



Edição 41

DESAFIO: Conheça a API Twitter4J

Aprendendo a utilizar a API de
acesso a dados do Twitter

JSTL – Mais que um “rostinho bonito”

Como adicionar inteligência,
elegantemente, à camada de visão



SIMPLIFIQUE AS THREADS

Processamento paralelo de
modo eficiente e síncrono



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

Conteúdo sobre Boas Práticas

06 – JSP Standard Tag Library (JSTL): Adicionando recursos na camada de visão

[Paulo M. D. Bordin]

Conteúdo sobre Boas Práticas

16 – Programar com Threads em Java

[John Soldera]

Artigo no estilo Solução Completa

27 – Twitter4J API: Conhecendo a API

[Michel Pereira Fernandes]

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 41 • 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 a 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

CURSOS ONLINE

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

Feita para Desenvolvedores de Software e DBAs



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



DEV MEDIA



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



JSP Standard Tag Library (JSTL): Adicionando recursos na camada de visão

Como adicionar inteligência, elegantemente, a uma camada de visão

Desde o final dos anos 70, a separação de sistemas em camadas tornou-se uma espécie de mantra invocado por todos os desenvolvedores, em particular um modelo conhecido como MVC (Modelo, Visão e Controle), onde a ideia principal é ter camadas independentes e desacopladas, sendo o Modelo a representação do mundo real no sistema, a Visão é a interface com os usuários/sistemas e o Controle é a cola entre eles, que manipula os componentes de Modelo para serem exibidos na Visão e define o fluxo da aplicação.

Nos sistemas web, a camada de visão é representada por páginas HTML, estáticas ou providas pelos contêineres web. No caso do Java, temos os JSPs como os principais responsáveis pela geração da saída, que será enviada pelo contêiner web para o usuário. O JSP, do ponto de vista de um desenvolvedor, nada mais é do que um HTML com *tags* especiais para podemos inserir código Java nele e isto permite que, em teoria, um sistema inteiro (acessando banco de dados, web services e sistemas de arquivos) seja construído em um arquivo deste tipo.

No entanto, o cenário de um sistema de uma página só é o pesadelo de arquitetos e responsáveis pela manutenção deste, por ter como resultado arquivos grandes, complexos e sem fazer uso de boas práticas de orientação a objetos, tais como encapsulamento, herança, etc. O cenário ideal, do ponto de vista do MVC, é ter um ou mais *servlets*, acessando as classes de manipulação de dados, aplicando as regras de negócio e mandando as informações necessárias para exibição no JSP. O JSP, por sua vez, recebe essas informações e seu único trabalho é definir onde, como e se as informações serão exibidas.

Usando apenas JSP, suas tags e *expression language*, é possível apresentar as informações fornecidas pelo *servlet*. Contudo, quando uma página requer um pouco

Fique por dentro

O principal objetivo deste artigo é mostrar uma ferramenta que pode ser considerada um meio termo entre a utilização de robustos e badalados frameworks e o desenvolvimento espartano de JSPs fazendo uso apenas de *scriptlets*. Agregando a elegância de frameworks como JSF, Spring e Struts, (utilizando apenas tags e expression language, sem as classes de suporte para a camada de visão) e a simplicidade de projetos web dinâmicos (compostos apenas por *servlets* e JSPs), JSTL tem mais algumas vantagens que veremos no decorrer deste artigo.

de lógica (como percorrer uma lista para montar uma tabela, ou exibir um texto em preto ou azul, por exemplo), um *scriptlet* torna-se necessário. E qual é o problema nisso? Para um desenvolvedor, nenhum, exceto pela legibilidade do código. Agora, e se, digamos, parte da equipe do projeto responsável pela interface com o usuário for composta por web designers, ou pessoas sem tanta familiaridade com Java? Neste caso, é mais fácil ensinar para um web designer o funcionamento de algumas tags e seus atributos, pois este já deve ser familiarizado com tags HTML, a ensinar Java e seus conceitos (sintaxe da linguagem, orientação a objetos, etc.).

Uma ferramenta útil para fazer a interface entre o desenvolvedor e o web designer é o JSP Standard Tag Library, ou JSTL. O JSTL é um conjunto de classes e tags que levam à camada de visão, onde o web designer pode trabalhar as ferramentas utilizadas por desenvolvedores (estruturas de repetição, condicionais, etc.) sem que o web designer precise conhecer Java.

As classes que fazem parte do JSTL seguem uma especificação, a JSR-52, a qual pode ser implementada por qualquer pessoa ou empresa. Uma instituição que implementou e distribuiu a sua versão do JSTL foi a Apache Foundation. Nesta distribuição são encontradas classes que são heranças ou implementações de interfaces suportadas por todos os servidores de aplicação compatíveis

com Java EE 5. Por conta desta condição, para adicionar o suporte à JSTL em uma aplicação, não é necessária a adição de nenhuma biblioteca adicional (exceto, como veremos um pouco mais à frente, quando são utilizadas tags que processam XML, pois estas dependem de bibliotecas terceiras para fazer tal processamento) além da implementação de JSTL distribuída pelo fornecedor, não causando grande impacto no tamanho final de sua aplicação. As classes definidas pela especificação compõem um conjunto de quatro grupos de tags, listados a seguir:

- **Core:** conjunto de tags que fornece as ferramentas mais comuns de linguagens de programação, tais como: estruturas condicionais, de repetição, definição de variáveis e impressão de conteúdo (neste caso para uma saída HTML);
- **Internationalization:** conjunto de tags que auxilia na disponibilização de conteúdo para outras línguas ou outros países. São úteis para formatação de datas, horários e números e para exibição de mensagens de acordo com o *Locale*/idioma do cliente. Na disponibilização de conteúdos em diferentes idiomas, mesmo que um sistema seja concebido inicialmente apenas em um idioma, vale a pena utilizar as tags de mensagens, pois elas substituem textos por chaves que são definidas, assim como seus valores, em um arquivo texto, de maneira centralizada e de fácil manutenção. Também é conhecida por i18n, ou seja, i + 18, das letras que compõem o meio da palavra internationalization, + n;
- **SQL:** é um conjunto de tags que possibilita a criação de conexões com bancos de dados, bem como a posterior manipulação dos destes (instruções de inserção, exclusão, atualização e consulta dos dados). No entanto, a utilidade deste conjunto é questionável, pois a manipulação de dados é algo que definitivamente não é aconselhável que seja feita na camada de visão. Lembrando que uma das ideias do uso de JSTL é que um web designer não precise conhecer Java, no entanto com as tags de SQL, pressupõe-se que ele conheça SQL;
- **XML:** semelhante ao conjunto de tags Core, contudo destinado à manipulação de arquivos XML. Embora útil para manipulação de algum conteúdo remoto (RSS, por exemplo), cai na mesma questão levantada para as tags SQL, a manipulação de dados na camada de visão.

Com estes quatro conjuntos de tags conseguimos cobrir a maioria das necessidades de um desenvolvedor em tarefas repetitivas ou mais complexas e mesmo quando isto não é suficiente, é possível a criação de novas tags, aumentando a padronização do código e a produtividade das equipes de desenvolvimento em níveis equiparáveis aos de frameworks – obviamente que em termos de desenvolvimento da camada de visão. Embora a criação de *custom tags* não seja o foco deste artigo, é essencial citá-la, pois é uma importante ferramenta complementar às tags JSTL.

Fazendo uma análise mais aprofundada no código-fonte dos frameworks web, é possível notar que estes utilizam-se de *custom tags* para construir seus componentes, que posteriormente são empregados no desenvolvimento dos JSPs. Mas aí você pode se perguntar: “Como é que apenas utilizando *custom tags* os

frameworks conseguem gerar componentes gráficos tão bonitos e ricos?”. A resposta para esta pergunta é simples: JSF, Spring, Struts, etc. fazem uso massivo de CSS e JavaScript, diretamente ou através de outras ferramentas como Dojo e jQuery, para criar estas interfaces. A partir disso, podemos dizer que o uso dos frameworks web para desenvolver telas poupa tempo do time de desenvolvimento? Definitivamente, sim! No entanto, não vamos dar os louros da fama apenas aos frameworks Java. Façamos justiça às demais ferramentas envolvidas no processo e com um adendo, você também pode desenvolver componentes à sua maneira, valendo-se das mesmas ferramentas que JSF, Spring e companhia fazem uso.

Usando JSTL em seu projeto web

Antes de começarmos a dissertar sobre as tags JSTL e seus exemplos, é importante adicionar o suporte a esta tecnologia em seu projeto. Para isso, basta seguirmos dois passos: copiar dois jars (`javax.servlet.jsp.jstl-1.2.jar` e `javax.servlet.jsp.jstl-api-1.2.1.jar`) na pasta `lib` e adicionar uma diretiva `taglib` nas páginas JSP.

Um projeto que utiliza JSTL é um projeto web comum criado pelo *wizard* de sua IDE preferida. A **Figura 1** mostra a estrutura de um projeto *Web Dynamic* do Eclipse.

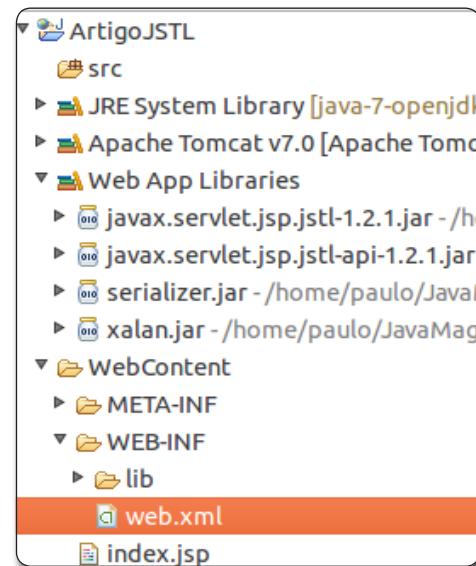


Figura 1. Estrutura de projeto com JSTL

Nota

As bibliotecas `xalan.jar` e `serializer.jar` são necessárias somente quando utilizar o conjunto de tags para XML. Estas são responsáveis pela leitura e parse do conteúdo do XML, tornando-o disponível para uso através do conjunto de tags XML. Caso o projeto não precise de manipulação de XML, elas podem ser descartadas.

Uma tag não HTML em um JSP, seja ela standard (JSTL) ou custom (desenvolvida por terceiros), é utilizada através da diretiva `taglib`. Esta diretiva tem como principais atributos `prefix` e `uri`. O primeiro, como o próprio nome sugere, é um prefixo para ser

usado em todas as tags fornecidas pelo conjunto de taglibs. Já o segundo atributo é uma referência a um valor definido em uma tag dentro de um arquivo TLD (*Tag Library Descriptor*). Pode-se dizer que é um apelido pelo qual o conjunto de taglibs em questão será chamado. A **Listagem 1** mostra um trecho do TLD do conjunto de tags core.

Nota

Um TLD é um arquivo XML que tem a configuração das taglibs e é distribuído dentro do JAR do JSTL. Suas definições incluem: nome/uri do conjunto de taglibs descrito neste TLD, quais classes implementam as tags e a declaração de atributos, seus tipos, se são obrigatórios, se aceitam expressões regulares, etc. Um TLD também pode definir um conjunto de tags feitas pelo desenvolvedor. Neste caso o TLD deve estar dentro da pasta WEB-INF do projeto web.

Listagem 1. Trecho de TLD das tags core.

```
01 <taglib xmlns="http://java.sun.com/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd" version="2.1">
02   <description>JSTL 1.2 core library</description>
03   <display-name>JSTL core</display-name>
04   <tlib-version>1.2</tlib-version>
05   <short-name>c</short-name>
06   <uri>http://java.sun.com/jsp/jstl/core</uri>
07   <validator>
08     <description>Provides core validation features for JSTL tags.</description>
09   <validator-class>org.apache.taglibs.standard.tlv.JstlCoreTLV</validator-class>
10   </validator>
11   <tag>
12     <description>Catches any Throwable that occurs in its body and optionally
           exposes it.</description>
13     <name>catch</name>
14     <tag-class>org.apache.taglibs.standard.tag.common.core.CatchTag</tag-class>
15     <body-content>JSP</body-content>
16     <attribute>
17       <description>
18         Name of the exported scoped variable for the exception thrown from a nested
           action. The type of the scoped variable is the type of the exception thrown.
19       </description>
20       <name>var</name>
21       <required>false</required>
22       <rtpxvalue>false</rtpxvalue>
23     </attribute>
24   </tag>
```

Para cada conjunto de tags JSTL, temos uma ocorrência da diretiva taglib. A **Listagem 2** nos mostra como se parece um JSP com algumas diretivas. Como pode ser observado, nas linhas 2 a 5 estão declaradas as *taglibs* que serão utilizadas neste JSP.

Core tags

Para começar nossos estudos, veremos as tags fornecidas pelo conjunto core. Este conjunto disponibiliza ferramentas básicas para o desenvolvimento, como estruturas de repetição, condicionais e impressão de conteúdo. Logo após o detalhamento destes recursos, será fornecido um exemplo englobando o uso de todas elas.

Listagem 2. Inclusão de JSTL no JSP.

```
01 <%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8"%>
02 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
03 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
04 <%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
05 <%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml"%>
06 <!DOCTYPE html>
07 <html>
08 <body>
09 </body>
10 </html>
```

Out

Responsável pela exibição de um conteúdo na página, conteúdo este que pode ser o valor de uma variável ou um texto fixo. É equivalente à tag JSP <% %>, porém tem um tratamento para caracteres especiais em HTML (esta tag dá a opção de processar ou não conteúdo HTML, ex: <, > e &) e conteúdo inexistente ou nulo é tratado como uma **String** vazia. Os atributos da tag **out** são:

- **value**: contém o valor a ser impresso. Aceita um texto ou expression language, podendo imprimir o valor de uma variável ou atributo de um bean. É um atributo obrigatório;
- **escapeXml**: recebe o valor **true** ou **false**, sendo **true** o valor **default**. Isto significa que se o **value** impresso tem tags HTML, estas serão ignoradas e tratadas como texto. Não é um atributo obrigatório;
- **default**: quando o conteúdo de **value** não existe (por exemplo, uma variável que não foi definida), é possível definir um valor padrão a ser impresso pela tag. Também não é obrigatório.

Set

Com esta tag é possível definir uma variável, ou propriedade de um bean, e seu valor em algum dos quatro escopos web (*page*, *request*, *session* e *application*). Seus atributos são:

- **var**: nome da variável a ser definida;
- **target**: bean que terá a propriedade configurada. Este atributo e **var** não podem ser usados ao mesmo tempo;
- **property**: nome da propriedade do bean a ser configurada. Torna-se obrigatório quando **target** é especificado;
- **scope**: define em qual escopo a variável/propriedade será definida. Quando não informado, assume o escopo padrão, **page**;
- **value**: valor que será atribuído à variável. Se o valor for **null**, a variável será removida do escopo.

If

Tag utilizada para fazer testes em EL. Quando o teste for verdadeiro, o corpo da tag será processado. Uma particularidade desta tag é que ela corresponde ao **if** das linguagens de programação, porém não tem o bloco **else**. Assim como a tag **out**, tem apenas três atributos:

- **test**: recebe a EL a ser testada. O resultado desse teste tem que ser booleano;
- **var**: nome da variável para armazenar o resultado do teste;
- **scope**: define em qual escopo a variável será armazenada.

Nota

Existem quatro escopos web, que resumidamente podem ser considerados repositórios de informação. Neles podem ser armazenados objetos com suas respectivas chaves (assim como em um `HashMap`) e a principal diferença entre eles é o “tempo” pelo qual uma informação existe. A seguir, cada um desses escopos é analisado:

- `page`: é o escopo de menor tempo de vida. Uma variável ou bean definido neste escopo só existe durante o processamento de uma página JSP.
- `request`: é um pouco mais abrangente que `page`. O tempo de vida do escopo `request` vai do momento da requisição feita por um usuário até o envio da resposta para ele. Atributos definidos neste escopo podem ser passados, por exemplo, de um servlet para um JSP, para que este possa utilizar os atributos e suas propriedades para serem exibidas na página resultante do JSP, desde que isso se dê entre a requisição do usuário e o envio de sua resposta.
- `session`: escopo com o tempo de vida um pouco maior que o do tipo `request`, pois, ao contrário deste, que perde suas informações logo após a resposta de uma requisição, atributos definidos em uma `session` podem ser recuperados em requisições futuras. Os valores contidos em uma `session` ficam armazenados no cliente durante o período pelo qual a `session` é válida. Neste ponto é importante frisar “o cliente”, pois existe uma `session` por cliente.
- `application`: é o escopo mais abrangente. Acaba funcionando como uma área de memória global, onde seu conteúdo pode ser acessado por qualquer cliente (diferentemente da `session`) e em qualquer momento (diferentemente do `request`).

Choose/When/Otherwise

Esta é uma estrutura que define três tags a serem utilizadas em conjunto para cumprir o seu objetivo. Primeiro deve-se declarar a tag `choose` e, dentro dela, as tags `when` e `otherwise`. É uma mistura de `if/else` com `switch`, pois pode verificar vários testes e apenas um bloco ser executado. Cada teste é realizado por uma tag `when` e caso nenhuma delas seja verdadeira, o bloco `otherwise` é executado. Apenas a tag `when` tem um atributo, chamado `test`, o qual recebe a EL que será avaliada. Baseado no resultado, o bloco contido no corpo da tag é executado.

Import

É uma tag que permite incluir conteúdo de outras páginas no local em que ela é declarada e, diferentemente da diretiva `include` e da standard action `<jsp:include>`, até mesmo sites externos ao servidor onde se encontra a sua aplicação. Seus principais atributos são:

- `url`: define de onde será importado o conteúdo. Pode ser uma página interna da aplicação, uma página de outra aplicação, no mesmo servidor, ou mesmo outro site em um servidor externo;
- `context`: especifica o caminho raiz no servidor, caso a URL seja relativa, ou seja, não tenha o caminho completo do recurso a ser importado;
- `var`: variável onde o conteúdo será armazenado no formato de uma `String`;
- `scope`: define em qual escopo a variável será armazenada;
- `charEncoding`: define o tipo de codificação de caracteres utilizado para a leitura do conteúdo importado.

URL

Esta tag é utilizada para a criação de links para outras URLs. A diferença entre utilizar esta tag e simplesmente utilizar a tag

HTML `<a>` é que com a primeira, a reescrita da URL é feita anexando ao final do link informações referentes à `session` do usuário. Isto é útil para garantir que haja sessão independentemente da existência de `cookies` ou não. Os atributos de URL são:

- `value`: a URL propriamente dita;
- `context`: especificação do caminho raiz no servidor, caso a URL seja relativa, ou seja, não tenha o caminho completo do recurso a ser importado;
- `var`: variável onde o conteúdo produzido pela tag URL será armazenado no formato de uma `String`;
- `scope`: em qual escopo a variável será armazenada.

