacaoVO.

```
01 package br.com.javamagazine.simulador.vo;
02
03 import java.io.Serializable;
04 import br.com.javamagazine.simulador.domain.Avaliacao;
05 import br.com.javamagazine.simulador.util.Cronometro;
06
07 public class AvaliacaoVO implements Serializable {
08
09     private static final long serialVersionUID = -1624741886719509697L;
10     private Pergunta pergunta;
11     private Avaliacao avaliacao;
12     private String dataAvaliacao;
13     private Long tempoAvaliacao;
14
15     public AvaliacaoVO() {
16         this.dataAvaliacao = Cronometro.obterDataAtualAvaliacao();
17     }
18
19     public String getTempoFormatado() {
20         return Cronometro.formatarTempoAvaliacao(getTempoAvaliacao());
21     }
22
23     // gets e sets omitidos
24 }
```

Ademais, para obter um melhor aproveitamento dos recursos computacionais é importante equilibrar ao máximo o uso da memória e do processamento, caso contrário corremos o risco de comprometer significativamente o desempenho da aplicação.

Autor



Marcos Vinícius Turisco Dória

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



Links:

Página de download do iReport.

<http://community.jaspersoft.com/project/ireport-designer/releases>

Página de download do Eclipse Luna.

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

Página de download do Tomcat.

<http://tomcat.apache.org/download-70.cgi>

Página de download do MySQL.

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

JSF Reference Documentation.

<http://www.javaserverfaces.org/documentation>

PrimeFaces Reference Documentation.

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

PrimeFaces Components.

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

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!





DEVMEDIA

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior
portal para
desenvolvedores
da América
Latina!



20
mil
posts

430
mil
cadastrados

10
milhões de
page-views
por mês

Primeiros passos na programação reativa com Vert.x

Conheça esta poderosa plataforma para desenvolvimento de aplicações web e serviços

OVert.x é um projeto independente, hospedado atualmente pela Eclipse Foundation e com contribuições feitas por vários indivíduos e organizações, sendo Tim Fox seu principal desenvolvedor e fundador. Conforme definição do próprio Tim Fox, o Vert.x é uma plataforma poliglota para desenvolvimento de aplicações que se beneficiam do modelo moderno de aplicações reativas e assíncronas. As suas principais características são destacadas a seguir:

- **Poliglota:** é possível criar diversos componentes para sua aplicação em diferentes linguagens. Atualmente suportando Java, JavaScript, Groovy, Ruby e Python, mas com outras linguagens como Scala, Closure e PHP em estágios de desenvolvimento;

- **Reativo e assíncrono:** a base do desenvolvimento com Vert.x é feita utilizando-se APIs assíncronas e não bloqueantes. Este modelo facilita a escalabilidade através do uso de múltiplas instâncias dos componentes da aplicação, modelo este dificultado quando a aplicação adota uma abordagem bloqueante. Esta proposta de desenvolvimento é baseada no conceito de Reactive Programming ou Programação reativa (vide **BOX 1**);

- **Modelo de concorrência baseado em Atores (Actors):** as unidades de código escritas para serem executadas no Vert.x, conhecidas como *verticles*, são fundamentalmente single threaded. Isso significa que cada verticle é executado sempre em uma thread específica, consequentemente reduzindo a complexidade quando comparado com a computação concorrente tradicional, dado que toda a necessidade de sincronização e locks entre threads simplesmente se torna desnecessária;

Fique por dentro

Este artigo é útil por apresentar, na teoria e na prática, as principais características, conceitos e detalhes da plataforma Vert.x, solução que tem como principais pilares a simplicidade no desenvolvimento, escalabilidade e assincronicidade através da sua natureza reativa, suportando múltiplas linguagens de programação, como Java, JavaScript, Ruby, entre outras. Uma das principais vantagens da plataforma Vert.x é sua versatilidade para o desenvolvimento de aplicações web e serviços sem forçar uma arquitetura pré-estabelecida, porém ao mesmo tempo fornecendo todo um ferramental bastante produtivo.

- **Plataforma para aplicações:** o Vert.x é uma plataforma genérica para uso geral que pode ser utilizada para diversos tipos de aplicação, compreendendo desde aplicações web, com destaque especial para as aplicações web em ‘tempo real’, quanto tradicionais aplicações back end.

BOX 1. Programação Reativa

A programação reativa pode ser definida como sendo a possibilidade de se desenvolver uma aplicação baseado em um fluxo assíncrono de dados. Em linhas gerais, a aplicação deve responder a eventos e estímulos assim que estes ocorrerem. Estes estímulos podem ser desde um simples clique em um botão, a requisições HTTP ou alterações em arquivo em disco

Do ponto de vista arquitetural, percebe-se que o Vert.x compartilha muito da abordagem adotada por frameworks e plataformas como Akka e Erlang/OTP, inspirando-se em conceitos neles implementados, como actors e seus sistemas de eventos assíncronos.

O que é um Verticle?

Um verticle é uma unidade executável de código em uma instância do Vert.x e pode ser escrito em quaisquer das linguagens suportadas e já mencionadas. Uma particularidade interessante é a possibilidade de vários verticles poderem ser executados concorrentemente em uma mesma instância do Vert.x, independente destes terem sido implementados em uma mesma linguagem ou não.

Outro aspecto de destaque é que uma aplicação pode ser – e geralmente é – composta por mais de um verticle e estes podem estar disponíveis em diferentes nós da rede, comunicando-se entre si através do event bus do Vert.x, conforme veremos mais adiante neste artigo.

Para aplicações mais simples, como exemplos, testes ou provas de conceito simplistas, verticles podem ser organizados unitariamente e executados diretamente na linha de comando, mas quando falamos em aplicações reais, geralmente os verticles são empacotados utilizando o conceito de módulos.

Configurando seu ambiente de desenvolvimento

Para os exemplos e testes apresentados neste artigo, iremos utilizar além do Vert.x 2.1.5, o Java 8 update 31 e o NetBeans IDE 8.0.2. O primeiro passo para a codificação destes exemplos é fazer o download destas ferramentas nos seus respectivos websites, todos disponíveis na seção **Links**. Feito isso, realize a instalação do Java e do NetBeans.

Para a instalação do Vert.x, basta descompactar a versão baixada na pasta de sua escolha e colocar a pasta *bin* na variável de ambiente *PATH* do seu sistema operacional. Para mais detalhes sobre a instalação, o guia de instalação oficial do Vert.x pode ser consultado (veja o endereço indicado na seção **Links**). Para testar a instalação, execute no console/terminal o comando *vertx version* e algo similar a “2.1.5 (built 2014-11-13 15:15:56)” deve ser impresso.

Vamos então criar nosso primeiro Verticle de exemplo. Para isso, crie um arquivo chamado *Server.java* em qualquer pasta no seu sistema. Como sugestão, vamos assumir que este foi criado na pasta de usuário padrão do sistema, por exemplo: */home/jm/Server.java*. Logo após, adicione o código da **Listagem 1** no mesmo.

A partir desse código podemos executar nosso primeiro verticle. Portanto, abra o terminal ou o prompt de comando do seu ambiente e dirija-se para a pasta onde o verticle foi salvo (no nosso caso, */home/jm*). Agora, basta executar o comando:

```
cd /home/jm  
vertx run Server.java
```

Deste modo, a mensagem *Succeeded in deploying verticle* deve ser exibida no console/terminal. Agora abra o navegador, entre com o caminho do nosso servidor, *http://localhost:8080*, e observe que a mensagem ‘Olá do Vert.x!’ deve ser exibida.

Vale destacar que não foi necessário executar o passo comum de compilação de uma classe Java, pois este processo é realizado diretamente pelo Vert.x.

Listagem 1. Exemplo de verticle chamado Server. Iniciará um servidor HTTP em localhost:8080.

```
import org.vertx.java.core.Handler;  
import org.vertx.java.core.http.HttpServerRequest;  
import org.vertx.java.platform.Verticle;  
  
import java.util.Map;  
  
public class Server extends Verticle {  
  
    public void start() {  
        vertx.createHttpServer().requestHandler(requisicao -> {  
            requisicao.response().headers().set("Content-Type", "text/html");  
            requisicao.response().end("<html><body><h1>Olá do Vert.x!</h1>  
            </body></html>");  
        }).listen(8080, "localhost");  
    }  
}
```

Este modo de execução a partir do código fonte é bastante interessante durante o processo de desenvolvimento, entretanto o cenário mais comum para execução de uma aplicação é a partir dos códigos já compilados.

Os objetos container e vertx

Toda instância de verticle dispõe de variáveis membro auxiliares, denominadas **container** e **vertx**. Estes objetos têm por objetivo dar acesso a recursos da própria plataforma Vert.x.

A variável **container** representa a porta de entrada ao container no qual o verticle está sendo executado. Por ela é possível acessar as configurações, variáveis de ambientes e log, além de permitir o *deploy* e *undeploy* do verticle programaticamente. A seguir, temos um exemplo de como realizar o log de alguma atividade:

```
final Logger logger = container.logger();  
logger.info("Logando uma atividade qualquer");
```

Já o objeto **vertx** é responsável por permitir o acesso à API principal do Vert.x. É por este objeto que é feito o acesso a recursos como TCP, HTTP, ao sistema de arquivos, ao event bus, timers, etc. Em nosso primeiro exemplo, apresentado na **Listagem 1**, fizemos acesso a este objeto para criarmos nosso servidor HTTP.

Finalização de um verticle

Vale ressaltar que todos os recursos controlados pelo Vert.x, como servidores, clientes, timers e manipuladores (*handlers*) de event bus serão corretamente encerrados e/ou cancelados quando seus respectivos verticles forem finalizados. Entretanto, se sua aplicação tiver inicializado algum outro recurso que necessite ser devidamente finalizado junto com o verticle, como uma conexão JDBC, escrita ou leitura de arquivo, entre outros, basta sobrescrever o método **stop()** em seu verticle. Este método é sempre invocado durante o processo de finalização de um verticle.

Nota

O conceito de manipuladores ou Handlers, como tratados na documentação oficial, é algo muito presente no Vert.x. Sua ideia básica é parametrizar comportamento, algo muito presente na programação funcional e reativa. Como a arquitetura do Vert.x é baseada em eventos, quando desejamos ser informados ou tomar alguma ação quando algum evento ocorre, um manipulador deve ser registrado. Convertendo em termos da linguagem Java, isso significa que uma classe interna anônima, ou no Java 8 uma expressão Lambda, deve ser passada como parâmetro em métodos que esperam estes manipuladores. Para os desenvolvedores JavaScript, este conceito é conhecido como callbacks.

Concorrência

Como a própria plataforma do Vert.x garante que cada instância de um verticle não será executada em mais de uma thread concurrentemente, o desenvolvedor não precisa se preocupar com a responsabilidade de controlar a execução concorrente da aplicação. Profissionais acostumados a trabalhar com o modelo tradicional de sincronização de execução concorrente, onde o controle da concorrência é feito de forma manual, sabem o quanto desafiador e propenso a erros este tipo de abordagem pode ser. Logo, o modelo adotado pelo Vert.x acaba se tornando mais simples e atrativo para o projeto de aplicações que exigem essa característica.

Como destacado anteriormente, os verticles são fundamentalmente **single threaded**. Isso significa que a concorrência de uma aplicação no Vert.x se dá pelo fato de que múltiplas instâncias de um verticle são executadas e trocam mensagens concorrentemente, e não pelo fato de uma instância de um verticle ser executada em múltiplas threads concorrentemente. A comunicação entre estas instâncias de verticles se dá através de mensagens utilizando o seu event bus.

Para instruir o Vert.x a executar mais de uma instância de determinado verticle, utiliza-se o parâmetro **instances**, como no exemplo a seguir:

```
vertx run Server.java -instances 10
```

Event Loop

Os event loops são threads que ficam monitorando a existência de eventos pendentes a serem consumidos.

Sempre que um evento ocorrer, o event loop é responsável por notificar o manipulador registrado. Por exemplo, como demonstrado na [Listagem 1](#), o tratamento da requisição HTTP recebida é feito através do manipulador registrado. Assim, toda vez que uma nova requisição for recebida pelo contêiner do Vert.x, este manipulador será chamado. Outros exemplos de eventos são dados recebidos em um socket, requisição UDP ou mesmo mensagens compartilhadas através do Event Bus, como veremos mais adiante.

Uma característica do Vert.x é que cada instância de um verticle é atribuída, por padrão, a um event loop específico no momento de seu deploy e todo evento que ocorrer a partir deste momento será sempre despachado fazendo uso da mesma thread. Para evitar uma sobrecarga no hardware, geralmente o número de event loops limita-se à quantidade de núcleos da CPU do ambiente de execução.

Obviamente, como centenas ou milhares de instâncias de verticles podem estar em execução no mesmo momento, uma mesma thread poderá ser responsável por executar inúmeras instâncias de verticles. Apesar disso, é garantido que cada instância de verticle estará sempre atrelada à mesma thread. Esta premissa nos leva a uma regra fundamental no Vert.x: Nunca bloqueie o event loop!

Como cada event loop será potencialmente responsável por atender a inúmeras instâncias de verticles, conforme exemplificado na **Figura 1**, todo e qualquer atraso ou paralisação será sentido pela aplicação e no caso do event loop ser bloqueado, nenhum outro evento será tratado. Consequentemente, nenhum outro handler será notificado e a aplicação ficará travada. Por este motivo é extremamente importante que nenhuma operação longa seja executada utilizando-se dos verticles executados em um event loop.

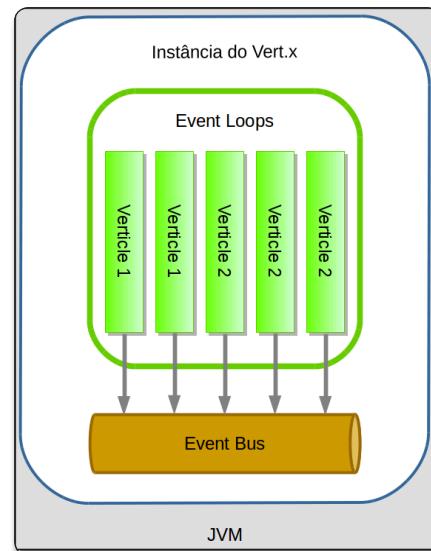


Figura 1. Instância do Vert.x com verticles instalados

Para suportar operações potencialmente bloqueantes, a plataforma do Vert.x disponibiliza uma opção específica para este fim, chamada Worker Verticles.

Basicamente, bloquear o event loop significa executar qualquer operação que possa travar ou ocupar a thread em execução. Algumas das operações que não devem ser executadas no event loop são:

- `Thread.sleep();`
- `Object.wait();`
- Operações bloqueantes do pacote `java.util.concurrent`, como `CountDownLatch.await();`
- Uso indiscriminado de estruturas de repetição como loops;
- Executar longas operações, como cálculos custosos;
- Uso de APIs ou operações bloqueantes, como o uso de IO intensivo ou queries SQL via JDBC.

Operações bloqueantes com Worker Verticles

Como discutido anteriormente, o modelo padrão para verticles é ser totalmente assíncrono e não bloqueante, porém existem

vários cenários onde operações de computação intensa ou mesmo bloqueantes são necessárias, como cálculos e processamentos complexos ou mesmo uma chamada à API JDBC tradicional. Para estes cenários é possível executar verticles específicos em um pool de threads apropriadas, chamados *worker pool*.

Seguindo a mesma linha dos verticles padrão, as instâncias dos worker verticles nunca são executadas concorrentemente por mais de uma thread. Entretanto, diferentemente das instâncias dos verticles padrão, que são sempre executadas em uma mesma thread, as instâncias dos worker verticles podem ser executadas em threads diferentes.

Para que um verticle seja executado em modo worker, o parâmetro *-worker* deve ser informado no momento da execução, conforme o exemplo a seguir:

```
vertx run Server.java -instances 10 -worker
```

Neste caso, estamos executando o exemplo da **Listagem 1** com 10 instâncias do verticle em modo worker.

Nota

É importante estar ciente que operações bloqueantes são difíceis de escalar e consequentemente dificultam o desenvolvimento de sistemas que precisam atender a muitas conexões concorrentes.

