



Edição 136 :: R\$ 14,90

 DEVMEDIA

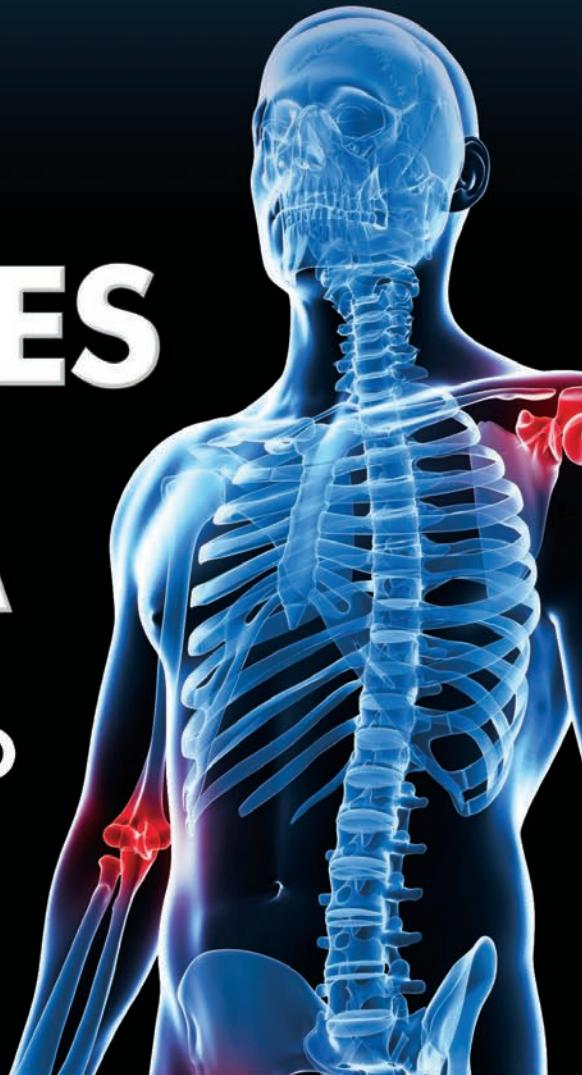
ESPECIAL:
Mão na massa com Spring 4
Construindo uma aplicação web com
a nova versão do framework

Desenvolva sistemas web com JSF e HTML5
Empregando recursos do HTML5 em aplicativos JSF

Testes de software com SoapUI e JMeter
Compartilhe responsabilidades
entre desenvolvimento e teste

EXPRESSÕES LAMBDA

Um novo recurso do Java 8



Tarefas assíncronas com EJBs
Aprenda a utilizar os recursos
assíncronos da Java EE

DevOps na prática
Criando ambientes virtuais para
testar suas aplicações

OptaPlanner
Aprenda a escolher o algoritmo
de otimização mais eficiente

MVP

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

APlique esse investimento
na sua carreira...

E mostre ao mercado
quanto você vale!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

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



POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

 **DEVMEDIA**



Edição 136 • 2015 • ISSN 1676-8361



MVP

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

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico 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 à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

CURSOS ONLINE

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



CONHEÇA ALGUNS DOS CURSOS:

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

Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Sumário

Artigo no estilo Solução Completa, Artigo no estilo Curso

06 – Spring Framework: Criando uma aplicação web – Parte 1

[Francisco A. Garcia]

Conteúdo sobre Novidades

18 – Como aproveitar o máximo das Expressões Lambda em Java

[Marlon Silva Carvalho]

Artigo no estilo Solução Completa

28 – Como escolher o algoritmo mais eficiente com OptaPlanner – Parte 2

[Marcelo A. Cenerino]

Conteúdo sobre Novidades

38 – Crie sistemas web com JSF e HTML5

[Luiz Henrique Zambom Santana e Eduardo Felipe Zambom Santana]

Artigo no estilo Engenharia de Software

46 – Apache JMeter: Testes de software com SoapUI

[Renata Eliza e Fabiana Alencar]

Artigo do tipo Mentoring

55 – Como trabalhar com tarefas assíncronas em Java EE

[Bruno F.M. Attorre]

Conteúdo sobre Novidades

64 – DevOps: Como criar ambientes virtuais para testes

[Leonardo Comelli]



Dê seu feedback sobre esta edição!

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

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

www.devmedia.com.br/javamagazine/feedback



CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

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

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

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

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

Na suite ToolsCloud você tem a sua disposição:

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

Acompanhe a Toolscloud:



toolscloud@toolscloud.com



twitter.com/toolscloud



Spring Framework: Criando uma aplicação web – Parte 1

Construindo uma aplicação web com a nova versão deste articulado framework

ESTE ARTIGO FAZ PARTE DE UM CURSO

Spring Framework chega à versão 4 ultrapassando 10 anos de existência, contados a partir do seu release inicial – mais especificamente, março de 2004. Na época, esta versão inicial já era bastante inovadora, apresentando-se como um *container* leve que oferecia a criação e gerenciamento de *beans*, injeção de dependências, controle de transações (de forma declarativa e programática), suporte ao Hibernate, um *framework* MVC completo, o Spring MVC, classes utilitárias para o desenvolvimento de DAOs, uso da tecnologia JDBC e a integração com EJBs.

O conjunto das soluções providas pelo Spring Framework permite a construção de qualquer tipo de aplicação, não estando restrito apenas a soluções para a plataforma Web. Além disso, a facilidade de configuração e execução do *container* simplifica o desenvolvimento e a criação do ambiente para execução de testes integrados.

Durante estes mais de 10 anos, o Spring obteve uma evolução grandiosa, acumulando inovações em suas principais funcionalidades. Podemos citar, por exemplo, o uso de anotações para a criação, gerenciamento e injeção de *beans*, bem como o lançamento de novos módulos como o Spring Security, Spring Data, Spring Batch, Spring Integration, entre outros.

A versão 4 do Spring Framework, lançada em Dezembro de 2013, vem com o compromisso de manter a plataforma alinhada com as tecnologias mais recentes

Fique por dentro

Este artigo é útil porque apresenta alguns dos recursos disponíveis a partir da versão 4 do Spring Framework. Para isso, será demonstrado como fazer uso desses recursos no decorrer do desenvolvimento de uma aplicação web. Esse conteúdo é bastante importante principalmente por ajudar o leitor a se familiarizar com as novas soluções disponibilizadas pelo Spring, possibilitando uma análise do funcionamento destas e facilitando a identificação de quando adotá-las.

para o desenvolvimento Java. Nesta versão, além de melhorias em funcionalidades do próprio *framework*, foram adicionados suporte para as plataformas Java 8 e Java EE 7, a evolução de recursos no desenvolvimento de interfaces REST e o suporte à tecnologia de WebSockets. Dito isso, o objetivo deste artigo é apresentar de forma prática algumas destas novidades trazidas no lançamento do Spring Framework 4, por meio do desenvolvimento das funcionalidades de uma aplicação web.

Nesta primeira parte, alguns dos novos recursos serão apresentados na implementação de funcionalidades de infraestrutura e *backend* da aplicação, enquanto a segunda parte terá o foco em expor recursos utilizados na criação da camada de apresentação.

A aplicação proposta

Para realizar a demonstração dos recursos disponíveis no Spring Framework 4, foi idealizado um sistema para controle de tarefas. Sendo assim, no decorrer do desenvolvimento deste sistema, serão utilizadas as novas funcionalidades do Spring de uma maneira simples, focando sempre em facilitar o entendimento do funcionamento e o propósito (solução) oferecido por cada recurso.

O sistema de controle de tarefas

O sistema de Controle de Tarefas tem como objetivo principal facilitar o controle das atividades gerenciadas por um grupo de pessoas.

Assim, para cada pessoa com acesso ao sistema, denominada usuário, existirá uma associação com um perfil, e cada perfil concede níveis de operação diferentes dentro do sistema.

No sistema de Controle de Tarefas, foram definidos dois perfis de usuário, detalhados a seguir:

1. **Administrador:** Este perfil permite ao usuário a criação de tarefas e o acompanhamento (em tempo real) de todas as tarefas cadastradas no sistema;
2. **Usuário:** O usuário com este perfil pode assumir tarefas e consequentemente mudar o status das tarefas para as quais ele é o responsável.

Além das informações de usuário, o sistema também manterá os dados que permitem a identificação, autoria, responsabilidade e o controle dos possíveis estados de uma tarefa.

Os estados das tarefas gerenciadas pelo sistema são:

- **Cadastrada:** Este é o estado inicial de uma tarefa, atribuído na criação da tarefa por um usuário com perfil **Administrador**;
- **Iniciada:** A tarefa neste estado foi iniciada por um usuário com perfil **Usuário**, que neste caso assumiu o papel de responsável por ela;

- **Concluída:** Ao alcançar este estado, a tarefa foi concluída com sucesso pelo usuário responsável;

- **Descartada:** Este estado conclui a tarefa com insucesso, indicando que o usuário responsável optou por anular a tarefa.

A seguir estão as principais classes do domínio do sistema Controle de Tarefas, criado a partir da descrição das funcionalidades e exibido em detalhes na **Figura 1**:

- **br.com.jm.tarefas.domain.IdentificadorUsuario:** Classe que representa o identificador do usuário no sistema. Em nosso exemplo utilizamos o e-mail como identificador;
- **br.com.jm.tarefas.domain.PerfilUsuario:** Enumeração com o domínio de perfis de usuário;
- **br.com.jm.tarefas.domain.Usuario:** Classe que representa um usuário no sistema, contendo alguns atributos básicos e a associação com um identificador e perfil, além do relacionamento com as tarefas para as quais é o autor ou responsável;
- **br.com.jm.tarefas.domain.IdentificadorTarefa:** Classe que representa o identificador da tarefa no sistema. Em nosso exemplo utilizamos um código sequencial numérico;
- **br.com.jm.tarefas.domain.StatusTarefa:** Enumeração contendo o domínio de estados de uma tarefa;
- **br.com.jm.tarefas.domain.Tarefa:** Classe que representa uma tarefa no sistema. Possui algumas informações pertinentes à tarefa e associação com identificador, autor e responsável;

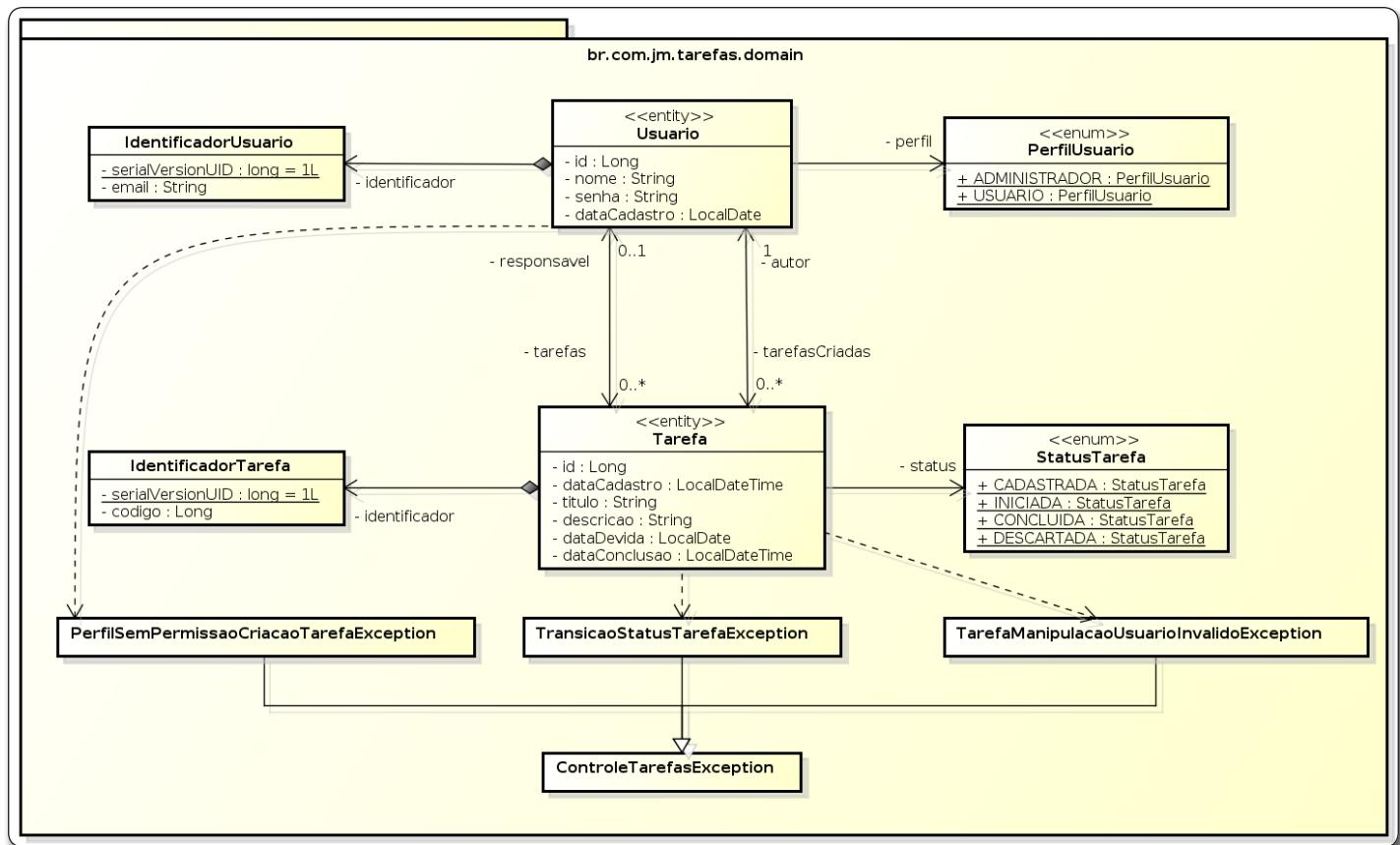


Figura 1. Modelo de classes do sistema

Spring Framework: Criando uma aplicação web – Parte 1

- **br.com.jm.tarefas.domain.ControleTarefasException:** Esta é a classe base de exceção para os erros ocorridos nas operações de gerenciamento das tarefas;
- **br.com.jm.tarefas.domain.TransicaoStatusTarefaException:** Esta classe de exceção representa o erro na tentativa de realizar a transição entre status de uma tarefa;
- **br.com.jm.tarefas.domain.TarefaManipulacaoUsuarioInvalidoException:** Exceção que representa a tentativa de manipulação de uma tarefa por um usuário que não seja o responsável por esta;
- **br.com.jm.tarefas.domain.PerfilSemPermissaoCriacaoTarefaException:** Esta exceção indica a tentativa de criação de tarefa por um usuário cujo perfil não permite esta operação.

Para construção do sistema de Controle de Tarefas será utilizada a arquitetura multicamadas, concebida com o uso da abordagem *Domain Driven Design*. As quatro camadas (*domain*, *application*, *infrastructure* e *user interface*) foram organizadas em pacotes conforme detalhado a seguir:

- **br.com.jm.tarefas.domain:** Este pacote simboliza a camada principal, que abriga as classes que representam o domínio e encapsulam as regras de negócios. Neste pacote estão:

- As classes do domínio apresentadas na **Figura 1**;
- As interfaces **TarefaRepository** e **UsuarioRepository**, contendo as operações para o acesso aos dados das entidades **Tarefa** e **Usuario**, respectivamente;
- O subpacote **dto**, contendo as classes do padrão **DTO** que representam, de forma simplificada, as informações do domínio.
- **br.com.jm.tarefas.application:** Neste pacote está representada a camada de aplicação, armazenando as classes de serviço responsáveis por receber as requisições da camada de apresentação, recuperar objetos de domínio necessários e disparar nestes objetos as ações solicitadas pelo usuário. A classe **GerenciamentoTarefasService**, presente neste pacote, processa as requisições da camada de apresentação, recuperando as informações de usuário e tarefa envolvidas na solicitação.
- **br.com.jm.tarefas.infrastrucuture:** Pacote que armazena a camada com código de infraestrutura necessário para o funcionamento da aplicação; por exemplo: o acesso aos dados persistidos, configurações, envio de e-mail, etc. Neste pacote temos os seguintes artefatos que merecem destaque:
 - A implementação da anotação **@ServiçoTransacional**, que será detalhada no

decorrer do artigo, no tópico Anotações Compostas;

- A classe **InicializadorBD** corresponde a uma classe utilitária para inicialização do banco de dados com alguns dados de exemplo, recurso útil no ambiente de desenvolvimento;

- A classe **PublicadorAtualizacaoTarefa**, que executa a publicação das informações de criação e atualização de tarefas através de um subsistema de mensagens que será detalhado no tópico WebSockets, apresentado na segunda parte deste artigo;

- O subpacote **config**, que contém, principalmente, as classes de configuração do *container* Spring, utilizando a abordagem de configuração com o uso de classes (JavaConfig);

- O subpacote **security**, que contém algumas classes criadas para a customização do uso do módulo Spring Security.

- **br.com.jm.tarefas.interfaces.web:** Este pacote é a camada de apresentação do sistema, sendo responsável por receber as solicitações do usuário e repassar estas solicitações para a camada de aplicação. A classe **GerenciamentoTarefaController**, contida neste pacote, oferece todas as operações para o gerenciamento das tarefas por uma interface REST, com a troca de informações em formato JSON. O desenvolvimento da camada de apresentação será abordado na segunda parte deste artigo.

Na **Figura 2** podemos ver, com mais detalhes, a organização dos pacotes (camadas) e também algumas das principais classes do sistema de Controle de Tarefas, já citadas anteriormente.

A interface web

A interface do sistema de Controle de Tarefas será bastante simples, contando apenas com duas telas:

- A tela de login utilizada para autenticação dos usuários;
- A tela principal home, organizada com um mecanismo de abas, sendo esta a tela de destino após a autenticação efetuada com sucesso.

