



Edição 122 :: Ano X :: R\$ 14,90

 DEVMEDIA

ESPECIAL:

Primeiros passos em PaaS com Heroku

Crie, instale e escale suas aplicações web na nuvem

Serviços sem demora com o Play!

Desenvolvimento RESTful diferenciado

Hadoop: fundamentos e instalação

Saiba configurar um ambiente

para desenvolvimento BigData

JAX-RS 2.0 E EL 3.0

A evolução de importantes recursos da Java EE 7



O Cache de Segundo Nível no Hibernate Melhore o desempenho de caches no disco

Conhecendo a API de Fragmentos do Android Aprenda a reusar componentes de forma fácil

ISSN 1676836-1



VOCÊ É ASSINANTE MVP?

ENTÃO PODE TER CERTEZA:
O MERCADO
PREFERE VOCÊ!



DESENVOLVEDOR ATUALIZADO É O MAIS
VALIOSO DO MERCADO!

Muitos profissionais estão acomodados, não se atualizam, não leem livros e não vão a eventos. Acham que é a empresa que deve mantê-los atualizados. Gastam dinheiro com tudo, mas acham caro investir menos de R\$60,00 por mês em suas carreiras.

Mas você não! Você é MVP! Parabéns!

TENHA ACESSO A:

+DE 260 CURSOS ONLINE

09 REVISTAS MENSais

7.850 VÍDEO-AULAS

POR APENAS **59,90** MENSais



QUEM TEM ESTÁ TRANQUILO.



DEV MEDIA

Acesse: www.devmedia.com.br/mvp

Sumário

Artigo no estilo Solução Completa

06 – Primeiros passos em PaaS com Heroku

[Daniel Stori]

18 – Hadoop: fundamentos e instalação

[Cláudio Martins]

Conteúdo sobre Novidades

26 – Conheça os recursos da Expression Language 3.0

[Marcelo A. Cenerino]

Conteúdo sobre Novidades

33 – Construindo RESTful Web Services com JAX-RS 2.0

[Fernando Rubbo]

Conteúdo sobre Boas Práticas

43 – O Cache de Segundo Nível no Hibernate – Parte 2

[Cleber Muramoto]

Artigo no estilo Solução Completa

53 – Reusando componentes com a API de Fragmentos

[Aécio L. Vieira da Costa, Edilson M. Bizerra Junior e Luiz Artur Botelho da Silva]

Artigo no estilo Solução Completa

66 – Desenvolva web services com o Play Framework

[Marlon Silva Carvalho]



Dê seu feedback sobre esta edição!

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

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

www.devmedia.com.br/javamagazine/feedback



Ano X • Edição 122 • 2013 • ISSN 1676-8361



MVP

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

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Davi Costa (davigc_08@hotmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa Romulo Araujo

Diagramação Janete Feitosa

Distribuição

FC Comercial e Distribuidora S.A

Rua Teodoro da Silva, 907, Grajaú - RJ

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

Atendimento ao leitor

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

Publicidade

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

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

Java, o logotipo da xícara de café Java e todas as marcas e logotipos baseados em/ou referentes a Java são marcas comerciais ou marcas registradas da Sun Microsystems, Inc. nos Estados Unidos e em outros países.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique 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 Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

CURSOS ONLINE

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



CONHEÇA ALGUNS DOS CURSOS:

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

Para mais informações :

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



DEVMEDIA



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



Primeiros passos em PaaS com Heroku

Crie, instale e escale suas aplicações web na nuvem de maneira simples usando o Heroku

O advento da Cloud Computing e o aumento da quantidade de serviços que são oferecidos por plataformas na nuvem facilitaram e muito o processo de desenvolvimento e deploy de sistemas web. Atualmente o desenvolvedor tem à sua disposição diversas opções de serviços na nuvem. Estes abrangem vários tipos de aplicação, desde um simples web service até uma aplicação complexa. As ofertas se diferem também de acordo com o nível de customização permitido, sendo possível citar desde soluções de infraestrutura totalmente configuráveis até plataformas prontas para receber a aplicação.

Neste cenário, o Heroku vem ganhando mercado como um fornecedor de Plataforma como Serviço (PaaS). Este tipo de solução abstrai o desenvolvedor dos detalhes de infraestrutura ao disponibilizar containers para instalação das aplicações, facilitando a manutenção, extensão e escalabilidade, além de oferecer maior agilidade para disponibilizar uma aplicação na web, com menor custo inicial.

Com estes atrativos, soluções de PaaS têm chamado a atenção principalmente de pequenos projetos devido ao baixo custo inicial, pois a maioria dos fornecedores disponibiliza um plano gratuito que permite o deploy de aplicações, porém com limites de uso de recursos como CPU, memória, rede, etc.

Com base nisso, neste artigo iremos apresentar o Heroku, um fornecedor de PaaS que vem conquistando desenvolvedores. Vamos entender os pontos importantes da sua arquitetura e falaremos sobre boas práticas no desenvolvimento de aplicações PaaS. Para não ficar apenas na teoria, criaremos uma aplicação Java simples, sem o auxílio de plugins, e a instalaremos no Heroku, usando o Heroku Toolbelt. Para completar, criaremos uma aplicação mais sofisticada, baseada em templates, usando um plugin para o Eclipse.

Resumo DevMan

Porque este artigo é útil:

Para os desenvolvedores que querem se preocupar cada vez menos com infraestrutura e processos de deploy, concentrando-se apenas no desenvolvimento, o Heroku oferece um excelente serviço de hospedagem de aplicações com uma boa oferta de complementos como busca, cache, mensagens, etc, além de simplificar o processo de escalar a aplicação.

Pensando nisso, este artigo irá apresentar o Heroku, uma solução de Plataforma como Serviço que permite ao desenvolvedor criar e subir rapidamente suas aplicações para a nuvem. Para isso, analisaremos o Heroku Toolbelt, um poderoso cliente que permite gerenciar as aplicações pela linha de comando, e também o plugin do Eclipse, que facilita a criação e atualização das aplicações.

Plataforma como Serviço

O Heroku se enquadra na categoria de serviços da computação em nuvem conhecida como Plataforma como Serviço (Platform as a Service, ou PaaS), no qual o fornecedor entrega para o cliente um ambiente pronto para receber a aplicação. Diferente do IaaS (Infraestrutura como Serviço), no qual cliente contrata máquinas (reais ou virtuais) e é responsável pela instalação de bibliotecas, montagem das estruturas do sistema de arquivos, entre outros recursos, o PaaS é uma solução de alto nível que abstrai este tipo de preocupação.

Como o ambiente é entregue pelo fornecedor, ao cliente basta se concentrar em desenvolver e instalar a aplicação. Normalmente nos serviços PaaS a instalação ou atualização é feita através de commits em repositórios remotos vinculados à aplicação.

No mercado, atualmente, existem muitas opções de fornecedores de serviços de PaaS além do próprio Heroku. Podemos citar como exemplo o RedHat OpenShift, AWS Elastic Beanstalk, Jelastic, CloudBees, dentre outros. Basicamente o que difere cada um são as tecnologias que eles permitem utilizar, como

linguagens ou containers, quantidades de serviços adicionais e, é claro, a especificação, ou seja, o custo por memória, CPU e disco de cada instância da aplicação. Outra variável é a facilidade de criação e instalação de um aplicativo. Neste ponto o Heroku está um passo à frente dos outros fornecedores, pela simplicidade que é criar, manter e escalar uma aplicação, como veremos nos próximos tópicos.

Heroku

O Heroku foi criado em 2007 por três desenvolvedores norte-americanos: James Lindenbaum, Adam Wiggins e Orion Henry. Inicialmente o suporte era apenas para aplicativos desenvolvidos em Ruby, rodando no servidor web Rack. Em 2010 o projeto foi adquirido pela Salesforce e em 2011 iniciou um processo para suportar outras linguagens e frameworks. Atualmente o Heroku suporta Ruby, Java, Clojure, Python, Scala e Node e possui um parque de mais de três milhões de aplicações instaladas.

A respeito das instalações físicas, o Heroku utiliza a consolidada infraestrutura da Amazon e opera na US Region (que contempla datacenters nos Estados Unidos), sendo que a EU Region (datacenters localizados na Irlanda) está em fase beta. Em relação à disponibilidade, independente da região, relatórios exibidos no próprio site garantem que o Uptime nos últimos 12 meses tem sido sempre superior a 99%.

Arquitetura do Heroku – Apresentando os Dynos

Diferente de um serviço IaaS, que provê ao cliente uma máquina inteira para que ele configure os serviços e instale seu aplicativo, o Heroku, assim como os demais serviços PaaS, disponibiliza um ambiente de execução de aplicações. Este tipo de solução abstrai do cliente detalhes do sistema operacional como bibliotecas, serviços de startup, gestão de memória, sistema de arquivos, entre outros, provendo uma maneira muito mais simples e prática de subir e escalar as aplicações.

Cada fornecedor dá um nome para este tipo de ambiente de execução. Na Amazon, por exemplo, ele é chamado de instância, no OpenShift, de gear, e no Heroku, temos os **dynos**. Os dynos são definidos como containers leves que permitem que o desenvolvedor execute sua aplicação em um ambiente isolado e seguro.

As aplicações criadas para rodar no Heroku podem ser compostas por vários dynos. De acordo com a tarefa que realizam, os dynos podem ser classificados em três tipos:

- **Web Dynos:** Responsáveis por receber e atender as requisições web. Cada aplicação pode ter no máximo um dyno desse tipo;
- **Worker Dynos:** Executam tarefas em background, sendo recomendado para processamentos mais pesados. Cada aplicação pode ter vários dynos do tipo worker;
- **One-Off Dynos:** São criados apenas para uma tarefa eventual e logo são destruídos. Um exemplo clássico é a leitura de logs da aplicação.

As aplicações normalmente contam com um web dyno e opcionalmente workers dynos. Eventualmente é necessário executar

uma tarefa específica, então lançamos mão dos one-off dynos. Em relação ao porte dos dynos, temos apenas duas opções com variações de CPU e quantidade de memória. Os números estão dispostos na **Tabela 1**.

Tamanho do Dyno	Memória RAM	CPU Share
1X	512MB	1x
2X	1024MB	2x

Tabela 1. Opções de tamanho dos dynos

No Heroku, a possibilidade de escalar um dyno verticalmente está restrita à **Tabela 1**. Em geral, nas soluções PaaS, opta-se por escalar horizontalmente, e no Heroku isso é possível adicionando novas cópias (novas instâncias) de dynos de acordo com o seu tipo (Web, Worker e One-Off). Por exemplo, uma aplicação pode ter um dyno do tipo web e outro do tipo worker. Imaginando um aumento repentino na quantidade de requisições web, pode-se optar por aumentar a quantidade de cópias do dyno de tipo web, mantendo apenas uma cópia do dyno de tipo worker. No Heroku esse tipo de aumento ou redução é feito com a intervenção do administrador da aplicação.

Os dynos de todas as aplicações instaladas no Heroku são monitorados pelo **Dyno Manager** (antigamente chamado de Dyno Manifold), que é um super gerenciador que reside no core do Heroku, sendo responsável pela manutenção das aplicações no ar.

Uma das coisas mais interessantes que o Dyno Manager faz pela aplicação é mantê-la resistente à erosão (software erosion). Toda aplicação que fica determinado tempo no ar começa a sofrer com certa lentidão devido ao fenômeno conhecido como erosão do software. Às vezes o problema está no próprio aplicativo, outras vezes em bibliotecas, ou até no sistema operacional. O Dyno Manager identifica se a aplicação está apresentando lentidão baseado em critérios próprios de tempo de resposta, e reinicia a aplicação sem a intervenção do desenvolvedor. Além disso, o Dyno Manager pode identificar que a máquina física em que a aplicação está instalada está apresentando problemas. Neste caso ele pode automaticamente movimentar a aplicação de máquina. Ademais, ele é responsável também pelo balanceamento das requisições web feitas ao dyno do tipo web, caso exista mais de uma cópia deste (veja o tópico “Escalando a aplicação”, mais à frente).

Criando uma aplicação

Antes de criar uma aplicação, devemos criar uma conta no Heroku. Para isso, o leitor deve acessar a página de criação de contas no site [1] e fornecer o seu e-mail, conforme mostra a **Figura 1**.

Depois de informar o e-mail, clique no botão *Sign Up*. Uma mensagem deve ser exibida informando que uma mensagem de confirmação foi enviada para você. Assim, abra o e-mail e acesse o link indicado na mensagem. Você será redirecionado para uma página para concluir o cadastro, como demonstra a **Figura 2**.

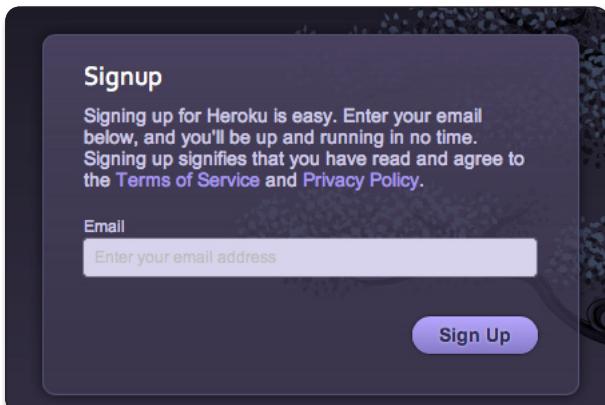


Figura 1. Página de criação de contas do Heroku

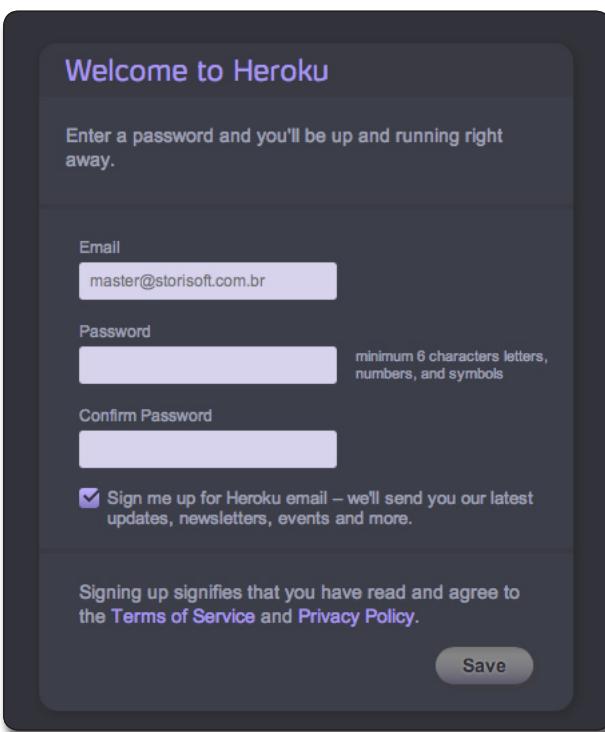


Figura 2. Página para confirmação do cadastro

Após informar e confirmar a senha, ao clicar em *Save* a sua conta será criada e você estará automaticamente logado. Com isso você será redirecionado ao dashboard do usuário, que exibe uma lista com as suas aplicações.

Existem duas formas de criar e manter aplicações no Heroku. Uma delas é através do próprio site. A outra, bem mais atraente para os desenvolvedores, é através da linha de comando, utilizando o Heroku Toolbelt. O artigo abordará as duas formas, mas inicialmente vamos utilizar o Toolbelt. Para baixá-lo, acesse a página desta ferramenta [2] e você será apresentado às opções de download de acordo com as plataformas disponíveis. Existem versões para Mac OS X, Windows, Debian/Ubuntu e uma opção Standalone. Selecione a opção que lhe convém, baixe o instalador e instale o aplicativo.

Após a instalação, entre no terminal e digite o comando *heroku* para testar o Toolbelt. Caso a instalação tenha sido feita com sucesso, será exibido na tela um mini-help do comando; caso contrário, verifique a etapa de instalação e se for o caso, reinstale o aplicativo.

Antes de criar uma aplicação é necessário informar as suas credenciais para o Heroku Toolbelt. Para isso, digite o comando *heroku auth:login*, como demonstra a **Listagem 1**.

Será solicitado o e-mail e a senha utilizados na criação da conta. Após a autenticação você pode verificar a lista das aplicações que você administra digitando *heroku apps*. Observe a **Listagem 2**.

Listagem 1. Autenticando no console usando o Heroku Toolbelt.

```
$ heroku auth:login  
Enter your Heroku credentials.  
Email: master@storisoft.com.br  
Password (typing will be hidden):  
Authentication successful.
```

Listagem 2. Verificando suas aplicações.

```
$ heroku apps  
You have no apps.
```

Como ainda não criamos nenhuma aplicação, o Heroku responde gentilmente: “Você não tem apps”.

Concluída essa etapa, vamos ver a seguir como criar um simples Hello World usando o Toolbelt. Para isso, crie um diretório que será o repositório local do fonte da nossa aplicação e entre neste diretório pelo console. Como o sistema de build do Heroku se baseia em repositórios Git, você deverá inicializar o diretório como um repositório desse tipo, conforme expõe a **Listagem 3**.

Para criar uma aplicação pelo Toolbelt usamos o comando *heroku apps:create*, conforme a **Listagem 4**.

Listagem 3. Inicializando um repositório git para a aplicação.

```
$ mkdir heroku-hello-world  
$ cd heroku-hello-world  
$ git init  
Initialized empty Git repository in /private/var/root/heroku-hello-world/.git/
```

Listagem 4. Criando uma aplicação usando o Heroku Toolbelt.

```
$ heroku apps:create heroku-hello-world-jm  
Creating heroku-hello-world-jm... done, stack is cedar  
http://heroku-hello-world-jm.herokuapp.com/  
| git@heroku.com:heroku-hello-world-jm.git
```

Este comando aceita como parâmetro o nome da aplicação a ser criada; no caso do exemplo, *heroku-hello-world-jm*. É importante ressaltar que este nome é único entre as aplicações do Heroku. Sendo assim, o leitor deve alterar o nome da aplicação antes de executar o comando, caso contrário o erro “Name is already taken” (nome já utilizado) será retornado.

Outro detalhe é que o nome da aplicação faz parte do domínio reservado para ela. Portanto, já podemos acessá-la no navegador de acordo com a URL retornada pelo comando (<http://heroku-hello-world-jm.herokuapp.com/>). Observe a Figura 3.

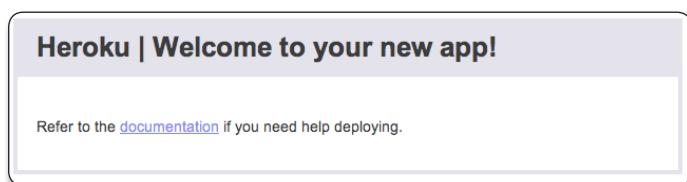


Figura 3. Tela da aplicação instalada no Heroku

Além de digitar a URL manualmente no navegador para abrir a aplicação, podemos fazer isso de maneira automática usando o comando `heroku apps:open`.

Vale ressaltar que alguns comandos do Heroku Toolbelt, como o `heroku apps:open`, dependem de um contexto de aplicação. Isto é, na prática, você precisa estar no diretório da aplicação para executar este tipo de comando, para que o Toolbelt saiba em qual aplicação você está querendo atuar, pois sua conta pode ter várias apps criadas. Se você tentar executar este tipo de comando em um diretório que não contenha uma aplicação do Heroku, uma mensagem semelhante à exibida na Listagem 5 será exibida.

Listagem 5. Resultado de um comando fora de contexto.

```
! No app specified.  
! Run this command from an app folder or specify which app to use with --app APP.
```

A própria mensagem de erro já dá a dica. A partir de qualquer diretório é possível executar um comando para uma aplicação. No entanto, para isso, é necessário informar o parâmetro `--app APP`. Isto vale para qualquer comando que dependa de uma aplicação para ser executado.

Voltando para nosso Hello World, notamos que o Heroku está apresentando uma página padrão para aplicações recém-criadas. Isso acontece porque não temos nenhum Web Dyno sendo executado e por esse motivo não existe conteúdo web para ser exibido. Para confirmar isso, execute o comando `heroku ps` dentro do diretório da aplicação. Este comando retorna os processos (os dynos) da aplicação e o estado de cada um deles. Como não temos nenhum ainda, o comando não retorna nada.

Precisamos agora criar a aplicação propriamente dita e subi-la para o Heroku. Para que seja detectada pelo Heroku, uma aplicação Java deve usar o Maven como gerenciador de dependências e de build. Sendo assim, devemos criar a aplicação respeitando as convenções do Maven. Dito isso, a partir da raiz da aplicação, crie um diretório `src/main/java` aonde será criada a classe `HelloWorld`, conforme Listagem 6.

Vejamos o código do Hello World. Primeiro é importante ressaltar que toda aplicação no Heroku deve ser executável, ou seja, não pode depender de um container pré-executado. No Heroku

não temos um Tomcat/Jetty/JBoss/etc. à nossa disposição, pois o Heroku prega que a aplicação deve ser autossuficiente. Existe uma série de explicações que justificam esta arquitetura, mas não vem ao caso detalhá-las. Apenas temos que ter em mente que precisamos definir sempre um ponto de início em nossa aplicação.

Listagem 6. Código da classe `HelloWorld`.

```
import java.io.IOException;  
import javax.servlet.ServletException;  
import javax.servlet.http.*;  
import org.eclipse.jetty.server.Server;  
import org.eclipse.jetty.servlet.*;  
  
public class HelloWorld extends HttpServlet {  
  
    @Override  
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)  
        throws ServletException, IOException {  
        resp.getWriter().print("Hello from Java!\n");  
    }  
  
    public static void main(String[] args) throws Exception{  
        Server server = new Server(Integer.valueOf(System.getenv("PORT")));  
        ServletContextHandler context = new ServletContextHandler(ServletContext  
            Handler.SESSIONS);  
        context.setContextPath("/");  
        server.setHandler(context);  
        context.addServlet(new ServletHolder(new HelloWorld()),"/*");  
        server.start();  
        server.join();  
    }  
}
```

Neste exemplo, o ponto de início da aplicação é o método `main()` da classe `HelloWorld`. Logo no começo do método é declarado um servidor Jetty embarcado e atribuído uma porta em que as requisições web serão recebidas. Outro ponto importante é que toda aplicação web no Heroku deve escutar a porta declarada na variável de ambiente `PORT`, porque não temos como saber em tempo de desenvolvimento qual porta que o dyno criado para a aplicação irá escutar. Na sequência, criamos um contexto de servlet, registramos a própria classe (que é um servlet) e informamos ao servidor Jetty que ele irá usar este servlet para atender todas as requisições (através do wildcard `/*`). Posteriormente iniciamos o servidor com os métodos `start()` e `join()`.

Normalmente não lidamos com este tipo de código (iniciar um container manualmente) porque criamos aplicações com servlets e as instalamos em algum container. No exemplo, a própria aplicação é um container, porque o Heroku assim exige.

Como a gestão de dependências da aplicação e o processo de construção (build) se baseiam no Maven, também devemos criar um `pom.xml` e colocá-lo na raiz da aplicação. Veja o código do nosso `pom.xml` na Listagem 7.

Conforme explicado anteriormente, a aplicação deve ser autossuficiente, então declaramos como dependência o container Jetty e a API de servlet. Além disso, existe uma configuração específica para instruir o Maven a copiar as dependências do projeto (`maven-dependency-plugin`). Isto se faz necessário porque como nossa

aplicação será um JAR, não podemos colocar as dependências dentro dela, como fazemos normalmente com um WAR.

Depois de criada a classe e o pom, basta executar o build com o comando a seguir:

```
$ mvn clean install
```

Listagem 7. Arquivo POM da aplicação.

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <version>1.0-SNAPSHOT</version>
    <artifactId>helloworld</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-servlet</artifactId>
            <version>7.6.0.v20120127</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <version>2.4</version>
                <executions>
                    <execution>
                        <id>copy-dependencies</id>
                        <phase>package</phase>
                        <goals><goal>copy-dependencies</goal></goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Após a finalização do build, examine a pasta *target* gerada pelo Maven. Você irá verificar que o JAR da aplicação está na raiz do *target* e todas as dependências da aplicação foram copiadas para a pasta *target/dependency*. O resultado disso é que temos uma aplicação que é executável a partir de uma simples chamada ao executável do Java, exatamente como o Heroku prega que as aplicações devem ser.

Para testar a aplicação localmente, inicie uma variável de ambiente chamada PORT (para simular o ambiente do Heroku) e atribua um valor que deseja que a aplicação use. Em seguida, faça a chamada ao Java para executar a aplicação, como realizado na **Listagem 8**.

Agora, abra um navegador e informe a URL <http://localhost:8080> para ver a aplicação rodando. Para fazer o shutdown, apenas pressione *Ctrl-C* no console usado para iniciar a aplicação.

Antes de instalar a aplicação no Heroku, no entanto, devemos criar o arquivo *Procfile* no diretório raiz da aplicação.

Este arquivo informa ao Heroku os tipos de dynos que serão criados e o comando para iniciar cada um deles. Como teremos apenas um dyno do tipo web, nosso *Procfile* deve ficar com o conteúdo exibido na **Listagem 9**.

Listagem 8. Testando a aplicação localmente.

```
//No MAC ou no Linux:
$ export PORT=8080
$ java -cp target/classes:"target/dependency/*" HelloWorld
```

```
//No Windows:
$ set PORT=8080
$ java -cp target\classes;"target\dependency\*" HelloWorld
```

Listagem 9. Procfile da aplicação.

```
web: java -cp target/classes:target/dependency/* HelloWorld
```

Primeiro informamos o tipo do dyno (web), acrescentamos dois pontos e o comando utilizado para iniciar o dyno. Lembrando que podemos ter apenas um tipo de dyno web e diversos do tipo worker.

Com isso temos todos os arquivos necessários, sendo preciso apenas comitar nosso código para o repositório remoto do Heroku (criado automaticamente no comando *heroku apps:create*) e o serviço de deploy contínuo do Heroku fará o resto do trabalho. Antes de comitar, no entanto, temos que registrar um par de chaves pública/privada para garantir a segurança no tráfego dos dados. Este registro, exibido na **Listagem 10**, deve ser feito apenas uma vez em cada máquina utilizada para o desenvolvimento da aplicação.

Finalmente, realizamos o commit para o repositório remoto da aplicação no Heroku, conforme demonstra a **Listagem 11**.

Listagem 10. Gerando uma chave RSA.

```
$ ssh-keygen -t rsa
$ heroku keys:add
```

Listagem 11. Realizando o commit para o repositório remoto da aplicação.

```
$ git add .
$ git commit -m "Commit inicial"
$ git push heroku master
```

Neste momento o Heroku irá logar todo o processo de build e iniciar a aplicação. Feito isso, podemos abrir a aplicação já instalada no Heroku a partir do comando: *heroku apps:open* (este comando é um facilitador para não termos que abrir o browser e digitar a URL). A tela da aplicação deve ser similar à exibida na **Figura 4**.

O processo de atualização da aplicação é mais simples que o de criação, pois exige apenas que você atualize o repositório remoto. Para confirmar, faça uma alteração na classe criada, por exemplo, alterando a mensagem para “Hello from Java Again!”, salve o arquivo e atualize o repositório, conforme a **Listagem 12**.

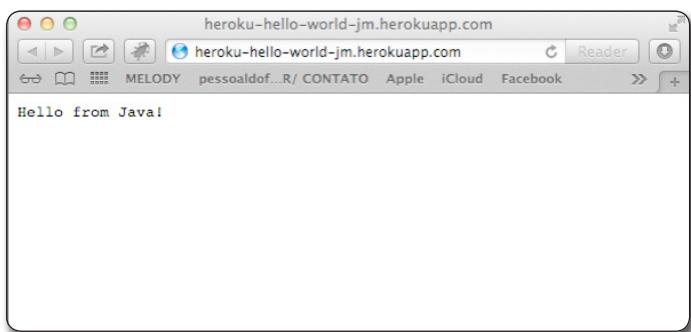


Figura 4. Tela da aplicação atualizada

Listagem 12. Realizando o commit da alteração para o repositório remoto da aplicação.

```
$ git add .  
$ git commit -m "Alteração da mensagem"  
$ git push heroku master
```

Assim como na criação, o Heroku irá logar todo o processo de build e atualização da aplicação. Após a conclusão, acesse novamente a aplicação no browser e você verá a mensagem alterada.

Escalando a Aplicação

Conforme vimos no início do artigo, é possível criar cópias de dynos para conseguir escalar a aplicação horizontalmente. Por exemplo, se a aplicação começar a apresentar lentidão, pode ser que apenas um dyno do tipo web não seja suficiente para atender a demanda. Veja que isso não significa que você deve declarar dois dynos do tipo web no Procfile, pois isso não é permitido, mas sim **copiar** o dyno existente, consequentemente criando várias instâncias que atenderão as requisições feitas à aplicação. Como já informado, as requisições são balanceadas pelo Dyno Manager.

Para escalar a aplicação, você deve utilizar o comando *heroku ps:scale*, conforme a **Listagem 13**.

Listagem 13. Escalando a aplicação com o comando *ps:scale*.

```
$ heroku ps:scale web=2
```

Neste momento o Heroku irá criar mais uma instância do dyno do tipo web (de acordo com o Procfile) para a aplicação, permitindo assim que a carga de requisições seja distribuída entre duas instâncias. É importante informar que ao escalar um dyno para mais de uma instância o Heroku passará a cobrar, pois o plano gratuito se restringe apenas a aplicações com apenas uma instância de cada tipo de dyno.

Com isso concluímos todo o processo de desenvolvimento e deploy de uma simples aplicação Java na plataforma Heroku. Antes de apresentarmos mais detalhes do Toolbelt e partirmos para uma aplicação próxima do nosso cotidiano, vamos abrir um parêntese para ver alguns pontos importantes de como devemos pensar e projetar uma aplicação para rodar em uma estrutura PaaS.

The Twelve-Factor App

O leitor deve ter reparado que usamos um modelo diferente para desenvolver e subir aplicações web em Java. Normalmente criamos um pacote do tipo WAR ou EAR e o instalamos em algum container como Tomcat, Jetty, JBoss, etc. A aplicação que criamos declara o container como dependência e o inicia dentro dela própria.

Esta técnica, conhecida como embedded container (container embarcado), vai justamente de encontro com o modelo de aplicação que o Heroku adota, no qual todas as dependências da aplicação, inclusive o container em que ela deve executar, devem ser declaradas explicitamente como dependência, fazendo com que todo o conteúdo da aplicação seja empacotado no build.

Dentre outras vantagens, usar um container embarcado facilita o processo de criação de instâncias (dynos) para executar a aplicação. No modelo PaaS, escalar basicamente significa criar mais instâncias (mais dynos) da aplicação e distribuir a carga entre elas (balanceamento). Sendo assim, criar uma aplicação completamente autocontida facilita o processo de criação de instâncias, pois reduz a dependência da aplicação de artefatos externos, deixando a instância mais leve, mais fácil de ser criada ou clonada.

Esta característica autossuficiente de uma aplicação é abordada na metodologia chamada de **The Twelve-Factor App** [3],

**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**

criada por Adam Wiggins, um dos fundadores do Heroku. Esta metodologia descreve, em doze capítulos, boas práticas de desenvolvimento de aplicações web com um grande enfoque em computação em nuvem e escalabilidade. Não é coincidência que o modelo do Heroku se apoie fortemente nesta metodologia. Vale ressaltar que ela não se aplica somente ao Heroku, mas sim em qualquer modelo PaaS.

Avançando no Heroku Toolbelt

O Heroku Toolbelt possui inúmeros comandos adicionais que permitem ao usuário gerenciar as suas aplicações e obter informações da plataforma. A seguir iremos expor os principais.

Geralmente os comandos têm o seguinte formato: *heroku comando:opcao parâmetros*. É possível obter uma ajuda de um determinado comando digitando *heroku comando --help* ou *heroku comando:opcao --help*.

heroku addons

O Heroku, assim como outras plataformas PaaS, permite ao desenvolvedor adicionar complementos na aplicação, como veremos adiante. Estes complementos são chamados de add-ons. É possível listar, adicionar e configurar os add-ons a partir do comando *heroku:addons*. Por exemplo, para listar os complementos instalados na aplicação, use o comando *heroku addon:list*.

heroku apps

No início do artigo usamos o comando *heroku apps* para criar uma aplicação e para abri-la no navegador. Estas são apenas duas das formas de utilização deste comando. Também podemos invocá-lo para obter informações a respeito da aplicação ou até mesmo removê-la. Por exemplo, ao executar: *heroku apps:info*, teremos o resultado indicado na **Listagem 14**.

Listagem 14. Obtendo informações de uma aplicação com o *apps:info*.

```
$ heroku apps:info
== heroku-hello-world-jm
Addons: heroku-postgresql:dev
Git URL: git@heroku.com:heroku-hello-world-jm.git
Owner Email: master@storisoft.com.br
Region: us
Repo Size: 23M
Slug Size: 49M
Stack: cedar
Tier: Legacy
Web URL: http://heroku-hello-world-jm.herokuapp.com/
```

heroku config

O comando *heroku config* permite listar as variáveis de ambiente disponíveis para a aplicação, bem como criar novas variáveis ou alterar as existentes.

Muitas configurações de uma aplicação devem residir em variáveis de ambiente para facilitar a sua execução em vários cenários sem a necessidade de recompilação. Por exemplo, a URL de conexão com o banco de dados. Colocando a URL como variável,

podemos usar um tipo de banco para testar a aplicação localmente e outro em produção. A **Listagem 15** mostra o uso deste comando para esta finalidade.

Listagem 15. Listando e adicionando variáveis de ambiente de uma aplicação.

```
$ heroku config
== heroku-hello-world-jm Config Vars
DATABASE_URL:
postgres://dudrpctynszctv:4rfpHLaCiJuCirwU5mbj5M7wGb@ec2-54-235-173-50.
compute-1.amazonaws.com:5432/d8o514s1rp16vv
HEROKU_POSTGRESQL_YELLOW_URL:
postgres://dudrpctynszctv:4rfpHLaCiJuCirwU5mbj5M7wGb@ec2-54-235-173-50.
compute-1.amazonaws.com:5432/d8o514s1rp16vv
JAVA_OPTS: -Xmx384m -Xss512k -XX:+UseCompressedOops
MAVEN_OPTS: -Xmx384m -Xss512k -XX:+UseCompressedOops
PATH: /app/.jdk/bin:/usr/local/bin:/usr/bin:/bin

$ heroku config:set TEST_VARIABLE='Value of the thest variable'
Setting config vars and restarting heroku-hello-world-jm... done, v8
TEST_VARIABLE: Value of the thest variable
```

heroku logs

No Heroku, todos os logs da aplicação são gerenciados pelo Logplex [4], que é um sistema de entrega de logs com foco em aplicações distribuídas, que é o caso do Heroku. O Logplex centraliza as informações de todos os containers (dynos) evitando que tenhamos que monitorá-los individualmente. Por padrão ele não persiste o conteúdo, pois funciona como um stream. Sendo assim, fica a cargo do cliente armazenar os logs, caso julgue necessário.

Ao executar o comando *heroku logs* o Toolbelt irá exibir somente as últimas linhas dos logs da aplicação. No entanto, se acrescentarmos o parâmetro *-t*, um stream será mantido na janela do console que o comando for executado (equivalente ao comando *tail* do Linux/Unix), sendo interrompido quando usuário enviar uma solicitação de interrupção (*Ctrl+c*).

heroku maintenance

Apesar de ser altamente recomendável realizar atualizações que não gerem impacto para o usuário, às vezes isso é inevitável. Por exemplo, podemos ter que executar um longo script de atualização do banco de dados que deixe a aplicação quebrada por um período de tempo. Para isso, o Heroku disponibiliza uma configuração chamada modo de manutenção. Você pode ativar esse modo enquanto estiver realizando alguma operação que inviabilize o uso da aplicação, tornando a desativá-lo ao finalizar este processo. Para ativar o modo de manutenção, apenas digite: *heroku maintenance:on*. A sua aplicação passará a responder de acordo com a **Figura 5**.

Um complemento importante possibilita customizar uma página de manutenção. Para isto, basta setar a variável de ambiente *MAINTENANCE_PAGE_URL* conforme a **Listagem 16**.

Para retornar a aplicação para o estado normal, digite: *heroku maintenance:off*.

Application Offline for Maintenance

This application is undergoing maintenance right now. Please check back later.

Figura 5. Aplicação em modo de manutenção

Listagem 16. Informando uma página de manutenção customizada.

```
$ heroku config:set MAINTENANCE_PAGE_URL=  
http://heroku-hello-world-jm.herokuapp.com/custompage.html  
Setting config vars and restarting heroku-hello-world-jm... done, v10  
MAINTENANCE_PAGE_URL: http://heroku-hello-world-jm.herokuapp.com/custom-  
page.html
```

heroku releases

O comando *heroku releases* lista todas as manutenções feitas na aplicação desde a sua criação. Entre as manutenções estão os deploys feitos (a partir dos commits), criação de variáveis de ambiente, habilitação de logs, etc. A **Listagem 17** mostra esse comando em ação.

Para cada manutenção feita, uma nova linha é exibida associada a uma versão (v1, v2, v3, etc.). Com este comando também é possível retornar para uma das versões anteriores, informando, por

exemplo: *heroku releases:rollback v8* (que neste caso retorna para a v8). Assim, as operações feitas depois da versão informada no comando rollback serão desfeitas.

Listagem 17. Listando as manutenções da aplicação.

```
$ heroku releases  
==== heroku-hello-world-jm Releases  
v9 Add MAINTENANCE_PAGE_URL config  
2013/07/21 08:52:42 (~ 6m ago) master@storisoft.com.br  
v8 Add TEST_VARIABLE config  
2013/07/21 08:29:14 (~ 30m ago) master@storisoft.com.br  
v7 Deploy 54fd067  
2013/07/16 14:55:29 master@storisoft.com.br  
v6 Deploy a4f0fb6  
2013/07/14 10:36:38 master@storisoft.com.br  
v5 Add PATH, JAVA_OPTS, MAVEN_OPTS config  
2013/07/14 10:36:38 master@storisoft.com.br  
v4 Add DATABASE_URL config  
2013/07/14 10:36:37 master@storisoft.com.br  
v3 Attach HEROKU_POSTGRESQL_YELLOW resource master@storisoft.com.br  
2013/07/14 10:36:37 master@storisoft.com.br  
v2 Enable Logplex  
2013/07/14 08:54:47 master@storisoft.com.br  
v1 Initial release  
2013/07/14 08:54:44 master@storisoft.com.br
```

Existem mais comandos a serem explorados no Heroku Toolbelt. Para conhecer todos eles, digite apenas *heroku* e a lista será exibida. Para ver detalhes de um comando, digite: *heroku comando –help*.

Aplicações Java

Na Impacta você tem a formação mais completa na linguagem mais relevante do mercado*

 **ESCOLA DESENVOLVIMENTO**

Treinamentos em: JEE - JavaServer Pages + Servlets • JAVA - Design Patterns • JAVA - Hibernate • JAVA Spring Framework • JEE - Apache Struts • JEE - Enterprise • JAVABeans • JEE - Java Web Services • JME - Aplicações para dispositivos móveis • JSF - JavaServer Faces • Java Programmer • Android

* Segundo o índice TIOBE, que considera a posição de cada linguagem de programação nos buscadores.



Conheça também a Pós-Graduação em Engenharia de Software na Faculdade Impacta

- Os melhores professores e instrutores do mercado;
- A melhor Infraestrutura;
- A melhor metodologia;

Desenvolvendo com o Eclipse

No início do artigo criamos um projeto manualmente sem o auxílio de uma IDE com o propósito de compreender o funcionamento do processo de criação e deploy de uma aplicação no Heroku. Para conseguir mais produtividade, podemos usar o Eclipse com um plugin desenvolvido pela equipe do Heroku. Estas informações estão no próprio site do Heroku [5].

