



Edição 40



Aprendendo a utilizar o JTable

Como criar JTables com
acesso a banco de dados

Debug no Eclipse

Elimine os erros do seu código

SPRING MVC E GAE

Simplificando o
desenvolvimento
para a nuvem



ISSN 2179625-4





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.**

➤ A linguagem Java está presente em complexos sistemas como em aplicativos mobile e desktop, servindo a milhares de pessoas. Inicie sua carreira e certifique-se no instituto que valoriza seu currículo. Faça Infnet!

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer



Sumário

Artigo no estilo Solução Completa

06 – JTable: Aprendendo a utilizar tabelas em Java

[John Soldera]

Artigo no estilo Solução Completa

16 – Google App Engine: Utilizando Spring MVC

[Marcos Alexandre Vidolin de Lima]

Conteúdo sobre Boas Práticas, Artigo no estilo Curso

30 – Eclipse Debug: Conheça os recursos de Debug do Eclipse – Parte 2

[José Fernandes A. Júnior]

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



Edição 40 • 2014 • ISSN 2173625-4



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 Davi Costa (davigc_08@hotmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

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.

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 Easy 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

DÊ UM SALTO EM CONHECIMENTO!



Acesse o maior
portal para
desenvolvedores
da América
Latina!





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



JTable: Aprendendo a utilizar tabelas em Java

O **JTable** é um componente de interface gráfica que permite manipular planilhas (tabelas) de dados, possibilitando ao usuário informar o valor de cada célula a fim de preencher/atualizar a tabela. Como este é um componente para exibição de tabelas, ele não realiza a persistência dos dados. Assim, quando a GUI é fechada, os dados informados no **JTable** são perdidos.

Com o objetivo de tornar os dados exibidos no **JTable** persistentes e, ao mesmo tempo, oferecer uma interface gráfica fácil para inserir, editar e excluir registros, este artigo propõe a construção de uma aplicação exemplo que tem o intuito de gerenciar uma tabela de cursos de um banco de dados qualquer usando os componentes da API Swing, focando, principalmente, no componente **JTable**, para realização da entrada de dados na tabela e exibição dos registros.

A API Swing é parte do conjunto das classes básicas da linguagem Java chamado de Java Foundation Classes (JFC). Este oferece uma série de pacotes com classes que servem de apoio para os mais diversos tipos de aplicações, onde está contido o Swing, que foi criado para ser uma opção ao AWT (*Abstract Window Toolkit*), oferecendo componentes GUI com melhor aparência gráfica e com mais opções disponíveis para customização.

Sendo um componente do Swing, o **JTable** é inteiramente customizável e oferece métodos para acessar, inserir e excluir registros, adicionar *listeners* (eventos) e outras funções como edição de propriedades das células da tabela. A partir desses recursos, e pensando na exibição de dados recuperados de um banco de dados, podemos utilizar *listeners* para chamar métodos específicos de acordo com cada tipo de iteração do usuário com o **JTable**, possibilitando assim manter os dados do banco e do componente sincronizados.

Os principais construtores e métodos da classe **JTable** são:

- **JTable():** Constrói um novo **JTable** que é inicializado com o modelo de dados default, modelo de coluna default e modelo de seleção default;
- **JTable(int numRows, int numColumns):** Constrói um novo **JTable** com o número de linhas e colunas indicados e preenche a tabela com células vazias;
- **JTable(Object[][] rowData, Object[] columnNames):** Constrói um novo **JTable** com o array de dados e os nomes de colunas indicados;

Fique por dentro

Este artigo é útil por apresentar, na teoria e na prática, os principais construtores e métodos da classe **JTable**, demonstrando que é possível customizar diversos elementos deste componente que representa uma tabela, como o título das colunas, o modo de seleção das células, o formato gráfico da tabela, além de oferecer comandos para acesso a dados selecionados, listeners para eventos de alteração na tabela, e métodos para manipulação dos dados presentes nesta. Para abordar tudo isso, uma aplicação exemplo em Swing fará uso de um **JTable** para exibir e editar os registros de uma tabela do banco de dados.

- **JTable(TableModel dm):** Constrói um novo **JTable** usando o modelo de tabela indicado;
- **JTable(TableModel dm, TableColumnModel cm):** Constrói um novo **JTable** com o modelo de tabela e o modelo de coluna indicados;
- **JTable(TableModel dm, TableColumnModel cm, ListSelectionModel sm):** Constrói um novo **JTable** usando o modelo de tabela, modelo de coluna e modelo de seleção indicados;
- **JTable(Vector rowData, Vector columnNames):** Constrói um novo **JTable** usando os nomes das colunas indicadas no vetor **columnNames** e os dados contidos no vetor **rowData**;
- **void addColumn(TableColumn coluna):** Adiciona uma nova coluna no **JTable**;
- **void clearSelection():** Desmarca a seleção de qualquer linha ou coluna;
- **int getEditingColumn():** Retorna o índice da coluna sendo editada;
- **int getEditingRow():** Retorna o índice da linha sendo editada;
- **Class getColumnClass(int column):** Retorna a classe da coluna, ou seja, retorna a classe possível de ser aplicada naquela coluna;
- **public TableColumnModel getColumnModel():** retorna o modelo de coluna da tabela;
- **TableModel getModel():** Retorna o modelo da tabela que está sendo usado para a exibição dos dados da tabela;
- **ListSelectionModel getSelectionModel():** Retorna o **ListSelectionModel** associado ao **JTable**;
- **Object getValueAt(int row, int column):** Retorna o valor da célula dada a linha e a coluna;
- **boolean isEditing():** Retorna **true** se existe uma célula sendo editada;

- **void moveColumn(int column, int targetColumn):** move a coluna indicada para a posição indicada;
- **void setColumnModel(TableColumnModel columnModel):** Permite definir o modelo de colunas;
- **void setColumnSelectionAllowed(boolean columnSelectionAllowed):** Define se colunas inteiras podem ser selecionadas;
- **void setColumnSelectionInterval(int index0, int index1):** Seleciona colunas no intervalo de **index0** para **index1** (inclusive);
- **void setModel(TableModel dataModel):** Define o modelo da tabela;
- **void setSelectionMode(int selectionMode):** Define o modelo de seleção de colunas, que pode ser coluna simples, um intervalo de colunas ou múltiplos intervalos;
- **void setSelectionModel(ListSelectionModel newModel):** Define o modelo de seleção de colunas;
- **void setShowGrid(boolean showGrid):** Define se é desenhada a grade de linhas ao redor das células da tabela;
- **void setValueAt(Object aValue, int row, int column):** Define o valor da célula na linha e coluna indicadas, atualizando o **TableModel** usado;
- **void tableChanged(TableModelEvent e):** Permite definir um *listener* (**TableModelEvent**) que é chamado no caso de modificação do **JTable**.

Através dos métodos apresentados, pode-se verificar que **JTable** é uma classe gerenciável através do uso de interfaces e uma das principais é a **TableModel**. Esta define métodos para realizar a manipulação das células da tabela, como acessar o valor e as devidas propriedades, incluindo também o acesso aos nomes das colunas. Os principais métodos da interface **TableModel** são:

- **void addTableModelListener(TableModelListener l):** Adiciona um *listener* à tabela que é chamado quando ocorre uma modificação em uma célula;
- **Class getColumnClass(int columnIndex):** Retorna a classe mais comum em todas as células daquela coluna, pois os campos podem ser de diferentes tipos;
- **int getColumnCount():** Retorna a quantidade de colunas da tabela;
- **String getColumnName(int columnIndex):** Retorna o nome de uma coluna, dado o seu índice;
- **int getRowCount():** Retorna o número de linhas da tabela;
- **Object getValueAt(int rowIndex, int columnIndex):** Retorna o valor da célula na posição indicada;
- **boolean isCellEditable(int rowIndex, int columnIndex):** Retorna verdadeiro se a célula indicada é editável;
- **void removeTableModelListener(TableModelListener l):** Remove o *listener* de modificações da tabela;
- **void setValueAt(Object aValue, int rowIndex, int columnIndex):** permite definir o valor de uma célula na tabela.

Como **TableModel** é uma interface, tais métodos precisam ser implementados em uma classe filha para serem utilizados.

Uma classe importante que já implementa **TableModel** é a **DefaultTableModel**, presente na API Swing. Esta classe será empregada no exemplo desse artigo para a realização da manipulação dos dados e dos nomes das colunas da nossa tabela.

Outra interface importante é a **ListSelectionModel**. Esta interface contém diversos métodos para acessar elementos selecionados na tabela, definir o modo de seleção das células, linhas ou colunas, desmarcar seleções de elementos e definir *listeners* para eventos de edição da tabela.

Por último, temos a interface **TableColumnModel**, que permite acessar as definições das colunas, como comprimento e largura das mesmas, modo de seleção, entre outros recursos.

A partir das classes e interfaces apresentadas, pode-se gerenciar todos os aspectos gráficos do componente **JTable**. Porém, como já informado, tal componente não realiza o acesso ao banco de dados. Pensando nisso, com o intuito de mostrar como pode ser implementada a persistência de dados presentes no **JTable**, vamos desenvolver uma aplicação exemplo.

Persistência dos dados

Como o **JTable** não é um componente com acesso direto ao banco de dados, para realizar a persistência dos dados exibidos nele, na **Listagem 1** é proposta a classe **AcessoDatabase**. Esta tem a função exclusiva de acessar o banco de dados. Para simplificar a organização da nossa aplicação, todas as funcionalidades de acesso ao banco de dados serão centralizadas nessa classe. Assim, **AcessoDatabase** é responsável pelas operações de criar a conexão com o banco de dados, criar, alterar, excluir e consultar os registros presentes na tabela Cursos.

A tabela gráfica representada pelo **JTable** será o espelho da tabela Cursos do banco de dados. Nesta tabela, um curso é definido por três campos de texto: **nome**, **tipo** e **duracao**. O gerenciamento da tabela Cursos no banco de dados ocorre pelo uso da linguagem SQL (*Structured Query Language*), que suporta comandos de consulta sobre os dados das tabelas do banco de dados, além de comandos para criação de tabelas, criação, edição e exclusão de registros e outras operações em geral.

Os comandos SQL são representados na linguagem Java na forma de Strings e tais comandos são executados usando **Statements**, que nada mais são do que objetos criados pela conexão com o banco de dados com o fim de executar os comandos SQL.

Como peça chave necessária à persistência dos dados, o banco de dados utilizado nesse artigo foi o Java DB, conhecido como Derby. Este é um banco de dados relacional (RDBMS) no qual os dados são gravados ou recuperados pelo uso da linguagem SQL. O Java DB é disponibilizado no site da Oracle ou Apache e está embutido no JDK. Para importar as bibliotecas do Java DB para o nosso projeto, é necessário adicionar no *build path* todas as bibliotecas do diretório: `\jdk<versão>\db\lib`.

Iniciando a declaração do código fonte, na **Listagem 1** é apresentado o cabeçalho da classe **AcessoDatabase**, incluindo seus *imports* e variáveis estáticas. A primeira variável declarada foi a **URL**, responsável por conter o endereço para conexão com o banco

JTable: Aprendendo a utilizar tabelas em Java

de dados, e a segunda foi o **DRIVER**, que armazena o nome da classe do *driver* de acesso ao banco.

Na **Listagem 2** é apresentado mais um trecho da classe **AcessoDatabase**, o método **getConnection()**. Observe, na linha 18, que este método utiliza **URL** para criar a conexão com o banco. Neste momento, se a base de dados não existir, ela é criada automaticamente pelo Derby.

Listagem 1. Cabeçalho da classe **AcessoDatabase**.

```
01. package pkg;
02. import java.sql.Connection;
03. import java.sql.DriverManager;
04. import java.sql.ResultSet;
05. import java.sql.SQLException;
06. import java.util.Vector;
07. import java.sql.Statement;
08.
09. public class AcessoDatabase {
10.
11.     private static final String URL = "jdbc:derby:derbyTestDatabase;create=true;
user=derby;password=derby";
12.
13.     private static final String DRIVER = "org.apache.derby.jdbc.EmbeddedDriver";
```

Listagem 2. Código do método **getConnection()**.

```
14.     public static Connection getConnection() {
15.         System.out.println("Conectando ao Banco de Dados.");
16.         try {
17.             Class.forName(DRIVER);
18.             Connection conexao = DriverManager.getConnection(URL);
19.             System.out.println("Conexão Criada com Sucesso.");
20.             return conexao;
21.         } catch (ClassNotFoundException e) {
22.             System.out.println("Driver Não Encontrado.");
23.             System.exit(0);
24.         } catch (SQLException e) {
25.             System.out.println("Erro ao Acessar a Base.");
26.             System.exit(0);
27.         }
28.         return null;
29.     }
```

Nota-se que na linha 17 é carregado o *driver* de acesso ao banco de dados, declarado na linha 13 da **Listagem 1**. O *driver* JDBC tem a função de prover a ponte de comunicação entre a aplicação Java e o banco de dados utilizado. Depois disso, a conexão com o banco de dados é criada na linha 18, usando a URL de conexão. Como podem ocorrer erros tanto no carregamento do *driver* como na criação da conexão, são usados os comandos *try/catch* para capturar qualquer erro e mostrar a mensagem correspondente no console, finalizando a aplicação com o método **System.exit()** no caso de erro.

Continuando a declaração da classe **AcessoDatabase**, na **Listagem 3** é apresentado o método **getExamples()**, que tem a função de criar alguns cursos de exemplo em memória para inicializar a base de dados da nossa aplicação.

Este método será usado para criar dados de exemplo de cursos que, em outra classe, serão gravados na tabela **Cursos** na criação da base de dados na primeira vez que a aplicação for executada. Em **getExamples()**, os campos de cada curso são representados através de um array de **String**, como na linha 34, e um **Vector** é usado para representar uma coleção de cursos (linha 33).

Diferentemente de **getExamples()**, o método **inserirCurso()**, exposto na **Listagem 4**, tem a capacidade de alterar o estado do banco de dados, sendo um dos principais métodos relacionados à persistência dos dados. Este tem a função de armazenar os dados recebidos por parâmetro na tabela **Cursos**, e para isso, faz uso do comando SQL *insert*.

Listagem 3. Código do método **getExamples()**.

```
30.
31.     public static Vector<String[]> getExamples() {
32.         Vector<String[]> examples = new Vector<String[]>();
33.         String[] curso1 = {"Ciências da Computação", "Exatas", "5 anos"};
34.         examples.add(curso1);
35.         String[] curso2 = {"Medicina", "Humanas", "6 anos"};
36.         examples.add(curso2);
37.         String[] curso3 = {"Engenharia Elétrica", "Exatas", "5 anos"};
38.         examples.add(curso3);
39.         return examples;
40.     }
41. }
```

Listagem 4. Código do método **inserirCurso()**.

```
42.     public static boolean inserirCurso (Connection conexao, String nome,
String tipo, String duracao) {
43.         try {
44.             Statement st = conexao.createStatement();
45.             st.execute("insert into Cursos (nome, tipo, duracao)
values ('" + nome + "','" + tipo + "','" + duracao + "')");
46.             st.close();
47.             return true;
48.         } catch (SQLException e) {
49.             System.out.println(e.getMessage());
50.             return false;
51.         }
52.     }
```

Como podemos verificar, o método **inserirCurso()** realiza a persistência de qualquer novo curso e se ocorrer erro na inserção, é mostrada uma mensagem de erro e o método retorna falso.

Continuando a implementação dos métodos relacionados à persistência dos dados, para a alteração dos dados, foi implementado o método **atualizarCurso()**, apresentado na **Listagem 5**. O curso a ser alterado é determinado pelo **nome** e os campos a serem redefinidos são o **tipo** e a **duracao**. O comando SQL executado por este método é o *update*.

Na **Listagem 6** é apresentado o método **apagarCurso()**, que tem a função de eliminar registros da tabela **Cursos**. Para isso, faz-se uso do comando SQL *delete*, apagando um curso do banco de dados de acordo com o seu **nome**.

Em seguida, foi implementado o método **getCursos()**, apresentado na **Listagem 7**. Este método tem a função de consultar todos os registros da tabela **Cursos**. Tal operação é realizada pelo comando SQL *select* sem critérios de busca.

Listagem 5. Código do método atualizarCurso().

```
53. public static boolean atualizarCurso (Connection conexao,
String nome, String tipo, String duracao) {
54.     try {
55.         Statement st = conexao.createStatement();
56.         st.execute("update Cursos set tipo=" + tipo +
57.                    ",duracao=" + duracao + " where nome = " + nome + "");
58.         st.close();
59.     } catch (SQLException e) {
60.         System.out.println(e.getMessage());
61.         return false;
62.     }
63. }
```

Listagem 6. Código do método apagarCurso().

```
64. public static boolean apagarCurso (Connection conexao, String nome) {
65.     try {
66.         Statement st = conexao.createStatement();
67.         st.execute("delete from Cursos where nome = " + nome + "");
68.         st.close();
69.     } catch (SQLException e) {
70.         System.out.println(e.getMessage());
71.         return false;
72.     }
73. }
74. }
```

Listagem 7. Código do método getCursos().

```
75. public static Vector<String[]> getCursos(Connection conexao) {
76.     Vector<String[]> cursos = new Vector<String[]>();
77.     try {
78.         Statement st = conexao.createStatement();
79.         ResultSet rs = st.executeQuery("select * from Cursos");
80.         while (rs.next()) {
81.             String nome = rs.getString(1);
82.             String tipo = rs.getString(2);
83.             String duracao = rs.getString(3);
84.             String[] curso = new String[]{nome, tipo, duracao};
85.             cursos.add(curso);
86.         }
87.         rs.close();
88.         st.close();
89.     } catch (SQLException e) {
90.         System.out.println(e.getMessage());
91.     }
92.     return cursos;
93. }
```

Como próximo passo, foi implementado o método **criarTabela()** – vide Listagem 8. Este tem a função de realizar a criação da tabela Cursos no banco de dados a partir do comando SQL *create table*. Esta tabela é definida com três campos de texto (*varchar*): **nome**, **tipo** e **duração**. Se a tabela for criada com sucesso, é retornado verdadeiro, caso contrário, a mensagem de erro é impressa no console e o valor **false** é retornado.