Nota

Uma session (sessão) pode armazenar informações no cliente através de cookies, pequenos arquivos texto que contêm informações que podem ser acessadas por aplicações e que são manipulados pelos navegadores. No entanto, a maioria dos navegadores, se não todos, permite desabilitar o suporte a cookies. Quando esse suporte é desativado, deixa de ser possível armazenar informações da sessão no cliente. Para evitar esse tipo de problema, é possível utilizar a reescrita de URL. Esta técnica consiste em produzir um hash code, correspondente aos valores que seriam armazenados na sessão, e passá-lo como um parâmetro da URL acessada. Desta maneira, o cliente sempre passaria como parâmetro de sua nova requisição os valores contidos na sessão.

**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**

E se você precisar passar parâmetros para a URL, para, por exemplo, fornecer um nome para uma consulta? Existem duas formas de se fazer isto. A primeira é simplesmente adicionar no atributo **value** os parâmetros da mesma forma que se faz em uma requisição do tipo GET (ex: *url?param1=valor¶m2=valor*), porém a URL não será codificada para tratar caracteres especiais (espaços, acentos, etc.), por exemplo: *url?nome=Java Magazine&titulo=Titulo do artigo*. A segunda maneira de se passar parâmetros para a URL formada pela tag **url** é adicionando no corpo desta a tag **param**, que tem apenas dois atributos: **name**, para identificar o parâmetro, e **value**, que, além de receber caracteres normais (letras e números), pode trabalhar com caracteres especiais, codificando-os no formato aceito pelas URLs (ex: *url?nome=Java+Magazine&titulo=Titulo+do+artigo*).

ForEach

Com certeza uma das tags mais utilizadas para percorrer os valores de uma lista ou de um *array*. Consiste de uma estrutura de repetição que define a quantidade de repetições, equivalente à estrutura **for** em Java. Seus atributos são:

- **items**: é a fonte dos dados, a lista ou *array* de objetos que será percorrido;
- **var**: é a variável onde o resultado de cada iteração será armazenado. Esta variável só existe dentro do corpo da tag **forEach**;
- **varStatus**: é o nome de uma variável onde serão armazenadas informações de status do *loop*. Dentre essas informações, temos um contador, o passo atual e o início da iteração;
- **begin** e **end**: determinam o trecho a ser percorrido do objeto definido em **items**;
- **step**: atributo que define como será feita a iteração, se de um em um, de dois em dois e assim por diante. Se não for informado, o valor padrão é 1.

Exemplo com Core tags

Para demonstrar os recursos que acabamos de analisar na prática, vamos codificar um exemplo que emprega quase todas as tags core. O exemplo segue a arquitetura descrita pela **Figura 2**. Trata-se de uma pequena aplicação que tem como objetivo simular a variação dos valores de ações de empresas na bolsa de valores.

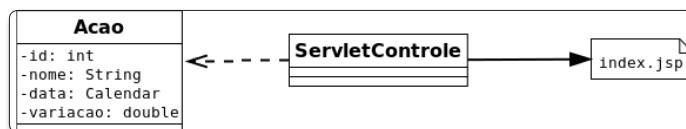


Figura 2. Arquitetura do exemplo de core tags

A **Listagem 3** mostra o código do JavaBean **Acao**. Note que não há lógica alguma nele. Esta é apenas uma classe que representa o modelo do sistema, possui algumas propriedades (**id**, **nome**, **variacao** e **data**) e métodos para manipulação destas, os sets e gets, conforme a boa prática do encapsulamento. Os objetos criados a partir desta classe serão utilizados para compor uma lista que será exibida em um JSP.

Listagem 3

```

01 package com.exemplo.bean;
02
03 import java.util.Calendar;
04
05 public class Acao {
06     private int id;
07     private double variacao;
08     private String nome;
09     private Calendar data;
10
11     public Acao() {
12     }
13
14     public int getId() {
15         return id;
16     }
17
18     public void setId(int id) {
19         this.id = id;
20     }
21
22     public double getVariacao() {
23         return variacao;
24     }
25
26     public void setVariacao(double variacao) {
27         this.variacao = variacao;
28     }
29
30     public String getNome() {
31         return nome;
32     }
33
34     public void setNome(String nome) {
35         this.nome = nome;
36     }
37
38     public Calendar getData() {
39         return data;
40     }
41
42     public void setData(Calendar data) {
43         this.data = data;
44     }
45
46     @Override
47     public boolean equals(Object obj) {
48         if (obj instanceof Acao) {
49             return ((Acao) obj).id == this.id;
50         }
51         return false;
52     }
53
54 }
  
```

Já a **Listagem 4** apresenta o código do Servlet que cria uma lista de objetos do tipo **Acao** que posteriormente será exibida em um JSP.

Sobre o servlet **ServletControle**, não há muito o que ser dito, pois ele apenas cria uma lista de ações aleatória com variações que podem ser positivas, negativas ou zeradas. O método **gerarEmpresas(int quantidade)**, descrito na linha 29, é o responsável pelas instâncias dos beans do tipo **Acao** e a configuração dos seus atributos (**id**, **nome**, **variacao** e **data**). Em um sistema real este método seria substituído pelo acesso a um banco de dados.

Listagem 4. Código do ServletControle.

```
01 package com.exemplo.controle;
02
03 import java.io.IOException;
04 import java.util.ArrayList;
05 import java.util.Calendar;
06 import java.util.Random;
07
08 import javax.servlet.ServletException;
09 import javax.servlet.annotation.WebServlet;
10 import javax.servlet.http.HttpServlet;
11 import javax.servlet.http.HttpServletRequest;
12 import javax.servlet.http.HttpServletResponse;
13
14 import com.exemplo.bean.Acao;
15
16 @WebServlet("/ServletControle")
17 public class ServletControle extends HttpServlet {
18     protected void doGet(HttpServletRequest request,
19         HttpServletResponse response) throws ServletException, IOException {
20         this.doPost(request, response);
21     }
22     protected void doPost(HttpServletRequest request,
23         HttpServletResponse response) throws ServletException, IOException {
24         request.setAttribute("timeStamp", Calendar.getInstance());
25         ArrayList<Acao> acoes = gerarEmpresas(5);
26         request.setAttribute("listaAcoes", acoes);
27         getServletContext().getRequestDispatcher("/index.jsp").forward(request, response);
28     }
29     private ArrayList<Acao> gerarEmpresas(int quantidade){
30         ArrayList<Acao> tmp = new ArrayList<Acao>();
31         Random rnd = new Random();
32         while (tmp.size() < quantidade){
33             Acao e = new Acao();
34             e.setId(tmp.size() + 1);
35             e.setNome("Empresa " + e.getId());
36             int positivoNegativoZero = rnd.nextInt(3);
37             switch (positivoNegativoZero){
38                 case 0:
39                     e.setVariacao(rnd.nextDouble());
40                     break;
41                 case 1:
42                     e.setVariacao(rnd.nextDouble()*-1);
43                     break;
44                 case 2:
45                     e.setVariacao(0);
46                     break;
47             }
48             e.setData(Calendar.getInstance());
49             tmp.add(e);
50         }
51         return tmp;
52     }
53 }
```

O `ArrayList` instanciado pelo servlet é então passado para o `index.jsp` através do `RequestDispatcher` (linha 26), que basicamente diz ao servidor web: "Eu, `ServletControle`, não sei mais o que fazer daqui para frente, mas acredito que `/index.jsp` sabe, portanto estou encaminhando para ele as minhas referências de `request` e `response` para que ele tenha acesso às informações que recebi até então (`request`) e como responder à requisição que me chamou (`response`)". Quando o método `forward()` de `RequestDispatcher`

é invocado, todos os atributos, parâmetros, cabeçalhos que o `ServletControle` recebeu da requisição HTTP são repassados para o JSP, criando um meio de `index.jsp` acessar, por exemplo, o `ArrayList` de `Acao`.

A Listagem 5 é a que realmente nos interessa, pois mostra o JSP que faz uso de JSTL para exibir o conteúdo fornecido pelo `ServletControle`.

Nesta listagem podemos ver o uso de quase todas as core tags. Na linha 41, por exemplo, fazemos uso do `forEach` para percorremos a lista de ações que o servlet disponibilizou no escopo de `request` (conforme mostra a linha 25 da Listagem 4), mas como `forEach` sabia onde procurar por "listaAcoes"? Esta é uma característica de EL, não de JSTL. Quando o escopo da variável não é conhecido, é feita uma busca pela variável em todos os escopos até ela ser encontrada. A ordem de busca é do escopo mais restritivo para o mais abrangente, portanto: `page`, `request`, `session` e `application`. Ainda na linha 41, vemos a definição da variável `acao`, que receberá cada um dos objetos de `listaAcoes`. Esta variável só existe dentro do bloco `forEach`.

Nas linhas 42 a 49 podemos observar a tríade `Choose/When/Otherwise` agindo como um bloco de `if/else` e no meio deste bloco, nas linhas 44 e 47, a criação e remoção da variável "zebra" através do uso da tag `set`. O conteúdo desta variável vai determinar a aplicação ou não do CSS em cada linha da tabela, criando o efeito "zebrado".

Já nas linhas 50 a 54 é possível observar o uso das tags `url`, `param` e `out`, para a criação de um link para outra página. Por exemplo, uma página que exiba os detalhes de uma empresa (o que não faz parte deste artigo), passando parâmetros que ajudem a identificar a empresa em questão. No exemplo, o parâmetro passado é o `id` da empresa.

Como resultado até aqui, este exemplo, executando em um servidor de aplicações web, deve apresentar uma página HTML semelhante à mostrada na Figura 3.

Nome	Variação
Empresa 1	-0.45179450645814534
Empresa 2	0.0
Empresa 3	0.0
Empresa 4	-0.018424401209918284
Empresa 5	0.6367129164344436

Figura 3. Resultado produzido pelo JSP

Internationalization

Vejamos agora as principais tags de `Internationalization` e `Formatting`. Este conjunto de tags está diretamente relacionado a como uma página é exibida, pois tem funções para alterar o formato de datas, números, moedas e mesmo o idioma no qual as mensagens de uma tela são exibidas.

`setLocale`

Esta tag é utilizada para definir o idioma/configurações regionais (formato de moeda, data e número) a ser usado pelo sistema

desenvolvido. O principal atributo é o **value**, que pode receber como valores os nomes aceitos pela própria classe **java.util.Locale** (Ex: *pt_BR*, *en_US*, *fr*, etc.) ou uma instância desta.

setBundle

Tag utilizada para carregar as informações de chave/valor de um arquivo texto. Para isso, estes arquivos devem adotar a seguinte sintaxe para a sua nomenclatura: *<nome_qualquer>_<Locale>.<extensão qualquer>*, o que permite criar nomes como *artigo_pt_BR.recursos*.

Em uma aplicação, é possível ter mais de um *Bundle* configurado. Por exemplo, poderíamos ter um grupo de arquivos para mensagens de erro (um *Bundle*) e um grupo de arquivos para labels do sistema (outro *Bundle*).

A definição do idioma é algo fundamental para que o *Bundle* possa utilizar o arquivo de mensagens correto. Esta definição é feita através do **Locale** configurado pela tag **setLocale**. Se isto não for feito, o idioma do request (idioma configurado no navegador do usuário que fez a requisição HTTP e que é enviado via cabeçalho para o servidor de aplicação) é utilizado como padrão.

Os atributos de **setBundle** são:

- **basename**: é a parte do nome do arquivo de informações anterior ao Locale. Por exemplo, em um arquivo chamado *artigo_pt_BR.properties*, o valor do atributo seria **artigo**;
- **var**: referência para o objeto do tipo *Bundle* que está sendo configurado pela tag. Assim é possível utilizá-lo em outras páginas ou mesmo servlets para a exibição de mensagens;
- **scope**: define em qual escopo a variável/propriedade será definida. Quando não informado, **page** é o escopo padrão.

Message

Definitivamente a tag mais utilizada, imprime o valor de uma chave de acordo com o idioma e bundle definidos para o sistema. Esta tag tem como principais atributos:

- **key**: corresponde ao nome da chave a ser procurada em um arquivo texto. Aceita como valores um texto ou o resultado de uma EL;
- **bundle**: indica em qual *Bundle* a chave deve ser procurada. Se o atributo **var** da tag **setBundle** foi configurado, o que indica que temos mais de um *Bundle* no sistema, é necessário informar à tag **message** o nome referenciado no atributo **var** de **setBundle**.

Listagem 5. Conteúdo do arquivo index.jsp.

```

01 <%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>
02 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
03 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
04 <!DOCTYPE html>
05 <html>
06   <head>
07     <style type="text/css">
08       table {
09         width: 50%;
10        text-align: center;
11        border-collapse: collapse;
12        border-spacing: 0;
13        border-color: black;
14        border-style: solid;
15      }
16      thead tr {
17        color: white;
18        background-color: black;
19      }
20      .zebra {
21        background-color: #CCCCCC;
22      }
23      .positivo,.negativo {
24        color: #5450C6;
25        font-weight: bold;
26      }
27      .negativo {
28        color: #EF2121;
29      }
30    </style>
31  </head>
32  <body>
33    <table>
34      <thead>
35        <tr>
36          <th>Nome</th>
37          <th>Variação</th>
38        </tr>
39      </thead>
40      <tbody>
41        <c:forEach var="acao" items="${listaAcoes}" varStatus="status" step="1">
42          <c:choose>
43            <c:when test="${status.index mod 2 == 0}">
44              <c:set var="zebra" value="class=zebra"/>
45            </c:when>
46            <c:otherwise>
47              <c:set var="zebra" value=""></c:set>
48            </c:otherwise>
49          </c:choose>
50          <c:url value="detalhe.jsp" var="det">
51            <c:param name="id" value="${acao.id}"/>
52          </c:url>
53          <tr <c:out value="${zebra}" />>
54            <td><a href="

```

De acordo com a implementação, o conteúdo de uma mensagem pode precisar de argumentos fornecidos em tempo de execução da página (por exemplo, a data de hoje). Para viabilizar essa necessidade, pode-se fazer uso de argumentos definidos nos arquivos texto através da sintaxe `{índice_do_argumento}`. A partir disso, podemos ter textos como: Hoje é dia {0} e a variação das ações da empresa {1} foi de {2}). Os valores destes argumentos podem ser passados para a mensagem inserindo no corpo da tag `message` ocorrências da tag `param`, a qual tem apenas um atributo, `value`, usado exatamente para definir o valor do argumento.

FormatNumber

Esta tag é utilizada para formatação de números, moeda e porcentagem. Além dos padrões definidos pelo `Locale`, é possível criar padrões de formatação customizados. Ela é equivalente à classe `java.text.NumberFormat`. Seus principais atributos são:

- **type**: define o tipo do número a ser formatado, tendo três possíveis valores: `number`, `currency` e `percent`;
- **value**: valor a ser formatado;
- **minFractionDigits** e **maxFractionDigits**: definem a quantidade mínima e máxima de números depois da vírgula;
- **minIntegerDigits** e **maxIntegerDigits**: similar aos atributos anteriores com o diferencial de estes tratarem da parte inteira

de um número. Se o número não atingiu a quantidade mínima configurada, será preenchido com zeros. Por sua vez, se o número ultrapassar a quantidade máxima de dígitos, este será cortado da esquerda para direita até se encaixar no valor definido.

FormatDate

Esta tag faz o trabalho correspondente ao da tag `formatNumber` para datas e tempo. Ela equivale à classe `java.text.SimpleDateFormat`. Seus principais atributos são:

- **value**: define a data a ser formatada. Este objeto deve ser uma instância de `java.util.Date`;
- **type**: define o tipo da formatação, tendo três possíveis valores: `date`, `time` e `both`;
- **dateStyle** e **timeStyle**: define o estilo de data e hora. Ambas têm cinco possíveis valores: `full`, `long`, `medium`, `short` e `default`. A forma como cada uma das opções é tradada depende do `Locale`;
- **pattern**: quando os estilos definidos não atendem à sua necessidade, é possível fazer uma customização do padrão e esta pode ser feita através deste atributo.

Exemplo com Internationalization tags

Para demonstrar os recursos de Internationalization, realizaremos algumas adaptações no exemplo utilizado em Core tags,

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

para que este tenha suporte a dois idiomas. Dito isso, a primeira coisa que precisa ser feita é a criação de três arquivos de texto, os quais são mostrados nas **Listagens 6, 7 e 8**.

Repare que as chaves, tudo que se encontra antes do sinal de igual (=), são as mesmas em todos os arquivos. O que muda é apenas o valor de cada uma delas. O arquivo apresentado na **Listagem 6**, por exemplo, é o *Locale* indefinido, utilizado quando

Listagem 6. Conteúdo do arquivo artigo.properties.

```
nome=Nome (locale desconhecido)
variacao=Variação (locale desconhecido)
```

Listagem 7. Conteúdo do arquivo artigo_pt_BR.properties.

```
nome=Nome
variacao=Variação
```

Listagem 8. Conteúdo do arquivo artigo_en_US.properties.

```
nome=Name
variacao=Variancia
```

Listagem 9. Conteúdo do arquivo index.jsp modificado para i18n.

```

01 <%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>
02 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
03 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
04 <!DOCTYPE html>
05 <fmt:setLocale value="pt_BR"/>
06 <fmt:setBundle basename="artigo"/>
07 <html>
08 <head>
09 <style type="text/css">
10 table {
11   width: 50%;
12   text-align: center;
13   border-collapse: collapse;
14   border-spacing: 0;
15   border-color: black;
16   border-style: solid;
17 }
18 thead tr {
19   color: white;
20   background-color: black;
21 }
22 .zebra {
23   background-color: #CCCCCC;
24 }
25 .positivo, .negativo {
26   color: #5450C6;
27   font-weight: bold;
28 }
29 .negativo {
30   color: #EF2121;
31 }
32 </style>
33 </head>
34 <body>
35   <table>
36     <thead>
37       <tr>
38         <th><fmt:message key="nome"/></th>
39         <th><fmt:message key="variacao"/></th>
40       </tr>
41     </thead>
42     <tbody>
43       <c:forEach var="acao" items="${listaAcoes}" varStatus="status">
44         <c:choose>
45           <c:when test="${status.index mod 2 == 0}">
46             <c:set var="zebra" value="class=zebra"/>
47           </c:when>
48           <c:otherwise>
49             <c:set var="zebra" value=""></c:set>
50           </c:otherwise>
51         </c:choose>
52         <curl value="detalhe.jsp" var="det">
53           <c:param name="id" value="${acao.id}"/>
54         </curl>
55         <tr <c:out value="${zebra}" />>
56           <td><a href=<c:out value="${det}" />></a>
57             <c:out value="${acao.nome}" /></a>
58           </td>
59           <c:choose>
60             <c:when test="${acao.variacao > 0}">
61               <font class="positivo"><fmt:formatNumber type="percent"
62                 minFractionDigits="2" value="${acao.variacao}" /></font>
63             </c:when>
64             <c:when test="${acao.variacao < 0}">
65               <font class="negativo"><fmt:formatNumber type="percent"
66                 minFractionDigits="2" value="${acao.variacao}" /></font>
67             </c:when>
68             <c:otherwise>
69               <c:out value="${acao.variacao}" />
70             </c:otherwise>
71           </c:choose>
72         </td>
73       </tr>
74     </c:forEach>
75   </tbody>
76 </table>
77 <fmt:formatDate type="both" dateStyle="default" value="${timeStamp.time}" />
78 </body>
79 </html>
```

não há compatibilidade entre o *Locale* do usuário e as opções fornecidas pela aplicação. Estes arquivos devem estar localizados no classpath da aplicação, para que possam ser encontrados pela mesma, conforme demonstra a **Figura 4**.