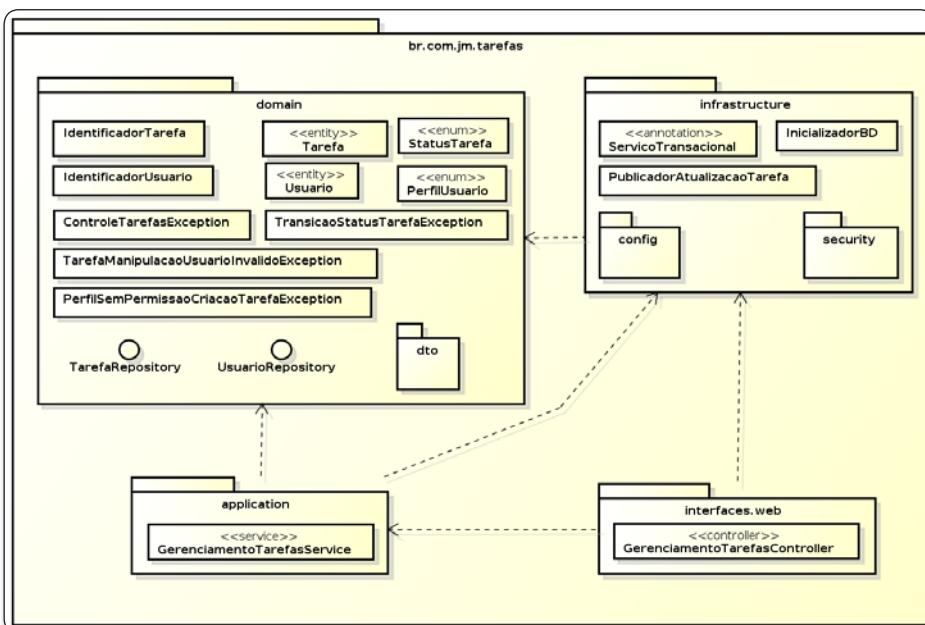


Figura 2. Organização dos pacotes (camadas)

As abas disponíveis, o propósito de cada uma e o acesso pelos perfis de usuário serão detalhados a seguir.

Autenticação dos usuários

A autenticação dos usuários no sistema será efetuada mediante e-mail e senha (previamente cadastrados), solicitados pela tela de login apresentada na **Figura 3**. Após o login efetuado com sucesso, o usuário será direcionado para a tela home, onde serão exibidas as abas com as informações de tarefas de acordo com o perfil identificado no processo de autenticação.



Figura 3. Tela de login

A aba Tarefas

Caso o usuário autenticado esteja associado com o perfil **Administrador**, será apresentada a tela home disponibilizando apenas a aba *Tarefas*, conforme exibido na **Figura 4**.

Os três pontos destacados nesta figura estão detalhados a seguir:

1. O botão de inclusão de tarefas, exibido somente para o perfil **Administrador**, aciona um *popup* modal (**Figura 5**) que permite a inclusão de uma nova tarefa;
2. O perfil **Administrador** visualiza somente a aba *Tarefas*, que lista todas as tarefas cadastradas no sistema;
3. A coluna de nome Responsável, ainda na aba *Tarefas*, indica qual é o usuário responsável pela tarefa, caso exista um.

Para facilitar o acompanhamento pelos administradores, esta aba será atualizada em tempo real, refletindo as modificações ocorridas nas tarefas cadastradas no sistema.

Figura 4. Funcionalidades da aba Tarefas

Figura 5. Pop-up para criação de uma nova Tarefa

Figura 6. Funcionalidades da aba Minhas Tarefas

Figura 7. Funcionalidades da aba Tarefas Cadastradas

A aba Minhas Tarefas

Na **Figura 6** é exibida a tela home para o usuário com perfil **Usuário**. Como pode ser verificado, para esse tipo de usuário é disponibilizada a aba *Minhas Tarefas*.

Assim como realizado na **Figura 5**, na **Figura 6** temos três pontos a detalhar, a saber:

1. A aba *Minhas Tarefas* lista todas as tarefas para as quais o responsável corresponde ao usuário logado no sistema;
2. Botão que permite a conclusão da tarefa relacionada;
3. Botão que permite descartar a tarefa relacionada.

3. Botão que permite descartar a tarefa relacionada.

A aba Tarefas Cadastradas

Além da aba *Minhas Tarefas*, o usuário com perfil **Usuário** também possui acesso à aba *Tarefas Cadastradas*, exposta na **Figura 7**. Note que temos dois pontos que requerem detalhamento:

1. A aba *Tarefas Cadastradas* lista todas as tarefas que ainda não possuem um usuário responsável, ou seja, ainda não foram iniciadas;

2. Botão que permite iniciar (assumir como responsável) a tarefa relacionada.

Definição das ferramentas e tecnologias

Para o desenvolvimento do sistema de Controle de Tarefas, foi escolhido o seguinte conjunto de ferramentas e tecnologias:

- **JDK 8:** Apesar do Spring 4 exigir como versão mínima o JDK 6, como o propósito é também demonstrar o suporte a alguns recursos do Java 8, esta será a versão do JDK utilizado no desenvolvimento;
- **Maven 3:** Para o gerenciamento das dependências, compilação, empacotamento e distribuição dos artefatos do projeto;
- **Eclipse Luna:** Versão da IDE Eclipse compatível com o JDK 8;
- **Spring Framework 4.1.0.RELEASE:** Versão atual do Spring Framework, utilizado como recurso essencial na construção do sistema por causa das suas principais funcionalidades, como: injeção de dependências, controle transacional e suporte à construção de interfaces Web/RESTful. Este é o foco principal do artigo;
- **Spring Security 4.0.0.M2:** Versão atual do módulo Spring específico para o desenvolvimento da segurança (autenticação e autorização);
- **Spring Data JPA 1.7.0.RELEASE:** Versão atual do módulo Spring para construção da camada de acesso a dados persistidos em bases de dados relacionais utilizando a tecnologia JPA;
- **JPA 2.1 e Hibernate 4.3.6 Final:** Versão atual da API de persistência JPA em conjunto com o provedor de persistência Hibernate;
- **HyperSQL (HSQLDB) 2.3.2:** Banco de dados em memória utilizado no ambiente de desenvolvimento;
- **PostgreSQL 9.1:** Banco de dados para utilização nos demais ambientes suportados pelo sistema;
- **Jetty 9.2:** Servidor Web utilizado para execução da aplicação no ambiente de desenvolvimento via plugin Maven. Já possui suporte a WebSockets (JSR 356);
- **jQuery 2:** Biblioteca JavaScript para simplificação do desenvolvimento com esta linguagem no lado cliente (navegador);
- **Knockout 3.2.0:** Biblioteca JavaScript para manipulação e visualização do modelo de dados através da aplicação do pattern MVVM (*Model-View-ViewModel*);
- **Bootstrap 3:** Utilizado para a criação do *layout* e componentes visuais;
- **stomp.js:** Implementação JavaScript de cliente STOMP;
- **SockJS:** Cliente JavaScript para emulação de WebSockets, no caso de indisponibilidade deste recurso em navegadores mais antigos.

Configuração das dependências Spring

O artefato final do projeto do sistema de Controle de Tarefas será uma aplicação web no formato WAR, gerado através do processo de construção Maven.

Como o sistema irá requerer a importação de várias dependências (bibliotecas) Spring, a definição das versões utilizadas seguirá o padrão BOM (*Bill of Materials*). Esse padrão é um re-

curso de configuração de versões de dependências do Maven que permite importar uma configuração pré-definida contendo todas as bibliotecas relacionadas e suas respectivas versões. Este recurso facilita a utilização correta do conjunto de dependências e também evita problemas com o uso de versões não compatíveis entre si.

Na **Listagem 1** está o trecho do arquivo de configuração do projeto Maven (*pom.xml*) contendo a declaração das propriedades que estabelecem a versão do Spring Framework e também as versões dos módulos Spring Security e Spring Data JPA utilizados. Além das versões, consta também a definição do artefato BOM *spring-framework-bom*; este na seção de gerenciamento de dependências (*dependencyManagement*).

Listagem 1. Trecho do arquivo pom.xml com declaração do artefato BOM Spring Framework.

```
<properties>
    <org.springframework-version>4.1.0.RELEASE
    </org.springframework-version>
    <org.springframework.security-version>4.0.0.M2
    </org.springframework.security-version>
    <org.springframework.data.jpa-version>1.7.0.RELEASE
    </org.springframework.data.jpa-version>
</properties>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-framework-bom</artifactId>
            <version>${org.springframework-version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Com a definição das versões das dependências dos módulos do Spring por meio do artefato BOM, a importação das bibliotecas deste para uso no projeto fica simplificada, eliminando a necessidade de informar a versão, uma vez que esta já foi devidamente fornecida através do artefato *spring-framework-bom*. O exemplo desta configuração pode ser verificado na **Listagem 2**.

As dependências dos módulos Spring que não estão inclusas no artefato **BOM**, como Spring Security e Spring Data JPA, devem ser declaradas separadamente, com a indicação da versão a ser importada, conforme o trecho do *pom.xml* demonstrado na **Listagem 3**.

Até este ponto apresentamos o projeto do sistema de Controle de Tarefas com uma visão geral do propósito da aplicação, do modelo de classes, da organização das camadas e pacotes, das tecnologias utilizadas e do conjunto de configurações das dependências Spring.

Na sequência, iremos demonstrar alguns dos recursos disponibilizados a partir da versão 4 do Spring Framework. Cada recurso será demonstrado através de sua adoção no desenvolvimento das funcionalidades do sistema exemplo.

Listagem 2. Declaração das dependências dos módulos inclusos no artefato BOM Spring Framework.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-messaging</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-websocket</artifactId>
</dependency>
```

Listagem 3. Declaração das dependências dos módulos não definidos no artefato BOM Spring Framework.

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>${org.springframework.security-version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>
    <version>${org.springframework.security-version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>${org.springframework.security-version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-messaging</artifactId>
    <version>${org.springframework.security-version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>${org.springframework.data.jpa-version}</version>
</dependency>
```

Anotações compostas (Meta Annotations)

A configuração de *beans* no *container* Spring pode ser realizada por meio de definições no formato XML e/ou pelo uso de algumas anotações pré-definidas. Mesmo com o uso de anotações, é necessário configurar o escaneamento dos pacotes que contêm as classes anotadas, que serão utilizadas para a criação dos *beans*.

Durante o escaneamento dos pacotes, as classes contendo as anotações detalhadas a seguir serão marcadas como candidatas para criação dos *beans* no *container* Spring. Estas anotações são definidas na forma de estereótipos, ou seja, carregam um signifi-

cado concebido para um determinado tipo de objeto, estando divididas em:

- **@org.springframework.stereotype.Service:** As classes de objetos com este estereótipo geralmente oferecem um conjunto de operações na camada de negócios. Estes objetos do tipo **Service** são utilizados pelas demais camadas da aplicação;
- **@org.springframework.stereotype.Repository:** Este estereótipo é aplicado em classes de objetos que irão encapsular e lidar com o acesso a dados. Os dados manipulados por estes objetos podem, por exemplo, estar persistidos em um banco de dados relacional. Os objetos do tipo **Repository** geralmente são utilizados pela camada de negócios, composta por objetos do estereótipo **Service**;
- **@org.springframework.stereotype.Controller:** Classes com esta anotação definem objetos que irão controlar o fluxo de execução, por exemplo: atuando na camada de apresentação de uma aplicação, recebendo as requisições e despachando as chamadas para os métodos de serviço adequados;
- **@org.springframework.stereotype.Component:** Este é um estereótipo generalista, apenas indicando que a classe anotada resultará em um objeto criado pelo *container* Spring. Os estereótipos apresentados anteriormente são especializações de **Component**.

Em aplicações que necessitam de acesso a informações persistidas em banco de dados, como é o caso do sistema de Controle de Tarefas, é comum utilizar o controle de transações oferecido pelo Spring em sua forma declarativa. Este controle pode ser conseguido através do uso da anotação **@org.springframework.transaction.annotation.Transactional** em conjunto com as configurações adequadas na execução do *container* Spring. Estas configurações basicamente consistem na disponibilização de um *bean* do tipo **org.springframework.transaction.PlatformTransactionManager** e a ativação do controle de transações por meio do escaneamento das anotações **@Transactional**.

Com a configuração realizada, as classes e/ou métodos de *beans* anotados com **@Transactional** recebem o comportamento transacional, definindo o limite existencial de uma transação, ou seja, onde esta é iniciada e finalizada. O controle transacional executado pelo *container* Spring é feito de acordo com os atributos fornecidos na anotação **@Transacional** em relação à propagação, isolamento, *timeout*, entre outros.

Em uma aplicação desenvolvida em camadas, como a que estamos apresentando, a anotação **@Transactional** geralmente é aplicada nas classes de objetos de serviço, ou seja, aquelas anotadas com o estereótipo **@Service**. As classes de serviço encapsulam a lógica de negócios, que para ser executada, pode requerer a recuperação e/ou alteração de dados persistidos através do uso de um ou mais objetos do tipo **@Repository**.

O processamento da lógica de negócios associada com determinada funcionalidade pode demandar a chamada de vários métodos em sequência, e neste caso é necessário obter a atomicidade desta execução, ou seja, todos os métodos envolvidos devem ser

executados com sucesso para que a alteração seja efetivada, ou então nada será feito.

A necessidade de atomicidade na execução é conseguida com a aplicação da anotação `@Transactional` nas classes/métodos onde serão iniciadas e finalizadas as operações.

Esta anotação possui o seguinte comportamento padrão para *commit* e *rollback* de alterações:

- O lançamento de exceções não checadas (*Runtime*) durante a execução de um método transacional irá marcar a transação corrente para *rollback*;
- O lançamento de exceções checadas durante a execução de um método transacional não irá marcar a transação corrente para *rollback*.

Este comportamento padrão pode ser modificado através dos atributos da anotação `@Transactional`, por exemplo, definindo o *rollback* da transação corrente em caso de lançamento de exceções checadas com a indicação de uma exceção específica ou uma hierarquia de exceções.

No sistema de Controle de Tarefas, a lógica de negócios relativa ao gerenciamento das tarefas é sempre iniciada por métodos da classe de serviço `br.com.jm.tarefas.application.GerenciamentoTarefasService`. Os *beans* desse tipo exigem os seguintes comportamentos para permitir o funcionamento adequado da lógica de negócio desenvolvida:

- Como será utilizado o controle transacional declarativo do Spring, os objetos (*beans*) desta classe devem ser criados e gerenciados pelo Spring;
- Os métodos desta classe deverão estar contemplados com comportamento transacional;
- Alguns métodos irão lançar exceções checadas, subclasses da exceção `br.com.jm.tarefas.domain.ControleTarefasException`, que expressam a violação de regras de negócios. A ocorrência destas exceções deverá promover o *rollback* da transação corrente.

Os comportamentos requeridos levantados anteriormente em relação à classe `GerenciamentoTarefasService` podem ser cumpridos facilmente por meio da utilização do estereótipo `@Service` e da anotação `@Transactional` com as devidas configurações de *rollback*. Entretanto, como estas anotações são utilizadas separadamente, pode ser esquecida a aplicação da anotação ou a configuração do comportamento transacional adequado, levando a um funcionamento incorreto da aplicação.

Para minimizar a ocorrência desse tipo de erro, como também consolidar a configuração necessária através de uma única anotação, utilizaremos o recurso de meta anotações, já disponível no Spring Framework em versão anterior à versão 4. Uma meta anotação é uma anotação composta de outras anotações, ou seja, uma ou mais anotações aplicadas a uma anotação.

Com o intuito de evidenciar a evolução do recurso de meta anotações do Spring, vamos criar duas versões da anotação customizada `@br.com.jm.tarefas.infrastructure.ServicoTransacional` para consolidação da configuração.

A primeira irá demonstrar como atingir este objetivo utilizando o suporte para criação de meta anotações do Spring 3, e a segunda com recursos de criação de meta anotações presentes a partir do Spring 4.

A implementação da primeira versão da anotação `@ServicoTransacional` pode ser verificada no código da [Listagem 4](#).

Listagem 4. Anotação composta @ServicoTransacional.

```
package br.com.jm.tarefas.infrastructure;  
  
//imports omitidos...  
  
@Documented  
@Service  
@Transactional(rollbackFor = ControleTarefasException.class)  
@Target({ ElementType.TYPE })  
@Retention(RetentionPolicy.RUNTIME)  
public @interface ServicoTransacional {  
}
```

Nesse código podemos notar que estão contemplados os requisitos de funcionamento para o serviço do tipo `GerenciamentoTarefasService`. Com esta composição das anotações `@Service` e `@Transactional` em uma anotação customizada, a ocorrência de erros por falta de configuração é minimizada.

Porém, esta versão da anotação define um conjunto de configuração estático, que não permite, por exemplo, redefinir de maneira simples o comportamento transacional de *rollback* caso seja necessário. Na existência desta necessidade, a anotação `@Transactional` teria que ser aplicada separadamente com as configurações desejadas para sobreescriver o comportamento definido na anotação composta.

No Spring Framework 4, o recurso de meta anotações foi estendido, permitindo flexibilizar as anotações customizadas com a redefinição de alguns dos atributos existentes nas anotações encapsuladas pela anotação composta, com exceção do atributo `value`, que possui contexto distinto para cada anotação.

Na [Listagem 5](#) está a segunda versão da anotação customizada `@ServicoTransacional`. Desta vez, explorando o recurso de redefinição de atributos.

Agora, ao aplicar a anotação `@ServicoTransacional`, é possível redefinir os atributos `propagation`, `readOnly`, `rollbackFor` e `noRollbackFor`, todos originados da anotação encapsulada `@Transactional`. Merece destaque o atributo `rollbackFor`, que já recebe como configuração padrão a classe de exceção `ControleTarefasException`. Por meio da redefinição destes atributos na anotação `@ServicoTransacional` torna-se possível reconfigurar os atributos da anotação encapsulada `@Transactional`.

Para que uma classe obtenha todo o comportamento estabelecido pela anotação customizada `@ServicoTransacional`, basta aplicá-la conforme mostra o trecho de código da classe `GerenciamentoTarefasService`, exibido na [Listagem 6](#).

Nesse caso não foi necessário redefinir nenhum atributo para atingir o comportamento requerido. Caso fosse necessário que

as transações tivessem o comportamento apenas de leitura, basta redefinir o atributo `readOnly` com o valor `true`, conforme o exemplo: `@ServicoTransacional(readOnly = true)`.

Listagem 5. Anotação composta `@ServicoTransacional` com redefinição de atributos.