Outra funcionalidade importante é a de exclusão da tabela, implementada na Listagem 9, no método **apagarTabela()**. Para isso, este método faz uso do comando SQL *drop table*.

Uma observação importante a respeito dos métodos que realizam a persistência dos dados é a necessidade de fechar cada **Statement** com o método **close()** para garantir a integridade dos dados.

A partir da criação de **AcessoDatabase**, a GUI não precisará criar e executar explicitamente comandos SQL, promovendo assim uma boa separação entre as camadas da aplicação.

Listagem 8. Código do método criarTabela().

```
94. public static boolean criarTabela (Connection conexao) {
95.     try {
96.         Statement st1 = conexao.createStatement();
97.         st1.execute("create table Cursos (nome varchar(100),
98.                    tipo varchar(100), duracao varchar(100))");
99.         st1.close();
100.    } catch (SQLException e) {
101.        System.out.println(e.getMessage());
102.        return false;
103.    }
104. }
```

Listagem 9. Código do método apagarTabela().

```
105. public static boolean apagarTabela (Connection conexao) {
106.
107.    try {
108.        Statement st1 = conexao.createStatement();
109.        st1.execute("drop table Cursos");
110.        st1.close();
111.    } catch (SQLException e) {
112.        System.out.println(e.getMessage());
113.        return false;
114.    }
115. }
116. }
117. }
```

**Não perca tempo
reinventando a roda!**

COBREBEMX

**Componente completo para sua
Cobrança por Boleto Bancário
e Débito em Conta Corrente**

Mais de 40 exemplos
em diversas linguagens
de programação

Geração e leitura de arquivos
(remessa e retorno) nos padrões
FEBRABAN e CNAB

Testes e Downloads
gratuitos em nosso site

ACESSE E CONHEÇA O COMPONENTE EM:
WWW.COBREBEM.COM

Construindo a GUI da aplicação exemplo

Agora que foi apresentada a camada de persistência, é necessário criar uma interface gráfica que permita ao usuário manipular os registros da tabela Cursos sem que ele precise conhecer detalhes do funcionamento do banco de dados. Com esse intuito, nas Listagens 10 e 11 é apresentada a classe **CursosApp**, que tem a função de oferecer uma interface gráfica simples e eficiente para manipular a tabela Cursos usando componentes gráficos da API Swing, como é o caso do **JTable**, empregado para exibir os registros desta tabela e viabilizar sua edição.

A GUI proposta na classe **CursosApp** contém botões de ação no topo, para inserir e remover cursos, e o restante de sua área gráfica é ocupada por um **JTable** que exibe os dados registrados no banco de dados. Para que a aplicação grave automaticamente todas as alterações feitas em qualquer campo quando o **JTable** for editado, deve-se definir um *listener* no **JTable**. Deste modo, quando uma célula tiver seu valor modificado, o curso correspondente será atualizado no banco de dados. A **Figura 1** apresenta a interface gráfica da nossa aplicação.



Figura 1. Interface gráfica da aplicação exemplo

Na Listagem 10 é apresentada a primeira parte da classe **CursosApp**, contendo a declaração, alocação e configuração dos componentes da GUI. Além disso, são criados *listeners*, como o *listener* para o botão *Adicionar Curso*, que corresponde à classe **AdicionarCursoListener**, usada para chamar uma janela auxiliar a ser definida mais adiante. Já o botão *Remover Curso Selecionado* não necessita de uma janela auxiliar, pois ele simplesmente remove o curso selecionado usando o código definido no *listener* **RemoverCursoListener**.

Para inicializar a GUI, o construtor da classe **CursosApp** (veja a linha 18) tem a função de criar cada componente da GUI e de configurá-los para serem exibidos de acordo com a **Figura 1**. Ainda na linha 18, pode-se verificar que o construtor recebe como parâmetros uma conexão com o banco de dados e uma lista de cursos, a fim de que tais variáveis sejam acessíveis pela GUI.

Logo no início do processamento do construtor, o tamanho da janela principal é definido. Depois disso, na linha 21 é instanciado o modelo de tabela que será usado pelo **JTable**, **ModeloTabela**, com a função de gerenciar o acesso aos dados da tabela e também aos nomes das colunas. Nas linhas 22 a 24 são definidos os nomes das colunas que serão exibidos no **JTable** e repassados ao modelo de tabela. Uma vez que o modelo de tabela foi criado, é necessário obter os dados do banco de dados para alimentá-lo, o que é feito

nas linhas 26 a 28, onde cada curso recebido como parâmetro no construtor é adicionado ao modelo para exibição no **JTable**.

Listagem 10. Código da classe **CursosApp** – Parte 1.

```
01. package pkg;
02. import java.awt.BorderLayout;
03. import java.sql.Connection;
04. import java.util.Vector;
05. import javax.swing.JButton;
06. import javax.swing.JFrame;
07. import javax.swing.JPanel;
08. import javax.swing.JScrollPane;
09. import javax.swing.JTable;
10. import javax.swing.table.DefaultTableModel;
11.
12. public class CursosApp extends JFrame {
13.
14.     private static final long serialVersionUID = 1L;
15.     private DefaultTableModel modelo;
16.     private JTable tabela;
17.
18.     public CursosApp(Connection con, Vector<String[]> cursos) {
19.
20.         setSize(500, 300);
21.         modelo = new ModeloTabela();
22.         modelo.addColumn("Curso");
23.         modelo.addColumn("Tipo");
24.         modelo.addColumn("Duração");
25.
26.         for (String[] linha: cursos) {
27.             modelo.addRow(linha);
28.         }
29.         tabela = new JTable(modelo);
30.         tabela.getSelectionModel().addListSelectionListener(new
31.             EventoTabelaListener(tabela, modelo, con));
32.         JButton addButton = new JButton("Adicionar Curso");
33.         addButton.addActionListener(new AdicionarCursoListener
34.             (tabela, modelo, con, this));
35.         JButton removeButton = new JButton("Remover Curso Selecionado");
36.         removeButton.addActionListener(new RemoverCursoListener
37.             (tabela, modelo, con));
38.         this.addWindowListener(new WindowListener(tabela, modelo, con));
39.
40.         JPanel painel = new JPanel();
41.         painel.add(addButton);
42.         painel.add(removeButton);
43.
44.         getContentPane().add(new JScrollPane(tabela), BorderLayout.CENTER);
45.         getContentPane().add(painel, BorderLayout.NORTH);
46.     }
```

Com o modelo de tabela alimentado com os dados, o **JTable** é criado na linha 29 recebendo como parâmetro este modelo. A fim de gerenciar as mudanças no **JTable** realizadas pelo usuário, na linha 30 é criado o *listener* para os eventos de modificação do **JTable** usando a classe **EventoTabelaListener**, a ser definida posteriormente.

Depois disso, na linha 32 é criado o botão para inserir novos cursos na tabela e na linha 33 é criado o *listener* para esse botão com a função de instanciar a classe **AdicionarCursoListener**, que será definida posteriormente.

Já na linha 35 é criado o botão para remover os cursos da tabela e na linha 36 é criado o *listener* para o mesmo, instanciando a classe **RemoverCursoListener**, a ser definida posteriormente. O último *listener* é criado na linha 38, com a função de fechar a GUI, mas antes salvando as alterações pendentes nos dados exibidos no **JTable**, função que é realizada pelo *listener* **WindowListener**.

Por fim, nas linhas 40 a 45 são organizados os botões da janela na parte superior e o **JTable** na parte inferior.

Continuando a declaração da classe **CursosApp**, o método **main()** é apresentado na **Listagem 11**. Com a função de iniciar a aplicação, ele cria uma conexão com o banco de dados, consulta os cursos existentes na tabela Cursos do banco de dados (para exibição) e inicia a GUI.

Listagem 11. Método Main da Classe CursosApp.

```
47.  
48. public static void main(String args[]) {  
49.     try {  
50.         Connection con = AcessoDatabase.getConnection();  
51.         System.out.println("Conexão:" + con.toString());  
52.         //AcessoDatabase.apagarTabela(con);  
53.         Vector<String> cursos;  
54.         boolean sucesso = AcessoDatabase.criarTabela(con);  
55.         if (sucesso) {  
56.             cursos = AcessoDatabase.getExamples();  
57.             for (String[] curso : cursos) {  
58.                 AcessoDatabase.inserirCurso(con, curso[0], curso[1], curso[2]);  
59.             }  
60.         }  
61.     }  
62.     else {  
63.         cursos = AcessoDatabase.getCursos(con);  
64.     }  
65.     CursosApp app = new CursosApp(con, cursos);  
66.     app.setVisible(true);  
67.     } catch (Exception e){  
68.     e.printStackTrace();  
69. }  
70. }
```

Analisando este código, na linha 50 é criada uma conexão com a base de dados usando o método **getConnection()**. Na linha 54, é criada a tabela Cursos no banco a partir da execução do método **criarTabela()**. Se a tabela foi criada com sucesso, são obtidos dados de exemplo na linha 56 com o método **getExamples()** e estes inseridos na tabela na linha 58. Por outro lado, se ocorreu erro na criação da tabela (neste caso, por ela já existir), os dados desta são consultados na linha 62 com o método **getCursos()**.

Finalmente, a GUI é instanciada na linha 64, recebendo como parâmetros a conexão criada e a lista de cursos obtida, e a janela principal se torna visível na linha 65. Uma observação importante é que a linha 52 pode ser descomentada a fim de eliminar a tabela do banco de dados no início da aplicação. Isto deve ser feito caso o usuário tenha interesse em executar a aplicação exemplo como se fosse a primeira vez. Deste modo será executado o método referente à remoção da tabela Cursos do banco de dados.

Com a construção da classe que representa a janela principal da aplicação pronta, vamos analisar as classes auxiliares. Uma delas

é a **ModeloTabela**, que gerencia o acesso aos dados e aos nomes das colunas no **JTable**, apresentada no tópico a seguir.

O modelo de tabela do JTable da nossa aplicação

O modelo de tabela utilizado pelo **JTable** na aplicação exemplo é dado pela classe **ModeloTabela**, que herda os métodos de **DefaultTableModel** para gerenciamento dos dados e dos nomes das colunas. Tais métodos podem ser sobreescritos a fim de customizar o funcionamento de **DefaultTableModel**, como ocorre na **Listagem 12**, onde é sobreescrito o método **isCellEditable()** com o objetivo de não permitir a edição do campo *nome do curso* de qualquer registro.

O método **isCellEditable()** tem a função de indicar os campos que podem ser alterados pelo usuário, recebendo como parâmetros a linha e a coluna da célula a ser verificada. Assim, para evitar a edição de qualquer campo da coluna *nome do curso*, o método **isCellEditable()** foi implementado de forma a retornar falso quando o índice da coluna for zero (valor que representa essa coluna).

Listagem 12. Código da classe **ModeloTabela**.

```
01. package pkg;  
02.  
03. import javax.swing.table.DefaultTableModel;  
04. public class ModeloTabela extends DefaultTableModel {  
05.  
06.     private static final long serialVersionUID = 1L;  
07.  
08.     public boolean isCellEditable(int row, int column) {  
09.         return column >= 1;  
10.     }  
11. }
```

Como visto na classe **ModeloTabela**, somente um método da classe **DefaultTableModel** foi sobreescrito, porém a aplicação exemplo pode ser expandida pelo leitor de forma a necessitar da sobreescrita de outros métodos provenientes de **DefaultTableModel** e/ou **TableModel**.

Listener para o botão Adicionar Curso

Quando o usuário clicar no botão *Adicionar Curso* da GUI, será chamado o *listener* **AdicionarCursoListener**. Este tem a função de abrir uma janela secundária para receber do usuário os dados do novo curso. Na confirmação da adição do novo curso (ou cancelamento da adição), a janela auxiliar é fechada e o *listener* ganha o controle novamente da execução, salvando o curso no banco de dados e exibindo-o no **JTable**. Na **Listagem 13**, é mostrado o código desse *listener*.

A fim de manter o acesso ao **JTable** e demais objetos importantes, o construtor de **AdicionarCursoListener**, na linha 11, recebe o **JTable**, o modelo da tabela, a conexão com o banco de dados e a janela principal da aplicação (**JFrame**). Assim, ele pode acessar tais objetos durante o processamento da ação. Voltando à GUI principal, o método **actionPerformed()** da

JTable: Aprendendo a utilizar tabelas em Java

classe **AdicionarCursoListener** é chamado no clique do botão *Adicionar Curso* e cria uma janela secundária na linha 25, que é uma instância da classe **DialogoAdicionarCurso**, exposta na **Listagem 14**. Na linha 26, tal janela é exibida mantendo o foco até que seja fechada.

Listagem 13. Código da classe AdicionarCursoListener.

```
01. package pkg;
02.
03. import java.awt.event.ActionEvent;
04. import java.awt.event.ActionListener;
05. import java.sql.Connection;
06. import javax.swing.JFrame;
07. import javax.swing.JTable;
08. import javax.swing.table.DefaultTableModel;
09.
10. public class AdicionarCursoListener implements ActionListener {
11.
12.     private JTable tabela;
13.     private DefaultTableModel modelo;
14.     private Connection con;
15.     private JFrame frame;
16.
17.     public AdicionarCursoListener(JTable tabela, DefaultTableModel modelo,
18.                                     Connection con, JFrame frame) {
18.         this.tabela = tabela;
19.         this.modelo = modelo;
20.         this.con = con;
21.         this.frame = frame;
22.     }
23.
24.     public void actionPerformed(ActionEvent e) {
25.         DialogoAdicionarCurso dialogo = new DialogoAdicionarCurso(frame);
26.         dialogo.setVisible(true);
27.         boolean criarCurso = dialogo.sucesso();
28.         if (criarCurso) {
29.             String nome = dialogo.getNome();
30.             String tipo = dialogo.getTipo();
31.             String duracao = dialogo.getDuracao();
32.             String[] curso = {nome, tipo, duracao};
33.             boolean sucesso = AcessoDatabase.inserirCurso(con, nome, tipo, duracao);
34.             modelo.addRow(curso);
35.             if (sucesso)
36.                 System.out.println("Curso inserido com sucesso.");
37.         }
38.     }
39. }
```

Quando a janela auxiliar for fechada, a execução da aplicação passará para a linha 27, onde o método **sucesso()** indicará se o usuário inseriu um registro (verdadeiro) ou cancelou a inserção (falso). Na linha 28 é testado se o usuário confirmou a inclusão do curso e nas linhas 29 a 31 são acessados os valores digitados na janela auxiliar que, neste momento, se encontra invisível. Para completar, na linha 33, o método **inserirCurso()** é usado para criar o novo registro na tabela Cursos do banco de dados, e finalmente, na linha 34, o curso criado é inserido no **JTable**.

Criação da janela auxiliar para inserção de cursos

A janela auxiliar **DialogoAdicionarCurso** é responsável por receber os dados de um novo curso digitado pelo usuário, sendo

criada durante a execução do *listener* **AdicionarCursoListener**. Por ser um **JDialog**, ela se sobrepõe à GUI principal e não apresenta botão de minimização, além de ganhar o foco da GUI.

Para permitir a digitação dos campos de um novo curso, a classe **DialogoAdicionarCurso** apresenta os campos de texto para receber o nome do curso, o tipo deste e sua duração. A interface gráfica dada pela classe **DialogoAdicionarCurso** tem a organização apresentada na **Figura 2**.

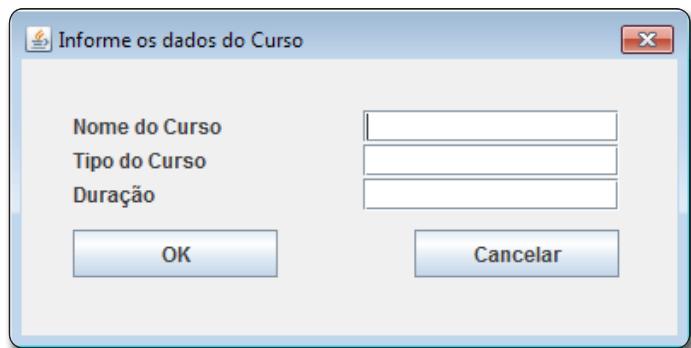


Figura 2. Interface gráfica da janela auxiliar para adição de cursos

A fim de indicar que o usuário preencheu todos os campos e clicou em **OK**, **DialogoAdicionarCurso** contém uma variável de estado chamada **sucesso**, que se torna verdadeira nessa situação e falsa quando o usuário cancela a operação. O código desta classe é apresentado na **Listagem 14**.

Nesta listagem, nas linhas 24 a 60 é apresentado o construtor, que tem a função de instanciar os componentes utilizados, adicioná-los à janela, definir as suas posições e tamanhos e vincular *listeners* aos botões de ação **OK** e **Cancelar**. Uma vez que o usuário tenha clicado em **OK**, os valores digitados podem ser obtidos pelos métodos **getNome()**, **getTipo()** e **getDuracao()**, codificados nas linhas 72 a 82. Além disso, o método **sucesso()**, implementado entre as linhas 84 e 86, retorna verdadeiro se o usuário clicou em **OK** e falso se a operação foi cancelada.