O arquivo *index.jsp* modificado para demonstrar os recursos de internacionalização é mostrado na **Listagem 9**.

Com as modificações apresentadas na **Listagem 9**, podemos perceber alguns detalhes referentes às configurações regionais, como ocorre na linha 5, a configuração do Locale a ser utilizado na página. Caso esta linha não existisse, o Locale fornecido pelo *request* seria o utilizado. Na linha 6 temos a criação do Bundle, onde podemos observar que só a primeira parte do nome do arquivo é suficiente para identificá-lo. Já nas linhas 38 e 39 é utilizada a tag **message** para carregar o valor das chaves especificadas no atributo **value**. Estas chaves serão buscadas nos arquivos de recursos, novamente, de acordo com o Locale e o Bundle, e os seus valores serão parte do HTML resultante deste JSP.

Prosseguindo com a análise do código, as linhas 43 e 72 marcam o início e o final do bloco **forEach**. Dentro deste bloco é utilizada

a tag **formatNumber** para a formatação de números, isto é, a definição do tipo de número (**percent**) e a quantidade mínima de dígitos depois da vírgula, como pode-se observar nas linhas 61 e 64. Por último, mas não menos importante, a data e horário que a página foi gerada é formatada através da tag **formatDate**, na linha 75. Como o tipo do atributo passado no *request* é um **java.util.Calendar**, tem-se que acessar a sua propriedade **time**, que retorna um objeto do tipo **java.util.Date**. O resultado da modificação em *index.jsp* é apresentado na **Figura 5**.

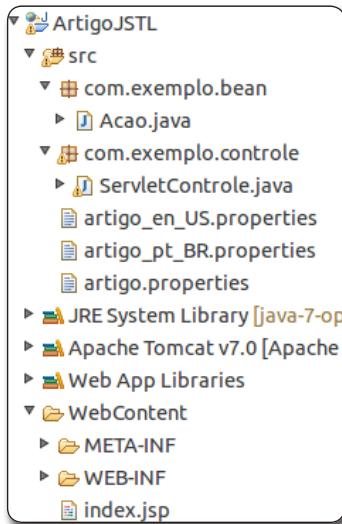


Figura 4. Projeto com arquivos de internacionalização

Nome	Variação
Empresa 1	0.0
Empresa 2	0.0
Empresa 3	36,41%
Empresa 4	-97,43%
Empresa 5	87,58%

08/04/2014 16:16:42

Figura 5. Resultado da modificação em *index.jsp*

Com esta arquitetura de internacionalização (arquivos distintos por idioma e uso de tags i18n), cada arquivo pode ser enviado para um tradutor, ou mesmo para o cliente que está requisitando a aplicação, para que eles façam a especificação dos textos que serão utilizados no sistema, tornando as páginas JSP imunes a posteriores mudanças, no que diz respeito aos textos a serem exibidos. Assim, se o cliente quiser mudar o label do campo A de um JSP, basta que ele altere o arquivo de recursos.

Além de melhorar a legibilidade do código e permitir que pessoas menos familiarizadas com Java possam contribuir com a aplicação, o uso de JSTL traz a facilidade de manutenção e internacionalização da aplicação. Portanto, conhecer este recurso é de grande importância para todo desenvolvedor que visa qualidade para seus sistemas.

Autor



Paulo M. D. Bordin

paulobordin@gmail.com

Graduado em Tecnologia em Informática pelo CEFET-PR, Pós-Graduado em Tecnologia Java pela UTF-PR. Certificado SCJP (1.4), SCWCD (1.5) e SCMAD. Atualmente é líder de projetos no HSBC Global Technologies e professor do curso de Especialização em Tecnologia Java e Desenvolvimento para Dispositivos Móveis da UTF-PR.



Links:

Página oficial do JSTL com os links para download das bibliotecas necessárias.
<https://jstl.java.net/>

Especificação do JSTL.

<https://www.jcp.org/en/jsr/detail?id=52>

Programar com Threads em Java

Como criar aplicações síncronas com Threads de forma simples e eficiente

Vamos abordar nesse artigo as principais classes e comandos da linguagem Java para se trabalhar com linhas de execução adicionais pertencentes ao mesmo programa, que são conhecidas como *threads*. Cada aplicação Java já contém um *thread* por padrão, que é o *thread* principal da própria aplicação, porém outras linhas de execução adicionais podem ser criadas para realizar tarefas simultâneas.

Um exemplo de aplicação *multi-thread* clássico é um servidor de chat, onde vários clientes podem se conectar a ele usando *sockets*, causando a realização de diversas tarefas simultaneamente, como criação de novas conexões, fechamento de conexões existentes, gerenciamento de interface gráfica, realização de *backup*, envio e recebimento de mensagens para cada cliente e outros. Usando *threads*, evitamos o caso de paralisar o servidor toda vez que uma dessas requisições precisar ser atendida, pois cada uma dessas tarefas é realizada ao mesmo tempo.

No dia a dia é comum usar aplicações que envolvam *threads*, como aplicações com interfaces gráficas (GUIs) e aplicações de comunicação. Em tais aplicações são executadas diversas tarefas simultaneamente por linhas de execução paralelas, onde o sistema operacional designa *threads* aos processadores durante intervalos de tempo que ele mesmo determina. Felizmente, Java oferece recursos de alto nível que escondem a verdadeira complexidade do tratamento de *threads* no nível do sistema operacional e do hardware, permitindo ao desenvolvedor se focar no desenvolvimento da correta sincronização do uso de recursos compartilhados por *threads* através do uso de comandos simples e eficazes.

Um problema que pode prejudicar aplicações *multi-thread* se não tratado corretamente é a sincronização do acesso a recursos compartilhados, mais comumente caracterizados na forma de variáveis e objetos. Portanto, em muitas vezes, organizações são prejudicadas por utilizarem software que contém falhas no controle de acesso a recursos compartilhados, levando a problemas

Fique por dentro

Este artigo apresenta os principais recursos da linguagem Java para o desenvolvimento de aplicações com linhas de execução concorrentes, abrangendo sincronização de sessões críticas, criação de linhas de execução adicionais com *threads*, o uso de classes auxiliares, entre outros. Tais recursos são acessíveis sob uma sintaxe simplificada e moderna, sem perder em eficiência nem legibilidade.

É apresentada também uma aplicação exemplo *multi-thread* que utiliza muitos dos recursos de concorrência, incluindo a implementação de semáforos disponibilizada pela linguagem Java. Em aplicações práticas, a correta sincronização de programas concorrentes pode evitar problemas sérios e difíceis de serem detectados. Para exemplificar a aplicação de soluções possíveis para problemas de sincronização, é proposta uma aplicação exemplo baseada numa interface gráfica que utiliza tais funcionalidades para garantir a sincronia no acesso a recursos compartilhados.

como o caso da aplicação tentar criar registros no banco de dados com violação de unicidade de chaves únicas, perda de dados, erros esporádicos na aplicação, lentidão excessiva do sistema, entre outros.

Na maioria das linguagens de programação, não é uma atividade trivial desenvolver um programa *multi-thread* que apresenta sincronização eficiente de acesso a recursos compartilhados por tais *threads*, porém, na linguagem Java, são disponibilizadas uma série de classes, comandos e métodos a fim de facilitar o controle de acesso a tais recursos, liberando o programador de precisar vincular fortemente a lógica da aplicação ao controle de *threads*, aumentando a legibilidade do código-fonte, além de sua eficiência.

Disputa de sessões críticas

Um problema muito comum que pode ocorrer na programação concorrente é a atualização de uma variável compartilhada entre vários *threads*. Na **Listagem 1** considere o trecho de código apresentado como sendo uma sessão crítica, ou seja, como sendo um trecho de código com variáveis que são visíveis e acessíveis para mais de um *thread*.

Se considerarmos que a sessão crítica na listagem é executada somente por um único *thread*, sempre será obtido o resultado correto ao final da sessão crítica. Como exemplo, levando-se em conta que *x* e *y* iniciam com o valor 1, o resultado será *x=2* e *y=4* ao final da linha 2.

Além disso, podemos considerar outro exemplo em que dois *threads* distintos, (a) e (b), estão a ponto de executar a sessão crítica e o sistema operacional, praticamente ao mesmo tempo, concede ao pedido de ambos os *threads* para executar essa sessão crítica. Nessa situação, levando-se em conta o caso de haver um único processador, enquanto um *thread* estiver executando a sessão crítica, ele pode perder o uso do processador para o outro *thread*, podendo assim levar a resultados finais incorretos, já que ambos os *threads* compartilham ambas as variáveis.

O mesmo problema ocorre em computadores com múltiplos processadores ou núcleos, pois não se pode garantir a ordem de intercalação na execução das operações de *threads* concorrentes. Considere o caso ótimo em que o *thread* (a) ganha acesso à sessão crítica e a completa e logo em seguida o *thread* (b) ganha acesso à sessão crítica e a completa, como na **Listagem 2**.

Nessa listagem, podemos verificar que o *thread* (a) executou a sessão crítica e obteve o resultado *x=2* e *y=4* (linha 2), e o *thread* (b) executou o mesmo código e obteve o resultado *x=6* e *y=11* (linha 4), que são os resultados corretos para cada *thread*. Considere agora a possibilidade em que o sistema operacional realizou a intercalação das mesmas operações em uma ordem diferente, como exemplificado na **Listagem 3**.

Listagem 1. Exemplo simples de sessão crítica.

```
01. x = x + y;
02. y = x + y + 1;
```

Listagem 2. Exemplo de sincronização de acesso à sessão crítica (por coincidência).

```
01.(a) x = x + y;           // (x=2, y=1)
02.(a) y = x + y + 1;       // (x=2, y=4) : resultado para (a)
03.(b) x = x + y;           // (x=6, y=4)
04.(b) y = x + y + 1;       // (x=6, y=11) : resultado para (b)
```

Listagem 3. Exemplo de acesso à sessão crítica com falha de sincronização.

```
01.(a) x = x + y;           // (x=2, y=1)
02.(b) x = x + y;           // (x=3, y=1) : resultado para (a)
03.(b) y = x + y + 1;       // (x=3, y=5)
04.(a) y = x + y + 1;       // (x=3, y=9) : resultado para (b)
```

Como podemos verificar, o *thread* (a) executou a sessão crítica e obteve o resultado *x=3* e *y=9* (linha 4), e o *thread* (b) executou o mesmo código e obteve o resultado *x=6* e *y=5* (linha 3), diferindo dos resultados obtidos na **Listagem 2**. Uma vez que o sistema operacional não tem como garantir a ordem de execução das operações intercaladas, confirmamos que a ordem de execução das operações pelo sistema operacional é importante para definir o resultado final das operações.

Para evitar tal problema, precisamos de mecanismos que nos façam garantir a execução sincronizada, como na **Listagem 2**, que

leva a resultados corretos, onde um *thread* executa a sessão crítica por vez. Tal situação é comumente referenciada como “disputa” (race), momento em que dois *threads* disputam uma sessão crítica, e a solução é fazer com que somente um *thread* acesse por vez a sessão crítica, ou seja, garantir a exclusão mútua. Se o outro tentar acessar a sessão crítica nesse intervalo de tempo, ficará bloqueado até o primeiro *thread* terminar.

O referido problema não ocorre somente com variáveis simples, mas sim com todo o tipo de recurso computacional, como com vetores, *sockets*, impressoras, recursos do sistema operacional e outros. Felizmente, Java oferece comandos simples para realizar a sincronização de sessões críticas, como o comando **synchronized** e suas várias aplicações.

Sincronização de métodos e de blocos de comandos

Em Java, a forma mais simples e intuitiva de controlar o acesso a uma sessão crítica no código-fonte é criar um método sincronizado usando a palavra-chave **synchronized**, como na **Listagem 4**. Dessa forma, a linguagem Java gerencia automaticamente a sincronização dos *threads*, impedindo a execução intercalada de código com acesso a recursos compartilhados.

Como o método **sessaoCritica()** é declarado com o modificador **synchronized**, o Java garante que nunca vai ocorrer problema de disputa na execução da sessão crítica, porque sempre que um *thread* estiver executando esse método, qualquer outro *thread* que tentar executar o mesmo Será automaticamente bloqueado, até que o primeiro *thread* termine a execução desse método.

Como *x* e *y* são variáveis de instância e o método **sessaoCritica()** é um método de instância, a sincronização ocorre em nível de instância, pois, nesse caso, não há necessidade de bloquear o acesso à sessão crítica de *threads* distintos que acessem diferentes instâncias, uma vez que as variáveis de uma instância são independentes de qualquer outra instância.

Porém, de forma análoga, é possível também usar **synchronized** no contexto da classe. Neste caso a sincronização ocorre para qualquer *thread*, independente de instância, sendo assim o método declarado como estático, como na **Listagem 5**.

Dessa forma, as variáveis *x* e *y* são estáticas, e, portanto pertencem à classe, e como o método **sessaoCritica()** também pertence à classe por ser estático, se torna adequado que o mesmo seja acessível por somente um *thread* por vez.

Listagem 4. Método sincronizado para a sessão crítica: contexto de instância.

```
01. public synchronized int[] sessaoCritica() {
02.     x = x + y;
03.     y = x + y + 1;
04. }
```

Listagem 5. Método sincronizado para a sessão crítica: Contexto de classe.

```
01. public static synchronized int[] sessaoCritica() {
02.     x = x + y;
03.     y = x + y + 1;
04. }
```

Programar com Threads em Java

Tal configuração é útil em casos onde se deseja sincronizar o acesso a um recurso compartilhado pertencente a uma classe, existindo diversas instâncias da classe que acessem esse recurso.

Uma forma alternativa e semelhante de realizar a sincronização de sessões críticas é usando blocos sincronizados, como na **Listagem 6**, onde um exemplo com uma classe completa é exibido, usando sincronização por instância.

Segundo a forma de sincronização por instância exemplificado na listagem, o método `sessaoCritica()` não é sincronizado, porém o bloco `synchronized` é. Quando `synchronized` é usado dessa forma, é necessário que seja informado um parâmetro, que deve ser um objeto (linha 11), e como efeito, o bloco será sincronizado em relação ao objeto informado, sendo, nesse caso, correspondente a `this`, que representa a própria instância. Além disso, essa configuração é equivalente ao caso de se usar um método `synchronized` de instância, como na **Listagem 4**.

De forma análoga, podemos ter também blocos sincronizados em relação à classe, como na **Listagem 7**.

Listagem 6. Blocos sincronizados: Contexto de instância.

```
01. package pkg;
02. public class Teste1 {
03.     int x = 1;
04.     int y = 1;
05.     public static void main(String[] args) {
06.         Teste1 p = new Teste1();
07.         p.sessaoCritica();
08.         System.out.println("x=" + p.x + " y=" + p.y);
09.     }
10.     public void sessaoCritica() {
11.         synchronized (this) {
12.             x = x + y;
13.             y = x + y + 1;
14.         }
15.     }
16. }
```

Listagem 7. Blocos sincronizados: Contexto de classe.

```
01. package pkg;
02.
03. public class Teste2 {
04.     static int x = 1;
05.     static int y = 1;
06.     public static void main (String[] args) {
07.         sessaoCritica();
08.         System.out.println("x=" + x + " y=" + y);
09.     }
10.     public static void sessaoCritica() {
11.         synchronized (Teste2.class) {
12.             x = x + y;
13.             y = x + y + 1;
14.         }
15.     }
16. }
```

Essa listagem apresenta uma formulação da classe onde as variáveis `x` e `y` são estáticas e são compartilhadas por todas as instâncias, ou seja, há somente um `x` e um `y` para a classe, levando o método `sessaoCritica()` (linha 10) a ser estático. Nesse caso de sincronização pela classe, o bloco `synchronized` (linha 11) recebe

como parâmetro o objeto `class` referente à classe em que se deseja sincronizar, que é retornada pelo atributo `Teste2.class`, sendo esse caso equivalente ao da **Listagem 5**, onde as variáveis `x` e `y` são também estáticas.

Outro problema muito comum que pode ocorrer em aplicações *multi-thread* é a espera ocupada, em que um *thread* fica preso em um laço esperando por uma condição de parada, levando ao desperdício do uso do processador, causando lentidão excessiva no processamento do programa que contém o *thread*, como exemplificado na **Listagem 8**.

Listagem 8. Exemplo de espera ocupada.

```
01. package pkg;
02. public class EsperaOcupada {
03.     public static int x = 1;
04.     public static void main(String[] args) {
05.         while (x != 2);
06.     }
07. }
```

Nesse código ocorre a espera ocupada, o que faz com que o programa fique preso em um laço infinito (linha 5), e mesmo se o outro *thread* alterar a variável compartilhada `x` para terminar a espera ocupada, uma quantidade muito grande de processamento já pode ter sido desperdiçada nesse tempo. Uma solução possível seria colocar o *thread* corrente para dormir durante um intervalo de tempo, perdendo a CPU, mas ainda assim estaremos desperdiçando o uso da CPU pois a espera ocorre, igualmente, dentro de um laço infinito. Além disso, o recurso compartilhado pode ser liberado logo no início de um dos tempos de espera, privando o *thread* de executar até completar a espera.

A solução correta é usar um comando que informe ao sistema operacional que esse *thread* vai dormir, fazendo com que fique bloqueado sem usar o processador até que outro *thread* o acorde, otimizando assim o uso dos recursos computacionais, sem prejuízo. O comando disponibilizado pela linguagem Java, nesse caso, é o método `wait()`, e será comentado em mais detalhes na apresentação da classe `Thread` mais à frente.

Linhas de execução adicionais com a interface Runnable

Vimos até aqui como criar blocos sincronizados. Agora será mostrado como criar linhas de execução adicionais de forma simples usando a interface `Runnable`, de forma que a classe que a implementa deve conter a implementação do método `run()`, que é o código a ser executado concorrentemente. Dada uma classe que implementa `Runnable`, para criar um *thread*, deve ser invocado o construtor da classe `Thread` recebendo como parâmetro o `Runnable`, e para iniciar o *thread*, deve-se usar o método `start()` da classe `Thread`, como visto na **Listagem 9**.

Na mesma listagem, a classe `Teste3` implementa a interface `Runnable` (linha 2) e cria as variáveis estáticas `x` e `y` (linhas 3 e 4) para serem compartilhadas por várias linhas de execução concorrentes. Através do correto uso de sincronização é evitada a

execução intercalada das operações da sessão crítica, fazendo que os valores corretos das variáveis sejam escritos corretamente ao final do processamento de cada linha de execução adicional.