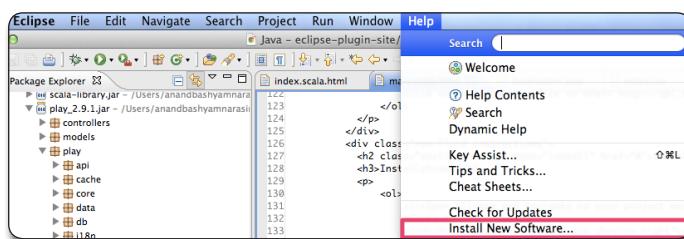


Figura 6. Iniciando a instalação do plugin

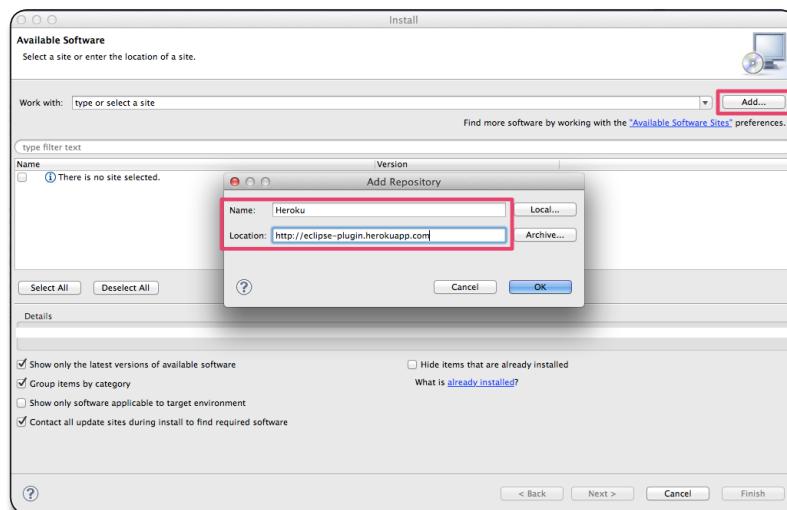


Figura 7. Selecionando o repositório para instalação do plugin

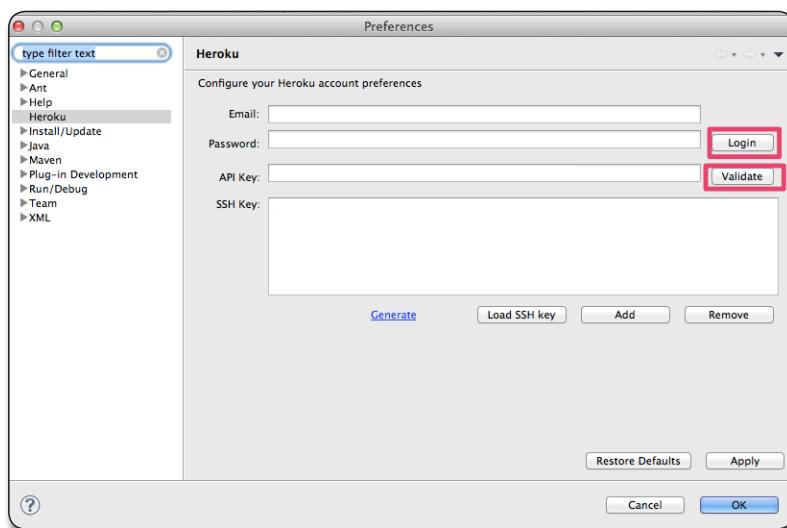


Figura 8. Efetuando o login via Eclipse a partir das configurações do plugin

O plugin funciona em versões 3.7 ou superiores do Eclipse. À vista disso, abra a IDE, clique no menu *Help* e depois em *Install New Software*, conforme a **Figura 6**.

Em seguida, clique no botão *Add...*, conforme a **Figura 7**. Dessa forma, será exibida a tela *Add Repository*. Nela, informe “Heroku” no campo *Name* e a URL “<https://eclipse-plugin.herokuapp.com/install>” no campo *Location*, de acordo com a **Figura 7**. Feito isso, clique no botão *OK*.

Após adicionar o site, selecione o checkbox com a opção *Heroku Eclipse Integration* e clique no botão *Next*. Clique em *Next* novamente e depois aceite o termo de licença e clique em *Finish*. Durante a instalação o Eclipse precisará ser reiniciado.

Depois de concluir a instalação e reiniciar a IDE, você deve configurar o plugin para vincular a sua conta do Heroku. Para isso, no menu *Window > Preferences* (ou *Eclipse > Preferences*, caso você esteja no Mac), selecione Heroku na lista que fica à esquerda da tela. Ao fazer isso, a tela demonstrada na **Figura 8** será exibida.

Nesse momento, informe o e-mail da sua conta e a senha e clique no botão *Login*. Caso o login seja realizado com sucesso, o campo API Key será preenchido automaticamente. Por fim, clique em *OK* para gravar as configurações.

Com o plugin instalado e a conta devidamente configurada, podemos importar o projeto que criamos no início do artigo. Portanto, clique no menu *File > Import*, selecione a opção *Import existing Heroku App* e clique em *Next*. Uma tela com as suas aplicações será exibida. Selecione então a aplicação que criamos neste artigo e clique mais uma vez em *Next*. Na próxima tela, marque a opção *Auto detect project* e clique em *Finish*. Com isso, por baixo dos panos, o plugin está realizando um *git clone* no repositório remoto da sua aplicação no Heroku.

Para verificar o funcionamento de um deploy com o plugin, abra a classe **HelloWorld** e altere o texto de saída do servlet, como o exemplo exposto na **Listagem 18**.

Após alterar a classe, clique com o botão direito do mouse no projeto e selecione o menu *Team > Commit*. O plugin irá exibir uma tela pedindo para o usuário digitar a mensagem do commit para o controle de versão. Digite a mensagem e depois clique no botão *Commit and Push*. Com isso, o plugin irá realizar um *git add*, um *git commit* e um *git push* para atualizar o repositório remoto da sua aplicação. Assim que detecta uma atualização no repositório, o Heroku inicia o processo de build atualizando automaticamente a aplicação. Encerrado esse processo, abra o navegador, accese a aplicação e você poderá notar que a mensagem mudou.

Usando o Template de Spring MVC e Hibernate

Agora que vimos o funcionamento do plugin, vamos seguir adiante e criar uma aplicação usando Spring MVC e Hibernate com o auxílio do plugin do Heroku para o

Eclipse. O Hibernate dispensa apresentações por ser o framework mais popular de mapeamento objeto relacional (ORM). Ele será responsável pela persistência de dados da nossa aplicação.

O Spring MVC, por sua vez, é um projeto da SpringSource para o desenvolvimento de aplicações web que implementa o padrão MVC (Model-View-Controller), equivalente aos populares Struts, Tapestry e JSF. Ao usar o Spring MVC contamos também com outros módulos do Spring, como DI (Injeção de Dependências) e gestão de transação automática.

Listagem 18. Alterando a classe HelloWorld.

```
...
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    resp.getWriter().print("Hello from Java and Eclipse!\n");
}
...
```

A junção do Spring com o Hibernate forma uma solução extremamente leve que permite que a aplicação seja executada em um container como o Jetty ou Tomcat, dispensando o uso de um container EJB. É importante considerar aspectos como esse quando

formos utilizar uma infraestrutura de PaaS devido às limitações de memória e pela agilidade no start das aplicações.

Dito isso, vamos à implementação da aplicação. Com o Eclipse aberto, clique no menu *File > New* e selecione a opção *Other*. Entre as alternativas disponíveis, expanda a pasta *Heroku* e selecione o subitem *Create Heroku App from Template*. Em seguida, clique em *Next* (veja a **Figura 9**).

Nesta etapa serão exibidas as opções de template de aplicação que o plugin do Heroku oferece. Para o nosso exemplo, selecione a primeira opção: *Spring MVC & Tomcat application* (veja a **Figura 10**). Feito isso, no campo *Application Name* informe o nome da aplicação e depois clique em *Next*.

Após concluir o wizard o plugin irá baixar o template da aplicação, inicializar um repositório git local, criar uma aplicação no Heroku com o nome selecionado, vincular a aplicação ao repositório git local criado e realizar o commit e o push do código gerado pelo template para o repositório remoto do Heroku, fazendo com que a aplicação seja instalada. Ou seja, tudo o que fizemos manualmente utilizando o console, o plugin fez apenas com alguns cliques.

A aplicação criada pelo plugin é um cadastro que permite incluir, alterar, consultar e excluir pessoas. Os campos solicitados são nome e sobrenome (*First Name* e *Last Name*).

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

Primeiros passos em PaaS com Heroku

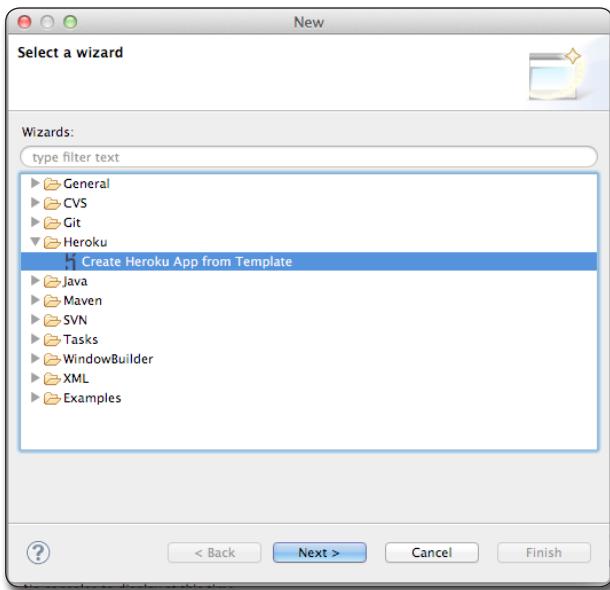


Figura 9. Criando uma nova aplicação usando o plugin do Eclipse

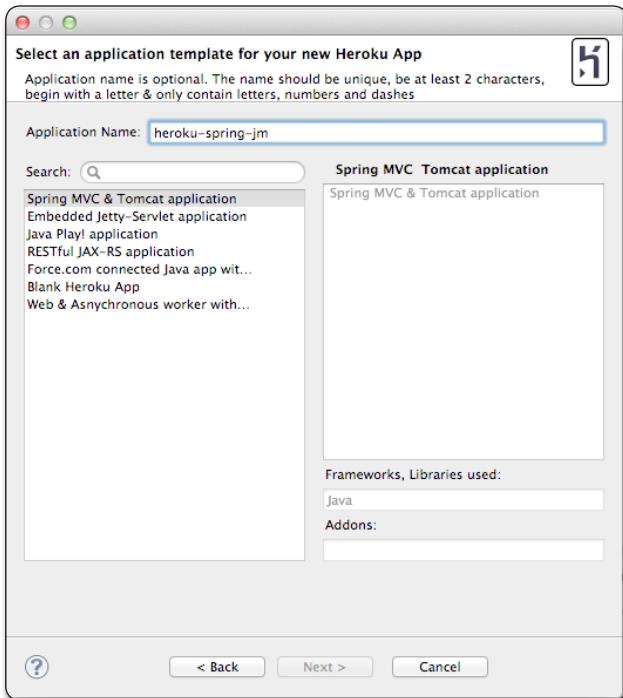


Figura 10. Selecionando o tipo de template da aplicação

Depois de criada, a aplicação é automaticamente comitada no repositório remoto e consequentemente entra no ciclo de build e atualização do Heroku, podendo ser acessada pelo browser por meio da URL: <http://heroku-spring-jm.herokuapp.com/people/>. Vale lembrar que você deve alterar o trecho da URL com `heroku-spring-jm` para o nome do seu projeto. Observe a Figura 11.

Assim como toda aplicação desenvolvida para rodar no Heroku, a aplicação criada a partir do template também conta com um arquivo de nome Procfile na raiz do projeto para descrever os

processos (dynos). Para verificar o comando utilizado para iniciar a aplicação, veja o conteúdo do Procfile gerado pelo plugin, mais especificamente a linha que descreve o dyno do tipo web. Executando o comando descrito neste arquivo no diretório da aplicação na sua máquina, a aplicação será iniciada localmente, como o exemplo no início do artigo.

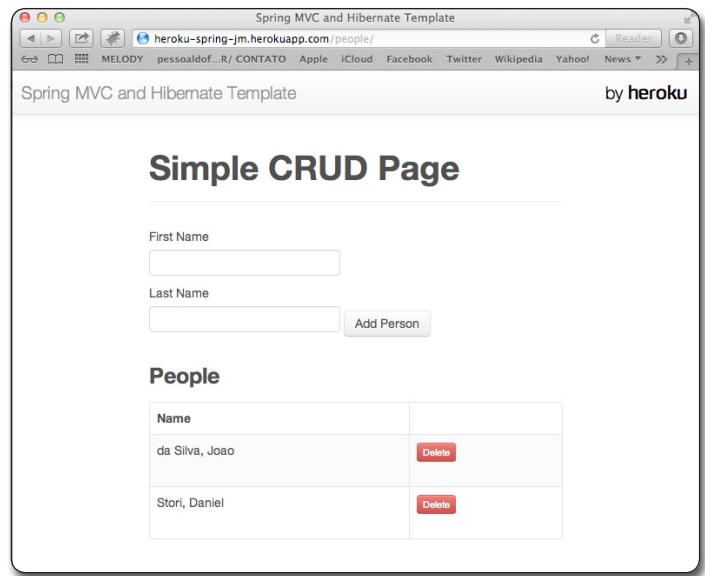


Figura 11. Aplicação construída a partir do template

Add-Ons

Como em outras plataformas PaaS, o Heroku oferece uma grande quantidade de complementos, chamados de add-ons. Estes são divididos em categorias como Data Stores, Mobile, Search, entre outras. A lista completa está disponível no site do Heroku em uma seção dedicada aos complementos [6]. Os add-ons contam com uma página que descreve brevemente seu funcionamento e o modelo de especificação. Assim como ocorre com o próprio Heroku, normalmente eles possuem uma versão gratuita com limite de uso. A Figura 12 mostra detalhes da especificação do MemCachier, que é uma implementação de cache distribuído.

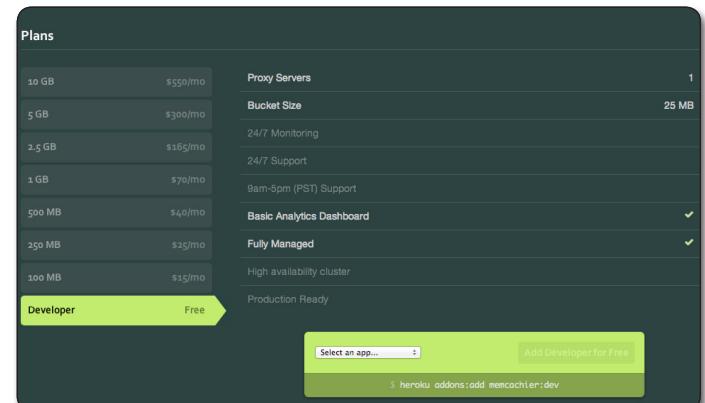


Figura 12. Página do add-on MemCachier

Existem duas formas de adicionar um add-on à aplicação. Uma delas é fornecida por meio do combo que fica na parte inferior da página, com a descrição *Select an app....* A partir dele o leitor pode selecionar a aplicação desejada e clicar no botão *Add Developer for Free* para adicionar o add-on. A outra opção é usar o Heroku Toolbelt com o comando exibido logo abaixo do combo.

Todo add-on possui uma boa documentação. Os links para esta documentação são exibidos depois da tabela de especificação. No caso da documentação do MemCachier, podemos encontrar, dentre outras coisas, informações como a API utilizada, exemplos de código em várias linguagens e formas de monitoração.

Conclusão

Soluções do tipo PaaS são sem dúvida o caminho mais rápido para subir e escalar uma aplicação na nuvem. Quando comparado a ofertas de IaaS, a agilidade e a facilidade de manutenção são muito superiores. Nesse contexto, o Heroku se coloca no mercado como uma solução simples, expansível no que diz respeito aos seus add-ons e muito fácil de escalar horizontalmente. Um dos pontos negativos é a pouca quantidade de oferta para escalar verticalmente, pois os dynos são oferecidos apenas em dois tamanhos, conforme descrito no início do artigo. Consequentemente, se a sua aplicação exigir muita memória e não for possível refatorá-la para reduzir esta necessidade, o Heroku não é indicado para ela, pois irá sofrer com lentidão devido o swap de memória em disco.

Em contrapartida, para a maioria das aplicações – principalmente as startups – o Heroku se encaixa como uma luva, pois tem uma especificação que permite o funcionamento por um bom tempo de maneira gratuita.

Outro ponto que chama atenção é a documentação. O Heroku possui uma vasta documentação, tanto de sua arquitetura quanto de seus complementos, disponíveis nas páginas do chamado Heroku Dev Center [8].

Por fim, vale lembrar também do Heroku Toolbelt, um diferencial que atrai principalmente os desenvolvedores, pois permite realizar todo o gerenciamento e manutenção das aplicações pela linha de comando, além de possibilitar a criação de scripts para automatizar esses processos.

Autor



Daniel Stori

dstori@gmail.com

É Físico pelo Centro Universitário Fundação Santo André.

Desenvolve sistemas desde 1989 quando ganhou seu Apple II.



Já trabalhou com C/C++ e desde 2002 trabalha com Java. Passou por empresas como Banco ABN, Tata e atualmente está na TOTVS trabalhando na plataforma Fluig (<http://www.fluig.com>).

Links:

[1] Página de criação de conta do Heroku.

<https://id.heroku.com/signup>

[2] Página do Heroku Toolbelt.

<https://toolbelt.heroku.com>

[3] Página da metodologia The Twelve-Factor App.

<http://www.12factor.net>

[4] Página com informações do sistema de log Logplex.

<https://devcenter.heroku.com/articles/logplex>

[5] Página de ajuda do plugin do Heroku para o Eclipse.

<https://devcenter.heroku.com/articles/getting-started-with-heroku-eclipse>

[6] Página que lista os complementos (add-ons) disponíveis no Heroku.

<https://addons.heroku.com>

[7] Página de documentação do complemento MemCachier.

<https://devcenter.heroku.com/articles/memcachier>

[8] Documentação do Heroku e dos add-ons.

<https://devcenter.heroku.com>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Hadoop: fundamentos e instalação

Aprenda a configurar um ambiente para desenvolvimento de aplicações Big Data

Vivenciamos a era da informação, na qual volumes expressivos de dados são produzidos pelas mais diversas organizações e estruturas de sistemas, alcançando dimensões que superam com facilidade os *petabytes* diários. Tal volume surge de diversas fontes de dados, como, por exemplo, medições coletadas por sensores dos mais diversos tipos, histórico dos serviços oferecidos por sistemas Web, variados conteúdos produzidos pelos usuários em redes sociais, acesso a bases de dados de imagens e mapas, e muito mais. Tais fontes heterogêneas produzem uma quantidade de dados acima da capacidade que normalmente pode ser processada por tecnologias tradicionais de banco de dados relacional.

Nesse contexto, o termo Big Data (aqui denominado Bigdata) foi definido considerando as seguintes questões: (a) **volume** de dados em grande quantidade (acima de terabytes); (b) **velocidade** na criação e captura de dados brutos a taxas muito rápidas, podendo ser arquivos em lote, obtidos de bancos de dados, ou dados gerados em tempo real (em *streaming*); e, (c) **variedade** no formato dos dados, podendo ser estruturado, semiestruturado, e até mesmo não estruturado, ou uma combinação dessas variações. Essas três questões estão ilustradas na **Figura 1**.

Assim, considerando as características extremas do Bigdata, uma nova classe de aplicações deve ser construída para analisar grandes bases de dados, processar pesados cálculos sobre esses dados, identificar comportamentos e disponibilizar serviços especializados em seus domínios. Entretanto, não é uma tarefa trivial implementar tais soluções, pois há, na maioria das vezes, a inviabilidade de executá-las no modelo computacional tradicional, usando tecnologias baseadas em banco de dados relacional, e processando em máquinas com escalaabilidade baixa. Os ditos problemas grandes ou complexos chegam a consumir horas ou dias de processamento

Resumo DevMan

Porque este artigo é útil:

Este artigo aborda os fundamentos básicos e a instalação da tecnologia Apache Hadoop em um ambiente para desenvolvimento de aplicações que manipulam grandes volumes de dados (big data). Hadoop destaca-se como uma tecnologia aberta, baseada no paradigma MapReduce, que utiliza a computação paralela e distribuída para resolver o problema da escalabilidade no processamento de Bigdata, com garantias de tolerância a falhas. Das vantagens em adotá-lo, está o fato de se utilizar aglomerados de máquinas convencionais, tornando-o eficaz como solução de baixo custo de implantação. Ademais, com ele as empresas podem conquistar uma grande vantagem competitiva, dispondendo de um mecanismo que possibilita avaliar grandes quantidades de dados em busca de novas informações.

nas arquiteturas convencionais. Embora em constante evolução, os recursos computacionais convencionais são insuficientes para acompanhar a crescente complexidade das novas aplicações.

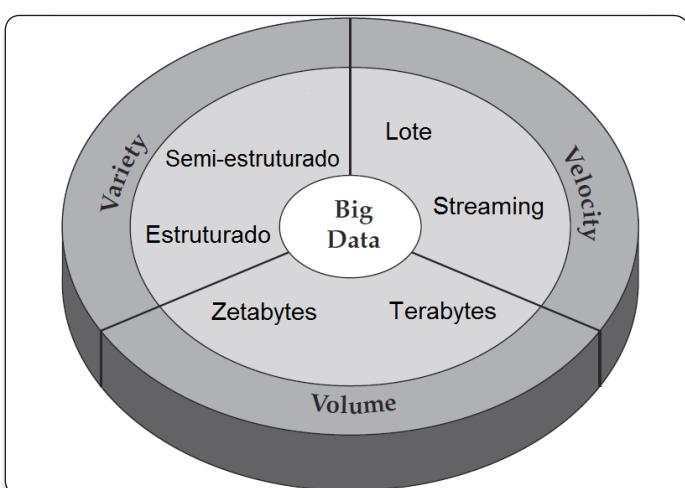


Figura 1. As três dimensões do Bigdata: volume, variedade e velocidade

Como proposta para superar os desafios, surge o Apache Hadoop, um *framework* para o processamento de grandes quantidades de dados em aglomerados e grades computacionais. A ideia de promover soluções para os desafios dos sistemas distribuídos em uma só plataforma é o ponto central do projeto Hadoop. Nessa plataforma, problemas como integridade dos dados, disponibilidade dos nós, escalabilidade da aplicação e recuperação de falhas são simplificadas para quem desenvolve as aplicações. Além disso, seu modelo de programação e sistema de armazenamento dos dados promove um rápido processamento, muito superior às outras tecnologias similares. Atualmente, além de estar consolidado no mundo empresarial, o Hadoop tem obtido crescente apoio da comunidade acadêmica, proporcionando, assim, estudos científicos e práticos.

Com base em tudo o que foi citado, este artigo apresenta os fundamentos das técnicas e dos conceitos envolvidos no projeto Apache Hadoop, em especial o modelo de programação MapReduce. Em seguida, são detalhadas as etapas para a instalação e configuração de um ambiente Hadoop a ser utilizado no desenvolvimento e testes de aplicações Bigdata.

Histórico

O Apache Hadoop é um *framework open source* para o armazenamento e processamento de dados em larga escala. Ele oferece como ferramentas principais uma implementação do modelo MapReduce, responsável pelo processamento distribuído, e o *Hadoop Distributed File System* (HDFS), para armazenamento de grandes conjuntos de dados, também de forma distribuída.

Embora recente, o Apache Hadoop tem se destacado como uma ferramenta eficaz, sendo utilizado por grandes corporações como IBM, Oracle, Facebook, Yahoo!, entre outras. Mas para chegar nesse ponto, alguns eventos importantes ocorreram nos últimos dez anos, como demonstram os fatos históricos a seguir:

- Fevereiro de 2003: Jeffrey Dean e Sanjay Ghemawat, dois engenheiros do Google, desenvolvem a tecnologia MapReduce, que possibilitou otimizar a indexação e catalogação dos dados sobre as páginas Web e suas ligações. O MapReduce permite dividir um grande problema em vários pedaços e distribuí-los em diversos computadores. Essa técnica deixou o sistema de busca do Google mais rápido mesmo sendo executado em computadores convencionais e menos confiáveis, diminuindo assim os custos ligados à infraestrutura;
- Outubro de 2003: O Google desenvolve o *Google File System*, um sistema de arquivos distribuído (o *GoogleFS*, depois chamado de GFS), criado para dar suporte ao armazenamento e processamento do grande volume de dados da tecnologia MapReduce;
- Dezembro de 2004: o Google publica o artigo *Simplified Data Processing on Large Clusters*, de autoria dos engenheiros Dean e Ghemawat, onde eles apresentam os principais conceitos e características da tecnologia MapReduce, porém, sem detalhes sobre a implementação;
- Dezembro de 2005: o consultor de software Douglas Cutting divulgou a implementação de uma versão do MapReduce e

do sistema de arquivos distribuídos com base nos artigos do GFS e do MapReduce publicados pelos engenheiros do Google. A implementação faz parte do subprojeto Nutch, adotado pela comunidade de software livre para criar um motor de busca na Web, normalmente denominado *web crawler* (um software que automatiza a indexação de páginas) e um analisador de formato de documentos (*parser*). Tempos depois o Nutch seria hospedado como o projeto Lucene, na *Apache Software Foundation* (ver seção **Links**), tendo como principal função fornecer um poderoso mecanismo de busca e indexação de documentos armazenados em diversos formatos, como arquivos de texto, páginas web, planilhas eletrônicas, ou qualquer outro formato do qual se possa extrair informação textual;

- Fevereiro de 2006: a empresa Yahoo! decide contratar Cutting e investir no projeto Nutch, mantendo o código aberto. Nesse mesmo ano, o projeto recebe o nome de **Hadoop**, passando a ser um projeto independente da *Apache Software Foundation*;
- Abril de 2007: o Yahoo! anuncia ter executado com sucesso uma aplicação Hadoop em um aglomerado de 1.000 máquinas. Também nessa data, o Yahoo! passa a ser o maior patrocinador do projeto. Alguns anos depois, a empresa já contava com mais de 40.000 máquinas executando o Hadoop (White, 2010);
- Janeiro de 2008: o Apache Hadoop, na versão 0.15.2, amadurece como um projeto incubado na fundação Apache, e torna-se um dos principais projetos abertos da organização;
- Julho de 2008: uma aplicação Hadoop em um dos aglomerados do Yahoo! quebra o recorde mundial de velocidade de processamento na ordenação de 1 *terabyte* de dados. O aglomerado era composto de 910 máquinas e executou a ordenação em 209 segundos, superando o recorde anterior que era de 297 segundos;
- Setembro de 2009: a empresa Cloudera, especializada em Bigdata, contrata Cutting como líder do projeto. Cloudera é uma empresa que redistribui uma versão comercial derivada do Apache Hadoop;
- Dezembro de 2011: passados seis anos desde seu lançamento, o Apache Hadoop disponibiliza sua versão estável (a 1.0.0). Entre as melhorias, destaca-se o uso do protocolo de autenticação de rede Kerberos, para maior segurança de rede; a incorporação do subprojeto HBase, oferecendo suporte a *BigTable*; e o suporte à interface WebHDFS, que permite o acesso HTTP para leitura e escrita de dados;
- Maio de 2012: a Apache faz o lançamento da versão da 2.0 do Hadoop, incluindo alta disponibilidade no sistema de arquivos (HDFS) e melhorias no código.

Ao ser hospedado como um projeto da Apache Software Foundation, o Hadoop segue o modelo de licenciamento da Apache, bem mais flexível que outras modalidades de licença para software livre, permitindo modificações e redistribuição do código-fonte. Dessa forma, várias empresas surgiram no mercado distribuindo implementações do Hadoop. Cada uma dessas implementações normalmente acrescentam novas funcionalidades, aplicam especificidades de um nicho de mercado, ou ainda limitam-se a prestação

de serviços como implantação, suporte e treinamento. Dentre algumas empresas com estes objetivos temos a Amazon Web Service, Cloudera, Hortonworks, KarmaSphere, Pentaho e Tresada. Atualmente, a Cloudera é uma das líderes no mercado, chefiada por Douglas Cutting, um dos criadores do Apache Hadoop original.

Nota

A licença Apache exige a inclusão do aviso de direitos autorais (copyright) e termo de responsabilidade, mas não é uma licença totalmente livre, com copyleft, permitindo seu uso em um software comercial.

Arquitetura Hadoop

Os componentes chave do Hadoop são o modelo de programação MapReduce e o sistema de arquivos distribuído HDFS. Entretanto, em meio a sua evolução, novos subprojetos, que são incorporados como componentes à arquitetura Hadoop, completam a infraestrutura do *framework* para resolver problemas específicos. Uma visão simplificada dessa organização de componentes pode ser vista na **Figura 2**.

Na camada de armazenamento de dados há o sistema de arquivos distribuído Hadoop Distributed File System (HDFS), um dos principais componentes do *framework*. Já na camada de processamento de dados temos o MapReduce, que também figura como um dos principais subprojetos do Hadoop. Na camada de acesso aos dados são disponibilizadas ferramentas como

Pig, Hive, Avro, Mahout, entre outras. Estas ferramentas tendem a facilitar a análise e consulta dos dados, fornecendo uma linguagem de consulta similar às utilizadas em bancos de dados relacionais (como a SQL, por exemplo). Assim, todo um ecossistema em volta do Hadoop é criado com ferramentas que suprem necessidades específicas; por exemplo, ZooKeeper, Flume e Chukwa, que melhoraram a camada de gerenciamento. Essas ferramentas fornecem uma interface com o usuário que busca diminuir as dificuldades encontradas no manuseio das aplicações que rodam nessa plataforma.

Para funcionar, uma aplicação Hadoop exige no mínimo a utilização das ferramentas da camada de armazenamento (HDFS) e processamento (MapReduce). As demais camadas podem ser adicionadas conforme a necessidade. A seguir, cada componente é explicado em sua essência.

Componentes principais

O projeto Hadoop, em sua versão estável (a 1.0), atualmente sob a tutela da Fundação Apache, inclui os seguintes módulos, mantidos como subprojetos:

- **Hadoop Common:** contém um conjunto de utilitários e a estrutura base que dá suporte aos demais subprojetos do Hadoop. Utilizado em toda a aplicação, possui diversas bibliotecas como, por exemplo, as utilizadas para serialização de dados e manipulação de arquivos. É neste subprojeto também que são disponibilizadas as inter-

faces para outros sistemas de arquivos, tais como Amazon S3 e CloudSource;

- **Hadoop MapReduce:** implementa um modelo de programação na forma de uma biblioteca de classes especializadas no processamento de conjuntos de dados distribuídos em um aglomerado computacional. Abstrai toda a computação paralela em apenas duas funções: *Map* e *Reduce*;

- **Hadoop Distributed File System (HDFS):** um sistema de arquivos distribuído nativo do Hadoop. Permite o armazenamento e transmissão de grandes conjuntos de dados em máquinas de baixo custo. Possui mecanismos que o caracteriza como um sistema altamente tolerante a falhas.

Componentes adicionais

Além desses, há outros projetos na comunidade Apache que adicionam funcionalidades ao Hadoop, como:

- **Ambari:** ferramenta baseada na Web para o suporte, gerenciamento e monitoramento de outros módulos Hadoop, como HDFS, MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig e Sqoop;
- **Avro:** sistema de serialização de dados;
- **Cassandra:** banco de dados escalável, com tolerância a falhas;
- **Flume e Chukwa:** sistemas que tratam da coleta de ocorrências (*logs*) para o monitoramento do Hadoop;
- **HBase:** banco de dados escalável e distribuído que suporta o armazenamento de dados estruturados para grandes tabelas;
- **Hive:** infraestrutura de *data warehouse* que fornece sumarização de dados e consultas *ad hoc*;
- **Mahout:** sistema para desenvolvimento de aplicações de aprendizagem de máquina e biblioteca com funções de mineração de dados;
- **Pig:** fornece uma linguagem de consulta de alto nível (PigLatin) orientada a fluxo de dados, e uma estrutura de execução para computação paralela;
- **ZooKeeper:** serviço de coordenação de alto desempenho para aplicações distribuídas.

Funcionamento da arquitetura básica

O Hadoop fornece uma arquitetura para que aplicativos MapReduce funcionem

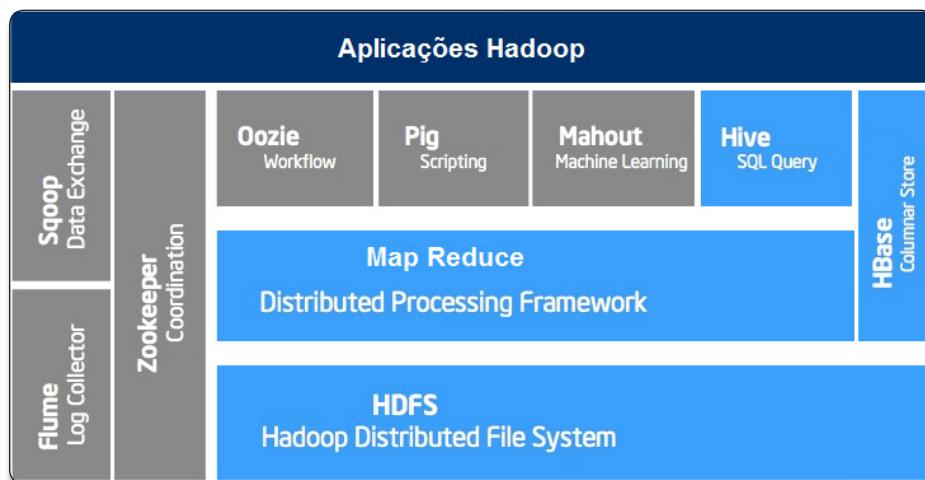


Figura 2. Componentes da arquitetura Hadoop (versão 1.x)

de forma distribuída em um cluster de máquinas, organizadas em uma máquina *mestre* e várias *escravo*. Para simplificar o desenvolvimento dessas aplicações, é possível instalar e executar o framework no modo simplificado, utilizando apenas uma máquina (que irá simular um ambiente paralelizável/distribuído).

Para que o Hadoop funcione, é necessário cinco processos: NameNode, DataNode, SecondaryNameNode, JobTracker e TaskTracker. Os três primeiros são integrantes do modelo de programação MapReduce, e os dois últimos do sistema de arquivo HDFS. Os componentes NameNode, JobTracker e SecondaryNameNode são únicos para toda a aplicação, enquanto que o DataNode e Job Tracker são instanciados para cada máquina do *cluster*.

Considerando os dois principais componentes do Hadoop (MapReduce e HDFS), a arquitetura básica será explicada a seguir.

HDFS (Hadoop Distributed File System)

Um sistema de arquivos distribuído é responsável pela organização, armazenamento, localização, compartilhamento e proteção de arquivos que estão distribuídos em computadores de uma rede. Em sistemas distribuídos, quando mais de um usuário tenta gravar um mesmo arquivo simultaneamente, é necessário um controle da concorrência (acesso simultâneo ao mesmo recurso) para que haja uma operação atômica dos processos a fim de garantir a consistência das informações. Neste caso, um sistema de arquivos distribuídos deve garantir a atomicidade nas operações de leitura, escrita, criação ou remoção de um arquivo, de forma transparente para quem manipula os dados, como se fosse similar a um sistema de arquivos local.

Nota

Um sistema de arquivos é um componente do sistema operacional que permite ao usuário interagir com os arquivos e diretórios, seja para salvar, modificar ou excluir arquivos e diretórios (pastas), bem como instalar, executar ou configurar programas. Um sistema de arquivos distribuído faz tudo isso, mas em um ambiente de rede, onde os arquivos estão fisicamente espalhados em máquinas distintas. Para quem usa tais arquivos, o sistema deve permitir as mesmas facilidades de um sistema de arquivos local.

O HDFS atua como um sistema de arquivos distribuído, localizado na camada de armazenamento do Hadoop, sendo otimizado para alto desempenho na leitura e escrita de grande arquivos (acima dos gigabytes) que estão localizados em computadores (nós) de um *cluster*.

Dentre as características do HDFS estão a escalabilidade e disponibilidade graças à replicação de dados e tolerância a falhas. O sistema se encarrega de quebrar os arquivos em partes menores, normalmente blocos de 64MB, e replicar os blocos um número configurado de vezes (pelo menos três cópias no modo *cluster*, e um no modo *local*) em servidores diferentes, o que torna o processo tolerante a falhas, tanto em hardware quanto em software. O fato é que cada servidor tem um grande número de elementos com uma probabilidade de falha, o que significa que sempre haverá algum componente do HDFS falhando. Por serem críticas, falhas devem

ser detectadas de forma rápida e eficientemente resolvidas a tempo de evitar paradas no sistema de arquivos do Hadoop.

A arquitetura do HDFS é estruturada em *master-slave* (mestre-escravo), com dois processos principais, que são:

- **Namenode:** responsável por gerenciar os dados (arquivos) armazenados no HDFS, registrando as informações sobre quais *datanodes* são responsáveis por quais blocos de dados de cada arquivo, organizando todas essas informações em uma tabela de metadados. Suas funções incluem mapear a localização, realizar a divisão dos arquivos em blocos, encaminhar os blocos aos nós escravos, obter os metadados dos arquivos e controlar a localização de suas réplicas. Como o NameNode é constantemente acessado, por questões de desempenho, ele mantém todas as suas informações em memória. Ele integra o sistema HDFS e fica localizado no nó mestre da aplicação, juntamente com o JobTracker;
- **Datanode:** responsável pelo armazenamento do conteúdo dos arquivos nos computadores escravos. Como o HDFS é um sistema de arquivos distribuído, é comum a existência de diversas instâncias de DataNode em uma aplicação Hadoop, permitindo que os arquivos sejam particionados em blocos e então replicados em máquinas diferentes. Um DataNode poderá armazenar múltiplos blocos, inclusive de diferentes arquivos, entretanto, eles precisam se reportar constantemente ao NameNode, informando-o sobre as operações que estão sendo realizadas nos blocos.

MapReduce

O MapReduce é um modelo computacional para processamento paralelo das aplicações. Ele abstrai as dificuldades do trabalho com dados distribuídos, eliminando quaisquer problemas que o compartilhamento de informações pode trazer em um sistema dessa natureza. Consiste das seguintes funções:

- **Map:** Responsável por receber os dados de entrada, estruturados em uma coleção de pares chave/valor. Tal função *map* deve ser codificada pelo desenvolvedor, através de programas escritos em Java ou em linguagens suportadas pelo Hadoop;
- **Shuffle:** A etapa de shuffle é responsável por organizar o retorno da função Map, atribuindo para a entrada de cada Reduce todos os valores associados a uma mesma chave. Este etapa é realizada pela biblioteca do MapReduce;
- **Reduce:** Por fim, ao receber os dados de entrada, a função Reduce retorna uma lista de chave/valor contendo zero ou mais registros, semelhante ao Map, que também deve ser codificada pelo desenvolvedor.

A arquitetura do MapReduce segue o mesmo princípio *master-slave*, necessitando de três processos que darão suporte à execução das funções *map* e *reduce* do usuário, a saber:

- **JobTracker:** recebe a aplicação MapReduce e programa as tarefas *map* e *reduce* para execução, coordenando as atividades nos TaskTrackers. Sua função então é designar diferentes nós para processar as tarefas de uma aplicação e monitorá-las enquanto estiverem em execução. Um dos objetivos do monitoramento é,

Hadoop: fundamentos e instalação

em caso de falha, identificar e reiniciar uma tarefa no mesmo nó, ou, em caso de necessidade, em um nó diferente;

- **TaskTracker:** processo responsável por executar as tarefas de map e reduce e informar o progresso das atividades. Assim como os DataNodes, uma aplicação Hadoop é composta por diversas instâncias de TaskTrackers, cada uma em um nó escravo. Um TaskTracker executa uma tarefa *map* ou uma tarefa *reduce* designada a ele. Como os TaskTrackers rodam sobre máquinas virtuais, é possível criar várias máquinas virtuais em uma mesma máquina física, de forma a explorar melhor os recursos computacionais;
- **SecondaryNameNode:** utilizado para auxiliar o NameNode a manter seu serviço, e ser uma alternativa de recuperação no caso de uma falha do NameNode. Sua única função é realizar pontos de checagem (*checkpointing*) do NameNode em intervalos pré-definidos, de modo a garantir a sua recuperação e atenuar o seu tempo de reinicialização.

Na **Figura 3** observa-se como os processos da arquitetura do Hadoop estão interligados, organizados em nós mestre e escravos. O mestre contém o NameNode, o JobTracker e possivelmente o SecondaryNameNode. Já a segunda camada, constituída de nós escravos, comporta em cada uma de suas instâncias um TaskTracker e um DataNode, vinculados respectivamente ao JobTracker e ao NameNode do nó mestre. Uma tarefa (*task*) que roda em um nó escravo pode ser tanto de uma função *map* quanto de uma função *reduce*.

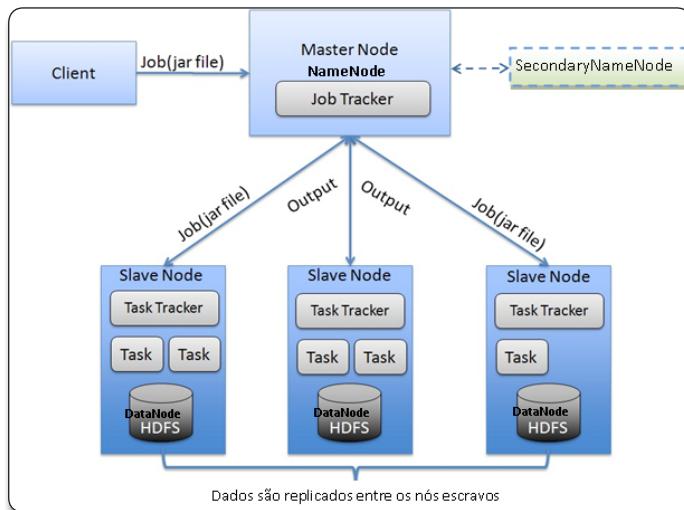


Figura 3. Funcionamento da arquitetura básica do Hadoop

Instalação do ambiente Hadoop

O Hadoop possui três formas de instalação e execução da plataforma:

- **Modo Local ou Independente:** Por padrão, o Hadoop foi configurado para executar em modo independente não distribuído. Esse modo é útil para desenvolver e testar um aplicativo;
- **Modo Pseudo distribuído:** Pode executar em um único nó em modo pseudo distribuído. Nesse caso, cada instância de processo Hadoop executa como um processo Java diferente;

- **Modo Totalmente distribuído:** O Hadoop é configurado em *cluster* com máquinas físicas (ou virtualizadas), cada qual com um endereço IP válido.