Listener para o botão Remover Curso

Outro *listener* da GUI principal é chamado quando o usuário clica no botão *Remover Curso Selecionado*, ação que invoca o método **actionPerformed()** da classe **RemoverCursoListener**. Tal método tem a função de remover do banco de dados o registro selecionado no **JTable**. A classe **RemoverCursoListener** é exibida na **Listagem 15**.

Detalhando o processamento desse *listener*, na linha 23 é obtido o índice da linha selecionada pelo usuário. Em seguida, este índice é usado na linha 24 para acessar os dados do respectivo registro. O nome do registro é obtido na linha 25 e passa a não ser mais exibido no **JTable** na linha 26, uma vez que todos os dados relacionados ao curso são removido do modelo da tabela. Finalmente, na linha 27, ao informar o nome do curso como parâmetro para o método **apagarCurso()**, os dados deste são removidos do banco de dados.

Listagem 14. Código da classe DialogoAdicionarCurso.

```
01. package pkg;
02.
03. import java.awt.event.ActionEvent;
04. import java.awt.event.ActionListener;
05. import javax.swing.JButton;
06. import javax.swing.JDialog;
07. import javax.swing.JFrame;
08. import javax.swing.JLabel;
09. import javax.swing.JTextField;
10.
11. public class DialogoAdicionarCurso extends JDialog {
12.
13.     private static final long serialVersionUID = 1L;
14.     private JLabel nomeLabel;
15.     private JLabel tipoLabel;
16.     private JLabel duracaoLabel;
17.     private JTextField nomeField;
18.     private JTextField tipoField;
19.     private JTextField duracaoField;
20.     private JButton ok;
21.     private JButton cancelar;
22.     private boolean sucesso = false;
23.
24.     public DialogoAdicionarCurso(JFrame owner) {
25.
26.         super(owner, true);
27.         setSize(400, 200);
28.         setTitle("Informe os dados do Curso");
29.
30.         nomeLabel = new JLabel("Nome do Curso");
31.         tipoLabel = new JLabel("Tipo do Curso");
32.         duracaoLabel = new JLabel("duração");
33.         nomeField = new JTextField("");
34.         tipoField = new JTextField("");
35.         duracaoField = new JTextField("");
36.         ok = new JButton("OK");
37.         cancelar = new JButton("Cancelar");
38.
39.         getContentPane().setLayout(null);
40.         getContentPane().add(nomeLabel);
41.         getContentPane().add(tipoLabel);
42.         getContentPane().add(duracaoLabel);
43.         getContentPane().add(nomeField);
44.         getContentPane().add(tipoField);
45.         getContentPane().add(duracaoField);
46.         getContentPane().add(ok);
47.         getContentPane().add(cancelar);
48.
49.         nomeLabel.setBounds(30, 30, 130, 19);
50.         tipoLabel.setBounds(30, 50, 130, 19);
51.         duracaoLabel.setBounds(30, 70, 130, 19);
52.         nomeField.setBounds(200, 30, 150, 19);
53.         tipoField.setBounds(200, 50, 150, 19);
54.         duracaoField.setBounds(200, 70, 150, 19);
55.         ok.setBounds(30, 100, 120, 28);
56.         cancelar.setBounds(230, 100, 120, 28);
57.
58.         ok.addActionListener(new ActionListener() {
59.             public void actionPerformed(ActionEvent e) {
60.                 sucesso = true;
61.                 setVisible(false);
62.             }
63.         });
64.         cancelar.addActionListener(new ActionListener() {
65.             public void actionPerformed(ActionEvent e) {
66.                 sucesso = false;
67.                 setVisible(false);
68.             }
69.         });
70.     }
71.
72.     public String getNome() {
73.         return nomeField.getText();
74.     }
75.
76.     public String getTipo() {
77.         return tipoField.getText();
78.     }
79.
80.     public String getDuracao() {
81.         return duracaoField.getText();
82.     }
83.
84.     public boolean sucesso(){
85.         return sucesso;
86.     }
87. }
```

Listagem 15. Código da classe RemoverCursoListener.

```
01. package pkg;
02.
03. import java.awt.event.ActionEvent;
04. import java.awt.event.ActionListener;
05. import java.sql.Connection;
06. import java.util.Vector;
07. import javax.swing.JTable;
08. import javax.swing.table.DefaultTableModel;
09.
10. public class RemoverCursoListener implements ActionListener {
11.
12.     private JTable tabela;
13.     private DefaultTableModel modelo;
14.     private Connection con;
15.
16.     public RemoverCursoListener(JTable tabela, DefaultTableModel modelo,
17.                                 Connection con) {
17.         this.tabela = tabela;
18.         this.modelo = modelo;
19.         this.con = con;
20.     }
21.
22.     public void actionPerformed(ActionEvent e) {
23.         int linha = tabela.getSelectedRow();
24.         Vector dados = (Vector) modelo.getDataVector().get(linha);
25.         String nome = (String) dados.get(0);
26.         modelo.removeRow(tabela.getSelectedRow());
27.         boolean sucesso = AcessoDatabase.apagarCurso(con, nome);
28.         if (sucesso)
29.             System.out.println("Curso Removido com Sucesso.");
30.     }
31. }
```

JTable: Aprendendo a utilizar tabelas em Java

Listener para edição do JTable

Outro *listener* chamado pela GUI principal está relacionado às alterações feitas pelo usuário diretamente no **JTable**, o que gera a necessidade de atualizar o registro correspondente na tabela do banco de dados. Para realizar essa persistência, implementamos a classe **EventoTabelaListener**, que é chamada automaticamente pelo **JTable** quando um elemento é alterado e depois pressionado a tecla *Enter*.

Na **Listagem 16** é apresentado o código da classe **EventoTabelaListener**. Nesta classe, na linha 16 é iniciada a codificação do construtor, que recebe dados importantes da aplicação, como a tabela, o modelo desta e a conexão com o banco. Tais dados são acessados no método **valueChanged()**, que inicia na linha 22, e é responsável pelo código de ação do *listener*. Em seu processamento, na linha 23 é obtido o índice da linha selecionada e na linha 25, o respectivo registro é acessado pelo modelo da tabela. Os campos do registro corrente são recuperados nas linhas 26 a 28 e as alterações nesse registro são aplicadas no banco de dados na linha 29, usando o método **alterarCurso()**, que recebe como parâmetro o nome do curso a ser modificado seguido dos novos valores para tipo do curso e duração.

Listagem 16. Código da classe **EventoTabelaListener**.

```
01. package pkg;
02.
03. import java.sql.Connection;
04. import java.util.Vector;
05. import javax.swing.JTable;
06. import javax.swing.event.ListSelectionEvent;
07. import javax.swing.event.ListSelectionListener;
08. import javax.swing.table.DefaultTableModel;
09.
10. public class EventoTabelaListener implements ListSelectionListener {
11.
12.     private JTable tabela;
13.     private DefaultTableModel modelo;
14.     private Connection con;
15.
16.     public EventoTabelaListener(JTable tabela, DefaultTableModel modelo,
17.         Connection con) {
18.         this.tabela = tabela;
19.         this.modelo = modelo;
20.         this.con = con;
21.     }
22.     public void valueChanged(ListSelectionEvent e) {
23.         int linha = tabela.getSelectedRow();
24.         if (linha > 0) {
25.             Vector curso = (Vector) modelo.getDataVector().get(linha);
26.             String nome = (String) curso.get(0);
27.             String tipo = (String) curso.get(1);
28.             String duracao = (String) curso.get(2);
29.             boolean sucesso = AcessoDatabase.atualizarCurso(con, nome, tipo, duracao);
30.             if (sucesso)
31.                 System.out.println("Curso Atualizado com Sucesso.");
32.         }
33.     }
34. }
```

Listener de fechamento da GUI

O último *listener* da nossa aplicação é chamado no fechamento da GUI, mais comumente, quando o usuário clica no botão fechar da janela. Como em nosso caso, ao clicar para fechar a GUI ainda podem existir alterações nos dados do **JTable** pendentes a serem persistidas no banco de dados, o *listener* **WindowListener** (veja a **Listagem 17**) é usado para atualizar o último registro editado e fechar a aplicação.

Listagem 17. Código da classe **WindowListener**.

```
01. package pkg;
02.
03. import java.awt.event.WindowAdapter;
04. import java.awt.event.WindowEvent;
05. import java.sql.Connection;
06. import java.util.Vector;
07. import javax.swing.JTable;
08. import javax.swing.table.DefaultTableModel;
09.
10. public class WindowListener extends WindowAdapter {
11.
12.     private JTable tabela;
13.     private DefaultTableModel modelo;
14.     private Connection con;
15.
16.     public WindowListener(JTable tabela, DefaultTableModel modelo,
17.         Connection con) {
18.         this.tabela = tabela;
19.         this.modelo = modelo;
20.         this.con = con;
21.     }
22.     public void windowClosing(WindowEvent e) {
23.
24.         super.windowClosing(e);
25.         int linha = tabela.getSelectedRow();
26.         if (linha > 0)
27.             {
28.                 Vector curso = (Vector) modelo.getDataVector().get(linha);
29.                 String nome = (String) curso.get(0);
30.                 String tipo = (String) curso.get(1);
31.                 String duracao = (String) curso.get(2);
32.                 boolean sucesso = AcessoDatabase.atualizarCurso
33.                     (con, nome, tipo, duracao);
34.                 if (sucesso)
35.                     System.out.println("Curso Atualizado com Sucesso.");
36.             }
37.         System.exit(0);
38.     }
39.     public void windowClosed(WindowEvent e) {
40.         super.windowClosed(e);
41.     }
42. }
```

Visto que esse *listener* é exclusivo para o fechamento de janelas, o código dele é um pouco diferente dos demais. Note que na linha 22 é declarado o método **windowClosing()**, chamado antes do fechamento da aplicação. Neste método, na linha 25 é acessado o registro selecionado e nas linhas 29 a 31 os dados deste registro são obtidos, para serem gravados no banco de dados na linha 32. Assim, qualquer alteração sobre o registro selecionado é persistida antes do fechamento da aplicação.

A partir do conhecimento aqui exposto, o leitor pode expandir a aplicação a fim de possibilitar o gerenciamento de tabelas maiores ou ainda exibir e editar dados combinados de diversas tabelas.

Como sugestão, tente também adicionar mais recursos visuais ao **JTable**, pois ele viabiliza o uso de outros componentes GUI em suas células, como *check-boxes*, *combo-boxes* e outros que forem adequados ao objetivo proposto.

Além dos métodos analisados no artigo, esse componente disponibiliza métodos para inserção ou remoção de colunas, customização das cores do *grid* e das células, definição da largura e altura das células, configuração dos modos de seleção de células, linhas e colunas, entre muitos outros, mostrando que ele é um componente GUI completo para exibição e edição de tabelas. Portanto, explore ao máximo os recursos deste. Certamente, ao desenvolver aplicações desktop, você irá utilizá-lo.

Autor



John Soldera

johnsoldera@gmail.com

É bacharel em Ciências da Computação pela UCS (Universidade de Caxias do Sul), mestre em Computação Aplicada pela Unisinos e cursa atualmente doutorado em Ciências da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul). Trabalha com Java há 12 anos e possui a certificação SCJP.



Links:

Javadoc da plataforma Java SE 7.

<http://docs.oracle.com/javase/7/docs/api/>

Endereço para download do Banco de Dados Java DB no site da Oracle.

<http://www.oracle.com/technetwork/java/javadb/overview/index.html>

Endereço para download do Banco de Dados Java DB no site da Apache.

<http://db.apache.org/derby/>

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

Google App Engine: Utilizando Spring MVC

O desenvolvimento de software na nuvem não se restringe apenas aos desenvolvedores que atuam em times de grandes empresas de TI. Hoje podemos encontrar uma variedade de infraestruturas ou plataformas como serviço para que qualquer desenvolvedor possa ingressar nesse mundo, inclusive de forma gratuita, como é o caso do Google App Engine. E o que é que tem a ver esse kit com o GAE? Melhorar ligação.

O Google App Engine, ou GAE, como é popularmente conhecido, é uma plataforma robusta e altamente escalável para hospedagem de aplicações web na nuvem. Além de fornecer esta plataforma, o Google também disponibiliza um excelente kit de desenvolvimento (SDK) para a criação de aplicações específicas para este ambiente. Com esse kit é possível simular localmente os serviços de Memcache, Task Queue (serviço de fila para execução de processos em background), Mail e Datastore (banco de dados não relacional altamente escalável) encontrados no App Engine. Desta forma, o desenvolvedor consegue emular o ambiente do GAE em seu próprio computador, agilizando o desenvolvimento. Então, uma vez que a aplicação esteja pronta em seu computador, basta realizar um simples deploy, tornando-a disponível na nuvem do Google. Este cenário faz do GAE uma plataforma de desenvolvimento muito atrativa, conquistando a cada dia mais adeptos.

Apesar de todas essas vantagens, o Java Runtime Environment (ou JRE) do GAE não possui todas as classes que conhecemos e estamos acostumados. Na documentação do Google App Engine podemos encontrar uma *white list* informando as classes do JRE que podem ser utilizadas (veja a seção [Links](#)).

Com esta restrição, algumas APIs da plataforma Java EE e alguns frameworks que dependem das classes que ficaram de fora do JRE não podem ser utilizados no GAE. Como exemplo de APIs da Java EE que não são suportadas, podemos citar EJB, JMS e JNDI.

No entanto, mesmo com essas restrições, veremos neste artigo que é possível, e de forma simples, criar aplicações web para o App Engine empregando alguns dos frameworks mais populares do mercado, o Spring MVC e o Apache Tiles.

Fique por dentro

Este artigo é útil para desenvolvedores que desejam iniciar no Google App Engine utilizando frameworks já conhecidos e consolidados no mercado, como é o caso do Spring MVC e do Apache Tiles. Assim, veremos ao longo desse estudo como configurar uma aplicação web que será hospedada no GAE. Nesta, criaremos um CRUD e aplicaremos os conceitos do padrão Model-View-Controller com o auxílio dos frameworks citados.

Google App Engine

O Google App Engine é uma plataforma como serviço (Platform as a Service, ou PaaS) para desenvolvimento e hospedagem de aplicações web criada em 2008 pelo Google.

Nessa plataforma, encontramos diversos serviços que podemos fazer uso sem que tenhamos que nos preocupar com configurações ou infraestrutura. Como exemplos de serviços desse tipo podemos citar o Mail, Memcache, OAuth, Task Queue, XMPP, entre outros.

Em sua versão atual, o App Engine suporta as seguintes linguagens de programação: Java, Python, PHP e Go, mas tem planos para suportar mais linguagens no futuro.

Apesar de todos esses benefícios, a plataforma também possui algumas restrições, a saber:

- Os desenvolvedores têm acesso somente leitura ao sistema de arquivos do diretório da aplicação, ou seja, não é possível escrever dados neste;
- Uma requisição é interrompida caso a aplicação leve mais de 60 segundos para processar e retornar uma resposta;
- Nem todas as classes do Java (JRE) estão disponíveis para uso.

No entanto, mesmo com essas restrições, o GAE é uma ótima opção de Cloud. Isso pode ser facilmente notado pelo fato de grandes empresas como a Rovio, MAG Interactive e Blossom.io passarem a utilizar o App Engine como plataforma para suas aplicações web, e até mesmo mobile.

Spring MVC

O Spring MVC é um módulo do Spring Framework que foi desenvolvido para ser utilizado em aplicações web, tendo como

característica fornecer mecanismos para que seja possível aplicar de forma fácil o padrão MVC (Model-View-Controller).

Os desenvolvedores do Spring sentiram a necessidade de desenvolvê-lo porque notaram que os frameworks existentes na época não eram fieis às camadas do MVC, deixando alguns pontos a desejar.

Uma característica positiva e marcante do Spring MVC é a facilidade de integração com os diversos módulos do Spring Framework, e também com outros frameworks de mercado, como é o caso do Velocity, Apache Tiles (que abordaremos no próximo tópico), dentre muitos outros.

Apache Tiles

O Tiles é um framework para criação de templates utilizado em interfaces de usuário em aplicações Java voltadas para a web e que possuem como base o padrão MVC.

Através de pequenos fragmentos de página, os chamados tiles, é possível criar templates reutilizáveis que serão construídos em tempo de execução para compor uma página a ser exibida ao usuário. Essa característica de composição descreve o padrão conhecido como Composite.

O uso do Tiles reduz consideravelmente a duplicidade de código, o tempo de desenvolvimento e a manutenção da aplicação, uma vez que é possível reutilizar cada fragmento de página na criação de templates, inclusive a combinação de templates para compor outros e assim originar páginas mais elaboradas.

Podemos considerar como exemplo muito comum de reutilização de fragmento de página, o seguinte cenário: imagine que sua aplicação possua um sidebar fixo que deve ser exibido em todas as telas, e este possua alguns componentes como logo da empresa, nome da aplicação, nome do usuário logado, botão de logout, entre outros. Para evitar a replicação deste trecho de código em todas as páginas, podemos, com a ajuda do Apache Tiles, criar um fragmento com o conteúdo do sidebar e incorporá-lo a um template. Feito isso, bastaria às páginas do sistema estender esse template, e todas elas passariam a ser compostas pelo sidebar.

Configuração do projeto

Agora que conhecemos um pouco das principais tecnologias que utilizaremos na aplicação exemplo, passaremos para a fase de preparação e configuração do projeto, para então iniciarmos o desenvolvimento.