Listagem 9. Linha de comandos adicionais com Runnable.

```
01. package pkg;
02. public class Teste3 implements Runnable {
03.     static int x = 1;
04.     static int y = 1;
05.     public void run() {
06.         sessaoCritica();
07.     }
08.     public static void sessaoCritica() {
09.         synchronized (Teste2.class) {
10.             x = x + y;
11.             y = x + y + 1;
12.             System.out.println("x=" + x + " y=" + y);
13.         }
14.     }
15.     public static void main(String[] args) {
16.         Thread t1 = new Thread(new Teste3());
17.         Thread t2 = new Thread(new Teste3());
18.         Thread t3 = new Thread(new Teste3());
19.         t1.start();
20.         t2.start();
21.         t3.start();
22.     }
23. }
```

Outra informação importante é que a classe **Teste3** implementa o método **run()** (linhas 5 a 7), que chama o método estático **sessaoCritica()** (linhas 8 a 14), que contém um bloco sincronizado (em relação à própria classe), pois **x** e **y** são variáveis de classe. Como resultado da sincronização, é garantida a exclusão mútua, levando a uma correta manipulação das variáveis **x** e **y** dentro da sessão crítica, não importando quantas linhas de execução requisitem a execução da sessão compartilhada.

Ainda na **Listagem 9**, o método **main()** (linhas 15 a 22), cria três **threads** distintos (linhas 16 a 18) passando como parâmetro um **Runnable**, que é, nesse caso, uma instância da classe **Teste3**. Por fim, os **threads** são disparados para execução concorrente nas linhas 19 a 21 usando o método **start()** da classe **Thread**.

Observe que mesmo chamando o método **start()**, a linha de execução principal fica liberada para processar o próximo comando, pois esse método dispara a execução de um **thread** que passa a executar concorrentemente com qualquer outro **thread** da aplicação.

Apesar da simplicidade de se usar os poderosos recursos da interface **Runnable**, não acabam aqui os recursos da linguagem Java para criar e gerenciar as linhas de execução, pois a classe **Thread** também pode ser utilizada independentemente para criar **threads** e ainda oferece uma série de comandos para controle dos mesmos.

Linhas de execução adicionais com a classe Thread

Outra forma de se criar linhas de execução adicionais é usando diretamente a classe **Thread**. No entanto, **Runnable** se torna a opção correta quando se deseja converter em linha de execução

uma classe que já estende alguma outra, visto que em Java não há herança múltipla. **Thread** é a classe que representa qualquer linha de execução, que pode ser de uma das seguintes categorias: **thread** principal (que executa o método **main()**), **daemon** (segundo plano) e **thread** do usuário (threads normais). Além disso, a classe **Thread** oferece métodos capazes de controlar a execução das linhas execução, abrangendo operações como parar, pausar por um determinado tempo, retomar a execução, entre outros.

Os principais construtores da classe **Thread** são:

- **Thread()**: Cria um novo **thread**;
- **Thread(Runnable target)**: Cria um novo **thread** usando o objeto **target** que implementa **Runnable**. O método **run()** desse objeto contém o código a ser executado concorrentemente;
- **Thread(Runnable target, String name)**: Cria um novo **thread** usando o objeto **target** que implementa **Runnable**. O método **run()** desse objeto contém o código a ser executado concorrentemente. Além disso, permite informar o nome do **thread**;
- **Thread(String name)**: Cria um novo **thread**, informando ao mesmo tempo o nome do mesmo.

E os principais métodos da classe **Thread** são:

- **Thread currentThread()**: Retorna o **thread** corrente;
- **void dumpStack()**: Imprime a **stacktrace** (pilha de métodos) do **thread** corrente na saída de erro;
- **ClassLoader getContextClassLoader()**: Retorna o **ClassLoader** usado para criar esse **thread**, ou seja o carregador de classes;
- **long getId()**: Retorna o identificador do **thread**;
- **String getName()**: retorna o nome do **thread**;
- **int getPriority()**: Retorna a prioridade de execução desse **thread**, dada por uma constante da classe **Thread**. As opções são: MAX_PRIORITY, MIN_PRIORITY e NORM_PRIORITY;
- **Thread.State getState()**: Retorna o estado do **thread**;
- **void interrupt()**: Interrompe o **thread**;
- **boolean interrupted()**: Testa se o **thread** corrente está interrompido;
- **boolean isAlive()**: Testa se o **thread** está vivo;
- **boolean isDaemon()**: Testa se o **thread** é um **daemon**;
- **boolean isInterrupted()**: Testa se esse **thread** está interrompido;
- **void join()**: Aguarda até esse **thread** terminar;
- **void join(long milliseconds)**: Aguarda o término desse **thread** até, no máximo, a quantidade informada de milissegundos;
- **void run()**: Corresponde ao código a ser executado pelo **thread**, mas primeiro **run()** precisa ser sobreescrito através de herança com a classe **Thread**. Se esse **thread** foi criado informando um **Runnable** no construtor, esse método não faz nada, pois ao invés dele, será executado o **run()** do objeto **Runnable**;
- **void setContextClassLoader(ClassLoader loader)**: Define o **ClassLoader** para esse **thread**, ou seja o carregador de classes;
- **void setDaemon(boolean on)**: se o parâmetro for verdadeiro, define esse **thread** como um **daemon**, caso contrário, define como **thread** do usuário;
- **void SetName()**: Define o nome do **thread**;
- **void setPriority()**: Define a prioridade do **thread**;

- **void sleep(milliseconds):** Faz o *thread* dormir pela quantidade indicada de milissegundos, perdendo o processador;
- **void start():** Faz o *thread* iniciar a sua execução, ou seja, a máquina virtual chama o método **run()** do thread (ou do **Runnable** informado);
- **String toString():** Retorna uma representação textual desse *thread*;
- **static void yield():** Avisa ao sistema operacional que o *thread* corrente não necessita executar nesse momento, liberando o processador;
- **void wait():** Método herdado da classe **Object**, faz o *thread* corrente dormir (parar sua execução) até que outro *thread* chame o método **notify()** desse *thread* ou **notifyAll()**;
- **void wait(timeout):** Método herdado da classe **Object**, faz o *thread* corrente parar sua execução até que outro *thread* chame o método **notify()** ou **notifyAll()** desse *thread*. Se a quantidade informada de milissegundos for atingida na espera, o *thread* retorna a sua execução normalmente;
- **void notify():** Método herdado da classe **Object**, acorda o *thread* que está aguardando no monitor desse objeto;
- **void notifyAll():** Método herdado da classe **Object**, acorda todos os *threads* que estão aguardando no monitor desse objeto.

Classes auxiliares

Além da classe **Thread**, existem outras classes e interfaces que foram disponibilizadas desde o Java 5 que facilitam a tarefa da programação concorrente. Antes da criação dessas classes, o programador deveria criar suas próprias classes auxiliares, visando facilitar a programação concorrente, como a criação de semáforos, listas sincronizadas e outras, o que é uma atividade não muito simples. Felizmente, um conjunto de classes e interfaces auxiliares estão disponíveis no pacote **java.util.concurrent**, e as principais são:

- **Semaphore:** É um semáforo que contabiliza a quantidade de *threads* que podem usar a sessão crítica ao mesmo tempo, sendo usado para realizar a sincronização de *threads*, ou seja, para proteção da sessão crítica. Seus principais métodos são:

- **Semaphore(int permits):** Cria um novo objeto **Semaphore** com o número indicado de *threads* que podem ser permitidas ao mesmo tempo no semáforo;
 - **acquire():** Faz uma requisição para obter uma posição disponível no semáforo e espera até conseguir, liberando o thread para acessar a sessão crítica;
 - **release():** Libera a posição ocupada no semáforo;
 - **reducePermits():** Diminui a quantidade de *threads* permitidos por vez no semáforo;
 - **tryAcquire():** Se está disponível uma posição no semáforo, bloqueia essa posição, liberando o acesso do thread à sessão crítica, caso contrário não espera e nega o acesso;
 - **drainPermits():** Adquire o bloqueio e inclusive de todas as outras posições disponíveis.
- **Exchanger:** Classe que é usada para realizar a troca de dados entre dois *threads*. O **Exchanger** aguarda até ambos os *threads* realizarem a troca usando recursos de sincronização;

- **ThreadFactory:** Interface que permite definir classes que criam *threads* usando como parâmetro objetos da classe **Runnable**;
- **Phaser:** Cria uma barreira de execução para vários *threads*, ou seja, cria um ponto em que todos os *threads* precisam parar e esperar juntos para, após isso, continuar;
- **FutureTask:** Implementa a interface **Future** para definir uma tarefa que será realizada futuramente após o processamento de outro *thread*;
- **SynchronousQueue:** Uma fila com bloqueio em que uma operação com a inserção de um elemento causa um bloqueio de execução até que outro *thread* execute a remoção correspondente, liberando o objeto bloqueado.

GUI da aplicação exemplo

Para demonstrar a utilização prática dos principais recursos de programação concorrente da linguagem Java, é proposta uma aplicação exemplo que cria vários *threads*. Cada *thread* assume o papel de um corredor que corre horizontalmente na tela, pintando bloco por bloco, até chegar ao final do percurso. Quando o último *thread* chegar ao final (à direita), todos são liberados para reiniciar a corrida.

A cor verde é utilizada para demarcar o passo não percorrido, azul para demarcar o passo percorrido e vermelho para determinar passo bloqueado pelo usuário. Durante o percurso horizontal de cada *thread* os botões verdes vão sendo coloridos de azul da esquerda para a direita e o usuário pode clicar em qualquer passo verde para causar a paralisação do corredor quando ele chegar àquele passo. Para liberar o corredor, o usuário deve desbloquear o passo bloqueado, clicando no referido passo em vermelho.

A GUI da aplicação exemplo é exposta na **Figura 1**, onde nenhum corredor iniciou ainda a correr, mas já há alguns bloqueios do usuário disponíveis.



Figura 1. Estado inicial dos corredores (*threads*) com três posições bloqueadas

Podemos ver que existem cinco *threads* corredores, e pela **Figura 2**, podemos verificar um exemplo no qual os 5 corredores correram até o final, mas os três corredores de baixo pararam no bloco de bloqueio (em vermelho). Se o usuário clicar em cada um desses blocos bloqueados, esses três corredores terminarão e a GUI voltará ao estado da **Figura 1**, porém sem bloqueios.

A aplicação exemplo é composta por várias classes e a principal delas é a interface gráfica, que tem sua primeira parte declarada na **Listagem 10**, onde os nomes dos *threads* são escritos por

rótulos (objetos da classe **Label**) e os blocos são representados por botões (objetos da classe **Button**), ou seja, eles são os retângulos coloridos nas **Figuras 1 e 2**.

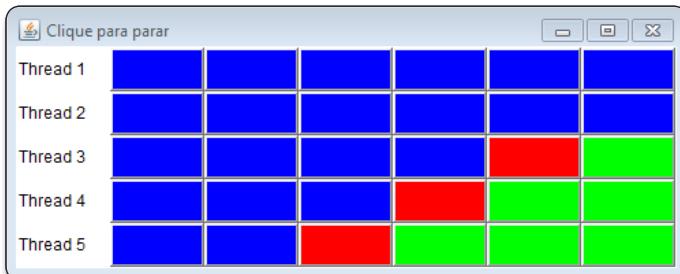


Figura 2. Estado intermediário com três posições bloqueadas

Listagem 10. Parte inicial da classe ThreadAppGUI.

```

01. package pkg;
02.
03. import java.awt.Button;
04. import java.awt.Frame;
05. import java.awt.GridLayout;
06. import java.awt.Label;
07. import java.awt.event.WindowAdapter;
08. import java.awt.event.WindowEvent;
09.
10. public class ThreadAppGUI extends Frame
11. {
12.     private static final long serialVersionUID = 1L;
13.
14.     private Label labelThread1 = new Label("Thread 1");
15.     private Label labelThread2 = new Label("Thread 2");
16.     private Label labelThread3 = new Label("Thread 3");
17.     private Label labelThread4 = new Label("Thread 4");
18.     private Label labelThread5 = new Label("Thread 5");
19.
20.     private static final int numeroPassos = 6;
21.     private Button[] botoesThread1 = new Button[numeroPassos];
22.     private Button[] botoesThread2 = new Button[numeroPassos];
23.     private Button[] botoesThread3 = new Button[numeroPassos];
24.     private Button[] botoesThread4 = new Button[numeroPassos];
25.     private Button[] botoesThread5 = new Button[numeroPassos];
26.
27.     protected ThreadCorredor thread1 = new ThreadCorredor(botoesThread1, 600);
28.     protected ThreadCorredor thread2 = new ThreadCorredor(botoesThread2, 700);
29.     protected ThreadCorredor thread3 = new ThreadCorredor(botoesThread3, 800);
30.     protected ThreadCorredor thread4 = new ThreadCorredor(botoesThread4, 900);
31.     protected ThreadSincroniza thread5 = new ThreadSincroniza
        (botoesThread5, 1000, new ThreadCorredor[]
        {thread1, thread2, thread3, thread4});
    }

```

Nesse código é exibido o início da declaração da classe **ThreadAppGUI**, que é um Frame AWT pois estende **Frame** (linha 10). Os *labels* de cada *thread* são criados nas linhas 14 a 18. O número de passos que cada *thread* deve percorrer é determinado na linha 20, sendo definido estaticamente como seis, ou seja, serão criados seis botões para cada *thread* que servirão para desenhar o caminho a ser percorrido. Os botões são criados nas linhas 21 a 25.

Nas linhas 27 a 30 são criados os quatro primeiros *threads*, usando como tempo para percorrer cada passo 600, 700, 800 e 900 milissegundos, sendo **ThreadCorredor** a classe de cada *thread*.

O último *thread*, de classe **ThreadSincroniza**, é criado na linha 31, e tem a função de esperar as outras pararem para reiniciar o percurso. Ele recebe os outros *threads* como parâmetro para que possa acessá-los.

Na **Listagem 11** é apresentado o construtor da classe **ThreadAppGUI**, tendo a função de configurar a janela principal.

Listagem 11. Construtor da classe ThreadAppGUI.

```

32.     public ThreadAppGUI () {
33.         super("Clique para parar");
34.         add(labelThread1);
35.         adicionarVetorBotoes(botoesThread1, thread1);
36.         add(labelThread2);
37.         adicionarVetorBotoes(botoesThread2, thread2);
38.         add(labelThread3);
39.         adicionarVetorBotoes(botoesThread3, thread3);
40.         add(labelThread4);
41.         adicionarVetorBotoes(botoesThread4, thread4);
42.         add(labelThread5);
43.         adicionarVetorBotoes(botoesThread5, thread5);
44.         setLayout(new GridLayout(5, numeroPassos+1));
45.         setSize(500,200);
46.         addWindowListener(new WindowAdapter() {
47.             public void windowClosing(WindowEvent we) {
48.                 System.exit(0);
49.             }
50.         });
51.     }

```

Na linha 33 é chamado o construtor da classe **Frame** com o título da janela. Na linha 34 é adicionado o rótulo do primeiro *thread* à janela e na linha 35 é usado o método **adicionarVetorBotoes()** para criar e adicionar à janela os botões pertencentes a cada corredor (*thread*). Para cada outro *thread*, ocorre de forma semelhante à criação do rótulo e dos botões, até terminar na linha 43.

Na linha 44 é definido o *layout* da janela, que é um **GridLayout** e tem a função de desenhar e gerenciar o *layout* de uma matriz de 5 linhas e 7 colunas que contém os elementos de interface gráfica pertencentes à janela; no caso, somente rótulos e botões. Finalmente, na linha 45, é definido o tamanho da janela e na linha 46, é definido o *listener* para fechar a GUI quando o usuário clicar no botão de fechar janela, chamando **System.exit()**.

Na **Listagem 12** é detalhada a implementação do método **adicionarVetorBotoes()**, que tem a função de, dado um *thread* e seu vetor de botões (que está preenchido inicialmente com *nulls*) (linha 52), cria cada objeto da classe **Button** e insere-o no vetor (linha 54), adiciona-o na janela (linha 55) e cria um **ActionListener** para ele (linha 56), que tem a função de ser chamado quando o usuário clicar no botão.

Na **Listagem 13**, por sua vez, é apresentado o método **startThreads()**, que informa a cada *thread* o *thread* sincronizador (linhas 60 a 63) e por fim cada um dos cinco *threads* é iniciado pelo método **start()** da classe **Thread** (linhas 64 a 68).

O último método da classe **ThreadAppGUI** é apresentado na **Listagem 14**. Este corresponde ao método **main()**, que tem a função de iniciar a aplicação de forma a criar a janela gráfica (linha 70) e torná-la visível (linha 71). Por fim, na linha 73, são iniciados todos os *threads* pelo método **startThreads()**.

Programar com Threads em Java

Listagem 12. Método adicionarVetorBotoes() da classe ThreadAppGUI.

```
52. public void adicionarVetorBotoes(Button[] botoesThread,  
    ThreadGenerico thread) {  
53.     for (int numeroBotao = 0; numeroBotao < botoesThread.length;  
        numeroBotao++) {  
54.         botoesThread[numeroBotao] = new Button("");  
55.         add(botoesThread[numeroBotao]);  
56.         botoesThread[numeroBotao].addActionListener(new ActionListener  
            Thread(numeroBotao, thread));  
57.     }  
58. }
```

Listagem 13. Método startThreads() da classe ThreadAppGUI.

```
59. public void startThreads() {  
60.     thread1.setThreadSincronizador(thread5);  
61.     thread2.setThreadSincronizador(thread5);  
62.     thread3.setThreadSincronizador(thread5);  
63.     thread4.setThreadSincronizador(thread5);  
64.     thread1.start();  
65.     thread2.start();  
66.     thread3.start();  
67.     thread4.start();  
68.     thread5.start();  
69. }
```

Listagem 14. Método main() da classe ThreadAppGUI.

```
70. public static void main(String[] args) {  
71.     ThreadAppGUI gui = new ThreadAppGUI();  
72.     gui.setVisible(true);  
73.     gui.startThreads();  
74. }  
75. }
```

Nessa listagem, o método **main()** é executado pelo *thread* principal da aplicação, que cria o GUI (linha 71) e o exibe (linha 72). Os *threads* do usuário são criados na linha 73, causando o encerramento do *thread* principal, pois ele termina junto com o término do método **main()**, deixando em execução os outros *threads*. Se for necessário finalizar a aplicação, o que inclui eliminar todos os *threads*, pode-se usar o comando **System.exit()**, como na linha 48 da Listagem 11.

Outra classe importante para o processamento da GUI de nossa aplicação é a classe **ActionListenerThread**. Esta executa o código resultante da ação de cada botão exibido na GUI, usando corretamente o controle de concorrência (veja a Listagem 15).