Na prática, é possível alternar entre essas configurações bastando que se editem as propriedades relacionadas em três arquivos: *core-site.xml*, *hdfs-site.xml* e *mapred-site.xml*. A seguir, é realizada a instalação do modo padrão (local), e no final é demonstrado o funcionamento da plataforma com a execução de uma aplicação exemplo.

Configuração do ambiente no modo local

O Hadoop está disponível como pacote *open-source* no portal da Apache (ver seção **Links**). Neste endereço você encontra a versão mais estável (1.2.x), a versão preliminar de atualização (a 2.x) e todas as versões anteriores a mais estável. Entretanto, há no mercado versões que empacotam todo o ambiente de execução, bem como as configurações da plataforma operacional para a maioria das distribuições Linux atualmente em uso (Ubuntu, CentOS, RedHat, etc.). Entre essas versões comerciais, porém gratuitas, uma das mais utilizadas é a oferecida pela empresa Cloudera (ver seção **Links**), denominada CDH (atualmente nas versões 3.x e 4.x), que pode ser instalada em uma máquina virtual (VMware, por exemplo), baseada no Linux CentOS.

Para efeito de demonstração, foi escolhida a instalação padrão do projeto Apache. Neste caso, é necessário verificar se a instalação Linux escolhida está configurada com os pacotes Java e SSH. Para os exemplos deste artigo, foi utilizado o seguinte ambiente: Linux Ubuntu 12.x, Java JDK 1.7, Apache Hadoop-1.2.1.

Após a definição e ativação do sistema operacional Linux, é recomendado criar um usuário e um grupo dedicados para o Hadoop, executando os comandos vistos na **Listagem 1**.

Listagem 1. Comandos para criação de usuário e grupo.

```
// Criando grupo com o nome "hadoopgrupo":  
$ sudo addgroup hadoopgrupo  
  
// Adicionado o usuário "hadoop" ao grupo:  
$ sudo adduser --ingroup hadoopgrupo hadoop
```

Em seguida, providencie uma chave de autenticação SSH. Isso se faz necessário porque o Hadoop utiliza conexão segura (SSH) para gerenciar seus nós (mestre e escravos). No caso de conexão local (localhost), não há necessidade de senha na configuração do SSH. Os comandos vistos na **Listagem 2** mostram como criar a chave de conexão SSH para o localhost.

Certifique-se que o Hadoop esteja instalado na versão definida neste artigo (a 1.2.x). Uma forma de instalar o Hadoop no Linux é baixando o pacote a partir de um endereço homologado pela Apache. Em nosso exemplo, a opção foi oferecida pela Unicamp. Os comandos para esse tipo de instalação podem ser vistos na **Listagem 3**.

Listagem 2. Comandos para criação da chave SSH.

```
$ ssh-keygen -t rsa -P ""  
$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

Listagem 3. Comandos para instalação do pacote Hadoop.

```
$ cd /usr/local  
$ sudo wget http://ftp.unicamp.br/pub/apache/hadoop/core/hadoop-1.2.1/  
hadoop-1.2.1.tar.gz  
$ sudo tar xzf hadoop-1.2.1.tar.gz  
$ sudo ln -s hadoop-1.2.1 hadoop  
$ sudo chown -R hadoop:hadoopgrupo hadoop-1.2.1
```

Logo após, adicione a variável de ambiente da instalação Java (variável `$JAVA_HOME`) com os comandos (confirme os locais e versões do Java) mostrados na **Listagem 4**.

A instalação exige que se crie um diretório que servirá de apoio ao armazenamento temporário dos arquivos de uma aplicação cliente, que serão manipulados no HDFS. Para isso, crie esse diretório com o nome `tmp`, executando os comandos da **Listagem 5**.

Agora, atualize a referência ao caminho do diretório no arquivo de configuração, editando o arquivo `core-site.xml`, como é visto na **Listagem 6**.

Listagem 4. Configuração da variável de ambiente para o compilador Java.

```
$sudo nano /usr/local/hadoop/conf/hadoop-env.sh  
export JAVA_HOME=/usr/lib/jvm/java-7-oracle
```

Listagem 5. Diretório "tmp" para apoio ao ambiente HDFS.

```
$sudo mkdir /home/hadoop/tmp  
$sudo chown hadoop:hadoopgrupo /home/hadoop/tmp  
$sudo chmod 750 /home/hadoop/tmp
```

Listagem 6. Editando o arquivo de configuração core-site.xml.

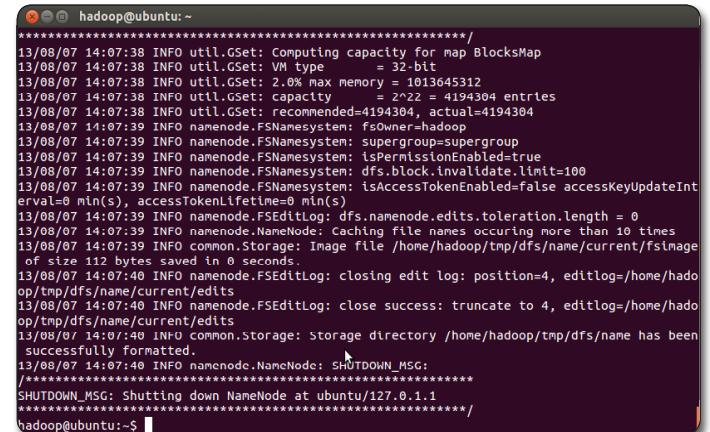
```
$sudo nano /usr/local/hadoop/conf/core-site.xml
```

Neste arquivo de configuração, adicione o código visto na **Listagem 7** entre as tags `<configuration>` e `</configuration>`. Dentro dessa marcação XML ficam todas as propriedades de configurações básicas do Hadoop, como a localização da pasta de arquivos temporários e o host do serviço do HDFS (`hdfs://localhost:54310`).

Do mesmo modo, edite o arquivo de configuração `hdfs-site.xml`, como visto na **Listagem 8**.

No código deste arquivo, adicione o conteúdo mostrado na **Listagem 9**. Na propriedade `dfs.replication` você estabelece o fator de replicação padrão dos blocos de dados no HDFS. Esse valor define o número de réplicas que serão criadas e, normalmente, em uma instalação local do Hadoop (usada neste artigo), assume o valor “1”.

Feito isso, formate o sistema de arquivos para inicializar as pastas do nó principal do HDFS e permitir o uso do MapReduce, conforme os comandos da **Listagem 10**. O resultado desse processo é visto na **Figura 4**.



```
*****  
13/08/07 14:07:38 INFO util.GSet: Computing capacity for map BlocksMap  
13/08/07 14:07:38 INFO util.GSet: VM type = 32-bit  
13/08/07 14:07:38 INFO util.GSet: 2.0% max memory = 1013645312  
13/08/07 14:07:38 INFO util.GSet: capacity = 2^22 = 4194304 entries  
13/08/07 14:07:38 INFO util.GSet: recommended=4194304, actual=4194304  
13/08/07 14:07:39 INFO namenode.FSNamesystem: fsOwner=hadoop  
13/08/07 14:07:39 INFO namenode.FSNamesystem: supergroup=supergroup  
13/08/07 14:07:39 INFO namenode.FSNamesystem: isPermissionEnabled=true  
13/08/07 14:07:39 INFO namenode.FSNamesystem: dfs.block.invalidate.limit=100  
13/08/07 14:07:39 INFO namenode.FSNamesystem: isAccessTokenEnabled=false accessTokenUpdateInterval=  
13/08/07 14:07:39 INFO namenode.FSNamesystem: edits.toleration.length = 0  
13/08/07 14:07:39 INFO namenode: Caching file names occurring more than 10 times  
13/08/07 14:07:39 INFO common.Storage: Image file /home/hadoop/tmp/dfs/name/current/fsimage  
of size 112 bytes saved in 0 seconds.  
13/08/07 14:07:40 INFO namenode.FSEditLog: closing edit log: position=4, editlog=/home/hadoop/tmp/dfs/name/current/edits  
13/08/07 14:07:40 INFO namenode.FSEditLog: close success: truncate to 4, editlog=/home/hadoop/tmp/dfs/name/current/edits  
13/08/07 14:07:40 INFO common.Storage: Storage directory /home/hadoop/tmp/dfs/name has been  
successfully formatted.  
13/08/07 14:07:40 INFO namenode.NameNode: SHUTDOWN_MSG:  
*****  
SHUTDOWN_MSG: Shutting down NameNode at ubuntu/127.0.1.1  
*****  
hadoop@ubuntu:~$
```

Figura 4. Tela com as mensagens da formatação do NameNode

Listagem 7. Atualizando a configuração do arquivo “core-site.xml”.

```
<configuration>  
  
<property>  
  <name>hadoop.tmp.dir</name>  
  <value>/home/hadoop/tmp</value>  
  <description>A base for other temporary directories.</description>  
</property>  
  
<property>  
  <name>fs.default.name</name>  
  <value>hdfs://localhost:54310</value>  
  <description>The name of the default file system. A URI whose  
  scheme and authority determine the FileSystem implementation. The  
  uri's scheme determines the config property (fs.SCHEME.impl) naming  
  the FileSystem implementation class. The uri's authority is used to  
  determine the host, port, etc. for a filesystem.</description>  
</property>  
  
</configuration>
```

Listagem 8. Editando o arquivo hdfs-site.xml.

```
$ sudo nano /usr/local/hadoop/conf/hdfs-site.xml
```

Listagem 9. Configuração do arquivo hdfs-site.xml.

```
<configuration>  
<property>  
  <name>dfs.replication</name>  
  <value>1</value>  
  <description>Default block replication.  
  The actual number of replications can be specified when the file is created.  
  The default is used if replication is not specified in create time.  
</description>  
</property>  
</configuration>
```

Listagem 10. Formatação do sistema de arquivos.

```
$su - hadoop  
$ /usr/local/hadoop/bin/hadoop namenode -format
```

O passo seguinte é preparar o Hadoop para ser utilizado, levantando todos os seus processos na memória. Para isso, execute o arquivo de comandos `start-all.sh`, confirmando o processo no final com “yes”. Depois, para verificar se tudo foi corretamente

Hadoop: fundamentos e instalação

executado, chame o comando *jps*. Veja a sequência das instruções na **Listagem 11** e o resultado na **Figura 5**.

Teste do ambiente Hadoop

Para validar o ambiente, o Hadoop disponibiliza o clássico exemplo de contar palavras (WordCount), que ilustra de forma didática a execução de uma aplicação MapReduce. Esse exemplo utiliza como entrada de dados um conjunto de arquivos texto, a partir dos quais a frequência das palavras será contada. Como saída, será gerado outro arquivo texto contendo cada palavra e a quantidade de vezes que cada uma foi encontrada.

Para ilustrar esse processamento, baixe uma coleção de dados sobre livros da biblioteca Gutenberg (ver seção **Links**), cujo conteúdo está disponível para uso livre em diversos formatos (HTML, PDF, TXT, ePub, entre outros). Neste caso, foi feito o download do arquivo texto *pg20417.txt* para uma pasta chamada *gutenberg*, conforme demonstra os comandos da **Listagem 12**.

```
hadoop@ubuntu:~$ jps
8309 TaskTracker
7785 DataNode
7575 NameNode
8353 Jps
8006 SecondaryNameNode
8090 JobTracker
```

Figura 5. Processos do Hadoop em execução

Listagem 11. Inicialização do Hadoop.

```
$ /usr/local/hadoop/bin/start-all.sh
$ jps
```

Listagem 12. Download do arquivo de entrada.

```
$ mkdir gutenberg
$ lynx -dump http://www.gutenberg.org/cache/epub/20417/pg20417.txt >
gutenberg/pg20417.txt
```

Nota

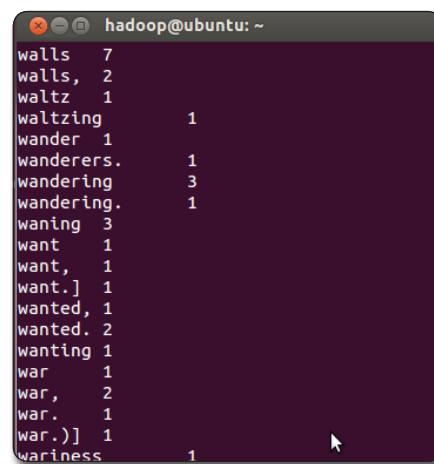
Para evitar a necessidade de digitar o caminho completo do Hadoop até os arquivos executáveis no momento de executar um comando, configuraremos a variável de ambiente PATH. Considerando que o caminho (diretório) onde estão localizados os comandos (arquivos executáveis) é o diretório de instalação (*/usr/local/hadoop/bin*), abra o arquivo *bashrc* (use *nano ~/.bashrc*) e acrescente esse caminho à variável PATH usando o código: *export PATH=\$PATH:/usr/local/hadoop/bin*.

Em seguida, usando o comando de cópia de pasta (*copyFromLocal*) do sistema HDFS, transfira o conteúdo da pasta que contém o arquivo de palavras (*pg20417.txt*) ao ambiente Hadoop. Para isto, execute: *hadoop dfs -copyFromLocal gutenberg gutenberg*.

Feito isso, abra a pasta do Hadoop e execute o exemplo WordCount (distribuído no pacote do framework), como demonstra a **Listagem 13**. Observe que o caminho */user/hadoop/gutenberg* representa a entrada (*in*), e */user/hadoop/gutenberg-output*, a pasta com a saída do processo *reduce*.

No final do MapReduce é produzido um arquivo (denominado *part-r-00000*) que armazena a contagem de palavras presentes nos

arquivo da pasta de entrada do Hadoop. Para ver esse conteúdo, execute o comando *hadoop dfs -cat /user/hadoop/gutenberg-output/part-r-00000*. A **Figura 6** apresenta um trecho do resultado.



```
walls 7
walls, 2
waltz 1
waltzing 1
wander 1
wanderers. 1
wandering 3
wandering. 1
waning 3
want 1
want, 1
want.] 1
wanted, 1
wanted. 2
wanting 1
war 1
war, 2
war. 1
war.)] 1
wariness 1
```

Figura 6. Trecho do resultado do processo de contagem de palavras

Listagem 13. Executando as funções map e reduce do contador de palavras.

```
$ cd /usr/local/hadoop
$ hadoop jar hadoop-examples-1.2.1.jar wordcount /user/hadoop/gutenberg
/usr/hadoop/gutenberg-output
```

Conclusão

Este artigo apresentou a ferramenta Hadoop como proposta para o processamento de grandes conjuntos de dados, que aqui chamamos de “Bigdata”. A ideia principal do funcionamento dessa ferramenta está no uso da técnica MapReduce, que permite a análise e tratamento desses dados facilitando a construção de aplicações que sigam o modelo previsto em duas funções, uma para o *map* e outra para o *reduce*.

Definidas as funções do MapReduce, Hadoop realiza o processamento distribuído em um conjunto (*cluster*) de computadores de baixo custo. O modelo de programação e a infraestrutura disponível na arquitetura MapReduce se encarregam de partitionar e distribuir os dados de entrada, escalar as execuções das funções *map* e *reduce* em máquinas distintas, tratar as falhas e a comunicação entre essas máquinas. Para assegurar a integridade e o controle dos dados na rede, Hadoop também apresenta um sistema de arquivos distribuído, o HDFS, fundamental para o funcionamento da arquitetura. O HDFS fornece os mecanismos que garantem a transparência na manipulação dos arquivos, com segurança e alto desempenho.

As características supracitadas constituem o sucesso da tecnologia, mas a sua adoção é restrita ao domínio de problemas que possam ser formulados e resolvidos dentro do contexto do paradigma; no caso, os dados devem estar organizados em uma coleção do tipo chave/valor e o processamento deve ser dividido em duas funções que se complementam, o *map* e o *reduce*.

Apesar de fazer parte de um projeto de código aberto, mantido pela comunidade Apache, Hadoop mostra ser, em pouco tempo de vida, uma tecnologia com maturidade e confiabilidade. Prova disso é a decisão que várias empresas de tecnologia da informação fizeram ao adotá-la para resolver seus problemas. Além de usar essa tecnologia, gigantes como IBM, Google, Yahoo! e Oracle apostam e investem em projetos relacionados ao Hadoop e MapReduce.

Autor



Cláudio Martins

claudiomartins2000@gmail.com

É Mestre em Computação pela Universidade Federal do Rio Grande do Sul (UFRGS), professor do Instituto Federal do Pará (IFPA), e analista de sistemas da Companhia de Informática de Belém (Cinbesa). Trabalha há dez anos com a plataforma Java.



Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Links e Referências:

Página oficial do projeto Apache Hadoop.

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

Página oficial do projeto Lucene.

<http://lucene.apache.org>

Artigo que apresenta a técnica MapReduce.

<http://www.devmedia.com.br/fundamentos-sobre-mapreduce/28644>

Artigo que explora os principais recursos da arquitetura do HDFS.

<http://www.ibm.com/developerworks/br/library/wa-introhdfs/>

Site da distribuição Hadoop oferecido pela Cloudera, em sua versão comunitária (aberta).

<http://www.cloudera.com/content/support/en/downloads.html>

Endereço do Projeto Gutenberg para livros eletrônicos grátis em língua portuguesa.

<http://www.gutenberg.org/browse/languages/pt>

Livros

Hadoop: The Definitive Guide - 3rd Edition. Tom White. O'Reilly. 2012.

O livro aborda o tema Hadoop de forma didática e atualizada em sua atual versão (2.x), apresentando estudos de caso usados para resolver problemas no modelo mapreduce.

Conhecimento faz diferença!



The magazine cover features a large red starburst graphic containing the text '+ de 290 vídeos para assinantes' (Over 290 videos for subscribers). The background shows a robotic arm performing a task.

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

Faça um upgrade em sua carreira

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



DEV MEDIA

Conheça os recursos da Expression Language 3.0

A JSR 341 aumenta ainda mais o poder das linguagens de expressões para a plataforma Java EE 7

No último mês de junho foi lançada a Java EE 7 trazendo uma série de novidades para a plataforma *enterprise*. Dentro desse pacote está a nova versão 3.0 da Expression Language (ou simplesmente EL).

A EL 3.0 foi especificada pela JSR-341 e teve Kin-man Chung como líder do *Expert Group*. Esta é a primeira vez que a EL é tratada por uma JSR (*Java Specification Request*) independente. Nas versões anteriores ela fazia parte da JSR do JSTL e, mais recentemente, do JSP. Na última versão a EL já havia começado a dar sinais de uma iminente separação quando foi publicada em uma especificação apartada, porém ainda dentro da mesma JSR do JSP. Essa mudança reflete a visão do Expert Group de que a EL é útil por si só e mostra o quanto ela se tornou importante dentro da Java EE, inclusive sendo integrada a outras tecnologias fora do contexto web, como Bean Validation e CDI.

A nova versão da EL contempla adições significativas na linguagem de expressões, principalmente por conta do suporte a expressões lambda e das funcionalidades para manipulação de coleções inspiradas no modelo de programação funcional que está sendo incorporado à Java SE 8, cujo lançamento está previsto para 2014. A partir de agora os desenvolvedores poderão ordenar, filtrar e transformar coleções de forma bastante simplificada utilizando a EL e expressões lambda.

Outro ponto bastante importante é que a EL 3.0 desacopla a linguagem de expressões das tecnologias web (JSP e JSF) e cria um grande potencial ao trazer suas funcionalidades para fora do container Java EE através da API *standalone*. Com essa nova API os desenvolvedores poderão escrever e resolver expressões EL dentro do bom e velho código Java.

Além dessas principais mudanças, a EL 3.0 traz outros recursos e melhorias à linguagem de expressões a fim de atender pedidos antigos da comunidade e maximizar a

Resumo DevMan

Porque este artigo é útil:

A EL é uma linguagem de expressões utilizada na criação de páginas web dinâmicas na plataforma Java EE. A versão recém-lançada (3.0) introduziu diversas melhorias e funcionalidades que tornam a linguagem mais poderosa e produtiva. Entre as principais novidades estão o suporte a expressões lambda, operações para manipulação de coleções e a API standalone, a qual leva toda a capacidade da EL também para fora do container web. O artigo aborda com detalhes esses e outros assuntos e apresenta exemplos de código que mostram o uso de cada funcionalidade.

O tema é útil para desenvolvedores Java que utilizam a Expression Language no dia a dia para construir páginas web dinâmicas utilizando puramente JSPs e/ou frameworks web como o JSF. Também pode ser útil para aqueles que enxergavam potencial na EL, mas nunca puderam utilizá-la dentro do código Java.

produtividade dos desenvolvedores. Entre essas melhorias estão a introdução de novos operadores (incluindo um para concatenação de Strings) e o acesso a construtores e membros estáticos de classes Java.

Este artigo é mais um da série que a Java Magazine está publicando para levar ao leitor as novidades da Java EE 7. Nele veremos um pouco do histórico e da importância da linguagem de expressões no contexto do desenvolvimento web e em seguida mergulharemos nas novas *features* incorporadas na EL 3.0. O código completo dos exemplos apresentados neste artigo está disponível no GitHub (veja a seção **Links**).

Para executar os exemplos e testar as features por conta própria, baixe e instale o servidor GlassFish 4 e também uma IDE de sua preferência que ofereça suporte à Java EE 7. O NetBeans é uma das IDEs que já oferecem suporte e, além disso, traz alguns exemplos de uso da EL 3.0 e de outras tecnologias da Java EE 7. Estando tudo preparado, mãos à obra!

Motivação e histórico

Quem já trabalhou com desenvolvimento de aplicações web em Java, seja utilizando puramente Servlets + JSPs ou mesmo com frameworks MVC *server-side*, como Struts, Spring MVC, VRaptor, JSF, entre outros, muito provavelmente já utilizou a Expression Language da Java EE para produzir páginas web dinâmicas.

A Expression Language é uma linguagem de programação simples que contém operadores, uma sintaxe própria e palavras reservadas. Ela nasceu como parte da versão 1.0 da JSTL (*Java-Server Pages Standard Tag Library*) e posteriormente foi adotada e expandida pela especificação JSP 2.0. Seu objetivo era fornecer uma maneira fácil de construir páginas dinâmicas, acessando e manipulando objetos Java, sem que para isso fosse necessário escrever *scriptlets*. Para quem não se lembra (melhor nem lembrar mesmo!), scriptlets são aqueles trechos de código Java criados dentro das páginas JSPs entre tags <%...%> para gerar HTML dinamicamente e que tornavam as aplicações muito difíceis de entender e de manter, principalmente para *designers de front-end* que não conheciam código Java. Por isso os scriptlets são considerados uma má prática há bastante tempo. Para efeitos de comparação, a **Listagem 1** mostra dois trechos de código, com scriptlet e EL, que fazem exatamente a mesma coisa.

Listagem 1. Scriptlet vs. EL.

```
<!-- Scriptlet -->
<%>
Pessoa p = (Pessoa) request.getAttribute("pessoa");
out.print(p.getNome());
%>

<!-- EL -->
${pessoa.nome}
```

Com o passar do tempo a tecnologia JSF (*JavaServer Faces*) nasceu e, por sua vez, precisando de uma linguagem de expressões. Na época, a EL do JSP 2.0 possuía algumas limitações e não atendia aos requisitos exigidos pelo novo framework. Os principais problemas encontrados estavam ligados ao fato de que as expressões EL eram imediatamente avaliadas e resolvidas pelo *container* web durante o processamento da página, enquanto o JSF precisava que a avaliação das expressões fosse adiada para fases intermediárias do seu ciclo de vida. Além disso, a EL não permitia alterar valores nem invocar métodos em objetos Java server-side, um requisito essencial do JSF. Resultado da história: o JSF criou uma variação da linguagem de expressões, mais robusta e com as funcionalidades necessárias para o funcionamento de seus componentes. Problema resolvido, certo? Só que não, pois isso gerou um problema para os desenvolvedores: quando as duas linguagens eram usadas em conjunto (tags JSF + tags JSTL), elas conflitavam. Por conta dessas incompatibilidades, foi necessário um trabalho de unificação das duas linguagens de expressão, que resultou no lançamento da EL 2.1 (conhecida como *Unified Expression Language*) como uma especificação independente na Java EE 5, porém dentro da mesma JSR-245 do JSP 2.1.

Novamente, vários anos se passaram até que em junho de 2013 a Java EE 7 foi lançada. Dentro desse pacote de novas especificações e atualizações está uma nova versão da Expression Language (a 3.0), a qual o leitor irá conhecer a partir de agora.

Suporte a expressões lambda

De maneira bem simplista, uma expressão lambda é função anônima que pode conter qualquer número de parâmetros e retornar ou não um valor. Uma função anônima pode ser atribuída a uma variável ou passada como argumento de métodos que aceitam outras funções como parâmetro. Um exemplo clássico disso seria passar uma função anônima que conhece como ordenar elementos de uma coleção para um método de ordenação. Lambda é um conceito originado na matemática, comum em linguagens funcionais e que está sendo adicionado na próxima versão da linguagem Java (Java SE 8). Esse recurso tornará a vida dos programadores um pouco mais simples ao possibilitar a definição de comportamentos, que hoje normalmente são criados com classes anônimas (implementações de Comparators, Runnables, etc.), como uma função anônima dentro da chamada do próprio método.

A EL 3.0 se adiantou ao lançamento do Java 8 e incluiu suporte a expressões lambdas já nessa versão. Aliás, de acordo com o líder da especificação, sua equipe trabalhou bem próxima à equipe do Java 8 para que a sintaxe fosse a mesma que será utilizada no Java.

A sintaxe de expressões lambda no Java 8 é bastante simples e alguns exemplos podem ser vistos na **Listagem 2**.

Listagem 2. Sintaxe de expressões lambda no Java SE 8.

1. $x \rightarrow x + 1$
2. $(x, y) \rightarrow x + y$
3. $() \rightarrow \text{true}$

Em uma expressão lambda, todos os elementos do lado esquerdo do operador “ \rightarrow ” são parâmetros da função anônima. Na **linha 1** não foi necessário utilizar parênteses, pois a função tem apenas um parâmetro. Caso a função possua nenhum ou mais de um parâmetro, como é o caso nas **linhas 2 e 3**, é necessário utilizar parênteses. A expressão colocada do lado direito do sinal “ \rightarrow ” é o próprio corpo da função anônima, ou seja, o comportamento que ela irá executar.

Nos exemplos da **Listagem 2** apenas declaramos algumas funções, mas não chegamos a utilizá-las. As funções anônimas podem ser invocadas de duas formas. A primeira delas é chamando a função imediatamente logo após declará-la. No exemplo a seguir, implementamos uma função que inverte o sinal de um número e, em seguida, a invocamos passando o número 9 como argumento:

```
((x) -> -1 * x)(9) // resultado: -9
```

A segunda alternativa é invocar a função indiretamente a partir de uma variável para a qual a função anônima foi atribuída.

Conheça os recursos da Expression Language 3.0

No exemplo a seguir, atribuímos uma função que transforma os caracteres de uma **String** em caixa alta para a variável “upper”. Na sequência, invocamos a função utilizando o próprio nome da variável e passando a **String** “Java” como argumento.

```
upper = (s) -> s.toUpperCase(); upper("Java") // resultado: JAVA
```

As funções anônimas também podem ser invocadas de forma recursiva. No próximo exemplo fazemos isso para calcular o oitavo termo da sequência Fibonacci. Observe o código:

```
fib = n -> n < 2 ? n : (fib(n-1) + fib(n-2)); fib(8) // resultado: 21
```

Outra forma de usar uma função anônima é passá-la como argumento de outra função. O próximo exemplo mostra isso, ao passar uma função anônima que compara dois números de uma coleção para o método **sorted()** que, por sua vez, ordena a coleção. Não se preocupe em entender essa sintaxe, pois isso será visto em detalhes um pouco mais à frente, quando abordarmos as funções para manipulação de coleções da EL 3.0. Atente-se apenas ao fato de que, neste exemplo, estamos passando uma função anônima para outra função. Perceba que até a Java SE 7 teríamos que criar uma classe anônima que implementasse a interface **Comparator** para fazer esse tipo de coisa.

```
[2, 3, 1].stream().sorted((a, b) -> a - b).toList() // resultado: [1, 2, 3]
```

Embora todos os exemplos dados até agora tenham sido apenas para demonstrar a sintaxe das expressões lambda no Java SE, esses mesmos exemplos funcionarão com a EL 3.0! Faça o teste: crie uma página JSP dentro de uma aplicação web e coloque cada um dos trechos de código citados anteriormente dentro da construção \${expressão}. Em seguida, implante a aplicação no servidor GlassFish e acesse a página. O mesmo pode ser feito com páginas JSF e com a API standalone, a qual será analisada logo mais à frente neste artigo. O resultado pode ser visto na **Figura 1**.

Descrição	Expressão	Resultado
Invocação imediata	<code>\$((x) -> -1 * x) (9)</code>	-9
Atribuição e invocação indireta	<code>\$(upper = (s) -> s.toUpperCase(); upper ("Java"))</code>	JAVA
Invocação recursiva	<code>\$(fib = n -> n < 2 ? n : (fib(n-1) + fib(n-2)); fib(8))</code>	21
Função anônima como argumento de método	<code>\$([2, 3, 1].stream().sorted((a, b) -> a - b).toList())</code>	[1, 2, 3]

Figura 1. Página JSP contendo expressões EL com lambdas

Embora a sintaxe das expressões lambda da EL 3.0 seja praticamente idêntica ao Java SE 8, existe uma diferença importante: o corpo da função anônima na EL também é uma “expressão EL”. Em outras palavras, isso significa que o conteúdo da função anônima

é avaliado e resolvido pelo mecanismo de processamento interno como uma expressão EL. Sendo assim, a sintaxe e palavras reservadas da EL podem ser utilizadas dentro do corpo da função. Isso pode ser visto no exemplo a seguir, no qual utilizamos o operador “gt” (*greater than*) para comparar dois números:

```
$(((n1, n2) -> n1 gt n2)(10, 5)}
```

Operações para manipulação de coleções

Um dos tópicos mais interessantes dessa nova versão da EL é o suporte a manipulação de coleções, que foi introduzido na linguagem através de um conjunto considerável de operações bastante semelhante ao modelo que estará presente no Java SE 8 e que também é encontrado em linguagens funcionais como Scala e bibliotecas como Underscore.js. São operações bastante úteis que permitem, entre outras coisas, fazer ordenação, filtragem e transformação dos elementos contidos em coleções.

As expressões lambda têm um papel fundamental nas novas operações sobre coleções da EL 3.0. A maior parte das operações recebe como argumento uma função anônima que contém a lógica específica a ser executada com cada elemento presente na coleção. Por exemplo, o método **sorted()**, que vimos em um dos exemplos anteriores, recebe uma função anônima de comparação e a utiliza para ordenar os elementos dentro da coleção.

A sintaxe das operações sobre coleções da EL segue o modelo “**fonte** → **operações intermediárias** → **operação terminadora**”. A fonte (ou *source*) nesse modelo não é exatamente o tipo da coleção em si (**Map**, **List** ou **Set**), mas sim a classe **Stream**, que é obtida através do método **stream()** invocado diretamente nas coleções. As operações intermediárias manipulam os elementos da coleção e retornam um novo objeto stream ao término do processamento. Ao final, é necessário invocar uma das operações terminadoras da API (qualquer método que não retorne um novo stream, como **get()**, **toList()**, **sum()**, entre outros). Veja a seguir algumas das principais operações definidas pela EL 3.0:

- **filter**: filtra os elementos da coleção que satisfazem a expressão booleana contida na função anônima. No exemplo a seguir, dado uma coleção de cidades, criamos uma nova lista contendo apenas aquelas que são do estado de São Paulo:

```
cidades.stream().filter(c -> c.estado eq 'SP').toList()
```

- **map**: aplica uma função anônima de transformação em cada item da coleção e mapeia o resultado para um novo stream (nova coleção). No exemplo a seguir, pegamos uma lista de objetos “Cidade” e aplicamos a função de transformação que retorna apenas o nome da cidade. O resultado final dessa transformação é uma lista contendo apenas os nomes das cidades:

```
cidades.stream().map(c -> c.nome).toList()
```

- **sorted**: ordena os elementos da coleção com base na lógica de comparação da função recebida como argumento. Se nenhuma

função de comparação for informada, utiliza a ordenação natural dos elementos. No exemplo a seguir, as cidades contidas na coleção são ordenadas pelo número de habitantes:

```
cidades.stream().sorted((a, b) -> b.populacao - a.populacao).toList()
```

- **reduce:** A operação **reduce** recebe uma função anônima contendo dois argumentos. O primeiro atua como um acumulador, enquanto o segundo representa o elemento atual da coleção. Ambos os argumentos são atualizados conforme os itens da coleção vão sendo percorridos. No caso do acumulador, o valor é computado no corpo da função anônima. O retorno da operação **reduce** é o valor final do acumulador após todos os itens terem sido percorridos. Para facilitar o entendimento, observe o exemplo a seguir: dado uma lista de cidades, queremos encontrar a cidade com a maior população. Para isso, criamos uma função anônima que, em cada iteração, compara dois elementos (o acumulador e o elemento corrente) e retorna a cidade com mais habitantes. Cada vez que uma cidade com mais habitantes é encontrada, o acumulador é atualizado e passa a referenciar essa cidade. No final, apenas a cidade com a maior população é retornada pela operação **reduce**. Veja o código:

```
cidades.stream().reduce((acum, elem) -> (acum.populacao > elem.populacao ? acum : elem)).get()
```

A maioria das operações definidas na classe **Stream** retorna um novo “stream” como resultado do processamento. Isso nos permite encadear e combinar novas operações sobre o resultado da operação anterior, formando um pipeline de transformações, para produzir um resultado mais elaborado. Para exemplificar essa possibilidade, vejamos o exemplo a seguir, no qual utilizamos as operações **filter**, **map** e **sum** de maneira combinada para obter a soma da quantidade de habitantes das cidades do estado do Paraná. Observe que o *output* das primeiras operações torna-se o *input* das operações subsequentes:

```
cidades.stream().filter(c -> c.estado eq 'PR').map(c -> c.populacao).sum()
```

A **Tabela 1** traz a lista completa das operações disponíveis na classe **Stream**. Exemplos com todas elas podem ser vistos no projeto disponível no GitHub (veja a seção **Links**).

A EL 3.0 introduziu também uma nova sintaxe para criação dinâmica de coleções. A **Listagem 3** mostra exemplos de criação de objetos **List**, **Set** e **Map** utilizando essa sintaxe.

Operação	Descrição
filter, map, sorted e reduce	Consulte os exemplos anteriores.
min	Obtém o menor elemento da coleção utilizando a função de comparação recebida como argumento. Caso não receba a função, espera que os elementos implementem a interface Comparable ou lançará uma exceção.
max	Funcionamento semelhante ao método min, porém obtém o maior elemento da coleção.
average	Calcula a média dos elementos da coleção. Funciona apenas com coleções de valores numéricos.
sum	Calcula a soma dos elementos da coleção. Funciona apenas com coleções de valores numéricos.
count	Retorna a quantidade de elementos na coleção.
distinct	Retorna um novo stream de elementos contendo apenas elementos não duplicados (utiliza o método equals() para fazer a diferenciação).
flatMap	A operação flatMap produz um único stream ao juntar/concatenar outros streams mapeados a partir dos elementos da coleção. Confuso, não? Pois bem, vamos imaginar uma lista de listas (<code>List<List<?>></code>). A operação flatMap nesse caso iria “achatar” essa estrutura bidimensional e transformá-la em uma única lista contendo os elementos de todas as subcoleções. Para facilitar ainda mais o entendimento, vejamos o exemplo: <code>\$(cidades.stream().flatMap(c -> c.bairros.stream()).toList())</code> Observe que cada elemento da coleção de cidades contém uma lista de bairros (c.bairros). O objetivo da função anônima passada como argumento é gerar uma nova lista contendo os bairros de todas as cidades.
peek	Essa operação é voltada para a realização de debug, pois permite inspecionar cada elemento do stream em determinados pontos da cadeia de operações sem interferir no resultado final.
iterator	Produz um Iterator a partir do stream.
limit	Reduz o tamanho do stream de acordo com o valor do argumento informado.
substream	Extrai uma fatia do stream original de acordo com os valores dos argumentos início e fim informados.
findFirst	Retorna o primeiro elemento do stream.
forEach	É uma operação terminadora que invoca a função anônima para cada elemento do stream.
anyMatch	Retorna um booleano indicando se pelo menos UM elemento na coleção satisfaz a condição da função anônima informada como argumento.
allMatch	Retorna um booleano indicando se TODOS os elementos na coleção satisfazem a condição da função anônima informada como argumento.
noneMatch	Retorna um booleano indicando se todos os elementos na coleção NÃO satisfazem a condição da função anônima informada como argumento.
toArray	Converte um stream de elementos para um array.
toList	Converte um stream de elementos para uma lista.

Tabela 1. Operações para manipulação de coleções com a EL 3.0

Conheça os recursos da Expression Language 3.0

Observe na **linha 1** que a sintaxe para criação de listas faz uso de um par de colchetes (“[...]”) ao redor dos elementos, enquanto que Sets e Maps (**linhas 2 e 3**) fazem uso de chaves (“{...}”). A sintaxe para Maps é um pouco diferente das primeiras pelo fato de a estrutura armazenar elementos compostos por pares chave/valor. A estrutura é idêntica a de um documento JSON: cada elemento é representado por um par de valores separados por um sinal de “：“. O primeiro refere-se à chave, enquanto o segundo ao valor de fato. Nos três casos utiliza-se vírgulas (”,”) para separação dos elementos. Além disso, como demonstra a **linha 4**, outras coleções podem ser aninhadas utilizando essa mesma sintaxe. Por fim, é válido citar que além de valores literais, qualquer expressão EL pode ser empregada na construção de coleções.

Listagem 3. Construção dinâmica de coleções.

```
1. ${list = ['um', 'dois', 'três', 'quatro']}
2. ${set = {1, 1, 2, 2, 3, 3, 4, 4}}
3. ${map = {1 : 'um', 2 : 'dois', 3 : 'três', 4 : 'quatro'}}
4. ${listips = [header.host, ['127.0.0.1', '192.168.0.1']]}
```

API standalone

Outra importante adição na EL 3.0 é a API standalone, que permite resolver expressões EL fora do container web. Isso significa que todo o poder da linguagem de expressões não está mais limitado às tecnologias web da Java EE (JSP e JSF), podendo agora ser utilizado também dentro do código Java de qualquer aplicação, mesmo standalone (Java SE). Essa adição traz inúmeras possibilidades para o desenvolvedor, pois dá a ele capacidade de manipular grafos de objetos Java em *runtime* usando expressões EL.

A principal classe da API é a **javax.el.ELProcessor**. É através dela que temos acesso aos métodos para resolver expressões, obter e atribuir valores a objetos e variáveis e definir métodos estáticos como funções EL.

A **Listagem 4** mostra um exemplo simples empregando a API. Neste exemplo instanciamos um **ELProcessor** e definimos um bean com o nome de “pessoa”. Esse último passo é necessário para tornar o objeto “pessoa” disponível no contexto do **ELProcessor** e fazer com que seus atributos sejam acessíveis pelas expressões EL. Na sequência, usamos o método **setValue()** para atribuir um valor ao atributo “idade” do bean “pessoa”. Esse trecho mostra como é possível utilizar a EL para atribuir/alterar valores de objetos em runtime. Por fim, invocamos o método **eval()** para resolver as expressões EL “pessoa.nome” e “pessoa.idade” e, assim, obter os valores dos respectivos atributos.

Listagem 4. Acessando atributos de um bean com a API standalone.

```
ELProcessor elp = new ELProcessor();
Pessoa joao = new Pessoa("João");
elp.defineBean("pessoa", joao);
elp.setValue("pessoa.idade", 30);
System.out.println(elp.eval("pessoa.nome")); // Imprime 'João'
System.out.println(elp.eval("pessoa.idade")); // Imprime '30'
```

Vale lembrar que todas as expressões EL que foram mostradas até agora neste artigo funcionam com a API standalone, inclusive aquelas que contêm expressões lambda e operações de manipulação de coleções, conforme pode ser visto na **Listagem 5**.

Listagem 5. Localizando a pessoa mais velha na coleção com a API standalone da EL 3.0.