O primeiro passo para a configuração da aplicação é criar a estrutura de diretórios do projeto, o que é realizado executando o comando apresentado seguir. Essa estrutura de diretórios é definida pelo Maven. Este, além de nos auxiliar com a estrutura de diretórios, também será essencial para o gerenciamento das dependências de nossa aplicação.

```
mvn archetype:generate -DgroupId=br.com.javamagazine -DartifactId=cursos  
-DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false
```

Ao executar o comando supracitado, teremos como resultado um projeto chamado "cursos". Para facilitar o desenvolvimento deste, vamos importá-lo para uma IDE. No caso deste artigo, optamos pelo Eclipse. Assim, execute o seguinte comando na pasta raiz do projeto:

```
mvn eclipse:eclipse -Dwtpversion=2.0
```

Com esses passos concluídos, podemos dar início à declaração das dependências da aplicação, como veremos no tópico a seguir.

Configurando o pom.xml

Um arquivo POM, ou Project Object Model, é usado basicamente para declararmos as dependências do projeto, os plugins utilizados para integração com ferramentas externas, e como iremos empacotar nossa aplicação (JAR ou WAR, por exemplo).

Iniciaremos a configuração de nosso *pom.xml* declarando o plugin do Google App Engine. Essa declaração deve ser inserida entre as tags `<plugins>` e `</plugins>` (veja a **Listagem 1**). Com esse plugin é possível rodar localmente nossa aplicação simulando o ambiente do GAE, ou até mesmo realizar o deploy da aplicação na nuvem.

Listagem 1. Configuração do plugin do Google App Engine no *pom.xml*.

```
<build>  
<plugins>  
<plugin>  
  <groupId>com.google.appengine</groupId>  
  <artifactId>appengine-maven-plugin</artifactId>  
  <version>${appengine.target.version}</version>  
  <configuration>  
    <jvmFlags>  
      <jvmFlag>-Xdebug</jvmFlag>  
      <jvmFlag>-agentlib:jdwp=transport=dt_socket,address=1044,server=y,suspend=n</jvmFlag>  
    </jvmFlags>  
    <disableUpdateCheck>true</disableUpdateCheck>  
  </configuration>  
</plugin>  
  
<!-- Outras declarações omitidas -->  
  
</plugins>  
</build>
```

Nota

Para ver a lista completa de comandos (goals) do plugin do Google App Engine, após a configuração deste no Maven, execute no terminal (ou prompt de comando): `mvn help:describe -Dplugin=appengine`.

Voltando ao *pom.xml*, iremos declarar entre as tags `<dependencies>` três dependências do projeto, conforme apresenta a **Listagem 2**. Essa declaração fará com que todas as bibliotecas necessárias para o correto funcionamento da aplicação sejam automaticamente baixadas em nosso projeto.

Google App Engine: Utilizando Spring MVC

A primeira delas é a dependência referente ao SDK do GAE, usado na compilação e para a execução da aplicação. Logo em seguida, temos a dependência do módulo MVC do Spring (Spring Web MVC). E por fim, declaramos a dependência do Apache Tiles. A versão completa do arquivo *pom.xml* pode ser baixada junto ao código completo do exemplo abordado neste artigo na página desta edição.

Listagem 2. Declaração das dependências no *pom.xml*.

```
<dependencies>

    <dependency>
        <groupId>com.google.appengine</groupId>
        <artifactId>appengine-api-1.0-sdk</artifactId>
        <version>${appengine.target.version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${org.springframework-version}</version>
    </dependency>

    <dependency>
        <groupId>org.apache.tiles</groupId>
        <artifactId>tiles-jsp</artifactId>
        <version>${org.apache.tiles-version}</version>
    </dependency>

    <!-- Outras declarações omitidas -->

</dependencies>
```

Configurando o *web.xml*

Para que possamos utilizar o Spring MVC em nosso projeto, é necessário realizar algumas configurações em nossa aplicação. Estas configurações devem ser feitas no Deployment Descriptor, o arquivo *web.xml*. O código deste arquivo da nossa aplicação pode ser visto na **Listagem 3**. Note que nas linhas 7 a 17 temos a declaração do **DispatcherServlet**, servlet que tem papel fundamental no Spring MVC, uma vez que é utilizado como Front Controller pelo framework.

O front controller é o componente responsável por manipular todas as requisições recebidas e encaminhá-las para o controller correspondente. O controller, por sua vez, executa a operação solicitada e atualiza o modelo. Assim que o controller finaliza sua operação, a execução volta para o front controller, que sabe exatamente qual view deve ser renderizada, de acordo com o retorno obtido do método executado no controller. Estes passos descrevem o ciclo de vida de processamento de uma requisição recebida pelo Spring MVC (veja a **Figura 1**).

Por padrão, durante a inicialização do **DispatcherServlet** no startup da aplicação, o Spring MVC varre o diretório WEB-INF à procura de um arquivo XML chamado *<nomeDoServlet>-servlet.xml*. Este contém algumas configurações essenciais para o funcionamento do Spring MVC, como a localização das classes de controle e a declaração do View Resolver. O arquivo XML em

questão deve ser criado com o nome dado ao servlet **DispatcherServlet** no *web.xml*, seguido pelo texto *"-servlet.xml"*.

Como em nosso exemplo declaramos **dispatcher** como o nome do servlet (veja a linha 19 da **Listagem 3**), o Spring MVC procurará pelo arquivo /WEB-INF/dispatcher-servlet.xml em nossa aplicação.

Por último, ainda no *web.xml* (**Listagem 3**), declaramos nas linhas 19 a 28 o filtro **HiddenHttpMethodFilter**. Com este filtro o Spring MVC consegue “simular” operações não suportadas por formulários HTML, como os métodos HTTP HEAD, DELETE e PUT.

Listagem 3. Conteúdo do arquivo *web.xml*.

```
01. <?xml version="1.0" encoding="utf-8"?>
02. <web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
03.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04.   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
05.     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
06.
07. <servlet>
08.   <servlet-name>dispatcher</servlet-name>
09.   <servlet-class>
10.     org.springframework.web.servlet.DispatcherServlet
11.   </servlet-class>
12.   <load-on-startup>1</load-on-startup>
13. </servlet>
14. <servlet-mapping>
15.   <servlet-name>dispatcher</servlet-name>
16.   <url-pattern>/</url-pattern>
17. </servlet-mapping>
18.
19. <filter>
20.   <filter-name>HttpMethodFilter</filter-name>
21.   <filter-class>
22.     org.springframework.web.filter.HiddenHttpMethodFilter
23.   </filter-class>
24.   </filter>
25. <filter-mapping>
26.   <filter-name>HttpMethodFilter</filter-name>
27.   <url-pattern>/*</url-pattern>
28. </filter-mapping>
29.
30. </web-app>
```

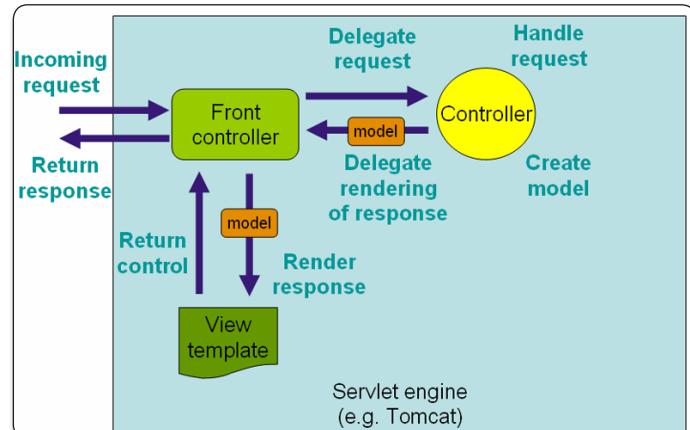


Figura 1. Ciclo de vida do processamento de requisições no Spring MVC

Configurando o dispatcher-servlet.xml

No arquivo *dispatcher-servlet.xml* estão concentradas as definições referentes ao Spring MVC. É neste arquivo que informamos as configurações necessárias para a renderização de páginas (view resolver), onde declaramos os conversores (converters), os interceptadores (interceptors), entre outros.

Analisando o código do *dispatcher-servlet.xml* na **Listagem 4**, observamos na linha 14 como informar ao Spring MVC para varrer o pacote `br.com.javamagazine.cursos.controller` e registrar todos os controllers (classes anotadas com `@Controller`) encontrados. Uma vez que o Spring instancia esses objetos e os registra em seu contêiner, esses objetos se tornam beans gerenciados pelo Spring. Com isso, podemos realizar a injeção de dependência (DI).

Para que possamos utilizar as anotações do Spring MVC em nossa aplicação, mais especificamente em nossos controllers, faremos uso da tag `<mvc:annotation-driven />` (vide linha 16). Ela habilita as anotações `@InitBinder`, `@RequestMapping`, `@PathVariable`, entre outras que veremos mais adiante.

Ao receber uma requisição qualquer, o Spring MVC tentará encontrar um controller para atendê-la. Porém, no caso de arquivos estáticos (imagens, CSS e JavaScripts), não há necessidade de implementarmos um controller para realizar esta operação, visto que o Spring MVC consegue tratar essas requisições utilizando o Servlet padrão do container. Deste modo, informaremos ao Spring MVC que os recursos estáticos serão atendidos pelo Servlet padrão. Essa configuração é feita através da tag `<mvc:default-servlet-handler />` e pode ser visualizada na linha 18.

Para mapearmos a URL dos arquivos estáticos, utilizamos a tag `<mvc:resources mapping="/resources/**" location="/resources/" />`, conforme a linha 35. Isto indica que qualquer requisição a `/resources/**` será mapeada para o diretório `/resources` do projeto. Essa abordagem facilita o mapeamento dos recursos. Por exemplo, imagine que nossa aplicação armazenasse os arquivos estáticos em `/webapp/public-resource/static/`. Assim, poderíamos simplesmente indicar para o Spring MVC que qualquer requisição para o endereço `/resources` seria mapeada para `/webapp/public-resource/static/`.

Agora vamos configurar o Apache Tiles, para que ele possa ser utilizado juntamente com o Spring MVC. A configuração da integração entre os frameworks pode ser vista nas linhas 22 a 33. O Tiles será a tecnologia empregada para trabalhar com a camada de visão da nossa aplicação, e para tal, informamos ao View Resolver onde estarão localizados os arquivos de definição do Tiles (linhas 29 a 30).

Nota

De forma geral, o View Resolver realiza o mapeamento das strings retornadas pelos métodos do controller para as views ou páginas que serão exibidas para o usuário.

Por último, na linha 37, através da tag `<mvc:view-controller path="/" view-name="index" />`, indicamos que as requisições efetuadas para o endereço root `/`, o Spring MVC irá devolver

como resposta a página `index.jsp`, localizada no diretório `/WEB-INF/views/`, de modo que não seja necessária a criação de um controller para realizar esta operação.

Listagem 4. Conteúdo do arquivo dispatcher-servlet.xml.

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <beans xmlns="http://www.springframework.org/schema/beans"
03.   xmlns:context="http://www.springframework.org/schema/context"
04.   xmlns:mvc="http://www.springframework.org/schema/mvc"
05.   xmlns:p="http://www.springframework.org/schema/p"
06.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
07.   xsi:schemaLocation="http://www.springframework.org/schema/beans
08.   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
09.   http://www.springframework.org/schema/context
10.   http://www.springframework.org/schema/context/spring-context-3.0.xsd
11.   http://www.springframework.org/schema/mvc
12.   http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
13.
14. <context:component-scan base-package="br.com.javamagazine.cursos.
15.   controller"/>
16. <mvc:annotation-driven />
17. <mvc:default-servlet-handler />
18. <context:annotation-config />
19.
20. <bean id="tilesviewResolver"
21.   class="org.springframework.web.servlet.view.tiles3.TilesViewResolver" />
22.
23. <bean id="tilesConfigurer"
24.   class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
25.   <property name="definitions">
26.     <list>
27.       <value>/WEB-INF/tiles/tiles.xml</value>
28.       <value>/WEB-INF/view/**/views.xml</value>
29.     </list>
30.   </property>
31. </bean>
32.
33. <mvc:resources mapping="/resources/**" location="/resources/" />
34.
35. <mvc:view-controller path="/" view-name="index" />
36.
37. <beans>
```

Configurando o appengine-web.xml

Numa aplicação Java para o Google App Engine, além do `web.xml`, devemos incluir em nosso projeto o arquivo `appengine-web.xml` no diretório `/WEB-INF`. Este arquivo contém o id e a versão da aplicação, bem como outras configurações específicas que determinam o comportamento da aplicação no ambiente de nuvem do Google.

A configuração mínima necessária para que uma aplicação seja implantada no GAE é apresentada na **Listagem 5**, onde declaramos apenas o id, usando a tag `<application>`, a versão, com a tag `<version>`, e com a tag `<threadsafe>`, indicamos que a aplicação pode receber múltiplas requisições simultaneamente.

O `appengine-web.xml` de nossa aplicação será definido conforme a **Listagem 6**. Note que foi utilizado para especificar o id e a versão da aplicação as variáveis `${project.artifactId}` e `${project.version}` , ambas declaradas no `pom.xml` utilizando as tags `<artifactId>` e `<version>`.

Google App Engine: Utilizando Spring MVC

Nosso arquivo ainda contém as declarações das propriedades de log nas linhas 8 a 10, e a configuração de arquivos estáticos como JavaScripts, CSS, e imagens, nas linhas 12 a 14.

Deste modo concluímos a configuração inicial da nossa aplicação. Podemos, enfim, passar para a codificação e colocar as mãos no Spring MVC.

Listagem 5. Configuração mínima do appengine-web.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <application>app-id</application>
  <version>versao</version>
  <threadsafe>true</threadsafe>
</appengine-web-app>
```

Listagem 6. Conteúdo do appengine-web.xml de nossa aplicação.

```
01. <?xml version="1.0" encoding="utf-8"?>
02. <appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
03.
04.   <application>${project.artifactId}</application>
05.   <version>${project.version}</version>
06.   <threadsafe>true</threadsafe>
07.
08.   <system-properties>
09.     <property name="java.util.logging.config.file"
10.       value="WEBINF/logging.properties"/>
11.   </system-properties>
12.
13.   <static-files>
14.     <include path="/resources/**" />
15.   </static-files>
16.
17. </appengine-web-app>
```

Hello, Spring MVC

Agora que já temos em nossa aplicação todas as dependências necessárias, o *web.xml*, o *appengine-web.xml* e o *dispatcher-servlet.xml* configurados, realizaremos um teste básico, pondo à prova tudo o que fizemos até aqui.

Para que nosso exemplo consiga atender as requisições, criaremos um controller chamado **HelloSpringMVC** no pacote **br.com.javamagazine.cursos.controller**, com o código apresentado na **Listagem 7**.

Feito isso, execute a aplicação conforme explicado anteriormente. Em seguida, acesse o endereço *http://localhost:8080/hello* em seu navegador. Como resposta, se tudo foi configurado corretamente, veremos a frase “Hello, Spring MVC!” no navegador.

Entendendo o exemplo HelloSpringMVC

Para entendermos o funcionamento do Spring MVC, iniciaremos pela análise do arquivo de configuração *dispatcher-servlet.xml* (veja a **Listagem 4**). Neste arquivo, indicamos para o Spring (linha 14) a localização de nossos controllers (ou beans) a serem gerenciados. Desta maneira, todas as classes anotadas com **@Controller** no pacote **br.com.javamagazine.cursos.controller** serão instanciados no contexto do Spring.

Já na classe **HelloSpringMVC**, conforme expõe a **Listagem 7**, declaramos a anotação **@RequestMapping** e informamos */hello* como parâmetro (vide linha 9). Com isso, o Spring MVC, através do front controller, mapeará as requisições feitas ao endereço */hello* para esse controller.

Listagem 7. Código do controller HelloSpringMVC.

```
01. package br.com.javamagazine.cursos.controller;
02.
03. import org.springframework.stereotype.Controller;
04. import org.springframework.web.bind.annotation.RequestMapping;
05. import org.springframework.web.bind.annotation.RequestMethod;
06. import org.springframework.web.bind.annotation.ResponseBody;
07.
08. @Controller
09. @RequestMapping("/hello")
10. public class HelloSpringMVC {
11.
12.   @RequestMapping(method = RequestMethod.GET)
13.   public @ResponseBody String sayHello() {
14.     return "Hello, Spring MVC!";
15.   }
16. }
```

Note que na linha 12 usamos novamente a anotação **@RequestMapping**, porém, agora, declaramos apenas o método de mapeamento (**method = RequestMethod.GET**). Neste ponto informamos que qualquer requisição feita via HTTP GET para */hello* será tratada pelo método **sayHello()**. Esta anotação suporta os seguintes métodos: DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT e TRACE.

Já na linha 13 temos a anotação **@ResponseBody**. Esta informa ao Spring MVC que a resposta obtida deve ser exibida no corpo da página como conteúdo. Se esta declaração for omitida, o framework, através do View Resolver, tentará encontrar uma página com o nome “Hello Spring MVC” e um erro como o apresentado a seguir será exibido:

Could not resolve view with name 'Hello Spring MVC' in servlet with name 'dispatcher'

Nota

Como não mapeamos outros métodos em nosso controller, uma requisição para o endereço *http://localhost:8080/hello* que não utilize o método GET não será tratada por ele.

Nota

Por padrão, quando digitamos um endereço no browser e pressionamos Enter, o navegador realiza uma requisição utilizando o método GET.