A classe **ActionListenerThread** recebe como parâmetro no construtor o número do botão a que ela se refere e o corredor do botão (*thread* correspondente). Quando ocorrer o evento relacionado ao *listener*, que é o clique do botão, o método **setNumeroBotao()** do *thread* é chamado para aplicar o evento na interface gráfica trocando a cor do botão e atualizando as variáveis internas do *thread* a fim de possibilitar o futuro bloqueio desse corredor quando ele chegar no passo correspondente ao botão clicado (marcado agora em vermelho).

Uma observação importante é que, na linha 17, é usado um bloco estático na implementação da ação do *listener* visando evitar a possibilidade de execução concorrente do método **setNumeroBotao()** no caso do usuário clicar quase que ao mesmo tempo em botões

do mesmo *thread*. Como a sincronização ocorre pela instância do *thread* (linha 17), se o usuário clicar em botões de *threads* diferentes não há nenhum problema, pois as variáveis de *threads* diferentes são distintas, permitindo o processamento concorrente.

Por fim, na linha 18 ocorre a chamada do método **setNumeroBotao()** e na linha 19 é usado **notify()** para acordar o *thread* caso ele esteja parado, ou seja, preso no **wait()** por ter encontrado o botão vermelho.

Uma observação importante para que se possa compreender corretamente a utilização de *threads* em aplicações GUI e evitar problemas de travamento da interface gráfica ao executar eventos do usuário demorados, é notar que a janela gráfica já contém um *thread* próprio para desenhar e atualizar a interface gráfica. Nesse caso, para evitar esse problema, pode-se criar um novo *thread* para realizar exclusivamente o processamento do evento.

Listagem 15. Código da classe ActionListenerThread.

```
01. package pkg;  
02.  
03. import java.awt.event.ActionEvent;  
04. import java.awt.event.ActionListener;  
05.  
06. public class ActionListenerThread implements ActionListener {  
07.  
08.     private int numeroBotao;  
09.     private ThreadGenerico thread;  
10.  
11.     public ActionListenerThread(int numeroBotao, ThreadGenerico thread) {  
12.         this.numeroBotao = numeroBotao;  
13.         this.thread = thread;  
14.     }  
15.  
16.     public void actionPerformed(ActionEvent e) {  
17.         synchronized (thread) {  
18.             thread.setNumeroBotao(numeroBotao);  
19.             thread.notify();  
20.         }  
21.     }  
22. }
```

Threads da aplicação exemplo

Existem dois tipos de *threads* na aplicação exemplo: **ThreadCorredor** e **ThreadSincroniza**. Na Figura 1, as quatro primeiras linhas correspondem a *threads* da classe **ThreadCorredor** e a última linha corresponde a um *thread* da classe **ThreadSincroniza**. **Thread corredor** é o *thread* designado para correr, e o *thread* sincronizador é o *thread* que, além de correr, espera as outras terminarem para iniciar uma nova corrida.

Como ambos os tipos de *thread* compartilham muitas definições em comum, pois ambos correm o percurso, é proposto criar os *threads* de forma hierárquica, em que elas herdam atributos e métodos de uma classe pai, chamada de **ThreadGenerico**, que tem sua primeira parte declarada na Listagem 16. Observe na linha 7 que ela estende **Thread**, porém, como sendo uma classe abstrata, ela não pode ser instanciada, pois há sentido somente em instanciar suas classes filhas.

Listagem 16. Código da classe ThreadGenerico – primeira parte.

```
01. package pkg;  
02.  
03. import java.awt.Button;  
04. import java.awt.Color;  
05. import java.util.concurrent.Semaphore;  
06.  
07. public abstract class ThreadGenerico extends Thread {  
08.  
09.     protected Button[] botoes;  
10.    protected int numeroBotao;  
11.    protected int passo;  
12.    Semaphore semaforo = new Semaphore(1);  
13.    protected static final Color PERCORRIDO = Color.blue;  
14.    protected static final Color AGUARDANDO = Color.green;  
15.    protected static final Color PARADO = Color.red;
```

Nesse código, pode-se conferir que os atributos que fazem parte de ambos os modelos de *threads* usados são um vetor de botões (linha 9), o número do botão marcado para parar o *thread* (linha 10), o passo do *thread*, ou seja, o tempo que ele demora para “percorrer” cada posição (linha 11), um semáforo que indica que o *thread* ainda está correndo (linha 12) e constantes para as cores usadas para colorir os botões da GUI, chamadas de *PERCORRIDO*, *AGUARDANDO* e *PARADO* (linhas 13 a 15). Na **Listagem 17** são apresentados o construtor e diversos métodos auxiliares.

Listagem 17. Código da classe ThreadGenerico – segunda parte.

```
16. public ThreadGenerico (Button[] botoes, int passo) {  
17.     this.botoes = botoes;  
18.     this.numeroBotao = botoes.length;  
19.     this.passo = passo;  
20. }  
21.  
22. public int getNumeroBotao() {  
23.     return numeroBotao;  
24. }  
25.  
26. public void liberaBotao() {  
27.     setNumeroBotao(getNumeroBotao());  
28. }  
29.  
30. public Semaphore getSemaforo() {  
31.     return semaforo;  
32. }  
33.  
34. public Button getBotoe(int numero) {  
35.     return botoes[numero];  
36. }  
37.  
38. public void pausar() {  
39.     synchronized (this) {  
40.         try {  
41.             wait();  
42.         } catch (InterruptedException e) {  
43.             e.printStackTrace();  
44.         }  
45.     }  
46. }
```

Na linha 16 é iniciada a codificação do construtor, e na linha 18, o número do botão marcado pelo usuário para pausar o *thread* é definido como sendo o tamanho do vetor de botões, valor que é usado para indicar que não há nenhum botão marcado, fazendo que todos os botões desse corredor fiquem verdes. Entretanto, se o usuário clicar em algum botão, o campo **numeroBotao** vai ser alterado para a posição do botão clicado, marcando ele como vermelho para ocorrer a parada programada nele.

Além disso, na **Listagem 17** são apresentados os métodos **getNumeroBotao()**, **liberaBotao()**, **getSemaforo()**, **getBotoe()** e **pausar()**. Dentre esses, **getNumeroBotao()**, **getSemaforo()** e **getBotoe()** são *getters*, ou seja, têm a função de retornar atributos.

O método **liberaBotao()** chama o método **setNumeroBotao()** para redefinir o número do botão marcado, e como está sendo informado o mesmo índice do botão que já está marcado, **liberaBotao()** simplesmente desmarca o botão marcado, sendo utilizado na **Listagem 18**.

Por fim, o método **pausar()** chama o método **wait()** para pausar o *thread* com a vantagem de não usar espera ocupada, pois **pausar()** libera o processador para ser utilizado em outros threads ou processos a critério do sistema operacional. É importante notar que, na linha 41, o método **wait()** é executado com *lock* da instância (a *thread* corrente é o objeto do *lock*, linha 39). Por último, é usado *try-catch* para capturar a exceção **InterruptedException** que pode ser lançada.

Listagem 18. Código da classe ThreadGenerico – última parte.

```
47. public void setNumeroBotao(int numeroBotao) {  
48.     if (botoes[numeroBotao].getBackground().equals(PERCORRIDO)) {  
49.         } else if (numeroBotao == this.numeroBotao) {  
50.             botoes[numeroBotao].setBackground(AGUARDANDO);  
51.             this.numeroBotao = botoes.length;  
52.         } else if (this.numeroBotao == botoes.length) {  
53.             botoes[numeroBotao].setBackground(PARADO);  
54.             this.numeroBotao = numeroBotao;  
55.         } else {  
56.             liberaBotao();  
57.             botoes[numeroBotao].setBackground(PARADO);  
58.             this.numeroBotao = numeroBotao;  
59.         }  
60.     }  
61. }  
62. }
```

Além disso, na listagem é apresentado o método que é responsável por realizar a marcação do botão clicado pelo usuário, atualizando as variáveis de estado do *thread* e atualizando a GUI. O método inicia na linha 47 e recebe como parâmetro a posição que o usuário clicou (dado pelo *listener* na **Listagem 15**).

Esse método é composto por uma série de testes lógicos encadeados. O primeiro teste não resulta em nenhum processamento (linha 48), pois testa o caso do usuário ter clicado em um botão já percorrido pelo *thread*. No segundo teste (linha 50), é verificado se o botão clicado pelo usuário se encontra marcado como bloqueado. Caso positivo, o botão é desbloqueado e marcado com o status *AGUARDANDO*.

No terceiro teste (linha 53), o botão está livre para ser marcado e é marcado como PARADO, ou seja, o *thread* vai parar quando chegar nele. E por fim, iniciando na linha 57, ocorre o caso do usuário marcar um botão, mas já houver um botão diferente marcado no mesmo *thread*. Sendo assim, ele é desmarcado na linha 57 e o botão clicado é marcado como PARADO no final.

O *thread* corredor é apresentado na **Listagem 19**, dado pela classe **ThreadCorredor**, filha de **ThreadGenerico** (linha 5). Ela tem a função de percorrer o caminho sempre esperando um determinado tempo para cada botão, dado pelo campo **passo** (linha 9). No início da declaração da classe **ThreadCorredor**, o seu construtor recebe como parâmetro o passo e um vetor de botões, que é o caminho a ser percorrido pelo corredor, e repassa ambos os parâmetros para o construtor da superclasse (linha 10).

Na linha 13 é declarado o método **setThreadSincronizador()**. Este tem a função de definir o *thread* que espera todos terminarem para reiniciar o percurso na GUI. Na linha 17, inicia-se a declaração do método **run()**, que representa o código a ser executado pelo *thread*. Primeiramente, o *thread* usa **acquire()** para adquirir o *lock* (bloqueio) do seu próprio semáforo para indicar que ele está executando a sua sessão crítica (linhas 20 a 24). Em seguida, o *thread* marca todos os botões do seu caminho como aguardando (linhas 26 a 28) para serem percorridos.

Agora é a hora do *thread* percorrer o caminho, sendo usado um laço **for** para percorrer os botões (linha 31). Para cada botão, a aplicação espera o tempo do **passo** (linha 32) e verifica se o passo correto corresponde a um botão com pausa (em vermelho), conforme a linha 33. Se sim, o processamento é pausado usando o método **pausar()** (linha 34), de forma que o processamento será retomado somente quando o usuário desmarcar esse botão vermelho, liberando o percurso e, mais tarde, o botão é percorrido e consequentemente marcado com azul (linha 37).

Depois disso, o *thread* corredor termina o seu percurso e tenta adquirir o *lock* do semáforo do *thread* sincronizador (linhas 43 a 47), ou seja, ele espera pelo *thread* sincronizador terminar o seu percurso (sessão crítica), pois todos os *threads* corredores devem terminar suas sessões críticas e aguardar o último *thread* completar o percurso para iniciar um novo ciclo. Uma vez que ele obtenha o *lock* do sincronizador, ele libera o *lock* do seu próprio semáforo (linha 49) e libera o *lock* do *thread* sincronizador para que outros *threads* corredores possam obter o *lock* dele (linha 50), completando suas sessões críticas. Por fim, ele pausa a si mesmo até que o *thread* sincronizador o desbloqueie (linha 51).

Quando o *thread* corredor acordar, ele vai iniciar uma nova iteração, pois esse processamento está dentro de um laço infinito (linha 18), repetindo tudo novamente. O *thread* sincronizador (classe **ThreadSincroniza**) é semelhante, mas contém uma implementação de sincronização diferente. Ele é detalhado na **Listagem 20**, sendo filho de **ThreadGenerico** (linha 5), recebendo no construtor o seu vetor de botões para percorrer, o passo para esperar em cada botão percorrido e o vetor dos outros *threads*, pois ele precisará acordá-los (linha 9) quando completar o seu próprio percurso e eles tiverem terminado os deles.

Listagem 19. Código da classe ThreadCorredor.

```
01. package pkg;
02.
03. import java.awt.Button;
04.
05. public class ThreadCorredor extends ThreadGenerico {
06.
07.     private ThreadSincroniza threadSincronizador;
08.
09.     public ThreadCorredor(Button[] botoes, int passo) {
10.         super(botoes, passo);
11.     }
12.
13.     public void setThreadSincronizador(ThreadSincroniza threadSincronizador) {
14.         this.threadSincronizador = threadSincronizador;
15.     }
16.
17.     public void run() {
18.         while (true) {
19.
20.             try {
21.                 semaforo.acquire();
22.             } catch (Exception e) {
23.                 e.printStackTrace();
24.             }
25.
26.             for (Button botao : botoes) {
27.                 botao.setBackground(AGUARDANDO);
28.             }
29.
30.             try {
31.                 for (int botaoAtual = 0; botaoAtual < botoes.length; botaoAtual++) {
32.                     sleep(passo);
33.                     if (botaoAtual == numeroBotao) {
34.                         pausar();
35.                         sleep(passo);
36.                     }
37.                     botoes[botaoAtual].setBackground(PERCORRIDO);
38.                 }
39.             } catch (Exception e) {
40.                 e.printStackTrace();
41.             }
42.
43.             try {
44.                 threadSincronizador.getSemáforo().acquire();
45.             } catch (Exception e) {
46.                 e.printStackTrace();
47.             }
48.
49.             semaforo.release();
50.             threadSincronizador.getSemáforo().release();
51.             pausar();
52.         }
53.     }
54. }
```

Logo no início do seu processamento, o *thread* sincronizador adquire o *lock* do seu próprio semáforo (linhas 17 a 21), indicando que ele está em sua sessão crítica, que primeiramente, limpa a tela, deixando todos os botões com a cor verde (linhas 23 a 25) e em seguida, realiza a caminhada exatamente da mesma forma que o *thread* corredor (linhas 27 a 38). Na linha 40 o *thread* sincronizador libera o seu próprio semáforo indicando que terminou a corrida, deixando o seu semáforo disponível para os *threads* corredores.

O *thread* sincronizador também deve esperar que todos os corredores terminem antes de sinalizar o reinício da corrida para todos os *threads*. Portanto, ele adquire o *lock* de cada *thread* corredor (linhas 42 a 49), mas se algum *thread* corredor não tiver terminado

a corrida, ele fica esperando até o thread pendente completar o percurso, e logo após, libera o *lock*. Depois disso, quando chegar na linha 51, todos os *threads* corredores estão no último passo, dormindo, e então é a hora certa do thread sincronizador acordar cada um deles (linhas 51 a 55), usando **notify()**. Agora, a iteração corrente está terminada e é iniciada uma nova, ou seja, todos os *threads* iniciam uma nova corrida, dando continuidade ao laço infinito do método **run()**.

Listagem 20. Código da classe ThreadSincroniza.

```

01. package pkg;
02.
03. import java.awt.Button;
04.
05. public class ThreadSincroniza extends ThreadGenerico {
06.
07.     private ThreadCorredor threads[];
08.
09.     public ThreadSincroniza(Button[] botoes, int passo, ThreadCorredor[] threads) {
10.         super(botoes, passo);
11.         this.threads = threads;
12.     }
13.
14.     public void run() {
15.         while (true) {
16.
17.             try {
18.                 semaforo.acquire();
19.             } catch (Exception e) {
20.                 e.printStackTrace();
21.             }
22.
23.             for (Button botao : botoes) {
24.                 botao.setBackground(AGUARDANDO);
25.             }
26.
27.             try {
28.                 for (int botaoAtual = 0; botaoAtual < botoes.length; botaoAtual++) {
29.                     sleep(passo);
30.                     if (botaoAtual == numeroBotao) {
31.                         pausar();
32.                         sleep(passo);
33.                     }
34.                     botoes[botaoAtual].setBackground(PERCORRIDO);
35.                 }
36.             } catch (Exception e) {
37.                 e.printStackTrace();
38.             }
39.
40.             semaforo.release();
41.
42.             for (ThreadCorredor thread : threads) {
43.                 try {
44.                     thread.getSemáforo().acquire();
45.                 } catch (Exception e) {
46.                     e.printStackTrace();
47.                 }
48.                 thread.getSemáforo().release();
49.             }
50.
51.             for (ThreadCorredor thread : threads) {
52.                 synchronized (thread) {
53.                     thread.notify();
54.                 }
55.             }
56.         }
57.     }
58. }
```

A linguagem Java oferece uma grande vantagem sobre muitas outras linguagens no processamento concorrente porque ela tem a habilidade de esconder a complexidade de tais operações, permitindo ao programador usar mecanismos de alto nível para criação e controle de *threads*. Java disponibiliza classes, interfaces e comandos simplificados, mas poderosos para realizar a sincronização de acesso a sessões críticas, além de ter classes auxiliares desde o Java 5 para facilitar a programação concorrente, melhorando a legibilidade e usabilidade do código.

Mesmo com esses sofisticados recursos, existem muitos casos onde programadores desenvolvem aplicações *multi-thread* mas não usam mecanismos adequados de sincronização ou mesmo não há qualquer controle de sincronização de *threads*, acarretando em erros de processamento que são muito difíceis de se determinar a causa, e que levam a falhas graves, como resultados com valores errados, violação de chaves primárias no banco de dados, perda de dados e exceções inesperadas. Tais erros podem acontecer com frequência ou muito esporadicamente, como uma vez por ano.

Portanto, é de grande importância o conhecimento e a correta utilização dos mecanismos de sincronização de *threads* no desenvolvimento de código *multi-thread*, além da criação de planos de testes para a aplicação em desenvolvimento. Como a linguagem Java garante o funcionamento correto dos seus recursos de sincronização, a questão torna-se apenas de empregá-los adequadamente, como na aplicação exemplo, que mostra o funcionamento correto da sincronização entre vários *threads* para chegar a um objetivo comum.

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/>

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

Twitter4J API: Conhecendo a API

Aprendendo a utilizar a API de acesso a dados do Twitter para automatizar ações e extrair dados relevantes

Com a popularização das redes sociais, sobretudo a rede social Twitter, determinados segmentos de negócio ou mesmo outros segmentos de interesses específicos têm a necessidade de acompanhar de uma maneira mais eficiente os principais assuntos e tendências, baseado no comportamento dos milhares de usuários da rede social que a utilizam diariamente pelo mundo inteiro.

Não é novidade que este tipo de informação seja aplicado em vários setores empresariais. Exemplos típicos vêm, na maioria das vezes, de departamentos de pesquisa de mercado e atendimento ao cliente. Estes departamentos recorrem às redes sociais para descobrir demandas por novos produtos, novos públicos-alvo e também para obter feedbacks dos produtos já lançados e da qualidade dos serviços prestados. Essas informações podem compor um termômetro de uma marca ou mesmo de um determinado departamento (atendimento ao cliente, serviços, lojas, etc.) em uma empresa, ou seja, a coleção de mensagens trocadas nas redes sociais, sejam positivas ou negativas, podem guiar ou fornecer informações importantes para tomada de decisões e aprimoramento nos processos empresariais.

Outro ponto importante é usufruir da interação com os usuários e poder transformar uma reclamação em um elogio, após um problema ser resolvido. Com o notável número de dados e a precisão que alguns deles apresentam, é possível ainda descobrir informações novas com a aplicação de análises mais complexas.