```
package br.com.jm.tarefas.infrastructure;  
  
//imports omitidos...  
  
@Documented  
@Service  
@Transactional  
@Target({ ElementType.TYPE })  
@Retention(RetentionPolicy.RUNTIME)  
public @interface ServicoTransacional {  
  
    Propagation propagation() default Propagation.REQUIRED;  
  
    boolean readOnly() default false;  
  
    Class<? extends Throwable>[] rollbackFor() default { ControleTarefasException.class };  
  
    Class<? extends Throwable>[] noRollbackFor() default {};  
  
}
```

Listagem 6. Utilização da anotação `@ServicoTransacional`.

```
package br.com.jm.tarefas.application;  
  
//imports omitidos...  
  
@ServicoTransacional  
public class GerenciamentoTarefasService {  
  
    //implementação omitida...  
}
```

Utilizando JPA 2.1 via Spring Data JPA

O Spring Framework já vem oferecendo recursos para facilitar o desenvolvimento de código para o acesso a dados em bancos relacionais desde seu início, em 2004. Estes recursos incluem o suporte ao Hibernate, JDBC, JDO, iBatis e, mais recentemente, JPA.

Com o lançamento do projeto Spring Data em 2011, o conjunto de recursos oferecidos para o acesso a dados ficou ainda melhor. Este projeto disponibiliza módulos específicos para o desenvolvimento de código para manipulação de dados de acordo com o armazenamento utilizado, não estando restrito apenas a bancos de dados relacionais, com acesso via JDBC ou JPA, mas também viabilizando suporte ao armazenamento NoSQL.

O módulo Spring Data JPA é específico para a tecnologia JPA, ou seja, trabalha exclusivamente com a manipulação de objetos (entidades) persistidos em bancos de dados relacionais. No sistema de Controle de Tarefas, foi utilizada a versão 1.7.0.RELEASE do Spring Data JPA, que além do suporte à API JPA 2.1, também utiliza como base o Spring Framework 4.

O objetivo do Spring Data JPA é facilitar a construção da camada de persistência em JPA, criando uma abstração dos conceitos que

envolvem o repositório de dados, ou seja, as operações básicas para a recuperação, o armazenamento e a remoção de dados. O Spring Data JPA disponibiliza algumas interfaces específicas para a criação de repositórios JPA, que são originadas de interfaces base disponibilizadas pelo módulo Spring Data e que contêm operações básicas e adicionais, como CRUD, paginação e ordenação.

A seguir, são listadas algumas das interfaces de repositório oferecidas pelo Spring Data e Spring Data JPA:

- **org.springframework.data.repository.Repository**: Interface base (marcadora) para todas as subinterfaces de repositório;
- **org.springframework.data.repository.CrudRepository**: Interface que oferece as operações básicas de inclusão, consulta, alteração e remoção de dados. Esta interface estende a interface **Repository**;
- **org.springframework.data.repository.PagingAndSortingRepository**: Interface de repositório que oferece as operações de paginação e ordenação de dados. Esta interface estende a interface **CrudRepository**;
- **org.springframework.data.jpa.repository.JpaRepository**: Enquanto as três interfaces anteriores são agnósticas ao tipo de armazenamento, esta interface já trata especificamente da manipulação dos dados via JPA. Ela oferece todas as operações das interfaces listadas anteriormente, já que estende a interface **PagingAndSortingRepository**, e ainda disponibiliza algumas operações específicas da JPA, como **flush()** e **saveAndFlush()**.

No sistema de Controle de Tarefas foram criadas as interfaces de repositório JPA `br.com.jm.tarefas.domain.TarefaRepository` e `br.com.jm.tarefas.domain.UsuarioRepository` para manipulação das entidades `Tarefa` e `Usuario`, respectivamente. De acordo com o código detalhado nas **Listagens 7** e **8**, além das operações comuns de um repositório herdadas da interface `JpaRepository`, também foram definidos métodos específicos relacionados a cada entidade.

Como pode ser visto no código das interfaces de repositório `TarefaRepository` e `UsuarioRepository`, o Spring Data JPA utiliza algumas convenções para criação das consultas a partir dos métodos declarados nestas interfaces. Por exemplo, o método `findByEmail(String email)` resulta na geração de uma consulta JPQL equivalente ao seguinte trecho: `"select entidade from Entidade entidade where entidade.email = ?"`. A lista com as opções convencionadas, como `And`, `Or`, `Is`, `Equals`, entre outras, é extensa e pode ser consultada na documentação de referência do Spring Data JPA.

O uso desta convenção pode ser notado pelos métodos das interfaces `TarefaRepository` e `UsuarioRepository`. No método `TarefaRepository.findByResponsavel(Usuario usuario, Sort sort)`, por exemplo, será gerada uma consulta que irá recuperar as tarefas cujo usuário responsável corresponde ao usuário recebido como argumento, além de ordenar as tarefas de acordo com os parâmetros de ordenação definidos pelo argumento recebido do tipo `org.springframework.data.domain.Sort`.

Spring Framework: Criando uma aplicação web – Parte 1

Listagem 7. Reppositório JPA para a entidade Tarefa.

```
package br.com.jm.tarefas.domain;  
  
// imports omitidos...  
  
@Profile("prod")  
public interface TarefaRepository extends JpaRepository<Tarefa, Long>, Gerador-  
CodigoTarefa {  
  
    @Override  
    @Procedure(name = "Tarefa.proximoCodigo")  
    public Long proximoCodigoTarefa();  
  
    public List<Tarefa> findByResponsavel(Usuario usuario, Sort sort);  
  
    public List<Tarefa> findByStatus(StatusTarefa status, Sort sort);  
  
    public Optional<Tarefa> findByIdentificador(IdentificadorTarefa identificador);  
  
}
```

Listagem 8. Reppositório JPA para a entidade Usuario.

```
package br.com.jm.tarefas.domain;  
  
// imports omitidos...  
  
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {  
  
    public Optional<Usuario> findByIdentificador  
        (IdentificadorUsuario identificadorUsuario);  
  
}
```

Caso seja necessário, ao invés de utilizar a consulta gerada, pode ser fornecida a consulta que deve ser executada na implementação de reppositório oferecida pelo Spring Data JPA, aplicando algumas anotações nos métodos do reppositório. Estas anotações estão listadas a seguir:

- **@org.springframework.data.jpa.repository.Query:** Permite informar qual consulta deverá ser executada pelo método. Neste caso, o Spring Data JPA não irá gerar uma consulta, mas sim utilizar a que foi fornecida;
- **@org.springframework.data.jpa.repository.query.Procedure:** Esta anotação, quando aplicada em um método de reppositório, resulta na execução de uma *Stored Procedure* disponível no banco de dados. Esta execução já utiliza o recurso de *Stored Procedures* existente na API JPA 2.1;
- **@org.springframework.data.jpa.repository.Modifying:** Com esta anotação é possível indicar que um método está realizando alteração nos dados, permitindo inclusive parametrizar a limpeza (*clear*) do contexto de persistência após a execução do mesmo.

Conforme demonstrado no código do reppositório de Tarefas da **Listagem 7**, foi utilizado o recurso de *Stored Procedures* da JPA 2.1 no método **TarefaRepository.proximoCodigoTarefa()**, através do uso da anotação **@Procedure**, indicando o nome da procedure **Tarefa.proximoCodigo**. Como pode ser visto no código da **Listagem 9**, esta *procedure* foi declarada no mapeamento da

entidade **Tarefa**, por meio da anotação da API JPA **@javax.persistence.NamedStoredProcedureQuery**.

A *Stored Procedure* **Tarefa.proximoCodigo** simplesmente fornece a geração de um número sequencial para a identificação das novas tarefas cadastradas.

Listagem 9. Mapeamento da Stored Procedure Tarefa.proximoCodigo na entidade Tarefa.

```
package br.com.jm.tarefas.domain;  
  
//imports omitidos...  
  
@Entity  
@NamedStoredProcedureQuery(name = "Tarefa.proximoCodigo",  
    procedureName = "PROXIMO_CODIGO_TAREFA",  
    parameters = { @StoredProcedureParameter(mode = ParameterMode.OUT,  
        type = Long.class,  
        name = "PROXIMO_CODIGO" ) })  
public class Tarefa {  
  
    //implementação omitida...
```

Com esta abordagem, conseguimos utilizar o recurso de *Stored Procedures* fornecido pela API JPA através da implementação de reppositórios oferecida pelo Spring Data JPA, o que evidencia a simplicidade na utilização de *procedures* via JPA 2.1. Pelo uso da informação de mapeamento JPA da *procedure* realizado na entidade, todo o código necessário para execução fica a cargo do Spring Data JPA, bastando apenas fornecer a assinatura do método (retorno e parâmetros) compatível com a *procedure* a ser executada.

Beans Condicionais (@Conditional)

Um aspecto importante durante o desenvolvimento de uma aplicação é a flexibilização permitida em seu conjunto de configuração. Durante o processo de evolução, esta aplicação sem dúvida será executada em diferentes ambientes e/ou situações.

No sistema de Controle de Tarefas temos um exemplo desta necessidade, a de permitir a variação na configuração para a utilização de um banco de dados em memória, o HyperSQL (HSQLDB), para facilitar a execução de testes e a execução do sistema no ambiente desenvolvimento. Para os demais ambientes suportados, simulando o ambiente de execução final, o banco de dados deve ser o PostgreSQL. Como a camada de persistência da aplicação é desenvolvida com tecnologia JPA, o suporte a bancos de dados distintos exige configurações diferentes de *Data Source* e Unidade de Persistência.

Com o uso do recurso de perfis existente no Spring, é possível definir trechos da configuração do *container* que serão utilizados de acordo com o(s) perfil(is) ativo(s) em tempo de execução.

Na **Listagem 10** está um trecho da classe **Config** com o conjunto de criação dos *beans* de *Data Source* e Unidade de Persistência utilizados na configuração do *container* Spring para o sistema de Controle de Tarefas. Este conjunto contém a configuração para execução utilizando o banco de dados PostgreSQL, prevendo um ambiente de produção, sendo ativado pelo uso do perfil “**prod**”.

Listagem 10. Trecho de configuração dos beans de Data Source e Unidade de Persistência para o perfil prod.

```
package br.com.jm.tarefas.infrastructure.config;

//imports omitidos...

@Configuration
@ComponentScan(basePackages = "br.com.jm.tarefas")
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = { "br.com.jm.tarefas.domain" })
@Import(DevConfig.class)
public class Config {

    @Bean
    @Profile("prod")
    public DataSource dataSource() {
        DriverManagerDataSource dmnds = new DriverManagerDataSource();
        dmnds.setUrl("jdbc:postgresql://localhost:5432/ctrl_tarefas");
        dmnds.setUsername("ctrl_tarefas");
        dmnds.setPassword("secret");
        dmnds.setDriverClassName("org.postgresql.Driver");
        return dmnds;
    }

    @Bean
    @Profile("prod")
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        HibernateJpaVendorAdapter jpaVendorAdapter =
            new HibernateJpaVendorAdapter();
        jpaVendorAdapter.setDatabase(Database.POSTGRES);
        LocalContainerEntityManagerFactoryBean emfb =
            new LocalContainerEntityManagerFactoryBean();
        emfb.setJpaVendorAdapter(jpaVendorAdapter);
        emfb.setDataSource(dataSource());
        return emfb;
    }

    //demais beans omitidos...
}

}
```

Outro conjunto de configuração, exibido na **Listagem 11**, é destinado para execução com o banco de dados em memória HyperSQL, sendo ativado pelo perfil de desenvolvimento “**dev**”.

Entretanto, mesmo com a utilização de perfis, é útil em determinadas situações registrar um *bean* no *container* Spring que não esteja diretamente ligado à ativação de um determinado perfil. Durante a execução do sistema de Controle de Tarefas, seja no ambiente de desenvolvimento ou em QA, é interessante existir a opção de realizar uma carga inicial no banco de dados, por exemplo, para testar o comportamento durante a implementação ou mesmo para a execução de testes funcionais.

Com a inclusão do novo recurso de *beans* condicionais disponível a partir da versão 4 do Spring Framework, é possível criar uma condição para a criação de um *bean* independente do perfil(s) selecionado(s). Este recurso foi utilizado para registrar um *bean* responsável por fazer a carga dos dados durante a criação do *container* Spring.

O primeiro passo para isso foi criar a classe **br.com.jm.tarefas.infrastructure.InicializadorBD**, exibida na **Listagem 12**. Com essa implementação, após a instanciação e injeção das dependências do *bean* do tipo **InicializadorBD**, o método **inicializar()**,

Listagem 11. Trecho de configuração dos beans de Data Source e Unidade de Persistência para o perfil dev.

```
package br.com.jm.tarefas.infrastructure.config;

//imports omitidos...

@Configuration
@Profile("dev")
public class DevConfig {

    @Bean
    public DataSource dataSource() {
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
        EmbeddedDatabase db = builder.setType(EmbeddedDatabaseType.HSQL).
            build();
        return db;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean emfb = new LocalContainer
            EntityManagerFactoryBean();
        emfb.setPersistenceUnitManager(defaultPersistenceUnitManager());
        HibernateJpaVendorAdapter jpaVendorAdapter = new HibernateJpa
            VendorAdapter();
        jpaVendorAdapter.setDatabase(Database.HSQL);
        emfb.setJpaVendorAdapter(jpaVendorAdapter);
        emfb.setDataSource(dataSource());
        return emfb;
    }

    //demais beans omitidos...
}

}
```

anotado com **@javax.annotation.PostConstruct**, será executado e irá inserir alguns dados de exemplo na base de dados em uso.

Depois do desenvolvimento da classe **InicializadorBD**, é necessário criar uma implementação específica da interface **org.springframework.context.annotation.Condition**, permitindo estabelecer a condição necessária para o registro do *bean* do tipo **InicializadorBD**. Uma implementação da interface **Condition** deve retornar **true/false**, indicando se o *bean/beans* deve(m) ser registrado(s) ou não no *container*.

A implementação da condição para carga do banco de dados foi efetuada na classe **br.com.jm.tarefas.infrastructure.config.InicializadorBDCondition**, com o código disponível na **Listagem 13**, baseando-se na existência e valor da propriedade de sistema **inicializadorBD**.

A partir deste ponto, com o uso da anotação **@org.springframework.context.annotation.Conditional** é possível informar uma ou mais condições para o registro de um ou mais beans no *container* Spring. De acordo a necessidade do sistema de Controle de Tarefas, vamos utilizar apenas a condição **InicializadorBDCondition**, conforme mostra o trecho de configuração da **Listagem 14**.

Com esse trecho de configuração, o *bean* **inicializadorBD** será registrado no *container* somente se a propriedade de sistema **inicializadorBD** existir e conter o valor **true**.

Spring Framework: Criando uma aplicação web – Parte 1

Listagem 12. Implementação da classe InicializadorBD.

```
package br.com.jm.tarefas.infrastructure;

//imports omitidos...

public class InicializadorBD {

    @Autowired
    private PlatformTransactionManager transactionManager;

    @Autowired
    private GeradorCodigoTarefa geradorCodigoTarefa;

    @Autowired
    private JpaRepository<Usuario, Long> usuarioRepository;

    @Autowired
    private JpaRepository<Tarefa, Long> tarefaRepository;

    private DadosExemplo dadosExemplo;

    @PostConstruct
    public void inicializar() {
        dadosExemplo = new DadosExemplo(geradorCodigoTarefa);
        TransactionTemplate transactionTemplate = new TransactionTemplate(
            transactionManager);
        transactionTemplate.execute((ts) -> {
            carregarDados();
            return null;
        });
    }

    private void carregarDados() {
        usuarioRepository.save(dadosExemplo.getFulano());
        usuarioRepository.save(dadosExemplo.getBeltrano());
        usuarioRepository.save(dadosExemplo.getSicrano());
        usuarioRepository.save(dadosExemplo.getJohnDoe());
        tarefaRepository.save(dadosExemplo.getTarefaCadastrada());
        tarefaRepository.save(dadosExemplo.getTarefaIniciada());
    }
}
```

Listagem 13. Implementação da condição InicializadorBDCondition.

```
package br.com.jm.tarefas.infrastructure.config;

//imports omitidos...

public class InicializadorBDCondition implements Condition {

    private static final String INICIALIZADOR_BD_PROPERTY_KEY = "inicializadorBD";

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        return Boolean.valueOf(System.getProperty(INICIALIZADOR_BD_PROPERTY_KEY));
    }
}
```

Listagem 14. Aplicação da condição para criação do bean inicializadorBD.

```
@Bean
@Conditional(InicializadorBDCondition.class)
public InicializadorBD inicializadorBD() {
    return new InicializadorBD();
}
```

Para demonstrar o funcionamento, na **Listagem 15** está o trecho de configuração do plugin *jetty-maven-plugin* no *pom.xml*, para execução do sistema de Controle de Tarefas com o servidor Jetty no ambiente de desenvolvimento.

Listagem 15. Trecho de configuração do pom.xml para execução do sistema no ambiente de desenvolvimento.