A aplicação exemplo

Para que possamos colocar em prática os conceitos vistos até o momento, desenvolveremos a partir de agora uma aplicação exemplo.

A aplicação proposta será implementada para uma escola de treinamentos fictícia e irá contemplar uma das características mais básicas e essenciais à maioria das soluções web, o CRUD (acrônimo

para *Create, Read, Update e Delete*). De forma objetiva, nosso sistema fará o gerenciamento de cursos, e para isso, permitirá o cadastro, edição e exclusão de cursos, além de possibilitar a listagem dos cursos já cadastrados.

Para facilitar o entendimento de nosso exemplo e não fugir do escopo do artigo, a solução implementada tem como objetivo ser bastante simples, e devido a isso, teremos apenas uma entidade (**Curso**), que será desenvolvida com base na representação exposta na **Figura 2**.

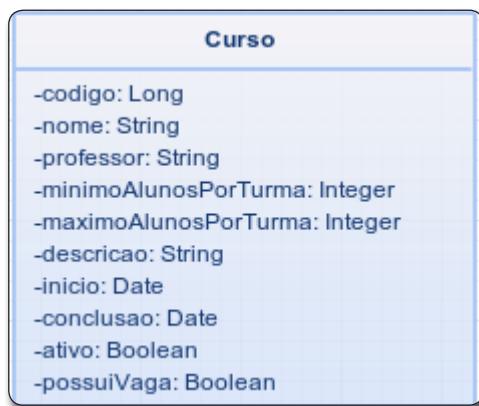


Figura 2. Diagrama de classe representando a entidade **Curso**

Início da implementação

Neste ponto já temos nosso projeto configurado e já sabemos o básico sobre o Spring MVC. Chegou a hora de colocar as mãos na massa e codificar nossas classes para atender os requisitos supracitados.

Sendo assim, realizaremos a implementação passando por cada uma das camadas do MVC, começando pela camada de modelo.

Implementando o Model

O modelo de nossa aplicação será representado pela classe **Curso**, de acordo com a especificação exibida no diagrama da **Figura 2**. Desta forma, criaremos uma classe no pacote **br.com.javamagazine.cursos.entity** chamada **Curso**, onde definiremos todos os campos utilizados no cadastro. O código de nossa classe deve ser igual ao apresentado na **Listagem 8**.

Dado o modelo apresentado, vamos implementar uma classe que possuirá alguns métodos estáticos para simular as operações em um repositório de dados.

O código da classe **Repository** pode ser visto na **Listagem 9**, e como podemos observar, definimos métodos para busca, inclusão/ atualização e remoção de cursos, sendo que todas essas operações serão realizadas em memória, simulando o acesso ao **Datastore** ou ao **Cloud SQL** do Google App Engine.

CURSOS ONLINE

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



CONHEÇA OS CURSOS MAIS RECENTES:

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

Para mais informações :

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

(21) 3382-5038

Google App Engine: Utilizando Spring MVC

Listagem 8. Código da classe Curso.

```
package br.com.javamagazine.cursos.entity;

import java.util.Date;

public class Curso implements java.io.Serializable {

    private static final long serialVersionUID = 1L;

    private Long codigo;
    private String nome = "";
    private String professor = "";
    private Integer minimoAlunosPorTurma;
    private Integer maximoAlunosPorTurma;
    private String descricao = "";
    private Date inicio;
    private Date conclusao;
    private Boolean ativo;
    private Boolean possuiVaga;

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

Implementando o Controller

Com a camada de modelo e o repositório prontos, podemos passar para a etapa principal de nossa aplicação, a codificação do controller, que ficará encarregado de receber as requisições das telas de listagem, inclusão, visualização e remoção da entidade **Curso**. O código do controller **CursoController** pode ser visto na [Listagem 10](#). Note que ele segue o mesmo princípio do exemplo anterior, **HelloSpringMVC**, acrescido de algumas anotações.

Assim, usando a anotação **@RequestMapping**, e com o auxílio dos parâmetros **value** e **method**, mapeamos um método de nosso controller para uma determinada URL. Seguindo nosso exemplo, podemos ver nas linhas 44 a 48 que informamos o valor `/create` para o parâmetro **value** e o enum **RequestMethod.GET** para o **method**. Deste modo informamos ao Spring MVC que as requisições do tipo GET para o endereço `/curso/create` devem ser repassadas para o método `create()` do controller **CursoController**.

Essa característica de mapear os métodos da aplicação atendendo os princípios do padrão arquitetural REST torna o nosso projeto uma solução RESTful. De acordo com as diretrizes do padrão REST, as ações de nosso sistema devem ser implementadas conforme os métodos HTTP (vide [Tabela 1](#)).

Listagem 9. Código da classe Repository.

```
01. package br.com.javamagazine.cursos.repository;
02.
03. import java.text.ParseException;
04. import java.text.SimpleDateFormat;
05. import java.util.Date;
06. import java.util.HashSet;
07. import java.util.Set;
08.
09. import br.com.javamagazine.cursos.entity.Curso;
10.
11. public class Repository {
12.
13.     public static final Set<Curso> cursos = new HashSet<Curso>();
14.     private static long codigo = 0;
15.
16.     private static Date toDate(String strDate) {
17.         SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
18.         Date date = null;
19.         try {
20.             date = formatter.parse(strDate);
21.         } catch (ParseException e) {
22.             e.printStackTrace();
23.         }
24.         return date;
25.     }
26.
27.     static {
28.         cursos.add(new Curso(1L, "Java EE", "Uguinho", 10, 30,
29.             "Conteúdo: EJB, JMS, JAX-RS, JAX-WS",
30.             toDate("03/03/2014"), toDate("28/03/2014"), true, true));
31.
32.         cursos.add(new Curso(2L, "Spring", "Zezinho", 10, 20,
33.             "Conteúdo: Introdução Spring IoC, Spring MVC, Spring Security",
34.             toDate("20/01/2014"), toDate("21/02/2014"), true, true));
35.
36.         cursos.add(new Curso(3L, "Google App Engine", "Luizinho", 20, 40,
37.             "Conteúdo: Cloud Computing, Ambiente GAE, Serviços & APIs",
38.             toDate("03/06/2014"), toDate("20/06/2014"), false, true));
39.
40.         cursos.add(new Curso(4L, "Groovy", "Moe", 20, 25,
41.             "Introdução à Linhagem Groovy",
42.             toDate("07/07/2014"), toDate("28/08/2014"), true, true));
43.
44.         cursos.add(new Curso(5L, "JRuby", "Larry", 10, 20,
45.             "Do básico ao Avançado com JRuby",
46.             toDate("06/01/2014"), toDate("30/01/2014"), false, false));
47.
48.         cursos.add(new Curso(6L, "Scala", "Curly", 20, 40,
49.             "Curso avançada de Scala", toDate("14/04/2014"),
50.             toDate("09/05/2014"), true, true));
51.
52.         codigo = 6L;
53.     }
54.
55.     public static Curso findCursoByCodigo(Long curso) {
56.         for (Curso c : cursos) {
57.             if (c.getCodigo().equals(curso))
58.                 return c;
59.         }
60.         return null;
61.     }
62.
63.     public static void saveCurso(Curso curso) {
64.         if (curso.getCodigo() == null) {
65.             codigo++;
66.             curso.setCodigo(codigo);
67.         }
68.         cursos.remove(curso);
69.         cursos.add(curso);
70.     }
71.
72.     public static void removeCurso(Curso curso) {
73.         cursos.remove(curso);
74.     }
75.
76. }
```

Listagem 10. Código da classe CursoController.

```

01. package br.com.javamagazine.cursos.controller;
02.
03. import java.text.SimpleDateFormat;
04. import java.util.Date;
05. import java.util.Set;
06.
07. import org.springframework.beans.propertyeditors.CustomDateEditor;
08. import org.springframework.stereotype.Controller;
09. import org.springframework.ui.ModelMap;
10. import org.springframework.validation.BindingResult;
11. import org.springframework.web.bind.WebDataBinder;
12. import org.springframework.web.bind.annotation.InitBinder;
13. import org.springframework.web.bind.annotation.PathVariable;
14. import org.springframework.web.bind.annotation.RequestMapping;
15. import org.springframework.web.bind.annotation.RequestMethod;
16. import org.springframework.web.bind.annotation.ResponseBody;
17.
18. import br.com.javamagazine.cursos.entity.Curso;
19. import br.com.javamagazine.cursos.repository.Repository;
20.
21. @Controller
22. @RequestMapping(value = "/curso")
23. public class CursoController {
24.
25.     @InitBinder
26.     private void dateBinder(WebDataBinder binder) {
27.         SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
28.         CustomDateEditor editor = new CustomDateEditor(dateFormat, true);
29.         binder.registerCustomEditor(Date.class, editor);
30.     }
31.
32.     @RequestMapping(method = RequestMethod.GET)
33.     public String list(ModelMap modelMap) {
34.         modelMap.addAttribute("cursos", Repository.cursos);
35.         return "/curso/list";
36.     }
37.
38.     @RequestMapping(value = "/{codigo}", method = RequestMethod.GET)
39.     public String view(@PathVariable Long codigo, ModelMap modelMap) {
40.         modelMap.addAttribute("curso", Repository.findCursoByCodigo(codigo));
41.         return "/curso/view";
42.     }
43.
44.     @RequestMapping(value = "/create", method = RequestMethod.GET)
45.     public String createForm(ModelMap modelMap) {
46.         modelMap.addAttribute("curso", new Curso());
47.         return "/curso/create";
48.     }
49.
50.     @RequestMapping(method = RequestMethod.POST)
51.     public String create(Curso curso, BindingResult result) {
52.         Repository.saveCurso(curso);
53.         return "redirect:/curso";
54.     }
55.
56.     @RequestMapping(value = "/update/{codigo}", method = RequestMethod.GET)
57.     public String updateForm(@PathVariable("codigo") Long codigo,
58.                             ModelMap modelMap) {
59.         modelMap.addAttribute("curso", Repository.findCursoByCodigo(codigo));
60.         return "/curso/update";
61.     }
62.
63.     @RequestMapping(method = RequestMethod.PUT)
64.     public String update(@Valid Curso curso, BindingResult result) {
65.         if (result.hasErrors())
66.             return "/curso/update";
67.         Repository.saveCurso(curso);
68.         return "redirect:/curso";
69.     }
70.
71.     @RequestMapping(value = "/delete/{codigo}", method = RequestMethod.GET)
72.     public String deleteForm(@PathVariable("codigo") Long codigo,
73.                             ModelMap modelMap) {
74.         modelMap.addAttribute("curso", Repository.findCursoByCodigo(codigo));
75.         return "/curso/delete";
76.     }
77.
78.     @RequestMapping(method = RequestMethod.DELETE)
79.     public String delete(Curso curso) {
80.         Repository.removeCurso(curso);
81.         return "redirect:/curso";
82.     }
83.
84. }
```

Método HTTP	Ação de CRUD
GET	READ
PUT	UPDATE
POST	CREATE
DELETE	DELETE

Tabela 1. Mapeamento de métodos HTTP para as operações de um CRUD

Implementando a View

Neste ponto já temos nosso controller com as ações codificadas. Agora, precisamos preparar nossas views (nossas páginas) para que os usuários, através da interface, possam realizar as chamadas ao controller.

Desta forma, vamos criar os fragmentos de página para compor um template, utilizando os recursos do Apache Tiles.

Os fragmentos definem as partes da tela que podem ser reutilizadas em todo o sistema. São eles: navbar (**Listagem 11**), menu (**Listagem 12**) e footer (**Listagem 13**). Esses arquivos JSP possuem

apenas declarações de tags HTML, e isoladamente não têm muita utilidade, porém serão usados para compor novas páginas.

Para armazenar esses arquivos, crie um diretório chamado *communs* em /WEB-INF/view/.

Listagem 11. Conteúdo do arquivo navbar.jsp.

```

<div class="navbar navbar-fixed-top">
    <div class="navbar-inner">
        <div class="container-fluid">
            <a class="brand" href="#">Easy Java Magazine - Cursos</a>
        </div>
    </div>
</div>
```

Com os fragmentos criados, podemos definir o layout padrão da aplicação. Para tal, criaremos um diretório em /WEB-INF/view/ chamado *layout*, e nesse diretório teremos um arquivo chamado *default.jsp*, conforme o código da **Listagem 14**.

Google App Engine: Utilizando Spring MVC

Listagem 12. Conteúdo do arquivo menu.jsp.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<div class="span3" id="sidebar">
<ul class="nav nav-list bs-docs-sidenav nav-collapse collapse">
<li class="nav-header">
<a href="#">Administra&ccedil;&atilde;o</a>
</li>
<li class="menutem">
<a href=<c:url value="/" />>Dashboard</a>
</li>
<li class="nav-header">
<a href="#"><i></i>Cursos</a>
</li>
<li class="menutem">
<a href=<c:url value="/curso/create" />>Cadastrar</a>
</li>
<li class="menutem">
<a href=<c:url value="/curso" />>Listar</a>
</li>
</ul>
</div>
```

Listagem 13. Conteúdo do arquivo footer.jsp.

```
<footer>
<p style="text-align:center;">Easy Java Magazine - Cursos</p>
</footer>
```

Listagem 14. Conteúdo do arquivo default.jsp.

```
01. <%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
02. <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
03. <%@ taglib prefix="tilesx" uri="http://tiles.apache.org/tags-tiles-extras" %>
04. <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
05. <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
06. <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
07. <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
08. <%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
09.
10. <!DOCTYPE html>
11. <html xmlns="http://www.w3.org/1999/xhtml">
12. <head>
13. <title><tiles:insertAttribute name="title" defaultValue="" /></title>
14. <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
15.
16. <link href=<c:url value="/resources/vendors/bootstrap/css/bootstrap.min.css" />
   rel="stylesheet" media="screen">
17. <link href=<c:url value="/resources/vendors/bootstrap/css/bootstrap-responsive.
   min.css" /> rel="stylesheet" media="screen">
18. <link href=<c:url value="/resources/css/default.css" /> rel="stylesheet"
   media="screen">
19.
20. <script src=<c:url value="/resources/vendors/js/jquery-2.0.3.min.js" />>
   </script>
21. <script src=<c:url value="/resources/vendors/bootstrap/js/bootstrap.min.js" />>
   </script>
22. <script src=<c:url value="/resources/js/menu.js" />></script>
23. <script src=<c:url value="/resources/vendors/js/Chart.min.js" />></script>
24. <!--[if lt IE 9]>
25. <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
26. <![endif]-->
27. </head>
28. <body>
29.
30. <tiles:insertAttribute name="navbar" defaultValue="" />
31.
32. <div class="container-fluid">
33. <div class="row-fluid">
34.
35. <tiles:insertAttribute name="menu" defaultValue="" />
36.
37. <div class="span9" id="content">
38. <div class="row-fluid">
39. <div class="span9">
40. <div class="block">
41. <div class="navbar navbar-inner block-header">
42. <div class="muted pull-left">
43.
44. <tiles:insertAttribute name="navigation" defaultValue="" />
45.
46. </div>
47. </div>
48. <div class="block-content collapse in">
49.
50. <tiles:insertAttribute name="body" defaultValue="" />
51.
52. </div>
53. </div>
54. </div>
55. </div>
56. </div>
57. </div>
58. <hr>
59.
60. <tiles:insertAttribute name="footer" defaultValue="" />
61.
62. </div>
63. </body>
64. </html>
```

Note que nas linhas 13, 30, 35, 44, 50 e 60 do arquivo *default.jsp* encontramos a declaração da tag `<tiles:insertAttribute />`. Com essa tag, informamos ao Tiles que iremos inserir conteúdo (que pode ser um simples texto ou até mesmo outro arquivo JSP) especificamente nesses pontos da página, de tal forma que podemos usar esse mesmo HTML definido em *default.jsp* como um padrão, e inserir conteúdo dinamicamente, mantendo assim o mesmo layout para todo o sistema.

Seguindo esse princípio, declararmos a tag `<tiles:insertAttribute name="title" defaultValue="" />` na linha 13.

Essa tag nos permite informar um título para a página. Caso nenhum valor seja passado para o atributo `"name"`, o valor declarado em `defaultValue` é atribuído. Já na linha 33, declararmos a tag `<tiles:insertAttribute name="navbar" defaultValue="" />` onde iremos inserir o header da aplicação. Na linha 38 inserirmos o menu e na linha 47 criamos um atributo chamado `navigation` para exibir no painel central da página o nome do item selecionado no menu.

Por sua vez, na linha 53 declararmos o principal atributo para o nosso layout (o **body**). É nesse ponto da página que vamos inserir a maior parte do conteúdo, como os formulários de cadastro e edição. Por último, na linha 63, definimos o footer (nossa rodapé).

Com os JSPs finalizados, podemos informar ao Apache Tiles, através de um arquivo XML, como iremos compor os templates de nossa aplicação. Para o nosso exemplo, vamos criar apenas um template, definido pelo arquivo *tiles.xml*, e que deve ser encontrado em */WEB-INF/tiles/*. Como pode ser visto na **Listagem 15**, criamos uma **definition**. Como o próprio nome diz, uma **definition** define uma página e todas as suas partes, especificando um nome para a *definição* e um modelo a ser usado. Em nosso caso, a nomeamos como *default* e informamos o arquivo *default.jsp* a ser usado como template.