No passado, a coleta de informações era mais complicada e demandava custosas minerações de dados, assim como bastante tempo. Às vezes, devido a essa lentidão ocasionada no processamento dos dados, quando as informações chegavam até a cadeia de tomada de decisão, eram simplesmente descartadas, pois já não eram mais

Fique por dentro

Hoje em dia, com o grande número de dados disponíveis, sobretudo nas redes sociais, extrair informações úteis e interagir de forma ágil para a tomada de decisões representam um grande desafio. Neste contexto, o desenvolvimento de sistemas que possibilitam obter esses dados, correlacioná-los e transformá-los em informações úteis é de grande valor para diversos ramos de negócio, especialmente para os que demandam mais contato com o cliente. Diante desse desafio, este artigo propõe o uso da API Twitter4J para viabilizar a integração com o Twitter e assim explorar suas diversas funcionalidades objetivando a coleta, busca e análise de dados, como também formas de interação automatizadas cada vez mais eficazes.

válidas devido à velocidade e o dinamismo com que as coisas acontecem.

Diante da abundância dos dados, o desafio de compreendê-los e, principalmente, convertê-los em informações úteis ou conhecimento, pode fazer a diferença na tomada de decisões importantes. Por conta disto, o uso de aplicações de descoberta e mineração de informações, ou seja, aplicações que correlacionam dados, que buscam por conexões em diferentes fontes de dados, é considerado como uma verdadeira necessidade nos dias atuais.

Para entendermos melhor como podemos obter, a partir do Big Data, informações de uma rede social e, por meio dela, identificar certos padrões, vamos tomar como exemplo uma empresa hipotética do setor de serviços. Nesta empresa, os setores de qualidade de serviço e atendimento ao consumidor monitoram as perguntas e feedbacks de cada um dos clientes por meio das postagens nas redes sociais.

Tais postagens não representam apenas dúvidas e soluções de problemas, elas podem contribuir também para conhecer e acompanhar a reputação da empresa. A quantidade de comentários, citações e respostas, podem formar um padrão específico que pode indicar alguma informação nova que talvez não tenha sido

descoberto pela empresa, como uma boa aceitação de um produto recém-lançado ou alguma referência negativa; por exemplo, um atendimento ruim ou produto com defeito. O refinamento destas informações pode ser feito com algoritmos específicos de reconhecimento de padrões e inteligência artificial com o objetivo de aprimorar a leitura das informações e classificá-las como neutras, negativas ou positivas, como uma forma de traduzir todos os dados extraídos das redes sociais em uma informação útil e relevante.

A correlação dos dados das redes sociais com os dados internos da empresa para a descoberta de padrões é essencial para torná-la cada vez mais ágil na tomada de decisões e assim evitar que um problema seja intensificado ou até mesmo aproveitar uma situação positiva, como um sucesso maior do que o esperado. Um exemplo deste tipo de correlação é acompanhar comentários, hashtags ou qualquer outro tipo de dado durante o lançamento de um produto. Adicionalmente, podemos também utilizar os dados de localização associados a essas mensagens de feedback, pois podem gerar subsídios para confirmar se a estratégia de lançamento está sendo bem conduzida, isto é, se está atingindo as metas planejadas, ou ainda tomar ações de ajuste até que se atinja o objetivo proposto.

Com base neste simples exemplo, vamos desenvolver uma aplicação cujo propósito é automatizar o acompanhamento de determinados assuntos no Twitter, utilizando para isto uma API específica e que já está se tornando bastante popular na comunidade Java, a Twitter4J. Assim como qualquer outra API, a Twitter4J possui uma série de funcionalidades úteis e simples de se utilizar, resultando em uma implementação rápida e sem grandes dificuldades.

Deste modo, este artigo propõe a apresentação da API Twitter4J e suas principais características, em conjunto com um tutorial de configuração passo-a-passo. Será apresentado também como utilizar uma conta de desenvolvimento no Twitter Developers, necessária para a integração da biblioteca Java com a base de dados do

Twitter. Para entendermos suas principais funcionalidades, serão mostrados exemplos práticos explicados passo-a-passo. Por fim, será desenvolvido um sistema de acompanhamento de popularidade de assuntos previamente selecionados. Com base na medição de popularidade, em função da quantidade de citações das palavras ou assuntos selecionados, será possível descobrir padrões nos dados analisados.

Criando as credenciais de acesso no Twitter Developers

Antes de desenvolvermos nosso próprio sistema para utilizar os recursos que a API Twitter4J oferece, é necessário obter as credenciais de acesso para conseguir a autenticação necessária para estabelecer a comunicação entre a API e o Twitter. Estas credenciais são fornecidas no site Twitter Developers (veja o endereço na seção [Links](#)).

Devido a questões de segurança e controle, o Twitter Developers liberará uma credencial para cada aplicação ou projeto. Logo, é preciso registrar individualmente todas as aplicações, caso exista mais de uma. Esta organização viabilizada pelo Twitter é bastante útil porque possibilita separar diferentes interfaces de comunicação e cada uma delas pode ter níveis de acesso individualizados. Desta forma, aplicações que por ventura ultrapassem os limites de utilização do Twitter (na seção [Links](#) há uma referência que apresenta informações

sobre os limites atuais), por exemplo, poderão ser isoladas das demais.

Para acessar o Twitter Developers, informe os dados de entrada de uma conta normal no Twitter. Efetuado o login, no canto superior direito, clique na foto/perfil do seu usuário e selecione *My applications*, como mostra a [Figura 1](#).

Feito isso, serão mostradas todas as aplicações configuradas para acesso ao Twitter. Por meio deste gerenciamento é possível obter os seguintes dados de autenticação a serem utilizados pela API Twitter4J: *Consumer key*, *Consumer secret*, *Request token URL*, *Authorize URL*, *Access token URL*, *Access token secret*.

Nosso próximo passo será criar uma aplicação no Twitter Developers (*Create New App*) e fornecer os dados cadastrais, como nome, descrição e URL. Um detalhe importante sobre a URL é que apesar de ter seu preenchimento obrigatório, ela não é verificada para o funcionamento da API, permitindo o uso da URL do site ou serviço mesmo que este ainda não esteja on-line.

Após concluir o registro da aplicação, é preciso configurar o nível de permissão de acesso. Por exemplo, se a aplicação somente irá consultar dados, deve ter a permissão somente leitura (*Read Only*). Se necessitar criar novas atualizações de status ou *tweets*, a permissão deve ser do tipo leitura e escrita (*Read and Write*). Caso a aplicação simule todos os privilégios que um usuário regular tem, ou seja, consultar

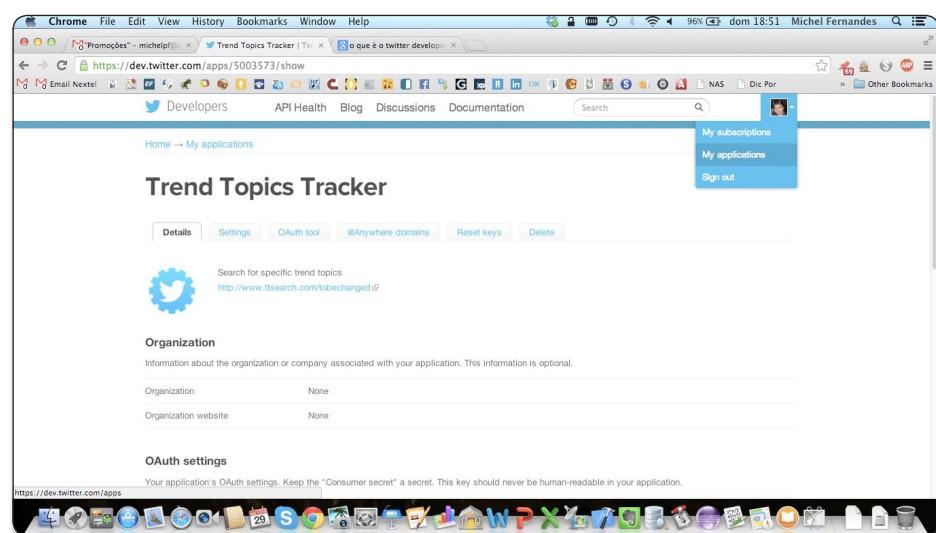


Figura 1. Acesso às aplicações do usuário

dados, enviar mensagens diretas e postar *tweets*, deverá ter acesso de leitura, escrita e acesso a mensagens diretas (*Read, Write and Access direct messages*). Para configurar o tipo de permissão apropriado, selecione o tipo adequado conforme a **Figura 2**. No nosso exemplo, vamos configurar o acesso mais abrangente, ou seja, leitura, escrita e acesso a mensagens diretas, a fim de testarmos todas as principais funcionalidades que serão explicadas posteriormente.

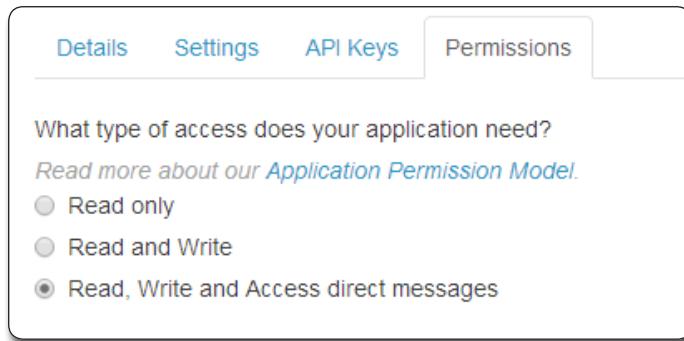


Figura 2. Configurando a permissão de acesso da aplicação

Depois de termos a aplicação devidamente cadastrada e o nível de permissão de acesso definido, o próximo passo é obter as credenciais para autenticação, necessárias para que a API Twitter4J se comunique com a base de dados do Twitter. Assim, clique na aba *API Keys*, onde serão mostradas as credenciais e os tokens de acesso da aplicação selecionada, como demonstra a **Figura 3**. Como utilizaremos estas informações no desenvolvimento da aplicação, guarde-as.

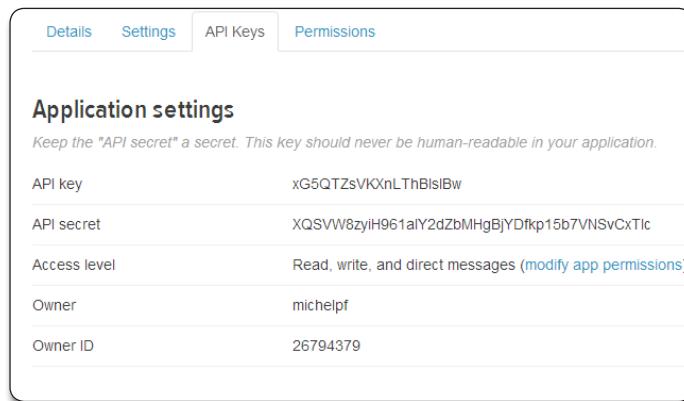


Figura 3. Informações gerais para autenticação da API

Os tokens de acesso (*Access Token*) são uma medida adicional e obrigatória de segurança. Sem eles não é possível acessar a base de dados do Twitter. Para criar os tokens de acesso, ainda em *API Keys*, clique em *Create access token* e aguarde alguns minutos até que as credenciais sejam disponibilizadas. Ao fazer isso, recomenda-se aguardar por dois minutos e depois atualizar a página para que sejam exibidos os dados de acesso, como demonstra a **Figura 4**.

Nesta mesma área também é possível cancelar (*Revoke token access*) ou criar uma nova chave (*Regenerate my access token*), quando for necessário como, por exemplo, se as informações de acesso forem descobertas por terceiros.

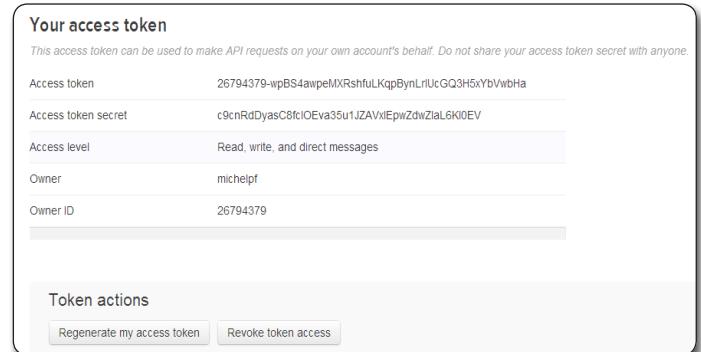


Figura 4. Token de acesso para autenticação da API

Com os dados de acesso prontos, podemos configurá-los na API Twitter4J e finalmente começar o desenvolvimento de uma aplicação que acessa os dados do Twitter.

Instalando as bibliotecas do Twitter4J

A instalação da API Twitter4J, como qualquer outra instalação de API, é bastante simples. Primeiramente, devemos acessar o site do Twitter4J (veja o endereço na seção *Links*) e baixar a última versão disponível na seção *Downloads*.

Após descompactar o arquivo baixado, você encontrará uma série de arquivos. Deste conjunto, vamos utilizar apenas os seguintes JARs presentes na pasta *lib*: *twitter4j-async-3.0.5.jar*, *twitter4j-core-3.0.5.jar*, *twitter4j-media-support-3.0.5.jar* e *twitter4j-stream-3.0.5.jar*.

Neste ponto já temos todos os requisitos básicos para iniciar o desenvolvimento da aplicação. Então, abra a IDE Eclipse e crie um novo projeto Java (*New Java Project*). Em seguida, precisamos importar os arquivos binários citados anteriormente. Para isso, acesse o menu *Project > Properties*. Nas propriedades do projeto, temos que configurar as bibliotecas no Build Path. Assim, acesse a opção *Java Build Path*, clique em *Add External JARs...* (ver **Figura 5**) e depois selecione os arquivos da API.

Apesar não haver regras sobre em qual pasta manter as bibliotecas da API, recomendamos que estas sejam adicionadas na mesma pasta do projeto no Workspace. Assim, caso o projeto seja copiado para outro lugar, as bibliotecas irão junto, não sendo necessário baixá-las novamente.

Com as bibliotecas importadas, podemos iniciar nossa primeira aplicação. Neste exemplo, vamos obter os *tweets* (e seus autores) da *timeline* de um usuário do Twitter.

Dito isso, vamos efetivamente iniciar o desenvolvimento da aplicação. A primeira parte a ser implementada, antes de iniciar qualquer tipo de interação com o Twitter, é a da autenticação de acesso. Para isso, devemos utilizar as credenciais que configuramos anteriormente, que são: *Consumer Key*, *Consumer Secret*, *Access*

Twitter4J API: Conhecendo a API

Token e Token Secret. Essas credenciais são definidas, conforme a **Listagem 1**, no objeto **twitter**, por meio dos métodos **setOAuthAccessToken(AccessToken)**, para os tokens de acesso, e **setOAuthConsumer()**, para *Consumer Key* e *Secret Key*. Com isso teremos as informações de acesso devidamente ajustadas, de acordo com o nível de privilégios concedido previamente.

A próxima etapa será carregar todas as atualizações de status ou *tweets* da *timeline* em uma coleção de valores do tipo **List**. Para isso, chamamos o método **twitter.getHomeTimeline()**, conforme a linha 18, e armazenamos o resultado na coleção **statuses**. Em seguida, para percorrer esta coleção e ler todos os tweets, foi utilizada a estrutura **foreach**. Deste modo foram obtidos os dados de cada *tweet*, como o nome do usuário (**status.getUser().getName()**) e a mensagem associada (**status.getText()**).

Após terminar o desenvolvimento da aplicação, revise as importações utilizadas no código fonte para evitar importações de classes indevidas. Verifique que na **Listagem 1** são apresentadas as importações de todas as classes utilizadas com o intuito de evitar qualquer associação indevida com outras classes de mesmo nome.

Na **Listagem 2** podemos verificar o resultado da execução da aplicação, onde são mostradas todas as atualizações da *timeline* com o nome do usuário e o *tweet* correspondente. Apesar de neste exemplo apresentarmos apenas estes dois dados, no tópico “Obtendo informações adicionais em cada Tweet” exploraremos outras características associadas, como a contagem de favoritos, *retweets* e geolocalização.

Um ponto interessante da comunicação da API com o Twitter é que sempre que houver algum problema ou erro em alguma requisição, são enviados links informativos de auxílio. Assim, problemas como falha na autenticação ou mesmo comandos indevidos enviados pela API podem ser facilmente identificados. Um exemplo destas mensagens pode ser verificado na **Listagem 3**, que mostra um erro de autenticação e ainda indica, além de causas prováveis, onde procurar ajuda nos links de discussões para solucionar o problema.

Explorando os principais recursos da Twitter4J

Com a API Twitter4J é possível realizar qualquer tipo de operação que poderia ser efetuada manualmente pelo usuário no site do Twitter, dentre elas: criar *tweets*, buscar por assuntos com critérios específicos, enviar e visualizar mensagens diretas, entre outras funcionalidades. Sendo assim, nos próximos tópicos apresentaremos exemplos de aplicações para explorar cada uma delas.

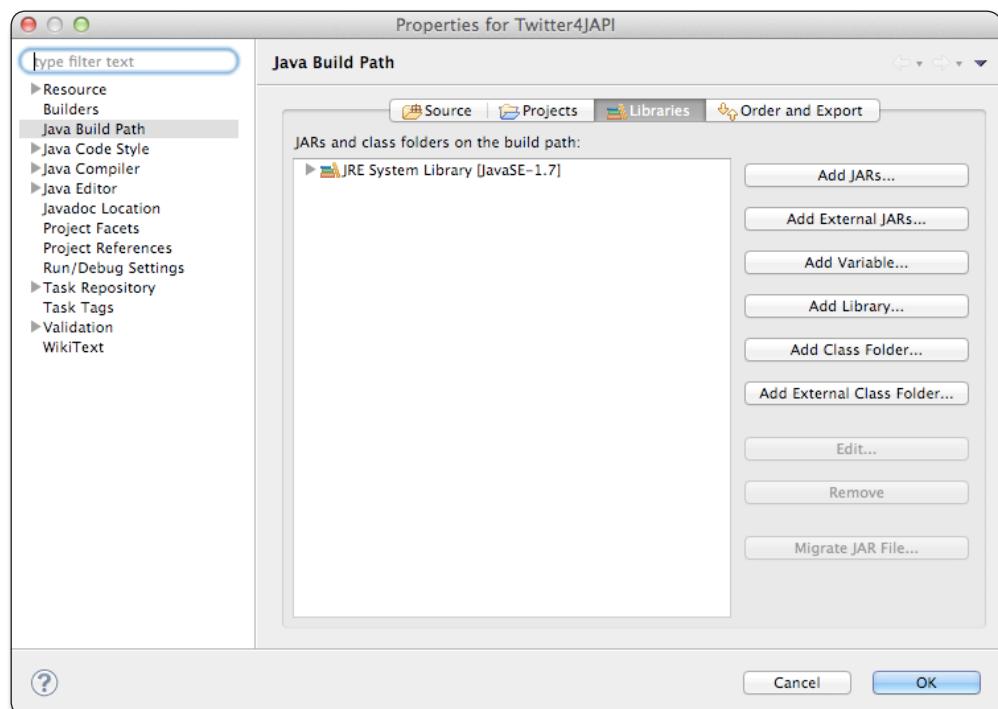


Figura 5. Adicionando as bibliotecas ao projeto

Listagem 1. Acessando os tweets da Timeline - Adaptado da documentação do Twitter4J.

```
01 import java.util.List;
02 import twitter4j.Status;
03 import twitter4j.Twitter;
04 import twitter4j.TwitterFactory;
05 import twitter4j.auth.AccessToken;
06
07 public class Main {
08
09     public static void main(String[] args) {
10
11         try {
12             TwitterFactory factory = new TwitterFactory();
13             AccessToken accessToken = loadAccessToken();
14             Twitter twitter = factory.getSingleton();
15             twitter.setOAuthConsumer("xG5QTZsVKXnLThBlsIBw",
16                                     "XQSVW8ziH961alY2dZbMHgBjYDfkp15b7VNsCxTlc");
17             twitter.setOAuthAccessToken(accessToken);
18
19             List<Status> statuses = twitter.getHomeTimeline();
20             System.out.println("Home Timeline:");
21             for (Status status : statuses) {
22                 System.out.println("Usuário: " + status.getUser().getName() + ":" +
23                                   "Tweet:" + status.getText());
24             }
25         } catch (Exception e) {
26             e.printStackTrace();
27         }
28
29         private static AccessToken loadAccessToken(){
30             String token = "26794379-EShOPlvlgLoIChdYO1AkuwEfri5PVfbzWXa591zY";
31             String tokenSecret = "NghghJuTEAOUAXWBNSGlshcGOXtoc78JlRKEjGVmUOjJ";
32             return new AccessToken(token, tokenSecret);
33         }
34     }
```

Listagem 2. Registros da Timeline.