```
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <version>9.2.2.v20140723</version>
    <configuration>
        <systemProperties>
            <systemProperty>
                <name>inicializadorBD</name>
                <value>true</value>
            </systemProperty>
            <systemProperty>
                <name>spring.profiles.active</name>
                <value>dev</value>
            </systemProperty>
        </systemProperties>
        <scanIntervalSeconds>10</scanIntervalSeconds>
        <webApp>
            <contextPath>/controle-tarefas</contextPath>
        </webApp>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>2.3.2</version>
        </dependency>
    </dependencies>
</plugin>
```

Na configuração do plugin *jetty-maven-plugin* consta a definição de duas propriedades de sistema, a saber:

- **inicializadorBD**: Recebe o valor **true** para ativar o registro do bean **inicializadorBD** e inserir os dados de exemplo na base de dados;
- **spring.profiles.active**: Recebe o valor **"dev"** para ativar o perfil de desenvolvimento no *container* Spring.

Com as propriedades ajustadas para estes valores, o *container* Spring será criado com o perfil de desenvolvimento, inicializando o banco de dados com os dados de exemplo a partir do bean **inicializadorBD**.

A definição do(s) *framework*(s) utilizado(s) é fator crucial na evolução de uma aplicação. Como exemplo, lembre-se que uma aplicação pode iniciar com uma pretensão simples e depois ter que evoluir para atender necessidades diferentes e/ou maiores. Sendo assim, a escolha inicial da plataforma base pode ser a di-

ferença entre uma evolução natural ou a reescrita completa para adequação frente a novas demandas.

Diante disso, a plataforma Spring tem como vantagem a constante evolução e a grande gama de soluções oferecidas pelo próprio framework e seus diversos módulos.

Apesar do exemplo neste artigo apresentar um projeto iniciado do zero, não existe impedimento para que o conteúdo aqui mostrado seja empregado como base para a migração de um projeto que faça uso de versões anteriores do Spring. Esta migração pode ser feita de forma gradativa, sendo até recomendada, já que a nova versão do framework obteve significante modernização e expansão do suporte a novas tecnologias.

Autor



Francisco A. Garcia

franciscogrc@gmail.com

É formado em Ciência da Computação pela Universidade Paulista. Trabalha como desenvolvedor Java desde 2006. Possui as certificações SCJP, SCWCD, OCBSD, OCEWSD.



Links:

Documentação de referência do Spring Framework versão 4.1.0 RELEASE (em formato PDF).

<http://docs.spring.io/spring/docs/4.1.0.RELEASE/spring-framework-reference/pdf/spring-framework-reference.pdf>

Documentação de referência do Spring Data JPA versão 1.7.0 RELEASE (em formato PDF).

<http://docs.spring.io/spring-data/jpa/docs/1.7.0.RELEASE/reference/pdf/spring-data-jpa-reference.pdf>

Post anunciando o release da versão 1.0 do Spring Framework.

<https://spring.io/blog/2004/03/24/spring-framework-1-0-final-released>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



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

Como aproveitar o máximo das Expressões Lambda em Java

Como aproveitar ao máximo as vantagens das expressões lambda em seu próximo projeto

O dia 25 de março de 2014 foi marcado pelo lançamento da tão aguardada versão 8 do Java. Esta nova versão introduziu dezenas de novidades, que abrangem desde novas APIs até mudanças na sintaxe da linguagem. Já vimos algumas destas novidades na edição 120 da Java Magazine, embora não tenhamos utilizado cenários reais onde elas pudessem ser aplicadas. Entretanto, passado todo o frenesi, comum em épocas de lançamento de novas versões, ainda são poucos os projetos escritos desde o início adotando essas novas características.

Isto ocorre por diversos motivos, sendo que um deles é que cada nova versão de uma linguagem requer um tempo de adaptação por parte dos desenvolvedores. Muitos deles sequer se acostumaram com as novidades do Java 7 e já têm à sua porta uma avalanche de mudanças que o Java 8 introduziu. Este é o cenário que você se encontra inserido agora? Caso sim, o objetivo deste artigo é lhe ajudar a se familiarizar mais rápido com uma dessas grandes mudanças: as expressões lambda!

As expressões lambda foram bastante comentadas, aguardadas e até mesmo contestadas pela comunidade Java, pois viriam para mudar a sintaxe do Java de uma forma nunca imaginada, introduzindo conceitos sómente vistos em linguagens funcionais. Com o Java 8 já lançado e ao nosso dispor, agora nos cabe aprender e pôr em prática as facilidades que essa nova funcionalidade nos brindou.

Neste contexto, o objetivo deste artigo é apresentar diversos exemplos de uso reais das expressões lambda e que lhe permitirão, mesmo que agora você não esteja em um projeto que use o Java 8, vislumbrar os possíveis

Fique por dentro

Embora a versão 8 do Java tenha sido lançada há pouco tempo e poucos projetos a estejam usando, é um grande diferencial antecipar-se às novidades da linguagem, aprendendo como essas mudanças irão impactar sua vida futuramente. Isto torna-se ainda mais importante quando uma dessas mudanças impõe uma forma diferente de se programar na linguagem Java, como é o caso das expressões lambda. Sendo assim, este artigo lhe introduzirá às expressões lambda e como você pode se preparar para elas. Após ler este artigo, temos certeza que você terá vontade de migrar o mais rápido possível para desde já utilizá-las em seus projetos!

cenários onde elas serão de extrema utilidade para seus futuros aplicativos. Esperamos que este artigo, e os exemplos aqui apresentados, lhe forneça os motivos pelos quais você deveria migrar para o Java 8 apenas para ter esta nova funcionalidade ao seu dispor.

Mergulhando nas Expressões Lambda

Como tudo começou

Antes de iniciarmos um estudo mais detalhado sobre as expressões lambda, começaremos entendendo os motivos de elas terem sido introduzidas na linguagem, pois, para alguns desavisados, sua inclusão pode parecer apenas um capricho ou que o Java está incluindo características desnecessárias. Entretanto, o cenário é muito diferente disto, pois as expressões lambda surgiram para solucionar um problema histórico e que demorou bastante tempo até ser totalmente aceito e resolvido pela comunidade de desenvolvedores Java.

Para entender melhor este panorama, precisamos relembrar da primeira versão do Java, quando o *Abstract Toolkit Window* (AWT)

foi sugerido como ferramenta para a construção de interfaces para *desktop*. Em sua primeira versão, o AWT forçava que o desenvolvedor sobrescrevesse métodos herdados de suas classes para que determinados eventos, gerados a partir da interação do usuário com o aplicativo, fossem devidamente tratados.

Um exemplo desta solução pode ser visto na **Listagem 1**, onde o método **action()**, oriundo da classe **java.awt.Component**, é sobreescrito para que o clique em um botão seja tratado. Esta solução, entretanto, não é recomendada como uma boa prática por diversos motivos, por exemplo, devido ao forte acoplamento com a classe-pai, e logo foi substituída, a partir da versão 1.1, pelas interfaces conhecidas apenas como *Listeners*.

A partir da criação dos *Listeners*, tornou-se desnecessário estender classes, pois, agora, precisamos implementar determinadas interfaces que contêm um ou mais métodos específicos para o tratamento de eventos. Por exemplo, há uma interface que deve ser implementada para tratar o evento gerado pelo clique em um botão (veja um exemplo na **Listagem 2**).

Listagem 1. Como a primeira versão do AWT tratava as interações do usuário com a tela.

```
import java.applet.Applet;
import java.awt.*;

public class AWT1Exemplo extends java.awt.Frame {
    private Button botao;

    public void init() {
        botao = new Button("Botão");
        add(botao);
    }

    public boolean action(Event e, Object args) {
        // Tratar o evento aqui.
        return true;
    }
}
```

Listagem 2. Exemplo de uso dos *Listeners* para tratar eventos.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

class AWT1Exemplo extends Frame {
    private Button botao;

    public AWT1Exemplo() {
        botao = new Button("Botão");
        add(botao);

        botao.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                System.out.println("Clicou")
            }
        });
    }
}
```

Esta listagem nos apresenta duas novidades com relação à primeira versão do Java. A primeira novidade refere-se ao uso da interface **ActionListener** para tratar o evento de clique no botão, em oposição ao antigo modelo de sobreescrita de métodos da classe herdada. A segunda novidade é a forma como uma instância desta interface foi criada, pois utilizamos uma nova característica, adicionada a partir da versão 1.1, chamada Classe Interna Anônima (*Anonymous Inner Class*).

Este novo recurso foi incluído com o intuito de facilitar a vida dos programadores, evitando a necessidade de criar uma classe em separado, que estende de **ActionListener**, apenas para tratar uma situação específica, e que não será reutilizada em outras partes do projeto, como o tratamento do clique em um botão contido em uma tela específica do aplicativo.

O problema

Após a sua criação, as Classes Internas Anônimas passaram a ser também adotadas em outros cenários, como para a instanciação de objetos do tipo **Runnable** com o objetivo de executar um trecho de código em paralelo através do uso das **Threads**. Seguindo esta mesma linha, as classes de coleções passaram a aceitar uma instância da interface **Comparator** para ordenar o seu conteúdo, e que, na maioria das vezes, é instanciada como uma classe anônima. Posteriormente, o Swing seguiu a mesma estratégia do AWT e também criou classes *Listeners* que quase sempre são instanciados de forma anônima.

Embora as Classes Internas Anônimas solucionassem o problema, o fato é que elas impõem uma grande quantidade de código *boilerplate* a ser escrito, ou seja, código sem utilidade real. Com o surgimento de novas linguagens, como o Groovy, que adotam um modelo híbrido entre linguagens orientadas a objeto e linguagens funcionais, tornou-se claro que o modelo empregado pelo Java já estava ultrapassado e que o mercado pedia uma melhora nesse aspecto, visando um código mais limpo, enxuto e elegante.

A solução

A saída encontrada para solucionar este problema consistiu em incorporar ao Java algumas das características largamente adotadas em linguagens funcionais, o que implicou em um conjunto de mudanças na sintaxe que tornaram a tarefa de criar trechos de código para futura execução mais simples e prática. O nome escolhido para esta nova funcionalidade foi expressões lambda, embora também seja possível encontrá-la com as alcunhas *closures* ou métodos anônimos (*anonymous methods*). No decorrer deste artigo usaremos estes três nomes, de forma intercambiável, para nos referir a este mesmo conceito.

Nosso primeiro exemplo de uso dos métodos anônimos encontra-se na **Listagem 3**, onde reescrivemos o código da **Listagem 2** para usar as expressões lambda no lugar de classes anônimas. Nesta listagem, adotamos a sintaxe das expressões lambda para criar um método anônimo e passá-lo como parâmetro para o método **addActionListener()** do objeto **botao**. Observe como o código da **Listagem 3** é mais limpo e conciso.

Como aproveitar o máximo das Expressões Lambda em Java

Listagem 3. Usando expressões lambda para simplificar o uso de Listeners em Swing.

```
01. class Swing1Exemplo extends Frame {  
02.     private JButton botao;  
03.  
04.     public Swing1Exemplo () {  
05.         botao = new JButton("Botão");  
06.         add(botao);  
07.  
08.         botao.addActionListener(e-> System.out.println("Clicou"));  
09.     }  
10. }
```

Revisando a sintaxe

Antes de analisarmos os cenários onde as expressões lambda são úteis, faremos uma rápida revisão de sua sintaxe usando as três expressões listadas como exemplo a seguir. Na primeira expressão, temos um método virtual que recebe dois parâmetros do tipo **double** e retorna a multiplicação entre ambos. Neste exemplo, o primeiro detalhe a ser observado é uma das possíveis notações para definir os parâmetros que esta expressão receberá. Neste caso em particular, eles foram definidos entre parênteses e separados por vírgula. Os últimos dois detalhes importantes são o fato de que não necessitamos usar a palavra-chave **return** para informar o valor retornado por esta função e não necessitamos informar o tipo do valor retornado (ele é inferido a partir do tipo do objeto retornado).

```
01. (double a, double b) -> a * b  
02. (s) -> { System.out.println("Olá " + s); }  
03. a -> a * 2
```

A segunda expressão, mais simples que a anterior, recebe como parâmetro uma **String** e imprime este valor concatenado com o texto “Olá”. Este segundo trecho demonstra mais um detalhe importante sobre a definição de parâmetros: não há a necessidade de informar o tipo do parâmetro, pois ele será dinamicamente inferido quando esta lambda for passada como parâmetro para um método. Por último, temos uma lambda que recebe como parâmetro um número e retorna a multiplicação deste número por dois. Este último trecho demonstra mais uma peculiaridade: quando temos apenas um parâmetro para a lambda, não necessitamos sequer colocá-lo entre parênteses.

Concluindo esta revisão, a sintaxe das expressões lambda requer que logo após a definição dos parâmetros utilizemos a notação denominada **arrow**, representada pelo caractere – (traço) seguido de um > (sinal de maior). Logo em seguida, caso a expressão contenha mais de uma linha, devemos definir um bloco de código contido dentro de chaves. Entretanto, caso a lambda possua apenas uma linha de código, basta colocar seu código logo após o **arrow**, sem necessidade de finalizar com um ponto e vírgula.

Lambda na prática

Após uma breve revisão da sintaxe, já estamos aptos a iniciar o uso das expressões lambda através de um exemplo prático.

Durante as próximas subseções descreveremos a aplicação que usaremos como base para explicar as lambdas, além de demonstrarmos algumas novas APIs que tiram proveito da simplicidade da sua sintaxe. Contudo, destacamos que essas novas APIs poderiam existir sem essas expressões, mas seu uso torna-se muito mais simples e intuitivo quando utilizadas em conjunto com os métodos anônimos.

Aplicação de exemplo: Cervejaria

A aplicação fictícia que criaremos será um sistema de cadastro e avaliação de cervejas, através da qual teremos operações básicas de cadastro (criar, editar, remover e obter), além da possibilidade de dar notas, que variam entre 0 e 10, às cervejas que cadastramos. Ressaltamos que não detalharemos exaustivamente todas as nuances que não estão diretamente relacionadas aos conceitos dos métodos virtuais, pois nosso foco estará voltado para os trechos de código que envolvem o uso das *closures*. Caso tenha interesse em ter este código executando em sua máquina, basta baixá-lo do GitHub do autor, cujo link está disponível na seção **Links**.

Esta aplicação contará apenas com três entidades, que serão nomeadas como **Cerveja**, **Fabricante** e **Tipo**. As entidades **Fabricante** (**Listagem 4**) e **Tipo** (**Listagem 5**) contarão apenas com os atributos **id** e **nome**, enquanto a entidade **Cerveja** (**Listagem 6**) será um pouco mais complexa, contendo os atributos **id**, **nome**, **fabricante**, **tipo**, **nota** e **data**.

Além destas classes de entidade, também contaremos com uma classe chamada **Cervejaria**, apresentada na **Listagem 7**. Esta será responsável por manter uma coleção contendo as cervejas cadastradas no sistema, além das coleções de fabricantes e de tipos de cervejas. **Cervejaria** conta com métodos que permitem realizar ações nestas coleções, ou nos dados contidos nelas, como os métodos **salvar(Cerveja)** e **excluir(Long)**. Observe ainda que esta classe implementa dois padrões bastante conhecidos no mundo Java: *Façade* e *Singleton*. No decorrer deste artigo, conforme apresentamos as características das expressões lambda, adicionaremos outros métodos a esta classe.

Primeira parada: Removendo itens de coleções

O impacto mais visível causado pelas expressões lambda pode ser notado no *framework* de coleções, onde encontraremos novos métodos e conceitos que foram adicionados visando simplificar seu uso. Nesta seção, analisaremos algumas dessas mudanças, como a inclusão de alguns métodos que vão facilitar enormemente a vida do desenvolvedor.

Inicialmente, vamos analisar as linhas 17 a 19, contidas na **Listagem 7**, onde usamos uma expressão lambda para excluir uma cerveja de uma coleção através do método **excluir(Long)**. Lembra-se como este mesmo trabalho era realizado nas versões anteriores do Java? Precisávamos iterar na coleção, usando um **for**, até encontrar o objeto que desejávamos e então excluí-lo usando o método **remove(T)** de **java.util.List<T>**.

Com o Java 8, esta tarefa tornou-se bastante simples, pois agora temos o método **removeIf(Predicate<T>)**, que recebe

como parâmetro um objeto do tipo **Predicate<T>**, responsável por escolher quais objetos serão removidos. Para entendermos por completo o uso desta nova interface, disponível apenas a partir do Java 8, antes precisamos compreender o conceito de interfaces funcionais. Portanto, vamos dar uma rápida parada nesta subseção para analisarmos esta característica.

Listagem 4. Definição da entidade Fabricante.

```
public class Fabricante {  
    private Long id;  
    private String nome;  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public Long getId() {  
        return id;  
    }  
  
    public void setId(Long id) {  
        this.id = id;  
    }  
}
```

Listagem 5. Definição da entidade Tipo.

```
public class Tipo {  
    private Long id;  
    private String nome;  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public Long getId() {  
        return id;  
    }  
  
    public void setId(Long id) {  
        this.id = id;  
    }  
}
```

Listagem 6. Definição da entidade Cerveja.

```
import java.util.Date;  
  
public class Cerveja {  
    private String nome;  
    private Integer nota;  
    private Fabricante fabricante;  
    private Tipo tipo;  
    private Date data;  
  
    // Omitimos propositalmente os getters e setters.  
}
```

Listagem 7. Código fonte da classe de fachada chamada Cervejaria.

```
01 public class Cervejaria {  
02     private List<Fabricante> fabricantes = new ArrayList<>();  
03     private List<Tipo> tipos = new ArrayList<>();  
04     private List<Cerveja> cervejas = new ArrayList<Cerveja>();  
05     private static final Cervejaria instancia = new Cervejaria();  
06  
07     private Cervejaria() {  
08     }  
09  
10    public static Cervejaria getInstancia() {  
11        return instancia;  
12    }  
13  
14    public void salvar(Cerveja cerveja) {  
15        cervejas.add(cerveja);  
16    }  
17    public void excluir(Long id) {  
18        cervejas.removeIf(c -> c.getId().equals(id));  
19    }  
20}
```

Intervalo: Interfaces Funcionais

Apesar de possuir um nome próprio, uma interface funcional nada mais é do que uma interface comum do Java, mas que contém apenas um método, contando os métodos herdados de outras interfaces. Neste contexto, diversas interfaces que já existiam no Java, antes mesmo da versão 8, serão, a partir de agora, consideradas como interfaces funcionais. Exemplificando, temos as interfaces **Comparator**, **Runnable** e **ActionListener**.