```
ELProcessor elp = new ELProcessor();
elp.defineBean("pessoas", Arrays.asList(new Pessoa("João", 30),
                                         new Pessoa("Maria", 25),
                                         new Pessoa("José", 50)));
elp.setVariable("oldest", "p1, p2 -> p1.idade - p2.idade");
String nome = (String) elp.eval("pessoas.stream().max(oldest).get().nome");
System.out.println(nome); // Imprime 'José'
```

No exemplo da **Listagem 5**, criamos uma função anônima para encontrar a pessoa mais velha da coleção e utilizamos o método **setVariable()** para atribuir essa função a uma variável denominada “oldest”. Na penúltima linha de código passamos a função recém-criada como argumento da operação **max** (observe que tudo até esse momento é apenas uma **String**) e resolvemos a expressão chamando o método **eval()**. Poderíamos, inclusive, ter pouparido um dos passos nesse exemplo se tivéssemos definido a função anônima diretamente dentro da operação **max**. No entanto, fizemos uso do método **setVariable()** para mostrar como atribuir uma expressão EL a uma variável. As expressões atribuídas ao contexto do **ELProcessor** através desse método não são resolvidas imediatamente. Isso significa que a função “oldest”, neste caso, poderia ser reaproveitada dentro de outras expressões.

A **ELProcessor** também possibilita a definição de funções EL a partir de métodos estáticos de classes Java. As funções EL eram a forma que as versões anteriores da Expression Language ofereciam para invocar métodos Java a partir de páginas JSP sem o uso de scriptlets. No entanto, o processo de criação das funções EL era um tanto quanto complexo, principalmente por exigir a configuração de arquivos “.tld”. Veremos mais à frente que, a partir dessa versão, invocar métodos em classes Java se tornou algo bastante simples.

Para definir funções EL através da classe **ELProcessor**, utilize um dos métodos **defineFunction()**, como mostra a **Listagem 6**. Observe no exemplo que a assinatura do método **defineFunction()** recebe quatro argumentos. O primeiro e o segundo são, respectivamente, o prefixo (ou *namespace*) e o nome da função EL. Os dois últimos determinam qual a classe e qual o método estático a função se refere. A API fornece ainda uma variação do método **defineFunction()** que recebe como parâmetro um **java.lang.reflect.Method** (além, é claro, do prefixo e nome da função) com a referência exata a ser mapeado pela função.

Outra classe importante dentro da API standalone é a **ELManager**, obtida através do método **getELManager()** da classe **ELProcessor**. A classe **ELManager** é usada para configurar informações necessárias ao contexto do processador EL para resolver as expressões EL. Por exemplo, se determinada expressão EL utiliza membros estáticos ou construtores de classes não perten-

centes ao pacote `java.lang.*`, torna-se necessário importar essas dependências. A importação pode ser feita no nível de pacote, de classe e até do próprio método/atributo estático referenciado na expressão. Observe que no exemplo da **Listagem 7** importamos a classe `java.math.BigInteger` e depois instanciamos um objeto via expressão EL (não se preocupe agora com a sintaxe do construtor, pois falaremos sobre isso mais à frente).

Listagem 6. Definindo e invocando uma função EL.

```
ELProcessor elp = new ELProcessor();
elp.defineFunction("fn", "random", "java.lang.Math", "random");
System.out.println(elp.eval("fn:random()"));
```

Listagem 7. Importando classes para uso dentro de expressões EL.

```
ELProcessor elp = new ELProcessor();
elp.getELManager().importClass("java.math.BigInteger");
System.out.println(elp.eval("BigInteger('1000')"));
```

Caso o leitor tiver interesse em conhecer mais sobre a API standalone, a última versão do NetBeans traz um projeto contendo uma série de exemplos de expressões que podem ser resolvidas com a API. Para gerá-lo, vá a *File > New Project > Categories > Samples > Java EE* e escolha o projeto *Expression Language (Java EE 7)*, conforme mostrado na **Figura 2**.

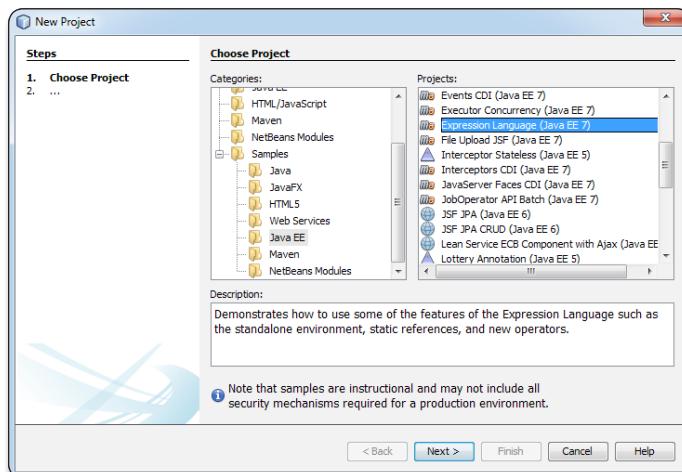


Figura 2. Wizard do NetBeans para geração de projeto contendo exemplos com a API standalone da EL 3.0

Para utilizar a API fora do container Java EE, apenas adicione a dependência apresentada na **Listagem 8** ao arquivo `pom.xml` (caso utilize Maven) ou copie o jar `javax.el.jar` localizado no diretório de instalação do GlassFish 4 para dentro do seu projeto.

Acesso a construtores e membros estáticos

Uma das reclamações de desenvolvedores levada em consideração na elaboração da nova versão da especificação foi a questão da dificuldade e complexidade de se criar funções EL para invocar métodos Java. Outro pedido antigo era por uma forma

de acessar atributos estáticos (constantes) de classes Java em páginas JSP com a Expression Language. Pois bem, esses pedidos foram ouvidos e atendidos. A EL 3.0 permite que construtores, métodos e atributos estáticos de classes Java sejam referenciados dentro de expressões.

Nota

Durante a elaboração deste artigo foi percebido alguns problemas em relação ao funcionamento correto do acesso a construtores e membros estáticos de classes em páginas JSP. Foi aberto um bug report (GLASSFISH-20778) no site do projeto GlassFish para correção do problema. Segundo a equipe técnica, a correção seria integrada posteriormente ao GlassFish 4. Por esse motivo, utilizamos a API standalone para implementar os exemplos.

Listagem 8. Dependência da EL 3.0 declarada no pom.xml

```
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.el</artifactId>
    <version>3.0.0</version>
</dependency>
```

Como pode ser visto na **Listagem 9**, a sintaxe para acesso a construtores e membros estáticos em expressões EL é bastante simples e dispensa muitos comentários. Na **linha 2** instanciamos um objeto `Long` com um construtor que recebe uma `String` como argumento. Note que a sintaxe de construção não utiliza a palavra reservada `"new"`, diferentemente de como fazemos no Java. Nas **linhas 3 e 4** acessamos membros estáticos (um atributo e um método, respectivamente) de duas classes. Em todos os três casos, a classe referenciada deve ser pública e importada no contexto do processador EL, caso esta esteja fora do pacote `java.lang`. Outra restrição é que apenas membros estáticos e públicos podem ser referenciados, além de não ser permitido modificar o valor de atributos estáticos.

Listagem 9. Acesso a construtores e membros estáticos através de expressões EL.

1. `ELProcessor elp = new ELProcessor();`
2. `System.out.println(elp.eval("Long('1000')"));`
3. `System.out.println(elp.eval("Integer.MAX_VALUE"));`
4. `System.out.println(elp.eval("Math.max(20, 50)"));`

Novos operadores

A Expression Language possui a maioria dos operadores básicos do Java, além de outros específicos de sua própria linguagem. Basicamente estão divididos em quatro grandes grupos: aritméticos (`+, -, /, *, div, %, mod, --`), relacionais (`<, >, <=, >=, lt, gt, le, ge, ==, !=, eq, ne`), lógicos (`&&, ||, and, or, !, not`) e miscelânea/outros (`?:, [], . (ponto), empty`). Mesmo com tantos operadores disponíveis na linguagem, até a versão anterior ainda não era possível, por exemplo, concatenar strings dentro de expressões EL. Isso mudou! A EL 3.0 introduziu três novos operadores (além do operador lambda “`->`”) que trazem mais capacidade e flexibilidade à linguagem de expressões.

Conheça os recursos da Expression Language 3.0

O operador “=” pode ser utilizado para fazer a atribuição de valores, e até mesmo de expressões lambdas, para variáveis definidas dentro de expressões EL ou para objetos implícitos das JSPs (**pageScope**, **requestScope**, **param**, etc.). Inclusive fizemos isso em alguns exemplos no início do artigo. A **Listagem 10** traz alguns exemplos de atribuição usando o novo operador. Observe que a sintaxe é idêntica ao Java. É importante ter em mente que a expressão do lado esquerdo do operador “=” deve resultar em uma propriedade que permita a escrita de valor, caso contrário será lançado uma **PropertyNotWritableException**. Caso a variável/bean não exista, um novo objeto é gerado dinamicamente e o valor atribuído.

Listagem 10. Uso do operador de atribuição.

```
 ${requestScope.pessoa.nome = "João"} // atribuindo valor a um atributo do request  
 ${x = 2 + 3}  
 ${soma = (x, y) -> x + y} // atribuindo uma expressão lambda
```

O operador “;” (ponto-e-vírgula) serve para separar múltiplas expressões (ou instruções) dentro de uma expressão EL composta. Nesse caso, apenas o valor da última expressão é retornado como resultado do processamento de toda a linha. Veja que no exemplo a seguir temos três diferentes segmentos separados por ponto-e-vírgula dentro da mesma expressão EL. Nos dois primeiros definimos duas variáveis e no terceiro atribuímos o retorno da invocação da função anônima para a variável **resultado**. Apenas o valor resultante do último trecho, ou seja, o conteúdo da variável **resultado**, é de fato retornado ou renderizado. Observe o código:

```
 ${num1 = 10; num2 = 30; resultado = ((x, y) -> x + y)(num1, num2)} // imprime '40'
```

Por fim, mas não menos importante, temos o operador de concatenação de Strings. Embora existam formas de se atingir o mesmo objetivo nas versões anteriores, um operador para concatenação às vezes faz falta. Para cobrir esse *gap*, foi introduzido o operador “+=” que, apesar de não ser idêntico ao seu respectivo no Java, possui a mesma lógica. Veja o exemplo:

```
 {"Olá" += pessoa.nome += " " += pessoa.sobrenome += ", seja bem vindo!"}
```

Conclusão

Neste artigo conhecemos os novos recursos da Expression Language 3.0, que faz parte da recém-lançada Java EE 7. Vimos que importantes funcionalidades foram adicionadas à linguagem de expressão, tornando-a mais poderosa e flexível, mas sem perder sua simplicidade característica.

A EL deu um passo à frente da própria linguagem Java ao oferecer suporte a expressões lambda e operações avançadas para coleções. Funcionalidades semelhantes só estarão disponíveis na linguagem no ano que vem quando a Java SE 8 for lançada. O fato de o *Expert Group* da Expression Language ter se preocupado em disponibilizar esse novo recurso de forma aderente à sintaxe que será utilizada no Java 8 é bastante positivo. Isso será importante

para garantir a interoperabilidade entre ambos no futuro.

Vimos também que *features* menores, e até baseadas em pedidos antigos da comunidade Java, foram implementadas nessa nova versão.

Entre tantas novidades, a que mais chama a atenção é a API standalone. Finalmente os desenvolvedores terão a possibilidade de levar a EL para fora do container web e usá-la no código Java. Inclusive, outras especificações da própria Java EE já estão fazendo isso, como é o caso da Bean Validation 1.1.

Mas, como poucos meses se passaram desde que a versão final foi lançada, ainda existe relativamente pouca documentação detalhada e atualizada sobre o assunto. Várias coisas mudaram desde que o Expert Group iniciou os trabalhos de elaboração da especificação. Por esse motivo, o leitor poderá encontrar na internet informações que não condizem com o que de fato foi publicado na versão final. Portanto, não deixe de consultar a especificação se tiver algum problema ou precisar esclarecer dúvidas. Se mesmo assim não for suficiente, procure o fórum no site do projeto GlassFish.

Autor



Marcelo A. Cenerino

marcelocenerine@gmail.com

É mestrado em Ciência da Computação e especialista pela Universidade Federal de São Carlos. Trabalha com desenvolvimento de software na plataforma Java há cinco anos. É editor do site InfoQ Brasil e possui as certificações IBM SOA Associate, SCJA, SCJP, SCWCD, SCBCD, SCDJWS e SCEA.



Links:

Código-fonte dos exemplos demonstrados no artigo.

github.com/marcelocenerine/JavaMagazineEL

Página de download do GlassFish 4.

glassfish.java.net/download.html

Página de download do NetBeans (a distribuição Java EE já contém o instalador do GlassFish 4).

netbeans.org/downloads/

Site oficial da JSR 341 – Expression Language 3.0.

jcp.org/en/jsr/detail?id=341

Javadoc da API EL 3.0.

javaee-spec.java.net/nonav/javadocs/javax/el/package-summary.html

Fórum e mailing list do projeto GlassFish.

glassfish.java.net/forum.html

Página do Projeto Lambda no JDK.

openjdk.java.net/projects/lambd/

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Construindo RESTful Web Services com JAX-RS 2.0

Aprenda com um exemplo prático a desenvolver aplicações JAX-RS

RESTful Web Services são serviços que podem ser publicados na internet para que clientes espalhados ao redor do mundo possam ter acesso aos seus recursos disponibilizados. Estes serviços são construídos com base na arquitetura REST (*Representational State Transfer* ou, em português, Transferência de Estado Representativo), a qual fornece um estilo de aplicação cliente-servidor para a transferência de **representações de recursos** sobre o protocolo HTTP.

Nesta arquitetura, dados e funcionalidades são considerados recursos e são acessados via URIs (*Uniform Resource Identifiers* ou, em português, Identificador Uniforme de Recursos); os tão conhecidos links na web. Um recurso pode ser qualquer informação, como dados de uma pessoa, de uma tarefa ou mesmo um post em um blog. Por exemplo, podemos considerar a previsão do tempo de uma determinada cidade como um recurso. Na internet, este tipo de informação é normalmente representado como uma página HTML, um arquivo de imagem ou mesmo um documento XML. Um cliente qualquer na web pode acessar este recurso e, dependendo do caso, pode solicitar a alteração ou a remoção do mesmo. Esta manipulação de informações é normalmente baseada nas operações GET, PUT, POST e DELETE do protocolo HTTP, onde PUT e POST são as operações que criam e alteram recursos, GET é a responsável por buscar as informações e DELETE por removê-las.

É importante destacar que na arquitetura REST os recursos são completamente desacoplados de sua representação, de forma que um mesmo recurso pode possuir diversas representações. Entre as mais utilizadas atualmente estão: JSON, XML, páginas HTML, documentos PDF e arquivos JPG. Esta diversidade de representações de um mesmo recurso facilita a integração de diferentes tipos de clientes, como web browsers, smartphones, tablets, aplicativos desktop, entre outros.

Resumo DevMan

Porque este artigo é útil:

A versão 2.0 do JAX-RS adicionou uma porção de melhorias importantes que simplificam o desenvolvimento e permitem a criação de aplicações mais robustas e sofisticadas. Neste artigo, um exemplo prático é utilizado para demonstrar como construir, a partir do zero, uma aplicação completa (cliente e servidor) empregando tal tecnologia. O estudo realizado será útil em situações nas quais o leitor queira escrever aplicações para disponibilizar serviços baseados na arquitetura REST e, também, caso o mesmo necessite consumir tais tipos de recursos. O exemplo discutido mostra como o framework JAX-RS 2.0 pode ser aplicado em situações corriqueiras no desenvolvimento de software, tais como, na criação de cadastros básicos (os bem conhecidos CRUDs) e na publicação e consumo de arquivos binários (por exemplo, fotos, imagens, relatórios PDF e etc.).

Diferentemente dos web services tradicionais que trafegam envelopes SOAP (*Simple Object Access Protocol* ou, em português, Protocolo Simples de Acesso a Objetos) sobre qualquer protocolo de comunicação, RESTful Web Services foram desenhados para trafegar principalmente sobre o protocolo HTTP. A grande vantagem disto é que estes web services podem se beneficiar de padrões amplamente utilizados e consolidados na internet. Dentre eles estão o suporte a cache de recursos, autenticação, segurança da informação, entre outros. Além disto, por possuir um conteúdo normalmente menor do que os tradicionais envelopes SOAP, estes web services são considerados “leves” (*lightweight*). O tempo de transferência dos dados na rede e a quantidade de processamento necessário para empacotar/desempacotar as mensagens também é menor do que os baseados em SOAP. Ainda mais, como foi mencionado anteriormente, RESTful Web Services podem se beneficiar de várias funcionalidades já providas pelo protocolo HTTP, fazendo com que sua implementação seja mais simples e livre de complexidades desnecessárias.

O acesso de um cliente a um web service publicado na arquitetura REST também é mais simples do que o acesso aos web services tradicionais. Isto é, basta ter um browser ou a API básica de uma linguagem de programação e pronto. É só fazer a requisição HTTP e analisar a resposta. Em contra partida, para acessar ou testar web services baseados em SOAP é necessário ter ferramentas e frameworks complexos, os quais devem atender a todas as normas da especificação de tal solução.

É importante destacar que REST é apenas um estilo arquitetural e não, como muitos pensam, um modelo formalmente definido. Por este motivo, não existe uma definição detalhada e padronizada de como a troca de mensagens entre cliente e servidor deve ocorrer. Isto varia de acordo com a interpretação de cada pessoa. Em outras palavras, cada empresa/desenvolvedor pode definir seus serviços com um estilo próprio. Por este motivo, apesar de serem serviços mais “leves” que os baseados em SOAP, muitas empresas preferem continuar utilizando os web services tradicionais para obter o padrão e formalismo desejado.

No Java, o suporte para a implementação de RESTful Web Services foi adicionado em 2008 pela especificação JSR-311, a qual recebeu o nome JAX-RS. Esta especificação foi criada para simplificar o desenvolvimento de aplicações REST e, rapidamente, se tornou de extrema importância, pois foi um dos primeiros frameworks baseados em classes POJO e anotações capaz de publicar serviços RESTful. Agora, cinco anos depois, o JAX-RS 2.0 foi lançado juntamente com a especificação Java EE 7. A nova versão, baseada na JSR-339, adicionou uma porção de melhorias importantes que simplificam ainda mais o desenvolvimento e permitem a criação de aplicações mais robustas e sofisticadas. São elas:

- **Suporte aprimorado a negociação de conteúdo.** Negociação de conteúdo é o processo de seleção da melhor representação de uma resposta do servidor quando múltiplas opções são disponíveis. Com o intuito de permitir priorizar certas representações durante este processo, a anotação `@Produces` foi enriquecida com a adição do fator relativo de qualidade do servidor (*o qs-value*);
- **Criação de uma API para o cliente.** A API do JAX-RS 1.0 era voltada estritamente para o lado do servidor, forçando o desenvolvedor a abrir conexões HTTP “na mão” ou utilizar alguma implementação de terceiros no lado do cliente. O JAX-RS 2.0, por sua vez, adicionou construtores (*builders*) para invocar um RESTful Web Service a partir do cliente. Esta nova API, além de simplificar o desenvolvimento, padroniza com uma forma elegante os acessos a este tipo de web service;
- **Adição de Filtros e Interceptadores.** A nova API fornece a capacidade de encadear filtros de Servlets de acordo com o padrão *Chain of Responsibility*. Isso é útil para implementar funcionalidades ortogonais na aplicação, como o clássico *logging* e customizações de autenticação e autorização. Os interceptadores são similares aos filtros, exceto pela capacidade de “embrulhar” (*wrap*) uma invocação de método em um ponto de extensão especificado. No JAX-RS 2.0 mais especificamente, os interceptadores são usados para manipular e enriquecer os dados do corpo de uma mensagem;

- **Supporte a especificação JSR 349: Bean Validation 1.1.** O framework *Bean Validation* está integrado ao novo JAX-RS, agindo como facilitador para especificar metadados de validação de parâmetros. Por exemplo, a anotação `@NotNull` indica que o parâmetro anotado não pode ser nulo. É possível também utilizar outras anotações pré-definidas ou personalizadas para garantir a “corretude” dos dados sendo manipulados;

- **Supporte a chamadas assíncronas.** No JAX-RS 1.0 o cliente tinha que esperar o servidor responder suas requisições uma vez que a API não suportava chamadas assíncronas. Já, na versão 2.0, o suporte assíncrono foi introduzido para permitir que um cliente faça uma chamada RESTful e, opcionalmente, obtenha uma referência de `Future` para ser notificado quando a resposta estiver completa;

- **Supporte a HATEOAS.** O HATEOAS (*Hypermedia as the Engine of Application State*, ou em português, Hipermídia como Motor de Estado da Aplicação) é um princípio da arquitetura REST que se distingue da maioria das outras arquiteturas. Este princípio requer que o cliente interaja com o servidor somente através dos links fornecidos dinamicamente pela aplicação. Em outras palavras, a ideia é a mesma de quando uma pessoa acessa uma página HTML na web. Isto é, inicialmente é necessário entrar em uma URL fixa e, após isto, basta navegar nos links provados para usufruir das funcionalidades disponíveis.

É importante destacar que por motivos de escopo este documento não abrange nem chamadas assíncronas e nem HATEOAS, porém, aconselhamos fortemente o leitor a pesquisar sobre tais assuntos devido a sua grande importância.

O restante deste artigo está dividido em: (1) uma visão geral do exemplo usado como base para implementar uma aplicação utilizando os recursos do JAX-RS 2.0; (2) os conceitos e detalhes necessários para uma codificação efetiva no lado do servidor; (3) as novas funcionalidades que facilitam a utilização de REST no lado do cliente; e (4) algumas considerações finais sobre os assuntos aqui discutidos.

0 exemplo: uma agenda de contatos

A implementação de uma agenda de contatos foi escolhida como exemplo deste artigo por ser um sistema relativamente simples e que possui, pelo menos, dois tipos diferentes de recursos. Um deles é o agrupamento das informações do contato e o outro (também discutido na literatura como sub-recurso) é a imagem visual (foto) do mesmo. Esta diversidade de recursos nos permite demonstrar para o leitor como utilizar adequadamente as anotações e o framework do JAX-RS em diferentes situações.

A **Figura 1** apresenta uma visão geral do exemplo que iremos discutir. É importante destacar nesta imagem que existem duas áreas bem distintas. Uma delas é o “Lado do Cliente” e a outra é o “Lado do Servidor”. Na primeira delas, iremos demonstrar detalhes de como implementar o componente utilizado como fachada para o recurso contato com a nova API do cliente. Na segunda, iremos demonstrar como construir um RESTful Web Service fazendo

uso da integração com *Bean Validation* para validar a criação e alteração dos contatos. Além disto, também será discutido como implementar provedores de “mapeamento de exceções”, “filtro de rastro” e a transferência de arquivos binários, neste caso, a foto do contato.

Para tornar o exemplo mais prático, foi desenvolvido uma interface com o usuário utilizando a tecnologia Swing (por questões de brevidade e foco no framework JAX-RS, detalhes desta implementação não serão abordados). Porém, para os mais interessados, existe um link no final deste artigo com o código fonte completo deste exemplo). Esta interface fornece uma tela simples para criação, alteração e remoção de contatos na agenda. O processo começa sempre que o usuário realiza uma interação com a interface gráfica. Neste momento, um evento é disparado e a “Fachada para o Contato” é acionada. É dentro deste componente que é centralizada toda a conversação com o servidor. Isto evita que o código JAX-RS fique espalhado dentro da interface gráfica da aplicação.

Deste modo, sempre que uma ação é requisitada para este componente, ele se comunica com o recurso Contato do lado do servidor via protocolo HTTP. Vale destacar que o “Filtro de Rastro” é responsável por logar toda e qualquer requisição realizada pelo cliente, bem como toda e qualquer resposta fornecida pelo servidor. Além disto, caso ocorram problemas de validação das informações do contato ou algum erro inesperado na aplicação, “Mapeadores de Exceções” são responsáveis por converter estes erros em respostas HTTP. Vale destacar que tais tipos de conversões não são padronizadas na especificação JSR-339, visto que cada aplicação costuma utilizar

uma estratégia diferente para tratamento de erros.

Por motivos de simplicidade, este exemplo não gerencia várias agendas ao mesmo tempo. Isto é, existe apenas uma agenda no lado do servidor e todos os clientes irão acessar e consultar os mesmos contatos. Para uma implementação mais completa, sugerimos fortemente que o leitor crie a possibilidade de acesso a diversas agendas. Assim, cada cliente teria sua agenda particular e não uma compartilhada com todos os usuários. Este tipo de aplicação faz mais sentido no mundo real, uma vez que estamos trabalhando com sistemas web centralizados e baseados na arquitetura REST.

Lado do Servidor

A primeira coisa que deve ser feita sempre que se deseja criar uma nova aplicação JAX-RS é definir as configurações iniciais. A forma mais simples de realizar tal tarefa é a demonstrada na **Listagem 1**. Nesta listagem podemos ver a definição da classe **AgendaApplication**, a qual estende **javax.ws.rs.core.Application**. Esta configuração é necessária para informar ao container Java EE que estamos configurando uma aplicação JAX-RS. Em situações mais complexas esta classe também poderá ser utilizada para implementar outras configurações dos RESTful Web Services da aplicação. Por exemplo, é possível informar propriedades de configurações e se um recurso REST deve ser *Singleton* ou não.

Além de definir uma classe que estende **Application**, uma configuração comumente realizada é o caminho da URL básica para acessar os serviços RESTful desta aplicação. Este caminho é definido no nosso exemplo na linha 1 com a anotação @ApplicationPath(“api”). Com esta anotação estamos dizendo ao container Java EE que toda e qualquer requisição do tipo `http://<servidor>:<porta>/<contexto>/api/<qualquer_coisa>` é uma requisição destinada a um serviço REST. Um exemplo de URL completa seria o seguinte: `http://localhost:8080/agenda/api/contato`. Neste exemplo o cliente está solicitando o recurso **contato**, o qual deve ser implementado como um serviço RESTful.

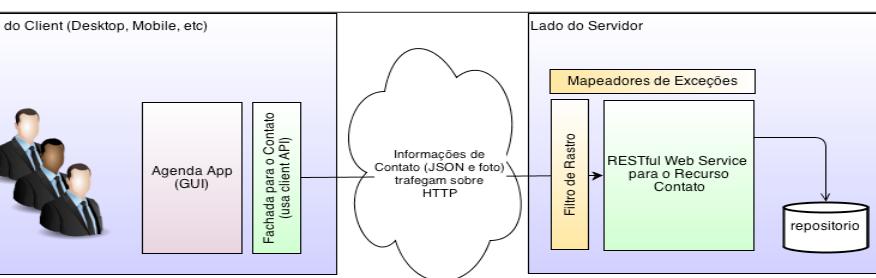


Figura 1. Visão geral da arquitetura do sistema

tação `@ApplicationPath("api")`. Com esta anotação estamos dizendo ao container Java EE que toda e qualquer requisição do tipo `http://<servidor>:<porta>/<contexto>/api/<qualquer_coisa>` é uma requisição destinada a um serviço REST. Um exemplo de URL completa seria o seguinte: `http://localhost:8080/agenda/api/contato`. Neste exemplo o cliente está solicitando o recurso **contato**, o qual deve ser implementado como um serviço RESTful.

Listagem 1. Definindo uma Application.

1. `@ApplicationPath("api")`
2. `public class AgendaApplication extends Application {}`

Classes de recursos

Após a configuração inicial de nossa aplicação, torna-se necessária a criação dos recursos que serão disponibilizados para os clientes. Como foi discutido anteriormente, o exemplo utilizado neste artigo é uma agenda de contatos. Sendo assim, faz todo o sentido definirmos um recurso chamado **contato**.

Em JAX-RS, um recurso é implementado por um POJO anotado com `@Path`. Sendo assim, a **Listagem 2** apresenta a definição da classe **ContatoResource** anotada com `@Path("contato")` para que seja publicada via HTTP como um serviço RESTful. O valor da anotação `@Path` determina o caminho relativo da URI que a nossa classe vai ser disponibilizada para receber requisições. Neste caso, a anotação está definindo que a classe **ContatoResource** será acessível através da URI `/contato`. Lembrando que todos os recursos sempre ficam “abaixo” de `/api/`, uma vez que fizemos tal configuração na seção anterior. Desta forma, para acessar este recurso é necessário utilizar a seguinte URI: `http://<SERVIDOR>:<PORTA>/<CONTEXTO>/api/contato`.

Listagem 2. @Path no recurso contato.

1. `@Path("contato")`
2. `public class ContatoResource {`
3. `...`
4. `}`

Verbo	Propósito	Anotação Java
GET	Buscar recursos.	@GET
POST	Criar ou alterar recursos. Obs: pode ser usado para fazer alterações parciais de recursos.	@POST
PUT	Criar ou alterar recursos. Obs: não deve ser usado para fazer alterações parciais de recursos.	@PUT
DELETE	Remover recursos definitivamente.	@DELETE
OPTIONS	Buscar as opções suportadas pelo servidor.	@OPTIONS
HEAD	Buscar, mas sem trazer o conteúdo. Normalmente utilizado para verificar se um determinado recurso existe.	@HEAD

Tabela 1. Métodos HTTP e seus propósitos

Mapeando requisições HTTP para métodos Java

Implementar operações básicas como criar, buscar, alterar e remover (bem conhecidas na literatura como operações CRUD) é uma tarefa relativamente simples com JAX-RS. Para isto, basta entender para que cada verbo HTTP se destina, conhecer a anotação Java correspondente e criar os métodos dentro da nossa classe de recurso com os devidos metadados.

A Tabela 1 foi definida para facilitar o entendimento dos verbos HTTP existentes atualmente, seus propósitos e respectivas anotações que podem ser adicionadas nos métodos das classes Java. Tais anotações também são conhecidas na literatura como *designators* (designadores), uma vez que o objetivo é designar uma requisição HTTP para um determinado método de acordo com o verbo utilizado na requisição.

A Listagem 3 apresenta três métodos que foram implementados na classe **ContatoResource**. O método **buscar()** que recebe um **id** por parâmetro e tem a finalidade de consultar as informações de um determinado contato. O método **salvar()** recebe um **contato** como parâmetro e, caso o contato já exista, altera as suas informações, caso contrário, cria um novo. Por fim, tem-se o método **remover()**, que recebe um **id** como parâmetro e tem o objetivo de apagar o contato definitivamente.

Listagem 3. Operações de CRUD.

```

01. @GET
02. @Path("{id}")
03. @Produces("application/json")
04. public Contato buscar(@PathParam("id") Integer id){
05.     ...
06. }
07.
08. @POST
09. @Consumes("application/json")
10. @Produces("application/json")
11. public Contato salvar(Contato contato) {
12.     ...
13. }
14.
15. @DELETE
16. @Path("{id}")
17. public void remover(@PathParam("id") Integer id){
18.     ...
19. }
```

Como podemos perceber, as definições destes métodos são bastante comuns. A grande diferença neste exemplo está nas anotações anexadas a cada assinatura. Inicialmente vamos discutir os *designators*. Notem que os métodos **buscar()**, **salvar()** e **remover()** possuem as anotações **@GET**, **@POST** e **@DELETE**, respectivamente. Estas anotações são responsáveis por mapear requisições HTTP para os seus respectivos métodos de acordo com o verbo utilizado.

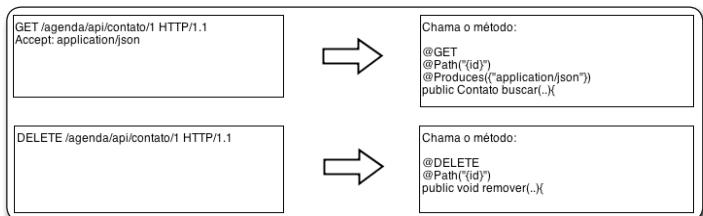


Figura 2. Requisições GET e DELETE

A Figura 2 mostra duas requisições HTTP (no lado esquerdo) e os seus respectivos mapeamentos de métodos no servidor (no lado direito) realizados pelo JAX-RS. Estes mapeamentos ocorrem com base nas anotações presentes nos métodos implementados pelo programador. Isto é, o framework do JAX-RS possui um algoritmo que, inicialmente, verifica qual verbo HTTP foi utilizado na requisição e se existe algum método Java com a respectiva anotação. Caso afirmativo, o algoritmo analisa então a URI solicitada com o objetivo de verificar se algum destes métodos é capaz de atender a requisição.

Vale destacar que a anotação **@Path** é responsável por definir o caminho relativo da URI e que, nesta anotação, também é possível adicionar templates. Um template sempre inicia com o caractere “{” e termina com “}”. Exemplos de sua utilização podem ser vistos nas anotações **@Path("{id}")** definidas tanto no método **buscar()** quanto no método **remover()** da Figura 2. O objetivo principal desta demarcação é permitir que diversas URIs sejam mapeadas para o mesmo método Java. Por exemplo, as requisições **GET /agenda/api/contato/79** e **GET /agenda/api/contato/954** são mapeadas pelo método **buscar()** e as **DELETE /agenda/api/contato/79** e **DELETE /agenda/api/contato/954** para o método **remover()**.

A Figura 3 mostra um exemplo similar ao da Figura 2, porém, neste caso, não existe a anotação **@Path** no método **salvar()**

(lado direito da imagem). Quando esta anotação não é informada, o método responde a requisições com URIs apontando para o **@Path** da classe do recurso (no nosso caso, `/ contato`). Além disto, por ser um POST, o protocolo HTTP sugere que os dados estão no corpo da mensagem e que o tipo destes dados está definido no cabeçalho HTTP *Content-type* (no nosso exemplo, `application/json`). Já, no lado direito da imagem, pode-se notar que a anotação **@Consumes("application/json")** foi adicionada no método `salvar()`. Esta anotação determina o tipo de informação que o serviço RESTful consegue consumir. Caso o valor especificado na anotação **@Consumes** não seja o mesmo informado no cabeçalho HTTP *Content-type*, o método não é eleito para receber a requisição, uma vez que não saberá como ler os dados nela contidos.

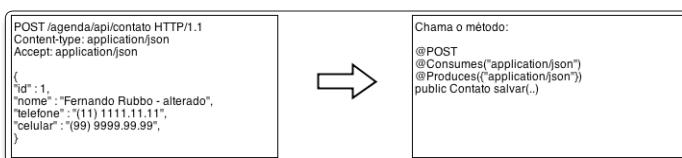


Figura 3. Requisição POST

Ainda nas **Figuras 2 e 3**, é possível verificar que existem anotações do tipo **@Produces()**. Esta anotação é similar a **@Consumes()**, porém informa qual tipo de informação o serviço irá produzir e anexar no conteúdo da resposta HTTP. Para que o mapeamento seja efetivo, o cliente deve enviar o cabeçalho HTTP *Accept*. Neste cabeçalho deve conter os tipos de dados que o cliente consegue entender. Se o cliente informar que aceita respostas escritas em XML (cabeçalho *Accept: application/xml*) e o servidor só souber escrever JSON (anotação **@Produces("application/json")**), o mapeamento não é efetivo e a requisição, por sua vez, falha, pois cliente e servidor não conversam na mesma “língua”.

Entendido como mapear uma requisição a um método Java, ainda é necessário discutir quais são as principais formas de capturar os dados provenientes destas requisições. A **Tabela 2** foi criada com o intuito de resumir esta explicação. Nesta tabela são exibidas as anotações disponíveis na API do JAX-RS, uma breve descrição da sua utilização, um exemplo de código-fonte e uma requisição HTTP que será mapeada para o método apresentado.

Um exemplo de uso destas anotações está na **Listagem 3**, nas linhas 4 e 17, e na **Listagem 4**, na linha 3. No primeiro caso, a anotação **@PathParam("id")** informa que o parâmetro do método irá receber a parte da URI correspondente ao padrão definido na anotação **@Path("{id}")**. Já no segundo caso, a anotação **@QueryParam("nome")** informa que o respectivo parâmetro do método irá receber o valor informado na query da URI. Por exemplo, se a URI for `/agenda/api/contato?nome=Marina`, o parâmetro **nome** do método **pesquisar()** irá receber a **String** “Marina”. Caso o nome não seja definido no parâmetro da URI, o valor vazio será recebido pelo método. A razão para tal afirmação está baseada na anotação **@DefaultValue("")**.

Listagem 4. Operação de pesquisa.

1. **@GET**
2. **@Produces({"application/json", "application/xml"})**
3. **public List<Contato> pesquisar(@DefaultValue("") @QueryParam("nome") final String nome){**
4. **...**
5. **}**

Na **Listagem 4** é possível verificar outra peculiaridade que ainda não foi discutida e que é uma das novidades da versão 2.0 do JAX-RS. Esta peculiaridade é o suporte a negociação de conteúdo da anotação **@Produces()**. Como podemos perceber

Anotação	Descrição	JAX-RS exemplo	Requisição exemplo
@PathParam	Atribui o valor de um template URI ao parâmetro do método Java.	@Path("{id}") void buscar (@PathParam("id") Integer id) { ... }	GET /api/contato/5
@QueryParam	Atribui o valor de um parâmetro de query ao parâmetro do método Java.	@Path("pesquisar") void pesquisar(@QueryParam("nome") String nome, @QueryParam("idade") int idade) { ... }	GET /api/contato/pesquisar?nome=Fernand o&idade=33
@CookieParam	Atribui o valor de um cookie ao parâmetro do método Java.	@Path("exec") void executar(@CookieParam("JSESSIONID") String sess) { ... }	GET /api/contato/exec Accept:/* Cookie: JSESSIONID=2K13SH5K
@HeaderParam	Atribui o valor de um cabeçalho HTTP ao parâmetro do método Java.	@Path("exec") void executar(@HeaderParam("Accept") String accept) { ... }	GET /api/contato/exec Accept:/* Cookie: JSESSIONID=2K13SH5K
@FormParam	Atribui o valor de um campo de um formulário HTML ao parâmetro do método Java.	@Path("exec") void executar(@FormParam("nome") String nome) { ... }	<form action="..."> <input type="text" name="nome">
@MatrixParam	Atribui o valor de um parâmetro de matrix ao parâmetro do método Java.	@Path("pesquisar") void pesquisar(@MatrixParam("nome") String nome, @MatrixParam("idade") int idade) { ... }	GET /api/contato/pesquisar;nome=Fernand o;idade=33

Tabela 2. Mapeamento de parâmetros da requisição

neste exemplo, a anotação define um conjunto de formatos que o serviço é capaz de gerar. Mais especificamente falando, o método `pesquisar()` pode escrever uma resposta HTTP contendo em seu corpo a representação JSON (`application/json`) ou XML (`application/xml`) da lista de contatos. Sendo assim, se o cliente informar o formato da representação desejado (por exemplo, `Accept: application/json`), o servidor irá responder de acordo. Porém, caso o cliente informe que aceita qualquer representação (por exemplo, `Accept: application/*` ou `Accept: */*`), o servidor não saberá qual das representações utilizar. Para solucionar esta questão, foi criado na nova especificação o fator relativo de qualidade do servidor (o `qs-value`), que deve ser usado para remover esta ambiguidade da anotação `@Produces()`.

Listagem 5. Fator relativo de qualidade do servidor.

```
1. @GET
2. @Produces({"application/json; qs=1", "application/xml; qs=0.75"})
3. public List<Contato> pesquisar(@DefaultValue("") @QueryParam("nome") final
   String nome){
4. ...
5. }
```

A **Listagem 5** apresenta o mesmo exemplo da **Listagem 4**, porém reescrito para definir os respectivos **qs-values**. Vale destacar que o valor do **qs-value** pode variar de 0 (indesejado) até 1 (altamente desejado), sendo 1 usado como valor padrão em caso de omissão desta informação. Deste modo, com a anotação `@Produces()` modificada na **Listagem 5** e chegando uma requisição GET com o cabeçalho `Accept: */*`, a implementação do JAX-RS irá selecionar o tipo `application/json`, pois ele possui o maior valor de `qs`.

Trabalhando com arquivos binários

A arquitetura REST foi criada para trabalhar com recursos, sendo estes recursos informações de pessoas, contatos, relatórios em PDF ou mesmo imagens. Desta forma, um arquivo binário, como a foto de um contato, é possível de ser manipulado utilizando a API do JAX-RS. Na **Listagem 6** temos um exemplo de como definir um RESTful Web Service para receber tal tipo de informação no lado do servidor.

Listagem 6. Upload da foto.

```
1. @POST
2. @Path("{id}/foto")
3. @Consumes("application/octet-stream")
4. public void uploadFoto(@PathParam("id") final Integer id, final InputStream in)
5. throws Exception {
6. ...
7. }
```

Primeiramente, notem o detalhe da anotação `@Path("{id}/foto")`. Esta anotação determina que a requisição POST deve conter a URI `/agenda/api/contato/{id}/foto` para fazer o upload de uma foto, onde `{id}` deve ser substituído pelo identificador do contato desejado.

Além deste item, a anotação `@Consumes("application/octet-stream")` informa que o cliente deve enviar no corpo da mensagem um arquivo binário. No nosso caso, a foto do contato, que será recebida como parâmetro do método para que seu conteúdo seja lido, validado e salvo.

Como é possível verificar na **Listagem 7**, para fazer o download da foto de um determinado contato, o caminho é o mesmo utilizado para fazer o upload. A grande diferença aqui é que o cliente terá que fazer uma requisição GET no lugar de POST e terá que enviar o cabeçalho `Accept: application/octet-stream` ou não informar este cabeçalho (neste caso o servidor assume `Accept: */*`).

Listagem 7. Download da foto.