Event Bus

O event bus do Vert.x, como definido pela própria documentação oficial, pode ser considerado como o sistema nervoso da plataforma. Ele serve como um canal de comunicação entre distintas instâncias de verticles, independente da thread de execução ou mesmo da linguagem utilizada na implementação dos mesmos.

Endereçamento

Essencialmente, as mensagens a serem enviadas de um verticle são sempre vinculadas a algo conhecido como endereço. Estes endereços, como veremos, são bastante simples, sendo representados apenas por uma **String**, sem quaisquer restrições de caracteres ou padrão pré-determinado. Alguns exemplos de endereços válidos são: 'X', 'abc', 'endereco.mensagem.a' e 'endereco.mensagem.b'.

No entanto, mesmo qualquer **String** sendo considerada válida pela plataforma, o recomendado é que algum tipo de padrão seja adotado. Isso facilita a manutenção e organização destes endereços conforme a aplicação sendo desenvolvida cresce. O indicado é que alguma forma de namespace seja empregado. Algo similar ao adotado em pacotes de classes Java ou mesmo URLs. Alguns exemplos de recomendação para endereços são: 'jm.artigos' e 'jm.autores'.

Manipuladores

Os manipuladores, chamados na API de handlers, são responsáveis por processar as mensagens recebidas em um determinado endereço. É permitido tanto que mais de um manipulador seja

registrado para um mesmo endereço, quanto um verticle registrar manipuladores em vários endereços.

Padrão publish/subscribe de mensageria

Este padrão segue o modelo também conhecido como tópico. O seu propósito é que todos os manipuladores registrados para um determinado endereço sejam notificados sempre que uma mensagem for publicada em um endereço. Podemos entender este modelo como algo similar a um broadcast.

Padrão ponto a ponto de mensageria

Quando uma mensagem é enviada a um endereço, apenas um dos manipuladores será invocado. Caso mais de um manipulador esteja registrado, apenas um deles será selecionado, aplicando para isso o algoritmo *non-strict round-robin*. Em termos gerais, são atribuídas frações de tempo para cada processo em partes iguais e de forma circular, manipulando todos os processos sem prioridade definida.

Este padrão também permite que um manipulador de resposta seja registrado. Esta opção possibilita que, quando necessário, o responsável pela geração da mensagem possa se comunicar com o manipulador que for consumir a mensagem. A proposta é que, quando a resposta for recebida pelo consumidor, exista a chance deste consumidor se comunicar com o manipulador que originou a mensagem. A API atual possibilita ainda que este canal de comunicação seja utilizado por tempo indeterminado. Em outras palavras, este canal pode continuar ativo enquanto o manipulador que gerou a mensagem e seu manipulador consumidor necessitarem trocar mensagens. Este padrão é conhecido também como request-response.

Módulos

O Vert.x também oferece suporte a módulos, possibilitando às aplicações desenvolvidas serem componentizadas e consequentemente melhor organizadas e distribuídas. Alguns dos benefícios de se utilizar módulos são:

1. Encapsulamento do classpath para facilitar a execução da aplicação;
2. Dependências encapsuladas diretamente no artefato zip dos módulos;
3. Publicação de módulos diretamente no repositório Maven ou Bintray;
4. Possibilidade de catalogar módulos no registro oficial de módulos do Vert.x;
5. Download automático de módulos.

A recomendação é que as aplicações sejam distribuídas em forma de módulos e não puramente em verticles. Este último formato deve ser restrito apenas para prototipação, exemplos e pequenos testes.

Arquitetura

O modelo arquitetural do Vert.x é bastante simples e idealizado para possibilitar a criação de aplicações sem forçar um modelo de

desenvolvimento pré-definido, diferentemente de frameworks, que costumam definir o fluxo e eventualmente até o modelo arquitetural das aplicações.

Na **Figura 2** temos um exemplo de arquitetura de uma aplicação em execução na plataforma Vert.x.

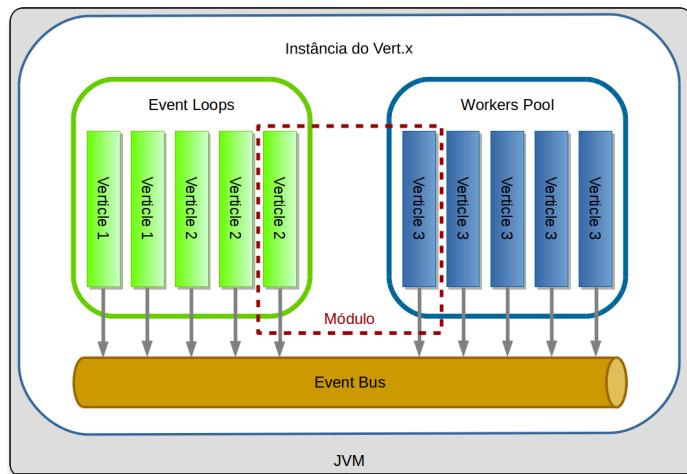


Figura 2. Visão geral da arquitetura do Vert.x

Nesta imagem destacamos a existência de vários verticles sendo executados por seus event loops, assim como worker verticles no seu respectivo pool. Podemos ainda identificar que a comunicação entre estes verticles é realizada através do event bus. Outro ponto de destaque é que uma aplicação pode ter seus verticles organizados em módulos distintos.

Isso significa que uma aplicação devidamente componentizada pode ter seus módulos executados em uma mesma instância do Vert.x, com seus verticles devidamente distribuídos em vários event loops e ainda seus workers corretamente gerenciados pelo Worker Pool e todos estes se comunicando através de um event bus comum.

APIs de destaque

Neste tópico analisaremos algumas APIs adicionais relevantes da plataforma Vert.x e que servem de complemento para as APIs já apresentadas.

Acesso ao sistema de arquivos local

Grande parte das aplicações necessita de algum tipo de acesso ao sistema de arquivos local, seja para disponibilizar certo tipo de conteúdo estático, como imagens, ou mesmo para fazer a persistência de algum registro, como logs. Para esse requisito o Vert.x disponibiliza uma API específica, referenciada eventualmente como File System API. Com ela, podemos realizar várias operações com arquivos e diretórios, como copiar, mover, excluir, alterar permissões, criar e remover links *hard*s e simbólicos, entre outras.

A interface de acesso ao sistema de arquivos se dá através do método `fileSystem()` do objeto `vertx`, disponível no verticle. Na **Listagem 2** temos um exemplo que realiza a cópia de arquivos.

Listagem 2. Exemplo que realiza a cópia de arquivos.

```
vertx.fileSystem().copy("origem.dat", "destino.dat", resultado -> {
    if (resultado.succeeded()) {
        logger.info("Cópia realizada com sucesso");
    } else {
        logger.error("Falha durante processo de cópia", resultado.cause());
    }
});
```

Superação de WebSockets e SockJS

Com o advento de aplicações cada vez mais responsivas e usuários cada vez mais exigentes, tecnologias para comunicação contínua entre cliente e servidor foram desenvolvidas e homologadas nos últimos anos, permitindo que clientes, normalmente navegadores, tenham uma conexão permanente com o servidor de origem. Um destes exemplos é a tecnologia WebSocket, que permite a comunicação full-duplex entre as pontas. Esta tecnologia é bastante recente e nem todo framework ou plataforma já apresenta suporte a tal recurso, entretanto o Vert.x suporta WebSockets, tanto para a implementação de servidores WebSocket quanto para consumi-los através de sua API cliente.

Um dos inconvenientes do WebSocket é que, por se tratar de uma tecnologia relativamente nova, nem toda aplicação cliente ainda o suporta. Um exemplo são alguns ambientes corporativos mais restritivos, onde apenas versões homologadas de navegadores são permitidas e nem toda versão mais atual destes navegadores está disponível. Outro exemplo são restrições de proxies que impedem que este tipo de tecnologia seja aplicável. São nestes momentos que o SockJS se apresenta como uma alternativa.

Esta biblioteca JavaScript tem o objetivo de suportar diferentes estratégias de comunicação entre cliente e servidor. Como primeira opção, ela tenta realizar a comunicação com WebSocket nativo, mas se esta opção não for bem-sucedida, outras estratégias, como APIs específicas de cada navegador, são utilizadas. Assim, WebSocket passa a ser apenas mais uma opção, possibilitando que nós desenvolvedores façamos uso apenas de uma simples biblioteca como o SockJS, e esta por sua vez abstraia o protocolo real que está sendo empregado.

No tópico a seguir criaremos uma simples aplicação que fará uso de SockJS para viabilizar a comunicação entre clientes. Como não iremos nos aprofundar na API do SockJS, para mais detalhes sobre esta tecnologia, recomendamos o site do desenvolvedor, informado na seção [Links](#).

Aplicação de exemplo SockJS

Para criar o exemplo, no IDE NetBeans, acesse o menu *Arquivo > Novo Projeto*, selecione a categoria *Maven* e então *Projeto do Arquétipo* e clique em *Próximo*. Feito isso, pesquise por 'vertx', selecione o arquétipo *vertx-maven-archetype* e clique novamente em *Próximo*. No momento da escrita deste artigo a versão disponível é a 2.0.7-final.

Na próxima tela mais informações serão requeridas. Para o nome do projeto, informe "*vertx-sockjs-exemplo*" e o coloque em uma

pasta de sua escolha. Para o ID de grupo deve ser informado o valor “jm”, e o pacote padrão como **jm.sockjs**.

Nosso próximo passo será limpar o projeto dos artefatos que não precisaremos. Neste momento você pode estar se perguntando o motivo de tantos artefatos, como testes, serem gerados neste arquétipo. A resposta é simples: este é o arquétipo oficial do Vert.x e tem por objetivo não apenas criar um projeto novo, mas já vir com um pequeno exemplo de verticle e como testá-lo nas mais diferentes linguagens suportadas pela plataforma. Desta forma, dependendo da linguagem do seu projeto, você já pode fazer uso de alguns destes artefatos. No nosso cenário, vamos eliminar o pacote completo de testes, localizado sob os nós *Pacotes de Teste* e *Outros Códigos-fonte de testes* na aba *Projeto* do NetBeans. Além destes pacotes, vamos remover ainda, sob o nó *Outros Códigos fontes/src/main/resources*, os arquivos *GroovyVerticle.groovy*, *Java-SourcePingVerticle.java*, *ping_verticle.js* e *ping_verticle.py*.

Agora vamos configurar nosso verticle. Para isso, iniciaremos renomeando o verticle existente sob o nó *Pacotes de Códigos-fonte*, de **PingVerticle** para **App**. Em seguida, precisamos configurar este verticle como sendo o principal da aplicação, aquele que é chamado quando a executamos. Deste modo, em *Outros Códigos fontes/src/main/resources*, abra o arquivo *mod.json*. Este contém as principais propriedades de um módulo Vert.x, conforme veremos a seguir. Para configurar o verticle principal da nossa aplicação, defina o valor da propriedade **main** para corresponder à nossa classe **App**. A configuração resultante deve ser “**main**”: “**jm.sockjs.App**”. O restante do arquivo pode ficar inalterado, mas a título de curiosidade, pode-se visualizar neste mesmo arquivo como seria a configuração de verticles principais em outras linguagens, além de outras propriedades, como o autor do módulo e sua descrição.

Como o Vert.x ainda não fornece plugins ou configurações especiais para IDEs, apenas uma excelente integração com o Maven através de *goals* específicos da sua própria plataforma, nosso objetivo agora é facilitar nosso ciclo de desenvolvimento dentro do IDE NetBeans criando um atalho para execução destes *goals* do Vert.x de forma direta. Deste modo, no NetBeans, aba *Projetos*, clique com o botão direito do mouse sobre nosso projeto e depois em *Propriedades*. Após selecionar a categoria *Ações*, procure pela ação *Executar projeto* e em *Executar Metas* digite “install vertx:runMod”, remova todas as propriedades definidas e confirme a alteração. Caso você deseje executar o projeto na linha de comando, basta chamar o comando *mvn install vertx:runMod* e tudo deverá funcionar corretamente. Realizadas as configurações, nosso projeto está totalmente funcional e pronto para execução.

Deste modo, podemos iniciar o desenvolvimento propriamente dito do nosso exemplo, começando pela criação do arquivo *index.html* com o conteúdo da **Listagem 3** (na pasta *Outros Códigos fontes/src/main/resources/sockjs*). Este HTML apresenta um campo onde o usuário pode escrever uma mensagem e enviá-la a todos os outros usuários conectados à mesma página. O objetivo é que esta mensagem seja exibida diretamente no console do navegador.

Para implementarmos a comunicação entre o servidor e os clientes, faremos uso da biblioteca SockJS. O primeiro passo neste momento é realizar a sua instalação, introduzindo no arquivo *index.html*, dentro da seção *head*, a tag `<script src="//cdn.jsdelivr.net/sockjs/0.3.4/sockjs.min.js"></script>`. Em seguida, podemos inicializar uma seção do SockJS configurando o endereço, porta e contexto da nossa aplicação no servidor. A título de exemplo, vamos estabelecer que esta será executada na máquina local, porta 8080 e com um contexto chamado *app*. Para criar uma seção SockJS devemos instanciar um objeto específico, conforme o exemplo a seguir, e armazená-lo em uma variável para uso posterior:

```
var sock = new SockJS('http://localhost:8080/app');
```

Este objeto SockJS apresenta várias funções, porém faremos uso de apenas algumas, como **onopen**, **onclose**, **send** e **onmessage**, com destaque especial para as duas últimas. As funções **onopen** e **onclose** servem para realizar verificações ou operações durante a abertura ou encerramento de uma conexão. Por sua vez, para o tratamento das mensagens recebidas, faz-se uso da função **onmessage** e para o envio de mensagens, a função **send** deve ser empregada. Na **Listagem 3** temos o código do *index.html* da nossa aplicação exemplo fazendo uso destas funções.

Listagem 3. *index.html* da aplicação exemplo.

```
<html>
<head>
<title>Exemplo com SockJS</title>
<script src="//cdn.jsdelivr.net/sockjs/0.3.4/sockjs.min.js"></script>
<meta charset="UTF-8">
</head>
<body>
<script>
var sock = new SockJS('http://localhost:8080/app');
sock.onopen = function () {
    console.log('socket aberto.');
};
sock.onmessage = function (e) {
    console.log('mensagem', e.data);
};
sock.onclose = function () {
    console.log('socket fechado.');
};

function enviar(mensagem) {
    if (sock.readyState === SockJS.OPEN) {
        console.log("enviando mensagem...");
        sock.send(mensagem);
        console.log("mensagem enviada.");
    } else {
        console.log("O socket não está aberto.");
    }
}
</script>
<form onsubmit="return false;">
<input type="text" name="mensagem" value="Olá, Java Magazine!" />
<input type="button" value="Enviar mensagem" onclick="enviar(this.form.mensagem.value)"/>
</form>
</body>
</html>
```

Vejamos agora como criar a parte servidor do nosso exemplo. Sendo assim, na pasta *Códigos-Fontes*, vamos editar o conteúdo da classe **App**. Como sabemos, este é nosso verticle principal, o qual implementará os quatro principais aspectos da nossa aplicação: o servidor de conteúdo estático, como HTMLs; configurar a conexão do SockJS; enviar as mensagens recebidas para todos os clientes conectados; e, por último, registrar todas as mensagens trafegadas em um arquivo.

A primeira necessidade a ser implementada é o servidor de conteúdo estático, que será responsável por disponibilizar o arquivo *index.html*. Para implementar isso, a partir da variável **vertx** criamos um servidor HTTP invocando o método **vertx.createHttpServer()** e, em seguida, registramos o manipulador para cada requisição que este servidor receber. Por padrão, sempre que uma requisição no contexto principal da aplicação for recebida, nosso arquivo *index.html* será retornado. Na **Listagem 4** temos o código deste servidor.

A segunda atribuição de **App** será preparar o servidor SockJS para a conexão dos sistemas clientes, representados neste cenário pelos navegadores. O código para criação e configuração deste servidor pode ser analisado na **Listagem 5**.