A relação entre interfaces funcionais e as expressões lambda está no fato de que uma lambda pode ser passada como parâmetro para um método que aguarda como um de seus parâmetros um objeto do tipo de uma interface funcional. Apesar de a explicação ser um pouco complicada, a prática é bastante simples, como no caso da interface **Predicate<T>**, que é uma interface funcional e que contém apenas o método **test(T)**. Ao invés de criarmos um objeto desse tipo, podemos simplesmente passar uma lambda no seu lugar, desde que essa lambda receba como parâmetro apenas um objeto do tipo T.

Para tornar esta ideia mais simples de ser entendida, imagine que uma expressão lambda é um objeto que contém apenas um método, mas que não tem um tipo explicitamente definido. Este “objeto” pode ser usado no lugar de instâncias de interfaces funcionais, desde que a assinatura de seu único método coincida com a assinatura do método desta interface funcional. Contudo, alertamos que uma expressão lambda não é um objeto e que este exemplo foi apenas para ilustrar e facilitar o entendimento.

Nota

Na edição 120 da Java Magazine, no artigo Java 8: Do Lambda ao Metaspace, há uma explicação mais detalhada sobre as interfaces funcionais e seu uso. Para informações mais detalhadas sugerimos a leitura deste artigo.

Continuando com as Coleções

Voltando ao nosso exemplo, note que no método **excluir(Long)** criamos uma expressão lambda que será passada como parâmetro para o método **removeIf(Predicate<T>)**, no lugar de criar explicitamente um objeto desse tipo usando o operador **new**. Neste método virtual verificamos se o identificador da cerveja passada como parâmetro é igual ao identificador da cerveja contida na coleção. Caso seja idêntico, esta cerveja será removida.

O que o método **removeIf()** está fazendo internamente é uma iteração na lista de cervejas passando cada cerveja encontrada como parâmetro para o método **test(T)** e, caso este método retorne **true**, esta cerveja será excluída da coleção.

Nota

A interface **Predicate<T>** é nova no Java e seu objetivo, através do método **test(T)**, é verificar se o objeto passado como parâmetro atende a determinados critérios definidos pelas classes que a implementam ou pelas expressões lambda usadas no lugar dessas instâncias. Uma instância de **Predicate<T>** nunca deve mudar o estado do objeto passado como parâmetro, pois fere seu conceito básico.

Adeus às iterações usando for e while?

Nesta seção vamos conhecer o método **forEach(Consumer<T>)**, adicionado recentemente à interface **java.lang.Iterable<T>**. Não sabe que interface é esta? Não tem problemas, pois você só precisa saber que **java.util.Collection<E>** herda dela e, portanto, toda coleção tem esse método à sua disposição. Para demonstrar seu uso, vamos adicionar duas novas funcionalidades à nossa classe **Cervejaria** e que serão responsáveis por excluir um tipo e um fabricante do nosso sistema.

Embora seja uma funcionalidade bastante simples, precisamos tomar um pequeno cuidado ao tentar remover uma destas duas entidades, pois quando uma delas é excluída, necessitamos fazer algo com as cervejas cadastradas, pois existe uma relação entre estas entidades. Em nosso sistema, por exemplo, caso um tipo de cerveja seja excluído, teremos que verificar quais cervejas desse tipo existem e definir para nulo o atributo **tipo**.

Para estas duas novas funcionalidades, adicionaremos dois métodos à nossa classe **Cervejaria**: **excluirTipo(Long)** e **excluirFabricante(Long)**. Observe na **Listagem 8** que ambos funcionam da mesma forma. Inicialmente, eles excluem o tipo e o fabricante de suas respectivas coleções usando o método **removeIf()**, visto na seção anterior, e, logo em seguida, varrem a coleção de cervejas em busca daquelas que possuem alguma relação com essas duas entidades. Note que para varrer a coleção de cervejas usamos o método **forEach(Consumer<T>)**, passando como parâmetro para ele uma expressão lambda.

Internamente, o método **forEach(Consumer<T>)** irá iterar por cada objeto contido em nossa coleção passando cada elemento encontrado como parâmetro para o método **apply(T)**, da interface **Consumer<T>**. Neste método, podemos executar qualquer operação no elemento, inclusive, alterar seus dados. Entretanto, observe que não criamos uma instância desta

interface. Ao invés disso, passamos uma expressão lambda no seu lugar.

Esse método virtual checa se o tipo ou o fabricante da cerveja é igual ao tipo ou fabricante sendo excluído, definindo o valor nulo em seus respectivos atributos, caso sejam iguais. Lembra-se como este mesmo código era feito com um **for**? Precisávamos, sem dúvida, de um pouco mais de código. Note que agora podemos descartar o uso do **for** para tarefas tão básicas como essa.

Listagem 8. Excluindo Tipos e Fabricantes.

```
01 public void excluirTipo(Long id) {  
02     tipos.removeIf((t) -> (t.getId() != null && t.getId().equals(id)));  
03     cervejas.forEach((c) -> {  
04         if (c.getTipo().getId() == id) {  
05             c.setTipo(null);  
06         }  
07     });  
08 }  
09  
10 public void excluirFabricante(Long id) {  
11     fabricantes.removeIf((f) -> (f.getId() != null && f.getId().equals(id)));  
12     cervejas.forEach((c) -> {  
13         if (c.getFabricante().getId() == id) {  
14             c.setFabricante(null);  
15         }  
16     });  
17 }
```

Nota

A interface **Consumer<T>** foi adicionada apenas no Java 8 e é amplamente usada para trabalhar com coleções. Uma instância desse tipo recebe como entrada um objeto do tipo **T**, no método **apply(T)**, e pode realizar qualquer operação com este objeto, inclusive alterar seus atributos. Este comportamento é diferente da interface **Predicate<T>**, que não deve alterar o estado do objeto passado como parâmetro.

API de Stream

A API de Stream está entre uma das características mais interessantes adicionadas ao *framework* de coleções, e vamos utilizá-la em nossos exemplos para filtrar e executar ações em nossa lista de cervejas. Destacamos que o objetivo desta seção não é analisar a exaustão as streams, mas apenas demonstrar como podemos usar as expressões lambda em conjunto com essa API para facilitar diversas tarefas que envolvem a manipulação de coleções. De fato, a API de Stream torna-se interessante de ser usada devido à inclusão das expressões lambdas. O uso de streams com classes anônimas tornaria seu código extenso, dificultando sua manutenção e compreensão.

Antes de partirmos para a sua utilização na prática, vamos analisar um pouco o conceito de uma stream? De forma simplificada, ela pode ser considerada como uma sequência de elementos oriundos de uma fonte de dados e que suporta operações de agregação. Complicado? Vamos tentar explicá-la quebrando esta definição em partes:

- **Sequência de elementos** – Uma stream provê uma interface para acessar uma sequência de elementos de um mesmo tipo,

entretanto, ela não os armazena internamente. Ao contrário, ela apenas realiza operações sob demanda nestes elementos. Este comportamento é diferente das coleções, que armazenam internamente os elementos;

- **Fonte de dados** – Uma stream consome os dados de uma determinada fonte, como uma coleção, um *Array* ou qualquer tipo de recurso de I/O. Conforme destacado no tópico anterior, streams não armazenam internamente estes dados;

- **Operações agregadas** – Uma stream dá suporte a operações semelhantes àquelas encontradas na linguagem SQL e também em linguagens funcionais, como **filter**, **map**, **reduce**, **find**, **match**, **sorted** e outras. Estas operações são executadas através de um *pipeline*;

- **Pipeline** – Toda operação executada em uma stream sempre retorna outra stream, o que permite que as operações sejam executadas sequencialmente até que um método *terminal* (este conceito será explicado mais à frente) seja invocado;

- **Iteração interna** – Para acessar os elementos de uma coleção, você precisa iterar neles de forma explícita, usando um comando **for** ou **while**. Ao contrário das coleções, uma stream realiza a iteração internamente, sem necessidade de escrever um **for** ou um **while**.

Listagem 9. Usando streams para filtrar elementos.

```
01 public List<Cerveja> obterCervejasPorTipo(Long id){  
02     return cervejas.stream().filter(c -> c.getTipo().getId().equals(id)).  
03         collect(Collectors.toList());  
04 }
```

Para tornar este conceito mais palpável, vamos adicionar duas novas *features* em nosso exemplo que fazem uso desta nova API. A primeira baseia-se em obter uma lista de cervejas de um determinado tipo e está representada na **Listagem 9**, através do método **obterCervejasPorTipo(Long)**. A segunda mudança, representada na **Listagem 10**, através do método **obterMaisRecentes()**, consiste em obter as últimas cinco cervejas cadastradas, ordenadas por data.

Vamos iniciar analisando o código da **Listagem 9**, onde usamos streams para filtrar os tipos de cervejas que gostaríamos de obter. Esta listagem nos apresenta o método **obterCervejasPorTipo(Long)**, que recebe como parâmetro um **Long** informando o identificador do tipo da cerveja e retorna uma lista de cervejas deste tipo.

Neste código invocamos o método **stream()** da coleção de cervejas para obter uma instância de **java.util.Stream**. Com uma stream em mãos, podemos executar as operações que citamos anteriormente! Por exemplo, vamos executar o método **filter(Predicate<T>)**, que recebe como parâmetro uma instância de **Predicate<T>**, e cujo objetivo é filtrar quais itens da coleção nós queremos manter no resultado final.

Internamente, o método **filter(Predicate<T>)** irá iterar em cada elemento da coleção, passando-o como parâmetro para o método

test(T) de Predicate<T>. Nesse método, devemos retornar **true** caso queiramos que o elemento seja mantido no resultado final e **false** caso contrário.

Entretanto, observe que não passamos uma instância desta interface como parâmetro. No lugar dessa instância, passamos uma lambda que recebe como parâmetro o tipo da cerveja e retorna **true** caso o identificador do tipo passado como parâmetro seja igual ao identificador do tipo da cerveja contida na stream.

Para finalizar o uso desta stream, e solicitar um resultado dela, precisamos executar algum método *terminal*. Métodos terminais são aqueles que finalizam a stream, retornando um resultado, ao mesmo tempo em que tornam essa stream indisponível para uso futuro. No nosso exemplo, o método terminal que chamamos é o **collect(Collector)**. Observe que este método recebe como parâmetro uma instância de **java.util.stream.Collector**, que é conhecido como coletor.

Neste caso em específico, este coletor funciona apenas como um container no qual o resultado das operações que executamos na stream será armazenado. Para o nosso exemplo, usamos o método **Collectors.toList()** para informar que o resultado da stream deve ser armazenado em uma lista. Destacamos, contudo, que os coletores podem ser usados para os mais diversos tipos de situações e para um melhor entendimento indicamos a leitura do artigo **Reduction**, disponível na seção **Links**.

O segundo método que analisaremos será o **obterMaisRecentes()**, apresentado na **Listagem 10**. Iniciamos este método obtendo uma instância de **java.util.Stream** e, logo depois, executamos o método **limit(long)**, que irá limitar o número de resultados retornados por essa stream.

Em seguida, executamos o método **sorted(Comparator<T>)**, cuja função é ordenar o resultado da stream usando uma instância de **Comparator<T>** como base. A interface **Comparator<T>** continua funcionando da mesma forma como nas versões anteriores do Java: contém apenas o método **compare(T,T)**, que recebe dois parâmetros de um mesmo tipo e no qual você deve retornar -1 caso o primeiro argumento seja inferior ao primeiro, 0 caso sejam iguais e 1 caso o primeiro argumento seja superior ao segundo.

Listagem 10. Usando streams para filtrar, limitar e ordenar elementos.

```
01 public List<Cerveja> obterMaisRecentes(){  
02     return cervejas  
03         .stream()  
04         .sorted((c1, c2) -> c1.getData().compareTo(c2.getData()))  
05         .limit(5)  
06         .collect(Collectors.toList());  
07 }
```

Em nosso exemplo, usamos o método **compareTo()** de **java.util.Date**, que funciona exatamente igual ao método **compare()** de **Comparator**, para comparar as datas dos objetos. Finalizamos este método obtendo o resultado que desejamos da coleção, usando o método terminal **collect()** da mesma maneira como fizemos anteriormente.

Como aproveitar o máximo das Expressões Lambda em Java

Referência a métodos

Já entendemos como as expressões lambda podem ajudar bastante a tornar seu código mais conciso. Mas, e se o código que queremos criar para resolver um problema já existe e está em uma classe de nosso projeto ou nas bibliotecas nativas do Java? Precisamos criar um método virtual e fazer a chamada a este método existente a partir desta lambda? Ou podemos simplesmente tratar um método de uma classe como um método anônimo? A resposta é sim, nós podemos usar um método já definido em uma classe como uma expressão lambda e isto é possível devido à referência a métodos.

Para tornar esta ideia mais clara, incluiremos a possibilidade de o usuário do nosso sistema pedir para imprimir no console o nome de todas as cervejas cadastradas. Esta funcionalidade está representada na [Listagem 11](#), através do método `imprimir()`. Nesta listagem, observe como passamos o método `println(String)` de `System.out` como uma referência para o método `forEach(Consumer<T>)`.

Listagem 11. Imprimir o nome das cervejas no console.

```
01 public void imprimir() {  
02     cervejas.forEach(System.out::println);  
03 }
```

A sintaxe para passar um método como referência é simples e consiste em usar a notação `Classe::nomeMetodo` ou `Objeto::nomeMetodo`. Em nosso exemplo, usamos `System.out::println` para informar que o método `println()`, contido no objeto `out`, dentro da classe `System`, será passado como referência para o método `forEach()`. Internamente, o método `forEach()` irá iterar na lista de cervejas e passar cada cerveja como parâmetro para o método `println()`. O resultado gerado por este método é bastante intuitivo: será impresso no console o texto retornado pelo método `toString()` da nossa classe `Cerveja`.

Note que um método passado como referência funciona exatamente da mesma forma como funciona uma expressão lambda. De forma ilustrativa, é como se o método fosse “arrancado” da classe e transformado em uma expressão lambda. Neste contexto, um método passado por referência “ganha” de presente todas as características que estamos discutindo até este ponto do artigo.

Referência a construtores

Uma referência a construtores funciona igual a uma referência a métodos, diferenciando-se apenas no uso da expressão `new` no lugar do nome do método. Para analisarmos esta característica, vamos criar uma nova funcionalidade para o nosso sistema. Esta consiste em permitir o cadastro de um lote inteiro de fabricantes, bastando, para isso, passar uma lista de nomes para o nosso sistema. Feito isso, nosso sistema irá iterar nessa lista de nomes, criar um fabricante com este nome e adicioná-lo à lista de fabricantes.

A primeira modificação que precisamos fazer no projeto é adicionar um novo construtor para a classe `Fabricante`. Este construtor irá receber como parâmetro uma `String` contendo o nome do fabricante e definirá o atributo `nome` do objeto com este valor, conforme a [Listagem 12](#). A segunda modificação necessária é a adição de um método chamado `cadastrarFabricantes(List<String>)`, que irá receber como parâmetro uma lista de nomes. Dentro deste método, iremos usar mais uma vez as streams para facilitar nosso trabalho.

Iniciaremos invocando o método `stream()` para obter uma instância de `java.util.Stream` e, logo após, invocaremos o método `map(Function<T, R>)`, passando para ele uma referência ao construtor da classe `Fabricante`.

A interface `Function<T, R>` foi incluída a partir do Java 8 e possui apenas o método `apply(T)`, responsável por retornar um objeto do tipo `R`. A ideia desta interface é que instâncias dela trabalhem de forma similar a uma função matemática: ela recebe um valor de entrada e retorna um outro valor como saída. Em outras palavras, uma instância de `Function<T, R>` receberá uma instância do tipo `T`, através do método `apply(T)` e retornará um objeto do tipo `R`.

Listagem 12. Novo construtor na classe Fabricante.

```
01 public class Fabricante {  
02     private Long id;  
03     private String nome;  
04  
05     public Fabricante() {}  
06  
07     public Fabricante(String nome) {  
08         this.nome = nome;  
09     }  
10 }
```

Listagem 13. Adicionando método para cadastrar fabricantes em lotes.

```
01 public void cadastrarFabricantes(List<String> nomes) {  
02     nomes.stream().map(Fabricante::new).forEach((f) -> {  
03         fabricantes.add(f);  
04     });  
05 }
```

Em nosso caso, ao invés de criarmos uma lambda ou instanciar um objeto do tipo `Function<T, R>`, passamos o construtor da classe `Fabricante` no lugar (veja a [Listagem 13](#)). Em primeiro lugar, isto só é possível porque criamos um construtor que recebe como parâmetro uma `String` e também porque todo construtor retorna um objeto do tipo da classe que ele está inserido. Em segundo lugar, porque quando “arrancamos” o construtor dessa classe e o tratamos como uma lambda, ele funciona de forma similar ao método `R apply(T)` de `Function<T, R>` e, portanto, pode ser usado no lugar de uma instância desta interface.

Criando uma Interface Funcional

Nas seções anteriores discutimos algumas das características das interfaces funcionais, entretanto, ainda não criamos nossa própria interface funcional em nosso projeto. Para demonstrar

isto, vamos melhorar ainda mais nosso sistema, adicionando a ele uma nova funcionalidade que o permitirá se integrar com sistemas de terceiros, recebendo dados oriundos deles e cadastrando-os internamente.

Contudo, os dados oriundos desses aplicativos externos poderão vir nos mais diferentes formatos, podendo ser uma cadeia de `String`, um objeto `java.util.Map` ou até mesmo um objeto `org.json.JSONObject`. Para dar essa flexibilidade ao nosso sistema, criaremos uma interface chamada `Conversor<T>` que contará apenas com o método `converter(T)`, que recebe como parâmetro um objeto do tipo `T` e retorna um objeto do tipo `Cerveja`.