Home Timeline:

...
Usuário: Vagas.com.br:Tweet:Vagas p/ Assistente de Suporte - TI - FH Consulting - Curitiba/PR - http://t.co/Gqf47qDPqs
Usuário: Marlos Vidal:Tweet:Nissan anuncia fábrica de motores em Resende (RJ): Em visita às obras de fábrica da Nissan em Resende (RJ), Ca... http://t.co/eoKtYE173e
Usuário: VAGAS.com.br:Tweet:Vagas p/ Gerente Administrativo - RGB Consultoria em RH - Itapetininga/SP - http://t.co/ZFZvlpVtLw
Usuário: Meio Bit:Tweet:CES 2014: Lenovo anuncia novos AIO com Android, ultrabook e tablet com Windows 8 Pro http://t.co/A1kbP3mVa1 #HojenoMeioBit
...

Listagem 3. Mensagem de retorno do Twitter para erro de autenticação.

400:The request was invalid. An accompanying error message will explain why.
This is the status code will be returned during version 1.0 rate limiting(https://dev.twitter.com/pages/rate-limiting). In API v1.1, a request without authentication is considered invalid and you will get this response.
message - Bad Authentication data
code - 215

Relevant discussions can be found on the Internet at:

<http://www.google.co.jp/search?q=d35baff5> or
<http://www.google.co.jp/search?q=1446301f>

TwitterException{exceptionCode=[d35baff5-1446301f], statusCode=400, message=Bad Authentication data, code=215, retryAfter=-1, rateLimitStatus=null, version=3.0.5}

Estas funcionalidades permitirão automatizar diversas operações e tornar mais ágil a comunicação com os usuários do Twitter. Por exemplo, é possível desenvolver um sistema que identifique mensagens de reclamação, por meio da busca de temas específicos, e que automaticamente responda estas mensagens solicitando mais informações ou fornecendo algum tipo de feedback. Também é possível que seja estabelecido uma interface com os sistemas próprios da empresa para facilitar e tornar ainda mais ágil este tipo de interação.

Criando um Tweet

Como um dos principais recursos, a Twitter4J pode ser utilizada para automatizar a criação de novos *tweets*. Com isso é viabilizado a uma empresa sinalizar ou comunicar na rede social sobre determinada situação, como um problema, promoção ou um simples aviso. Na **Listagem 4** segue um exemplo de como postar um novo *tweet* utilizando a API.

Um detalhe importante é que as permissões devem estar adequadas para este tipo de funcionalidade, ou seja, a aplicação deve ter, pelo menos, o acesso de escrita (*Write*), conforme configuramos no Twitter Developers anteriormente.

O código apresentado segue o mesmo padrão de todas as aplicações que analisaremos neste artigo. Inicialmente são criados os objetos de autenticação com o Twitter, para prover o acesso. Estes objetos e toda esta parte de configuração é realizado nas linhas 10 a 14. Note que na linha 12 chamamos o método *getSingleton()* para que seja criada apenas uma instância do objeto *twitter* na aplicação. Com isso deixamos de criar múltiplas conexões toda vez que for utilizada alguma operação da API, pois o padrão *Singleton* permite estabelecer apenas uma conexão independente das operações que sejam executadas, mesmo sendo em um ambiente *multithread* (aplicações com processamento paralelo).

Após a autenticação, a etapa seguinte é executar o método *twitter.updateStatus()* para postar um novo *tweet* (vide linha 16). Nesta mesma linha, o retorno da execução do método será armazenado na variável *status*, do tipo *Status*. Este tipo de variável reúne todas as informações associadas ao *tweet*, como o idioma, a localização geográfica, o número de *retweets*, etc. Tais informações enriquecem o conteúdo do *tweet*, pois uma mensagem pode ter um significado amplificado a depender da localização e da popularidade. Por exemplo, mensagens com um grande número de citações e grande popularidade podem indicar uma informação relevante.

Podemos verificar o resultado da execução da aplicação na **Listagem 5**, que indica que o *tweet* foi publicado com sucesso. Ao visitar a conta do respectivo usuário no Twitter (veja a **Figura 6**), temos a comprovação de que o *tweet* foi postado.

Listagem 4. Código para postar um tweet.

```
01 import twitter4j.Status;
02 import twitter4j.Twitter;
03 import twitter4j.TwitterFactory;
04 import twitter4j.auth.AccessToken;
05
06 public class Main {
07
08     public static void main(String[] args) {
09         try {
10             TwitterFactory factory = new TwitterFactory();
11             AccessToken accessToken = loadAccessToken();
12             Twitter twitter = factory.getSingleton();
13             twitter.setOAuthConsumer("xG5QTZsVKXnLThBlsIBw",
14                                     "XQSVW8zyiH961alY2dZbMHgBjYDfkp15b7VNSvCxTlc");
15             twitter.setOAuthAccessToken(accessToken);
16
17             Status status = twitter.updateStatus("Teste de Atualização com Twitter4J");
18             System.out.println("Tweet postado com sucesso! [" + status.getText() + "]");
19         } catch (Exception e) {
20             e.printStackTrace();
21         }
22
23     private static AccessToken loadAccessToken(){
24         String token = "26794379-wpBS4awpeMXRshfULKapBynLlUcGQ3H5xYbVwbHa";
25         String tokenSecret = "c9cnRdDyasC8fcI0Eva35u1JZA VxIEpwZdwZlaL6Kl0EV";
26         return new AccessToken(token, tokenSecret);
27     }
28 }
```

Listagem 5. Resultado da execução da aplicação.

Tweet postado com sucesso! [Teste de Atualização com Twitter4J].



Figura 6. Tweet enviado pela API Twitter4J

Twitter4J API: Conhecendo a API

Enviando uma mensagem direta

O envio de uma mensagem direta (*Direct Message*, ou DM) é bem similar a postar um novo *tweet*. A diferença é que o *tweet* é público e a mensagem direta é privada. De modo semelhante ao que fizemos anteriormente, vamos desenvolver uma aplicação para automatizar o envio deste tipo de mensagem.

Vale lembrar que para enviar uma mensagem direta é preciso que o usuário de destino da mensagem seja seguidor do usuário remetente. Devido a isso, para testar essa funcionalidade, devemos escolher um usuário que seja seguidor do usuário que foi utilizado no registro da aplicação no Twitter Developers.

Nota

Como limitação da API, para fazer uso de todos os recursos aqui apresentados, deve-se utilizar apenas os dados do usuário cadastrado durante o registro da aplicação no Twitter Developers.

Para enviar uma mensagem direta, utilizamos o método `twitter.sendDirectMessage()` na linha 16, da **Listagem 6**. Similarmente ao exemplo de postar um novo *tweet*, também utilizamos uma variável que armazenará o retorno do envio da mensagem direta. Essa variável, `message`, é do tipo `DirectMessage`, e nela é possível obter informações básicas acerca do envio da mensagem, como a própria mensagem e a data de envio.

Como resultado, podemos verificar na **Listagem 7** que a mensagem direta foi enviada com sucesso.

Da mesma forma, também podemos acessar o site do Twitter e comprovar, de acordo com a **Figura 7**, que a ela foi enviada como esperado.

Listagem 6. Código para postar uma mensagem direta.

```
01 import twitter4j.DirectMessage;
02 import twitter4j.Twitter;
03 import twitter4j.TwitterFactory;
04 import twitter4j.auth.AccessToken;
05
06 public class Main {
07
08     public static void main(String[] args) {
09
10         try {
11             TwitterFactory factory = new TwitterFactory();
12             AccessToken accessToken = loadAccessToken();
13             Twitter twitter = factory.getSingleton();
14             twitter.setOAuthConsumer("xG5QTZsVKXnLThBlsBw",
15                                     "XQSVW8zylH961alY2dZbMHgBjYDfkp15b7VNSvCxTlc");
16             twitter.setOAuthAccessToken(accessToken);
17             DirectMessage message = twitter.sendDirectMessage
18                     ("@JNarezzi", "Teste de DM do artigo do Twitter4J.");
19             System.out.println("Sent: " + message.getText() + " to @" +
20                               message.getRecipientScreenName());
21         } catch (Exception e) {
22             e.printStackTrace();
23         }
24
25     private static AccessToken loadAccessToken(){
26         String token ="26794379-wpBS4awpeMRshfuLKqpBynLrlUcGQ3H5xYbVwbHa";
27         String tokenSecret ="c9cnRdDyasC8fcIOEva35u1JZAVxlEpwZdwZlaL6KI0EV";
28         return new AccessToken(token, tokenSecret);
29     }
30 }
```

Listagem 7. Resultado do envio de uma mensagem direta.

Sent: Teste de DM do artigo do Twitter4J. to @JNarezzi

Conhecimento faz diferença!

The advertisement highlights the magazine's focus on agile project management, software engineering, and testing. It emphasizes the availability of over 290 videos for subscribers.

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

Faça um upgrade em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. Assine Já!



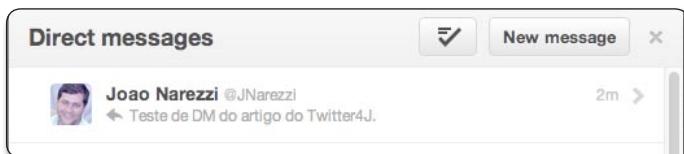


Figura 7. Mensagem Direta enviada pela Twitter4J

Visualizando as mensagens diretas

Por sua vez, para visualizar as mensagens diretas recebidas, precisamos utilizar o método `twitter.getDirectMessages()` e armazenar seu retorno na coleção `mensagens`, do tipo `List`, que é a listagem de todas as mensagens diretas recebidas pelo usuário utilizado na API. Veja a [Listagem 8](#).

Para obter informações detalhadas do remetente e do destinatário da mensagem direta, utilize os métodos `mensagem.getSender()` e `mensagem.getRecipient()`, ambos do objeto `mensagem`, do tipo `DirectMessage`. Por meio destes métodos é possível verificar a localização geográfica (`mensagem.getSender().getLocation()`), o número de seguidores (`mensagem.getSender().getFollowersCount()`), entre outras informações.

Na listagem também é mostrada a aplicação para visualizar as mensagens diretas. Como podemos verificar, essa implementação também é simples. Assim como fizemos anteriormente, carregamos todas as mensagens em uma coleção do tipo `List`, conforme a linha 18. Depois, listamos cada uma delas exibindo a mensagem, o usuário remetente e a data enviada, utilizando a estrutura `foreach`, da linha 19 à linha 24.

Como resultado, obtemos a [Listagem 9](#), onde são mostradas todas as mensagens diretas recebidas pelo usuário configurado na API, com as informações do usuário que enviou, a mensagem e a data de envio. Por questões de privacidade, foram removidos o conteúdo e o remetente das mensagens.

Buscando por assuntos e por temas específicos

Esta é a parte mais interessante e o foco principal deste artigo. Tal importância se deve ao fato que a busca e a descoberta de informações úteis em um universo de dados tão extenso são cada vez mais complicadas, frente à quantidade/diversidade dos dados. Deste modo, este cenário necessita de meios cada vez mais aperfeiçoados de pesquisa para viabilizar a extração de informações relevantes.

Uma estratégia para descobrir novas informações sobre uma determinada área pode ser feita pela monitoração de palavras chave associadas. Por exemplo, se o objetivo é monitorar e descobrir novas informações sobre as empresas fabricantes de veículos, as palavras chave poderiam ser o nome de cada uma destas empresas. A monitoração destas palavras, associada à popularidade de cada uma, isto é, a quantidade de *tweets* e *retweets*, pode indicar uma nova informação ao comparar estes dados com a base histórica em períodos anteriores.

Devido às limitações da API do Twitter, realizar uma busca sem definir critérios ou filtros específicos, como data inicial e final, limitam os resultados em até 15 *tweets* por pesquisa. Para

não termos o resultado das buscas limitada, recomendamos que sempre sejam utilizados filtros. Se o objetivo for monitorar determinados assuntos diariamente, podemos, por exemplo, limitar as datas de início e fim.

Listagem 8. Aplicação para visualizar mensagens diretas.

```
01 import java.util.List;
02 import twitter4j.DirectMessage;
03 import twitter4j.Twitter;
04 import twitter4j.TwitterFactory;
05 import twitter4j.auth.AccessToken;
06
07 public class Main {
08
09     public static void main(String[] args) {
10
11         try {
12             TwitterFactory factory = new TwitterFactory();
13             AccessToken accessToken = loadAccessToken();
14             Twitter twitter = factory.getSingleton();
15             twitter.setOAuthConsumer("xG5QTZsVKXnLThBslBw",
16                                     "XQSVW8zyiH961alY2dZbMHgBjYDfkp15b7VNSvCxTlc");
17             twitter.setOAuthAccessToken(accessToken);
18
19             List<DirectMessage> mensagens = twitter.getDirectMessages();
20             for (DirectMessage mensagem : mensagens) {
21                 System.out.println("Mensagem: " + mensagem.getText());
22                 System.out.println("Remetente: " + mensagem.getSenderScreenName());
23                 System.out.println("Data: " + mensagem.getCreatedAt());
24             }
25         } catch (Exception e) {
26             e.printStackTrace();
27         }
28     }
29
30     private static AccessToken loadAccessToken(){
31         String token = "26794379-wpBS4awpeMXRshfULKqpBynLrIucGQ3H5xYbVwbHa";
32         String tokenSecret = "c9cnRdDyasC8fcIOEva35u1JZA VxIEpwZdwZlaL6Kl0EV";
33         return new AccessToken(token, tokenSecret);
34     }
35 }
```

Listagem 9. Resultado da listagem de mensagens diretas recebidas.

```
...
Remetente: XXXXXXXX
Data: Mon Feb 25 07:36:37 BRT 2013
Mensagem: AAAAAAAAAAAA
Remetente: YYYYYYYY
Data: Fri Feb 22 18:43:13 BRT 2013
Mensagem: BBBBBBBBBB
Remetente: UUUUUUUUUU
Data: Wed Feb 06 19:59:31 BRST 2013
Mensagem: ZZZZZZZZZZ
Remetente: CCCCCCCCCC
Data: Tue Feb 05 20:03:43 BRST 2013
...
```

Em função da grande quantidade de dados, como também para melhorar a precisão dos dados que se pretende buscar, é recomendado utilizar a maior quantidade de critérios ou filtros de busca possível.

A seguir são apresentados alguns recursos que podem ser aplicados no objeto `Query` para refinar o resultado das buscas:

Twitter4J API: Conhecendo a API

- Idioma: realizar a busca pelo idioma desejado torna os resultados menos dispersos, pois se concentra em termos e palavras-chave somente no idioma selecionado. Definimos o idioma pelo método `Query.setLang(String lang)`, onde `lang` é o idioma. A relação de todos os idiomas e suas abreviações está reunida em uma página indicada na seção [Links](#);
- Tipo dos resultados: neste tipo de critério podemos filtrar por `tweets` postados recentemente, como também pelos mais populares. Esta definição é realizada pelo método `Query.setResultType(String resultType)`, onde `resultType` pode ser `Query.MIXED`, valor padrão que reúne os resultados recentes e populares, `Query.RECENT`, para filtrar apenas os resultados recentes, ou `Query.POPULAR`, que filtra apenas os resultados com maior popularidade;
- Desde uma data específica: quando é necessário buscar por algo a partir de uma determinada data. Para este filtro, chamamos o método `Query.setSince(String data)` passando a data no formato “ano/mês/dia”, por exemplo: `2014/01/14`;
- Até uma data específica: quando é necessário buscar até uma data determinada. Para este filtro, chamamos o método `Query.setUntil(String data)` passando a data no formato “ano/mês/dia”;
- Limitar por uma região em específico: quando for necessário filtrar os resultados por uma determinada região geográfica. Fornece a possibilidade de determinar um ponto geolocalizado e, a partir dele, traçar um raio e assim determinar uma região específica. Para realizar esta busca, devemos chamar o método `Query.setGeoCode(GeoLocation geoLocation, Double radius, String unit)`, onde `GeoLocation` é um objeto de geolocalização representado pelas coordenadas geográficas de latitude e longitude, `radius` define o raio de limitação da região de busca, e `unit` especifica a unidade de comprimento do raio, que pode ser `Query.MILES` para milhas e `Query.KILOMETERS` para quilômetros.

Além desses recursos, precisamos definir as palavras-chave das buscas. Tal como as buscas no Google, o Twitter disponibi-

liza certas regras para viabilizar uma consulta mais detalhada. Assim, para realizar a busca por palavra-chave, por exemplo, é necessário passá-la ao construtor da classe `Query`, como: `Query busca = new Query(String query)`, ou utilizar o método `Query.setQuery(String query)`, onde `query` é o termo de busca.

A [Tabela 1](#) apresenta os principais operadores e regras para realizar as buscas no Twitter.

Na [Listagem 10](#) apresentamos uma pesquisa realizada pela palavra-chave “java” na qual desejamos recuperar apenas os `tweets` que estejam no intervalo de 01/01/2014 a 10/01/2014.

Com esses tipos de filtros é possível verificar se houve algum aumento significativo da popularidade desta palavra-chave e posteriormente refinar a pesquisa para descobrir a origem desse aumento.