Listagem 4. Criação do servidor de conteúdo estático.

```
final HttpServer servidor = vertx.createHttpServer();
servidor.requestHandler((HttpServerRequest requisicao) -> {
    if (requisicao.path().equals("/")) {
        requisicao.response().sendFile("sockjs/index.html");
    }
});
```

Listagem 5. Criação do servidor SockJS.

```
final SockJSHandler sockServer = vertx.createSockJSHandler(servidor);
sockServer.installApp(new JsonObject().putString("prefix", "/app"),
(final SockJSocket socket) -> {
    socket.dataHandler((Buffer mensagem) -> {
        // Publica a mensagem recebida no event bus
        eventBus.publish(CANAL_MENSAGEM, mensagem);
    });

    // Registra o manipulador do event bus apropriadamente
    eventBus.registerHandler(CANAL_MENSAGEM, new SockJSMessageHandler(
        socket));

    // Tratamento de eventuais exceções
    socket.exceptionHandler((Throwable event) -> {
        logger.error("Erro em conexão com o socket.", event);
    });
});
```

Como verificado, nossa primeira ação foi criar um servidor SockJS a partir do método **createSockJSHandler()** do objeto **vertx** e guardar este servidor na variável **sockServer**.

Para a correta configuração e registro deste servidor SockJS, invocamos o método **installApp()** de **sockServer** informando alguns parâmetros importantes. Primeiramente, temos que informar o nome do contexto ao qual este servidor irá responder, o que é realizado utilizando-se a propriedade **prefix** na configuração do servidor.

Tal parâmetro é obrigatório e é adotado nas definições das regras internas de roteamento das requisições SockJS recebidas pelo servidor, fazendo com que cada servidor SockJS receba apenas as mensagens do seu contexto. Este mesmo contexto é definido na parte cliente da nossa aplicação, no momento que criamos o objeto **SockJS**, conforme visto na **Listagem 3**. Em nosso exemplo, o contexto será definido como */app*. O segundo parâmetro do método **installApp()** é o manipulador responsável por tratar cada socket aberto pelos clientes.

A terceira responsabilidade do nosso **App** é enviar as mensagens recebidas a todos os sockets conectados. Em outras palavras, uma espécie de broadcast. Para isso, vamos aplicar um manipulador no event bus, conforme o código a seguir:

```
eventBus.registerHandler(CANAL_MENSAGEM, new SockJSMessageHandler(socket));
```

O manipulador **SockJSMessageHandler** tem o objetivo de receber as mensagens repassadas ao event bus e então redirecioná-las ao socket informado como parâmetro na sua construção. Isso fará com que todos os sockets sejam notificados com o conteúdo de todas as mensagens recebidas pelo event bus no canal de mensagem informado, independente do socket de origem destas mensagens. Em outras palavras, o manipulador **SockJSMessageHandler** fará a função de uma ponte entre os sockets, apenas redirecionando as mensagens recebidas para todos os sockets informados. A implementação deste manipulador é bastante simples, dado que precisamos apenas redirecionar o conteúdo das mensagens entre os sockets (vide **Listagem 6**).

Outro detalhe importante de uma aplicação são os tratamentos de erros. O Vert.x possibilita registrar manipuladores de exceções em vários de seus serviços, sempre seguindo uma receita similar. Tratamos possíveis erros no nosso socket registrando um manipulador de exceções através do método **exceptionHandler()**, exemplificado também na **Listagem 6**. No caso de algum erro ocorrer com nosso socket, estes problemas serão devidamente registrados no log do sistema.

A última responsabilidade deste verticle será fazer o log de todas as mensagens recebidas em um arquivo na pasta do usuário. Chamaremos este arquivo de *mensagens-log.dat*. Na **Listagem 7** temos a abertura para leitura e escrita do referido arquivo no sistema local.

No caso de a abertura do arquivo ocorrer com sucesso, verificada através do método **resultado.succeeded()**, registra-se no event bus um manipulador específico para o arquivo, fazendo com que toda mensagem recebida, independente do socket, seja gravada no arquivo apropriadamente. Em caso de falha ao abrir o arquivo, o erro então será registrado no log pelo sistema.

Caso neste momento o projeto não esteja compilando, é provável que seja necessário configurar o plugin de compilação do Maven, no *pom.xml*, para suportar o Java 8. Em nosso cenário, nas linhas 159 e 160 deste arquivo, foi necessário alterar a versão da plataforma Java de 1.7 para 1.8, conforme indicado na **Listagem 8**.

Agora que temos a implementação concluída, podemos compilar e executar nosso exemplo. Para isso, com o botão direito sobre o

Listagem 6. Implementação completa de servidor HTTP e SockJS em localhost:8080.

```
package jm.sockjs;

//imports omitidos...

public class App extends Verticle {

    private static final String CANAL_MENSAGEM = "jm.mensagem";

    @Override
    public void start() {
        final Logger logger = container.logger();
        final EventBus eventBus = vertx.eventBus();
        final HttpServer servidor = vertx.createHttpServer();

        // Cria manipulador de chamadas HTTP para servir página estáticas
        servidor.requestHandler((HttpServerRequest requisicao) -> {
            if (requisicao.path().equals("/")) {
                requisicao.response().sendFile("sockjs/index.html");
            }
        });
    }

    // Cria e instala uma aplicação SockJS
    final SockJSHandler sockServer = vertx.createSockJSHandler(servidor);
    sockServer.installApp(new JsonObject().putString("prefix", "/app"),
        (final SockJSHandler socket) -> {
            socket.dataHandler((Buffer mensagem) -> {
                // Publica a mensagem recebida no event bus
                eventBus.publish(CANAL_MENSAGEM, mensagem);
            });
        });

    // Registra o manipulador do event bus apropriadamente
    eventBus.registerHandler(CANAL_MENSAGEM, new SockJSHandler(socket));

    // Tratamento de eventuais exceções
    socket.exceptionHandler((Throwable event) -> {
        logger.error("Erro em conexão com o socket:", event);
    });
}

final String pastaUsuario = System.getProperty("user.home");
// Registra o uso de arquivo no sistema de arquivos local, na pasta de usuário
vertx.fileSystem().open(pastaUsuario + "/mensagens-log.dat",
    (AsyncResult<AsyncFile> resultado) -> {
    if (resultado.succeeded()) {
        final AsyncFile arquivo = resultado.result();
        // Registra o manipulador do event bus para gravação em arquivo de log
        eventBus.registerHandler(CANAL_MENSAGEM, new ArquivoMensagemHandler(arquivo));
    } else {
        logger.error("Falha ao abrir arquivo.", resultado.cause());
    }
});

eventBus.registerHandler(CANAL_MENSAGEM,
    new ArquivoMensagemHandler(arquivo) {
        @Override
        public void handle(Message<Buffer> evento) {
            // Toda mensagem encaminhada ao event bus é gravada em arquivo
            arquivo.write(evento.body()).appendString("\n");
            arquivo.flush();
        }
    });
}

private static class ArquivoMensagemHandler implements Handler<Message<Buffer>> {
    private final AsyncFile arquivo;

    ArquivoMensagemHandler(final AsyncFile arquivo) {
        this.arquivo = arquivo;
    }

    @Override
    public void handle(Message<Buffer> evento) {
        // Toda mensagem encaminhada ao event bus é gravada em arquivo
        arquivo.write(evento.body()).appendString("\n");
        arquivo.flush();
    }
}
```

Listagem 7. Criação do arquivo de log e registro das mensagens transmitidas.

```
vertx.fileSystem().open(pastaUsuario + "/mensagens-log.dat",
    (AsyncResult<AsyncFile> resultado) -> {
    if (resultado.succeeded()) {
        final AsyncFile arquivo = resultado.result();
        eventBus.registerHandler(CANAL_MENSAGEM, new ArquivoMensagemHandler(arquivo));
    } else {
        logger.error("Falha ao abrir arquivo.", resultado.cause());
    }
});
```

Listagem 8. Configuração de plugin de compilação do Maven.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${maven.compiler.plugin.version}</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
</plugin>
```

projeto, selecione *Limpar e Construir*. Após o término desta operação, basta no menu *Executar*, selecionar a opção *Executar projeto*.

Nosso próximo passo será testar a aplicação que acabamos de executar. Para isso, abra o navegador, acesse nossa página principal – <http://localhost:8080/> – em duas abas e abra o console de cada aba no navegador. Em seguida, altere a mensagem exibida em apenas uma das abas, por exemplo, para ‘Olá, Java Magazine 2!’ na segunda aba. Logo após, pressione o botão *Enviar mensagem* em ambas as abas algumas vezes. Desta forma será possível verificar, no console de cada aba, que as mensagens de ambas as abas são recebidas. Verifique também o arquivo de log *mensagens-log.dat*, na sua pasta de usuário. Todas as mensagens devem estar registradas neste arquivo.

Deployment

Neste tópico vamos abordar algumas estratégias e boas práticas para o empacotamento e distribuição da sua aplicação Vert.x. Será tratado ainda como podemos organizar a inicialização programática de nossos verticles de forma organizada.

Coordenando a inicialização de uma aplicação

Para aplicações que contenham muitos verticles, a recomendação é que um verticle principal seja criado para coordenar a inicialização e deploy de todos os outros e eventualmente suas configurações. Este verticle é conhecido também como verticle de inicialização.

Por exemplo, pode-se criar um verticle chamado **VerticleIniciador** e em seu método **start()** inicializar três verticles fictícios. Conforme o código a seguir, seriam inicializados os verticles *verticle1.js*, escrito em JavaScript, e *foo.Verticle3* e *foo.Verticle4* escritos em Java, sendo este último inicializado como um worker verticle:

```
// Inicializando verticles
container.deployVerticle("verticle1.js");
container.deployVerticle("foo.Verticle3");
container.deployWorkerVerticle("foo.Verticle4");
```

Conforme mencionado, é possível iniciar verticles a partir da linha de comando informando os mais variados parâmetros. Para suprir esta necessidade, todos estes métodos de deploy têm sobrecargas que permitem que outras propriedades e configurações sejam informadas.

A abordagem mais comum é que os verticles sejam, de certa forma, independentes em relação à ordem de inicialização de seus serviços, promovendo assim a independência entre verticles. Porém, nos cenários em que a ordem de inicialização seja realmente importante, há sobrecargas nos métodos de deploy de modo que callbacks de notificação podem ser utilizados para sermos informados da situação destes deploys. De qualquer forma, o comportamento do Vert.x é que os verticles são inicializados na ordem de sua publicação.

Na **Listagem 9** temos um exemplo de tentativa de deploy de um verticle e então imprimimos no console se ele foi inicializado com

sucesso ou o stacktrace de um possível erro caso ocorra falha na instalação. Esta mesma abordagem poderia ser utilizada para coordenação no deploy de verticles, onde um segundo verticle somente seria inicializado se um primeiro fosse instalado com sucesso.

Listagem 9. Exemplo de inicialização de verticle programaticamente.

```
container.deployVerticle("verticle1.js", callbackRealizacaoDeploy -> {
    if (callbackRealizacaoDeploy.succeeded()) {
        System.out.println("Deploy realizado com ID " + callbackRealizacaoDeploy.result());
    } else {
        callbackRealizacaoDeploy.cause().printStackTrace();
    }
});
```

JARs executáveis

Um suporte interessante do Vert.x é a distribuição da aplicação utilizando um arquivo chamado *fat jar* ou *jar executável*. Estes jars, além de empacotarem todo o módulo e suas dependências, também contêm a própria plataforma do Vert.x. Com isso é possível distribuir estes artefatos de forma que o ambiente de destino não necessite ter a plataforma do Vert.x previamente instalada e configurada.

Para criar um JAR executável, o comando a seguir deve ser utilizado:

```
vertx fatjar <nome_módulo>
```

E para executar o JAR, basta chamar o comando tradicional: `java -jar <jar-gerado>`.

Vert.x 3

No momento da escrita deste artigo a versão mais atual do Vert.x é a 2.1.5, porém a versão 3.0 já se encontra em desenvolvimento. O principal objetivo desta nova versão, além de aprimorar vários aspectos para facilitar o desenvolvimento por parte dos usuários da plataforma, é simplificar a manutenção e evolução da própria plataforma do Vert.x. Para expor esta evolução, a seguir enumeramos algumas das principais alterações planejadas para a próxima versão.

Sem sistema de módulos próprios

A partir do Vert.x 3.0 o suporte a módulos foi totalmente remodelado e os módulos a partir desta nova versão serão apenas arquivos JARs devidamente registrados em repositórios Maven ou Bintray, como qualquer artefato Maven convencional. Desta forma, basta que as dependências sejam devidamente registradas como dependências padrão do Maven para que sejam corretamente resolvidas em tempo de build e não mais em tempo de execução, como nas versões atuais.

Outras linguagens também poderão empacotar suas dependências em arquivos JAR, mas o suporte a outras estruturas de pacotes ainda será estudado e eventualmente integrado a este

modelo. Isso possibilitaria que outras linguagens fizessem uso de seus empacotamentos padrão e não necessariamente depender do empacotamento Java tradicional, o JAR.

Geração de código para outras linguagens

A inclusão e manutenção de uma nova linguagem na atual arquitetura do Vert.x tem um custo relativamente alto, principalmente por ser um trabalho extremamente manual e não trivial. Dado que o time de manutenção é bastante reduzido, manter várias linguagens pode ser um fator que limite a inclusão e manutenção de muitos recursos na plataforma.

O objetivo nesta nova versão é que apenas a plataforma desenvolvida com a linguagem Java seja mantida diretamente pelos desenvolvedores, e que o suporte a outras linguagens seja gerado automaticamente por um novo sistema de geração de código. Assim, novas linguagens poderão ter seu suporte implementado e mantido de forma mais simplificada, reduzindo o custo de manutenção da própria plataforma.

Java 8 como versão mínima

A nova versão do Vert.x obrigará que o Java 8 ou superior seja utilizado. Isso se dará porque serão empregados recursos disponibilizados apenas a partir desta versão da plataforma Java para a implementação do Vert.x. Os destaques ficam para a utilização e suporte das expressões lambda e a adoção do motor Nashorn como padrão para a execução de códigos JavaScript.

Adoção do Maven como build padrão

O projeto Vert.x teve sua estrutura de construção migrada para o Maven. No entanto, isso não irá, de forma alguma, forçar que os projetos criados com Vert.x também tenham que fazer uso do Maven. É permitido que qualquer sistema de construção seja utilizado sem restrições, como é o caso do Gradle.

O Vert.x apresenta inúmeras características interessantes, sendo uma das principais sua abordagem orientada a eventos totalmente assíncrona. Este paradigma, bastante difundido sob a alcunha de programação reativa, tenta acomodar a necessidade crescente de aplicações que consigam atender a um número cada vez maior de usuários e conexões, possibilitando o desenvolvimento de soluções mais performáticas e com custos reduzidos de manutenção.

Outro grande destaque do Vert.x é sua completa e atualizada documentação, bem como a grande quantidade de módulos desenvolvidos e disponibilizados pela comunidade, o que o torna bastante ativo. Como resultado de tudo isso, em 2014 o Vert.x

recebeu o JAX Innovation Awards na categoria de tecnologia Java mais inovadora.

Esta plataforma vem com uma proposta de não ser um framework ou determinar um fluxo de desenvolvimento padrão. O objetivo do projeto é fornecer recursos para que o paradigma de desenvolvimento adotado seja viável e ao mesmo tempo deixar os desenvolvedores e arquitetos livres para projetar suas aplicações da forma que lhes for adequada.

Autor



Michel Graciano

@mgraciano

Graduado em Sistemas de Informação pela UNISUL e com mais de 12 anos de experiência em desenvolvimento de software com a plataforma Java, trabalha atualmente como Arquiteto de Sistemas para a Beta Sistemas e instrutor na Código Efetivo. É também membro do Grupo de Usuários Java de Santa Catarina - GUJavaSC - e colaborador ativo de projetos open source como o NetBeans. Já fez apresentações em diversos eventos como JavaOne USA e Brasil, TDC Floripa e JustJava.



Links:

Códigos-fonte e exemplos referentes ao artigo.

<https://github.com/mgraciano/jm-vert.x>

Site oficial do projeto Vert.x.

<http://vertx.io/>

Download do Vert.x 2.1.5.

<http://vertx.io/downloads.html>

Guia de instalação do Vert.x.

<http://vertx.io/install.html>

SockJS website.

<https://github.com/sockjs/sockjs-client>

Registro oficial de módulos do Vert.x

<http://modulereg.vertx.io/>

Download da versão mais atual do Java 8.

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

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



PrimeFaces 5.1 na prática

Aprenda a utilizar os novos componentes de interface do PrimeFaces 5.1

O JavaServer Faces é um dos principais frameworks MVC disponíveis no mercado, mas a sua especificação disponibiliza apenas componentes básicos para criação de interfaces, como caixas de texto e tabelas para entrada e saída de dados. Para melhorar o desenvolvimento de aplicações JSF foram elaboradas diversas bibliotecas de componentes que simplificam a construção de interfaces ricas. Dentre essas bibliotecas, podemos destacar o RichFaces, o ICEfaces e, principalmente, o PrimeFaces.

A principal vantagem do PrimeFaces é a enorme quantidade de componentes disponíveis. Além disso, o framework, cuja versão 5.1 foi lançada em outubro de 2014, possui uma comunidade bastante ativa. Nessa versão, além das melhorias esperadas, foram adicionados componentes que podem ser bastante úteis para aplicações em diversos domínios, como é o caso do componente *Ribbon*, que facilita a criação de barras de ferramentas, e o *BarCode*, que possibilita a criação de códigos de barras em diversos padrões.

Dito isso, este artigo irá apresentar os principais componentes do PrimeFaces e os destaques da versão 5.1. Para tal, veremos como instalar e configurar o ambiente de desenvolvimento para um projeto que inclua PrimeFaces, revisar os componentes de uma aplicação web simples e, finalmente, abordar como aplicar os novos componentes para proporcionar uma melhor experiência ao usuário.

Instalação e configuração do ambiente de desenvolvimento

Para o desenvolvimento da nossa aplicação vamos adotar o Eclipse e o Maven. Obviamente poderíamos apresentar os exemplos sem essas ferramentas, mas o uso de uma IDE e de um gerenciador de dependências facilita bastante o processo de codificação. Além disso, é necessário ter acesso – local ou remoto – a um servidor que implemente a especificação do JSF; neste caso, optamos pelo Tomcat (também poderíamos utilizar

Fique por dentro

Esse artigo é útil para estudantes e profissionais que tenham interesse em adotar a nova versão do PrimeFaces em projetos que utilizam o JavaServer Faces (JSF). Pensando nisso, serão apresentados os novos componentes do PrimeFaces – como o *Ribbon*, *BarCode* e *InputSwitch* – e as melhorias realizadas em alguns componentes antigos. Além disso, analisaremos o Grid CSS, um conjunto de classes CSS do PrimeFaces para facilitar o desenvolvimento de interfaces responsivas, que se adaptam automaticamente de acordo com o dispositivo através do qual são acessadas.

servidores de aplicação como GlassFish, WildFly, entre outros).

O primeiro passo é baixar (veja a seção **Links**) e instalar essas ferramentas. Para a instalação das mesmas, basta descompactar os arquivos e colocá-los em qualquer diretório de sua preferência. Em seguida é preciso adicionar o Tomcat ao Eclipse (o que é feito acessando a aba *Servers* desta IDE) clicando em *Add Server* e selecionando o caminho da instalação deste servidor web. Desse modo é possível executar os exemplos que serão apresentados no Tomcat diretamente através do Eclipse.

Com todas as ferramentas instaladas, podemos criar um projeto Maven no Eclipse e incluir as dependências do JSF e do PrimeFaces. Para tal, adicione o conteúdo presente na **Listagem 1** no seu *pom.xml*. As dependências do JSF são a *jsf-api*, que contém a especificação do framework e a *jsf-impl*, que contém a implementação padrão fornecida pela Oracle. Já as dependências do PrimeFaces são denominadas *primefaces*, *grgen* e *barcode4j-light*. A primeira contém a implementação do framework e as outras duas são utilizadas no componente *BarCode*, apresentado nas próximas páginas. É importante ressaltar que caso esse componente não seja utilizado, tais dependências não são necessárias, pois não fazem parte do core do PrimeFaces.

Desenvolvimento com o PrimeFaces

Após a definição das dependências do projeto, podemos iniciar o desenvolvimento da aplicação. Para isso, o primeiro passo é

configurar o *web.xml*, conforme ilustrado na **Listagem 2**. Como podemos verificar, configuramos neste arquivo o **FacesServlet**, recurso responsável pelo controle e gerenciamento das requisições que chegam ao servidor.

Listagem 1. Arquivo pom.xml para um projeto PrimeFaces com o componente BarCode.