O objetivo de uma instância de `Conversor<T>` é, através do método `converter(T)`, criar um objeto do tipo `Cerveja` a partir dos dados oriundos de alguma fonte de dados. Dito isso, nosso primeiro passo consiste em criar essa interface, conforme o código da **Listagem 14**. Observe que trata-se de uma interface comum, mas que contém apenas um método, e é anotada com `@FunctionalInterface`. Esta anotação não é obrigatória, mas é aconselhada, pois força o compilador, em tempo de compilação, a checar se sua interface realmente contém apenas um método. Caso contenha mais de um ou nenhum, a compilação falhará.

Em seguida, precisamos criar um método, na classe `Cervejaria`, que receberá como parâmetro uma coleção de objetos e um conversor informando como converter cada objeto desta coleção para um objeto do tipo `Cerveja`. Este método está ilustrado na **Listagem 15** e é chamado de `processar(List<T>, Conversor<T>)`.

Listagem 14. Código da interface Conversor.

```
01 @FunctionalInterface  
02 public interface Conversor<T> {  
03     Cerveja converter(T t);  
04 }
```

Listagem 15. Método processar na classe Cervejaria.

```
01 public <T> void processar(List<T> dados, Conversor<T> conversor) {  
02     dados  
03         .stream()  
04         .map(conversor::converter)  
05         .forEach(c -> {  
06             cervejas.add(c);  
07         });  
08 }
```

O trabalho realizado por este método é bastante simples e faz uso da API de Streams para torná-lo mais conciso. Observe que, mais uma vez, fazemos uso dos métodos `stream()`, `map()` e `forEach()`. Para o método `map()`, apenas passamos uma referência ao método `converter()` da instância de `Conversor<T>` passada como parâmetro. Em seguida, dentro da lambda passada para o método `forEach()`, incluímos a cerveja criada pelo conversor na coleção de cervejas.

Finalizando, vamos analisar o código da **Listagem 16**, onde fazemos uso deste novo método para passar ao nosso aplicativo

uma lista de objetos do tipo `org.json.JSONObject`, contendo informações sobre cervejas, e um conversor que informa como transformar um objeto desse tipo em um objeto do tipo `Cerveja`. Um objeto do tipo `JSONObject` funciona basicamente como um `array` associativo que relaciona uma chave a um valor. Escolhemos esse formato para ilustrar esse exemplo, pois o JSON vem sendo cada vez mais utilizado como padrão para a troca de dados entre sistemas.

Iniciamos esta listagem criando uma lista contendo um único objeto do tipo `JSONObject` para funcionar apenas como massa de testes. Em seguida, chamamos o método `processar()` passando a lista de objetos e uma lambda, no lugar de uma instância de `Conversor<T>`. Essa lambda realiza o trabalho de conversão de um objeto `JSONObject` para um objeto `Cerveja`, retornando-o logo em seguida.

Nota

O JSON é o padrão de troca de mensagens mais adotado nos dias atuais para a troca de informações com APIs que seguem os conceitos RESTful. Acrônimo para JavaScript Object Notation, é uma notação mais leve e simples, sendo ideal também para a troca de dados com dispositivos móveis. Para mais informações sobre RESTful e JSON, visite a seção [Links](#).

Listagem 16. Exemplo de uso do método processar.

```
01 List<JSONObject> lista = new ArrayList<>();  
02 JSONObject j1 = new JSONObject();  
03 j1.put("nome", "J1");  
04 lista.add(j1);  
05  
06 Cervejaria.getInstancia().processar(lista, (json) -> {  
07     Cerveja c = new Cerveja();  
08     c.setNome(json.getString("nome"));  
09  
10     return c;  
11 });
```

Threads

Outra área do Java que se beneficiou bastante da inclusão das expressões lambda é a API de `Threads`. Para demonstrar estes benefícios, vamos incrementar mais uma vez nossa aplicação. Para isso, consideraremos que a qualquer momento podemos solicitar ao sistema que ele gere um arquivo CSV contendo os dados de todas as cervejas do nosso sistema.

Nota

A sigla CSV vem do inglês e significa comma-separated values (valores separados por vírgula, em português). Um arquivo CSV nada mais é do que um arquivo texto, onde cada linha é um registro. Ademais, os valores, ou campos, de cada registro, são separados por vírgula, e a primeira linha deste arquivo informa o nome dos campos dos registros. Para mais informações sobre o formato CSV.

Observe que esta é uma tarefa da qual não necessitamos aguardar uma resposta imediata e que pode ser realizada em *background*, sem prejuízo para o sistema ou para o usuário. Mesmo que ocorra algum erro no processamento, isto não acarretará em problemas,

pois o usuário pode realizar essa solicitação novamente em um momento futuro.

O código para realizar esta tarefa está descrito na **Listagem 17**, no qual instanciamos uma **Thread** e passamos como parâmetro para o seu construtor uma lambda, ao invés de criar uma instância de **Runnable**. Isto é possível porque **Runnable** contém apenas um método e, portanto, é uma interface funcional, o que nos permite passar expressões lambda no lugar de objetos deste tipo.

Nesta listagem, note que já estamos adotando alguns dos métodos que discutimos nas seções anteriores, como é o caso do método **forEach()**, utilizado para iterar na coleção. Como parâmetro para este método, passamos uma expressão lambda que cria uma instância de **java.io.PrintWriter** e cria um arquivo com o nome *cervejas.csv*, escrevendo nele os dados de cada cerveja no formato CSV.

Listagem 17. Uso de expressões lambda com Threads.

```
01 public void gerarCSV() {  
02     new Thread(() -> {  
03         try {  
04             final PrintWriter writer = new PrintWriter("cervejas.csv", "UTF-8");  
05             cervejas.forEach(c -> {  
06                 writer.println(c.getId() + "\n");  
07                 writer.println(c.getNome() + "\n");  
08                 writer.println(c.getFabricante().getNome() + "\n");  
09                 writer.println(c.getTipo().getNome() + "\n");  
10                 writer.println(c.getData() + "\n");  
11             });  
12             writer.close();  
13         } catch (FileNotFoundException e) {  
14         } catch (UnsupportedEncodingException e) {  
15         }  
16     }).start();  
17 }
```

Swing

O Swing é um toolkit para desenvolvimento de interfaces gráficas para desktops, adicionado ao Java a partir da versão 1.2, que também se beneficiou das características das expressões lambda. Na maior parte dos casos, esse benefício é proveniente do uso dos métodos virtuais no lugar de instâncias de **ActionListener**. De fato, o uso de lambdas nesse contexto é fortemente indicado, pois os códigos de tratamento provenientes de ações do usuário com uma interface gráfica são, na maioria dos casos, específicos ao contexto onde esta ação foi executada.

Para ilustrar o que foi mencionado, considere uma tela de cadastro que possui um botão chamado *Salvar*. Note que, em geral, o código de tratamento associado ao clique neste botão faz sentido apenas nesta tela, pois trata-se da única tela do sistema onde esta ação está disponível. Assim, não faz muito sentido, e nem é uma prática comum, criar classes que herdam de **javax.swing.JButton** que especializam o comportamento deste botão para tratar a ação de salvar uma entidade qualquer.

Para demonstrar o uso das expressões lambda com o Swing, vamos modelar uma janela de cadastro de cervejas, na qual teremos todos os campos da entidade **Cerveja**, criada nas seções anteriores, além de um botão salvar, que irá armazenar esta cerveja na nossa coleção.

Parte do código que monta a tela e trata o clique no botão está na **Listagem 18**. Entretanto, este código não está completo, pois omitimos algumas partes, já que são trechos que servem apenas para ajustar a posição e o tamanho dos campos na janela. Na verdade, o código que mais nos interessa nesta listagem é o que define uma expressão lambda para tratar o clique no botão salvar. Neste trecho de código, observe que, ao invés de instanciarmos um objeto do tipo **ActionListener**, usamos um método virtual em seu lugar. Lembre-se que isto é possível apenas porque a interface **ActionListener** é uma interface funcional que contém apenas o método **actionPerformed()**.

Listagem 18. Tela de Cadastro de Cervejas em Swing.

```
01 salvar.addActionListener(e -> {  
02     Cerveja cerveja = new Cerveja();  
03     cerveja.setNome(nome.getText());  
04     cerveja.setNota(Integer.valueOf(notas.getText()));  
05     cerveja.setFabricante((Fabricante)fabricante.getModel().getSelectedItem());  
06     cerveja.setTipo((Tipo)tipo.getModel().getSelectedItem());  
07     Cervejaria.getInstance().salvar(cerveja);  
08 });
```

Analisando este código, notamos que trata-se de um trecho bastante simples, onde apenas criamos um objeto do tipo **Cerveja** e definimos os valores dos seus atributos conforme os dados informados pelo usuário em nossa tela. Apenas para tornar ainda mais claro o conceito de interfaces funcionais, note que não é possível passar uma expressão lambda para o método **addMouseListener()**, pois ele aguarda uma instância de **MouseListener**, uma interface que contém mais de um método e, portanto, não pode ser substituído por um método virtual.

Embora este seja o exemplo mais emblemático e corriqueiro do uso de expressões lambda no Swing, note que lhe poupará muitas linhas de código, pois clique em botões é uma atividade bastante comum em interfaces para desktop.

O Java 8 já é uma realidade! Como consequência disso, temos que estar preparados para nos adaptar a importantes mudanças introduzidas tanto na linguagem como no *core* da máquina virtual. Entre essas mudanças estão as expressões lambda, que adicionam conceitos de linguagens funcionais e híbridas, visando tornar nosso código mais limpo e conciso. Embora exista um tempo natural de adaptação quando uma nova versão do Java é lançada, acreditamos ser de suma importância estar preparado para esta grande mudança, pois há uma tendência natural de que outras bibliotecas do Java se adaptem para tirar o máximo proveito desta nova *feature*.

Esta tendência, aliás, já pode ser vista na própria versão 8, com o lançamento da API de Stream. Destacamos que essa API tem muito mais a oferecer ao seu projeto, indo além daquilo que apre-

sentamos nesse artigo. Portanto, sugerimos fortemente que você se aprofunde mais nesse assunto, pois ele pode ser fundamental para tornar seu código mais simples, uma vez que tarefas que antes requeriam muitas linhas de código, agora podem ser escritas com menor esforço.

Apesar de ter sido apresentado neste artigo um único cenário de uso das expressões lambda, cabe destacar que elas podem ser utilizadas nas mais diversas situações, não se limitando ao contexto do exemplo apresentado. Lembre-se também que, assim como ocorreu com a API de Coleções, que foi modificada para dar suporte a essa nova característica, seus aplicativos mais antigos, assim que migrarem para o Java 8, também poderão se beneficiar das expressões lambda, bastando, para isso, usar os métodos *default*.

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback
Ajude-nos a manter a qualidade da revista!



Autor



Marlon Silva Carvalho

marlon.carvalho@gmail.com

Programador poliglota há 20 anos, é fã de programação para dispositivos móveis e para a web. Tem na tecnologia sua paixão, na família sua motivação e na degustação de cervejas artesanais seu hobby predileto. Também é Bacharel pela Universidade Católica do Salvador, Pós-graduado em Sistemas Distribuídos pela Universidade Federal da Bahia e Organizer do Google Developers Group de Salvador. Siga seu twitter em @marlonscarvalho ou acesse sua página pessoal em <http://marlon.silvacarvalho.net/>.



Links:

Exemplo no GitHub.

<https://github.com/marloncarvalho/javamagazine-lambda>

Artigo sobre Reduction.

<https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html>

JSON – Javascript Object Notation.

<http://json.org/>

CURSOS ONLINE

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



CONHEÇA OS CURSOS MAIS RECENTES:

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

Para mais informações :

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

(21) 3382-5038

Como escolher o algoritmo mais eficiente com OptaPlanner – Parte 2

Escolha o algoritmo de otimização mais eficiente para cada caso de uso

ESTE ARTIGO FAZ PARTE DE UM CURSO

A primeira parte do artigo introduziu ao leitor o universo dos problemas de planejamento e mostrou o quanto difícil é resolvê-los. Vimos que mesmo problemas de dimensões pequenas podem gerar uma quantidade gigantesca de combinações possíveis, tornando-se inviável ou impossível testar todas as possibilidades até encontrar a solução ótima.

Nesse contexto, o OptaPlanner foi apresentado como uma alternativa aos métodos de busca exaustiva para resolver problemas de planejamento (como o *bin packing*). O OptaPlanner reúne um conjunto de algoritmos de otimização (heurísticas e meta-heurísticas) que são empregados para percorrer de forma eficiente o altíssimo número de soluções possíveis e retornar um resultado bom em tempo viável.

Para explorar suas funcionalidades e comprovar sua eficiência, demos início na primeira parte do artigo à implementação de um exemplo de código que utiliza o OptaPlanner para resolver um problema de planejamento baseado em um cenário do mundo real. O cenário descreve um problema enfrentado por uma empresa de

Fique por dentro

O OptaPlanner é um framework que auxilia na resolução de problemas de planejamento. Ele reúne um conjunto de heurísticas e meta-heurísticas que podem ser aplicadas para encontrar, de forma eficiente, a melhor solução para um problema dentro de um período de tempo aceitável.

Nesta segunda e última parte do artigo daremos continuidade à implementação do exemplo que tem como objetivo resolver um caso de uso de uma empresa de turismo. Conheceremos o funcionamento de alguns dos algoritmos de otimização implementados no OptaPlanner e como utilizar a ferramenta de benchmarking para escolher a configuração mais eficiente para cada caso de uso.

turismo para alocar eficientemente seus passageiros em sua frota de veículos. Até o momento passamos pela definição das classes do modelo de domínio, identificamos as restrições de negócio e implementamos as regras para cálculo do *score* utilizando o Drools.

Nesta segunda parte daremos continuidade ao exemplo e finalmente veremos o OptaPlanner em funcionamento. Além disso, o artigo abordará conceitualmente alguns dos algoritmos de otimização implementados na solução da Red Hat e mostrará como utilizar a ferramenta de benchmarking para comparar diferentes algoritmos/configurações e escolher o mais eficiente para o nosso caso de uso.

Configuração do solver

Dando continuidade à implementação do exemplo, iremos agora configurar o componente de mais alto nível da API do OptaPlanner, com o qual aplicações interagem para resolver problemas de planejamento: o **solver**.

A configuração do solver pode ser feita de duas maneiras: estática (via arquivo XML) ou dinâmica (via API Java). A segunda opção é muito útil quando alguns dos parâmetros de configuração (como tempo máximo de execução, etc.) puderem ser informados dinamicamente por usuários ou aplicações clientes. Em nosso exemplo, utilizaremos a opção de configuração via arquivo XML, uma vez que todos os parâmetros serão pré-definidos. Sendo assim, crie um novo arquivo na pasta `src/main/resources/solver` do projeto chamado `solverConfig.xml`. A **Listagem 1** mostra o código completo do arquivo.

O conteúdo do arquivo de configuração do solver está dividido em três partes (observe os comentários deixados na listagem): modelo de domínio, score e algoritmos de otimização.

Na parte referente ao modelo de domínio, temos de informar o nome qualificado das classes *planning entity* e *planning solution* do problema.

No segundo bloco de elementos, definimos alguns aspectos referentes ao cálculo do score, como o tipo de score sendo usado e o caminho para o arquivo de regras do Drools no *classpath* da aplicação. Se o leitor optar por não escrever as regras no Drools, mas sim por utilizar uma das implementações em Java, informe o nome qualificado da classe através dos elementos `<easyScoreCalculatorClass>` ou `<incrementalScoreCalculatorClass>` (dependendo de qual das duas interfaces decidir implementar). O último elemento do bloco, `<initializingScoreTrend>`, especifica como o score evolui à medida que as *planning variables* do problema vão sendo inicializadas com os *planning values*. Essa informação é útil para que alguns tipos de algoritmos possam ser mais performáticos. Como todas as restrições do nosso problema de planejamento (e da maioria dos casos de uso) são negativas, então o valor correto a ser informado é **ONLY_DOWN**.

A terceira e última parte é certamente a mais importante do arquivo, pois refere-se à configuração dos algoritmos de otimização. Neste momento, no entanto, não vamos nos preocupar com otimização, mas sim em conseguirmos executar nosso exemplo o quanto antes e da forma mais simples possível. Por essa razão, vamos utilizar um algoritmo de busca exaustiva, o **Brute Force**. Ao fazermos isso, não teremos muito ganho ao utilizar o OptaPlanner. No entanto, conseguiremos ver nosso exemplo funcionando e atingindo um resultado previsível. Ao usar o algoritmo de força bruta, o leitor também vai poder observar a degradação do tempo de resposta se adicionar novos valores ao problema (não deixe de fazer esse teste). Mas não se preocupe, pois na sequência veremos como selecionar outros algoritmos para tornar nosso exemplo mais eficiente.

Executando a aplicação

Neste momento já temos quase tudo preparado para resolver o problema de alocação de passageiros da empresa de turismo.

Precisamos apenas de uma instância do problema a ser resolvido (*data set* contendo grupos de passageiros e veículos disponíveis) e também produzir algumas linhas de código para interagir com o solver. Vamos começar pelo segundo item.

Deste modo, crie uma nova classe chamada **Alocador**. Essa classe deve disponibilizar um método público que recebe como argumento um objeto do tipo **AlocacaoPassageiros** (instância do problema). Veja o código-fonte completo da classe na **Listagem 2**.

Listagem 1. Arquivo de configuração do solver (`solverConfig.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>
<solver>
    <!-- Modelo de domínio -->
    <solutionClass>br.com.devmedia.agenciaturismo.planejamento.
        AlocacaoPassageiros</solutionClass>
    <entityClass>br.com.devmedia.agenciaturismo.dominio.Grupo</entityClass>

    <!-- Score -->
    <scoreDirectorFactory>
        <scoreDefinitionType>HARD_MEDIUM_SOFT</scoreDefinitionType>
        <scoreDrl>solver/allocacaoPassageirosScoreRules.drl</scoreDrl>
        <initializingScoreTrend>ONLY_DOWN</initializingScoreTrend>
    </scoreDirectorFactory>

    <!-- Algoritmos de otimização -->
    <exhaustiveSearch>
        <exhaustiveSearchType>BRUTE_FORCE</exhaustiveSearchType>
    </exhaustiveSearch>
</solver>
```

Listagem 2. Código-fonte da classe Alocador.