Listagem 15. Conteúdo do arquivo tiles.xml.

```
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
"http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>

<definition name="default" template="/WEB-INF/view/layout/default.jsp">
<put-attribute name="title" value="Default Layout"/>
<put-attribute name="navbar" value="/WEB-INF/view/commons/navbar.jsp"/>
<put-attribute name="menu" value="/WEB-INF/view/commons/menu.jsp"/>
<put-attribute name="body" value="" />
<put-attribute name="navigation" value="" />
<put-attribute name="footer" value="/WEB-INF/view/commons/footer.jsp" />
</definition>

</tiles-definitions>
```

Perceba que compomos nosso template chamado **default** com os fragmentos criados anteriormente, utilizando para isso a tag **<put-attribute />**. Note que para cada tag **<tiles:insertAttribute />** declarada no arquivo *default.jsp*, temos um **<put-attribute />** indicando o seu conteúdo. Este elemento possui dois atributos, chamados **body** e **navigation**, que serão informados de acordo com a página selecionada no menu da aplicação. Assim, se clicarmos no item **Cadastrar**, por exemplo, uma página com o formulário de cadastro de cursos será apresentado no **body**.

Com o template concluído, vamos então codificar as páginas do CRUD de nossa aplicação. Para melhor organização destas, criaremos um diretório chamado *curso* em */WEB-INF/view/*. Este diretório armazenará as páginas: *create.jsp*, *delete.jsp*, *list.jsp*, *update.jsp* e *view.jsp*.

Ainda no diretório *curso*, teremos um arquivo XML chamado *view.xml* que informará para o Tiles como as páginas do sistema serão compostas. Em outras palavras, quais os valores que serão passados para os atributos **title**, **body** e **navigation** do template **default** criado. Na **Listagem 16** podemos ver na íntegra o código do *view.xml*.

Neste arquivo, damos para cada bloco de código da tag **<definition />**, um nome, via atributo **name**. Este valor corresponde a um *outcome* disparado no controller. Em outras palavras, se um determinado método do controller retornar a String “minha_pagina”, o Spring MVC irá procurar por uma definição (**definition**) chamada “minha_pagina”.

Como exemplo, veja o método *list()* na **Listagem 10**. Este método tem como retorno a String */curso/list*.

Listagem 16. Conteúdo do arquivo view.xml.

```
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
"http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>

<definition name="/curso/create" extends="default">
<put-attribute name="title" value="Easy Java Magazine Cursos" />
<put-attribute name="navigation" value="Curso > Cadastrar" />
<put-attribute name="body" value="/WEB-INF/view/curso/create.jsp" />
</definition>

<definition name="/curso/list" extends="default">
<put-attribute name="title" value="Easy Java Magazine Cursos" />
<put-attribute name="navigation" value="Curso > Listar" />
<put-attribute name="body" value="/WEB-INF/view/curso/list.jsp" />
</definition>

<definition name="/curso/view" extends="default">
<put-attribute name="title" value="Easy Java Magazine Cursos" />
<put-attribute name="navigation" value="Curso > Visualizar" />
<put-attribute name="body" value="/WEB-INF/view/curso/view.jsp" />
</definition>

<definition name="/curso/update" extends="default">
<put-attribute name="title" value="Easy Java Magazine Cursos" />
<put-attribute name="navigation" value="Curso > Atualizar" />
<put-attribute name="body" value="/WEB-INF/view/curso/update.jsp" />
</definition>

<definition name="/curso/delete" extends="default">
<put-attribute name="title" value="Easy Java Magazine Cursos" />
<put-attribute name="navigation" value="Curso > Excluir" />
<put-attribute name="body" value="/WEB-INF/view/curso/delete.jsp" />
</definition>

</tiles-definitions>
```

Dado este retorno, o *view resolver* irá buscar no *view.xml* a definição correspondente, ou seja, uma **definition** nomeada como / *curso/list*. Desta forma, uma página seguindo o layout *default*, com o conteúdo “Easy Java Magazine Cursos” para o atributo **title**, “Curso > Listar” para o atributo **navigation** e o arquivo *list.jsp* como valor para o atributo **body**, será exibida. O mesmo conceito se aplica para as outras definições declaradas neste XML.

Agora que declaramos todos os arquivos XML necessários, vamos construir as páginas de nosso CRUD.

Iniciaremos a implementação pelo formulário de edição de Curso (vide **Listagem 17**). Neste código, encontramos na linha 7 a declaração da tag **<form>** contendo a ação que será executada ao submeter o formulário (atributo **action="/curso"**), o método HTTP que deve ser executado (atributo **method="PUT"**) e o objeto que será usado para receber os valores do formulário (atributo **modelAttribute="curso"**).

Essa declaração faz com que o formulário realize uma chamada HTTP PUT para */curso*. Porém, como mencionado anteriormente, os métodos HTTP suportados pelos formulários HTML são apenas GET e POST. Assim, para que seja possível realizarmos o método PUT, o Spring MVC irá submeter o formulário usando o método POST, e armazenará o nome do método (no caso, PUT) num campo oculto. Com isso o Spring MVC consegue ler e executar o método de acordo com o valor contido no campo oculto.

Google App Engine: Utilizando Spring MVC

A tela do sistema referente à edição de um curso pode ser vista na **Figura 3**.

Para que o artigo não fique muito extenso, o código das outras páginas foi omitido. Você pode baixar o código do projeto na íntegra na página desta edição da revista.

A **Figura 4** apresenta a estrutura de diretórios da aplicação.

The screenshot shows a web-based application interface. On the left, there's a sidebar with 'ADMINISTRAÇÃO' and four menu items: 'Dashboard', 'CURSOS', 'Cadastrar', and 'Listar'. The main area has a title 'Curso > Atualizar'. It contains several input fields: 'Nome:' with 'Java EE', 'Professor:' with 'Uguinho', 'Minimo alunos por turma:' with '10', 'Maximo alunos por turma:' with '30', 'Descricao:' with 'Conteúdo: EJB, JMS, JAX-RS, JAX-WS', 'Inicio:' with '03/03/2014', 'Conclusao:' with '28/03/2014', and two checkboxes: 'Ativo:' checked and 'Possui vaga:' checked. At the bottom are two buttons: 'Voltar' and 'Salvar'.

Figura 3. Tela de edição de curso

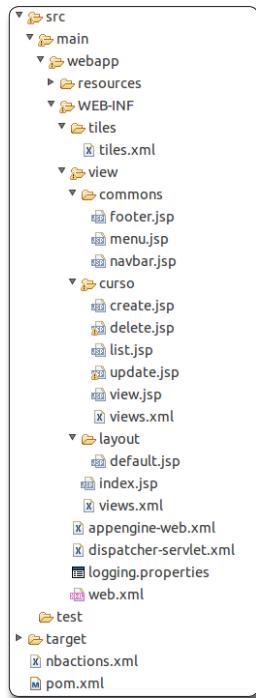


Figura 4. Estrutura de diretórios do projeto

Listagem 17. Código da página update.jsp.

```
01. <%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
02. <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
03. <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
04. <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
05. <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
06.
07. <form:form action="/curso" method="PUT" modelAttribute="curso">
08. <div>
09. <label for="nome">Nome:</label>
10. <form:input cssStyle="width:250px" maxlength="30" path="nome" size="45" />
11. </div>
12. <div>
13. <label for="professor">Professor:</label>
14. <form:input cssStyle="width:250px" path="professor" size="45" />
15. </div>
16. <div>
17. <label for="minimoAlunosPorTurma">Minimo alunos por turma:</label>
18. <form:input cssStyle="width:250px" path="minimoAlunosPorTurma"
   size="30" />
19. </div>
20. <div>
21. <label for="maximoAlunosPorTurma">Maximo alunos por turma:</label>
22. <form:input cssStyle="width:250px" path="maximoAlunosPorTurma"
   size="30" />
23. </div>
24. <div>
25. <label for="descricao">Descricao:</label>
26. <form:input cssStyle="width:250px" path="descricao" size="30" />
27. </div>
28. <div>
29. <label for="inicio">Inicio:</label>
30. <form:input cssStyle="width:250px" path="inicio" size="30" />
31. </div>
32. <div>
33. <label for="conclusao">Conclusao:</label>
34. <form:input cssStyle="width:250px" path="conclusao" size="30"
   disabled="disabled"/>
35. </div>
36. <div>
37. <label for="ativo">Ativo:</label>
38. <form:checkbox path="ativo" />
39. </div>
40. <div>
41. <label for="possuiVaga">Possui vaga:</label>
42. <form:checkbox path="possuiVaga" />
43. </div>
44. <div class="submit">
45. <a href=<c:url value="/curso" /> id="voltar" class="btn btn-
   default">Voltar</a>
46. <input id="salvar" type="submit" value="Salvar" class="btn btn-primary" />
47. </div>
48. <form:hidden path="codigo"/>
49. </form:form>
```

Trabalhando com JSON

O JavaScript Object Notation, ou simplesmente JSON, é um formato padrão criado para troca de mensagens entre sistemas. Por ser leve e possuir notações de fácil leitura e interpretação, tem ganhado muito espaço em cenários de troca de mensagens, e em muitos casos, até substitui o XML.

Assim, neste tópico do artigo, preparamos nosso projeto para usarmos JSON com Spring MVC. A partir disso, nossa aplicação conseguirá receber e responder requisições utilizando esse formato.

Neste exemplo, vamos incrementar nosso projeto para incluir uma funcionalidade que utilizará um gráfico para exibir o número máximo de alunos suportados por cada curso.

Para tanto, adicionaremos ao projeto um plugin escrito em JavaScript responsável pela plotagem dos gráficos. O plugin utilizado será o Chart.js, e a integração entre nossa aplicação e o Chart.js será realizada via JSON.

Para que seja possível retornarmos JSON como resposta, faremos uso do Jackson, uma biblioteca Java que viabiliza a conversão de objetos Java em JSON e vice-versa.

Dito isso, adicione a biblioteca Jackson em nossa aplicação a incluindo como dependência no *pom.xml*, como mostra a **Listagem 18**.

Listagem 18. Declaração da dependência do Jackson no *pom.xml*.

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.9.9</version>
</dependency>
```

Agora que já temos a dependência da biblioteca declarada, vamos incluir em nosso controller o seguinte método:

```
@RequestMapping(value = "/json", method = RequestMethod.GET)
public @ResponseBody Set<Curso> getAllCursosJSON() {
    return Repository.cursos;
}
```

Com a anotação `@ResponseBody` declarada no tipo de retorno no método `getAllCursosJSON()`, o Spring MVC retornará os dados num formato que seja conhecido pelo cliente. Assim, se a solicitação do cliente tiver um header para aceitar JSON (`accept=application/json`), o Spring MVC irá utilizar o Jackson para serializar o valor de retorno para JSON. Implementando esse código, o retorno `Set<Curso>` é serializado e obtemos como resposta algo semelhante à saída exibida a seguir, ao acessarmos o endereço `http://localhost:8080/cursso/json`:

```
[{"codigo":3,"nome":"Google App Engine","professor":"Luizinho",
"minimoAlunosPorTurma":20,"maximoAlunosPorTurma":40,"descricao":"Conteúdo:
Cloud Computing, Ambiente GAE, Serviços & APIs","inicio":1401753600000,"conclusao":1
403222400000,"ativo":false,"possuiVaga":true}]
```

Com isso, temos nosso método retornando todos os cursos cadastrados no formato desejado. Podemos então criar uma página *index.jsp* em */WEB-INF/view/* e incluir uma função Ajax via JavaScript para realizar a chamada ao método `getAllCursosJSON()`. Realizada essa implementação, uma vez que a página *index.jsp* seja carregada no navegador, o código Ajax será executado, e assim que a resposta for obtida, a função JavaScript passará os dados para o plugin declarado entre as linhas 12 e 20 plotar o gráfico, conforme mostra a **Listagem 19**.

Na linha 22 podemos ver a chamada AJAX `$.getJSON()` que invoca o nosso método `getAllCursosJSON()`. O resultado disso é apresentado na **Figura 5**.

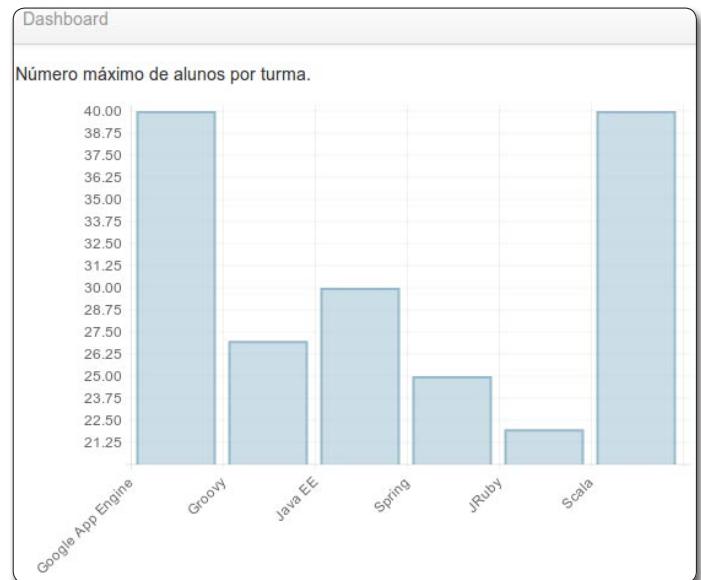


Figura 5. Gráfico criado utilizando JSON

Listagem 19. Código do arquivo *index.jsp*.