```
<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</groupId>
  <artifactId>prime</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>net.glxn</groupId>
      <artifactId>qrgen</artifactId>
      <version>1.4</version>
    </dependency>
    <dependency>
      <groupId>net.sfbarcode4j</groupId>
      <artifactId>barcode4j-light</artifactId>
      <version>2.0</version>
    </dependency>
  </dependencies>
  <repositories>
    <repository>
      <id>prime-repo</id>
      <name>PrimeFaces Maven Repository</name>
      <url>http://repository.primefaces.org</url>
      <layout>default</layout>
    </repository>
  </repositories>
</project>
```

Analisando mais detalhadamente o *web.xml*, constatamos as seguintes definições: a tag **<servlet>** especifica qual será o servlet responsável pelo controle da aplicação – no caso, o **FacesServlet**. A tag **<servlet-name>** define o nome desse servlet, a fim de facilitar sua identificação e referência no código. Qualquer nome pode ser utilizado, mas o padrão é adotar o “**FacesServlet**”. A tag **<servlet-class>** indica a classe que implementa o servlet do JSF (no caso, **javax.faces.webapp.FacesServlet**), a tag **<load-on-startup>** informa que o servlet deve ser instanciado assim que o Tomcat for iniciado e a tag **<servlet-mapping>** determina qual o padrão de endereço para que a requisição seja executada pelo JSF – em nossa configuração, todos os endereços que começem com **/faces/** ou que

terminem com ***.faces**, ***.jsf** ou ***.xhtml**. Por fim, a tag **<welcome-file-list>** especifica a página inicial da aplicação.

Para ilustrar o uso dos novos componentes do PrimeFaces, vamos desenvolver ao longo do artigo uma aplicação que consiste em um cadastro de usuários. Para isso, primeiramente vamos criar uma classe que represente os usuários que serão cadastrados. Assim, foi implementada a classe **Usuario**, exibida na **Listagem 3**. Os atributos criados para ela foram: **id**, **nomeUsuario**, **senha**, **dataNascimento**, **endereco**, **salario** e **email**, que serão utilizados também em outras partes da aplicação, como no managed bean e nas telas.

Listagem 2. Conteúdo do arquivo web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <welcome-file-list>
    <welcome-file>faces/home.xhtml</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>FacesServlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>FacesServlets</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>FacesServlets</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>FacesServlets</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>FacesServlets</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
  </servlet-mapping>
</web-app>
```

Listagem 3. Código da classe Usuario.

```
package com.javamagazine.model;

import java.util.Date;

public class Usuario {

    private int id;
    private String nomeUsuario;
    private String senha;
    private Date dataNascimento;
    private String endereco;
    private float salario;
    private String email;

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

Depois de definir a classe **Usuario**, será implementado o managed bean a ser utilizado pelas telas da aplicação, criadas mais à frente. Exposto na **Listagem 4**, um managed bean é uma classe cujo ciclo de vida é orquestrado pelo JSF e que pode possuir uma série de anotações para especificar como suas propriedades e métodos devem ser acessados pelas telas da aplicação. Neste exemplo, atribuímos o nome **UsuarioMB** ao nosso managed bean, o que é feito através da anotação `@ManagedBean`. É por meio deste nome que teremos acesso aos métodos desta classe nas páginas da aplicação. Ademais, com a anotação `@SessionScoped` determinamos que seu escopo será limitado pela seção do usuário.

Além de métodos, um managed bean também possui atributos. Estes facilitam a troca de informações entre a camada de visão e a camada de controle da aplicação. Neste exemplo foram definidos os atributos **usuario** e **usuarios**. O primeiro irá representar um objeto criado na interface PrimeFaces para passar os dados preenchidos na tela de cadastro para o *managed bean*. O segundo armazena os usuários já criados e a serem exibidos em um **DataTable**, também na interface PrimeFaces. Neste momento vale ressaltar que a aplicação exemplo não faz uso de um banco de dados com o intuito de manter o foco no PrimeFaces. Por isso os dados são armazenados apenas em uma lista.

Concluída essa etapa, nosso próximo passo é criar as páginas JSF com os componentes do PrimeFaces. O código de uma destas páginas é apresentado na **Listagem 5**. Como podemos verificar, são utilizados diversos componentes do PrimeFaces, e dentre eles, destacamos: `<p:panelGrid>`, que renderiza uma tabela com um número de colunas determinado pelo atributo **columns**; `<p:inputText>`, que exibe um campo de entrada de dados; `<p:spinner>`, que representa um campo de entrada de dados específico para números inteiros; `<p:calendar>`, que representa um campo de entrada específico para datas; e `<p:dataTable>`, que exibe uma listagem com todos os usuários já cadastrados na aplicação.

Listagem 4. Código do managed bean UsuarioMB.

```
package com.javamagazine.bean;

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

import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;

import com.javamagazine.model.Usuario;

@ManagedBean(name="UsuarioMB")
@SessionScoped
public class UsuarioMB {

    private List<Usuario> usuarios = new ArrayList<Usuario>();
    private Usuario usuario = new Usuario();

    public void cadastrarUsuario() {
        usuarios.add(usuario);
        usuario = new Usuario();

        FacesContext.getCurrentInstance().addMessage(
            null, new FacesMessage(FacesMessage.SEVERITY_INFO,"Sucesso!",
            "Usuário cadastrado com sucesso!"));
    }

    public List<Usuario> getUsuarios() {
        return usuarios;
    }

    public Usuario getUsuario() {
        return usuario;
    }

    public void setUsuario(Usuario usuario) {
        this.usuario = usuario;
    }
}
```

Na **Figura 1** é exibida a tela correspondente ao código da **Listagem 5**. Nela é possível observar que os itens do formulário estão alinhados em colunas. Esse comportamento é observado graças ao componente `<p:panelGrid>`, cujo código define sua apresentação em duas colunas (`columns="2"`). Além disso, seus elementos internos também estão dispostos em duas colunas distintas.

The screenshot shows a web page with a user registration form at the top and a user list table below it. The registration form has fields for ID (3), Name (Carlos), Password (*****), Address (Rua HZS), Salary (1010), Email (b@b.com), and Birthdate (17/06/15). A blue button labeled 'Cadastrar' is visible. Below the form is a table with columns: Nome, Senha, Endereço, Salário, E-mail, and Data Nascimento. It contains three rows of data corresponding to the users listed in the registration form.

ID:	3				
Nome:	Carlos				
Senha:	*****				
Endereço	Rua HZS				
Salário	1010				
E-mail	b@b.com				
Nascimento	17/06/15				
Cadastrar					
Nome	Senha	Endereço	Salário	E-mail	Data Nascimento
Eduardo	1234	Rua ABC	1000.0	a@a.com	17/06/2015
Luiz	123423	Rua XYZ	1010.0	b@b.com	17/06/2015
Carlos	123423	Rua HZS	1010.0	b@b.com	17/06/2015

Figura 1. Tela de cadastro e listagem da aplicação exemplo

No formulário, como diferencial, temos os seguintes campos: **Salário**, que possui as setas para cima e para baixo por corresponder ao componente `<p:spinner>`, geralmente utilizado quando o campo de entrada receberá valores numéricos; **Nascimento**, onde o usuário pode escolher uma data em um calendário, graças ao componente `<p:calendar>`; e **Senha**, que é determinado pelo componente `<p:password>`, exibe o caractere ******* em vez do valor realmente informado pelo usuário.

Listagem 5. Exemplo de tela de cadastro e listagem com componentes do PrimeFaces.

```
<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>
<h:form>
    <p:messages id="messages" />
    <p:panelGrid columns="2">
        <p:outputLabel for="id" value="ID:" />
        <p:inputText id="id" value="#{UsuarioMB.usuario.id}" />

        <p:outputLabel for="nome" value="Nome:" />
        <p:inputText id="nome" value="#{UsuarioMB.usuario.nomeUsuario}" />

        <p:outputLabel for="senha" value="Senha:" />
        <p:password id="senha" value="#{UsuarioMB.usuario.senha}" feedback="true" />

        <p:outputLabel for="endereco" value="Endereço" />
        <p:inputTextarea id="endereco"
                         value="#{UsuarioMB.usuario.endereco}" />

        <p:outputLabel for="salario" value="Salário" />
        <p:spinner id="salario"
                   value="#{UsuarioMB.usuario.salario}" />

        <p:outputLabel for="email" value="E-mail" />
        <p:inputText id="email"
                     value="#{UsuarioMB.usuario.email}" />

        <p:outputLabel for="dataAniversario" value="Nascimento" />
        <p:calendar id="dataAniversario" value="#{UsuarioMB.usuario.dataNascimento}" />

    <p:commandButton value="Cadastrar" />
    <action="#{UsuarioMB.cadastrarUsuario}" update="messages, tabelaUsuarios">
    </p:commandButton>
</p:panelGrid>
<p: dataTable style="width:600px;" id="tabelaUsuarios" var="user"
               value="#{UsuarioMB.usuarios}">
    <p:column headerText="Nome">
        <h:outputText value="#{user.nomeUsuario}" />
    </p:column>

    <p:column headerText="Senha">
        <h:outputText value="#{user.senha}" />
    </p:column>

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

    <p:column headerText="Salário">
        <h:outputText value="#{user.salario}" />
    </p:column>

    <p:column headerText="E-mail">
        <h:outputText value="#{user.email}" />
    </p:column>

    <p:column headerText="Data Nascimento">
        <h:outputText value="#{user.dataNascimento}">
            <f:convertDateTime pattern="dd/MM/yyyy" />
        </h:outputText>
    </p:column>
</p: dataTable>
</h:form>
</h:body>
</html>
```

Além disso, com o atributo **feedback** com o valor **true**, esse campo possui uma validação para indicar se a senha é forte ou fraca.

Por fim, para mostrar os dados cadastrados, foi inserido no formulário o componente **DataTable**, que na **Figura 1** já apresenta três usuários.

Novos componentes do PrimeFaces 5.1

Na versão 5.1 do PrimeFaces foram adicionados três novos componentes: o **Ribbon**, cuja função é facilitar a criação de barras de ferramentas semelhantes às que encontramos em soluções desktop, como Word e Excel; o **InputSwitch**, usado como campo de entrada com as opções de ligado e desligado (*On* e *Off*), representando, por exemplo, um valor booleano no managed bean; e o **Barcode**, que gera códigos de barras em diferentes formatos, entre eles o QR code (veja o **BOX 1**).

Além disso, analisaremos o Grid CSS, que foi adicionado visando possibilitar a adaptação automática das aplicações de acordo com o dispositivo de acesso. Esse componente é utilizado internamente pelo PrimeFaces para definir o layout de diversos componentes – tanto para os componentes antigos (**panelGrid**, **spinner**, **calendar**, etc.) quanto para os novos (**ribbon** e **barcode**).

Box 1. QR Code

O QR Code (Quick Response Code) é um código de barras bidimensional bastante utilizado em aplicações para celulares. Nestas aplicações, o usuário tira uma foto do código de barras e estas retornam alguma informação como um e-mail, uma URL ou uma posição geográfica. A grande vantagem do QR Code, se comparado ao código de barras tradicional, é que ele pode armazenar uma quantidade muito maior de informação

O componente Ribbon

Para melhor organizar os seus itens, as barras de ferramentas podem ser organizadas em abas, conforme demonstra o código da **Listagem 6**.

Nesse exemplo, as abas *Arquivo*, *Editiar* e *Opções* possuem internamente um **RibbonGroup**, que nada mais é que um agrupamento de componentes, e dentro de cada grupo podem ser incluídos quaisquer componentes do PrimeFaces, como botões e campos de formulário.

A **Figura 2** apresenta o trecho da tela que mostra a barra que acabamos de implementar. Nela é possível observar que existem três abas (*Arquivo*, *Editiar* e *Opções*). Em *Arquivo*, temos os grupos *Options*, *Clipboard* e *Fonte* e dentro de cada grupo existem ainda outros componentes, como comboboxes e alguns botões.

Listagem 6. Código com o componente Ribbon do PrimeFaces.