```
01. @GET
02. @Path("{id}/foto")
03. @Produces("application/octet-stream")
04. public StreamingOutput downloadFoto(final @PathParam("id") Integer id)
   throws Exception {
05. ...
06. return new StreamingOutput() {
07.     @Override
08.     public void write(final OutputStream out) throws IOException {
09.         ByteArrayInputStream in = ... //arquivo para fazer o download
10.         byte[] buf = new byte[16384];
11.         int len = in.read(buf);
12.         while(len!= -1) {
13.             out.write(buf,0,len);
14.             len = in.read(buf);
15.         }
16.     }
17. };
18. }
```

É interessante destacar a implementação realizada no método `downloadFoto()` (linhas 6-17), pois ela permite que o cliente não precise esperar o término do envio do conteúdo no lado do servidor para iniciar a leitura dos dados. Em outras palavras, a utilização do tipo de retorno `StreamingOutput` proporciona uma solução mais performática e um menor uso de memória do lado do servidor quando se deseja trabalhar com recursos grandes, como fotografias, imagens, documentos PDF, etc.

Como aplicar extensões ao JAX-RS

A especificação do JAX-RS foi definida de tal forma que se as funcionalidades básicas fornecidas pelo framework não forem suficientes para resolver um determinado problema o desenvolvedor pode optar por estendê-las. Para realizar tal tarefa basta definir classes que implementem uma ou mais interfaces introduzidas na especificação e que sejam anotadas com `@Provider` (para um registro automático). Assim, quando o container Java EE é inicializado, um pré-processamento varre todas essas classes e registra automaticamente a extensão de acordo com a interface implementada.

Um exemplo disto é a classe apresentada na **Listagem 8**. Esta classe, `GeneriExceptionMapper`, implementa a interface `javax.ws.rs.ext.ExceptionMapper` com o objetivo de mapear toda e

qualquer exceção para um objeto do tipo **Response**, o qual será enviado como resposta para o cliente sempre que um erro ocorrer no lado do servidor. Vale destacar que isto é necessário pois, como foi dito anteriormente, a especificação do JAX-RS não define um mapeamento padrão para tal situação. Sendo assim, cada aplicação deve implementar seus próprios mapeadores.

Nota

É possível definir quantos mapeadores de exceções forem necessários, desde que eles não conflitem o mapeamento de uma mesma subclasse de `Exception`.

Listagem 8. Mapeando exceções.

```
01. @Provider
02. public class GenericExceptionMapper implements ExceptionMapper<Exception>
{
03.
04.     @Override
05.     public Response toResponse(final Exception e) {
06.         return Response.status(500)
07.             .entity(new AgendaError(e))
08.             .build();
09.     }
10. }
```

Neste exemplo específico da **Listagem 8**, a resposta conterá o status HTTP igual a 500 (significa erro no lado do servidor) e adicionará no corpo da mensagem um objeto do tipo **Agenda-Error** contendo os detalhes do erro. Assim, o cliente que receber este status também receberá as informações necessárias para entender o problema ocorrido no lado do servidor e, se necessário, tomar as devidas providências para que isto não volte a acontecer.

Além de mapeadores de exceções, a nova especificação também permite definir *providers* de filtros. Por exemplo, a classe **ContainerLoggingFilter** definida na **Listagem 9** tem o objetivo de interceptar requisições (implementa `javax.ws.rs.container.ContainerRequestFilter`) e respostas (implementa `javax.ws.rs.container.ContainerResponseFilter`) para logar o rastro de tudo o que foi realizado na aplicação.

O primeiro método `filter()` (linhas 6 até 14) tem o objetivo de interceptar as requisições feitas a todo e qualquer recurso. Já o segundo método `filter()` (linhas 17 até 32) objetiva interceptar as respostas enviadas pelo servidor ao cliente. É interessante destacar que estes métodos fornecem objetos do tipo **ContainerRequestContext** e **ContainerResponseContext** como parâmetros. Com estes objetos de contexto o desenvolvedor pode acessar qualquer informação de requisição e de resposta, respectivamente. Além disto, em uma implementação de **ContainerRequestFilter** é possível parar a execução chamando o método `abortWith(Response)` no seu objeto de contexto correspondente. Por exemplo, em um filtro que implementa a autenticação ou a autorização de um sistema, é possível abortar uma requisição caso o usuário não tenha as devidas permissões.

Listagem 9. Logando todas as requisições e respostas.

```
01. @Provider
02. public class ContainerLoggingFilter
03.     implements ContainerRequestFilter, ContainerResponseFilter {
04.     ...
05.     @Override
06.     public void filter(final ContainerRequestContext requestContext)
07.         throws IOException {
08.         requestContext.setProperty(START_TIME, System.currentTimeMillis());
09.         String method = requestContext.getMethod();
10.         String path = requestContext.getUriInfo().getPath();
11.         MediaType mediaType = requestContext.getMediaType();
12.         String body = readBody(requestContext);
13.         LOGGER.log(Level.INFO, "Request: {0} {1} - Content-Type: {2} -> "
14.             + "body: {3}", new Object[]{method, path, mediaType, body});
15.     }
16.     @Override
17.     public void filter(final ContainerRequestContext requestContext,
18.         final ContainerResponseContext responseContext) throws IOException {
19.         String method = requestContext.getMethod();
20.         String path = requestContext.getUriInfo().getPath();
21.         int status = responseContext.getStatus();
22.         long totalTime = calcularTempoTotalDaRequest(requestContext);
23.         MediaType mediaType = responseContext.getMediaType();
24.
25.         Object entity = responseContext.getEntity();
26.         if(entity instanceof List) {
27.             int size = ((List<?>) entity).size();
28.             entity = "List com " + size + " elementos";
29.         }
30.         LOGGER.log(Level.INFO, "Response: {0} {1} : {2} in {3}ms - Content-Type: {4} -> {5}"
31.             new Object[]{method, path, status, totalTime, mediaType, entity});
32.     }
33.     ...
34. }
```

Validando dados utilizando Bean Validation

Validação é o processo de verificar se alguma informação obedece a um ou mais critérios pré-definidos. A especificação de **Bean Validation** (JSR 349) publica uma API para validar objetos em Java e é utilizada para dar suporte nativo, na versão 2.0 do JAX-RS, a validação de atributos, parâmetros e retorno de serviços RESTful.

A **Listagem 10** apresenta dois trechos de código, sendo o primeiro deles (linhas 1 a 12) a especificação do método `salvar()` dentro da classe **ContatoResource** e o segundo (linhas 13 a 27) a definição do objeto **Contato**, o qual é utilizado como parâmetro e retorno desse método.

Para habilitar a validação em parâmetros de serviços RESTful, basta adicionar as anotação do framework *Bean Validation* (como por exemplo, `@NotNull`, `@Valid` e etc.). Estas anotações informam para o *runtime* do JAX-RS que a validação do objeto determinado deve ocorrer antes mesmo que a execução do método seja realizada. Este é o motivo pelo qual o parâmetro `contato` (linha 8 da **Listagem 10**) foi anotado com `@Valid`.

Além disto, como é possível perceber na implementação do objeto **Contato**, é necessário anotar toda e qualquer propriedade que se deseja validar. Por exemplo, na propriedade `nome` (linhas 17 e 18 da mesma listagem) foi definido que o seu conteúdo não

pode ser nulo e que deve ter entre 2 e 100 caracteres. Para saber mais sobre validações complexas e/ou customizadas, visite o site oficial da especificação (veja a seção [Links](#)).

Listagem 10. Validação de dados utilizando Bean Validation.

```
01. // --- trecho 1 ---
02. @Path("contato")
03. public class ContatoResource {
04.     ...
05.     @POST
06.     @Consumes("application/json")
07.     @Produces({"application/json"})
08.     public Contato salvar(@Valid Contato contato) {
09.         ...
10.     }
11. }
12.
13. // --- trecho 2 ---
14. @XmlRootElement(name="contato")
15. public class Contato implements Cloneable, Comparable<Contato> {
16.     private Integer id;
17.     @NotNull @Size(min=2, max=100, message="O nome deve possuir entre {min} e {max} caracteres")
18.     private String nome;
19.     @Pattern(regexp="^([0-9]*$)", message="O telefone aceita apenas números")
20.     private String telefone;
21.     @Pattern(regexp="^([0-9]*$)", message="O celular aceita apenas números")
22.     private String celular;
23.     @Past(message="A data de nascimento deve estar no passado")
24.     private Date dataNascimento;
25.
26.     ...
27. }
```

Listagem 11. Configurando a utilização do framework Jackson para leitura e escrita de mensagens JSON.

```
01. @Provider
02. public class JacksonFeature implements Feature {
03.
04.     @Override
05.     public boolean configure(final FeatureContext context) {
06.         // INICIO - Glassfish only
07.         final String disableMoxy = jersey.config.disableMoxyJson;
08.         + context.getConfiguration().getRuntimeType().name().toLowerCase();
09.         context.property(disableMoxy, true);
10.         // FIM - Glassfish only
11.
12.         context.register(JacksonJaxbJsonProvider.class,
13.                         MessageBodyReader.class, MessageBodyWriter.class);
14.         return true;
15.     }
16. }
17.
18. ---
19.
20. @Provider
21. public class ObjectMapperResolver implements ContextResolver<ObjectMapper> {
22.     final ObjectMapper combinedObjectMapper;
23.
24.     public ObjectMapperResolver () {
25.         combinedObjectMapper = createCombinedObjectMapper();
26.     }
27.
28.     @Override
29.     public ObjectMapper getContext(final Class<?> type) {
30.         return combinedObjectMapper;
31.     }
32.     ...
33. }
```

Utilizando JavaScript Object Notation

JavaScript Object Notation (ou JSON) é um padrão bem consolidado e, atualmente, o mais utilizado para representações de objetos que necessitam ser transferidos entre o cliente e o servidor em sistemas web. Infelizmente a JSR 339 (JAX-RS 2.0) não define, por padrão, conversores de JSON para POJO. Desta forma, para usar este tipo de conversão torna-se necessário informar a implementação desejada ou aderir ao suporte fornecido pelo servidor de aplicações, o qual pode variar de um fornecedor para outro.

Uma vez que o framework Jackson (mais informações sobre ele podem ser encontradas na seção [Links](#)) é uma das bibliotecas Java para processamento de JSON mais utilizadas no mercado, a **Listagem 11** mostra os detalhes para registrar tal implementação como responsável por ler e escrever representações do tipo *application/json*.

É importante perceber que duas classes foram definidas nesta listagem e ambas são registradas automaticamente devido ao uso da anotação **@Provider** (linhas 1 e 20). A primeira delas, **JacksonFeature**, é responsável por registrar o provedor **JacksonJaxbJsonProvider** para a leitura e escrita de JSON (linhas 12 e 13). A segunda, **ObjectMapperResolver**, é responsável por fornecer informações de contexto para a primeira. Neste caso específico, a informação de contexto é a configuração de como a API do Jackson deve converter objetos Java em JSON e vice-versa. Vale destacar que para o correto funcionamento da aplicação, a forma de conversão entre Java e JSON deve ser a mesma adotada no servidor e no cliente.

Lado do Cliente

Como foi discutido anteriormente, o lado do cliente faz uso de uma fachada para não espalhar código JAX-RS por toda a interface gráfica. Nesta fachada são definidos os métodos que serão invocados pela interface gráfica. Tais métodos são especificados de forma que não seja necessário nenhum conhecimento dos detalhes de sua implementação para fazer acesso ao serviço RESTful de contato. A **Listagem 12** apresenta o construtor desta fachada. Note que foi utilizada a nova API do cliente de JAX-RS para acessar os recursos disponibilizados no lado do servidor. Inicialmente, na linha 6, um novo objeto do tipo **Client** é criado a partir da classe **ClientBuilder**. Como no lado do cliente não existe um pré-processamento para registrar automaticamente as classes anotadas com **@Provider**, para trabalhar com JSON torna-se necessário registrar manualmente a biblioteca do Jackson e a sua respectiva configuração (linhas 7 e 8). Uma vez criado o objeto **client** é necessário criar o objeto **target** do tipo **WebTarget** que aponta para a URI raiz do nosso recurso *contato* (linha 9). Este objeto será reutilizado por todos os métodos desta fachada e, por este motivo, ele foi definido como variável de instância.

O objeto **WebTarget** possui uma API fluente que permite invocações de métodos encadeadas para construir a URI que será utilizada para submeter uma requisição HTTP. Conceitualmente, os passos necessários para submeter uma requisição com a nova API são os seguintes:

1. Obter a classe **Client** através do **ClientBuilder**;
2. Registrar extensões do JAX-RS, caso necessário;
3. A partir do **Client** criar um **WebTarget**;
4. Utilizar os métodos do **WebTarget** para:
 - a. Concatenar caminhos;
 - b. Concatenar e resolver templates;
 - c. Adicionar parâmetros de query;
 - d. Adicionar cabeçalhos HTTP;
 - e. Adicionar cookies.
5. A partir do **WebTarget** executar **request()**;
6. Submeter a requisição com um dos seguintes métodos: **get()**, **post()**, **put()**, **delete()**, **head()** ou **options()**.

Nota

Os métodos definidos na classe **WebTarget** são métodos do tipo **builder**. Isto é, eles são métodos construtores de novos objetos **WebTarget**. Por este motivo, é possível reutilizar variáveis deste tipo realizando chamadas encadeadas de métodos sem ter o receio que estas chamadas alterem o objeto original.

Listagem 12. Fachada para Contato no lado do cliente.

```

01. public class ContatoResourceFacade {
02.   private final Client client;
03.   private final WebTarget target;
04.
05.   public ContatoResourceFacade(final String host, final int port) {
06.     client = ClientBuilder.newClient()
07.       .register(new JacksonFeature())
08.       .register(new ObjectMapperResolver());
09.     target = client.target("http://" + host + ":" + port + "/agenda/api/contato");
10.   }

```

Para demonstrar este fluxo, podemos analisar o método **buscar()** da **Listagem 13** (linhas 1 até 5). Neste método o objeto **target** é encadeado com a chamada do método **path("{id}")**, que concatena um template no final da URI, e com **resolveTemplate("id", id)**, que resolve o template adicionado anteriormente. Neste momento, a URI que existe dentro do objeto aponta para o contato que se deseja buscar do servidor. Então, uma requisição é criada (linha 3) dizendo que o conteúdo esperado na resposta é um JSON (isto automaticamente adiciona o cabeçalho *Accept: application/json* na requisição). Por fim, é executada a requisição com o método **get()** (linha 4). Como simplificador, a API do JAX-RS permite passar por parâmetro deste método a classe do objeto que se pretende ler da resposta quando esta retornar.

A implementação do método **remover()**, na mesma listagem, também utiliza o **target** para concatenar e resolver um template da URI. As principais diferenças da implementação do método **buscar()** é que a chamada do método **request()** não recebe parâmetros (linha 9), pois não é esperado nenhum retorno no corpo da resposta, e é executado o método **delete()** no lugar do **get()** (linha 10). Vale a pena destacar que neste caso é interessante analisar a resposta para checar se a exclusão aconteceu com sucesso (linha 11). Em caso de sucesso o status HTTP será 200; qualquer outro valor representa um erro.

Listagem 13. Métodos CRUD no lado do cliente.

```

01. public Contato buscar(final Integer id){
02.   return target.path("{id}").resolveTemplate("id", id)
03.     .request("application/json")
04.     .get(Contato.class);
05. }
06.
07. public void remover(final Integer id){
08.   Response resp = target.path("{id}").resolveTemplate("id", id)
09.     .request()
10.     .delete();
11.   throwExceptionIFError(resp);
12. }
13.
14. public Contato salvar(final Contato contato){
15.   Response resp = target
16.     .request("application/json")
17.     .post(Entity.json(contato));
18.   throwExceptionIFError(resp);
19.   return resp.readEntity(Contato.class);
20. }

```

Já o método **salvar()** difere um pouco mais dos demais, uma vez que em um POST é necessário informar o conteúdo que será trafegado no corpo da mensagem. Neste exemplo, na linha 17, é possível perceber que o objeto **contato** é convertido para JSON e adicionado no corpo da mensagem. Após a requisição ser finalizada é necessário analisar se algum erro aconteceu (linha 18); caso negativo, o corpo da resposta pode ser lido e transformado em um objeto do tipo **Contato** (linha 19).

De forma similar, a **Listagem 14** apresenta a fachada para o método **pesquisar()**. A principal diferença entre este caso e os anteriores é a adição de uma chamada encadeada do método **queryParam("nome", nome)** na linha 2. Este método adiciona um parâmetro de query **nome** na URI original, o qual será utilizado pelo servidor para encontrar o contato a ser pesquisado. Neste caso, a URI construída será a seguinte: */contato?nome=<nome_a_ser_pesquisado>*.

Listagem 14. Método pesquisar no lado do cliente.

```

1. public List<Contato> pesquisar(final String nome){
2.   return target.queryParam("nome", nome)
3.     .request("application/json")
4.     .get(LIST_CONTATO_TYPE);
5. }

```

A **Listagem 15**, por sua vez, implementa os métodos de upload e download da imagem de um contato. Similarmente aos métodos **buscar()** e **remover()**, um template é adicionado e resolvido e, então, uma **request()** é criada. O diferencial neste exemplo é que no método **uploadFoto()** os bytes que representam a foto são adicionados no corpo da requisição pelo método **post()** na linha 4 e no método **downloadFoto()** os bytes são lidos da resposta via **get()** na linha 11.

Para completar, a **Listagem 16** apresenta o método auxiliar **throwExceptionIFError()** usado para verificar se algum erro aconteceu no lado do servidor. No exemplo apresentado neste artigo, por motivo de brevidade, consideraram-se os status HTTP

entre 400 e 600 como problemáticos. Porém, isto pode variar de aplicação para aplicação e tal verificação deve ser implementada de acordo com a necessidade.

Por fim, é importante ressaltar que os objetos **Contato** e **AgendaError** utilizados em ambos os lados, cliente e servidor, não precisam ter os mesmos binários. Em outras palavras, o cliente pode conter informações e funcionalidades que não existam no servidor e vice-versa. As únicas regras que devem ser respeitadas são: a adição da anotação **@XmlRootElement** (linhas 1 e 8 da **Listagem 17**) e outras anotações JAXB para customizar a geração de XML, caso se deseje trabalhar com tal tipo de conteúdo; a criação de um construtor padrão (linhas 4 e 11 da mesma listagem) para que os conversores implementados possam construir os objetos necessários; e as propriedades que irão trafegar do cliente para o servidor (ou vice-versa), que devem estar definidas em ambos os lados.

Conclusão

Como foi discutido na introdução deste artigo, RESTful Web Services não é a solução para todos os problemas. O que eles

ganham em “leveza” e simplicidade perdem em padronização e formalismo. Porém, a cada dia que passa é possível perceber que a adoção deste estilo arquitetural cresce em passos largos. Empresas como Google, Amazon, Facebook, LinkedIn, entre outras, preferem disponibilizar APIs baseadas na arquitetura REST para que programadores possam acessar seus recursos de forma simples e prática.

Devido à flexibilidade natural desta arquitetura, serviços podem ser disponibilizados para fazer chamadas remotas de métodos (discutido neste artigo) ou para Hipermídia como Motor de Estado da Aplicação (HATEOAS), onde os produtores e consumidores concordam com um conjunto de links a serem utilizados para a correta navegação dentro da aplicação.

Vale destacar que assuntos como HATEOAS, chamadas assíncronas, interceptadores e o framework *Bean Validation* foram mencionados apenas superficialmente neste documento. Assim, o leitor é fortemente encorajado a continuar os seus estudos sobre estes assuntos. Apesar de muitas vezes a leitura ser um pouco pesada, as especificações são as melhores fontes para se aprender as novas funcionalidades que aparecem no mundo Java.

Listagem 15. Métodos de upload e download no lado do cliente.

```
01. public void uploadFoto(final Integer id, final byte[] foto) {
02.     Response resp = target.path("/{id}/foto").resolveTemplate("id", id)
03.         .request()
04.         .post(Entity.entity(foto, "application/octet-stream"));
05.     throwExceptionIfError(resp);
06. }
07.
08. public byte[] downloadFoto(final Integer id) {
09.     return target.path("/{id}/foto").resolveTemplate("id", id)
10.         .request("application/octet-stream")
11.         .get(byte[].class);
12. }
```

Listagem 16. Tratando exceção vinda do servidor.

```
01. private void throwExceptionIfError(final Response resp) {
02.     int status = resp.getStatus();
03.     if(status>=400 && status<600){
04.         AgendaError error = resp.readEntity(AgendaError.class);
05.         throw new AgendaServerException(error);
06.     }
07. }
```

Listagem 17. Classes Contato e AgendaError.

```
01. @XmlRootElement(name="contato")
02. public class Contato {
03.     ...
04.     public Contato(){}
05.
06.     ...
07.
08.     @XmlRootElement(name="agendaError")
09.     public class AgendaError {
10.         ...
11.         public AgendaError(){}
```

Autor



Fernando Rubbo

É mestre em Engenharia de Software pela Universidade Federal do Rio Grande do Sul (UFRGS). Tem experiência de mais de doze anos na área, atuando a maior parte deste tempo como consultor, arquiteto de software e professor universitário. Certificado pela Sun Microsystems na plataforma Java como programador (SCJP), associate (SCJA), desenvolvedor (SCJD), desenvolvedor de componentes de negócio (SCBCD) e também, componente WEB (SCWCD).



Links:

Site oficial da JSR 339: JAX-RS 2.0 - The Java API for RESTful Web Services.

<http://www.jcp.org/en/jsr/detail?id=339>

Código fonte utilizado neste artigo.

<https://github.com/fbrubbo/BrazilianJavaMagazine>

Site oficial da JSR 349: Bean Validation 1.1.

<http://www.jcp.org/en/jsr/detail?id=349>

Site oficial da biblioteca Jackson.

<http://wiki.fasterxml.com/JacksonHome>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



O Cache de Segundo Nível no Hibernate – Parte 2

Melhore o desempenho de caches no disco



ESTE ARTIGO FAZ PARTE DE UM CURSO

Na primeira parte do artigo vimos como configurar o EhCache para atuar como C2N do Hibernate e exploramos diferentes opções de configurações inerentes a API do Hibernate que devem ser refletidas na configuração do cache, por exemplo, opções transacionais no modo de acesso à entidade e definição de regiões em que serão armazenadas as consultas e entidades.

Nesta segunda parte do artigo faremos algumas considerações com respeito à utilização de caches em disco que, embora transparentes do ponto de vista de configuração, introduz novos aspectos aos quais devemos estar atentos antes de decidir se vamos fazer uso de caches em disco ou não.

Duas das principais razões que podem nos levar a refletir se devemos ou não armazenar dados em disco é a quantidade de memória disponível para a JVM e a quantidade de objetos que ficarão armazenados em cache. No primeiro caso, a preocupação principal é se o uso do cache pode tomar espaço de memória que deveria ser alocado para outros componentes necessários para a execução da aplicação. No segundo, a preocupação é se o número de objetos ocupará espaço suficiente para colocar pressão no Garbage Collector, degradando o desempenho geral da aplicação. Como sempre, em se tratando de cache, a melhor solução para esses problemas depende tanto do conhecimento do hardware como da aplicação.

Entretanto, ao começar a utilizar caches em disco, é possível identificar estruturas e padrões que geram

Resumo DevMan

Porque este artigo é útil:

Complementando o primeiro artigo, no qual discutimos como funciona o Cache de 2º Nível (C2N) no Hibernate, nesta segunda parte detalharemos o funcionamento do EhCache quando configurado para atuar em disco. Embora trivial para se configurar, um cache em disco comporta-se de maneira bem diferente de um cache puramente em memória, gerando picos de alocação de objetos que podem vir a ser inesperados quando se deseja manter a latência de uma aplicação sob controle. Ao longo do artigo iremos explorar como se comportam as estruturas do C2N quando serializadas para um arquivo e discutiremos técnicas para melhorar o desempenho na leitura desses dados.

custos para a aplicação, mas que podem ser atenuados, conforme veremos ao longo do artigo. Antes de discutir as estratégias que utilizaremos para melhorar o desempenho do C2N do Hibernate, precisamos primeiro entender como funciona o EhCache quando passamos a descarregar o conteúdo da memória para o disco.

Camadas de armazenamento

O EhCache define como Camada de Armazenamento (*Storage Tier*) a região física onde ficarão armazenados os dados contidos em um cache. Uma configuração standalone do EhCache permite que as entradas do cache (chave e valor) sejam armazenadas em três camadas. Em ordem decrescente de velocidade de acesso, essas camadas são enumeradas como:

- 1. Heap Memory:** as entradas são armazenadas como objetos Java “normais” e são diretamente acessíveis pela JVM;
- 2. Off Heap Memory:** as entradas são armazenadas de forma serializada em estruturas do tipo `java.nio.ByteBuffer`. Essa classe foi introduzida no JDK 1.4 como a estrutura fundamental para operações de input/output realizadas na API de NIO, fazendo

O Cache de Segundo Nível no Hibernate – Parte 2

o papel dos clássicos Streams e arrays de bytes. O diferencial de **ByteBuffer** é que ele pode armazenar uma quantidade grande de memória (até 2GB por instância) nativa. Quando um buffer é criado através do método **allocateDirect()**, a JVM aloca blocos de memória que não são gerenciados como os outros objetos, no sentido que não participam dos ciclos de garbage collection, o que permite que aplicações façam uso de centenas de GB de memória sem sofrer pausas devido à coleta de lixo;

3. Disk: as entradas são armazenadas de forma *serializada* em arquivos.

Neste artigo nos limitaremos a falar da camada de armazenamento do tipo *Disk*, porém os conceitos discutidos podem ser aplicados para a camada *Off Heap Memory*, que requer uma licença para ser utilizada.

O EhCache permite que todos os níveis sejam adotados em um único cache e trabalha para manter os dados mais frequentemente acessados nas camadas cujo acesso é mais rápido. Por exemplo, se um cache é configurado para conter até 100 elementos em memória (heap memory) e 100.000 no disco, ao longo da utilização do cache, a tendência é que os 100 elementos requisitados com maior frequência passem a residir na camada de acesso mais rápido.

A **Figura 1** mostra o processo de consulta no cache quando configurado com mais de uma camada de armazenamento. O EhCache sempre procura na camada de acesso mais rápido e caso não encontre, efetua a mesma pesquisa nas camadas de acesso mais lento (disco).

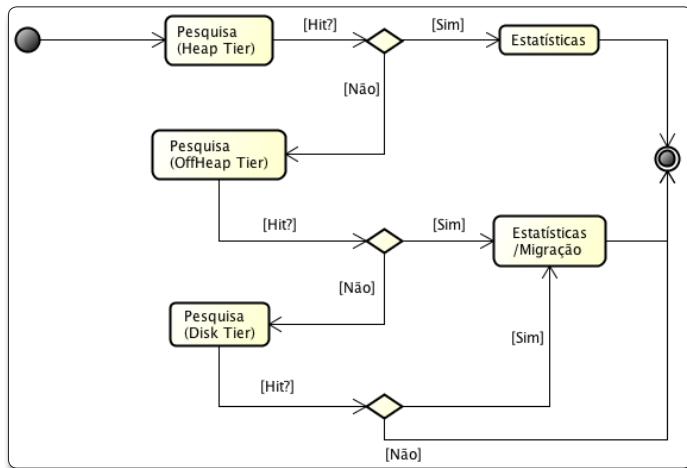


Figura 1. Consulta no cache com Armazenamento Off-Heap e em Disco

Neste diagrama estão caracterizadas duas outras etapas que ocorrem após consultas efetuadas com sucesso, a atualização de estatísticas e a migração. A atualização de estatísticas consiste em marcar quantas vezes um elemento foi acessado e o último momento em que foi acessado. Quando um elemento é encontrado em disco e o cache atingiu o número máximo de elementos em memória, ocorrerá a migração de um dos elementos em memória para o disco e vice-versa. Para determinar qual elemento da

memória deve migrar para o disco, o EhCache aplica uma política de expurgo em uma *amostra* do conjunto de dados em memória. Por exemplo, se a política de expurgo for do tipo LRU (padrão), a “vítima” escolhida será o elemento que foi acessado menos recentemente. Por outro lado, se a política for LFU, o elemento escolhido será o que tiver o menor número de acessos.

É importante ressaltar que a migração é baseada em pequenas *amostragens aleatórias*, o que implica que não necessariamente o melhor candidato será migrado, porém também diminui o custo de execução da política de expurgo. O EhCache utiliza uma amostragem de até 30 elementos e testes empíricos mostram que os algoritmos de LRU e LFU eliminam o melhor candidato 99% das vezes, considerando-se 25% de todos os elementos do cache. Em outras palavras, se um cache tem 1 milhão de elementos, ao utilizar amostras de 30 elementos, sobre um subconjunto de 250.000, cerca de 99% das vezes a política de expurgo eliminará o melhor candidato do subconjunto.

Características do armazenamento em disco

Um cache configurado para armazenamento em disco cria um arquivo no file system e gerencia as operações de I/O utilizando um objeto `java.io.RandomAccessFile`.

À medida que um cache é populado com pares [Chave,Valor], o EhCache atualiza as contagens de dados nas camadas de armazenamento e decide em qual delas irá guardá-los. Se o número máximo de elementos em memória for atingido, o cache passa a descarregar o conteúdo adicional para o disco.

Quando o *Disk Tier* for utilizado para inserir novos elementos, os mesmos serão armazenados de forma *serializada* no arquivo gerenciado pelo cache, porém as chaves correspondentes ainda serão mantidas em memória, encapsuladas em uma estrutura que contém também dois números, que servem para marcar o início e fim do bloco de bytes do elemento associado no arquivo. Esquematicamente poderíamos representar os registros do cache em disco de modo semelhante à **Figura 2**.

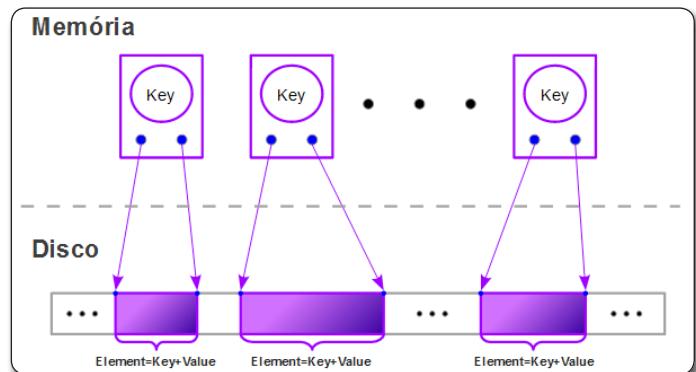


Figura 2. Associação de chaves em memória com blocos de um arquivo em disco

Ao se submeter uma consulta ao cache e o mesmo determinar que o valor correspondente encontra-se em disco, a engine do EhCache executará uma operação *seek* no arquivo para posicionar

e ler a quantidade correta de bytes, deserializando o elemento e trazendo-o novamente para a memória.

Como vemos, a solução de armazenamento em disco do EhCache é “incompleta”, no sentido que as chaves ainda são mantidas em memória! De fato, um cache em disco pode ser pensado como um *índice* em memória, o que quer dizer que o cache não precisa realizar leituras em disco para encontrar determinada chave. As consultas são sempre feitas em memória e só quando se encontrar um elemento associado o cache fará a leitura do conteúdo em disco para materializar os dados. Como as chaves são mantidas em memória, ao utilizar o armazenamento em disco o ganho em termos de consumo do Heap será proporcional ao tamanho dos objetos armazenados como valor no cache, isto é, se as chaves forem objetos proporcionalmente maiores em relação aos valores, o ganho será pequeno.

O armazenamento em disco utilizado pelo EhCache é uma solução do tipo “temporary swap only”, ou seja, não é efetivamente um mecanismo de persistência, pois não garante a integridade dos dados caso a aplicação falhe durante a escrita dos mesmos. Ao reiniciar uma aplicação, o conteúdo dos arquivos associados a um cache em disco é descartado e o cache deve ser “aquecido” novamente. Como diferencial, as versões enterprise do EhCache oferecem soluções de armazenamento do tipo “local restartable”, que permitem que um cache seja iniciado com os valores que foram salvos em disco.

Em termos de desempenho, um fator determinante que influencia a performance dos caches em disco é o mecanismo utilizado para escrita e leitura dos objetos. Para lidar com esta tarefa de armazenamento dos dados, a versão standalone do EhCache emprega a API de serialização padrão do Java (classes `java.io.ObjectOutputStream` e `java.io.ObjectInputStream`), a qual favorece a portabilidade dos dados em detrimento do desempenho de processamento e IO.

Hibernate com caches em disco

Na **Figura 3** da parte 1 do artigo vimos como é o formato dos objetos armazenados no C2N, representados pelas classes `CacheKey` e `CacheEntry`. Ao introduzir o armazenamento em disco do EhCache, instâncias dessas classes serão serializadas e armazenadas em um arquivo na inserção de um elemento no cache e deserializadas ao efetuar uma consulta com sucesso no cache (**Figura 2**).

Infelizmente a API de C2N do Hibernate foi concebida com um design pouco extensível, que não permite que sejam realizadas customizações e otimizações no que diz respeito ao formato das classes.

Enquanto estamos trabalhando apenas com registros em memória, não há grandes consequências com relação à maneira como são criados, estruturados e mantidos os objetos utilizados no C2N. Entretanto, ao se avaliar o custo para se transferir esses dados da memória para o disco e vice-versa, percebemos que boa parte do processamento poderia ser evitada ao se rearranjar a estrutura de algumas classes, de modo a promover uma representação binária mais compacta e eficiente.

Para compreender como podemos otimizar o C2N em disco, precisamos entender onde se localizam as operações custosas na API de serialização do Java e do lado do Hibernate quais as estruturas não são otimizadas para serem serializadas, conforme veremos nas seções seguintes.

Descritores de Tipos

Em geral, os maiores culpados pela “fama” de má performance atribuída a API de serialização são os chamados descritores de tipo (*type descriptors*), que são representados pela classe `java.io.ObjectStreamClass` (OSC).

A classe OSC é responsável por armazenar informações a respeito do formato de uma classe. Assim como os objetos do tipo `java.lang.Class`, essa classe é um contêiner de metadados, que arquiva “dados a respeito de dados” de uma classe a ser serializada.

Dado um tipo T, o objeto OSC associado a T guarda as informações como o nome da classe que representa T e uma coleção de informações de cada campo não transiente, que são encapsuladas em instâncias da classe `ObjectStreamField`. Por exemplo, a instância de OSC associada à classe `java.lang.Integer` pode ser pensada como a tupla `[Integer.class,[value],[+ outras informações]]`. Nesta tupla, `value` é o campo da classe `Integer` que de fato retém o valor numérico de 32 bits e está associado a uma instância de `ObjectStreamField`.

Durante o processo de serialização de um objeto, na primeira vez que um determinado tipo é encontrado, seu descriptor é escrito no stream de destino (`OutputStream`) e um identificador numérico é atribuído ao mesmo. Nas próximas vezes que esse tipo for encontrado, apenas esse “id” será escrito no stream, ao invés do descriptor completo. Por exemplo, os trechos de código da **Listagem 1** produzem, respectivamente, 81 bytes e 91 bytes.

Listagem 1. Overhead de type descriptors.

```
Integer first=1;
Integer second=2;

ByteArrayOutputStream bb = new ByteArrayOutputStream();
try(ObjectOutputStream oos = new ObjectOutputStream(bb)){
    oos.writeObject(first);
}
//81 bytes
byte[] b = bb.toByteArray();

bb.reset();

try(ObjectOutputStream oos = new ObjectOutputStream(bb)){
    oos.writeObject(first);
    oos.writeObject(second);
}
//91 bytes
b = bb.toByteArray();
```

Ou seja, para serializar uma única instância de `Integer` são necessários 81 bytes! Desses 81 bytes, aproximadamente 95% dos conteúdos são os *metadados* contidos no descriptor da classe `java.lang.Integer` e o que realmente interessa (valor do número, 4 bytes)

O Cache de Segundo Nível no Hibernate – Parte 2

corresponde apenas a 5%. Esquematicamente poderíamos representar a serialização de um **Integer** como mostrado na **Figura 3**.

Note que à medida que um descritor é serializado, um identificador numérico (7) é atribuído ao mesmo. O número 7 é apenas um exemplo, pois a atribuição número - descritor é realizada de maneira ad hoc e controlada pela API do Java, podendo variar de acordo com a ordem em que objetos são serializados. Quando um novo **Integer** for serializado no mesmo contexto, isto é, antes do método **close()** de **ObjectOutputStream** ser invocado, o identificador numérico será escrito ao invés de toda a informação do descritor (**Figura 4**). Por isso há um acréscimo de apenas 10 bytes quando se serializa dois **Integers**, totalizando 91 bytes.

Nota

Este é um esquema simplificado do processo de serialização padrão do Java. Na verdade a API escreve além do descritor da classe, todos os descritores da hierarquia, até chegar a uma classe que herde diretamente de `java.lang.Object`! No caso, como `Integer` é filha de `java.lang.Number`, o descritor associado à mesma também seria escrito no Stream. O número 'Flags' é utilizado como um controle interno da API, que marca, por exemplo, se a serialização é customizada. Como `Integer` só tem um campo, a serialização demarcará apenas um `ObjectStreamField`, representado pelo par `[l,value]`. Este par é composto por um caractere, conhecido como `typeCode` e por uma String que corresponde ao nome do campo associado. No exemplo, o caractere '`l`' corresponde ao `typeCode` do tipo primitivo `int` e `value` corresponde ao nome do campo encapsulado na classe `Integer`. No total existem apenas 10 `typeCodes`, um para cada tipo primitivo, um para objetos ('`L`') e um para arrays ('`[`').

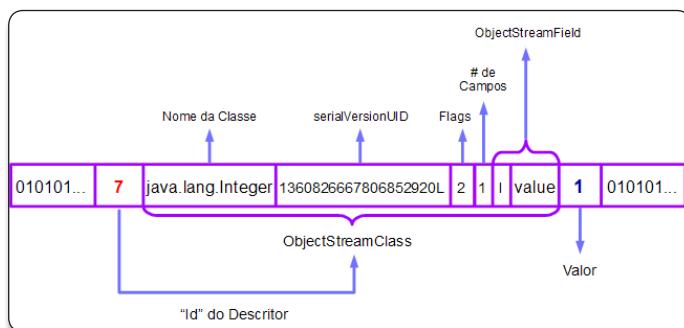


Figura 3. Serialização de um Integer

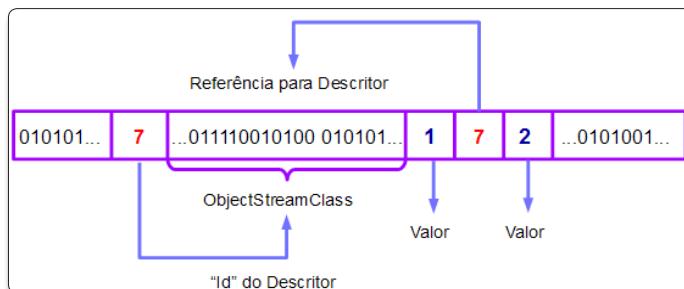


Figura 4. Serialização de dois Integers

Voltando ao mundo do C2N no Hibernate, se recordarmos da estrutura de `CacheEntry` (veja a **Figura 3** da Parte 1) e da forma em que é constituída (processo de *disassembling*), percebemos que uma instância de `CacheEntry` pode conter um array de

objetos serializáveis de diversos tipos, o que significa que todos os descritores distintos deverão ser integralmente serializados! Para termos uma ideia da dimensão deste custo, consideremos a entidade `RichTypeEntity` descrita pela **Listagem 2**.

Listagem 2. Entidade 'rica' em tipos diferentes.