```
01. <%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
02. <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
03. <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
04. <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
05. <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
06. <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
07. <%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
08.
09. <p>Número máximo de alunos por turma.</p>
10. <canvas id="myChart" width="600" height="400"></canvas>
11. <script>
12.     $(document).ready(function() {
13.         var chartData = {
14.             labels : [],
15.             datasets : [
16.                 fillColor : "rgba(151,187,205,0.5)",
17.                 strokeColor : "rgba(151,187,205,1)",
18.                 data : []
19.             ]
20.         };
21.
22.         $.getJSON(<spring:url value="curso/json"/>, function(data) {
23.             for (d in data) {
24.                 chartData.labels.push(data[d].nome);
25.                 chartData.datasets[0].data.push(data[d].maximoAlunosPorTurma);
26.             }
27.             var ctx = document.getElementById("myChart").getContext("2d");
28.             new Chart(ctx).Bar(chartData);
29.         });
30.     });
31. </script>
```

Realizando o deploy para o GAE

Com a implementação do projeto concluída, é hora de realizarmos o deploy da aplicação no GAE. Como pré-requisito para isso, deve-se criar uma aplicação no App Engine (veja o endereço na seção **Links**). Como o passo a passo dessa definição é bastante simples, omitiremos essa etapa.

Uma vez que a aplicação tenha sido criada, devemos informar o id (nome dado à aplicação) na tag <artefactId> do arquivo *pom.xml*. Com isso passamos para o comando apresentado a seguir a informação de qual é a aplicação na qual o deploy deve ser realizado.

Ao concluir esta configuração já é possível efetuarmos o deploy com o comando: *mvn appengine:update*. Este comando deve ser executado no terminal (ou prompt de comando, no Windows) no diretório raiz do projeto.

Após a execução desse comando, será solicitado ao usuário que informe as credenciais (e-mail e senha) da conta na qual a aplicação foi criada. Uma vez que a credencial tenha sido fornecida corretamente, o deploy será concluído e a aplicação já poderá ser acessada pelo endereço: <https://<nomeDaSuaApp>.appspot.com>.

Destes modo, concluímos todas as etapas do desenvolvimento do projeto e agora nossa aplicação já está hospedada no Google App Engine.

Durante a implementação, notamos que foram poucas as configurações específicas referentes ao GAE. Assim, constatamos que mesmo num ambiente tão particular como a nuvem do Google, podemos construir aplicações muito parecidas com as desenvolvidas para os servidores de aplicação que estamos acostumados, e ainda, contar com os recursos de infraestrutura do Google sem ter que abrir mão de frameworks populares e consolidados no mercado.

Autor



Marcos Alexandre Vidolin de Lima

marcosvidolin@gmail.com / www.marcosvidolin.wordpress.com

É Bacharel em Ciência da Computação, possui as certificações de programador Java pela Sun Microsystems (SCJP) e Oracle Certified Professional, Java EE 5 Web Component Developer (OCWCD), é entusiasta e amante de novas tecnologias. Atua profissionalmente como desenvolvedor Java EE na CI&T. Como hobby desenvolve e monta robôs para participar de competições. Escreve artigos em revistas especializadas como "Java Magazine" e "Easy Java Magazine" e nas horas vagas mantém seu blog pessoal.



Links:

App Engine Maven Plugin.

<https://code.google.com/p/appengine-maven-plugin/>

Página para criar aplicação no GAE.

<https://appengine.google.com/start>

Lista com as classes do JRE disponíveis no GAE.

<https://developers.google.com/appengine/docs/java/jrewhitelist?hl=pt-BR>

Documentação de referência do Spring MVC.

<http://docs.spring.io/spring/docs/3.2.3.RELEASE/spring-framework-reference/html/mvc.html>

Site do framework Apache Tiles

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

Portal do projeto Jackson

<https://github.com/FasterXML/jackson>

SEU CARRO TEM SEGURO, SUA SAÚDE TEM SEGURO, MAS E O SEU EMPREGO... TÁ SEGURO??



NÃO DEIXE JUSTAMENTE A SUA CARREIRA FICAR EM RISCO!

Manter-se atualizado com todas as novidades do mercado de desenvolvimento é obrigação de todo bom programador. Faça agora mesmo um seguro para a sua carreira. Seja um assinante MVP!

Saia do risco!

QUEM TEM ESTÁ TRANQUILO.

TENHA ACESSO A:



+DE 260 CURSOS ONLINE



09 REVISTAS MENSais



7.850 VÍDEO-AULAS

POR APENAS **69,90**
MENSais



DEVMEDIA

Acesse: www.devmedia.com.br/mvp

Conheça os recursos de Debug do Eclipse – Parte 2

Na primeira parte deste artigo tivemos um contato inicial com *debugging* de programas no Eclipse. Introduzimos o método de identificação, isolamento e correção de erros, mostramos como utilizar as funcionalidades padrão de depuração do projeto JDT, incluído no Eclipse, como alterar para a perspectiva de *debug*, adicionar *breakpoints*, controlar a execução passo a passo e alterar o valor de variáveis, sempre acompanhados de exemplos práticos para cada caso.

Nesta segunda parte do artigo, vamos nos aprofundar um pouco mais sobre este tema, aprendendo a utilizar novas técnicas e funcionalidades de *debugging*, mais complexas e poderosas, permitindo acelerar o processo de identificação e remoção de *bugs* de programas. Vamos falar sobre *breakpoints condicionais*, *watchpoints*, e *threads*, incluindo um exemplo de *debugging* de uma aplicação com problema de *deadlock*.

Breakpoints condicionais

Quando encontramos um erro em nosso programa, vamos querer saber o que o programa está fazendo pouco antes de gerar esse erro. Uma maneira de fazer isso é passar por todas as declarações do código, uma de cada vez, utilizando o *debugger*, até chegar ao ponto específico onde o erro acontece. No entanto, se pensarmos, por exemplo, que podemos estar no meio de um ciclo de mais de mil, ou um milhão de execuções, então esta não será, definitivamente, uma boa ideia. Queremos chegar imediatamente na zona do erro, no momento em que ele está prestes a acontecer. Como vimos no artigo anterior, podemos saber em que parte do código ele se encontra olhando para o *stack trace* no log ou janela de *output*, mas ainda temos que descobrir em que momento ele ocorreu, ou seja, que variáveis o programa estava tratando quando se deu o erro.

No caso de ciclos de execução, para chegar no local pretendido do código e no momento correto, é possível alterar os valores das variáveis de controle, como vimos na primeira parte deste artigo. Mas nem sempre estaremos dentro de um ciclo, ou nem sempre saberemos qual o valor exato de uma iteração em que ocorre o erro. Nestes casos, podemos utilizar outra opção: declarar pontos de interrupção (*breakpoints*) condicionais, que são acionados sempre que o valor de uma expressão mudar.

Fique por dentro

Esse artigo demonstra, de maneira avançada, como utilizar as ferramentas que estão disponíveis no IDE Eclipse para auxiliar desenvolvedores a encontrar erros de código em qualquer programa, sendo, portanto, bastante útil para quem quiser aumentar seus conhecimentos em *debugging*.

Para demonstrar este recurso, foi construído um programa, baseado nos tutoriais de Java da Oracle, que faz com que dois “amigos” se cumprimentem. Vamos tentar simplificar o máximo possível este exemplo para podermos nos focar nos ensinamentos de *debugging* e não no código em si.

Neste programa existe uma classe **Friend**, que cria um objeto que tem apenas o campo **name**, com o nome do amigo. Esta mesma classe possui dois métodos: o **bow()**, para cumprimentar um amigo, e o **bowBack()**, para cumprimentar um amigo de volta. Possui também o método **main()**, que constrói um *array* de amigos e cria uma *thread* para cada par de amigos se cumprimentar. Chamar os métodos em *threads* distintas vai ser importante para aprendermos mais detalhes interessantes sobre o *debugger*. O código do nosso exemplo é demonstrado na [Listagem 1](#).

Agora vamos executar o programa algumas vezes e analisar o seu *output*. Se você observar com atenção, notará que a ordem com que os amigos se cumprimentam nunca é a mesma, o que implica que não se cumprimentam na mesma ordem com a qual são feitas as chamadas dos métodos **bow()** e **bowBack()** em nosso programa. Isso acontece porque estamos executando *threads* distintas, e a JVM se encarrega de dizer qual *thread* será executada em cada momento. Não temos controle sobre isso. A única coisa que sabemos é que cada *thread* terá o seu tempo para ser executada. O exemplo de um possível *output* do programa é demonstrado na [Listagem 2](#).

O primeiro a cumprimentar um amigo foi Alberto, que cumprimentou Roberto e, se repararem, Roberto foi o último a cumprimentar de volta, mesmo tendo sido o primeiro a ter sido cumprimentado. Leo foi o segundo a cumprimentar um amigo, apesar de ter sido o último a ser chamado (pertence ao último par no *array* de amigos), e assim por diante.

Vamos agora forçar um erro comum, inserindo um valor nulo no *array* de amigos. Para isso, vamos escolher uma pessoa qualquer (Maria, por exemplo), e alterar a instância do seu objeto por **null**. Ou seja, onde temos: **new Friend("Maria")** no *array friends*, inserimos **null**.

Listagem 1. Código do exemplo Deadlock.

```
public class Deadlock {
    static class Friend {
        private static int instanceCounter = 0;
        private int instanceNumber = 0;
        private final String name;

        public Friend(String name) {
            this.name = name;
            instanceNumber = ++instanceCounter;
        }

        public String getName() {
            return this.name;
        }

        public synchronized void bow(Friend bower) {
            System.out.format("%d: %s cumprimentou %s!%n",
                instanceNumber, this.name,
                bower.getName());
            bower.bowBack(this);
        }

        public synchronized void bowBack(Friend bower) {
            System.out.format("%d: %s cumprimentou %s de volta!%n",
                instanceNumber, this.name,
                bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend[] friends = {
            new Friend("Alberto"), new Friend("Roberto"),
            new Friend("Chico"), new Friend("Anysio"),
            new Friend("Didi"), new Friend("Dedé"),
            new Friend("João"), new Friend("Maria"),
            new Friend("Eduardo"), new Friend("Mônica"),
            new Friend("Leo"), new Friend("Bia")
        };

        for (int i = 1; i <= friends.length; i += 2) {
            final Friend firstFriend = friends[i - 1];
            final Friend secondFriend = friends[i];

            new Thread(new Runnable() {
                public void run() {
                    firstFriend.bow(secondFriend);
                }
            }).start();
        }
    }
}
```

Listagem 2. Exemplo de output do programa Deadlock.

```
1: Alberto cumprimentou Roberto!
11: Leo cumprimentou Bia!
12: Bia cumprimentou Leo de volta!
9: Eduardo cumprimentou Mônica!
10: Mônica cumprimentou Eduardo de volta!
3: Chico cumprimentou Anysio!
4: Anysio cumprimentou Chico de volta!
5: Didi cumprimentou Dedé!
6: Dedé cumprimentou Didi de volta!
7: João cumprimentou Maria!
8: Maria cumprimentou João de volta!
2: Roberto cumprimentou Alberto de volta!
```

Feito isso, se executarmos de novo nosso programa, veremos o nosso velho conhecido **NullPointerException**, provavelmente um dos erros mais conhecidos em nossa área (veja a **Listagem 3**).

Listagem 3. Exemplo de output com erro causado pela inserção de um objeto nulo no array friends.

```
Exception in thread "Thread-3" java.lang.NullPointerException
at Deadlock$Friend.bow(Deadlock.java:17)
at Deadlock$1.run(Deadlock.java:46)
at java.lang.Thread.run(Unknown Source)
```

```
1: Alberto cumprimentou Roberto!
8: Eduardo cumprimentou Mônica!
9: Mônica cumprimentou Eduardo de volta!
10: Leo cumprimentou Bia!
11: Bia cumprimentou Leo de volta!
5: Didi cumprimentou Dedé!
6: Dedé cumprimentou Didi de volta!
3: Chico cumprimentou Anysio!
4: Anysio cumprimentou Chico de volta!
2: Roberto cumprimentou Alberto de volta!
```

Nota

Vale lembrar que, apesar deste ser um exemplo simples, feito para facilitar a compreensão sobre os ensinamentos da utilização do debugger, ao mesmo tempo contém toda a informação necessária para a resolução de problemas muito mais complexos. Esta nota foi feita porque o que vamos demonstrar aqui é usado, da mesma maneira, na indústria, em grandes empresas, com programas grandes e complicados.

No caso do nosso exemplo, se olharmos para o *output* gerado pelo programa, veremos que só faltou Maria cumprimentar João e João cumprimentar Maria. Mas imagine, mais uma vez, que estamos trabalhando numa empresa que tem milhões de clientes e que nosso *input* vem de um servidor externo com milhões de registros sendo enviados para nosso programa. Neste caso não vamos conseguir olhar para o *output* e entender imediatamente o que aconteceu, mas vamos conseguir fazer o *debug* da mesma maneira.

Dito isso, e olhando para a *stack trace* que temos no *output*, sabemos que o erro foi provocado na linha 17, pois esta foi a última linha do nosso código a ser executada. Se analisarmos o código do programa nesta mesma linha, saberemos que temos um amigo “bower” que é nulo! Vamos então analisar um pouco mais esta linha.

Na linha 17 temos a instrução: `System.out.format("%d: %s cumprimentou %s!%n", instanceNumber, this.name, bower.getName());`.

O `System.out.format` pode receber argumentos nulos e continuar normalmente sua execução sem causar problemas. Logo, ele não é o responsável por gerar a exceção. A instrução `this.name` também não pode gerar essa exceção neste ponto, visto que se o `this` fosse nulo, o método `bow()` não teria sido chamado, pois `bow()` é um método do próprio `this`. Resta-nos então o `bower.getName()`, que é a única variável que, se for nula, vai gerar um **NullPointerException**. Sendo assim, encontramos o causador do nosso problema.

Agora que isolamos a parte do código que gerou a exceção, temos que saber qual objeto está sendo enviado de forma errada.

Conheça os recursos de Debug do Eclipse – Parte 2

Para isso, vamos colocar um *breakpoint* antes de cada *thread* ser chamada, porque desta maneira teremos a informação sobre a criação de cada um dos objetos. Se colocarmos o *breakpoint* na linha que tem a seguinte instrução: `final Friend secondFriend = friends[i];`, vamos conseguir ver o valor da variável `secondFriend` (o nosso “bower”) apenas depois de pressionar a tecla *F6*. Como não sabemos o tamanho do *input* e não queremos ter que fazer isso infinitas vezes, vamos usar novamente o nosso *hack*, inserindo a instrução `i = i` na linha seguinte, e colocando aí o nosso *breakpoint*, como demonstra a **Listagem 4**.

Agora vem o diferencial, onde vamos inserir uma condição no *breakpoint*. Para fazermos isso, na visão *Breakpoint*, clique com o botão direito do mouse sobre o *breakpoint* que acabamos de adicionar e selecione a opção *Breakpoint Properties...*, como mostra a **Figura 1**.

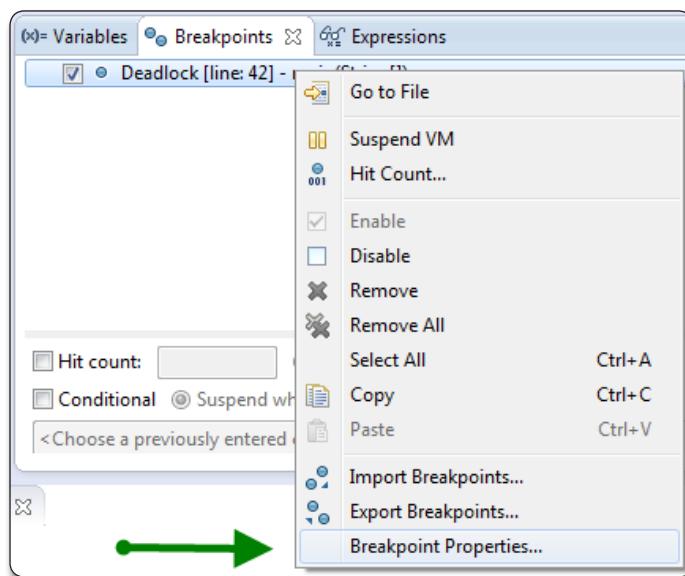


Figura 1. Acessando as propriedades do Breakpoint

Listagem 4. Inserção de um pequeno hack no código.

```
(...)  
final Friend secondFriend = friends[i];  
i = i; // Insira o breakpoint aqui!  
(...)
```

Nas propriedades, vamos selecionar as opções *Conditional* (afirmando que nosso *breakpoint* será condicional, ou seja, que obedece a uma condição) e *Suspended when true* (para dizer que o *debugger* só irá suspender a execução do programa quando a condição dada for verdadeira), e inserir a condição `secondFriend == null` (ou seja, pare a execução quando o segundo amigo não tiver sido instanciado). O resultado deste passo a passo é apresentado na **Figura 2**.

Em seguida, clique em *Ok* e execute a aplicação em modo *debug*. Deste modo, você poderá notar que o *debugger* para apenas quando existe um segundo amigo que é nulo. Exatamente o que

queremos! Podemos confirmar isso olhando para a vista *Variables* (veja a **Figura 3**), ou então passando o mouse sobre as variáveis, que nos dá o mesmo resultado que pressionar as teclas *Ctrl + Shift + I* para inspecionar o elemento.

Nota

Ao pressionar *Ctrl + Space*, você notará que temos o recurso de auto complete para ajudar na escrita das condições.

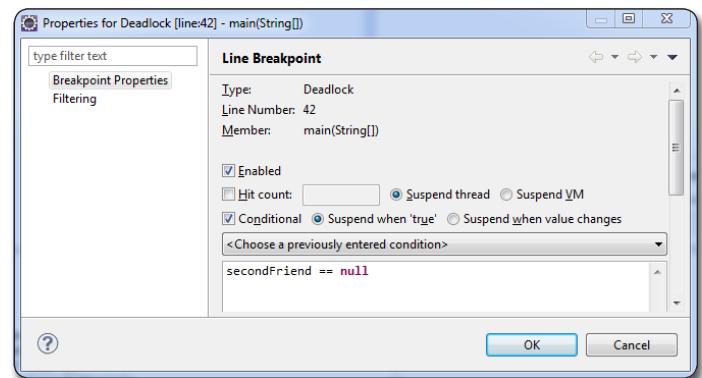


Figura 2. Configurando breakpoint condicional que procura por um objeto não instanciado

Name	Value
args	String[0] (id=24)
friends	Deadlock\$Friend[12] (id=26)
i	7
firstFriend	Deadlock\$Friend (id=28)
name	"João" (id=31)
secondFriend	null

Figura 3. Visão de variáveis no breakpoint condicional

Agora sabemos que existe uma pessoa chamada João que tenta cumprimentar alguém que não existe, e então podemos avisar a quem nos enviou o *input* para que olhe para esta pessoa/cliente/empregado/etc. e corrija o erro para este caso específico.

Este exemplo foi usado para encontrarmos um objeto não instanciado, mas poderíamos fazer o mesmo para inspecionar o estado das variáveis do programa para uma pessoa específica, por exemplo, dizendo para o *debugger* parar na condição em que o primeiro amigo tenha o nome Eduardo. Para isso, basta inserir um *breakpoint* na linha de código logo após a atribuição do valor à variável `firstFriend` e nas propriedades, escrever a seguinte instrução: `firstFriend.getName().equalsIgnoreCase("Eduardo")`, conforme demonstrado na **Figura 4**.

Experimente fazer o mesmo para várias condições. Por exemplo: procure por objetos não instanciados, por propriedades específicas de objetos, por índices específicos de ciclos ou de *arrays*, por nomes de exceções, etc. Esta é uma funcionalidade muito útil e versátil e que pode ajudar bastante.

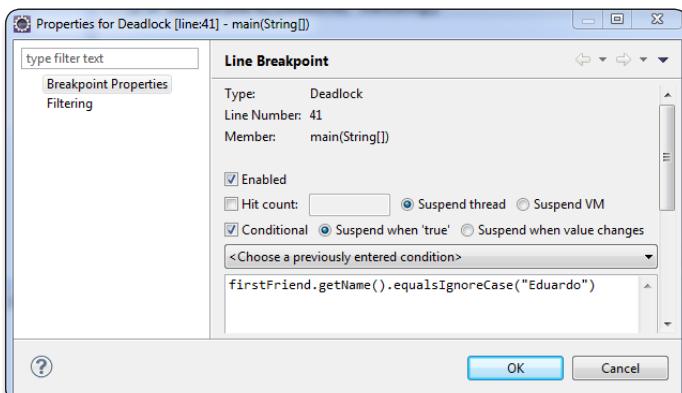


Figura 4. Breakpoint condicional procurando por objeto com uma propriedade específica

Watchpoint

Um *watchpoint* é um *breakpoint* definido em um campo de uma classe. Com a definição de um *watchpoint*, o *debugger* irá parar sempre que o campo for lido ou alterado – isso se não mudarmos as opções definidas por padrão.

Definimos um *watchpoint* exatamente da mesma forma que definimos um *breakpoint*, ou seja, clicando duas vezes na margem esquerda, junto à declaração do campo. Nas propriedades de um *watchpoint*, podemos configurar se a execução deve parar durante o acesso de leitura (*Access*), se deve parar no caso de alterações de valores (*Modification*), ou em ambos os casos.

Como exemplo, vamos inserir um *watchpoint* de alteração de valores no campo `instanceCounter` da classe `Deadlock`. Para isso, dê um duplo clique na barra esquerda da mesma linha, exatamente como fizemos para inserir um *breakpoint*. Quando fizer isso, vai notar que o símbolo que aparece agora é diferente da bolinha azul do *breakpoint*, fazendo realçar que temos um *watchpoint* (veja a Figura 5).

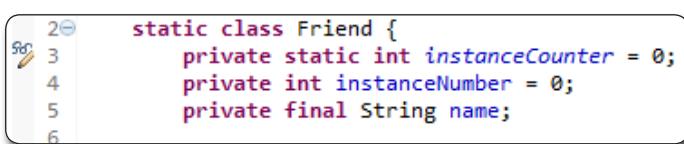


Figura 5. Criação de watchpoint

Em seguida, na visão *Breakpoints*, clique com o botão direito do mouse e escolha a opção *Breakpoint Properties...*. Nesta janela, deixe selecionado apenas a opção de modificação (*Modification*), retirando a seleção da opção de acesso (*Access*), conforme a Figura 6.

Agora, olhe para a barra do lado esquerdo onde inserimos o *watchpoint* e repare que só existe o ícone do lápis, tendo desaparecido a imagem dos óculos. Isso acontece porque agora não estamos mais observando a variável para acessos de leitura, mas apenas de escrita.

Para facilitar a visualização da alteração dos valores dessa variável (e para aprendermos mais uma funcionalidade do Eclipse), acesse a visão *Expressions*, clique na frase *Add new expression* e adicione o nome da variável que queremos observar, ou seja, `instanceCounter` (Figura 7).

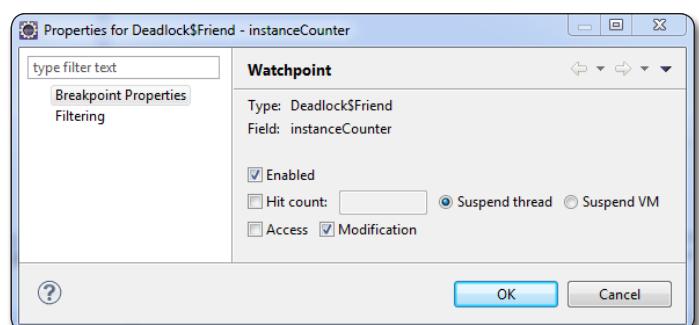


Figura 6. Configurando as propriedades do watchpoint

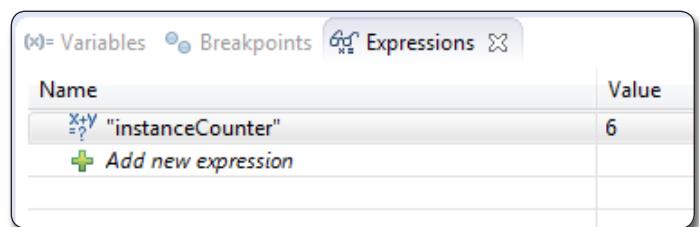


Figura 7. Inspeção do valor da variável "instanceCounter" na visão Expressions

Logo após, execute a aplicação em modo *debug*. Clique em *F8* várias vezes e veja que o *debugger* para em todas as linhas de código onde são feitas alterações em nossa variável que está sendo observada, mesmo que não existam *breakpoints* nestas linhas. No nosso código exemplo, ele para nas linhas 3 e 10, onde são atribuídos valores ao campo `instanceCounter`. Observe também o valor da variável sendo incrementado na visão *Expressions*.

Repararam no número de configurações diferentes que podemos fazer? Na quantidade de pesquisas que podemos implementar? Suponhamos, por exemplo, que queremos ver o estado do programa no momento da criação da sexta instância do objeto `Friend`. Para fazer isso, volte para a janela de propriedades do *watchpoint* e altere o valor do *Hit count* para 7, de acordo com a Figura 8. Sabendo que a primeira atribuição de valor é `instanceCounter = 0`, então o sétimo acesso de escrita será para atribuir o valor da sexta instância.

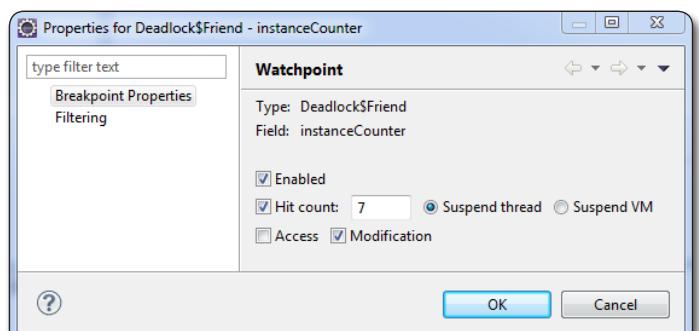


Figura 8. Alterando o Hit count para ver o estado de uma instância específica

Feito isso, execute novamente a aplicação e veja o valor do campo na visão *Expressions*, conforme mostra a Figura 7. Como constatado, ele para exatamente na atribuição do valor 6 à nossa variável (não se esqueça de pressionar *F6* para fazer a atribuição).

Conheça os recursos de Debug do Eclipse – Parte 2

Após esse estudo, experimente fazer o mesmo com outras opções. Seja criativo e imagine outros cenários que poderia resolver com estas propriedades. Essa experiência poderá facilitar a detecção de erros em uma situação real.

O problema dos deadlocks!

Para quem já trabalhou com *threads* e teve que sincronizar chamadas a métodos, sabe que o problema de *deadlock* pode dar uma grande dor de cabeça para resolver. Isto porque, devido à natureza não determinística dos programas *multithread*, não existe, matematicamente falando, maneira de se prever um *deadlock*.

Para testarmos nossas habilidades com as ferramentas de debug fornecidas pelo Eclipse, vamos criar este problema em nosso programa. Para isso, substitua o código do método **main()** atual pelo código indicado na **Listagem 5**.

Listagem 5. Alteração do programa Deadlock – novo método main().