```
<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>
<h:form>
<p:ribbon>
<p:tab title="Arquivo">
<p:ribbonGroup label="Options">
<p:commandButton value="Novo" icon="ui-ribbonicon-new"
    styleClass="ui-ribbon-bigbutton" type="button"/>
<p:commandButton value="Salvar" icon="ui-ribbonicon-save"
    styleClass="ui-ribbon-bigbutton" type="button"/>
</p:ribbonGroup>

<p:ribbonGroup label="Clipboard" style="width:120px">
<p:selectManyButton>
<p:commandButton value="Copiar" icon="ui-ribbonicon-paste"
    styleClass="ui-ribbon-bigbutton" type="button"/>
<p:commandButton value="Cortar" icon="ui-ribbonicon-cut"
    style="width:64px" type="button"/>
<p:commandButton value="Imprimir" icon="ui-ribbonicon-print"
    style="width:64px" type="button"/>
</p:selectManyButton>
</p:ribbonGroup>

<p:ribbonGroup label="Fonte" style="width:220px">
<p:selectOneMenu appendTo="@this">
<f:selectItem itemLabel="Arial" itemValue="0"/>
<f:selectItem itemLabel="Comis Sans" itemValue="1"/>
<f:selectItem itemLabel="Helvetica" itemValue="2"/>
<f:selectItem itemLabel="Times New Roman" itemValue="3"/>
<f:selectItem itemLabel="Verdana" itemValue="4"/>
</p:selectOneMenu>

<p:selectOneMenu appendTo="@this">
<f:selectItem itemLabel="10" itemValue="10"/>
</p:selectOneMenu>
</p:ribbonGroup>
</p:tab>

<p:tab title="Editar">
<p:ribbonGroup label="Zoom">
<p:commandButton value="In" icon="ui-ribbonicon-zoomin"
    styleClass="ui-ribbon-bigbutton" type="button"/>
<p:commandButton value="Out" icon="ui-ribbonicon-zoomout"
    styleClass="ui-ribbon-bigbutton" type="button"/>
</p:ribbonGroup>
</p:tab>

<p:tab title="Opções">
<p:ribbonGroup label="Zoom">
<p:commandButton value="In" icon="ui-ribbonicon-zoomin"
    styleClass="ui-ribbon-bigbutton" type="button"/>
<p:commandButton value="Out" icon="ui-ribbonicon-zoomout"
    styleClass="ui-ribbon-bigbutton" type="button"/>
</p:ribbonGroup>
</p:tab>
</p:ribbon>
</h:form>
</h:body>
</html>
```



Figura 2. Utilização do componente Ribbon

O componente InputSwitch

Outro componente adicionado à nova versão do PrimeFaces é o InputSwitch, utilizado na **Listagem 7**. Esse componente é bastante simples e funciona como um campo de entrada no qual os valores podem ser **true** e **false**, ou respectivamente On e Off. Na listagem é possível observar os atributos **onLabel** e **offLabel**, que definem os textos que serão exibidos para cada opção, e o **value**, que define qual é o atributo correspondente a esse elemento no *managed bean*. Deste modo, para utilizar o InputSwitch também é necessário implementar um *managed bean*, que receberá a opção selecionada pelo usuário.

Na **Listagem 8** mostramos o código deste *managed bean*. Neste exemplo, o valor do campo **inputswitch** é submetido para o método **envia()**, que simplesmente escreve no console do servidor qual foi o valor escolhido.

A **Figura 3** apresenta o resultado dos códigos das **Listagens 7 e 8**, uma página que renderiza o componente Ribbon com os labels **Ligado** e **Desligado**.



Figura 3. Exemplo com o componente InputSwitch

O componente Barcode

O último componente que iremos analisar é o Barcode, responsável por gerar códigos de barras. A **Listagem 9** mostra o código utilizado para isso, demonstrando como gerar as diferentes opções disponíveis. Conforme ilustrado, esse componente tem

dois atributos: *value*, que define o valor com o qual o código de barras será impresso; e *type*, que indica o padrão do código de barras adotado.

Listagem 7. Código com o componente InputSwitch do PrimeFaces.

```
<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>
<h:form>
<p:inputSwitch id="switch" onLabel="Ligado"
    offLabel="Desligado" value="#{SwitchMB.onOff}"></p:inputSwitch>
<br><br>
<p:commandButton value="Envia" action="#{SwitchMB.envia}">
</p:commandButton>
</h:form>
</h:body>
</html>
```

Listagem 8. Managed bean que recebe o valor definido no componente InputSwitch.

```
package com.devmedia.managedbeans;

import java.io.FileOutputStream;
import java.io.IOException;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;
import javax.faces.context.FacesContext;
import org.primefaces.event.FileUploadEvent;
import com.devmedia.model.Connect;
import com.devmedia.model.Local;

@ManagedBean(name = "SwitchMB")
@ViewScoped
public class SwitchManagedBean {

    private boolean onOff;

    public void envia() {
        System.out.println(onOff);
    }

    public boolean isOnOff() {
        return onOff;
    }

    public void setOnOff(boolean onOff) {
        this.onOff = onOff;
    }
}
```

Listagem 9. Código para criação de códigos de barras em diferentes padrões.

```
<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>
<h:form>
<p:panelGrid columns="2">
<h:outputText value="Interleaved 2 of 5" />
<p:barcode value="0123456789" type="int2of5" />

<h:outputText value="Codabar" />
<p:barcode value="0123456789" type="codabar" />

<h:outputText value="Code39" />
<p:barcode value="0123456789" type="code39" />

<h:outputText value="Code128" />
<p:barcode value="0123456789" type="code128" />

<h:outputText value="EAN-8" />
<p:barcode value="20123451" type="ean8" />

<h:outputText value="EAN-13" />
<p:barcode value="0123456789012" type="ean13" />

<h:outputText value="UPC-A (PNG)" />
<p:barcode value="01234567895" type="upca" format="png" />

<h:outputText value="UPC-E (Vertical)" />
<p:barcode value="01234133" type="upce" orientation="90" />

<h:outputText value="PDF417" />
<p:barcode value="0123456789" type="pdf417" />

<h:outputText value="DataMatrix" />
<p:barcode value="0123456789" type="datamatrix" />

<h:outputText value="Postnet" />
<p:barcode value="0123456789" type="postnet" />

<h:outputText value="QR" />
<p:barcode value="0123456789" type="qr" />
</p:panelGrid>
</h:form>
</h:body>
</html>
```

A **Figura 4** exibe o resultado da execução do nosso código. Nela podemos observar os diferentes padrões de códigos de barras gerados com o mesmo valor. Apenas por questões de espaço, nesta imagem a tabela com os códigos de barras foi dividida em duas.

Grid CSS

Além desses componentes, outra importante novidade é o Grid CSS, um recurso que implementa o conceito de layout responsivo, permitindo assim criar páginas com um design que se adequa para ser visualizado em diferentes dispositivos. Esse layout divide a tela em até 12 colunas e se reorganiza automaticamente dependendo do tamanho da tela do dispositivo de acesso.



Figura 4. Diferentes padrões de códigos de barras

Uma das vantagens do Grid CSS é que ele é bastante leve. De acordo com os desenvolvedores do PrimeFaces, ele tem apenas 1.4KB, o que é uma importante característica para redes não tão rápidas, como as redes móveis.

Para exemplificar o uso deste recurso, a **Listagem 10** apresenta um código com diversas divs que foram definidas com as classes CSS **ui-grid**, **ui-grid-responsive**, **ui-grid-row** e **ui-grid-col-3**. As classes **ui-grid** e **ui-grid-responsive** especificam a div principal da tela. Por sua vez, **ui-grid-row** define uma linha que pode ter outras divs dentro dela e **ui-grid-col-3** configura a div em que os componentes de interface serão inseridos. A partir disso o Grid CSS constrói a tela como uma tabela, organizando e ajustando o tamanho das divs de acordo com o dispositivo que está acessando a página.

As classes do tipo **ui-grid-col-*** definem o número de divs que pode existir dentro de uma linha, sempre em múltiplos de 12. Por exemplo, pode existir uma linha com 12 **ui-grid-col-1**, ou uma linha com quatro **ui-grid-col-3** ou uma linha com um **ui-grid-col-4** e um **ui-grid-col-8**, etc.

A **Figura 5** mostra a tela correspondente sendo acessada de um computador. Nela podemos verificar quatro elementos alinhados, sendo o primeiro um campo de entrada de texto, o segundo uma imagem, o terceiro um QR Code e o quarto um menu.

A **Figura 6** mostra a mesma página, mas sendo acessada de um smartphone. Nela é possível notar que o tamanho dos componentes e a organização da tela foram adaptados para uma tela menor automaticamente, facilitando o trabalho do programador, que não precisa programar uma tela que tenha diferentes definições para diferentes dispositivos.

Melhorias em componentes antigos

Além dos novos componentes, na versão 5.1 do PrimeFaces também foram feitas algumas mudanças em componentes antigos. Boa parte destas alterações foram realizadas visando uma melhoria de desempenho ou outros fatores não visíveis ao programador. Por outro lado, alguns componentes também ganharam

novas funcionalidades. Dentre eles, destacamos o agrupamento de itens no componente **autocomplete** e a opção **toggleable** do componente **menu**.

Listagem 10. Código para criação de uma tela com Grid CSS.

```
<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>
<h:form>
<p:outputPanel>
<div class="ui-grid ui-grid-responsive">
<div class="ui-grid-row">
<div class="ui-grid-col-3">
<p:inputText></p:inputText>
</div>
<div class="ui-grid-col-3">
<h:graphicImage value="/images/java.png"/></h:graphicImage>
</div>
<div class="ui-grid-col-3">
<p:barcode value="0123456789" type="qr"/>
</div>
<div class="ui-grid-col-3">
<p:menu>
<p:submenu label="Cadastro">
<p:menuitem value="Pessoa"/>
<p:menuitem value="Carro"/>
</p:submenu>
</p:menu>
</div>
</div>
</div>
</p:outputPanel>
</h:form>
</h:body>
</html>
```



Figura 5. Página com Grid CSS acessada de um computador



Figura 6. Página com Grid CSS acessada de um smartphone

AutoComplete

O componente autocomplete é uma variação do campo de entrada de texto que busca facilitar a vida do usuário ao sugerir termos relacionados ao que o usuário está digitando. Disponibilizado pelo PrimeFaces desde as primeiras versões, agora foi adicionada a opção de agrupamento dos dados que são exibidos. Para demonstrar esse recurso, vamos implementar primeiramente um managed bean na **Listagem 11**.

Nesta classe, o atributo **loc** receberá o valor digitado pelo usuário no campo e a lista **results** tem os itens que poderão ser sugeridos na tela. No construtor foram adicionados alguns valores que servirão para o exemplo na lista. Além disso, a classe tem dois métodos importantes: **completText()**, que é responsável pelo filtro na lista, retornando apenas os resultados desejados para o autocomplete; e **getGroup()**, que monta os grupos por algum parâmetro – neste caso os grupos são formados pela primeira letra das opções que serão exibidas para o usuário.

Listagem 11. Managed bean para utilização de um AutoComplete com agrupamento dos resultados.

```
package com.devmedia.managedbeans;

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

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

@ManagedBean(name = "LocalMB")
@ViewScoped
public class LocalManagedBean {

    private String local;
    List<String> results = new ArrayList<String>();

    public LocalManagedBean() {
        results.add("São Paulo");
        results.add("Lins");
        results.add("São Carlos");
        results.add("Limeira");
        results.add("Campinas");
    }

    public String getLocal() {
        return local;
    }

    public List<String> completeText(String query) {
        ArrayList<String> locais = new ArrayList<String>();
        for(String r : results) {
            if (r.contains(query)) {
                locais.add(r);
            }
        }
        return locais;
    }

    public void setLocal(String local) {
        this.local = local;
    }

    public char getGroup(String local) {
        return local.charAt(0);
    }
}
```

A **Listagem 12** mostra o código do XHTML com o componente **<p:autoComplete>**. Dentro desse componente, o atributo **value** recebe o nome do atributo do managed bean ao qual estará relacionado, **completeMethod** indica o método do managed bean que fará a busca para mostrar as opções ao usuário e o atributo **itemLabel** define os valores que serão mostrados para o usuário. Já **itemValue** determina o valor que será passado para o managed bean e **var** explica o nome do elemento a ser passado para o método que é definido no atributo **groupBy**, onde é informado o método do managed bean que definirá como será feito o agrupamento dos valores do autocomplete.

Listagem 12. Código XHTML com AutoComplete e agrupamento dos resultados.

```
<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>
        <h:form>
            <p:outputLabel value="Local:" for="local"/>
            <p:autoComplete id="local" value="#{LocalMB.local}"
                           completeMethod="#{LocalMB.completeText}"
                           itemLabel="#{local}" itemValue="#{local}"
                           var="local" groupBy="#{LocalMB.getGroup(local)}" scrollHeight="250"/>
        </h:form>
    </h:body>
</html>
```

A **Figura 7** apresenta a execução desse código. Nela é possível observar o campo *Local* com a letra “a” e o autocomplete exibindo todas as opções que tenham essa letra. Observe ainda que os nomes foram agrupados pelo primeiro caractere, mas poderia ser utilizado qualquer outro critério.

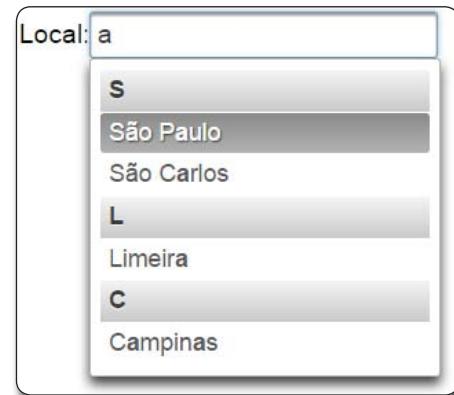


Figura 7. Componente AutoComplete com agrupamento dos resultados

Toggable-Menu

Dentre os diversos componentes disponibilizados pelo PrimeFaces para criar menus, um deles é o **<p:menu>**. Este cria um menu na horizontal onde é possível adicionar itens e sub menus para agrupar novos itens de forma organizada. Visando aprimorar esse recurso, no PrimeFaces 5.1 foi adicionada a melhoria que permite aos submenus serem exibidos com a opção **toggable**.

Para demonstrar essa funcionalidade, a **Listagem 13** mostra um exemplo de XHTML.

Nesse código é possível observar o componente `<p:menu>` com o atributo `toggleable` definido como valor `true`. Em seguida, são adicionados dois `<p:submenu>`, *Cadastro* e *Lista*, que contêm, respectivamente, dois e um componentes do tipo `<p:menuitem>`. Note que o atributo `toggleable`, apesar de ser colocado na tag `<p:menu>`, age sobre os sub menus, pois são eles que podem ser escondidos.

Para simplificar a explicação desse recurso, a **Figura 8** mostra duas imagens do menu criado: uma com todos os itens sendo exibidos e outra com os itens do sub menu *Cadastro* omitidos e o item do sub menu *Lista* em exposição.

Listagem 13. Código para criação de um Toggleable Menu.