```
@Entity
public class RichTypeEntity {

    @Id
    long id;

    @Column
    int a;

    @Column
    String b;

    @Temporal(TemporalType.DATE)
    @Column
    Date c;

    @Column
    Long d;

    @Column
    Double e;

    //gets e sets
}
```

Ignorando o campo `id` (que é atribuído à classe `CacheKey`), a estrutura gerada pelo *disassembling* do Hibernate ao colocar a `CacheEntry` associada a uma instância de `RichTypeEntity` no C2N conterá cinco tipos distintos. Se repetirmos o código da **Listagem 1**, utilizando, ao invés de inteiros, arrays com valores dos tipos encapsulados por `RichTypeEntity`, verificamos que serializar um objeto custa 275 bytes e dois objetos, no mesmo contexto, 344 bytes (considerando para o campo `b` strings com apenas um caractere). A conclusão que chegamos é similar à verificada na serialização de `Integers`, isto é, a maior parte dos dados que são serializados em uma invocação do método `writeObject()` são, na verdade, metadados!

Antes de mencionarmos como podemos otimizar os custos de serialização, vamos mostrar e analisar algumas evidências que indicam que, de fato, há um custo razoável atribuído à serialização de objetos. A **Tabela 1** apresenta algumas medições que evidenciam o efeito causado pela serialização de descritores.

Os resultados desta tabela medem o tempo médio para efetuar 10.000 operações do tipo `session.get(RichTypeEntity.class, id)` em diferentes configurações de cache e hardware (disco). O que é interessante notar é que mesmo com um disco de estado sólido muito superior a um disco mecânico (de 10 a 100 vezes mais rápido, dependendo do cenário de operações de IO), os resultados para leituras de cache em disco são muito próximos. Isso indica que o cache em disco é muito mais limitado pela qualidade da CPU do que do disco em si e, portanto, otimizações no nível da aplicação podem ser relevantes, resultando em ganhos da ordem de 80% neste caso.

O que é interessante observar também é o formato do Heap ao longo da execução desses testes. A **Figura 5** mostra um comparativo dos picos de utilização de memória nas duas execuções.

É notável o que um teste simples como esse pode revelar em termos de utilização de memória.

As montanhas nos gráficos mostram a evolução das regiões do Heap ao longo do tempo. Os gráficos superiores referem-se à utilização do GC padrão (PS Scavenge/MarkSweep), onde as montanhas em azul representam o espaço ocupado na região do Eden (novos objetos). Para esse coletor, as alocações atingem picos de 150MB para a versão default e 65MB para a versão com classes do C2N otimizadas.

Os gráficos inferiores mostram a evolução do Heap com o coletor G1, que é ativado com a opção de inicialização da VM: `-XX:+UseG1GC`, disponível desde o update 14 do JDK 6. Nesses gráficos, as montanhas verdes correspondem ao “G1 Eden”, e embora os picos de utilização de memória sejam similares em ambas as versões, o teste na versão não otimizada rodou por muito mais tempo (~45s contra ~33s da versão otimizada) e foi o único que constatou um tempo de GC superior a 1 segundo, ocasionado pela soma de todas as “minor collections”, que podem ser observadas na **Figura 6**. Embora essas medidas tenham sido obtidas com um profiler, é muito simples obter estatísticas de GC pela API de JMX, como mostra o trecho de código da **Listagem 3**.

No teste não foram registradas coleções do tipo “stop the world” (*full gc*), que para todas as threads da aplicação para tentar liberar memória de todas as regiões do heap. A função principal de uma coleção completa é tentar liberar memória de regiões que armazenam objetos “antigos”, que sobreviveram a diversos ciclos de minor collections, apresentadas nos gráficos.

Note que o tempo total de execução ficou muito próximo com estruturas otimizadas para ambos os GCs, enquanto que com as estruturas “padrão” a diferença entre MarkSweep (~39s) e G1 (~45s) é da ordem de 6s. Essa diferença pode ser atribuída à natureza dos coletores. O MarkSweep é

HDD	Sem Cache	Cache em Memória	Cache em Disco	Cache em Disco, otimizado
Toshiba, mecânico, 5400rpm	6.1s	60ms	1093ms	593ms
Corsair Neutron GTX, SSD	5.1s	62ms	1030ms	572ms

Tabela 1. Microbenchmarks de operações ‘get’ no C2N utilizando diferentes camadas de armazenamento

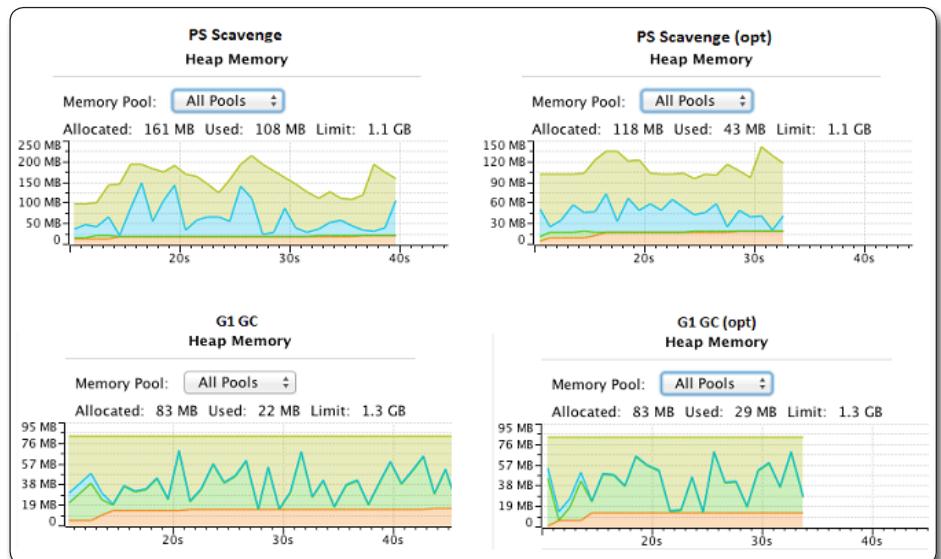


Figura 5. Heaps com estruturas padrão do C2N (esq.) e otimizadas (dir.)

Nota

Microbenchmarks são medidas de desempenho de uma de forma isolada, normalmente executada diversas vezes para ter uma média. Em Java os resultados de um microbenchmark podem variar bastante dependendo das opções de inicialização da JVM (e.g. `-XX:+AggressiveOpts`), do garbage collector e principalmente do compilador JIT. É uma boa prática quando se realiza um microbenchmark deixar a JVM ser “aquecida” antes de começar a tomar medições, pois algumas otimizações do compilador da JVM só surtem efeito após certo número de vezes que determinado método é invocado. Neste artigo, todos os testes foram realizados utilizando o Java 7 update 25 (jdk 7u25) com a opção `-server`. Os caches em disco foram realizados no sistema operacional OSX Mountain Lion, processadores Intel core i7 2.7GHz e 4GB de memória à 1333MHz. Os testes envolvendo replicação utilizaram servidores com sistema operacional Red Hat Enterprise Linux versão 5.5, processadores Intel Xeon E7440(x4) 2.4GHz e 32GB de memória à 2400MHz.

Listagem 3. Obtendo estatísticas de GC programaticamente.

```
public void dumpGCStatistics(){
    List<GarbageCollectorMXBean> gcs = ManagementFactory.getGarbageCollectorMXBeans();

    for (GarbageCollectorMXBean gc : gcs) {
        logger().info("GC: {}. Regiões Coletadas: {}. # de Coleções: {}. Tempo de Coleção: {}", new Object[]{
            gc.getName(), Arrays.toString(gc.getMemoryPoolNames()),
            gc.getCollectionCount(), gc.getCollectionTime()
        });
    }
}
```

um coletor do tipo “throughput”, isto é, enquanto há espaço, ele prioriza a alocação, adiando a coleta de lixo. Já o G1 é um coletor que prima por reutilizar o espaço disponível, compactando e rearranjando as regiões do Heap. Como veremos, as estruturas otimizadas reduzem a criação

de objetos que migram do disco para a memória, o que “facilitou a vida” do G1 no teste.

Tipos do Hibernate

Dentre os campos armazenados na classe `CacheKey` (veja a **Figura 3** da parte 1),

O Cache de Segundo Nível no Hibernate – Parte 2

podemos destacar o campo **type**, que é atribuído a uma implementação de uma interface de mesmo nome. A interface **org.hibernate.type.Type** define os contratos de mapeamento entre o mundo orientado a objetos e o banco dados.

A API do Hibernate apresenta dezenas de implementações de **Type** e toda entidade é invariavelmente associada a um conjunto dos mesmos, de acordo com o tipo de seus campos e relacionamentos, o que permite ao Hibernate gerar corretamente as instruções SQL nas operações de CRUD. O **Type** armazenado em **CacheKey** corresponde ao tipo do campo utilizado como identificador da entidade.

Por exemplo, se o id for um campo do tipo **Long**, uma **CacheKey** para uma instância da entidade irá referenciar uma instância de **org.hibernate.type.LongType**.

O que é interessante observar é que todas as instâncias dessas classes de mapeamento são de fato *metadados* e em sua grande maioria são definidos como *singletons* pelo Hibernate. Dentre os tipos que não são implementados como *singletons*, estão os tipos que devem ser parametrizados em tempo de execução (durante o “parsing” das entidades que ocorre na inicialização da **SessionFactory** que irá gerenciá-las), como por exemplo, o tipo **ManyToOneType**, que marca qual

entidade é referenciada por uma anotação **@ManyToOne**.

Independentemente de serem ou não *singletons*, as implementações de um determinado **Type** são mantidas em memória e devem ser de natureza **imutável**, isto é, uma vez que a **SessionFactory** é construída, os tipos não devem ser alterados. A estrutura responsável por manter os tipos associados a uma determinada entidade é a classe **org.hibernate.metadata.ClassMetadata** (CMD), que pode ser pensada como sendo a versão do Hibernate de um descritor de tipo do Java (OSC).

Infelizmente a API de C2N do Hibernate não se aproveita do fato de tipos serem *singletons* (ou mantidos em memória pela **SessionFactory**) quando o assunto é serialização, pois trata seus metadados como objetos comuns. Este ‘descuido’ faz com que o benefício da utilização do *design pattern* desapareça quando um elemento do C2N é lido a partir do disco, considerando que novos objetos são recriados durante o processo de deserialização. Dentre as implicações associadas a essa negligência por parte da API de C2N, podemos destacar dois efeitos colaterais:

- O custo de serialização de tipos é elevado. Serializar um **LongType** requer 883 bytes;
- Quando uma **CacheKey** migra do disco para a memória, ela fica com uma cópia de um tipo do Hibernate em memória. Para entender por que isto acontece, consideremos um cache que armazena alguns objetos em memória e um número muito maior em disco. Podemos imaginar o layout inicial da memória como o representado na **Figura 7**. Nesta figura, as instâncias de **CacheKey** guardam valores inteiros e consequentemente referenciam o tipo **IntegerType** do Hibernate. Os valores associados às chaves com identificadores 1 e 2 são mantidos em memória e os outros valores são mantidos em disco.

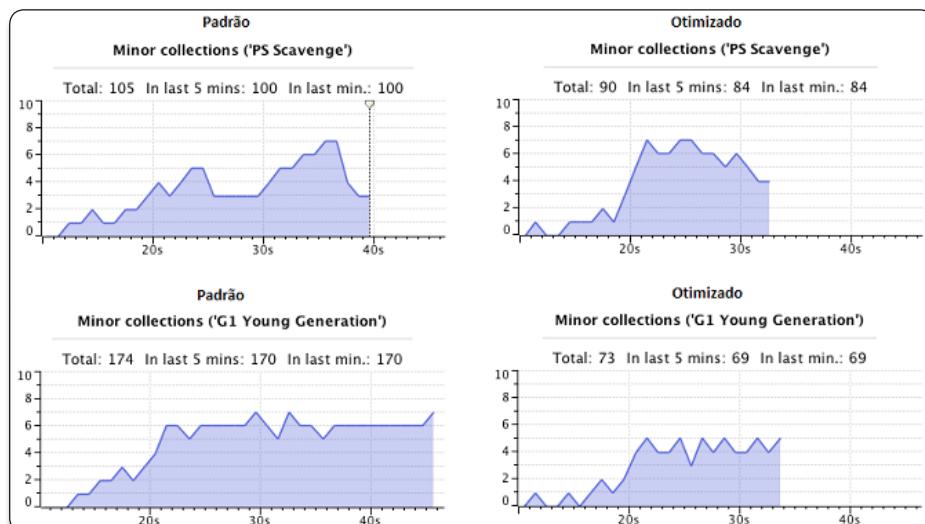


Figura 6. Garbage Collection: estruturas padrão (esq) e otimizadas (dir).

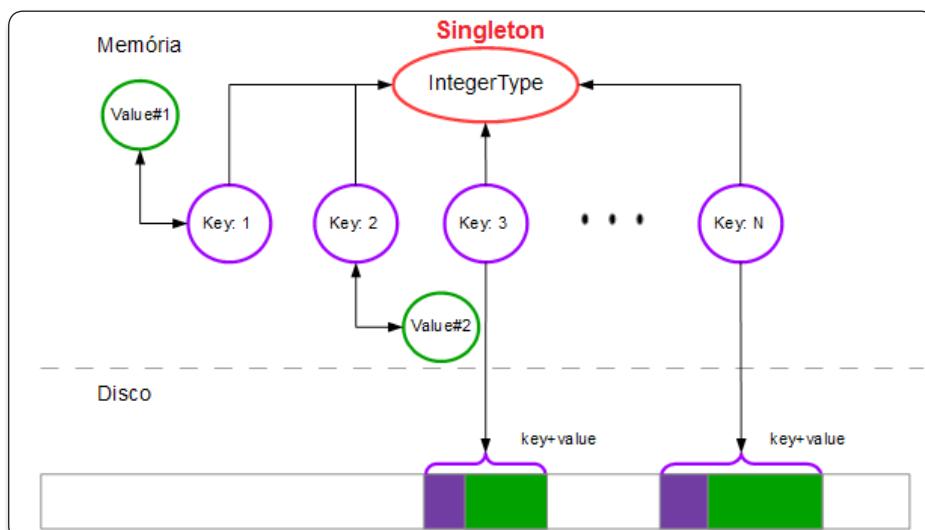


Figura 7. Layout Inicial da Memória

Quando se submeter uma leitura do identificador de número 3, o valor correspondente que estava em disco migrará para a memória e, se o número máximo de elementos em memória for atingido, algum elemento da memória poderá migrar

para o disco. Nesse processo, ilustrado na **Figura 8**, a chave que foi deserializada do disco substituirá a chave em memória, e esta nova instância estará associada a uma também nova instância de **IntegerType**, que deixará de ser um singleton por estar duplicado na memória.

Assim, se você dimensionou seu cache para manter até um milhão de objetos em memória, eventualmente haverá um desperdício devido às cópias de tipos. Cada instância dos tipos “simples” (**LongType**, **IntegerType**, etc.) custa cerca de 64 bytes no Heap. Portanto, se houver um milhão de cópias, estamos falando de cerca de 62MB de desperdício.

Queries

Consultas sofrem os mesmos problemas que discutimos anteriormente e podem ser ainda mais agravados dependendo do número e tipos dos parâmetros utilizados, além do tamanho da **String** de consulta em si.

No primeiro artigo da série verificamos o custo associado ao se adotar Criteria no lugar de NamedQueries, para o C2N em memória. Enquanto que em memória fica evidenciado um ganho da ordem de 200% ao se empregar NamedQueries com C2N em memória, em disco o ganho é muito menos pronunciado (~20%) devido ao fato das consultas serem serializadas e perdermos o benefício do cálculo de equals como uma operação de tempo constante – O(1) – quando uma **String** migra do disco para a memória (ver **Figuras 7 e 8**). Pelas mesmas razões, a redução de memória que obtemos pela escolha de NamedQueries ao invés de Criteria também é anulada quando o C2N é utilizado em disco.

As **Tabelas 2 e 3** mostram o resultado da execução do mesmo teste apresentado na seção “Prefira NamedQueries à Criteria” da parte 1, utilizando no entanto, o C2N em Disco. Nesse caso as otimizações apresentam um ganho da ordem de 60%.

Otimizando as estruturas do C2N

Conforme observamos nas seções anteriores, os objetos armazenados no C2N não possuem uma estrutura compacta quando serializados. Em poucas palavras

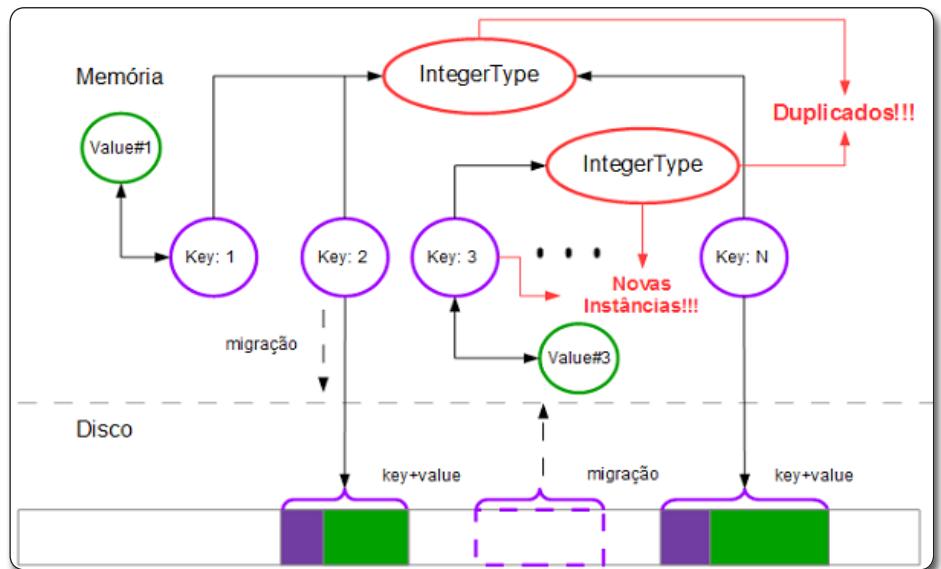


Figura 8. Layout da Memória pós-migração

HDD	Sem Cache	Cache em Memória	Cache em Disco	Cache em Disco, otimizado
Toshiba, mecânico, 5400rpm	6867s	140ms	3115	1984
Corsair Neutron GTX, SSD	6605ms	137ms	3080ms	1957ms

Tabela 2. Microbenchmarks de consultas com NamedQuery utilizando diferentes camadas de armazenamento

HDD	Sem Cache	Cache em Memória	Cache em Disco	Cache em Disco, otimizado
Toshiba, mecânico, 5400rpm	6001s	412ms	3715	2285
Corsair Neutron GTX, SSD	5795ms	416ms	3635ms	2260ms

Tabela 3. Microbenchmarks de consultas com Criteria utilizando diferentes camadas de armazenamento

poderíamos dizer que “há muita informação estática sendo armazenada de forma desnecessária”. Como a API do Hibernate não é “plugável” com respeito às classes que são utilizadas, se desejarmos otimizá-las temos três possibilidades:

1. Otimizar o mecanismo de serialização do EhCache. Embora possível, uma solução genérica não teria conhecimento das características específicas das classes do C2N do Hibernate;
2. Criar as classes otimizadas, que não violam os contratos do Hibernate, compilá-las e reempacotá-las no JAR do Hibernate;
3. Utilizar um *Java Agent*, que intercepta classes do C2N no momento em que são carregadas e as transforma em suas versões otimizadas.

Neste artigo optamos pela terceira opção por ser menos intrusiva e nos permitir

testar diferentes formatos sem ter todo o trabalho de reconstruir um JAR. Antes de discutir como vamos utilizar agente, primeiramente vamos mostrar as transformações que desejamos fazer com relação às classes do C2N. Essas transformações essencialmente envolvem implementar o padrão *Flyweight* de modo a otimizar o acesso a tipos do Hibernate e descritores de classe. A otimização de descritores é comumente empregada em frameworks de caches distribuídos como o próprio EhCache (quando conectado ao Terracotta) e o Oracle Coherence.

Criando um cache de Metadados

Como forma de otimizar as classes do C2N, vamos utilizar um segundo cache! Esse cache fará mapeamentos dos tipos do Hibernate para números e vice-versa, que serão utilizados para melhorar a serialização das classes do C2N,

O Cache de Segundo Nível no Hibernate – Parte 2

substituindo objetos complexos por números. O código relevante da classe responsável por efetuar estes mapeamentos é mostrado na **Listagem 4**.

Esta classe apresenta cinco métodos relacionados a mapeamentos de tipos do Hibernate. Esses métodos são utilizados para transformar um objeto em um número e vice-versa, o que reduz a quantidade de bytes que devem ser escritos na serialização e lidos na deserialização, otimizando a execução em ambos os casos.

Os métodos **roleToInt()** e **idToRole()** são responsáveis pelo mapeamento **String int**, que otimizarão a (de)serialização do campo **entityOrRoleName** da classe **CacheKey** e do campo **subclass** da classe **CacheEntry**. Os métodos de mapeamento, **idToType()** e **typeToInt()**, otimizarão o campo **type** da classe **CacheKey** e o campo **types** da classe **QueryKey**. Toda informação necessária para a

Nota

Os mapeamentos de tipos são construídos de forma ‘estável’ no sentido que toda vez que forem construídos, o mapeamento será idêntico desde que entidades não sejam modificadas ou novas entidades sejam adicionadas. O mapeamento de descritores e consultas, por sua vez, é construído de forma ‘ad hoc’, ou seja, a cada novo objeto não mapeado atribuímos um número sequencial, de modo que diferentes execuções de um programa podem resultar em diferentes mapeamentos de descritores e queries. Isso não tem implicações para caches locais, pois como mencionamos, o EhCache descarta o conteúdo do disco quando a aplicação é reinicializada. Entretanto, para caches replicados é importante que esses mapeamentos também sejam estáveis, caso contrário irão ocorrer erros de serialização se duas JVMs possuírem mapeamentos distintos!

construção desses mapeamentos pode ser consultada na **SessionFactory** assim que estiver construída. O único mapeamento que é construído de forma “lazy” é a transformação de uma **String**

Listagem 4. Código da classe MetadataCache: Flyweight de tipos do Hibernate

```
public class MetadataCache{  
  
    static final ConcurrentHashMap<Object, Object> DESCRIPTOR_CACHE;  
    final Type[] intToType;  
    final IdentityHashMap<Type, Integer> typeToInt;  
    final String[] intToRole;  
    final HashMap<String, Integer> roleToInt;  
    final ConcurrentHashMap<String, Integer> queryMap;  
  
    public MetadataCache(SessionFactory...factories){  
        //constrói mapeamentos de tipos e roles  
    }  
  
    //Métodos para mapeamento de tipos do Hibernate  
  
    public String idToRole(final int id){  
        if (id < 0 || id >= intToRole.length) {  
            outOfRange("roles", intToRole.length - 1, id);  
        }  
  
        return intToRole[id];  
    }  
  
    public int roleToInt(final String role){  
        final Integer val = roleToInt.get(role);  
  
        if (val == null) {  
            notRegistered("Role", role);  
        }  
  
        return val;  
    }  
  
    public Type idToType(final int id){  
        if (id < 0 || id >= intToType.length) {  
            outOfRange("types", intToType.length - 1, id);  
        }  
  
        return intToType[id];  
    }  
  
    public int typeToInt(final Type type){  
        final Integer val = typeToInt.get(type);  
        if (val == null) {  
            notRegistered("Type", type.getName());  
        }  
  
        return val;  
    }  
  
    public int getQueryId(String query){  
        Integer id = queryMap.get(query);  
  
        if(id==null){  
            synchronized (queryMap) {  
                id=queryMap.get(query);  
                if(id==null){  
                    id=queryMap.size();  
                    queryMap.put(query,id);  
                }  
            }  
        }  
  
        return id;  
    }  
  
    ...Métodos para mapeamento OSC<=>int  
  
    public static int lookup(final ObjectStreamClass type){  
        Integer val = (Integer) DESCRIPTOR_CACHE.get(type);  
  
        if (val == null) {  
            val = osc(type);  
        }  
  
        return val;  
    }  
  
    public static ObjectStreamClass lookup(final int id) throws ClassNotFoundException {  
        ObjectStreamClass osc = (ObjectStreamClass) DESCRIPTOR_CACHE.get(id);  
  
        if (osc == null) {  
            notRegistered("OSC ID", String.valueOf(id));  
        }  
  
        return osc;  
    }  
  
    static synchronized int osc(final ObjectStreamClass type){  
        Integer val = (Integer) DESCRIPTOR_CACHE.get(type);  
        if(val==null){  
            DESCRIPTOR_CACHE.put(type, val=DESCRIPTOR_CACHE.size());  
            DESCRIPTOR_CACHE.put(val, type);  
        }  
        return val;  
    }  
}
```

`sql` em um número inteiro (`String int`), a qual é contemplada pelo método `getQueryId()`. Este mapeamento em especial não precisa ser bidirecional, pois a classe `CacheKey` utiliza a `String sql` apenas na execução dos métodos `hashCode()` e `equals()`.

Com relação a descritores de classes, `MetadataCache` usa o map `DESCRIPTOR_CACHE` e os métodos `lookup()` para realizar as conversões `OSC int`. Este mapa, diferentemente dos outros utilizados na classe, é estático, pois o escopo dele é global (classes da aplicação), enquanto que as outras estruturas são populadas a partir de uma ou mais `SessionFactory`s específicas.

Modificando a classe CacheKey

Para podermos utilizar os mapeamentos de `MetadataCache` vamos criar classes que encapsulam números no lugar de objetos, porém sem modificar a assinatura dos métodos da API pública do Hibernate. No caso de `CacheKey`, apenas o método `getEntityOrRoleName()` é utilizado por outras classes do Hibernate, o que significa que se modificarmos o estado dessa classe, mas preservarmos sua “interface”, não vamos “quebrar” o código que faz uso de `CacheKey`.

O diagrama da Figura 9 exibe as transformações necessárias para gerar uma forma compacta da classe `CacheKey`. Como podem ser observados, todos os objetos, salvo o id, são transformados em números.

Os campos mapeados da classe `CompactCacheKey` são populados durante sua construção utilizando os dados contidos na classe `MetadataCache`, conforme mostra o código em destaque na Listagem 5.

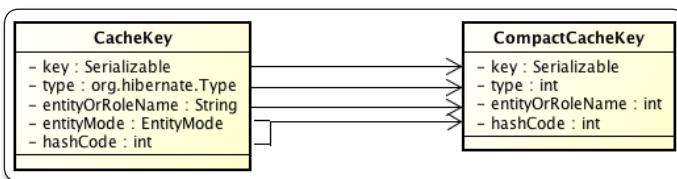


Figura 9. Transformação de CacheKey

Listagem 5. Construção de CacheKey compacta.

```

public class CompactCacheKey {

    public CompactCacheKey(final Serializable id, final Type type,
                          final String entityOrRoleName, final EntityMode entityMode,
                          final SessionFactoryImplementor factory) {
        this.key = id;

        final MetadataCache cache = MetadataCache.getInstance();

        this.type = cache.typeTold(type);
        this.entityOrRoleName = cache.roleTold(entityOrRoleName);
        this.hash = ...;
    }

    //Método da API pública
    public String getEntityOrRoleName() {
        return MetadataCache.getInstance().idToRole(entityOrRoleName);
    }
}
  
```

Na Figura 9 podemos perceber que o campo `entityMode` foi eliminado na versão compacta de `CacheKey`. De fato, como esse só pode assumir três valores (**POJO**, **MAP** ou **DOM4J**), em `CompactCacheKey` ele é codificado nos dois primeiros bits do `hashCode` como os números 0, 1 ou 2 (para mais detalhes, consulte o código disponibilizado na página desta edição da Java Magazine).

Modificando a classe CacheEntry

No exemplo apresentado para a classe `RichTypeEntity`, constatamos que o custo de serialização associado à classe `CacheEntry` vem da diversidade de descritores que podem ser serializados. Para alavancar o uso do mapeamento `OSC int` proporcionado pela classe `MetadataCache`, precisamos estender as classes `ObjectOutputStream` e `ObjectInputStream` de forma a utilizar o cache de descritores. Felizmente a API de serialização permite que esta customização seja feita através da sobreescrita dos métodos `writeClassDescriptor()` e `readClassDescriptor()`, como mostramos na Listagem 6.

Listagem 6. Streams para serialização otimizada.

```

static final class OOS extends ObjectOutputStream {
    public OOS(final OutputStream os) throws IOException {
        super(os);
    }

    @Override
    protected void writeClassDescriptor(final ObjectStreamClass osc)
        throws IOException {
        writeInt(MetadataCache.lookup(osc));
    }
}

static final class OIS extends ObjectInputStream {
    public OIS(final InputStream is) throws IOException {
        super(is);
    }

    @Override
    protected ObjectStreamClass readClassDescriptor() throws IOException,
        ClassNotFoundException {
        return MetadataCache.lookup(readInt());
    }
}
  
```

Como pode ser visto no projeto disponibilizado na página dessa edição da Java Magazine, ambas as classes da Listagem 6 são encapsuladas em uma terceira classe, `Serializer`, que as utiliza para serializar e deserializar objetos de forma mais eficiente. Nesta classe também são aplicadas outras otimizações de baixo nível, como compressão de números positivos e reuso de estruturas de dados.

Para fazer uso da serialização otimizada, a classe `CompactCacheKey` implementa a interface `Externalizable`, customizando a leitura e escrita de seu estado, conforme exposto na Listagem 7.

Idealmente quem deveria controlar a serialização de forma otimizada (ou permitir implementações plugáveis de `Streams`) deveria

O Cache de Segundo Nível no Hibernate – Parte 2

ser o EhCache, pois ele é o responsável por ler e escrever os objetos em disco, contudo isso poderia gerar uma incoerência quando o cache é utilizado de forma replicada, conforme veremos mais para frente. Do lado do EhCache essa deficiência já foi percebida e atualmente é uma melhoria em aberto, como pode ser visto no JIRA indicado na seção **Links**.

No projeto disponibilizado no site da revista, apresentamos também outra versão da classe **CompactCacheEntry** que é serializada utilizando a biblioteca Kryo, a qual oferece um ganho ainda maior de desempenho.

Listagem 7. Serialização otimizada em CacheEntry.

```
public final class CompactCacheEntry implements Externalizable{  
    //campos similares a CacheEntry  
  
    @Override  
    public void readExternal(ObjectInput in) throws IOException,  
        ClassNotFoundException {  
        byte[] buff = Serializer.toBytes(disassembledState);  
        //...  
    }  
  
    @Override  
    public void readExternal(ObjectInput in) throws IOException,  
        ClassNotFoundException {  
        byte[] b = ...;  
        disassembledState = (Serializable[]) Serializer.fromBytes(b);  
    }  
  
}
```

Modificando a classe QueryKey

Em se tratando de serialização, a classe **QueryKey** sofre dos mesmos problemas de **CacheKey**, que podem ser ainda mais agravados pois **QueryKey** encapsula não apenas um tipo do Hibernate, mas vários. Além disso, **QueryKey** também guarda a String SQL que pode ser muito extensa, variando conforme o número de campos e tabelas que são utilizados na consulta. Ao realizar as transformações sugeridas na **Figura 10**, boa parte do overhead de serialização será eliminado.

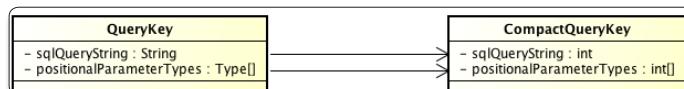


Figura 10. QueryKey otimizada

Os valores dos campos mapeados são obtidos através de **MetadataCache**, de maneira similar à **Listagem 5**.

Com as classes **CacheKey**, **CacheEntry** e **QueryKey** transformadas em suas representações compactas, podemos prosseguir para

o próximo passo, aplicar de fato essas modificações às classes através de um *Java Agent*, que veremos na terceira parte do artigo.

Conclusão

Nesta segunda parte do artigo avaliamos o desempenho do Cache de 2º nível do Hibernate em cenários em que todos os registros do cache encontram-se em disco. Observamos que o cache em disco não é uma “bala de prata” para a solução de problemas de falta de memória, principalmente porque as chaves ainda são mantidas em memória.

Verificamos que embora as consultas ao disco sejam muito mais lentas que as consultas em memória, o desempenho do cache ainda é bem superior em comparação com o acesso ao banco de dados e pode ser otimizado aplicando-se transformações simples às classes utilizadas pela API de C2N, melhorando consideravelmente tanto o tempo de consulta no cache como a utilização de memória na JVM.

Autor



Cleber Muramoto

Doutor em Física pela USP, é Especialista de Engenharia de Software na empresa Atech Negócios em Tecnologias-Grupo Embraer. Possui as certificações SCJP, SCBD, SCWCD, OCPJWSD e SCEA.



Links e Referências:

Site do EhCache.

<http://ehcache.org/>

Melhorias de Serialização no EhCache.

<http://jira.terracotta.org/jira/browse/EHC-1017>

Site do Hibernate.

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

Site do JGroups.

<http://docs.jboss.org/jbossas/jboss4guide/r4/html/jbosscache.chapt.html>

Repositório de dependências para o Projeto de Testes.

<http://repo.maven.apache.org/maven2/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Reusando componentes com a API de Fragmentos

Conheça a API do Android que elimina o desafio de se trabalhar com diferentes tamanhos de tela

O aumento da procura por dispositivos portáteis, como smartphones e tablets, que auxiliam os usuários em suas atividades do cotidiano, tem motivado a indústria na produção de aparelhos com vários tamanhos de tela para diferentes finalidades. Esse fato despertou o interesse dos desenvolvedores a criar soluções que aproveitem ao máximo o espaço de tela disponível nesses aparelhos.

Entretanto, tem sido um enorme desafio para os programadores desenvolver aplicativos que se adaptem aos mais variados tamanhos de tela, pois um layout portável com uma boa interface visual e que ocupe todo o espaço da tela, tanto em smartphones quanto em tablets, exige um grande esforço.

Visando reduzir esse esforço foi criada a API de Fragmentos, que tem como objetivos principais a construção de layouts dinâmicos e o reuso de componentes. Os fragmentos são descritos como uma espécie de mini activities, que por sua vez podem ser adicionadas a uma Activity. Dessa forma é possível compor uma tela de diferentes maneiras, agrupando um ou vários fragmentos dentro de uma Activity.

A API de Fragmentos foi introduzida na plataforma Android a partir da versão 3.x (Honeycomb). Apesar disso, está disponível uma biblioteca de compatibilidade (*android-support-v4*) que permite que alguns recursos presentes no Honeycomb – entre eles, fragmentos – sejam suportados nas versões do Android 2.x para smartphones.

Neste contexto, o intuito deste artigo é apresentar as principais funcionalidades dessa API por meio de exemplos práticos, onde serão expostas suas vantagens e desvantagens. Além disso, também será apresentado ao leitor o ciclo de vida de um fragmento, o uso da biblioteca de compatibilidade e como manipular fragmentos dinamicamente. Ao final, será desenvolvida uma aplicação de leitura de notícias via RSS que utiliza os principais conceitos de fragmentos.

Resumo DevMan

Porque este artigo é útil:

Este artigo apresenta ao leitor o uso, vantagens e desvantagens da API de Fragmentos do Android. É importante saber que desenvolver para smartphones já não é o bastante hoje em dia. Criar aplicações que possam ser utilizadas em tablets e até mesmo em TVs é um desafio principalmente pelos diferentes tamanhos de tela. A API de Fragmentos se encaixa como uma luva para essas situações, nas quais existem dispositivos com telas de tamanhos variados, pois permite ajustes na interface gráfica possibilitando um melhor aproveitamento dos espaços disponíveis.

A API de Fragmentos

Um fragmento é um componente reutilizável que auxilia os programadores na criação de layouts para dispositivos com tamanhos de tela variados. Ele representa uma parte da interface gráfica em uma Activity que tem seu próprio ciclo de vida, recebe seus próprios eventos de entrada e pode ser adicionado ou removido enquanto a Activity está em execução.

Para utilizar um fragmento é preciso criar uma classe que estenda `android.app.Fragment` ou uma de suas subclasses, como `ListFragment`, `DialogFragment` ou `PreferenceFragment`. A Tabela 1 descreve essas classes.

Com o intuito de demonstrar o uso de fragmentos na prática, criaremos uma aplicação de teste que exibe uma mensagem utilizando fragmentos e outra mensagem sem usar esse recurso. Para isso, codificamos a classe `MeuFrament`, que estende de `Fragment`. Veja o código desta classe na Listagem 1.

Conforme indicado na Listagem 1, devemos reimplementar o método `onCreateView()`. Este método é responsável por carregar o layout que será exibido na tela. Ele recebe como parâmetro, entre outros, um objeto do tipo `LayoutInflater`, que transforma um layout definido em XML em uma classe `View`, procedimento feito pelo método `inflate()`, que possui três argumentos:

Reusando componentes com a API de Fragmentos

Classe	Descrição
Fragment	Classe padrão usada para estender o comportamento de um fragmento.
ListFragment	Representa um fragmento que possui uma lista de elementos com métodos para gerenciar a exibição, rolagem e lidar com os eventos de clique em cada elemento.
DialogFragment	Fragmento que exibe uma caixa de diálogo sobreposta à Activity corrente. Nela podemos definir um título, uma mensagem e botões de interação com o usuário.
PreferenceFragment	Fragmento usado para armazenar e acessar dados de configuração de uma aplicação. Com isso os aplicativos podem manter esses dados mesmo depois de fechados.

Tabela 1. Principais classes da API de Fragmentos

1. O identificador do layout que será usado;
2. Um objeto do tipo **ViewGroup**, que será o pai da **View** que está sendo gerada;
3. Um parâmetro booleano que, se **true**, indica que a **View** carregada deve ser associada ao **ViewGroup**.

Como pode ser observado, um dos parâmetros do método **inflate()** é o identificador do layout que será carregado pelo fragmento. Esse identificador aponta para um arquivo XML onde o layout é especificado. A **Listagem 2** apresenta um exemplo de como esse arquivo é definido.

Como podemos observar, a **Listagem 2** possui um layout que exibe um texto na tela do dispositivo quando a aplicação é carregada.

Listagem 3. Associando um fragmento a uma Activity.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/activity_str"/>

    <fragment android:name="labs.org.br.ExemploFragmento"
        android:id="@+id/fragment"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

A tag **<fragment>** é usada para declarar um fragmento dentro de um arquivo de layout. Ela possui os atributos **name**, **id** e **tag**, descritos na **Tabela 2**.

name	Especifica a classe Fragment que será instanciada e incorporada ao layout.
id	Identifica o fragmento por meio de um identificador numérico gerado pelo Android.
tag	Identifica o fragmento por meio de um identificador em formato de String.

Tabela 2. Atributos da tag **<fragment>**

No exemplo demonstrado na **Listagem 3**, percebemos que por meio do atributo **name** instanciamos a classe de fragmento da **Listagem 1** e exibimos o seu conteúdo junto com o layout da Activity. Vale lembrar que cada fragmento possui um identificador único definido pelo atributo **id** ou **tag**. Podemos empregar qualquer um dos dois para obter a instância do fragmento que está em uso, sendo que o último recebe como parâmetro um texto e o primeiro um número. Assim, usa-se o método **getFragmentManager()** para obter o objeto da classe **FragmentManager** e, por meio dos seus métodos **findFragmentById()** e **findFragmentByTag()**, recuperar a instância do fragmento que se pretende encontrar. A **Listagem 4** mostra como podemos acessar **MeuFragment**.

A linha 6 recupera a instância do fragmento por meio do método **findFragmentById()**, que recebe como parâmetro o identificador do fragmento procurado. No exemplo, o identificador do fragmento foi definido pelo atributo **id** na **Listagem 3**.

Listagem 1. Exemplo de uma subclasse de Fragment

```
01. package labs.org.br;
02.
03. import org.cesar.br.R;
04. import android.app.Fragment;
05. import android.os.Bundle;
06. import android.view.LayoutInflater;
07. import android.view.View;
08. import android.view.ViewGroup;
09.
10. public class MeuFragment extends Fragment {
11.
12.     @Override
13.     public View onCreateView(LayoutInflater inflater, ViewGroup container,
14.         Bundle savedInstanceState) {
15.         return inflater.inflate(R.layout.exemplo_fragmento, container, false);
16.     }
17. }
```

Listagem 2. Exemplo do arquivo de layout que será carregado pelo fragmento.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/fragmento_str"/>
</LinearLayout>
```

Até agora criamos um fragmento e definimos o seu layout. O próximo passo consiste em adicioná-lo à activity. Para isso, devemos declarar o fragmento no arquivo de layout da activity, como ilustra a **Listagem 3**.

Depois de elaborar o layout da activity contendo o fragmento, precisamos adicioná-lo à activity por meio do método `setContentView()`. Isso pode ser feito criando uma classe que estende de `Activity` e sobrescrevendo o método `onCreate()`, como pode ser observado na **Listagem 5**.

É no layout passado por parâmetro para o método `setContentView()` que definimos por meio da tag `<fragment>` qual fragmento será utilizado pela nossa Activity. Com isso finalizamos o nosso primeiro exemplo. Ao executá-lo, o resultado deve ser semelhante ao apresentado na **Figura 1**.

Listagem 4. Acessando um fragmento de dentro de uma Activity.

```

1.  @Override
2.  public void onCreate(Bundle savedInstanceState) {
3.      super.onCreate(savedInstanceState);
4.      setContentView(R.layout.main);
5.
6.      MeuFragment meuFragment = (MeuFragment) getSupportFragmentManager().
    findFragmentById(R.id.fragment);
7.  }
```

Listagem 5. Criando uma classe que estende Activity.

```

01. package labs.org.br;
02.
03. import org.cesar.br.R;
04.
05. import android.app.Activity;
06. import android.os.Bundle;
07.
08. public class MinhaActivity extends Activity {
09.     @Override
10.    public void onCreate(Bundle savedInstanceState) {
11.        super.onCreate(savedInstanceState);
12.        setContentView(R.layout.main);
13.    }
14. }
```



Figura 1. Aplicação exemplo mostrando os textos definidos na Activity e no Fragmento

Ciclo de vida de um fragmento

Uma aplicação Android geralmente possui um conjunto de activities ligadas entre si. Elas podem chamar umas às outras a fim de executar ações diferentes, tais como exibir uma listagem, abrir uma tela de cadastro, um tocador de músicas favoritas, acessar um site, etc. Desde o momento em que uma activity é exibida até ser interrompida, ela pode assumir basicamente os estados executando, pausado ou parado. Para controlar esses estados, assim como o seu ciclo de vida, é necessário o uso dos métodos: `onCreate()`, `onStart()`, `onRestart()`, `onPause()`, `onStop()` e `onDestroy()` da classe `Activity`.

Um fragmento também possui um ciclo de vida, sendo composto pelos mesmos métodos da activity, mas os métodos `onAttach()`, `onCreateView()`, `onActivityCreated()`, `OnDestroyView()` e `onDetach()`. Cabe ressaltar que este artigo assume que o leitor já está habituado com os métodos do ciclo de vida de uma activity, e foca apenas nos métodos exclusivos do ciclo de vida dos fragmentos.