Como resultado, obtivemos na [Listagem 11](#) todos os `tweets` com a palavra-chave “java”, bem como a contagem do número de ocorrências, que foi de 90 no período especificado, levando em conta toda a base do Twitter. Esta informação indica um parâmetro de popularidade que poderia ser comparado com outras palavras-chave, como por exemplo, php, vb.net, etc. para medir a popularidade das linguagens de programação, sob o ponto de vista do Twitter.

Obtendo informações adicionais em cada Tweet

As informações associadas a cada `tweet` podem ser tão importantes quanto à própria mensagem. Diferente do que muitos possam imaginar, além da mensagem, é possível obter diversos outros dados relacionados que auxiliam a enriquecer significativamente o conteúdo da mensagem. As seguir são apresentadas as informações mais relevantes:

- Usuário: permite dar mais ênfase em um `tweet` baseado em usuários influentes ou com contas verificadas;
- Data e hora de criação: permite filtrar as informações pela data de interesse;
- Número de favoritos: `tweets` marcados como favoritos podem ter importância maior, se comparados com outros sem esse tipo de marcação. Podem sinalizar como um indicativo de popularidade;

Exemplo de Consulta	Descrição
Palavra1 Palavra2	Operador padrão, busca por tweets com a Palavra1 e a Palavra2
“Palavra1 Palavra2”	Busca por tweets com a frase “Palavra1 Palavra2”
Palavra1 OR Palavra2	Busca por tweets com a Palavra1 ou a Palavra2
Palavra1 -Palavra2	Busca por tweets que contêm a Palavra1 e que não contêm a Palavra2
#Palavra1	Busca por tweets com a hashtag Palavra1
from:Usuário1	Busca por tweets do usuário Usuário1
to:Usuário2	Busca por tweets para o Usuário2
@Usuário3	Busca por tweets com menção ao Usuário3
:)	Busca por tweets que contenham termos ou palavras com sentido positivo definido na engine de aprendizado do Twitter (por exemplo, bons reviews de um produto)
:(:	Busca por tweets que contenham termos ou palavras com sentido negativo definido na engine de aprendizado do Twitter

Tabela 1. Lista de operadores de pesquisa do Twitter (Fonte Development of Twitter Application #7 – Search)

- Geolocalização: a localização do *tweet* pode auxiliar bastante em questões de feedback regionalizados;
- Endereço físico de um determinado local: idem ao item anterior, pois permite obter feedbacks regionalizados por um determinado endereço;

Listagem 10. Buscando por palavra-chave e utilizando filtro por data.

```
import twitter4j.Query;
import twitter4j.QueryResult;
import twitter4j.Status;
import twitter4j.Twitter;
import twitter4j.TwitterFactory;
import twitter4j.auth.AccessToken;
public class Main {
    public static void main(String[] args) {

        try {
            TwitterFactory factory = new TwitterFactory();
            AccessToken accessToken = loadAccessToken();
            Twitter twitter = factory.getSingleton();
            twitter.setOAuthConsumer("xG5QTZsVKXnLTbIsIBw", "XQS
VW8zyiH961alY2dZbMHgBjYDfkp15b7VNSvCxTlc");
            twitter.setOAuthAccessToken(accessToken);

            Query query = new Query("java");
            query.setSince("2014-01-01");
            query.setUntil("2014-01-10");
            QueryResult result;
            int contador=0;

            result = twitter.search(query);

            while (result.hasNext())
            {
                query = result.nextQuery();
                for (Status status : result.getTweets()) {
                    contador++;
                    System.out.println("@ " + status.getUser().getScreenName() + ":" +
                           status.getText());
                }
            }

            result = twitter.search(query);
        }
        System.out.println("Tag java:" + contador + " tweets");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private static AccessToken loadAccessToken(){
    String token = "26794379-wpBS4awpeMXRshfuLKqpBynLrlUcGQ3H5xYbVwbHa";
    String tokenSecret = "c9cnRdDyasC8fclOEva35u1JZApxlEpwZdwZla6Kl0EV";
    return new AccessToken(token, tokenSecret);
}
```

Listagem 11. Resultado da busca por palavra-chave e filtro por data.

```
...
@mosesjones:All that shiny hype... but if you want a job? Learn #Java http://t.co/
ZNVi1xOycu
@annekathrine11:@bakerr4 brb goin to java, red box, and gettin Chinese 2 go. B
@ ur place n 5
@NevaehAdderi:Does java hasten rob tax? feasting tie-up is the limiting put to
silence: mGZIQFjd
...
Tag java:90 tweets de 2014-01-01 até 2014-01-10.
```

- Número de *retweets*: indica uma maior importância, tal como a marcação de favoritos. Um detalhe relevante é que a contagem de *retweets* pode sinalizar mensagens de maior interesse devido ao grau de popularidade associado.

Para demonstrar como obter essas informações, vamos modificar a aplicação exemplo analisada anteriormente. Essa modificação possibilitará explorar melhor cada *tweet* encontrado, exibindo as informações do usuário que criou *tweet*, a mensagem, quando foi criada, o número de favoritos, o número de *retweets*, sua geolocalização e seu lugar ou endereço.

Na **Listagem 12** verificamos que os dados associados, bem como o próprio *tweet*, ficam no mesmo objeto **Status**. Na linha 30 definimos a estrutura **foreach** para ler cada *tweet* retornado da busca e armazenado na lista **QueryResult**.

Os dados presentes no objeto **Status** são retornados pelos métodos específicos apresentados a seguir:

- Linha 33: **Status.getUser().getScreenName()**, retorna o nome do usuário;
- Linha 34: **Status.getText()**, retorna a mensagem;
- Linha 35: **Status.getCreatedAt()**, retorna a data de criação ou publicação;
- Linha 36: **Status.getFavoriteCount()**, retorna o número de vezes que o *tweet* foi marcado como favorito;
- Linha 37: **Status.getGeoLocation()**, retorna os dados de geolocalização do lugar de onde o *tweet* foi publicado, ou seja, as coordenadas de latitude e longitude;
- Linha 38: **Status.getPlace()**, retorna os dados de endereço do lugar de onde o *tweet* foi publicado;
- Linha 39: **Status.getRetweetCount()**, retorna o número de *retweets*.

Os resultados obtidos pela aplicação podem ser visualizados na **Listagem 13**. Note que em alguns *tweets* as informações de geolocalização ou localização não estão disponíveis, pois dependem do usuário ativar tais configurações em seus dispositivos.

Caso prático: acompanhando tendências de forma automatizada

Uma proposta interessante de uso das informações do Twitter é para o acompanhamento e monitoramento de determinados assuntos de forma automatizada, permitindo que essa identificação seja realizada rapidamente e, como consequência, a tomada de decisões possa ser mais eficiente.

Um exemplo simples desse tipo de monitoramento pode ser o acompanhamento da popularidade da hashtag **#selecaobrasileira**, ao pensar na realização da Copa do Mundo. A expectativa é que a popularidade desse termo aumente com a proximidade do evento.

Como demonstra a **Listagem 14**, foi desenvolvido um sistema que busca diariamente na base de dados do Twitter o número de ocorrências de um termo específico. No exemplo, definimos como critério a hashtag **#selecaobrasileira** e utilizamos os filtros de data de início e de fim para delimitar as contagens do termo diariamente.

Twitter4J API: Conhecendo a API

Listagem 12. Obtendo informações detalhadas de cada tweet.

```
01 import twitter4j.Query;
02 import twitter4j.QueryResult;
03 import twitter4j.Status;
04 import twitter4j.Twitter;
05 import twitter4j.TwitterFactory;
06 import twitter4j.auth.AccessToken;
07
08 public class Main {
09
10    public static void main(String[] args) {
11
12        try {
13            TwitterFactory factory = new TwitterFactory();
14            AccessToken accessToken = loadAccessToken();
15            Twitter twitter = factory.getSingleton();
16            twitter.setOAuthConsumer("xG5QTZsVKXnLThBlsLBw",
17                "XQSVW8zylH961aLY2dZbMHgBjYDfkp15b7VNSvCxTlc");
18            twitter.setOAuthAccessToken(accessToken);
19
20            Query query = new Query("java");
21            query.setSince("2014-01-01");
22            query.setUntil("2014-01-10");
23            QueryResult result;
24            int contador=0;
25            result = twitter.search(query);
26
27            while (result.hasNext())
28
29                query = result.nextQuery();
30
31                for (Status status : result.getTweets()) {
32                    contador++;
33
34                    System.out.println("Usuário: " + status.getUser().getScreenName());
35                    System.out.println("Mensagem: " + status.getText());
36                    System.out.println("Data de Criação: " + status.getCreatedAt());
37                    System.out.println("Número de Favoritos: " + status.getFavoriteCount());
38                    System.out.println("Geolocalização: " + status.getGeoLocation());
39                    System.out.println("Lugar: " + status.getPlace());
39                    System.out.println("Número de Retweets: " + status.getRetweetCount());
40
41                }
42            result = twitter.search(query);
43
44        } catch (Exception e) {
45            e.printStackTrace();
46        }
47    }
48
49    private static AccessToken loadAccessToken(){
50        String token = "26794379-wpBS4awpeMRshfuLKqpBynLrlUcGQ3H5xYbVwbHa";
51        String tokenSecret = "c9cnRdDyasC8fcI0Eva35u1JZAVxlEpwZdwZla6Kl0EV";
52        return new AccessToken(token, tokenSecret);
53    }
54 }
```

DÊ UM SALTO
EM CONHECIMENTO!



Acesse o maior
portal para
desenvolvedores
da América
Latina!

20
mil
posts

430
mil
cadastrados

10
milhões de
page-views
por mês

Listagem 13. Resultado da execução do programa da Listagem 12.

```
...
Usuário: r_mcnair721
Mensagem: ldc if its cold out, you can never go wrong with a Java chip frappuccino
Data de Criação: Thu Jan 09 21:59:53 BRST 2014
Número de Favoritos: 1
Geolocalização: GeoLocation{latitude=34.68504, longitude=-84.48321}
Lugar: PlaceJSONImpl{name='Ellijay', streetAddress='null', countryCode='US',
id='0bfcd4103d0e5ab6', country='United States', placeType='city', url='https://
api.twitter.com/1.1/geo/id/0_bfcfd4103d0e5ab6.json', fullName='Ellijay,
GA', boundingBoxType='Polygon', boundingBoxCoordinates=[[Ltwitter4j.
GeoLocation@2e21712e], geometryType='null', geometryCoordinates=null,
containedWithinIn=]]
Usuário: miesehtai
Mensagem: RT @jayteroris: Road to Java Jazz #10 Februari 2014 ##SRUDUK-
FOLLOW | @karmanmove @faisal_gudeg @dennyarivian http://t.co/bSXu1Za1U1
Data de Criação: Thu Jan 09 21:50:47 BRST 2014
Número de Favoritos: 0
Geolocalização: null
Lugar: null
Número de Retweets: 3
...
Tag java:90 tweets de 2014-01-01 até 2014-01-10.
```

Quando o número de ocorrências for superior a 50, todos os *tweets* serão exibidos, como uma forma de verificar os motivos de um incremento repentino no número de ocorrências.

A implementação do sistema seguiu o mesmo padrão dos exemplos anteriores. As mudanças relevantes estão relacionadas à incorporação do código para determinar as datas de início e fim (linhas 21 a 24) como critérios de busca dos *tweets*. Deste modo, utilizamos a estrutura de repetição **while** para realizar a busca em cada data, que se inicia na data definida em **gCalendarInicio** e termina na data atual, **gCalendarAgora**.

O resultado da execução desse código é apresentado na **Listagem 15**, e indica que pode ter ocorrido algum fato incomum no dia 08/05/14 relacionado com a palavra-chave **#selecaobrasileira**. Essa constatação é feita com base no aumento expressivo do número de *tweets* contendo essa *hashtag*. Tal aumento no número de *tweets* se deve ao fato que foi neste dia que ocorreu a divulgação dos atletas convocados para defender o Brasil na Copa do Mundo.

Listagem 14. Monitoramento de palavra-chave automatizado via Twitter

```
01. import java.text.SimpleDateFormat;
02. import java.util.GregorianCalendar;
03. import twitter4j.GeoLocation;
04. import twitter4j.Query;
05. import twitter4j.QueryResult;
06. import twitter4j.Status;
07. import twitter4j.Twitter;
08. import twitter4j.TwitterFactory;
09. import twitter4j.auth.AccessToken;
10.
11. public class Main {
12.
13.     public static void main(String[] args) {
14.         try {
15.             TwitterFactory factory = new TwitterFactory();
16.             AccessToken accessToken = loadAccessToken();
17.             Twitter twitter = factory.getSingleton();
18.             twitter.setOAuthConsumer("xG5QTZsVKXnLThBlsLBw","XQSVW8zyiH961aLY2d
ZbMHgBjYDfkp15b7VNSvCxTlc");
19.             twitter.setOAuthAccessToken(accessToken);
20.
21.             GregorianCalendar gCalendarInicio = new GregorianCalendar(2014, 5, 7);
22.             System.out.println(gCalendarInicio.getTime());
23.             GregorianCalendar gCalendarAgora = new GregorianCalendar();
24.             System.out.println(gCalendarAgora.getTime());
25.
26.             SimpleDateFormat formatador = new SimpleDateFormat("yyyy-MM-dd");
27.
28.             while (gCalendarInicio.before(gCalendarAgora)){
29.
30.                 Query query = new Query("#selecaobrasileira");
31.                 GregorianCalendar gCalendarAnterior = new GregorianCalendar();
32.                 gCalendarAnterior.setTime(gCalendarInicio.getTime());
33.                 gCalendarAnterior.add(GregorianCalendar.DAY_OF_MONTH, -1);
34.
35.                 query.setSince(formatador.format(gCalendarAnterior.getTime()));
36.                 query.setUntil(formatador.format(gCalendarInicio.getTime()));
```

```
37.             QueryResult result;
38.             int contador=0;
39.
40.             result = twitter.search(query);
41.
42.             while (result.hasNext()) {
43.                 query = result.nextQuery();
44.                 for (Status status : result.getTweets()) {
45.                     contador++;
46.                     if (contador>50){
47.                         System.out.println("Usuário: " + status.getUser().getScreenName());
48.                         System.out.println("Mensagem: " + status.getText());
49.                         System.out.println("Data de Criação: " + status.getCreatedAt());
50.                         System.out.println("Número de Favoritos: " + status.getFavoriteCount());
51.                         System.out.println("Geolocalização: " + status.getGeoLocation());
52.                         System.out.println("Lugar: " + status.getPlace());
53.                         System.out.println("Número de Retweets: " + status.getRetweetCount());
54.                     }
55.                 }
56.                 result = twitter.search(query);
57.             }
58.
59.             System.out.println(formatador.format(gCalendarInicio.getTime())+
": #selecaobrasileira "+contador+" tweets.");
60.             gCalendarInicio.add(GregorianCalendar.DAY_OF_MONTH, 1);
61.         }
62.     } catch (Exception e) {
63.         e.printStackTrace();
64.     }
65. }
66.
67. private static AccessToken loadAccessToken(){
68.     String token = "26794379-wpBS4awpeMXRshfuLKqpBynLrlUcGQ3H5xYbVwbHa";
69.     String tokenSecret = "c9cnRdDyasC8fcI0Eva35u1JZA VxlEpwZdwZlaL6Kl0EV";
70.     return new AccessToken(token, tokenSecret);
71. }
72. }
```

Twitter4J API: Conhecendo a API

A API Twitter4J representa um grande avanço para a construção de aplicações que se comunicam com o Twitter. Seu uso torna o desenvolvimento de aplicações desse tipo cada vez mais fácil, pois foi projetada para fornecer as funções do Twitter de forma simples, possibilitando ao desenvolvedor a preocupação apenas

com as regras do negócio, como a análise dos dados e geração de informação. Dito isso, continue explorando os recursos desta API. Certamente você conseguirá, ao menos, implementar funcionalidades curiosas para suas aplicações.

Listagem 15. Resultado do monitoramento do Twitter por #selecaobrasileira.

2014-05-07: #selecaobrasileira 45 tweets.

2014-05-08: #selecaobrasileira 330 tweets.

...

Usuário: DeOlhoJornal

Mensagem: Felipão apresenta a seleção que representará o Brasil na Copa #Copa #Mundial2014 #SeleçãoBrasileira <http://t.co/KcZ09BotH4>

Data de Criação: Wed May 07 17:14:35 BRT 2014

Número de Favoritos: 0

Geolocalização: null

Lugar: null

Número de Retweets: 0

...

2014-05-09: #selecaobrasileira 60 tweets.

...

Usuário: julien_menez

Mensagem: Confira os 23 lucky ones da #SelecaoBrasileira para a @fifaworldcup_pt <http://t.co/EBwyh8RW5T>

Data de Criação: Wed May 07 17:22:33 BRT 2014

Número de Favoritos: 0

Geolocalização: GeoLocation{latitude=-23.61093095, longitude=-46.69452058}

Lugar: PlaceJSONImpl{name='Sao Paulo', streetAddress='null', countryCode='BR', id='68e019afec7d0ba5', country='Brasil', placeType='city', url='https://api.twitter.com/1.1/geo/id/68e019afec7d0ba5.json', fullName='Sao Paulo, Sao Paulo', boundingBoxType='Polygon', boundingBoxCoordinates=[[Ltwitter4jGeoLocation@6e0e0380], geometryType='null', geometryCoordinates=null, containedWithIn=[]]}

...

2014-05-10: #selecaobrasileira 30 tweets.

2014-05-11: #selecaobrasileira 30 tweets.

Autor



Michel Pereira Fernandes

michelpf@gmail.com

É Mestre em Inteligência Artificial pelo Centro Universitário da FEI, docente dos cursos de Análise e Desenvolvimento de Sistemas e Automação Industrial na Universidade Paulista (UNIP), docente do curso de pós graduação MBA em Desenvolvimento de Soluções Corporativas Java e Soluções de Mobilidade na Faculdade de Informática e Administração Paulista (FIAP) e atuando como Coordenador de Projetos de Inovação em grandes empresas de telecomunicações como a Vivo e atualmente na Nextel. Trabalha com Java há oito anos fornecendo soluções automatizadas na área de telecomunicações.



Links:

Página da API Twitter4J.

<http://twitter4j.org/en/index.html>

Página do site Twitter Developers.

<https://dev.twitter.com/>

Informações sobre a API Twitter4J e limites definidos pelo Twitter.

<https://dev.twitter.com/docs/rate-limiting/1.1>

Listagem dos códigos dos idiomas no padrão ISO 639-1.

http://en.wikipedia.org/wiki/ISO_639-1

Material explicativo sobre os operadores de busca no Twitter.

<http://www.slideshare.net/onlyjiny/development-of-twitter-application-7-search>

Artigo da Revista Veja sobre Big Data.

<http://clippingmp.planejamento.gov.br/cadastros/noticias/2013/12/9/a-solucao-no-que-nao-se-ve>

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

ca
technologies

*De acordo com o estudo 2012 North America/Europe Service Virtualization Study, Coleman-Parkes.