```
<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>
<h:form>
    <p:menu toggleable="true">
        <p:submenu label="Cadastro">
            <p:menuitem value="Pessoa" actionListener="#{menuView.cadastraPessoa}"
                        icon="ui-icon-disk"/>
            <p:menuitem value="Carro" actionListener="#{menuView.cadastraCarro}"
                        icon="ui-icon-arrowrefresh-1-w"/>
        </p:submenu>
        <p:submenu label="Lista">
            <p:menuitem value="Pessoa" actionListener="#{menuView.listaPessoa}"
                        ajax="false" icon="ui-icon-close"/>
        </p:submenu>
    </p:menu>
</h:form>
</h:body>
</html>
```



Figura 8. Componente menu com a opção toggleable

Além dos novos componentes apresentados e das alterações nos componentes antigos, o PrimeFaces anunciou que a versão 5.1 trouxe 240 alterações no framework. Essas alterações, em sua grande maioria, são imperceptíveis do ponto de vista do programador, mas representam importantes aprimoramentos em componentes antigos e a correção de mais de 140 bugs.

Outro diferencial do PrimeFaces é que sempre que uma nova versão é lançada a equipe de desenvolvimento disponibiliza um documento mostrando como fazer a migração. Sendo assim, experimente migrar a versão adotada em seu projeto para a mais recente e explore os novos recursos. Essa atitude certamente garantirá mais qualidade e novos recursos para o seu projeto.

Autor



Eduardo Felipe Zambom Santana

ezambomsantana@gmail.com

É bacharel e mestre em Ciéncia da Computação pela UFSCar.

Possui mais de 10 anos de experiência em programação.

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



Autor



Luiz Henrique Zambom Santana

lhzsantana@gmail.com

É bacharel e mestre em Ciéncia da Computação. Possui mais de 10 anos de experiência em programação Java e há dois anos atua com tecnologias de Big Data. Trabalhou em projetos para grandes empresas no Brasil, Argentina e Alemanha. Atualmente é consultor, startupper na 33! e doutorando na UFSC.



Links:

Implementação de referência do JSF.

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

Site oficial do PrimeFaces.

<http://primefaces.org/>

Blog oficial do PrimeFaces.

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

Show case com exemplos de todos os componentes do PrimeFaces.

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

Guia do Usuário do PrimeFaces

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

Guia para migração para a versão 5.1 do PrimeFaces.

<https://code.google.com/p/primefaces/wiki/MigrationGuide>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



FÓRUM DEVMEDIA

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



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

ACESSE AGORA
www.devmedia.com.br/forum

Como definir a tecnologia para o desenvolvimento da interface web

Aprenda a escolher as opções mais adequadas para construir a interface com o usuário de suas aplicações Java EE

O desenvolvimento da interface gráfica de uma aplicação, devido à sua importância, é um assunto estudado em várias áreas da Ciência da Computação. A Interação Humano-Computador (IHC) ou Interface Homem-Máquina (IHM) é uma matéria obrigatória em alguns cursos de graduação ou especialização e trata exatamente deste assunto. O conceito de IHC nos mostra que são vários os requisitos necessários para a criação, bem definida, da interface entre a aplicação e o usuário como, por exemplo: usabilidade, simplicidade, ergonomia e acessibilidade. Além disso, percebemos que conforme a tecnologia avança, esses requisitos podem aumentar ou se modificar.

Nos primórdios da computação, a interface com o usuário era determinada pela própria arquitetura da máquina (ex: cartões perfurados e fitas magnéticas) e pouco depois pelo sistema operacional (ex: linha de comando e cartuchos). Quem teve o "privilegio" de ver funcionando um computador cuja interface com o usuário era uma leitora de cartão perfurado, poderá lembrar de algo nada intuitivo ou até mesmo comprehensivo pela maioria das pessoas. Neste início da era computacional, o modo e a forma da interface da máquina com o homem era com o intuito de atender às exigências e às necessidades do computador, sendo que o ser humano deveria se adaptar a ela. Com a evolução da informática, este conceito se inverteu e foram os sistemas informatizados que passaram a tentar atender às necessidades dos usuários. Assim, os próprios

Fique por dentro

Neste artigo serão abordados os conceitos e recursos oferecidos pelas tecnologias disponíveis para o desenvolvimento de interfaces web para aplicações na plataforma Java EE. Mostraremos como identificar os requisitos essenciais que definem os tipos de interfaces com o usuário e como escolher a melhor opção de mercado para cada um. O objetivo não é demonstrar ou comparar as alternativas oferecidas, mas expor como definir planos estratégicos que podem ser adotados para fazer, ou indicar, uma boa escolha. Com base nisso o leitor terá condições de avaliar o seu projeto e tomar a melhor decisão sobre a tecnologia ou conjunto de tecnologias que serão utilizadas para construção da interface web.

sistemas operacionais mudaram suas interfaces para os modos visuais, e em vez de linhas de comandos ou menus, passaram a existir ícones e janelas, que são interfaces muito mais intuitivas para maioria dos usuários.

Nota

O conceito de computação ubíqua diz que uma aplicação pode ser executada em qualquer tipo de equipamento informatizado e, portanto, não é mais possível prever com exatidão que tipo de dispositivo acessará nossos sistemas e, em muitos casos, muito menos o perfil do usuário.

Quando tratamos do desenvolvimento de aplicações Java para o ambiente web, começamos com os famosos Applets, que são praticamente uma versão do modelo Desktop (Swing) executando através do navegador. Estes surgiram como a primeira

alternativa para este tipo de desenvolvimento e estão por aí até hoje. O uso principal dos Applets, atualmente, é para possibilitar o acesso a dispositivos conectados no equipamento do usuário, como leitoras de cartões criptográficos, algo que não é possível fazendo uso apenas de recursos oferecidos pelo navegador por questões de segurança. No atual estado da tecnologia Java, chegamos a uma integração quase simbólica com os navegadores, o que é possibilitado através de componentes e frameworks que implementam as especificações da plataforma Java Enterprise Edition. Para atender essa relação, entre a tecnologia Java e os navegadores, existem no mercado diversas implementações de bibliotecas e componentes para a camada de interface de aplicações Java EE e que abordaremos neste artigo.

Quando seguimos o conceito de arquitetura dividida em camadas, mais especificamente quando optamos pelo padrão MVC (Modelo-Visão-Controle), em uma aplicação Java EE, na camada de visão é onde encontraremos a maior variedade de opções de tecnologias que podem ser utilizadas. Inclusive, algumas disponíveis no mercado não são exclusivas para a plataforma Java, como é o caso das bibliotecas JavaScript. Com base nisso, o principal objetivo deste artigo será analisar os principais requisitos de um projeto para se definir uma estratégia de escolha de tecnologia que visa minimizar o impacto no esforço do seu desenvolvimento.

O perfil do usuário

Quando compararmos a forma como era feita a interação Humano-Computador nos primórdios da computação com a que é feita hoje, podemos perceber uma grande evolução nas características destas interfaces. No princípio, o foco da relação (homem-máquina) era nas características do equipamento e atualmente o recomendado é focar no perfil do usuário. No início da era computacional, como já mencionado, o homem era obrigado a se adaptar à máquina, e com isso, para utilizar os computadores, os usuários precisavam ter conhecimentos técnicos ou passar por treinamentos para entender e saber usar esse tipo de interface.

Nos últimos anos, é perceptível um grande aumento do número de usuários de sistemas de informática, principalmente através do ambiente disponível na internet, que provê uma série de facilidades de acesso que antes não eram possíveis no modelo cliente-servidor ou puramente desktop. Há também uma mudança no perfil daqueles que já eram usuários de sistemas mesmo antes do advento da internet, e ao mesmo tempo uma maior diversificação no perfil dos novos usuários. Apesar dessa mudança clara no perfil dos usuários, é perceptível que as tecnologias e a forma de desenvolvimento de interface dos sistemas parecem ainda estar voltadas aos usuários especializados, perfil de usuário predominante na era pré-internet. Isso evidencia, em muitos casos, a pouca preocupação dos desenvolvedores em identificar qual é o seu usuário.

Parece ser bastante comum que quem decide a forma e a apresentação da interface dos sistemas sejam somente os analistas

de sistemas e/ou designers, em vez do próprio usuário, ou pelo menos com a participação deste. Claro que não devemos deixar a cargo do usuário decidir tudo, mas também é preciso considerar os casos onde deve haver maior participação. Algumas técnicas ágeis de desenvolvimento de sistemas até já preconizam uma maior participação do cliente em todas as etapas da construção de um sistema informatizado e, portanto, na definição da interface. Há, também, os casos em que o cliente não é explicitamente conhecido, como o caso de um serviço ou software que gostaríamos de oferecer no mercado. As aplicações para dispositivos móveis são os exemplos mais recentes disso.

É um grande fator de risco iniciar o desenvolvimento de qualquer interface, seguindo os conceitos de IHC, sem ao menos considerar que tipo de usuário a aplicação terá, e isso vale independentemente do dispositivo que será usado, assim como também parece estranho que a opinião ou gosto do usuário não seja considerada. É necessário entender que cada característica do usuário influenciará no seu perfil, como idade, profissão, condição física, entre outros, e com esse perfil podemos identificar o tipo de usuário que utilizará a aplicação. Somente assim é possível saber se o usuário é especializado ou não.

Perceba que se você não sabe qual é o perfil do usuário da sua aplicação, então não saberá exatamente para que ela serve. Diante disso, a pior opção é tentar se colocar no lugar do usuário, pois o que parece óbvio para você pode não ser para quem utiliza a aplicação. Por fim, sempre que possível, além de realizar pesquisas com os possíveis usuários, apresente protótipos ou, ao menos, um esboço de algumas alternativas. Lembre-se sempre daquela máxima do mundo das vendas: “O cliente sempre tem razão”, principalmente quando se trata da interface do sistema.

Os tipos de interface

Para a plataforma Java EE, a tecnologia padrão para o desenvolvimento de interfaces em sistemas web é o JavaServer Faces. Um dos diferenciais do JSF, na época do seu lançamento, foi justamente trazer para a interface web comportamentos e aparências das interfaces desktop mais avançadas da época, como o movimento de “arrastar e soltar”. Com isso, as bibliotecas que implementam a especificação JSF diminuíram bastante as diferenças que haviam entre as interfaces de sistemas do tipo desktop para aqueles para web.

Com o advento dos dispositivos móveis, como palmtops e handhelds, inicialmente as interfaces dos sistemas desenvolvidos em Java que atendiam esse tipo de equipamento eram criadas para serem acessadas via navegadores web ou através da criação de aplicativos nativos a serem instalados no dispositivo. No entanto, estes possuíam recursos visuais bastante restritos evidenciando, novamente, um processo cíclico de retrocesso e avanço nos tipos de interface. Mais recentemente, com a chegada dos smartphones e tablets, fortaleceu-se o uso dos aplicativos nativos, que representam praticamente um retorno ao modelo de interface desktop, porém com

comportamento próprio, por conta do “novo” mecanismo de entrada de dados, o touchscreen.

Sabemos que a linguagem Java é a base do kit para o desenvolvimento de aplicativos no sistema operacional mais utilizado em dispositivos móveis, o Android. Portanto, é com pouco esforço que um desenvolvedor que domina esta linguagem consegue criar soluções para este tipo de equipamento. No entanto, muitos casos não serão resolvidos apenas com aplicações nativas, pois podem possuir requisitos complexos (uso de tokens criptográficos, por exemplo) para serem resolvidos apenas com os recursos muitas vezes limitados dos smartphones, o que pode exigir uma abordagem mista. Assim, é bastante provável que muitos sistemas web continuem sendo criados de forma mais tradicional, a partir das tecnologias padrão da especificação Java EE, e acessados das mais diversas plataformas, inclusive móveis.

Para melhorar a visualização e usabilidade de aplicações Java EE por dispositivos móveis, algumas implementações de tecnologias como JSF passaram a oferecer componentes que simulam uma interface semelhante à dos aplicativos nativos. Esse tipo de interface, que faz uso de componentes JSF especializados para apresentação em dispositivos móveis, é conhecida como WEB-Mobile. Neste tipo de interface a aplicação web é executada através do navegador, mas mimetizando a sua aparência para se tornar adequada ao tamanho da tela do dispositivo.

Ultimamente, o que tem sido bastante comentado é o desenvolvimento de interfaces web do tipo responsivas, do inglês *Responsive Web Design* (RWD). Esta opção consiste em uma técnica de utilizar recursos de CSS, HTML e JavaScript para criar páginas web que podem adaptar seus componentes, como links ou menus, ao tamanho da tela em que o navegador é executado. Através desta técnica é possível criar interfaces que podem se adaptar, teoricamente, a qualquer tipo de interface. Explicando de forma simplificada, se a página responde ao tamanho da tela onde roda o navegador, e estiver em um equipamento do tipo desktop, onde o espaço da tela é maior, os componentes se

ajustarão para aparecerem maiores, enquanto que se forem telas de dispositivos móveis, seus tamanhos e formas serão reduzidos. Obviamente, por ser um recurso que objetiva ser comum a vários tamanhos de telas, pode não atender com a mesma precisão de um componente especializado as características desejadas para se alcançar uma boa apresentação, como usabilidade, aparência ou acessibilidade, pois essas características, em geral, são bastante dependentes da plataforma. Por exemplo, um botão com um ícone pode ficar distorcido se a imagem for a mesma para vários tamanhos de tela. Por outro lado, pode ser um caminho para evitar o desenvolvimento repetitivo de uma mesma interface de acordo com o dispositivo que a acessará.

Portanto, conhecer os requisitos para identificar quais são os tipos de interfaces que a aplicação oferecerá, assim como traçar o perfil do usuário, influenciará diretamente na escolha da tecnologia que será adotada para a construção da camada de visão. E dependendo do tipo da interface que for identificada como padrão para a aplicação, algumas vezes bastante simples, pode até não ser necessário adotar uma biblioteca acessória, usando assim apenas os recursos básicos da Java EE ou mesmo HTML puro. Se verificarmos a especificação HTML5, notaremos que ela traz diversos recursos que se integram diretamente ao navegador, o que antes não era possível sem a ajuda de frameworks especializados, e que podem ser o suficiente para desenvolver uma interface que não tenha muitos requisitos de ação do usuário.

As várias alternativas disponíveis no mercado

Atualmente, existem várias implementações que podem ser utilizadas de forma acessória para desenvolver interfaces de aplicações Java EE. A seguir, serão listadas algumas das mais conhecidas e utilizadas no mercado, dentre as quais algumas que também já foram objeto de estudo em outros artigos publicados na revista. Essa listagem será utilizada para estabelecer um comparativo entre essas tecnologias, que por sua vez formará a base para definição de um plano estratégico para adoção da solução ideal para o desenvolvimento da interface.