```
package br.com.devmedia.agenciaturismo.planejamento;

import org.optaplanner.core.api.solver.Solver;
import org.optaplanner.core.api.solver.SolverFactory;

public class Alocador {

    public AlocacaoPassageiros resolver(AlocacaoPassageiros problema) {
        SolverFactory solverFactory = SolverFactory.
            createFromXmlResource("solver/solverConfig.xml");
        Solver solver = solverFactory.buildSolver();
        solver.solve(problema);
        return (AlocacaoPassageiros) solver.getBestSolution();
    }
}
```

O método principal da classe **Alocador** cria um objeto **Solver** usando a classe **SolverFactory**. Essa *factory*, por sua vez, recebe o XML de configuração que definimos no passo anterior como argumento na sua construção. A resolução do problema ocorre de fato quando o método **solve(...)** é invocado no objeto **solver**. Após o processamento ser concluído, a melhor solução encontrada é obtida através do método **getBestSolution()**, também no objeto **solver**.

Em seguida, vamos criar uma nova classe chamada **Main**. Essa classe será o ponto de entrada para nossa aplicação. Nela iremos construir uma instância de dimensão pequena do problema, com valores capazes de produzir uma solução viável e com score previsível. Além disso, a classe irá imprimir no console o resultado obtido pelo OptaPlanner.

Como escolher o algoritmo mais eficiente com OptaPlanner – Parte 2

Veja o código completo na **Listagem 3**.

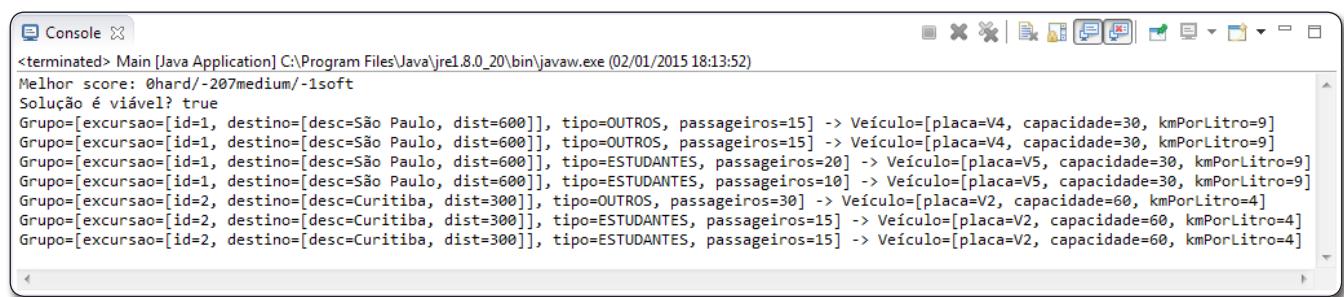
Finalmente já podemos executar a aplicação. Levará apenas alguns segundos até que o resultado, conforme mostrado na **Figura 1**, seja impresso no console. Se tudo estiver correto, é esperado que a solução encontrada para o problema tenha um score igual a “**0hard/-207medium/-1soft**”. Além do score, o leitor poderá observar no console quais grupos de passageiros foram atribuídos a quais veículos (não se esqueça de sobrescrever o método **toString()** nas classes do modelo de domínio para que os objetos sejam impressos de maneira inteligível).

O formato como o objeto score é impresso nos permite visualizar os três níveis de pontuação (*hard*, *medium* e *soft*) que o compõe. No primeiro nível, o valor “**0hard**” indica que nenhuma das restrições invioláveis do problema foi quebrada. Em outras palavras, isso significa que na solução encontrada pelo OptaPlanner não há nenhum veículo transportando grupos de excursões distintas,

nem passageiros excedentes (todos os integrantes dos grupos viajarão juntos). Isso faz com que a solução seja considerada viável. No segundo nível do score, o valor “**-207medium**” indica que serão consumidos 207 litros de combustível no total para realizar todas as excursões. Por fim, o valor “**-1soft**” no último nível do score indica que na solução encontrada haverá um veículo transportando grupos de estudantes misturados com grupos de outros perfis de passageiros. Todas essas informações podem ser conferidas e validadas no resultado impresso pela aplicação.

Para esse mesmo problema existem outras soluções também viáveis. No entanto, a solução apresentada é a que gera o menor consumo de combustível e que consegue alocar a maior quantidade de grupos de estudantes em veículos exclusivos.

Em uma aplicação do mundo real, os dados que utilizamos para construir o problema seriam possivelmente lidos de um banco de dados e salvos após o problema ter sido resolvido.



```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (02/01/2015 18:13:52)
Melhor score: 0hard/-207medium/-1soft
Solução é viável? true
Grupo=[excursao=[id=1, destino=[desc=São Paulo, dist=600]], tipo=OUTROS, passageiros=15] -> Veículo=[placa=V4, capacidade=30, kmPorLitro=9]
Grupo=[excursao=[id=1, destino=[desc=São Paulo, dist=600]], tipo=OUTROS, passageiros=15] -> Veículo=[placa=V4, capacidade=30, kmPorLitro=9]
Grupo=[excursao=[id=1, destino=[desc=São Paulo, dist=600]], tipo=ESTUDANTES, passageiros=20] -> Veículo=[placa=V5, capacidade=30, kmPorLitro=9]
Grupo=[excursao=[id=1, destino=[desc=São Paulo, dist=600]], tipo=ESTUDANTES, passageiros=10] -> Veículo=[placa=V5, capacidade=30, kmPorLitro=9]
Grupo=[excursao=[id=2, destino=[desc=Curitiba, dist=300]], tipo=OUTROS, passageiros=30] -> Veículo=[placa=V2, capacidade=60, kmPorLitro=4]
Grupo=[excursao=[id=2, destino=[desc=Curitiba, dist=300]], tipo=ESTUDANTES, passageiros=15] -> Veículo=[placa=V2, capacidade=60, kmPorLitro=4]
Grupo=[excursao=[id=2, destino=[desc=Curitiba, dist=300]], tipo=ESTUDANTES, passageiros=15] -> Veículo=[placa=V2, capacidade=60, kmPorLitro=4]
```

Figura 1. Output produzido no console pela aplicação.

Listagem 3. Código-fonte da classe Main do exemplo.

```
package br.com.devmedia.agenciaturismo;

import static br.com.devmedia.agenciaturismo.dominio.Grupo.TipoGrupo.ESTUDANTES;
import static br.com.devmedia.agenciaturismo.dominio.Grupo.TipoGrupo.OUTROS;
import static java.util.Arrays.asList;

import java.util.Date;

import br.com.devmedia.agenciaturismo.dominio.Destino;
import br.com.devmedia.agenciaturismo.dominio.Excursao;
import br.com.devmedia.agenciaturismo.dominio.Grupo;
import br.com.devmedia.agenciaturismo.dominio.Veiculo;
import br.com.devmedia.agenciaturismo.planejamento.AlocacaoPassageiros;
import br.com.devmedia.agenciaturismo.planejamento.Alocador;

public class Main {

    public static void main(String[] args) {
        AlocacaoPassageiros problema = construirProblema();
        Alocador alocador = new Alocador();
        AlocacaoPassageiros solucao = alocador.resolve(problema);
        imprimir(solucao);
    }

    private static AlocacaoPassageiros construirProblema() {
        Destino d1 = new Destino("São Paulo", 600);
        Destino d2 = new Destino("Curitiba", 300);

        Excursao ex1 = new Excursao(1, new Date(), d1);
        Excursao ex2 = new Excursao(2, new Date(), d2);

        Grupo g01 = new Grupo(ex1, OUTROS, 15);
        Grupo g02 = new Grupo(ex1, OUTROS, 15);
        Grupo g03 = new Grupo(ex1, ESTUDANTES, 20);
        Grupo g04 = new Grupo(ex1, ESTUDANTES, 10);
        Grupo g05 = new Grupo(ex2, OUTROS, 30);
        Grupo g06 = new Grupo(ex2, ESTUDANTES, 15);
        Grupo g07 = new Grupo(ex2, ESTUDANTES, 15);

        Veiculo v1 = new Veiculo("V1", 60, 3);
        Veiculo v2 = new Veiculo("V2", 60, 4);
        Veiculo v3 = new Veiculo("V3", 30, 7);
        Veiculo v4 = new Veiculo("V4", 30, 9);
        Veiculo v5 = new Veiculo("V5", 30, 9);

        return new AlocacaoPassageiros(asList(g01, g02, g03, g04, g05, g06, g07),
            asList(v1, v2, v3, v4, v5));
    }

    private static void imprimir(AlocacaoPassageiros solucao) {
        System.out.println("Melhor score: " + solucao.getScore());
        System.out.println("Solução é viável? " + solucao.getScore().isFeasible());
        solucao.getGrupos().stream().forEach(Main::imprimirGrupo);
    }

    private static void imprimirGrupo(Grupo grupo) {
        System.out.println("Grupo=" + grupo + " -> Veículo=" + grupo.getVeiculo());
    }
}
```

Outra opção seria recebê-los de uma aplicação cliente através de uma chamada remota, por exemplo, e devolver a solução da mesma maneira.

Nessa primeira versão do exemplo começamos com um data set bem pequeno (poucos veículos, poucas excursões e poucos grupos de passageiros), pois estamos usando o algoritmo de força bruta e não queremos esperar eternamente para saber se o nosso código funcionou ou não. Portanto, comece testando o exemplo com os mesmos valores mostrados na **Listagem 3**. Mas, em seguida, experimente outras combinações, de forma que algumas restrições sejam quebradas e observe o resultado.

Algoritmos de otimização

Nosso exemplo está funcionando perfeitamente e trazendo um resultado ótimo para o problema em um tempo muito curto. Mas isso não significa que podemos parar por aqui. No momento estamos empregando o algoritmo força bruta que, conforme sabemos, testa todas as combinações possíveis até encontrar a solução ótima. E, conforme vimos na primeira parte do artigo, algoritmos de busca exaustiva (como o força bruta) não escalam quando aplicados para resolver problemas de planejamento de dimensões maiores, como os encontrados no mundo real.

Não é preciso muito esforço para alcançar o limite de escalabilidade desses algoritmos. A **Figura 2** mostra o resultado de um experimento utilizando o exemplo que desenvolvemos até aqui. O gráfico compara o tempo gasto por duas variantes de algoritmos de busca exaustiva para encontrar a solução ótima para o problema, tendo como *input* data sets de diferentes dimensões. Observe no eixo horizontal que a variação de tamanho dos data sets (quantidade de grupos e veículos) é bem pequena. Entretanto, o efeito no tempo de resposta dos algoritmos é brutal, resultando em uma curva exponencial. Observe que o algoritmo **Brute Force** (barras vermelhas) não demora muito para atingir seu limite. Já o **Branch and Bound (favorite)**, que é uma variação mais inteligente do Brute Force, demora um pouco mais,

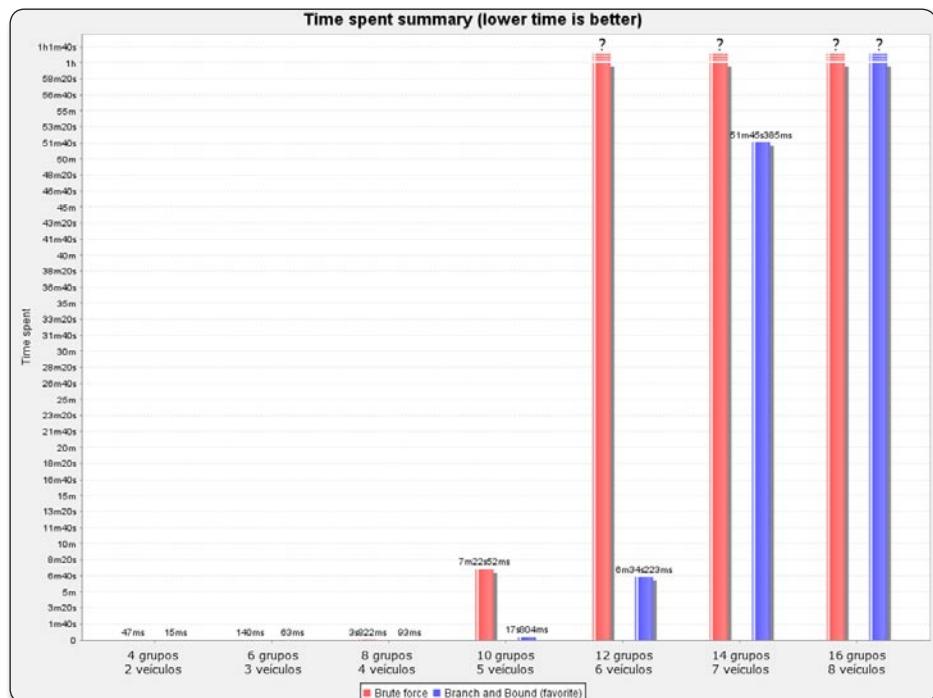


Figura 2. Gráfico mostrando a limitação de escalabilidade dos algoritmos de busca exaustiva

mas mesmo assim atinge o limite de escalabilidade ainda bastante cedo.

Esse experimento mostra que algoritmos de busca exaustiva não são alternativas consideráveis para problemas do mundo real. O que faríamos então se tivéssemos que implementar um algoritmo para resolver esse problema? Uma opção seria limitar o tempo de execução e fazer uma busca sequencial ou randômica, retornando a melhor solução encontrada dentro daquele período. Neste caso, muito provavelmente, o resultado ficaria longe da solução ótima. Agora considere o seguinte: e se selecionássemos uma solução possível aleatoriamente e, a partir daí, fizéssemos pequenas alterações nos valores (mova o grupo A do veículo X para o veículo Y) e, a cada mudança, avalíssemos o resultado? Se a solução piorar (score mais baixo), desfazemos a alteração e tentamos outra. Mas, se a solução for melhorada, a alteração é mantida e o processo é repetido outras vezes enquanto houver tempo disponível.

O passo a passo anterior descreve simplificadamente o funcionamento de uma meta-heurística de busca local chamada **Hill Climbing** que é apenas um dos

algoritmos de otimização que vêm implementados no OptaPlanner e que ajudam a resolver problemas de planejamento de maneira eficiente e escalável.

Embora os algoritmos venham “de graça” no OptaPlanner, desenvolvedores ainda precisam selecionar o mais eficiente para o caso de uso que estão tentando resolver. Infelizmente não existe “o” algoritmo mais eficiente para todos os casos. Às vezes, um determinado algoritmo é o mais eficiente para uma ordem de grandeza X do problema, porém não muito eficiente para outra. Para complicar um pouco mais, cada algoritmo possui diversas propriedades de configuração que podem influenciar bastante no resultado final. O lado positivo é que é muito simples alternar entre os diferentes algoritmos e refiná-las suas configurações. Além disso, o OptaPlanner disponibiliza uma ferramenta de benchmarking que permite comparar diferentes algoritmos/configurações, e assim ajudar a escolher aquele que melhor atende cada caso de uso.

Múltiplos algoritmos de otimização podem ser inclusive combinados para resolver o mesmo problema de planejamento. Nesse caso, o OptaPlanner executa

os algoritmos em fases sequenciais, como em um *pipeline*: o *output* do algoritmo anterior serve de *input* para o algoritmo seguinte. Na prática isso é comum de se acontecer. Uma das razões para isso é que alguns algoritmos (meta-heurísticas de busca local) requerem a construção de uma solução inicial antes que possam ser executados. Outra razão seria para alcançar resultados melhores. No geral, é comum haver uma fase inicial usando uma **heurística de construção** seguida de uma ou mais fases usando **meta-heurísticas**.

As heurísticas de construção são algoritmos especializados em construir um bom ponto de partida. Usá-los de forma isolada pode não ser o suficiente, já que a solução inicial nem sempre será viável e muito menos estará próxima de ser ótima.

Entre as heurísticas de construção disponíveis no OptaPlanner estão o **First Fit** e sua variação geralmente mais efetiva, o **First Fit Decreasing**.

O algoritmo First Fit percorre sequencialmente todas as planning entities do problema, associando cada uma ao melhor planning value disponível no momento. Tomando como cenário o problema bin packing, cada item é atribuído ao primeiro container onde couber, ou o que gerar o menor excedente possível.

O algoritmo First Fit Decreasing tem funcionamento parecido, porém primeiro ele ordena as planning entities de acordo com o grau de dificuldade (itens mais difíceis são alocados primeiro). No problema bin packing é mais difícil alocar os itens maiores nos espaços restantes dos containers.

Uma vez que as planning entities são associadas a um planning value durante a execução de uma heurística de construção, elas não são mais alteradas dentro daquela fase. Usando novamente o problema bin packing como exemplo, uma vez que um item é atribuído a um container, ele não é movido para nenhum outro. Fases subsequentes, no entanto, podem (e irão) fazer movimentações a fim de melhorar a solução.

As heurísticas de construção geralmente são rápidas e a fase correspondente é terminada automaticamente após a solução inicial ter sido construída. Fique atento, pois, conforme veremos na sequência, esse não é o mesmo comportamento esperado para as meta-heurísticas.

As meta-heurísticas (mais especificamente, as meta-heurísticas de busca local) assumem o trabalho iniciado pelas heurísticas de construção, fazendo movimentos/trocadas com o objetivo de melhorar a solução inicial. Elas operam sobre um estado corrente único e fazem movimentos apenas para os vizinhos desse estado. O algoritmo Hill Climbing, descrito anteriormente, é talvez o mais simples exemplo. Quase todas as demais meta-heurísticas são combinações mais elaboradas de Hill Climbing com busca randômica. Além do próprio Hill Climbing, o OptaPlanner fornece implementações de outros algoritmos de busca local, entre eles o **Tabu Search** e o **Late Acceptance**.

O algoritmo Tabu Search difere do Hill Climbing no sentido de que ele mantém um histórico de tamanho fixo dos objetos (planning entities, planning values, etc.) relacionados aos últimos movimentos efetuados. Novos movimentos que envolvem qual-

quer objeto contido nesse histórico são recusados. É como se ele se lembrasse por onde já passou e não usasse o mesmo caminho duas vezes. Essa estratégia visa evitar um dos problemas do Hill Climbing: ficar preso em um máximo local.

Nota

O OptaPlanner atualmente fornece diversos outros algoritmos além dos mencionados neste artigo. Consulte a documentação para conhecer detalhes sobre eles. Novos algoritmos, como meta-heurísticas evolutivas (algoritmos genéticos, etc.) e hiper-heurísticas, possivelmente serão incluídos em versões futuras.

Um máximo local acontece quando todos os movimentos possíveis levam a uma piora na solução, o que faz com que o algoritmo fique preso numa solução intermediária e nunca alcance o máximo global (a tão desejada solução ótima). Uma boa analogia para entender esse problema é considerar um alpinista tentando escalar o Everest sem GPS e no meio de um nevoeiro. Se ele subir por engano uma das montanhas menores ao redor, ele vai chegar ao topo errado e nunca vai sequer saber que não atingiu o topo mais alto.

Para conseguir escapar de um máximo local e, assim, explorar outras áreas no espaço de busca, o algoritmo Tabu Search pode aceitar, inclusive, movimentos que pioram a solução corrente.

O algoritmo Late Acceptance, por sua vez, mantém um histórico de tamanho fixo das melhores soluções obtidas com os últimos movimentos. Ele aceita qualquer novo movimento cuja solução resultante não seja pior que a última solução nesse histórico.

Diferentemente das heurísticas de construção, as meta-heurísticas precisam ser ditas explicitamente quando terminar, por exemplo, após um determinado período de tempo ou até que um score mínimo seja atingido. Caso contrário, elas continuarão executando indefinidamente.

Escolhendo a melhor configuração com ajuda de benchmarkings

Vamos agora melhorar a escalabilidade da nossa aplicação para suportar dimensões maiores do problema de alocação de passageiros. Com o auxílio da ferramenta de benchmarking incluída no OptaPlanner, iremos comparar o desempenho de alguns algoritmos de otimização e descobrir qual irá se mostrar mais eficiente para o nosso caso de uso. Em seguida, veremos como atualizar o código da aplicação para usar o(s) algoritmo(s) escolhido(s).

A primeira coisa que precisamos fazer é adicionar o módulo de benchmarking como uma nova dependência do projeto. Utilize a mesma versão do módulo principal do OptaPlanner para evitar qualquer problema de incompatibilidade. A **Listagem 4** mostra o trecho que precisa ser adicionado na seção de dependências do arquivo *pom.xml*.

O próximo passo é definir os data sets que utilizaremos e também o formato como eles serão lidos. A ferramenta de benchmarking lê os data sets a partir de arquivos localizados dentro ou fora do projeto. Cada arquivo deve conter um objeto **Solution**

serializado em algum formato texto ou binário. Apenas para refrescar a memória, os objetos **Solution** (planning solution) são aqueles que armazenam os conjuntos de planning entities, *problem facts* e planning values referentes ao problema de planejamento. No nosso exemplo, é a classe **AlocacaoPassageiros**.

Por padrão, a ferramenta utiliza a biblioteca XStream para deserializar os data sets do formato XML para objetos Java. Alternativamente é possível utilizar outras bibliotecas, como JAX-B, ou mesmo outros formatos de serialização, como JSON. Só que para isso é necessário fornecer uma implementação customizada para a interface **SolutionFileIO**. Além de ler o conteúdo do arquivo, opcionalmente a ferramenta pode usar a implementação de **SolutionFileIO** para salvar o resultado da resolução do problema em arquivo no mesmo formato de leitura. Por enquanto vamos manter nosso exemplo simples e utilizar a implementação padrão. Sendo assim, só precisaremos alterar a classe **AlocacaoPassageiros** para incluir algumas anotações do XStream. Veja na **Listagem 5** como a classe deve ficar.

Listagem 4. Dependência do módulo de benchmarking a ser adicionado ao POM do projeto.