O ciclo de vida do fragmento está intimamente relacionado ao da activity que o declarou, também chamada de *host-activity*. Desta forma, a chamada a um método do ciclo de vida da activity faz com que o mesmo método também seja invocado em todos os fragmentos dela. Por exemplo, quando o método `onResume()` de uma activity for chamado, o método `onResume()` de cada fragmento presente nesta activity também será.

A **Figura 2** mostra o fluxo do ciclo de vida de um fragmento.

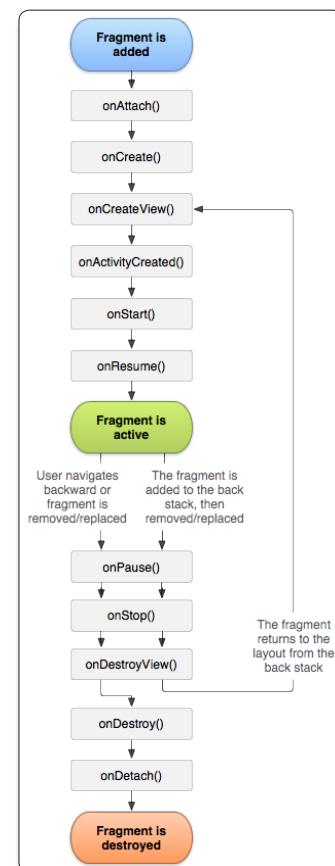


Figura 2. Ciclo de vida de um fragmento

Reusando componentes com a API de Fragmentos

Como podem ser observados, todos os métodos do ciclo de vida de uma activity também estão presentes no ciclo de vida de um fragmento. Entretanto, como já destacado, existem alguns métodos que são exclusivos a fragmentos, a saber:

- **onAttach(activity):** Método chamado assim que o fragmento é associado à activity. A activity que possui o fragmento é passada como parâmetro;
- **View onCreateView(inflater, viewgroup, bundle):** Método chamado no momento de carregar o layout do fragmento, devolve uma instância da classe View contendo o devido layout. A activity utiliza o retorno deste método para colocá-lo no espaço reservado para o fragmento;
- **onActivityCreated():** Chamado após o método **onCreate()** da activity ter sido finalizado;
- **onDestroyView():** Chamado quando a view associada ao fragmento está sendo removida;
- **onDetach():** Chamado quando o fragmento está sendo desvinculado da activity.

Biblioteca de compatibilidade

A API de Fragmentos do Android auxilia bastante os desenvolvedores na organização do código de uma Activity, mas o uso dela é possível apenas nas versões do Android 3.x ou superiores.

Com o objetivo de possibilitar o uso dos recursos de fragmentos na versão 2.x do Android, criou-se uma biblioteca de compatibilidade, um arquivo JAR que precisa ser adicionado à aplicação que está sendo construída. Para instalar esta biblioteca na aplicação, é necessário fazer o download do arquivo *android-support-v4.jar*, o que pode ser feito por meio do Android SDK Manager. Veja a Figura 3.

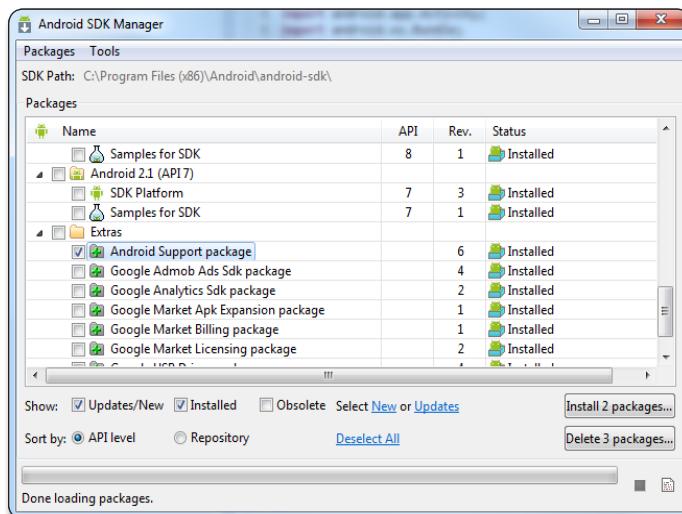


Figura 3. Instalando a biblioteca de compatibilidade

O Android SDK Manager possui um assistente de instalação que irá auxiliá-lo durante o processo de download. Após a instalação ser concluída, é necessário adicionar o JAR ao seu projeto. Isto é feito da seguinte forma:

1. Crie um diretório chamado *lib* na raiz do projeto;
2. Copie o arquivo *android-support-v4.jar*, localizado em *<sdk>/extras/android/support/v4/*, e cole na pasta *lib*;
3. Adicione o arquivo copiado ao build path do seu projeto.

Para criar aplicações em versões do Android que não possuem suporte nativo de fragmentos, se faz necessário o uso da biblioteca de compatibilidade. Desta forma, usaremos as classes disponíveis no pacote **android.support.v4**. Para isso, duas alterações devem ser feitas no código fonte: 1) estender a classe **android.support.v4.FragmentActivity** ao invés de **Activity**; e 2) utilizar o método **getSupportFragmentManager()** ao invés de **getFragmentManager()**. O método **getSupportFragmentManager()** retorna uma instância da classe **android-support.v4.app.FragmentManager**, responsável por gerenciar os fragmentos que estão sendo utilizados na aplicação.

Leitor de notícias via RSS

RSS (*Really Simple Syndication*) é uma tecnologia baseada em XML que permite a distribuição em tempo real do conteúdo de um website para milhares de outros em todo o mundo. Dessa forma é possível, entre outras coisas, agregar informações de diversos sites sem a necessidade de visitar um a um, realizar o download delas para o usuário acessá-las mesmo se não estiver conectado a internet e exibi-las em aplicações de diversas plataformas (*web, mobile, desktop, etc.*). Na prática, os websites fornecem suas notícias ou novidades em arquivos no formato XML, conhecidos como *feeds*. Neles, geralmente são encontradas informações como título, resumo, data e hora e um link para esses conteúdos.

Visando apresentar os principais conceitos da API de Fragmentos, construiremos uma aplicação para leitura de notícias. Nela listaremos os títulos das notícias de um RSS e possibilitaremos ao usuário visualizar os detalhes da mesma após selecioná-la, abrindo-a em outra tela. A Figura 4 mostra como ficará a aplicação em um smartphone.



Figura 4. Aplicação rodando em um smartphone

No smartphone, devido a uma área de visualização mais limitada, a lista de notícias é exibida em uma activity e o conteúdo delas é exibido outra. Já na versão para tablets, como não temos o problema relacionado ao tamanho da tela, tanto a lista de notícias quanto o conteúdo de cada uma são mostrados na mesma activity, como pode ser conferido na **Figura 5**.

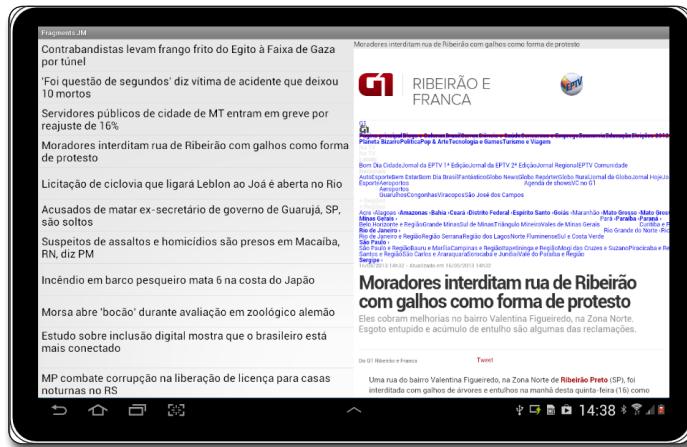


Figura 5. Aplicação rodando em um tablet

Para criar uma aplicação visando o reuso de componentes, separação de responsabilidades e baixo acoplamento, construiremos dois fragmentos: um que irá listar as notícias do RSS e outro que exibirá os detalhes da notícia selecionada. Além disso, é importante frisar que os fragmentos serão independentes e se comunicarão via parâmetros, o que nos permite definir comportamentos específicos para smartphones e tablets.

Para iniciarmos a construção do leitor de RSS, como de costume, é preciso criar o projeto. Nele vamos utilizar a API de compatibilidade de fragmentos. No Eclipse Juno, a biblioteca de compatibilidade já é incorporada automaticamente. No entanto, caso você utilize outras versões e isso não aconteça, deve-se adicionar o JAR ao build path do projeto, conforme explicado no tópico Biblioteca de Compatibilidade.

Em seguida, é necessária a criação de uma classe que busque as informações no *feed* e nos retorne uma lista de itens de RSS para que sejam exibidos na tela. Para isso, implementamos uma classe chamada **RSSItem**, que representa uma notícia do RSS consultado. Ela contém o título e a URL que, quando selecionada, exibirá seu conteúdo em um **WebView**. A **Listagem 6** mostra o código dessa classe.

CURSOS ONLINE

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



Para mais informações :

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

(21) 3382-5038

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)**



DEV MEDIA

Reusando componentes com a API de Fragmentos

Nota

WebView é um componente do Android que exibe uma página web. Ele permite a construção de um browser ou a exibição de qualquer conteúdo online em uma activity.

Listagem 6. Classe que representa um item de RSS.

```
01. package br.com.jm.fragments.rss;
02. import java.io.Serializable;
03.
04. public class RSSItem implements Serializable {
05.
06.     public static final String RSS_ITEM_PARAM = "RSS_ITEM_PARAM";
07.     private String title;
08.     private String url;
09.
10.    public RSSItem(String title, String url){
11.        this.title = title;
12.        this.url = url;
13.    }
14.    public String getTitle() {
15.        return title;
16.    }
17.    public String getUrl() {
18.        return url;
19.    }
20.    @Override
21.    public String toString() {
22.        return getTitle();
23.    }
24. }
```

Após isso, criaremos a classe **RSS**. Basicamente, ela implementa uma thread para buscar os dados no feed, tendo em vista que essa consulta pode demorar bastante, e é capaz de retornar esses dados em um **ArrayList** de itens de RSS representados por instâncias de **RSSItem**. Veja o código dessa classe na **Listagem 7**.

Nesta classe definimos uma interface chamada **ResultListener** (linhas 15-17) com o método **onResult(ArrayList<RSSItem>)**, chamado quando os dados forem retornados da consulta feita pela classe **AsyncTaskRSS**, subclasse de **AsyncTask**.

AsyncTask é uma recomendação do Android quando há necessidade de trabalhar com threads. Ela nos permite criar uma thread separada (em background) e atualizar os dados da thread de UI posteriormente. Cabe ressaltar que toda e qualquer operação que possa levar muito tempo em sua execução deve ser processada em uma thread, evitando assim problemas de ANR (*Application Not Responding*).

Diante disso, para criar uma thread devemos estender a classe **AsyncTask** e definir três tipos genéricos. O primeiro representa quais objetos serão passados para a execução da tarefa, o segundo indica o progresso da tarefa que está sendo executada e o terceiro define o que será recebido como resultado da operação. Caso um dos parâmetros não seja utilizado, deve-se definir o seu tipo como **Void**. Além disso, é preciso sobrescrever o método **doInBackground()**, onde são feitas as operações que podem durar muito

Listagem 7. Código da classe responsável por consultar dados no RSS.

```
01. package br.com.jm.fragments.rss;
02.
03. import java.io.IOException;
04. import java.util.ArrayList;
05. import javax.xml.parsers.DocumentBuilder;
06. import javax.xml.parsers.DocumentBuilderFactory;
07. import javax.xml.parsers.ParserConfigurationException;
08. import org.w3c.dom.Document;
09. import org.w3c.dom.NodeList;
10. import org.xml.sax.SAXException;
11. import android.os.AsyncTask;
12.
13. public class RSS {
14.
15.     public interface ResultListener {
16.         public void onResult(ArrayList<RSSItem> list);
17.     }
18.
19.     public void getNews(String url, ResultListener resultListener){
20.
21.         AsyncTaskRSS asyncTaskRSS = new RSS.AsyncTaskRSS();
22.         asyncTaskRSS.execute(url, resultListener);
23.     }
24.
25.     class AsyncTaskRSS extends AsyncTask<Object, Void, ArrayList<RSSItem>>{
26.
27.         ResultListener resultListener;
28.
29.         @Override
30.         protected ArrayList<RSSItem> doInBackground(Object... params) {
31.
32.             String url = (String) params[0];
33.             resultListener = (ResultListener) params[1];
34.             Document doc = null;
35.             DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
36.             DocumentBuilder db;
37.
38.             try {
39.                 db = dbf.newDocumentBuilder();
40.                 doc = db.parse(url);
41.             } catch (ParserConfigurationException e) {
42.                 e.printStackTrace();
43.             } catch (IOException e) {
44.                 e.printStackTrace();
45.             } catch (SAXException e) {
46.                 e.printStackTrace();
47.             }
48.
49.             NodeList listItem = doc.getElementsByTagName("item");
50.             ArrayList<RSSItem> itemList = new ArrayList<RSSItem>();
51.
52.             for(int i = 0; i < listItem.getLength(); i++){
53.                 //Title
54.                 String title = listItem.item(i).getchildNodes().item(0).getchildNodes().item(0).getNodeValue();
55.
56.                 //Link
57.                 String urlItem = listItem.item(i).getchildNodes().item(1).getchildNodes().item(0).getNodeValue();
58.
59.                 itemList.add(new RSSItem(title, urlItem));
60.             }
61.
62.             return itemList;
63.         }
64.
65.         @Override
66.         protected void onPostExecute(ArrayList<RSSItem> resultList) {
67.             resultListener.onResult(resultList);
68.             super.onPostExecute(resultList);
69.         }
70.     }
71. }
```

tempo, dentre elas requisições web. Para receber o resultado do processamento efetuado pela thread, faz-se necessário reescrever o método **onPostExecute()**, executado após o término do **doInBackground()**.

No **AsyncTaskRSS** (linhas 25-70), utilizamos **Object** como o primeiro tipo genérico. Nele recebemos uma **String** que representa a URL do RSS e um **ResultListener** para notificar quando a resposta da busca dos dados for obtida. Como não estamos exibindo o progresso da tarefa para o usuário, o segundo tipo foi definido como **Void**. E para o terceiro tipo, utilizamos um **ArrayList<RSSItem>**, que possui o resultado da consulta.

Tendo em vista que uma consulta a um RSS nada mais é que uma leitura de um XML, precisamos extrair os dados do arquivo de modo que possamos exibi-los para o usuário. Assim, utilizamos a API DOM (Document Object Model) para manipular o documento XML. Nela, para realizar o parser, primeiro obtemos uma instância de **DocumentBuilderFactory** a partir do método **newInstance()**.

Após isso, criamos um **DocumentBuilder** por meio do método **newDocumentBuilder()** de **DocumentBuilderFactory**. A classe **DocumentBuilder** tem um método chamado **parse()** que recebe a URL ou o local do arquivo XML e retorna uma instância da classe **Document**, que representa um novo objeto DOM com os dados.

Com a instância de um objeto **Document** em mãos (linha 40), podemos obter um **NodeList** através do método **getElementsByTagName(String)**. O **NodeList** contém a lista de itens do RSS, onde cada item é composto por Título e URL. Após isso, iteramos sobre a lista (linhas 52 a 60) e na iteração invocamos o método **getChildNodes()** para obter todos os filhos de cada item e seus respectivos valores através do método **getNodeValue()**. De posse do título e da URL da notícia, adicionamos os dados em uma lista de objetos **RSSItem** e a retornamos em seguida (linhas 54 a 62).

Após o término da execução do **doInBackground()**, o método **onPostExecute()** é invocado recebendo a lista de resultados. Essa lista é passada para o listener por meio do método **onResult(ArrayList<RSSItem>)**.

Por fim, na classe **RSS** implementaremos o método **getNews(String, ResultListener)**. Este receberá a URL e a implementação do listener que ouvirá e receberá os dados consultados no feed. Dentro desse método instanciamos a classe interna **AsyncTaskRSS** e chamamos o método **execute()** para iniciar a thread.

Definindo os fragmentos da aplicação

Agora que temos a infraestrutura para buscar as informações, iremos implementar os fragmentos. Como informado anteriormente, criaremos dois deles: um que será a lista de notícias, o qual chamamos de **FragmentRSSList**, e outro chamado de **FragmentDetail**, que exibirá o título e a página da notícia.

Para construir os fragmentos, basicamente iremos criar classes que estendam **android.support.v4.app.Fragment** da API de com-

patibilidade e, após isso, reescrever o método **onCreateView()**, carregando o layout do fragmento a partir de um arquivo de recurso XML.

Criando o FragmentRSSList

Para começar, criaremos o layout do fragmento responsável por exibir a lista de notícias disponibilizada pelo RSS. Esse layout será composto por um simples **LinearLayout** com um **ListView**, como apresentado na **Listagem 8**.

Listagem 8. Layout do fragmento que exibirá a lista de notícias.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ListView
        android:id="@+fragmentView/listView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
    </ListView>

</LinearLayout>
```

Em seguida, implementaremos a classe **FragmentRSSList**, que controlará o funcionamento da listagem de notícias. A **Listagem 9** apresenta o código desta classe.

Na construção do fragmento **FragmentRSSList**, exibido na **Listagem 9**, primeiramente definimos uma interface que representará um listener, chamada **ItemClickListener** (linhas 21 a 23). Ela será necessária para notificar a aplicação quando os dados consultados forem retornados. Assim, toda e qualquer activity que deseje utilizar o fragmento para listar notícias terá que implementar o método **onClickItem(RSSItem)**, recebendo o objeto **RSSItem** que foi selecionado pelo usuário.

Após isso criaremos o método **loadNews(ItemListener)**, que receberá o listener que será disparado pelo fragmento quando algum item da lista for selecionado (linhas 31 a 35). Esse **ItemListener** será guardado em um membro da classe (linha 32) para ser usado mais à frente. Em seguida instanciamos a classe **RSS** invocando o método **getNews()**, que recebe dois parâmetros. São eles:

- 1. String url:** URL do RSS na qual devem ser buscadas as notícias;
- 2. ResultListener resultListener:** Classe que contém um método que será invocado quando a consulta retornar os dados.

Para passarmos um **ResultListener** para o método **getNews()** é necessário que nossa classe **FragmentRSSList** implemente a interface **ResultListener** e reescreva o método **onResult(ArrayList<RSSItem>)**, que receberá a lista de itens do RSS retornados pelo feed. Estes itens são usados para criar um Adapter que posteriormente será atribuído ao componente **ListView** do nosso fragmento, para serem exibidos na tela (linhas 45 a 57).

Reusando componentes com a API de Fragmentos

Listagem 9. Código da classe FragmentRSSList.

```
01. package br.com.jm.fragments;
02.
03. import java.util.ArrayList;
04. import android.os.Bundle;
05. import android.support.v4.app.Fragment;
06. import android.view.LayoutInflater;
07. import android.view.View;
08. import android.view.ViewGroup;
09. import android.widget.AdapterView;
10. import android.widget.AdapterView.OnItemClickListener;
11. import android.widget.ArrayAdapter;
12. import android.widget.ListView;
13. import br.com.jm.fragments.rss.RSS;
14. import br.com.jm.fragments.rss.RSS.ResultListener;
15. import br.com.jm.fragments.rss.RSSItem;
16.
17. public class FragmentRSSList extends Fragment implements ResultListener {
18.
19.     private ItemListener mItemListener;
20.
21.     public interface ItemListener {
22.         public void onClickItem(RSSItem item);
23.     }
24.
25.     @Override
26.     public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
27.         View view = inflater.inflate(R.layout.list_rss_fragment, container, false);
28.         return view;
29.     }
30.

31.     public void loadNews(ItemListener itemListener){
32.         this.mItemListener = itemListener;
33.         RSS rss = new RSS();
34.         rss.getNews("http://g1.globo.com/dynamo/rss2.xml",this);
35.     }
36.
37.     public boolean isInTablet(){
38.         if (getActivity().getResources().getBoolean(R.bool.isTablet))
39.             return true;
40.         else
41.             return false;
42.     }
43.
44.     @Override
45.     public void onResult(ArrayList<RSSItem> resultlist) {
46.         ArrayAdapter<RSSItem> adapter = new ArrayAdapter<RSSItem>(getActivity(),
47.             android.R.layout.simple_list_item_1, resultlist);
48.         listView.setAdapter(adapter);
49.         listView.setOnItemClickListener(new OnItemClickListener() {
50.             @Override
51.             public void onItemClick(AdapterView<?> adapterView, View view, int position,
52.             long arg3) {
53.                 RSSItem rssItem = (RSSItem) adapterView.getItemAtPosition(position);
54.                 mItemListener.onClickItem(rssItem);
55.             }
56.         });
57.     }
58. }
```

Agora devemos identificar quando um item do `ListView` foi selecionado. Para isso, implementamos o método `setOnItemClickListener(OnItemClickListener)` do `ListView` por meio de uma classe anônima e recuperamos o `RSSItem` clicado (linhas 49 a 56). Em seguida, notificamos a nossa interface `ItemListener` por meio do método `onClickItem(RSSItem)`, e passamos o item de RSS que foi clicado no fragmento para a activity.

No intuito de diferenciar se o fragmento está em um Tablet ou não, a classe `FragmentRSSList` possui o método `isInTablet()`, que permite carregar um layout conforme as características do dispositivo (linhas 37 a 42).

Após a implementação da classe `FragmentRSSList`, podemos adicioná-la em qualquer tela da aplicação. Como dito anteriormente, neste exemplo utilizaremos fragmentos declarados em XML, porém também é possível adicionar e substituir fragmentos em tempo de execução por meio da API de fragmentos. A **Listagem 10** mostra como inserir o fragmento na tela principal da aplicação (`layout/activity_main.xml`). Esse arquivo será usado como layout no smartphone.

Criando o FragmentDetail

Construiremos agora o fragmento responsável por exibir a descrição da notícia e a página da mesma na tela. A **Listagem 11** mostra como ficará o layout dos detalhes da notícia, composto por um `TextView` e um `WebView`.

Listagem 10. Layout da tela principal usada no smartphone.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment
        android:id="@+id/fragment_list"
        android:name="br.com.jm.fragments.FragmentRSSList"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:tag="fragment_list"/>

</RelativeLayout>
```

Listagem 11. Layout do fragmento que exibirá o título e o detalhe da notícia.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView android:id="@+id/txtTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

    <WebView
        android:id="@+id/webViewDetail"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"/>

</LinearLayout>
```

Após a definição do layout, implementaremos a classe responsável por receber o título e a URL da notícia e exibi-los. A **Listagem 12** apresenta como ficará a classe **FragmentDetail**.

Listagem 12. Código da classe FragmentDetail, que exibe as informações da notícia.

```
01. package br.com.jm.fragments;
02.
03. import android.os.Bundle;
04. import android.support.v4.app.Fragment;
05. import android.view.LayoutInflater;
06. import android.view.View;
07. import android.view.ViewGroup;
08. import android.webkit.WebView;
09. import android.widget.TextView;
10. import br.com.jm.fragments.rss.RSSItem;
11.
12. public class FragmentDetail extends Fragment {
13.
14.     private WebView webView;
15.     private TextView txtTitle;
16.
17.     @Override
18.     public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
19.         View view = inflater.inflate(R.layout.detail_fragment, container, false);
20.         return view;
21.     }
22.
23.     @Override
24.     public void onActivityCreated(Bundle savedInstanceState) {
25.         super.onActivityCreated(savedInstanceState);
26.         webView = (WebView) getView().findViewById(R.id.webViewDetail);
27.         txtTitle = (TextView) getView().findViewById(R.id.txtTitle);
28.     }
29.
30.     public void loadData(RSSItem rssItem){
31.         webView.loadUrl(rssItem.getUrl());
32.         txtTitle.setText(rssItem.getTitle());
33.     }
34. }
```

Após estendermos a classe **Fragment** e inflarmos a view do nosso layout, implementamos o método **onActivityCreated(Bundle)**. Este indica que a activity que contém o fragmento já foi criada. Dessa forma podemos recuperar os componentes do layout pelo ID. Em seguida, criamos o método **loadData(RSSItem)**, que recebe um item de RSS como parâmetro. Assim, podemos invocar os métodos **setText(String)** e **loadUrl(String)** dos componentes (linhas 31 e 32) para atribuir os valores vindos no parâmetro **RSSItem**.

Fragments em Tablets

Como citado anteriormente, o layout que será exibido no smartphone é diferente do apresentado no Tablet. Portanto, para diferenciarmos os layouts, é necessário criar pastas de recursos diferentes. No nosso caso, criamos a pasta *layout-xlarge* e colocamos nela o layout descrito na **Listagem 13** (*layout-xlarge/activity_main.xml*).

Para produzirmos uma exibição diferente em um tablet, codificamos na **Listagem 13** dois fragmentos, o fragmento da lista e o fragmento de detalhe, na mesma tela. Colocamos cada declaração desses fragmentos dentro de um **LinearLayout** e utilizamos o atributo **android:layout_weight="1"** para dividir a tela ao meio.

Listagem 13. Layout principal em um tablet.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_weight="1">

        <fragment
            android:id="@+id/fragment_list"
            android:name="br.com.jm.fragments.FragmentRSSList"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:tag="list_tablet"/>

    </LinearLayout>

    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_weight="1">

        <fragment
            android:id="@+id/fragment_detail"
            android:name="br.com.jm.fragments.FragmentDetail"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:tag="detail"/>

    </LinearLayout>

</LinearLayout>
```

Construindo as activities da aplicação

Nos tópicos anteriores, ensinamos como criar os fragmentos e os layouts que serão usados na aplicação. Agora, será apresentado o código das activities empregadas no leitor de notícias. Elas permitem que a lógica de exibição seja diferente em smartphones e tablets utilizando o conceito de fragmentos.

A classe MainActivity

Para a tela principal, criamos a classe **MainActivity**, que é responsável por listar as notícias. Como pode ser observado, ela estende **FragmentActivity** para que tenhamos acesso ao método **getSupportFragmentManager()**. A **Listagem 14** mostra a implementação dessa classe, onde após setarmos o layout, na linha 6, utilizamos o método **getSupportFragmentManager()**, que retorna um objeto que disponibiliza alguns métodos para recuperação de fragmentos na tela, dentre eles o **findFragmentById(int idRes)**. Logo após, verificamos se o fragmento não é nulo e se o mesmo foi

Reusando componentes com a API de Fragmentos

encontrado no layout (linha 16). Em seguida invocamos o método **loadNews(ItemListener)**, responsável por buscar os dados no feed, e implementamos uma classe anônima para decidir o que deve ser feito quando um item for selecionado.

Listagem 14. Código da activity principal da aplicação.

```
01. public class MainActivity extends FragmentActivity {  
02.  
03.     @Override  
04.     protected void onCreate(Bundle savedInstanceState) {  
05.         super.onCreate(savedInstanceState);  
06.         setContentView(R.layout.activity_main);  
07.  
08.         final FragmentRSSList fragmentRSSList = (FragmentRSSList)  
09.             getSupportFragmentManager().findFragmentById(R.id.fragment_list);  
10.  
11.         if (fragmentRSSList != null && fragmentRSSList.isInLayout()){  
12.             fragmentRSSList.loadNews(new FragmentRSSList.ItemListener() {  
13.                 @Override  
14.                 public void onClickItem(RSSItem item) {  
15.                     if (fragmentRSSList.isInTablet()){  
16.                         FragmentDetail fragmentDetail = (FragmentDetail)  
17.                             getSupportFragmentManager().findFragmentById(R.id.fragment_detail);  
18.                         if (fragmentDetail != null && fragmentDetail.isInLayout()){  
19.                             fragmentDetail.loadData(item);  
20.                         }  
21.                     Intent intent = new Intent(MainActivity.this, DetailActivity.class);  
22.                     intent.putExtra(RSSItem.RSS_ITEM_PARAM, item);  
23.                     startActivity(intent);  
24.                 }  
25.             });  
26.         }  
27.     }  
28. }
```

A implementação da classe anônima (linhas 11 a 25) leva em consideração se o **FragmentRSSList** está sendo executado em um smartphone ou em um tablet. Para isso utilizamos o método **isInTablet()** da classe **FragmentRSSList**. Caso seja um tablet, devemos recuperar a instância de **FragmentDetail** e invocar o método **loadData(RSSItem)** (linhas 14 a 18), que por sua vez receberá os parâmetros e cuidará de exibir os dados. Porém, caso não seja um tablet, a aplicação terá que abrir uma activity que contém o **FragmentDetail**. Dessa forma, iniciamos a activity **DetailActivity** e passamos para ela o objeto **RSSItem** (linhas 20 a 22). Com isso conseguimos diferenciar os layouts de smartphones e tablets, possibilitando o reuso de componentes a partir dos fragmentos criados.

A classe **DetailActivity**

Quando acessarmos a aplicação de um smartphone, ela terá duas activities: uma que mostrará a lista de notícias e a outra os detalhes da notícia. A **Listagem 15** apresenta a implementação da classe responsável por mostrar o detalhamento das notícias em um smartphone.

Além de estendermos a classe **FragmentActivity**, implementamos dois métodos: **onCreate(Bundle)** e **onStart()**. No método

onCreate(), simplesmente inflamos o layout que contém o fragmento **DetailFragment** declarado. No método **onStart()**, exibido entre as linhas 14 e 28, obtemos a instância de **DetailFragment** e os parâmetros enviados pela Intent através do método **getExtras()** do objeto **Bundle**. Desta forma, conseguimos recuperar o **RSSItem** passado e assim invocar o método **loadData(RSSItem)** enviando o item como parâmetro.

Listagem 15. Tela de exibição de notícias.

```
01. package br.com.jm.fragments;  
02. import android.os.Bundle;  
03. import android.support.v4.app.FragmentActivity;  
04. import br.com.jm.fragments.rss.RSSItem;  
05.  
06. public class DetailActivity extends FragmentActivity {  
07.  
08.     @Override  
09.     protected void onCreate(Bundle savedInstanceState) {  
10.         super.onCreate(savedInstanceState);  
11.         setContentView(R.layout.activity_detail);  
12.     }  
13.  
14.     @Override  
15.     protected void onStart() {  
16.         super.onStart();  
17.  
18.         FragmentDetail fragmentDetail = (FragmentDetail)  
19.             getSupportFragmentManager().findFragmentById(R.id.fragment_detail);  
20.  
21.         if (fragmentDetail != null && fragmentDetail.isInLayout()){  
22.             Bundle bundle = getIntent().getExtras();  
23.  
24.             if (bundle != null){  
25.                 RSSItem rssItem = (RSSItem) bundle.getSerializable(RSSItem.RSS_ITEM_  
26. PARAM);  
27.                 fragmentDetail.loadData(rssItem);  
28.             }  
29.         }  
29.     }
```

Criando fragmentos dinamicamente

Existem duas formas de definir um fragmento: por meio da tag **<fragment>** que é usada no arquivo XML de layout e através da API de fragmentos. Nesta, é possível adicionar, remover ou substituir um fragmento em tempo de execução. Para isto, devemos utilizar as classes **FragmentManager** e **FragmentTransaction** da API de fragmentos.

A classe **FragmentManager** fornece uma instância de **FragmentTransaction** através do método **beginTransaction()**. De posse da transação, pode-se adicionar, remover ou substituir um fragmento em um layout qualquer da tela invocando os métodos **add()**, **remove()** e **replace()** da própria **FragmentTransaction**. Por fim, o método **commit()**, também da classe **FragmentTransaction**, deve ser chamado para confirmar a operação.

Para demonstrar como adicionar um fragmento em tempo de execução, a **Listagem 16** apresenta um trecho de código exemplificando o uso do método **add()** de **FragmentTransaction**.

Listagem 16. Adicionando um fragmento em tempo de execução.

```
FragmentManager fragmentManager = getFragmentManager()
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment, "exemploFragment");
fragmentTransaction.commit();
```

No código da **Listagem 16**, como já informado, o método **getFragmentManager()** devolve uma instância de **FragmentManager**. Esse método está presente nas classes **Activity** e **Fragment** e, portanto, qualquer subclasse destas tem acesso a ele. Em seguida, um **FragmentTransaction** é retornado pela chamada ao método **beginTransaction()** de **FragmentManager**. Após isso, o fragmento **ExampleFragment** é instanciado e adicionado dinamicamente à **Activity** depois da chamada ao método **add()** de **FragmentTransaction**. No final, o método **commit()** confirma a operação de adição do fragmento.

Como pode ser verificado, a versão do método **add()** que está sendo usada possui três parâmetros. No primeiro, deve ser informado o identificador do layout onde o fragmento será adicionado. No segundo, temos uma instância da classe **Fragment**, que representa o fragmento a ser adicionado. Por último, temos uma **String** descrevendo o nome da tag, um identificador opcional do fragmento, para que seja possível encontrá-lo futuramente utilizando o método **findFragmentByTag(tag)**.

O método **add()** possui também uma versão com dois parâmetros (o layout e o fragmento). Ele tem o mesmo efeito de invocar a versão com três parâmetros passando o valor **null** no parâmetro da tag. Neste caso, no entanto, não será possível achar o fragmento por meio do método **findFragmentByTag(tag)**.

Caso o desenvolvedor deseje remover um fragmento de uma **Activity**, isso pode ser feito obtendo-se uma instância de **FragmentManager**, que por sua vez fornece um **FragmentTransaction**, pelo qual acessamos o método **remove()**. Este método recebe como parâmetro apenas um objeto do tipo **Fragment**, representando o fragmento a ser removido. Após isso, a chamada ao método **commit()** assegura a remoção (ver **Listagem 17**).

Além da remoção, é possível a substituição de um fragmento dinamicamente, como mostra a **Listagem 18**.

Listagem 17. Removendo um fragmento em tempo de execução.

```
FragmentManager fragmentManager = getFragmentManager()
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
ExampleFragment fragment = // recupera o fragmento a ser removido
fragmentTransaction.remove(fragment);
fragmentTransaction.commit();
```

Listagem 18. Substituindo um fragmento em tempo de execução.

```
FragmentManager fragmentManager = getFragmentManager()
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.replace(R.id.fragment_container, fragment,
"exemploFragment");
fragmentTransaction.commit();
```

A sequência de chamadas de métodos, no caso da substituição de um fragmento por outro, é exatamente a mesma usada para adicionar ou removê-lo. A única diferença é que, ao invés de chamar o método **add()** ou **remove()**, invocamos o método **replace()**. A versão desse método empregada no exemplo recebe três parâmetros, assim como o **add()**. São eles: o identificador do layout, onde se encontra o fragmento a ser substituído e o novo fragmento deve ser mostrado, a instância do novo fragmento e uma tag para que seja possível encontrá-lo depois.

Há também outra versão do método **replace()**, apenas com dois parâmetros: o identificador do layout e o novo fragmento. Este método, internamente, chama a versão do **replace()** com três parâmetros, passando o valor **null** no parâmetro da tag.

Fragment Back Stack

O comportamento padrão do Android ao pressionar o botão voltar é de destruir a activity que está sendo mostrada, removendo-a da pilha de activities. Caso a activity contenha fragmentos, eles consequentemente também serão destruídos.

No entanto, às vezes é necessário que o botão voltar simplesmente desfaça uma operação efetuada pelo **FragmentTransaction**, antes de fechar a activity atual. Para casos assim, o Android permite adicionar cada transação em uma pilha de controle para o botão voltar, a **back stack**. Com isso, o botão voltar primeiro desfaz as transações realizadas, até que não haja mais nenhuma transação na pilha de controle, e em seguida, fecha a activity. O método **addToBackStack()** de **FragmentTransaction** é usado para adicionar a transação na back stack.

Como visto, uma transação, representada pela classe **FragmentTransaction**, consiste em um conjunto de mudanças aplicado aos fragmentos de uma activity ao mesmo tempo, tais como: adição, remoção ou substituição de fragmentos. Tais mudanças só são efetivadas mediante a chamada ao método **commit()**. No entanto, para colocar a transação na **back stack**, antes do **commit()**, o método **addToBackStack()** deve ser invocado. Isto permite que as alterações da transação sejam desfeitas ao acionar o botão voltar. Veja a **Listagem 19**.

Listagem 19. Adicionando uma transação na back stack.

```
FragmentManager fragmentManager = getFragmentManager()
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
ExampleFragment newFragment = new ExampleFragment();
fragmentTransaction.replace(R.id.fragment_container, newFragment,
"exemploFragment");
fragmentTransaction.addToBackStack ("replaceFrag");
fragmentTransaction.commit();
```

Nesse trecho de código, o fragmento **newFragment** substitui qualquer outro que esteja adicionado no layout cujo identificador é **R.id.fragment_container**. A chamada ao método **addToBackStack()** faz com que a operação de substituição seja inserida na **back stack** para que o usuário possa reverter a transação e trazer de volta o fragmento anterior pressionando o botão voltar.

Reusando componentes com a API de Fragmentos

Vale lembrar que ao chamar o **addToBackStack()**, todas as mudanças (adicionar, remover ou substituir fragmentos) feitas na transação são inseridas na *back stack* como uma simples transação, de forma que o acionar do botão voltar irá desfazer todas as alterações de uma só vez. O método **addToBackStack(name)** recebe um parâmetro opcional do tipo **String** que serve para identificar a transação na *back stack*. Para este parâmetro pode ser atribuído o valor **null** caso não se deseje identificar a transação.

É importante salientar que o fato de não chamar o método **addToBackStack()** ao efetuar uma operação de remoção de fragmento, implica na destruição deste caso haja uma confirmação da mesma, por meio do **commit()**. Isto faz com que não seja mais possível navegar por esse fragmento. Por outro lado, se o método **addToBackStack()** for chamado antes de confirmar a remoção de um fragmento, então o fragmento é apenas interrompido e pode ser retomado quando necessário.

Conclusão

O crescente número de aparelhos Android faz com que os desenvolvedores se preocupem em criar aplicativos que se adaptem aos mais variados tamanhos de tela. Para auxiliá-los nessa tarefa, foi criada a API de fragmentos. Os fragmentos são componentes reutilizáveis que possuem um ciclo de vida próprio e podem ser programados para adaptar-se a vários tipos de tela.

Com base nesse contexto, este artigo abordou essa API e como ela pode ser usada para criar aplicações que tenham seus layouts adaptáveis em vários tamanhos de tela. Além disso, foi apresentado como criar um fragmento, tanto via declaração no arquivo XML de layout quanto dinamicamente via código, o ciclo de vida do fragmento, a relação deste com o ciclo de vida da activity que o adicionou e como utilizar a biblioteca de compatibilidade, que permite usar a API de fragmentos em versões do Android mais antigas, visto que a API de fragmentos surgiu na versão 3.x (Honeycomb).

Com objetivo de assimilar esses conceitos e mostrar na prática como utilizá-los, foi desenvolvida uma aplicação que usa fragmentos. Nela, o usuário pode navegar em uma lista com títulos de notícias obtidas de um RSS e, ao clicar em uma notícia, são exibidos os seus detalhes. Essa aplicação foi projetada para funcionar em dois dispositivos com telas de tamanhos diferentes: um tablet e um smartphone. Para isso, foi necessária a criação de dois fragmentos: um responsável pelo layout que exibirá a lista com os títulos das notícias e outro com o layout para exibir os detalhes da notícia selecionada. No caso do tablet, como este possui uma grande área útil para a aplicação, ela coube em apenas

uma activity, exibindo à esquerda o fragmento da lista de títulos e à direita o fragmento com os detalhes. Já no smartphone, como a tela é menor, foram utilizadas duas activities, uma para exibir a lista de notícias e outra para exibir os detalhes da notícia desejada. A grande vantagem, neste caso, é que tanto no tablet quanto no smartphone foram usados os mesmos fragmentos, apenas dispositivos de formas diferentes, aumentando assim o reuso e evitando a duplicação de código.

Autor



Aécio Leite Vieira da Costa

contato@aeciocosta.com.br | www.aeciocosta.com.br

Possui graduação em Análise de Sistemas pelo Centro Universitário CESMAC (2009). Pós-graduação em Planejamento e Gestão Organizacional na FCAP – UPE. Atua como Engenheiro de Sistemas (C.E.S.A.R – www.cesar.org.br) desde 2011 e é Professor Titular da

Faculdade dos Guararapes.



Autor



Edilson Mendes Bizerra Junior

edilsonmendes@gmail.com

Mestre em Engenharia da Computação pela Universidade de Pernambuco (UPE). Trabalha na área de TI desde 2003. Atualmente atua como Engenheiro de Sistemas do Centro de Estudos e Sistemas Avançados do Recife (C.E.S.A.R – www.cesar.org.br) e Professor Titular da FAFICA.



Autor



Luiz Artur Botelho da Silva

arturbotelho_4@yahoo.com.br

Mestrando em Ciência da Computação pela UFPE. Pós-graduado em Engenharia de Software pela Faculdade Boa Viagem (FBV) e formado em Análise de sistemas pela Universidade Salgado de Oliveira, trabalha com TI desde 2002. Atualmente atua como Engenheiro de Sistemas do Centro de Estudos e Sistemas Avançados do Recife (C.E.S.A.R – www.cesar.org.br) e Professor Titular da FAFICA.



Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!





DEVMEDIA

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior
portal para
desenvolvedores
da América
Latina!