Citaremos primeiro as bibliotecas que implementam a especificação JSF, por ser o padrão de interface da plataforma Java EE:

- **ICEfaces:** Encontram-se no site oficial deste projeto cerca de 70 componentes básicos (aderentes à especificação), além de um subprojeto chamado ACE Components (*ICEfaces Advanced Components*), que oferece mais alguns conjuntos de componentes agrupados por funcionalidades como, por exemplo, de acessibilidade. Este produto possui dois tipos de versões: uma livre, chamada de ICEfaces, e outra paga, a ICEfaces EE. Como diferencial, a versão ICEfaces EE oferece um pacote de serviços que prevê suporte técnico para o uso dos componentes e também permite ao usuário pedir a customização de algum componente ou mesmo a criação de um novo. A versão livre, apesar de apresentar um bom conjunto de componentes, não possui alguns bastante importantes, como os de designer responsivo;



CURSOS ONLINE



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

CONHEÇA OS CURSOS MAIS RECENTES:

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



Para mais informações :

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

- **MyFaces:** É um projeto do tipo guarda-chuva, sendo mantido pela fundação Apache e dividido em quatro subprojetos: Core, Trinidad, Tobago e Tomahawk. O core contém uma implementação do que é definido na especificação JSF. O Trinidad, além do básico contido no core, possui componentes que contemplam requisitos de acessibilidade. O Tobago, por sua vez, traz componentes baseados na especificação JSF, mas que não utilizam as diretivas padrão de design (tags do tipo <h:>) de uma página web (ou XHTML, como é o caso do JSF), mas as substitui por uma sintaxe própria, definida pelo subprojeto com o objetivo de prover um design responsivo, mas com um modo de desenvolvimento parecido com o do JSF. E o Tomahawk, que representa um conjunto de componentes acessórios e é uma implementação com uma abordagem de acordo com a especificação do JSF, isto é, faz uso das diretivas de design do HTML (tags) junto com aquelas providas pelo componente, criando assim as páginas do tipo XHTML. O MyFaces possui ainda alguns pacotes de extensão, para suporte às integrações que o JSF possui com as especificações Bean Validation e CDI, a saber: Orchestra, Extensions Validator e CDI Extensions. Por ser um projeto da Apache, é totalmente livre;

- **RichFaces:** Na página de showcase do projeto são apresentados 39 componentes básicos. O RichFaces foi criado e é mantido pela JBoss, que em abril de 2006 foi adquirida pela Red Hat. Assim como o projeto da Apache, é de código livre e não possui subscrição. Além da implementação da especificação JSF, possui componentes extras projetados para interfaces de dispositivos móveis, os chamados componentes WEB-Mobile, e também componentes para atender aos requisitos de acessibilidade;

- **PrimeFaces:** Na página de showcase do projeto são apresentados 117 componentes. Por conta de sua maturidade, foi adotado como a implementação padrão da plataforma Java EE 6. O código do PrimeFaces é criado sob licença aberta, mas a empresa que o desenvolve oferece versões comerciais denominadas Elite/Case/PRO e uma versão livre denominada COMMUNITY.

Os usuários da versão livre têm acesso aos *releases* maiores (ex: 4.1, 5.0, 5.1), enquanto os detentores de licenças comerciais têm direito às correções menores (ex: 5.0.1). Na prática, quem usa a licença COMMUNITY precisa aguardar um release maior para obter acesso às correções que ocorreram nas versões menores. Diferentemente do projeto ICEfaces, onde o chamado ICEfaces EE possui um conjunto de componentes maior que a versão livre, no PrimeFaces, apesar da versão COMMUNITY não oferecer, temporariamente, o código mais recente, o que é liberado para a comunidade é uma *release* completa que contém todos os componentes e melhorias da versão maior. A versão paga (Elite, Case ou PRO) dá direito, além do acesso à versão mais recente, a algumas opções de suporte que incluem até o pedido de desenvolvimento de componentes customizados (subscrição PRO).

Como todas estas soluções implementam a especificação JSF, tais bibliotecas atendem a vários requisitos de forma parecida e costumam apresentar uma performance semelhante. O que difere cada uma, em geral, é a quantidade de componentes que está disponível.

Frameworks para JavaScript

Os frameworks para JavaScript estão se tornando uma boa opção para o desenvolvimento da camada de visão de aplicações Java EE, por serem agnósticos em relação às linguagens que podem ser usadas nas outras camadas. Assim como na lista para bibliotecas JSF, vamos listar algumas opções como exemplo para tomar como base para o nosso objetivo: definir um plano estratégico para seleção de uma tecnologia ou conjunto delas. Sendo assim, vejamos algumas das principais opções disponíveis em JavaScript:

- **Bootstrap:** Mantido pelos criadores do Twitter, tem bastante destaque entre os desenvolvedores de aplicações web. Desde a versão 2 tem sido uma referência em design responsivo e por isso possui diversas extensões de terceiros. Já existem, inclusive, extensões nacionais para essa solução como, por exemplo, o Globo-Bootstrap, criado pela equipe do site *globo.com*;

- **AngularJS:** Mantido pelo Google, adota a estratégia de fazer a ligação com a aplicação Java através de serviços REST, sendo considerado atualmente um dos grandes players deste tipo de abordagem. A versão estável mais recente é a 1.3.10 e há previsão de uma nova versão (1.4) ainda para 2015. De acordo com o que tem sido publicado em alguns blogs de especialistas, haverá bastante diferença entre as duas versões. Portanto, se for iniciar um projeto com este framework, pode ser interessante aguardar o lançamento da nova versão, para não correr o risco de criar uma aplicação que logo estará se tornando um legado;

- **Google Web Toolkit (GWT):** Como o próprio nome indica, é um projeto mantido pelo Google e a sua principal característica é permitir que o desenvolvedor crie a interface usando a linguagem Java, posteriormente processando a interface para JavaScript. Já não é tão avançado como o AngularJS, mas possui um bom grau de maturidade;



- **Vaadin:** É uma solução baseada no GWT, porém, enquanto o GWT “puro trabalha apenas no lado do cliente”, o Vaadin atua como uma arquitetura do lado do servidor. O interessante do Vaadin, assim como do GWT, é que não é necessário trabalhar editando páginas HTML ou XML. Toda a codificação é feita utilizando a linguagem Java, de modo bastante parecido com a programação com Swing. Assim, a curva de aprendizado para desenvolvedores que conhecem Java, mas ainda não trabalham com o ambiente web, é bastante reduzida;

- **Crux Framework:** Provavelmente o menos conhecido do leitor. Apesar disso, trata-se de uma solução nacional desenvolvida pela empresa Triggo Labs e que também utiliza o GWT como base. Diferentemente do Vaadin, optou por levar o processamento para o lado do cliente através do uso de HTML5. A comunicação com a camada de controle é feita a partir de serviços REST. Sendo um produto nacional, o contato com os desenvolvedores da solução pode ser mais direto. No entanto, é comum em nossa cultura dar mais crédito a produtos estrangeiros em detrimento à produção nacional, o que por consequência desestimula a criação local. Ainda assim, tem se destacado entre as opções disponíveis e merece ser explorado e o nosso apoio.

Como alguns desses frameworks em JavaScript são tecnologias bastantes recentes, não possuindo, ainda, um padrão formal, como já ocorre com o JSF, pode haver bastante diferença entre as funcionalidades oferecidas. Assim, caso deseje trocar de uma solução para outra durante a construção de um projeto, pode não ser um trabalho trivial ou mais previsível, como ocorre com as soluções JSF. Dito isso, uma vez escolhida a tecnologia, dificilmente haverá volta, pelo menos com o cenário atual.

Além disso, outra opção pode ser adotar o JavaScript sem nenhum framework e fazer tudo “na mão”. No entanto, estes são casos em que o volume de implementação não é grande e o conhecimento do desenvolvedor sobre a tecnologia é muito bom. Assim como tudo na vida, há pontos positivos e negativos. O próprio uso de frameworks Java é recomendado ou rechaçado por desenvolvedores. Portanto, devemos partir do princípio que cada caso tem suas próprias características e que o que é recomendado para um pode não ser uma boa opção para outro. Sendo assim, um estudo para determinar o uso ou não de um framework também faz parte da estratégia de adoção da tecnologia para implementação da interface.

Em geral, tanto as bibliotecas JSF como as JavaScript oferecem bons recursos para desenvolver uma interface rica e elegante. Praticamente todas, nas versões mais recentes, também oferecem recursos para atender aos requisitos de web-mobile e design responsivo. Em muitos casos, há uma mescla das bibliotecas, por exemplo: o PrimeFaces adota o Bootstrap para oferecer design responsivo e jQuery na criação de componentes do tipo web-mobile. No fundo, todos exploram recursos do JavaScript.



Todas as bibliotecas listadas neste artigo, tanto JSF como JavaScript, afirmam em suas documentações que atendem ao ARIA (*Accessible Rich Internet Applications*), um padrão de acessibilidade internacional que define os requisitos mínimos que uma aplicação deve oferecer para ser acessível às pessoas com alguma dificuldade física ou intelectual. No entanto, esta característica ainda é desconsiderada por muitos desenvolvedores, sendo raro encontrar um sistema web (principalmente no comércio eletrônico) que apresente estes recursos.

É preciso considerar também os casos em que a pesquisa para definir o perfil do usuário não é capaz de explicitar as necessidades de acessibilidade, pois apesar de não apresentar características físicas evidentes, é bastante comum alguns usuários terem preferência por interfaces mais acessíveis. A interação com o sistema através do teclado (teclas de atalho) em vez de apenas com o mouse, nos casos dos sistemas desktop, é um exemplo desse tipo de preferência. Por isso é importante verificar se a biblioteca atende aos requisitos de acessibilidade necessários ao sistema a ser desenvolvido, antes mesmo do início da codificação.

Identificar uma tecnologia e uma biblioteca disponível no mercado como candidata a ser adotada em um projeto significa avaliar os recursos nela disponíveis e qual o nível de conhecimento do desenvolvedor (ou equipe de desenvolvimento) em cada uma das possíveis opções.

Uma prática importante para balizar a definição da escolha da tecnologia e da biblioteca é elaborar um protótipo da mesma aplicação em cada uma das bibliotecas elencadas, fazer os testes de performance num ambiente de homologação e, por fim, pedir a avaliação da interface pelo próprio usuário ou um conjunto significativo deles. Esses resultados poderão apontar os principais pontos positivos e negativos de cada solução na visão do usuário, além do resultado da performance de cada uma.

Custo x Benefício

Umas das questões mais importantes em qualquer projeto, seja de software ou não, está na relação Custo x Benefício. O ideal é ter um baixo custo combinado com um alto benefício, mas nem sempre isso é possível. Os requisitos perfil de usuário e tipo de interface devem sinalizar o uso de uma ou algumas alternativas de bibliotecas disponíveis no mercado. O conjunto de funcionalidades que cada alternativa possui, ou uma única alternativa, nos fornecerá uma medida do fator de benefício. Quando a análise destes requisitos indicar que há mais de uma opção de biblioteca, é preciso considerar o custo total da aplicação com relação aos benefícios que cada alternativa trará. Em projetos de software, o tempo de desenvolvimento é impactante no custo total e normalmente é o que define a escolha, mas também devemos considerar o custo de operação e manutenção.

O tempo de desenvolvimento depende do conhecimento e experiência do desenvolvedor com a tecnologia a ser adotada. Quanto menor a bagagem intelectual maior será o tempo de construção. Já o custo de operação está atrelado ao consumo de recursos que a aplicação fará. Memória, espaço em disco, processamento, etc., tudo isso se soma ao valor de operação. E apesar desse custo apresentar uma tendência de redução nos últimos anos, o advento dos serviços de computação em nuvem faz com que o custo de operação tenha um peso importante no custo total da aplicação.

Após o desenvolvimento, vem o custo da manutenção, que pode envolver tanto as alterações no código como no ambiente operacional, pois um requisito novo pode gerar demanda de desenvolvimento e um aumento de consumo pode exigir mudança de recursos operacionais.

Dependendo da tecnologia adotada, o fator de equilíbrio na relação Custo x Benefício pode não atender ao objetivo ideal: que é o de baixo custo e alto benefício. Às vezes, a tecnologia tem custo reduzido no desenvolvimento por já ser utilizada a mais tempo pelos desenvolvedores, mas pode não oferecer alguns benefícios interessantes para o usuário, como uma interface responsiva. Por outro lado, tecnologias mais novas podem tomar mais tempo de desenvolvimento, aumentando o custo, mas muitas vezes oferecem mais benefícios, como o consumo menor de recursos operacionais. Sendo assim, cabe em cada caso identificar o que impactará mais na relação. Se não for possível um equilíbrio, a alternativa deve atender o que for mais adequado ao cenário geral do desenvolvimento da aplicação, que envolve: tempo, recursos financeiros, recursos humanos e o interesse do cliente. Exemplificando: se há escassez de recursos financeiros, opta-se pela alternativa de menor custo. Se existe um usuário definido, ou um cliente que custeará o projeto, caberá a quem paga decidir se investe nos benefícios ou adéqua o projeto a um custo.

Diante disso, saber analisar essa relação (Custo x Benefício) pode ser um argumento importante quando há resistências em relação à adoção de alguma alternativa. Em geral, a resistência à mudança é um comportamento natural do ser humano, e por isso, é importante apresentar os argumentos corretos para promover mudanças com sucesso.

Definindo uma estratégia para adoção de uma tecnologia

Existem alguns requisitos fundamentais para definir uma estratégia para adoção de uma tecnologia para o desenvolvimento da interface de um sistema. Neste artigo, estamos citando os principais deles, a saber: perfil do usuário, tipo de interface, relação custo x benefício e nível de conhecimento do desenvolvedor. Dentre eles, podemos eleger os dois mais importantes, analisados a seguir.

O primeiro é o perfil do usuário, que deve ser devidamente decifrado e que pode indicar indiretamente uma tecnologia. Para isso, é importante criar protótipos da interface em cada tecnologia possível. Em geral, é comum obter praticamente o mesmo resultado visual com qualquer alternativa disponível. Mesmo assim, podem existir algumas diferenças que são resultado do que é fornecido por cada tecnologia. Em alguns casos, notam-se estas diferenças na parte visual, quando lidamos com templates, por exemplo, ou na performance, que dependendo da tecnologia pode onerar a parte servidora (caso do JSF) ou o cliente (caso do JavaScript). Neste ponto é preciso saber com o que o cliente mais se importa: se com o visual da aplicação ou com o seu desempenho? A análise do perfil não deve ser usada apenas para identificar qual tipo de interface é adequado ao usuário, mas também para



que possamos apresentar os benefícios que interessam a ele e os pontos negativos de cada solução. Muitas vezes, uma alternativa pode ser recusada pelo usuário porque os recursos que lhe foram apresentados não eram focados nos pontos que lhe interessavam. Por isso a importância de identificar o perfil e escolher uma tecnologia mais adequada às suas necessidades.