```
public static void main(String[] args) {
    final Friend alberto = new Friend("Alberto");
    final Friend roberto = new Friend("Roberto");

    // Starting Alberto.
    new Thread(new Runnable() {
        public void run() {
            alberto.bow(roberto);
        }
    }).start();

    // Starting Roberto.
    new Thread(new Runnable() {
        public void run() {
            roberto.bow(alberto);
        }
    }).start();
}
```

O que pretendemos com isso é fazer com que Alberto cumprimente Roberto, Roberto cumprimente Alberto e os dois se cumprimentem de volta. Ao executarmos a aplicação normalmente, teremos o seguinte *output*:

1: Alberto cumprimentou Roberto!
2: Roberto cumprimentou Alberto!

Se olharmos para a visão de *Debug*, no canto superior esquerdo, veremos que o programa se encontra em execução, mas não faz mais nada. Porque será? Roberto não cumprimentou Alberto de volta e nem Alberto cumprimentou Roberto de volta. Respondendo a nossa pergunta, estamos numa condição de *deadlock*!

Vamos então usar nossos conhecimentos em *debugger* e depurá-lo para entendermos o que aconteceu.

Um erro bastante comum

Um erro bastante comum em casos como este é o que vamos demonstrar a seguir. Quando um desenvolvedor tem um problema de *deadlock*, ele pode colocar um *breakpoint* no bloco de

código de execução de uma *thread* para tentar descobrir onde está o problema. No entanto, para seu maior pesadelo, executando o programa passo a passo em modo *debug*, este funciona perfeitamente e não entra em *deadlock*. Porque será que isso acontece? Vamos experimentar!

Coloque um *breakpoint* no ponto de execução da primeira *thread*, ou seja, na instrução `alberto.bow(roberto);`; e execute o programa em modo *debug*. Assim que executarmos o programa, teremos o *output*:

2: Roberto cumprimentou Alberto!
1: Alberto cumprimentou Roberto de volta!

Ou seja, a segunda *thread* que manda Roberto cumprimentar Alberto já foi executada! Como veremos mais adiante, esse é um dado muito importante para nossa conclusão.

Continuando nosso *debug*, termine de executar a aplicação, pressionando a tecla *F8*. Ao fazer isso, você terá como resultado:

2: Roberto cumprimentou Alberto!
1: Alberto cumprimentou Roberto de volta!
1: Alberto cumprimentou Roberto!
2: Roberto cumprimentou Alberto de volta!

Ou seja, o programa terminou normalmente sua execução e não tivemos um *deadlock*! É aqui que o desenvolvedor pode pensar que algum milagre aconteceu e vai para casa tranquilo. Contudo, no outro dia, ao executar novamente a aplicação, ele constata que o *deadlock* persiste e pensa: "Mas ontem estava funcionando!".

Então, vamos rever os fatos mais importantes:
1. Estamos executando *threads*;
2. As *threads* são sincronizadas (note a palavra-chave `synchronized` em nosso código);
3. Estamos fazendo *debugging*.

O fato de estarmos fazendo *debugging* em nosso programa, e sendo nós humanos demasiadamente lentos quando comparados com as máquinas, faz com que uma *thread* tenha, em tempo de máquina, "uma infinidade" para realizar uma tarefa simples, apenas nos segundos que demoramos para olhar para a tecla *F8* e pressioná-la. O processador trabalha na escala dos nanosegundos, o que faz com que milisegundos seja muito tempo e segundos sejam "anos", onde um processador pode executar milhares de instruções. Foi exatamente isso que fez com que a JVM decidisse que poderia dar o tempo de execução para a segunda *thread*, que terminou todas as operações que tinha para executar enquanto decidíamos o que fazer (pressionar a tecla *F8*).

Dito isso, olhe novamente para o *output* gerado pela execução normal do programa, onde temos o *deadlock*.

Como podemos observar, a primeira execução foi da *thread* que diz para Alberto cumprimentar Roberto, chamando o método `bow()`. Neste mesmo método, existe uma nova chamada, que pede para que o amigo que foi cumprimentado o cumprimente de volta.

Nesta altura, a JVM decide passar o tempo de execução para a segunda *thread*, que diz para Roberto cumprimentar Alberto (como podemos ver pelo *output*) e, da mesma forma, no método **bow()**, é pedido para Alberto cumprimentar Roberto de volta. É aqui que entra em questão a palavra-chave **synchronized**.

Para entendermos melhor o que aconteceu, remova todos os *breakpoints* que existem no programa, usando a visão *Breakpoints* para isso. Em seguida, execute novamente o programa em modo *debug* sem nenhum *breakpoint*. Quando a aplicação entrar em *deadlock*, poderemos ver algo semelhante à **Figura 9**, na visão de *Debug*.

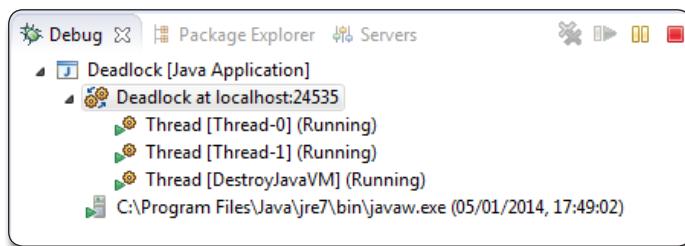


Figura 9. Visão de Debug no exemplo do deadlock

Reparou em algo diferente nesta visão? Lançamos duas *threads*, mas existem três! Isso acontece exatamente porque temos uma aplicação que executa *threads* e o método **main()** já terminou. A terceira *thread*, de nome **DestroyJavaVM**, aparece pela primeira vez imediatamente após o método **main()** terminar, sendo ela responsável por liberar a JVM quando todas as *threads* tiverem terminado sua execução.

Como podemos verificar na **Figura 9**, nossas duas *thread* continuam em execução (estado *running*), e não terminam nunca. Vamos então suspender nossas *threads* manualmente. Para isso, clique com o botão direito do mouse na *thread* que quer suspender e em *Suspend*, ou selecione uma *thread*, clicando em cima dela, e depois clique no ícone amarelo, em forma de pausa (o resultado será o mesmo).

Com as nossas duas *threads* paradas, teremos o resultado apresentado na **Figura 10**.

De acordo com essa figura, a primeira *thread* (*thread-0*) tem o objeto **Friend** com o *id*=20 e está à espera (*waiting*) do objeto **Friend** com *id*=21, que pertence à *thread-1*. Do mesmo modo, se olharmos para a *thread-1*, podemos ver que esta tem o objeto **Friend** com o *id*=21 e está à espera do objeto **Friend** de *id*=20, que pertence à *thread-0*. Ou seja, temos uma *thread* à espera da outra, e vice versa.

Voltando à questão da palavra chave **synchronized**, como os métodos **bow()** e **bowBack()** são sincronizados, quando foi feita a chamada para Alberto cumprimentar Roberto e a JVM chamou a execução para a segunda *thread* antes de se chamar o método **bowBack()**, o objeto **alberto** ficou bloqueado, impedindo que fosse executado qualquer método sincronizado para o objeto **Friend** enquanto ele não terminasse sua execução.

Por sua vez, Roberto cumprimenta Alberto no método **bow()** e tenta fazer a chamada do método **bowBack()** para que Alberto

o cumprimente de volta. Mas Alberto, como vimos, está bloqueado para métodos sincronizados enquanto não terminar sua execução.

A JVM decide então voltar para a *thread-0*, deixando o objeto **roberto** bloqueado no método **bow()**, e pede para que Roberto cumprimente Alberto de volta. Contudo, como acabamos de ver, o objeto **roberto** também está bloqueado, no método **bow()**, para qualquer método sincronizado deste objeto enquanto não terminar sua execução. É esta situação que caracteriza um *deadlock*!

Neste caso simples, como os métodos sincronizados não alteram valores de campos da classe, podemos simplesmente retirar a palavra-chave **synchronized** de um dos métodos, ou mesmo dos dois, para solucionar o problema. Por exemplo, altere a assinatura do método **bowBack()** para:

```
public void bowBack(Friend bower) {
```

Feito isso, execute novamente o programa e terá o seguinte resultado como possível *output*:

- 1: Alberto cumprimentou Roberto!
- 2: Roberto cumprimentou Alberto!
- 1: Alberto cumprimentou Roberto de volta!
- 2: Roberto cumprimentou Alberto de volta!

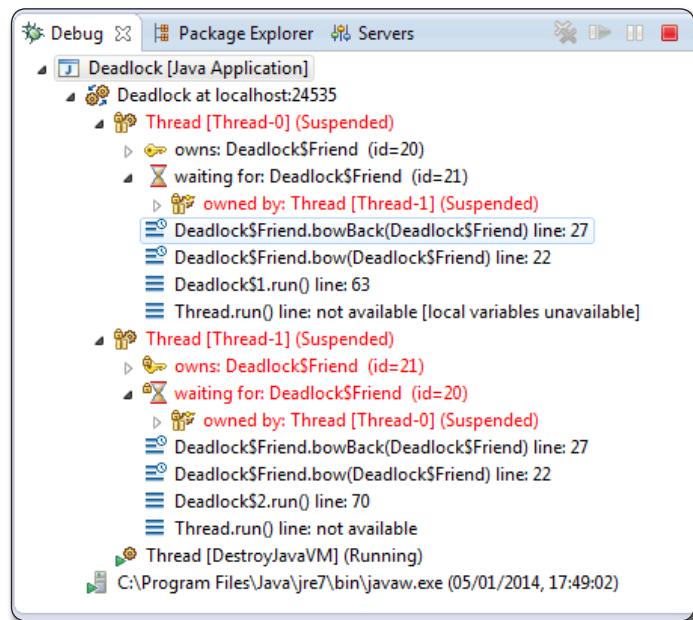


Figura 10. Suspensão manual das threads

Lembre-se que a ordem do *output* pode variar, pois depende das decisões da JVM. Para comprovar isso, experimente executar o programa várias vezes.

O IDE Eclipse oferece um *debugger built-in* da linguagem Java com todas as funcionalidades de depuração padrão, o que inclui a capacidade de realizar a execução passo a passo, definir pontos de

Conheça os recursos de Debug do Eclipse – Parte 2

interrupção (*breakpoints*) para inspecionar variáveis e valores em *runtime*, e a capacidade de suspender e retomar *threads* de execução. Além de permitir a análise do código localmente, o *debugger* do Eclipse também pode ser usado para depurar programas em máquinas remotas.

A plataforma Eclipse é, antes de tudo, um ambiente de desenvolvimento Java, mas as mesmas funcionalidades de *debug* também estão disponíveis para as linguagens C, C++, PHP e outras.

Por fim, vale ressaltar que a capacidade das ferramentas de *debug* oferecidas pelo Eclipse é algo que só poderá ser utilizado com grande eficiência no trabalho do dia a dia se for explorada com curiosidade pelo desenvolvedor. Portanto, experimente, faça testes, use! Deste modo você verá que existem opções excelentes para lhe ajudar a resolver os mais diversos problemas, de forma simples e eficiente!

Autor



José Fernandes A. Júnior

jfajunior@gmail.com

é Mestre em Engenharia Informática pela Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa (FCT-UNL). Trabalha com Java há 10 anos, mas vem trabalhando com diversas linguagens, em diversos ramos e áreas, desde core engines de empresas de telecomunicação, até aplicações móveis para Android e iOS. Trabalhou como formador autorizado da Sun Microsystems nos cursos de Java, Unix, Shell Programming e JCAPS. Possui as certificações de Java Swing, Enterprise Java Beans, Java Composite Application Platform Suite (JCAPS), Certificado de Aptidão Profissional (CAP) e Titanium Certified App Developer (TCAD).



Links:

Projeto do debugger no site do Eclipse.

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

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



DEV MEDIA

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

Ele está trabalhando em seu saque e não na sua nova aplicação.



Mais de 95%* dos times de desenvolvimento e testes de aplicativos relatam tempos de espera e atrasos para acessar os sistemas que necessitam para realizar suas atividades. A Virtualização de Serviços elimina estas dependências gerando ambientes simulados similares à realidade, permitindo o desenvolvimento em paralelo. Isto significa que suas aplicações serão lançadas mais rapidamente, com maior qualidade e menor custo. Game over!

Conheça mais em ca.com/br/GoDevOps