20
mil
posts

430
mil
cadastrados

10
milhões de
page-views
por mês

Desenvolva web services com o Play Framework

Serviços RESTful sem demora com o Play!

Desde o seu lançamento, o Play Framework vem chamando a atenção de muitos desenvolvedores. Não por ser “apenas mais um framework web com uma nova proposta”, como diriam os críticos contrários à proliferação de frameworks. Mas porque, de fato, revolucionou a forma como construímos aplicativos desta natureza. Para aqueles acostumados com o mundo JSF, por exemplo, o Play impressiona por sua simplicidade. Atualmente, ele é chamado apenas de Play e está com os holofotes ainda mais focados nele, pois a nova série 2.X agregou mais novidades, o que o leva a ser uma forte opção para o desenvolvimento de seu próximo projeto para a web.

O Play também é excelente para o desenvolvimento de serviços RESTful, pois já tem suporte nativo e facilita substancialmente a criação deles. Mas, não é apenas por ser fácil de usar, o Play é muito robusto e uma solução altamente escalável. Grandes empresas já trocaram sua API de serviços para o Play. Não acredita? O LinkedIn anunciou recentemente (ver seção **Links**) que toda sua API agora é baseada neste framework. Se o LinkedIn confia e usa, por que não você, ou sua empresa, também?

Ao desenvolver com o Play, você pode escolher entre programar com Java ou Scala, contudo, neste artigo, focaremos na vertente do Java, é claro! Caso você esteja chegando agora ao mundo do Play, notará uma diferença substancial: ele não usa a especificação do Servlet. Na verdade, ele é o que chamamos de um *full stack framework*. Em poucas palavras, o Play é a plataforma em si. Não é apenas uma biblioteca que você anexa ao seu projeto web e depois faz o deploy em um container como o Tomcat. Com ele, você desenvolve seu projeto e também faz o deploy deste projeto no próprio Play. Ele permite isto, pois tem seu próprio servidor web, que é baseado no JBoss Netty (veja a seção **Links** para mais informações).

Caso você já tenha usado o Play na versão 1.X, deve lembrar que era possível criar um arquivo WAR do seu projeto para publicar em um servidor como o Tomcat.

Resumo DevMan

Porque este artigo é útil:

Neste artigo veremos como construir um serviço RESTful com o Play Framework. O Play é uma ferramenta para construção de sistemas web que torna muito fácil criar e disponibilizar seus projetos. Embora ele nos permita implementar sistemas completos para a web, teremos como foco as principais características que nos permitem criar serviços RESTful com JSON. Deste modo, analisaremos uma maneira diferenciada de criar esse tipo de web service que já se tornou um padrão de desenvolvimento.

No entanto, a partir da versão 2.X o suporte oficial para realizar esta tarefa foi descontinuado, lhe deixando essa opção apenas através de plugins de terceiros (ver seção **Links**).

Neste artigo, vamos entender um pouco da arquitetura deste framework através de um aplicativo de exemplo baseado na ideia de listas de tarefas a fazer, os conhecidos TODOs. Nossa exemplo será, entretanto, baseado apenas em um serviço RESTful, sem uma interface com o usuário através de páginas HTML.

Sem Servlet? O que é Netty?

É provável que algumas pessoas torçam o nariz ao saber disto. Certamente dirão algo como “Servlet já é um padrão estabelecido, bem testado, etc. e etc.”. Contudo, será que Servlet é a única boa opção? É insubstituível?

Isto ocorreu não porque os desenvolvedores acharam a especificação Servlet ruim. Ela simplesmente não atendia ao que eles pretendiam para o framework. Então, porque usar o Netty e não a dupla Tomcat/Servlet, por exemplo?

O JBoss Netty não é, em sua essência, simplesmente um servidor HTTP. Ele é um framework para a construção de servidores e clientes para um determinado protocolo. Imagine que você acabou de inventar um protocolo melhor que o HTTP e agora quer construir um servidor igual a como é o Tomcat. Quais opções você teria? Primeiro, começar tudo do zero, codificando todo o seu servidor. Segundo, usar o Netty como base inicial, já que ele fornece uma boa parte do código que você precisaria criar.

O Netty não é novo, já existe desde 2004 e foi criado com uma ideia em mente: muitas vezes usamos um servidor ou protocolo para resolver problemas para os quais eles não foram destinados inicialmente. E isto pode significar, muitas vezes, uma perda de desempenho. Embora na maioria das vezes isto não seja um problema com o qual você precise se preocupar, há situações onde se faz necessário ter um servidor altamente otimizado para que ele tenha um desempenho satisfatório em ambientes de alta carga.

Mas este não é o único caso. Por exemplo, é possível que você esteja criando um *framework* próprio e queira ter um servidor feito sob medida para ele, que se encaixe perfeitamente, tirando o máximo de proveito. E foi exatamente isto que a equipe do Play Framework fez.

Neste contexto, o Netty provê um framework para você criar servidores sob medida, conforme suas necessidades específicas. Ele simplifica muito a criação de servidores que têm como base o TCP e/ou UDP, é altamente escalável e já possui suporte nativo para o protocolo HTTP.

Montando o Ambiente

Para ter seu ambiente com o Play pronto para uso, você precisa, inicialmente, baixar o pacote de instalação no site oficial. Neste artigo usaremos a versão 2.1.1, que se encontra disponível em formato ZIP. Após baixá-lo, descompacte este arquivo em qualquer pasta de sua preferência. Ao fazer isto, você terá uma estrutura de pastas igual à da **Figura 1**.

Observe também que após a descompactação teremos dois arquivos, chamados *play* e *play.bat*. Eles existem, pois o Play pode ser usado apenas através da linha de comando. Ou seja, você cria um novo projeto, inicia o servidor, entre outras tarefas, tudo através de comandos digitados em um terminal. Por isto, precisamos colocar estes dois arquivos no seu path, o que significa que será possível executá-los a partir de qualquer diretório. Para aqueles que usam sistemas baseados em Unix (Mac e Linux), basta executar o comando abaixo na linha de comando:

```
export PATH=$PATH:/diretorio-instalacao-play/
```

Nome	Data de Modif.	Tamanho	Tipo
CONTRIBUTING.md	02/04/2013 20:25	7 KB	TextW...ument
documentation	02/04/2013 20:25	--	Pasta
framework	02/04/2013 20:25	--	Pasta
play	02/04/2013 20:25	1 KB	Arquiv...l Unix
play.bat	02/04/2013 20:25	1 KB	Script
README.md	02/04/2013 20:25	3 KB	TextW...ument
repository	10/06/2013 10:28	--	Pasta
samples	02/04/2013 20:25	--	Pasta

Figura 1. Estrutura de pastas do Play

```
marloncarvalho — bash — 96x22
MacBook-Pro-de-Marlon:~ marloncarvalho$ play
play! 2.1.1 (using Java 1.6.0_45 and Scala 2.10.0), http://www.playframework.org
This is not a play application!
Use `play new` to create a new Play application in the current directory,
or go to an existing application and launch the development console using `play`.
You can also browse the complete documentation at http://www.playframework.org.
MacBook-Pro-de-Marlon:~ marloncarvalho$
```

Figura 2. Mensagem ao executar o comando play

Para usuários do Windows é necessário usar outro comando:

```
set CLASSPATH=%CLASSPATH%;c:/diretorio-instalacao-
play/
```

Uma vez que executamos os comandos anteriores, precisamos garantir que podemos executar o Play a partir de qualquer pasta. Para isto, abra uma janela do terminal e digite *play*. Lembre-se que este procedimento depende do sistema operacional que você está usando. Caso esteja tudo bem, aparecerá uma mensagem no console igual à da **Figura 2**.

Agora estamos prontos para começar nosso projeto. Antes, no entanto, temos uma pergunta para você, que é o título da próxima seção.

Precisamos de uma IDE?

O Play Framework não conta com um plugin para o Eclipse, NetBeans ou uma

IDE própria. Mas acredite, isso não é um problema com o qual você precise se preocupar muito. Gosta de trabalhar com o Eclipse? Sem problemas. Prefere o IntelliJ IDEA? Ótimo. Você pode usar a IDE de sua preferência para editar seu código, mas é perfeitamente viável usar um editor de textos sem muitos recursos como aqueles que conhecemos em IDEs como o NetBeans ou Eclipse.

Isto porque o Play nos provê recursos que permitem executar diversas tarefas apenas na linha de comando. Certamente, uma IDE facilitará para você na digitação do código, com características como o autocompletar.

Apesar disto, vamos usar apenas um editor de texto padrão e que terá como único recurso “avanhado” colorir a sintaxe do nosso código. Utilizaremos muito pouco a linha de comando e não será nada traumatizante, pode acreditar!

Iniciando nosso Projeto

Uma vez que temos nosso ambiente pronto, podemos iniciar a criação de nosso projeto. Antes de tudo, no entanto, crie um diretório com um nome de sua preferência e navegue até ele usando o terminal. Em seguida, dentro desta pasta, digite:

```
play new minhastarefas
```

Ao digitar este comando, você precisará responder a duas perguntas bem simples, conforme podemos ver na **Figura 3**. A primeira pergunta refere-se ao nome que você quer para seu projeto, enquanto a segunda pergunta sobre o tipo de projeto de sua preferência: um projeto Java ou Scala? Usaremos, neste artigo, um projeto do tipo Java.

Ao terminar a execução deste comando, teremos uma nova pasta com o nome que você deu ao seu sistema e com a estrutura padrão de um projeto do Play. Este diretório, de agora em diante, será nossa área de trabalho, portanto, navegue até ele usando o comando:

```
cd minhastarefas
```

Dentro desta nova pasta, apenas digite o comando *play*. Feito isto, será apresentado um prompt que ficará aguardando seus comandos. Agora digite *run*. Este comando vai compilar nosso projeto e iniciar o servidor Netty para que possamos acessar o projeto pelo navegador, através do endereço <http://localhost:9000/minhastarefas>. Como vemos na **Figura 4**, este endereço

```
javamagazine — bash — 96x26
bash                                bash
MacBook-Pro-de-Marlon:javamagazine marloncarvalho$ play new minhastarefas
[...]
play! 2.1.1 (using Java 1.6.0_45 and Scala 2.10.0), http://www.playframework.org
The new application will be created in /Users/marloncarvalho/javamagazine/minhastarefas
What is the application name? [minhastarefas]
>

Which template do you want to use for this new application?
1      - Create a simple Scala application
2      - Create a simple Java application
> 2
OK, application minhastarefas is created.
Have fun!
MacBook-Pro-de-Marlon:javamagazine marloncarvalho$
```

Figura 3. Criando nosso projeto.

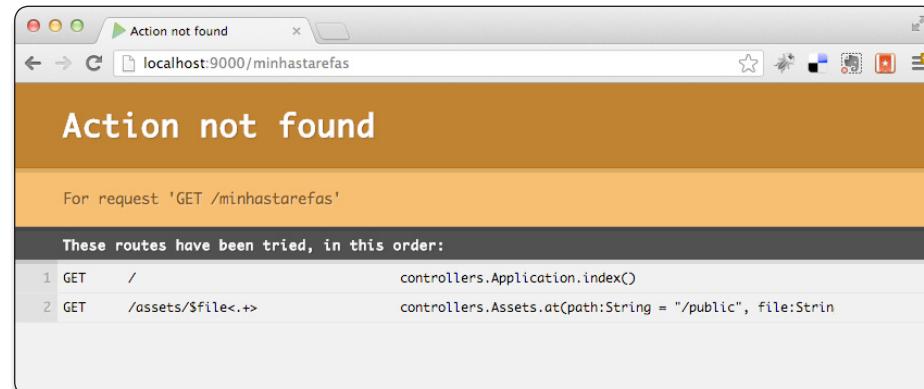


Figura 4. Tela de erro após executar o nosso projeto pela primeira ve

nos leva a uma página de erro. Isto acontece porque ainda não definimos as rotas e as classes de Controller. Faremos isto nas próximas seções!

Entendendo a estrutura de um Projeto Play!

Ao navegar por esta nova pasta, certamente você estranhou a estrutura de diretórios. É bem diferente daquilo que você está acostumado em projetos Java. No entanto, fique tranquilo, pois não tem mistérios.

A pasta *app* contém todo nosso código Java e também os templates das páginas. Nossos templates de páginas ficam localizados apenas na pasta */app/views/*, enquanto o código Java pode ficar organizado em subpastas dentro de *app*. Por exemplo, aqui já temos a pasta */app/controllers*, responsável por manter as classes que têm a função de ser Controllers do nosso aplicativo. Para este artigo, focaremos apenas na pasta de código Java, pois não faremos telas para visualização no navegador. Logo em seguida, temos a pasta *conf*, responsável por manter os arquivos de configuração. Aqui temos dois arquivos muito importantes: *application.conf* e *routes*.

O primeiro é importante para configurarmos informações como qual o banco de dados que usaremos, o Locale padrão e dados sobre o Log. O segundo será ainda mais utilizado, pois ele é responsável por manter as rotas. Como assim? Lembra-se da mensagem de erro na **Figura 4**? Essa mensagem apareceu porque nós não temos uma rota definida que informa o que deve ser feito quando o usuário acessa a tela inicial do nosso aplicativo (aqui chamado de *index*). Logo isto ficará mais claro!

Nota

Não conseguimos lhe convencer de que um editor de texto e a linha de comando são bons? Tudo bem, então, você ainda pode usar o Eclipse. Após você ter criado o projeto, conforme a seção Iniciando nosso Projeto, você deve digitar o comando "play eclipsify" dentro da pasta deste projeto. Ele será adaptado para ser importado dentro do Eclipse. Agora, abra a IDE e peça para importar um projeto em File > Import > Existing Projects into Workspace. Por fim, navegue até a pasta do projeto e o selecione.

A pasta *logs* é autoexplicativa, certo? São os logs gerados por nosso aplicativo. Na pasta *public* você colocará todos os recursos do nosso projeto. Recursos são imagens, JavaScript e stylesheets (CSS). Ainda temos a pasta *project*, que contém os arquivos do sbt (veja o tópico sbt para mais detalhes). Por fim, temos a pasta *target*, muito conhecida por quem usa o Maven. Ela está aí apenas para guardar o resultado da compilação do seu projeto.

Sbt

Sbt é a sigla para *Simple Build Tool* e trata-se de mais um gerenciador de builds. Você deve conhecer, e provavelmente ter usado, outros gerenciadores de build, como o Ant, Gradle e o Maven. Contudo, não vamos nos focar aqui em demonstrar as diferenças, ou até mesmo fazer um comparativo entre estas ferramentas. Cabe apenas destacar alguns detalhes importantes sobre o sbt.

Caso seu projeto tenha dependências para bibliotecas externas, o sbt lhe ajuda a encontrar estas dependências e resolver possíveis conflitos entre suas versões. É provável que você já conheça essa funcionalidade do Maven.

Essas dependências são encontradas através de um projeto da Apache chamado Ivy, uma ferramenta específica para gerenciar dependências para bibliotecas externas. Mas, o mais interessante de tudo para nós é que o sbt já vem configurado para usar o repositório central do Maven 2, o que facilita substancialmente nossas vidas, uma vez que esse repositório já conta com milhares de bibliotecas.

No entanto, e caso o repositório do Maven não tenha a biblioteca que procuramos? Sem problemas, podemos adicionar outros *solvers* para buscar dependências em outros repositórios, mas não nos aprofundaremos nesses detalhes aqui. Para mais informações sobre dependências com o sbt, vá à seção **Links** e procure pelo título Dependências com o sbt.

Lista de Tarefas

Nosso projeto será um serviço muito simples: vamos apenas cadastrar, excluir, marcar uma tarefa como feita e listar todas as tarefas cadastradas. Embora todo bom serviço disponível na internet tenha um mecanismo que permite aos usuários se autenticar, nós não implementaremos esta funcionalidade neste artigo. Esta implementação não será feita, pois nosso foco estará nas funcionalidades do Play que nos permitem disponibilizar serviços mais básicos, ainda sem mecanismos complexos que sistemas em produção exigem.

Vamos começar nosso projeto criando uma classe que representa uma tarefa e colocá-la dentro da pasta */app/models*. O conteúdo desta classe pode ser visto na **Listagem 1**.

Observe que nossa classe **Tarefa** terá apenas quatro atributos e todos serão públicos. Teremos um título, uma descrição e um booleano para indicar se a tarefa já foi feita. Por último, teremos um identificador que identifica unicamente a tarefa.

Com esta classe em mãos, agora precisamos definir uma classe que funcionará como Controlador. Mas, o que é um Controlador? Lembra-se dos Managed Beans do JSF? A ideia é bem parecida,

pois você precisa de uma classe que tratará uma requisição do usuário através do navegador. Imagine que ele digita o endereço de sua página e você quer processar algumas informações e depois exibir essas informações para o usuário. É aqui que entra o Controlador: tratar esta requisição e dar uma resposta apropriada.

No Play, nós definimos um controlador simplesmente criando uma classe que estende de **play.mvc.Controller** e colocando ela, preferencialmente, na pasta */app/controllers/*. Para nosso projeto, esta classe se chamará **Servico** e ficará na pasta */app/controllers*. O código inicial dela está na **Listagem 2**.

Listagem 1. Código da classe Tarefa.

```
package models;

public class Tarefa {
    public Long id;
    public String titulo;
    public String descricao;
    public boolean completada;
}
```

Listagem 2. Código da classe Servico.

```
package controllers;

import play.*;
import play.mvc.*;
import models.*;

public class Servico extends Controller {

    public static Result adicionar() {
        return null;
    }

    public static Result excluir(Long id) {
        return null;
    }

    public static Result completar(Long id) {
        return null;
    }

    public static Result listar() {
        return null;
    }
}
```

Observando esta listagem, vemos que esta classe contém apenas métodos estáticos e que retornam um objeto do tipo **play.mvc.Result**. Cada método deste é responsável por tratar uma funcionalidade específica de nosso serviço, conforme definimos nas seções anteriores. Assim, temos um método para criar uma tarefa, excluí-la, definir esta tarefa como finalizada e listar todas as tarefas cadastradas. Voltaremos a esta classe mais à frente para codificar cada método.

Definindo Rotas

Agora que temos nossa classe de controle, podemos criar rotas através do arquivo */conf/routes*. Ao abrir pela primeira vez este

Desenvolva web services com o Play Framework

arquivo, você verá que já existem duas rotas definidas. Vamos fazer uma pequena mudança e ver o resultado? Onde tem a linha `GET / controllers.Application.index()`, mude para `GET /minhastarefas controllers.Application.index()`.

Feito isso, novamente no navegador, peça para recarregar a página. Notou a diferença? Você deve estar vendo algo semelhante à **Figura 5**. Para entender, abra a classe `/app/controllers/Application.java`. Observe que ela tem apenas um método, chamado `index()` e que retorna `ok(index.render("Your new application is ready!"))`.

O método `ok()` retorna um objeto do tipo `Result` e que contém uma resposta HTTP

Nota

Observou que qualquer mudança que você faz nos arquivos de configuração e também nas classes tem efeito imediato no navegador? Sim, com o Play você não precisa aguardar 30, 40, 60 segundos para ver o resultado de suas mudanças. Para aqueles seus amigos que adoravam falar mal de Java com relação a ambientes como o PHP, onde o resultado era visto imediatamente, você já pode dar o troco!

com código 200, que significa um “OK, carregamos a página sem problemas”. Já o objeto `index` foi criado automaticamente para você pelo mecanismo de *templates* do Play, devido ao fato de termos um arquivo chamado `index.scala.html` dentro da pasta `/app/views`.

Apenas por existir esse arquivo, um objeto chamado `index` será criado automaticamente para representar, em código Java, este template. Os templates do Play são muito parecidos aos muitos mecanismos de template que existem em Java: seu conteúdo é uma mistura de código HTML com marcação específica da ferramenta. Neste template, podemos definir parâmetros que serão usados para preencher valores na tela. Por exemplo, podemos ter um parâmetro que será usado para definir o título da página.

Ao chamar o método `render()` de `index`, estamos pedindo que ele *desenhe* este template na tela do navegador. Este template pode definir quantos parâmetros quiser e passamos os valores para esses parâmetros através do método `render()`.

Não está no foco deste artigo explicar minuciosamente o mecanismo de templates do Play, uma vez que não o utilizaremos. Para mais informações sobre este poderoso mecanismo, sugerimos que vá até a seção **Links** e veja o endereço com título *Templates com Play*.

Agora precisamos definir como serão as nossas URLs e quais os métodos de nosso controlador, chamado **Servico**, serão chamados quando essas URLs forem acessadas através do navegador. Em poucas palavras: precisamos definir nossas rotas! Observe a **Listagem 3**.

Para o nosso serviço, vamos criar quatro rotas. Para fazer uma rota, primeiro definimos o tipo de método HTTP que vamos usar. Lembra que temos quatro tipos bem conhecidos? São eles: PUT, GET, DELETE e POST. Existem outros, mas nos importam, neste momento, somente estes. É importante destacar que cada método deste tem uma semântica e forma de trabalhar diferentes. Por exemplo, normalmente, usamos o GET quando queremos apenas obter informações. Já o método PUT é interessante para ser usado quando queremos adicionar, ou criar, algo, enquanto o método DELETE, por sua vez, é para excluir, e POST, para enviar dados para serem processados.

Depois de definir o método HTTP, precisamos informar como será formada nossa URL e, por último, é necessário informar quem será o responsável por tratar a requisição que será enviada no momento em que o usuário acessar essa URL através do navegador. No Play, o responsável por tratar isso é um método de um controlador específico.

Por exemplo, observe que para o usuário ter a lista de tarefas, criamos uma rota com o método GET e com a URL `/minhastarefas/listar`. Neste caso, quando o usuário digitar o endereço `http://seudominio.com/minhastarefas/listar`, o método `listar()` da classe **Servico** será chamado. Resta-nos, então, programar este método para que ele retorne uma lista de Tarefas em Json.

Antes, precisamos verificar mais alguns poucos detalhes: são os métodos para excluir e definir uma tarefa como já realizada. Observe a URL que criamos para a

Listagem 3. Rotas em `/conf/routes`.

GET	<code>/minhastarefas</code>	<code>controllers.Application.index()</code>
DELETE	<code>/minhastarefas/excluir/:id</code>	<code>controllers.Servico.excluir(id: Long)</code>
POST	<code>/minhastarefas/completar/:id</code>	<code>controllers.Servico.completar(id: Long)</code>
GET	<code>/minhastarefas/listar</code>	<code>controllers.Servico.listar()</code>
PUT	<code>/minhastarefas/adicionar</code>	<code>controllers.Servico.adicionar()</code>

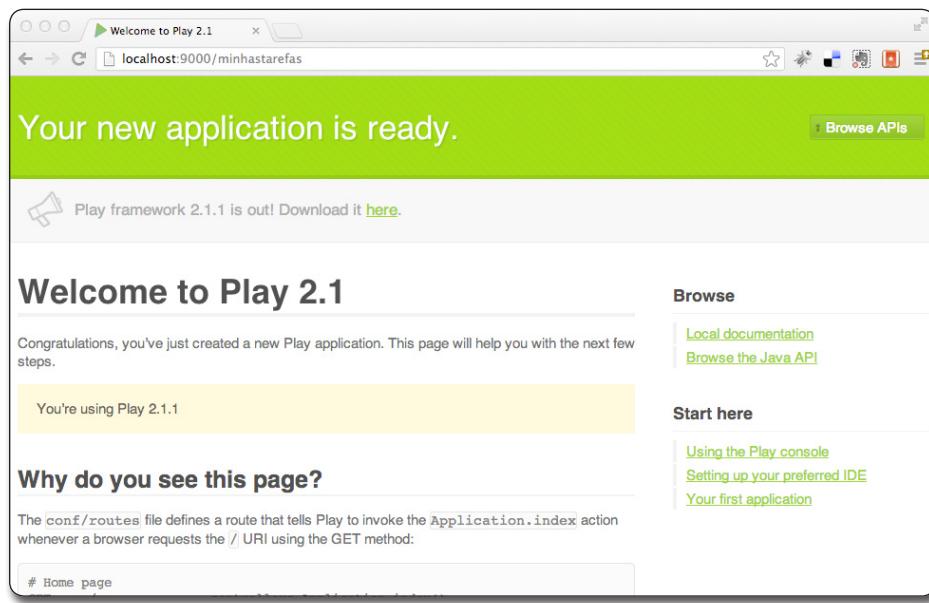


Figura 5. Tela de boas vindas do Play!

exclusão: `/minhasTarefas/excluir/:id`. Você notou o valor `:id` logo no final? Estamos definindo um parâmetro que será passado através da URL para o nosso controlador. Como assim? Observe a URL `http://localhost:9000/excluir/1`. O valor “1”, logo no final, será passado para o nosso controlador, conforme discutiremos a seguir.

Ainda para excluir uma tarefa, repare como informamos qual método será chamado: `controllers.Servico.excluir(id: String)`. Estamos dizendo que o parâmetro `id` da URL será repassado para o método `excluir()` do nosso Controller chamado **Servico** e que ele será do tipo **String**.

Por último, temos o método para adicionar uma tarefa. Usamos aqui o método PUT do HTTP e definimos a URL como `/minhasTarefas/adicionar`.

Codificando o serviço

Já criamos a nossa classe **Serviço**, mas ainda não a codificamos completamente. Vamos começar a fazer isso definindo o método que adiciona uma tarefa, conforme exibido na **Listagem 4**. Temos duas peculiaridades para analisar neste momento. Inicialmente, precisamos saber como obtemos os valores passados pelo usuário ao chamar o serviço. O outro é como retornamos uma resposta em formato Json para que o usuário tenha a confirmação de que sua tarefa foi adicionada sem problemas.

Listagem 4. Código do método adicionar() da classe Servico.

```
public static Result adicionar() {  
  
    ObjectNode response = Json.newObject();  
  
    Tarefa tarefa = new Tarefa();  
    tarefa.id = (long)tarefas.values().size();  
    tarefa.descricao = request().body().asMultipartFormData().asFormUrlEncoded().  
        get("descricao")[0];  
    tarefa.titulo = request().body().asMultipartFormData().asFormUrlEncoded().  
        get("titulo")[0];  
    tarefa.completada = false;  
    tarefas.put(tarefa.id, tarefa);  
  
    response.put("id", tarefa.id);  
    response.put("resposta", "Tarefa Adicionada com Sucesso!");  
  
    return ok(response);  
}
```

Vamos começar esse método obtendo uma referência a um **java.util.Map** que contém os valores passados pelo usuário. Primeiro, chamamos o método estático `request()`. Ele retorna um objeto do tipo **play.mvc.Http.Request** contendo os dados da requisição atual. Depois, chamamos o método `body()`, que retorna o corpo da requisição através de um objeto do tipo **play.mvc.Http.Body**.

CURSOS ONLINE

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



CONHEÇA ALGUNS DOS CURSOS:

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

Para mais informações :

www.devmedia.com.br/cursos/delphi
(21) 3382-5038

Este corpo da requisição contém os dados que foram enviados para o nosso serviço e agora precisamos extrair estes dados dele. Para simplificar o entendimento de todo este processo, podemos fazer um paralelo com a submissão de formulários em HTML.

Lembra que quando criamos formulários em HTML, definimos o formato que será usado para empacotar os dados que serão enviados para o servidor através do método POST? Informamos isto na tag `<form>` através do atributo `enctype`. Neste atributo, podemos definir dois valores possíveis: `application/x-www-form-urlencoded` e `multipart/form-data`. O primeiro é o método padrão e é usado para enviar dados em formato texto compatíveis com a tabela ASCII. Ele só tem um problema: não dá para enviar dados binários, arquivos ou textos que contenham caracteres que não sejam ASCII. Então, como resolver isso? Usando o `multipart/form-data`.

Voltando para o nosso serviço, é importante entender que quando enviamos dados para ele, usando o método POST, precisamos definir qual dos dois formatos que discutimos será utilizado. Para o nosso serviço, definiremos que estes dados sempre virão através de `multipart/form-data`.

Já podemos, então, partir para o código necessário para obter o conteúdo da requisição. Para fazer isto, precisamos chamar o método `asMultipartFormData()` e, logo em seguida, o método `asFormUrlEncoded()`, que nos retornará um `Map` contendo os dados que necessitamos.

Como temos um `Map`, podemos obter os valores usando o método `get()` e passando como parâmetro para ele uma `String` com a chave associada ao dado. Mas, por que cada chave do mapa retorna um `array`? Bem simples! Lembre que quando você cria formulários em HTML, mais de um campo dele pode ter o mesmo nome. Não é uma boa prática, mas nada lhe impede que você faça isso. Por exemplo, você pode colocar dois campos `<input>` com o nome igual a CPF e quando este formulário for submetido, chegará ao servidor como uma variável chamada CPF e com um array com dois valores!

Como o método é para adicionar uma Tarefa, necessitamos criar um objeto que representa esta tarefa. Nós já fizemos isso nas seções anteriores, lembra? Colocamos essa classe na pasta `/app/models/`. Precisamos, então, criar um objeto do tipo `models.Tarefa` e definir os valores dos atributos dele com os dados enviados na requisição e que temos no `Map`.

Uma vez que criamos a tarefa com os valores da requisição, temos que guardar essa tarefa em algum lugar. Poderia ser um banco de dados como o MySQL ou PostgreSQL, mas aqui usamos simplesmente um `HashMap` onde a chave será o id da tarefa e o valor dessa chave será o objeto que representa a tarefa. Ainda precisamos de mais um passo: dar uma resposta para o usuário informando que a operação foi um sucesso. Fazemos isso usando uma biblioteca para Json que já vem no Play: Jackson.

Antes de começarmos a trabalhar com o Jackson, você lembra o que é e o formato de um Json? Trata-se de um padrão para tráfego de informações que surgiu na programação com JavaScript. É um acrônimo para JavaScript Object Notation. O Json traduzia

os atributos de um objeto JavaScript para uma representação textual. Contudo, ele não se restringe apenas à programação com essa linguagem. Aliás, ele é hoje usado praticamente por todos os serviços RESTful que encontramos! Vamos analisar o seguinte código:

```
{"revista":"Javamagazine", "artigo":{"titulo":"PlayFramework", "autor":"Marlon"}}
```

Neste exemplo, temos uma informação representada na notação do Json. Observe que toda informação em Json vem sempre circunscrita por chaves. Conforme explicamos, o Json descreve os atributos de um objeto, então, o código acima se refere a um objeto que contém dois atributos. O primeiro atributo se chama `revista` e tem o valor Javamagazine. O segundo atributo se chama `artigo` e é uma referência para um segundo objeto, que tem outros dois atributos: `titulo` e `autor`.

Com esta breve explicação, vamos entender como criar um Json em código Java usando a biblioteca Jackson. Nesta biblioteca, um objeto em Json é representado pela classe `ObjectNode`. Para facilitar seu entendimento, vamos analisar a **Listagem 5**, que contém uma estrutura de `ObjectNodes` que representa o trecho de código que exibimos anteriormente. Observe que neste trecho criamos duas instâncias de `ObjectNode`, uma para o objeto que contém os atributos `revista` e `artigo` e outra para representar o objeto que está contido no atributo `artigo` e que contém os atributos `titulo` e `autor`.

Listagem 5. Exemplo de Criação de objetos do tipo `ObjectNode`.

```
ObjectNode objeto1 = Json.newObject();
ObjectNode objeto2 = Json.newObject();

objeto2.put("titulo", "PlayFramework");
objeto2.put("autor", "Marlon");

objeto1.put("revista", "Javamagazine");
objeto1.put("artigo", objeto2);
```

O uso do `ObjectNode` é simples, pois ele contém um método que nos permite criar os atributos: `put(String, String)`. Podemos, então, usar o código `put("revista", "Javamagazine")`, por exemplo, para criar o primeiro atributo. Faremos o mesmo para o segundo objeto, só que agora usando `put("titulo", "PlayFramework")` e `put("autor", "Marlon")`. Finalizando, temos `put("artigo", objeto2)` para criar o atributo `artigo`.

Observe que na **Listagem 4** fizemos um uso similar ao que descrevemos nos parágrafos anteriores. Criamos um objeto do tipo `ObjectNode`, criamos dois atributos (`id` e `resposta`) e definimos seus respectivos valores (identificador da tarefa e “Tarefa Adicionada com Sucesso”!). Para que este objeto possa servir como resposta para o nosso método de adicionar, temos apenas que chamar o método `ok()` passando ele como parâmetro. O Play faz todo o resto para nós! Ele pega esse JSON e transforma em uma resposta automaticamente.

A questão agora é: como testar nosso serviço para verificar que ele realmente adicionou a tarefa? Caso você apenas digite na barra de endereços do seu navegador o endereço <http://localhost:9000/minhastarefas/adicionar>, terá uma resposta dizendo que a ação não existe (Action Not Found). Isto acontece porque quando você faz isso, está usando o método GET do HTTP. Lembra que para a adição usamos PUT?

Então, para facilitar o teste de nosso serviço, vamos usar uma extensão muito interessante para o Google Chrome chamada Postman REST Client. Ela facilita substancialmente esta tarefa. Para instalar ele, vá até a Chrome Store e procure pelo nome da ferramenta. O uso da extensão é bastante simples e não vamos entrar em detalhes sobre ela. Caso tenha dificuldades com a extensão, preencha os dados da tela conforme a **Figura 6**.

Observe como ficou a resposta do serviço. É um JSON com as chaves e dados que definimos logo acima.

Vamos partir agora para o método que vai listar todas as tarefas cadastradas. O que precisamos fazer, basicamente, é navegar no nosso Map que contém as tarefas adicionadas pelo método **adicionar()** e criar objetos JSON a partir dos dados de cada tarefa. O código deste método encontra-se na **Listagem 6** e faz exatamente o que descrevemos aqui.

Observe que criamos um objeto do tipo **ArrayNode** que conterá uma lista de objetos **ObjectNode** contendo os dados de cada tarefa. Enfim, chamamos **ok()** com esse **ArrayNode** como parâmetro. O Play cuida, mais uma vez, do resto para nós. Veja a resposta para essa chamada na **Figura 7**. Aqui já podemos chamar nosso serviço diretamente da janela do navegador com o endereço <http://localhost:9000/minhastarefas/listar>, sem necessidade da extensão.

Já que temos tarefas, também precisamos completá-las, isto é, definir que elas já foram feitas. Por isso temos o método **completar()** recebendo um único parâmetro: o identificador da tarefa. Este método é bastante simples e apenas busca a tarefa no **Map** e muda o valor do atrí-

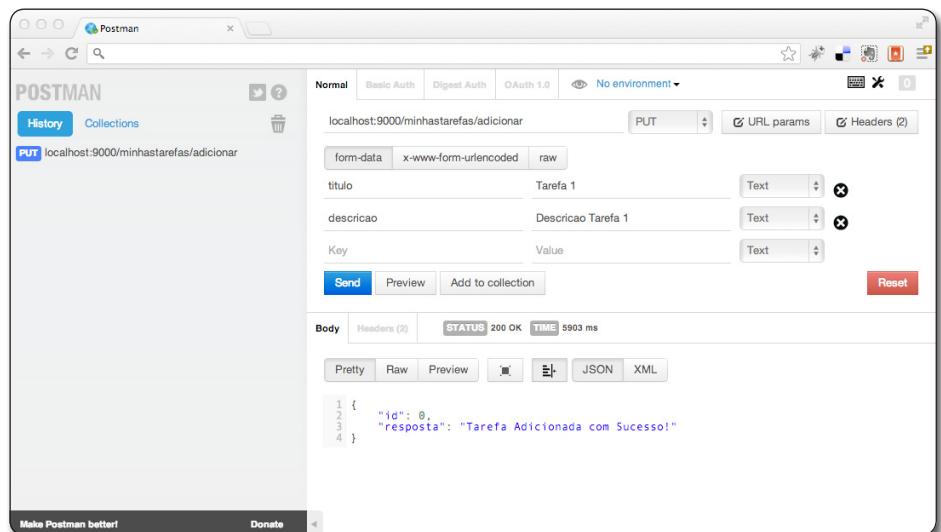


Figura 6. Método **adicionar()** testado usando o Postman!



Figura 7. Resposta do serviço para o método **listar()**

buto **completada** para **true**. Depois, montamos uma resposta em JSON informando ao usuário se a tarefa foi completada ou não. Veja este código na **Listagem 7**.

Finalizando nosso serviço, também temos que implementar o método para excluir uma tarefa. Observe que o código dele, apresentado na **Listagem 8**, é bem parecido com o código do método para completar a tarefa. A diferença é

Listagem 6. Código do método **listar()** da classe **Serviço**.

```
public static Result listar() {
    ObjectNode node = Json.newObject();
    ArrayNode response = node.arrayNode();
    for(Tarefa tarefa : tarefas.values()) {
        node = Json.newObject();
        node.put("id", tarefa.id);
        node.put("titulo", tarefa.titulo);
        node.put("descricao", tarefa.descricao);
        node.put("completada", tarefa.completada);
        response.add(node);
    }
    return ok(response);
}
```

Desenvolva web services com o Play Framework

que agora precisamos excluir a tarefa do Map, usando o seu método **remove()**.

Enfim, temos nosso serviço pronto e funcionando!

Listagem 7. Código do método completar() da classe Serviço.

```
public static Result completar(Long id) {
    ObjectNode response = Json.newObject();

    if(tarefas.containsKey(id)) {
        Tarefa tarefa = tarefas.get(id);
        tarefa.completada = true;
        response.put("resposta", "Tarefa completada com sucesso!");
    } else {
        response.put("resposta", "Tarefa não encontrada.");
    }
    return ok(response);
}
```

Listagem 8. Código do método excluir() da classe Serviço.

```
public static Result excluir(Long id) {

    ObjectNode response = Json.newObject();

    if(tarefas.containsKey(id)) {
        tarefas.remove(id);
        response.put("resposta", "Tarefa excluída com sucesso!");
    } else {
        response.put("resposta", "Tarefa não encontrada.");
    }
    return ok(response);
}
```

Conclusão

Neste artigo, focamos em apresentar como criar um serviço RESTful com o Play Framework de maneira fácil e sem muitos mistérios. Também descrevemos alguns detalhes desta ferramenta, embora não tenhamos feito uma análise muito profunda de todos os seus mecanismos.

O Play Framework vem despertando bastante interesse, sendo, inclusive, a escolha do LinkedIn para “motorizar” sua API on-line. Ele fornece uma forma diferente de construir aplicativos para a web. Além disso, o Play não utiliza a especificação Servlet e tem um próprio servidor HTTP baseado no Netty. Este servidor é rápido e simples de usar, não necessitando de uma IDE, pois tudo pode ser realizado apenas na linha de comando.

Com o Play Framework, você não precisa mais esperar tanto para ver suas mudanças funcionando, basta salvar seu arquivo, seja ele de código ou configuração, e em um piscar de olhos elas já estarão disponíveis para visualização. Também esqueça o Tomcat, pois um projeto Play não usa esse servidor. Ele é, por si só, a solução completa. Não apenas uma biblioteca para ser incluída no seu projeto, mas um framework *full stack*.

Autor



Marlon Silva Carvalho

marlon.carvalho@gmail.com

É desenvolvedor de longa data. Tem a programação poliglota como hobby predileto, a música como motivadora, a família como referência e trabalha com desenvolvimento mobile no Serpro. Seus olhos estão atualmente focados no Play Framework, Android e tudo o que envolve mobilidade. É organizador do Grupo de Desenvolvedores Google (GDG) de Salvador. Para conhecê-lo um pouco mais, acesse seu blog em marlon.silvacarvalho.net ou siga-o pelo seu twitter @marlonscarvalho.



Links:

Anúncio do LinkedIn sobre o uso do Play Framework.

<http://engineering.linkedin.com/play/play-framework-linkedin>

Sítio oficial do Play Framework.

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

Sítio oficial do JBoss Netty.

<http://www.netty.io/>

Mudando variáveis de ambiente no Windows.

http://www.java.com/pt_BR/download/help/path.xml

Dependências com o sbt.

<http://www.playframework.com/documentation/2.0/SBTDependencies>

Templates com Play.

<http://www.playframework.com/documentation/2.1.1/JavaTemplates>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



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

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

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

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



Estrutura

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



Suporte diferenciado

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



Serviços

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



1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
[Conheça!](#)



Porta 80

WEB HOSTING

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

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

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



DESDE 1996, FORMANDO PROGRAMADORES DE EXCELÊNCIA.

FAÇA AS **FORMAÇÕES JAVA** NO INSTITUTO PIONEIRO
NO ENSINO DA LINGUAGEM JAVA NO BRASIL.

FORMAÇÃO

DESENVOLVEDOR JAVA: SISTEMAS DISTRIBUÍDOS

Aprofunde-se em uma das linguagens mais valorizadas pelas empresas e saiba como desenvolver uma complexa aplicação distribuída nas plataformas JAVA ME e JAVA EE (com Servlets, JSP, Componentes EJB, Hibernate e WebServices).

Para mais informações:

www.infnet.edu.br/javadistribuidos

FORMAÇÃO

DESENVOLVEDOR JAVA:

Domine o desenvolvimento de aplicações orientadas a objetos e aprenda tópicos avançados de programação como o uso de Servlets, JSP, Struts e Hibernate dentro do padrão MVC.

Para mais informações:

www.infnet.edu.br/java