O segundo é o nível de conhecimento sobre a tecnologia que o desenvolvedor, ou equipe, detém. Aventurar-se numa onda tecnológica mais atrativa pode não levar aos melhores resultados. Sendo assim, é importante que se procure desenvolver com tecnologias atuais e inovadoras, mas também é preciso saber reconhecer as limitações do desenvolvedor, ou equipe, para que a própria tecnologia não se torne um fator de risco para o projeto. Se o sistema está sendo desenvolvido utilizando uma metodologia ágil, como Scrum, teremos como uma das principais características deste método uma entrega rápida e constante. Diante disso, se a tecnologia influenciar neste tempo, poderá comprometer o processo. Considere também as fontes de apoio que terá durante o desenvolvimento. Tecnologias mais recentes, em geral, terão menos material de suporte, mas esse não é o requisito essencial. O mais importante é a qualidade do material.

Nas soluções que oferecem alternativa para uma camada web-mobile, como é o caso do PrimeFaces, é preciso desenvolver a

camada especificamente para o acesso mobile e outra para o acesso através do navegador, pelo desktop. Obviamente, hoje em dia, já é possível usar a interface mobile no navegador desktop, e vice-versa, mas isso limitará alguns recursos visuais. Normalmente, no entanto, quando há acessos ao sistema por tipos diferentes de equipamentos (móveis e desktops), é necessário implementar duas interfaces distintas, e isso significa mais codificação.

As bibliotecas que possuem componentes para atender o chamado design responsivo oferecem uma alternativa “polimórfica” para a interface, não sendo, portanto, necessário implementar uma interface para desktop e outra para dispositivo móvel. Assim, a mesma implementação da interface irá se comportar de modo adaptável ao tamanho da tela do dispositivo que a acessa. O que transparece nas implementações de interfaces responsivas é que o resultado visual nos dispositivos mobile não é tão próximo de uma tela de um aplicativo móvel tradicional (desses que precisam ser instalados). Já em interfaces produzidas com componentes do tipo web-mobile, como oferece o PrimeFaces (em conjunto com o jQuery), o resultado é um pouco mais similar, mas como já mencionado, exige mais código.

Nas **Figuras 1 a 6** são apresentados alguns exemplos de interfaces desenvolvidas com algumas das tecnologias mencionadas, os quais consideraremos como nossos protótipos.

The screenshot shows a web application interface. At the top, there is a header bar with a logo (a blue book icon), a 'Bookmarks' dropdown, and a 'Sair' (Logout) link. Below the header is a toolbar with a refresh icon and a trash bin icon. The main content area is titled 'Lista de Links' (List of Links) and contains a table with 19 rows of links. The table has columns for 'ID', 'Descrição' (Description), and 'Link'. The data is as follows:

ID	Descrição	Link
10	Demoiselle Portal	http://www.frameworkdemoiselle.gov.br
11	Demoiselle SourceForge	http://sf.net/projects/demoiselle
12	Twitter	http://twitter.frameworkdemoiselle.gov.br
13	Blog	http://blog.frameworkdemoiselle.gov.br
14	Wiki	http://wiki.frameworkdemoiselle.gov.br
15	Bug Tracking	http://tracker.frameworkdemoiselle.gov.br
16	Forum	http://forum.frameworkdemoiselle.gov.br
17	SVN	http://svn.frameworkdemoiselle.gov.br
18	Maven	http://repository.frameworkdemoiselle.gov.br
19	Downloads	http://download.frameworkdemoiselle.gov.br

At the bottom of the page, there is a footer bar with the text 'Aplicação de exemplo do Demoiselle 2.5.0-RC1'.

Figura 1. Tela de uma aplicação JSF (PrimeFaces) tradicional

Como definir a tecnologia para o desenvolvimento da interface web

Para produzi-las, foram utilizados arquétipos Maven oferecidos pelo framework Demoiselle. Cada arquétipo gera a mesma aplicação, na parte de modelo e negócio, mas com uma interface diferente para demonstrar as tecnologias citadas neste artigo.



Figura 2. Tela de uma aplicação JSF (PrimeFaces-Mobile) para web-mobile

A primeira delas mostra uma tela com dados em forma de tabela. Neste caso, a aplicação foi desenvolvida com a biblioteca PrimeFaces e componentes JSF e está sendo acessada por um equipamento do tipo desktop.

A Figura 2 mostra os mesmos dados apresentados na Figura 1. Porém, foram utilizados componentes do tipo web-mobile do PrimeFaces para construir a aplicação a ser acessada a partir de dispositivos móveis.

Na Figura 3 também temos os mesmos dados, mas a aplicação foi construída com a biblioteca Bootstrap, que viabiliza o design responsivo. Neste caso, acessamos a página a partir de um equipamento do tipo desktop. A Figura 4, por sua vez, mostra a mesma aplicação, porém sendo acessada por um dispositivo móvel.

Já as Figuras 5 e 6 mostram exemplos de uso do Crux Framework, a partir do qual também podemos construir aplicações responsivas. Na Figura 5 a aplicação está sendo acessada por um desktop e na Figura 6 por um dispositivo móvel.

Por fim, na Figura 7 temos um exemplo utilizando o Vaadin, que apresenta um resultado que lembra uma aplicação do tipo Java Swing para desktop.

É claro que todas as interfaces poderiam ser melhoradas e, conforme os componentes utilizados e a experiência do desenvolvedor com cada biblioteca, poderiam produzir resultados diferentes. O que foi demonstrado retrata apenas uma implementação básica de telas para expor os mesmos dados e com requisitos iguais, com tecnologias e bibliotecas diferentes a título de comparação.

Bookmark		Bookmark	secret	Sair
		<input type="radio"/> Novo <input checked="" type="radio"/> Listar		
Mostrar 10 registros				
▲ Description		Link		
<input type="checkbox"/> Portal		http://www.frameworkdemoiselle.gov.br		
<input type="checkbox"/> Documentação		http://demoiselle.sourceforge.net/docs/framework/reference		
<input type="checkbox"/> Fórum		http://pt.stackoverflow.com/tags/demoiselle		
<input type="checkbox"/> Lista de usuários		https://lists.sourceforge.net/lists/listinfo/demoiselle-users		
<input type="checkbox"/> Blog oficial		http://frameworkdemoiselle.wordpress.com		
<input type="checkbox"/> Blog experimental		http://demoisellelab.wordpress.com		
<input type="checkbox"/> Repositório		http://github.com/demoiselle/framework		
<input type="checkbox"/> Bug Tracker		https://demoiselle.atlassian.net		
<input type="checkbox"/> Facebook		http://facebook.com/FrameworkDemoiselle		

Figura 3. Tela de uma aplicação com design responsivo (Bootstrap) acessada de ambiente desktop

CURSOS ONLINE



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



CONHEÇA ALGUNS DOS CURSOS

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

Para mais informações :

www.devmedia.com.br/cursos/delphi

(21) 3382-5038

Como definir a tecnologia para o desenvolvimento da interface web

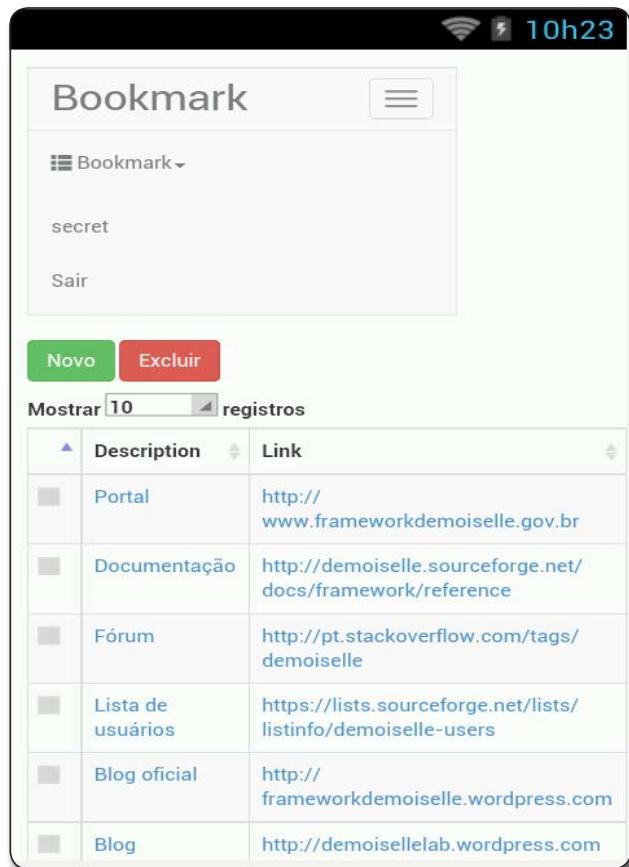


Figura 4. Tela de uma aplicação com design responsivo (Bootstrap) acessada de dispositivo móvel

Além disso, lembre-se que cada um dos exemplos apresentados poderá ter um Custo x Benefício diferente, que dependerá do perfil do usuário e da capacidade técnica do desenvolvedor.

A vantagem de criar protótipos funcionais é que o cliente poderá testar a aplicação do ponto de vista visual e ergonômico no equipamento de sua preferência (móvel ou desktop) e, ao mesmo tempo, podem ser feitos testes comparativos de performance e consumo de recursos computacionais provenientes dos componentes visuais no lado servidor e do tempo de resposta na apresentação das telas no lado cliente (navegador).

Nos experimentos que foram feitos para gerar os exemplos deste artigo, percebemos que as aplicações desenvolvidas com bibliotecas JavaScript na camada de visão e que acessam a camada de negócios através de serviços REST têm se mostrado mais econômicas com relação ao consumo de recursos do lado servidor. Por outro lado, o desenvolvimento ainda não conta com a quantidade de componentes e facilitadores que existem nas bibliotecas JSF, que apesar de consumirem um pouco mais de recursos do servidor, oferecem mais opções de itens como templates e temas para construir a interface, provavelmente por estarem a mais tempo no mercado. Isso pode refletir na produtividade e na qualidade visual das telas, levando-se em considerando que a questão de layout das telas poderá depender das preferências e perfil do usuário, que podem exigir a implementação de um recurso visual que não esteja disponível nas bibliotecas.

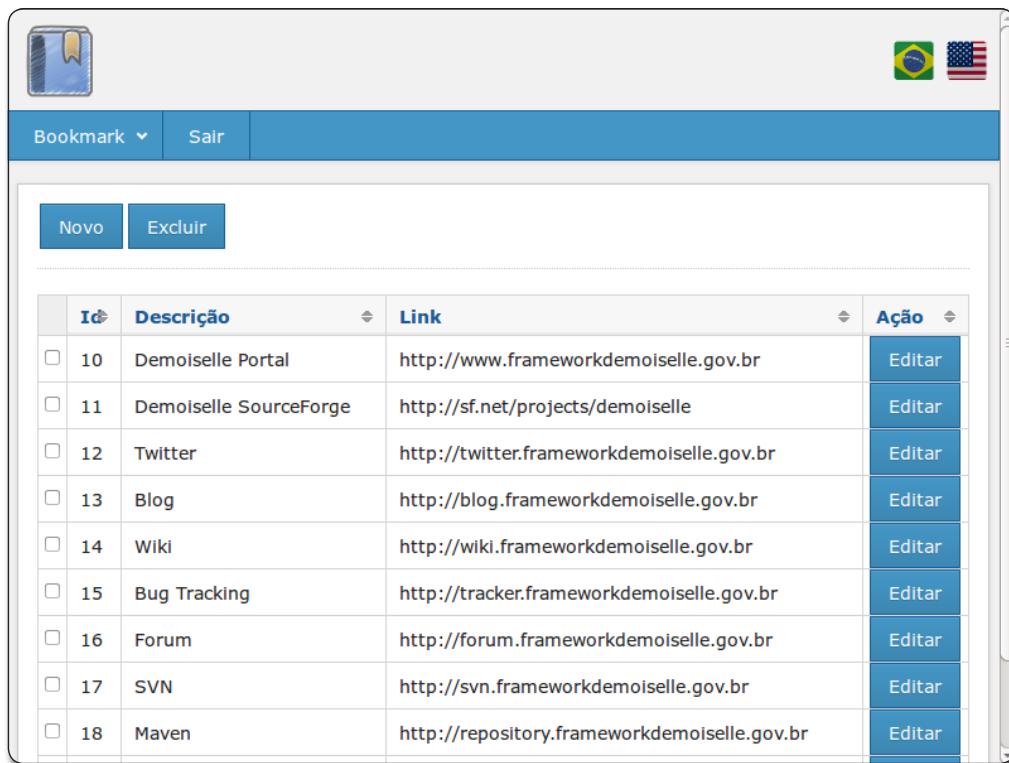


Figura 5. Tela de uma aplicação em Crux Framework acessada de ambiente desktop

Um fator determinante para a adoção de uma tecnologia que é comum a qualquer camada da aplicação é a capacidade do desenvolvedor, e quando se trata de uma equipe de desenvolvimento, esse fator pode ser ainda mais impactante, pois a capacidade de cada um dos membros influencia diretamente no resultado de todos. No caso de equipes, se o perfil predominante é de desenvolvedores com bastante experiência com a linguagem Java e boa capacidade de absorção de novos conhecimentos, a opção por tecnologias mais recentes e, portanto, com alguns recursos mais modernos, é recomendada. Entretanto, se o perfil é de pouca experiência ou resistência a mudanças, é melhor optar pela tecnologia que já é dominada pela equipe.

Há casos que a adoção de uma nova tecnologia pode ser bastante rápida e com resultado positivo e há outros que além de impactar no atraso do tempo total de desenvolvimento, também não geram bom resultado. Provavelmente o leitor já tenha lido ou ouvido uma frase do tipo: "A melhor linguagem (ou tecnologia) para desenvolver um sistema é aquela que você domina!". Uma construção menos restritiva desta frase seria dizer que a melhor linguagem é aquela que você pode dominar; ou dependendo da situação, aquela que você tem uma base de conhecimento e dispõe de tempo para aprender suficientemente. É importante ressaltar que jamais devemos nos limitar ao conhecimento que já temos, mas também devemos ser capazes de discernir quando não é viável arriscar.

The screenshot shows a mobile application interface. At the top, there's a header with a logo of a blue book with a yellow ribbon, the word 'Bookmark' in white, and two flags (Brazil and USA). Below the header are buttons for 'Novo' (New) and 'Excluir' (Delete). The main area is a table listing 19 bookmarks:

ID	Descrição	Link	Ação
10	Demoiselle Portal	http://www.frameworkdemoiselle.gov.br	Editar
11	Demoiselle SourceForge	http://sf.net/projects/demoiselle	Editar
12	Twitter	http://twitter.frameworkdemoiselle.gov.br	Editar
13	Blog	http://blog.frameworkdemoiselle.gov.br	Editar
14	Wiki	http://wiki.frameworkdemoiselle.gov.br	Editar
15	Bug Tracking	http://tracker.frameworkdemoiselle.gov.br	Editar
16	Forum	http://forum.frameworkdemoiselle.gov.br	Editar
17	SVN	http://svn.frameworkdemoiselle.gov.br	Editar
18	Maven	http://repository.frameworkdemoiselle.gov.br	Editar
19	Downloads	http://download.frameworkdemoiselle.gov.br	Editar

Figura 6. Tela de uma aplicação em Crux Framework acessada de dispositivo móvel

The screenshot shows a desktop application window titled 'Bookmark Manager'. The interface includes a toolbar with 'New', 'Help', 'Bookmarks' (selected), 'Save', 'Delete', and 'Clear' buttons. Below the toolbar is a form with fields for 'Category' (set to 'Bookmark's Category'), 'Description' (set to 'Bookmark's Description'), 'Link*' (set to 'Bookmark's Link'), and a 'Visited?' checkbox (unchecked). At the bottom of the form are buttons for 'Save', 'Delete', and 'Clear'. The main area is a table titled 'Bookmark List' with columns: CATEGORY, DESCRIPTION, ID, LINK, and VISITED?. The data in the table matches the list from Figure 6:

CATEGORY	DESCRIPTION	ID	LINK	VISITED?
Demoiselle	Demoiselle Portal	1	http://www.frameworkdemoiselle.gov.br	false
Demoiselle	Demoiselle SourceForge	2	http://sf.net/projects/demoiselle	false
Demoiselle	Twitter	3	http://twitter.frameworkdemoiselle.gov.br	false
Demoiselle	Blog	4	http://blog.frameworkdemoiselle.gov.br	false
Demoiselle	Wiki	5	http://wiki.frameworkdemoiselle.gov.br	false
Demoiselle	Bug Tracking	6	http://tracker.frameworkdemoiselle.gov.br	false
Demoiselle	Forum	7	http://forum.frameworkdemoiselle.gov.br	false
Demoiselle	SVN	8	http://svn.frameworkdemoiselle.gov.br	false
Demoiselle	Maven	9	http://repository.frameworkdemoiselle.gov.br	false
Demoiselle	Downloads	10	http://download.frameworkdemoiselle.gov.br	false

Figura 7. Tela de uma aplicação em Vaadin acessada por equipamento desktop

Por fim, é necessário preparar um relatório que aponte cada ponto positivo e negativo na adoção de cada uma das opções disponíveis, e que contenha também os requisitos de maior impacto no desenvolvimento, como o cronograma com a quantidade de tempo prevista para implementação e o volume estimado de recursos financeiros disponíveis para o projeto. Procure destacar os pontos que considera mais importantes, de acordo com o perfil do usuário.

Não existe bala de prata

A maioria dos profissionais de TI também já deve ter lido ou ouvido falar no termo “bala de prata”, uma expressão para denominar uma fictícia solução mágica que resolveria todos os problemas. Até pouco tempo era comum a divulgação de propagandas que mostravam produtos novos como sendo fantásticos e definitivos. Hoje em dia já não se apela tanto. Mas ainda é possível encontrar menções sobre determinada tecnologia como aquela com força. O próprio Java, que surgiu como uma das primeiras alternativas para o desenvolvimento para internet por volta dos anos 90, chegou a ser considerado ultrapassado para esse tipo de ambiente em meados dos anos 2000, mas depois de algumas mudanças na especificação e com a evolução da tecnologia, o Java voltou a ser uma das melhores opções para criação de aplicações web.

Com base nisso, em vez de acreditar apenas naquilo que lemos ou ouvimos de outras pessoas, devemos considerar nossa própria capacidade de aprender e avaliar qualquer tecnologia ou conhecimento, e concomitantemente, termos o discernimento para que não sejamos precipitados na adoção das novidades ou, no outro extremo, ficarmos parados no tempo.

Assim, este artigo termina sem indicar ou mesmo sugerir uma determinada tecnologia, pois esta conclusão deve ser alcançada de acordo com o próprio desenvolvedor ou equipe, quando for o caso, e também considerando o cenário/contexto e os requisitos para construção do sistema. O objetivo mais importante aqui foi viabilizar ao leitor um conhecimento inicial que seja suficiente para que ele possa estruturar e apresentar um estudo com a finalidade de propor a adoção da tecnologia mais adequada para o projeto.

Autor



Emerson Sachio Saito

emerson.saito@serpro.gov.br - <http://blogoosfera.cc/esaito/blog>
Bacharel em Análise de Sistemas pela PUC-PR, com especialização em Tecnologia da Informação pela UFPR. Trabalha há mais de 12 anos com a tecnologia Java. Trabalhou como analista de informática na CELEPAR (Companhia de informática do Paraná) de 2001 a 2009, onde entre outras diversas atividades, foi um dos desenvolvedores da plataforma de desenvolvimento Java/WEB chamada Pinhão(<http://www.frameworkpinhao.pr.gov.br>). Desde 2009 é analista de desenvolvimento no SERPRO (Serviço Federal de Processamento de Dados), onde faz parte da equipe dedicada ao desenvolvimento do Framework Demoiselle (<http://www.demaiselle.org>).



Links:

Página do ICEfaces.

<http://www.icesoft.org/java/projects/ICEfaces/overview.jsf>

Página do MyFaces.

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

Página do RichFaces.

<http://richfaces.jboss.org>

Página do projeto PrimeFaces.

<http://primefaces.org>

Página do Bootstrap.

<http://getbootstrap.com>

Página do AngularJS

<https://angularjs.org>

Site do Crux Framework.

<http://www.cruxframework.org>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



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**

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