```
<dependency>
  <groupId>org.optaplanner</groupId>
  <artifactId>optaplanner-benchmark</artifactId>
  <version>6.1.0.Final</version>
</dependency>
```

Listagem 5. Alteração na classe **AlocacaoPassageiros** para inclusão das anotações do XStream.

```
@PlanningSolution
@XStreamAlias("alocacaoPassageiros")
public class AlocacaoPassageiros implements Solution<HardMediumSoftScore> {

    @XStreamImplicit(itemFieldName = "grupo")
    private List<Grupo> grupos;

    @XStreamImplicit(itemFieldName = "veiculo")
    private List<Veiculo> veiculos;

    private HardMediumSoftScore score;

    // restante da classe permanece inalterado
}
```

Os arquivos XML contendo os data sets podem ser baixados do repositório do exemplo no GitHub (veja o endereço na seção **Links**) ou do espaço para download na página desta edição da Java Magazine. Cada um desses arquivos representa uma dimensão diferente do mesmo problema. Feito o download, copie todos os arquivos do diretório *src/main/resources/data* para a mesma estrutura dentro do seu projeto.

A configuração do benchmarking é feita através de um arquivo XML presente no *classpath* da aplicação. Deste modo, crie um novo arquivo chamado *localSearch_benchmarkConfig.xml* no diretório *src/main/resources/benchmark* do projeto. Veja na **Listagem 6** o conteúdo completo desse XML.

O arquivo inicia definindo algumas configurações gerais do benchmarking. O primeiro elemento, **<benchmarkDirectory>**, é requerido e serve para especificar o diretório no qual a ferramenta deve gerar o relatório com os resultados (tabelas, estatísticas e gráficos) ao final do processamento.

Por padrão, todos os *solver benchmarks* definidos no arquivo são executados sequencialmente. Para economizar tempo, podemos optar por executá-los paralelamente. O valor **AUTO**, informado no elemento **<parallelBenchmarkCount>**, habilita essa *feature* e deixa que a ferramenta decida quantas *threads* usar (é possível também informar a quantidade exata de threads).

Nota

Embora a ferramenta de benchmarking permita paralelizar a execução de múltiplos benchmarks, o OptaPlanner em si (core) atualmente não tira proveito da disponibilidade de múltiplos CPUs. Quando o método *Solver.solve(...)* é invocado, todo o processamento interno ocorre em uma única thread. No entanto, já existe um ticket em aberto no Jira do projeto (<https://issues.jboss.org/browse/PLANNER-76>) que visa adicionar essa capacidade ao framework futuramente.

Outra configuração importante definida ainda no início do arquivo (elemento **<warmUpSecondsSpentLimit>**) é o período para aquecimento (*warm-up*) antes que o benchmarking seja executado para valer. Isso evita eventuais distorções nas primeiras execuções, possivelmente causadas pela compilação das regras do Drools e também por otimizações posteriores feitas pelo JIT (compilador *just-in-time* da JVM).

No segundo trecho do arquivo temos o elemento **<inheritedSolverBenchmark>**, o qual reúne as configurações comuns herdadas por todos os solver benchmarks. Dentro de **<problemBenchmarks>** especificamos os arquivos contendo os data sets e a classe decorada com as anotações do XStream que deve ser usada para deserializar o conteúdo dos arquivos. Neste momento vamos utilizar apenas dois dos arquivos copiados. Em aplicações reais é recomendado fazer o benchmarking com data sets de tamanhos bem próximos aos encontrados em produção. Por fim, habilitamos a coleta de algumas estatísticas usando o elemento **<problemStatisticType>**.

Ainda aninhado dentro do elemento **<inheritedSolverBenchmark>** temos o elemento **<solver>**. Este contém basicamente a mesma estrutura/valores que usamos para configurar o arquivo *solverConfig.xml* logo no início do artigo. A única diferença é que agora estamos utilizando o elemento **<termination>** para definir a condição de parada do solver. Há vários tipos de critérios de terminação disponíveis, como tempo máximo total, tempo máximo após a última melhoria no score, score mínimo, etc. É possível, inclusive, combinar múltiplos critérios no mesmo solver ou fase. Para este exemplo, especificamos apenas um tempo máximo de 20 segundos, o qual é aplicado a cada um dos solver benchmarks no arquivo.

O restante dos elementos corresponde basicamente à configuração dos solver benchmarks. Há um total de quatro. Cada um executa um algoritmo diferente, porém todos têm como input

Como escolher o algoritmo mais eficiente com OptaPlanner – Parte 2

os mesmos data sets. Poderíamos também testar diferentes configurações do mesmo algoritmo, bastando para isso adicionar um novo bloco <solverBenchmark> e ajustar as propriedades conforme desejado.

O primeiro solver benchmark da lista utiliza apenas a heurística de construção First Fit Decreasing. Como sabemos, as heurísticas de construção montam apenas a primeira solução possível, sem realizar melhorias posteriores. A ideia é avaliarmos o quanto

eficiente será essa primeira solução comparada com as soluções melhoradas pelas meta-heurísticas de busca local.

Os demais solver benchmarks combinam heurísticas de construção (novamente o First Fit Decreasing) com meta-heurísticas de busca local (Hill Climbing, Tabu Search e Late Acceptance, respectivamente). Todos esses algoritmos têm algumas propriedades configuradas com valores iniciais recomendados pela documentação do OptaPlanner. Apenas o algoritmo Tabu Search

Listagem 6. Arquivo de configuração do benchmarking (localSearch_benchmarkConfig.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<plannerBenchmark>
  <!-- Configurações gerais do benchmarking -->
  <benchmarkDirectory>local/benchmark/report</benchmarkDirectory>
  <parallelBenchmarkCount>AUTO</parallelBenchmarkCount>
  <warmUpSecondsSpentLimit>20</warmUpSecondsSpentLimit>

  <!-- Configurações comuns para todos benchmarks -->
  <inheritedSolverBenchmark>
    <problemBenchmarks>
      <xStreamAnnotatedClass>br.com.devmedia.agenciaturismo.planejamento.
        AlocacaoPassageiros</xStreamAnnotatedClass>
      <inputSolutionFile>src/main/resources/data/0050grupos_0010veiculos.xml
        </inputSolutionFile>
      <inputSolutionFile>src/main/resources/data/0500grupos_0100veiculos.xml
        </inputSolutionFile>

      <problemStatisticType>BEST_SCORE</problemStatisticType>
      <problemStatisticType>STEP_SCORE</problemStatisticType>
    </problemBenchmarks>

    <solver>
      <solutionClass>br.com.devmedia.agenciaturismo.planejamento.
        AlocacaoPassageiros</solutionClass>
      <entityClass>br.com.devmedia.agenciaturismo.dominio.Grupo</entityClass>

      <scoreDirectorFactory>
        <scoreDefinitionType>HARD_MEDIUM_SOFT</scoreDefinitionType>
        <scoreDrl>solver/alocacaoPassageirosScoreRules.drl</scoreDrl>
        <initializingScoreTrend>ONLY_DOWN</initializingScoreTrend>
      </scoreDirectorFactory>

      <termination>
        <secondsSpentLimit>20</secondsSpentLimit>
      </termination>
    </solver>
  </inheritedSolverBenchmark>

  <!-- Benchmarks -->

  <solverBenchmark>
    <name>First Fit Decreasing</name>
    <solver>
      <constructionHeuristic>
        <constructionHeuristicType>FIRST_FIT_DECREASING
        </constructionHeuristicType>
      </constructionHeuristic>
    </solver>
  </solverBenchmark>

  <solverBenchmark>
    <name>Hill Climbing</name>
    <solver>
      <constructionHeuristic>
        <constructionHeuristicType>FIRST_FIT_DECREASING
        </constructionHeuristicType>
      </constructionHeuristic>
    </solver>
  </solverBenchmark>

  <solverBenchmark>
    <name>Tabu Search</name>
    <solver>
      <constructionHeuristic>
        <constructionHeuristicType>FIRST_FIT_DECREASING
        </constructionHeuristicType>
      </constructionHeuristic>
      <localSearch>
        <acceptor>
          <acceptorType>HILL_CLIMBING</acceptorType>
        </acceptor>
        <forager>
          <acceptedCountLimit>1000</acceptedCountLimit>
        </forager>
      </localSearch>
    </solver>
  </solverBenchmark>

  <solverBenchmark>
    <name>Late Acceptance</name>
    <solver>
      <constructionHeuristic>
        <constructionHeuristicType>FIRST_FIT_DECREASING
        </constructionHeuristicType>
      </constructionHeuristic>
      <localSearch>
        <acceptor>
          <lateAcceptanceSize>400</lateAcceptanceSize>
        </acceptor>
        <forager>
          <acceptedCountLimit>4</acceptedCountLimit>
        </forager>
      </localSearch>
    </solver>
  </solverBenchmark>
</plannerBenchmark>
```

foi configurado para usar alguns *move selectors* extras (o que, eventualmente, pode lhe dar alguma vantagem). Os move selectors são componentes internos usados pelos algoritmos de otimização para selecionar os próximos movimentos possíveis para uma determinada solução. Mais detalhes sobre eles podem ser vistos na documentação do OptaPlanner.

Para que possamos usar o algoritmo First Fit Decreasing é necessário implementar um

comparator. Esse comparador é usado pelo algoritmo para ordenar as planning entities de acordo com o grau de dificuldade. Sendo assim, crie uma nova classe chamada **GrupoDifficultyComparator** que implemente a interface **Comparator<T>**. A regra de comparação deve fazer com que os elementos sejam ordenados de forma ascendente. Internamente, se o algoritmo realmente precisar começar pelos itens mais difíceis (neste caso, os maiores), ele inverte a ordenação. Veja o código completo da classe na **Listagem 7**.

Listagem 7. Código-fonte da classe **GrupoDifficultyComparator**.

```
package br.com.devmedia.agenciaturismo.planejamento.score;

import java.util.Comparator;

import br.com.devmedia.agenciaturismo.dominio.Grupo;

public class GrupoDifficultyComparator implements Comparator<Grupo> {

    @Override
    public int compare(Grupo g1, Grupo g2) {
        return Integer.compare(g1.getQuantidadePassageiros(),
            g2.getQuantidadePassageiros());
    }
}
```

Na sequência, edite a classe **Grupo** e informe o comparador que acabamos de criar no atributo **difficultyComparatorClass** da anotação **@PlanningEntity**, conforme mostrado na **Listagem 8**. Fazendo isso, o comparador será identificado quando executarmos o benchmarking e nenhum erro será lançado.

Por fim, crie uma nova classe chamada **LocalSearchBenchmark** e adicione um método **main()**. Esse método será responsável por construir um novo benchmarking a partir do arquivo de configuração e executá-lo. Veja o código completo na **Listagem 9**.

Agora que temos tudo pronto, é só executar a classe **LocalSearchBenchmark** e aguardar alguns minutos. Quando o benchmarking for concluído, os resultados serão salvos no diretório *local/benchmark/report* dentro do projeto, conforme configuramos anteriormente. Ao acessar esse caminho, abra o arquivo *index.html* em um *browser* para visualizar o relatório.

O relatório produzido pela ferramenta traz uma série de gráficos e tabelas comparando diferentes aspectos das configurações utilizadas no benchmarking.

Solver	Problem	
	0050grupos_0010veiculos	0500grupos_0100veiculos
First Fit Decreasing 3	-7hard/-8543medium/-9soft 3 !	0hard/-54933medium/-55soft 3
Hill Climbing 2	-4hard/-7974medium/-7soft 2 !	0hard/-54302medium/-54soft 1
Tabu Search 0	0hard/-6077medium/-4soft 0	0hard/-53172medium/-39soft 0
Late Acceptance 1	-2hard/-7226medium/-1soft 1 !	0hard/-54458medium/-42soft 2

Figura 3. Comparação dos melhores scores por configuração x data set

A tabela exibida na **Figura 3** (algumas colunas foram suprimidas) apresenta os melhores scores obtidos por cada uma das configurações para os diferentes data sets.

Veja que, entre as configurações que foram testadas, o algoritmo Tabu Search é o que gera os melhores resultados. Portanto, ele é classificado pela ferramenta como o solver preferido.

Para ambos os data sets foi possível encontrar uma solução viável (nenhuma restrição hard foi quebrada). Perceba também que o algoritmo First Fit Decreasing consegue encontrar uma solução boa de início, mas às vezes ela não é viável (como no primeiro data set) ou não muito eficiente (como no segundo).

Listagem 8. Alteração na classe **Grupo** para inclusão do comparador de dificuldade.

```
@PlanningEntity(difficultyComparatorClass = GrupoDifficultyComparator.class)
public class Grupo {

    // restante da classe permanece inalterada
}
```

Listagem 9. Código-fonte da classe **LocalSearchBenchmark**.

```
package br.com.devmedia.agenciaturismo.planejamento.benchmark;

import org.optaplanner.benchmark.api.PlannerBenchmark;
import org.optaplanner.benchmark.api.PlannerBenchmarkFactory;

public class LocalSearchBenchmark {
```

```
    public static void main(String[] args) {
        PlannerBenchmarkFactory plannerBenchmarkFactory =
            PlannerBenchmarkFactory
                .createFromXmlResource("benchmark/localSearch_benchmarkConfig.xml");
        PlannerBenchmark plannerBenchmark =
            plannerBenchmarkFactory.buildPlannerBenchmark();
        plannerBenchmark.benchmark();
    }
}
```

O relatório de benchmarking também mostra alguns gráficos interessantes referentes às estatísticas que habilitamos no arquivo de configuração. No gráfico da **Figura 4** podemos observar como o melhor score evolui ao longo do tempo (apenas o nível hard é mostrado no gráfico). Em menos de três segundos o algoritmo Tabu Search (linha verde) já consegue encontrar uma solução viável. Enquanto isso, o algoritmo Hill Climbing (linha azul) tem uma ascensão bem mais lenta, chegando a ficar preso ao final sem conseguir melhorar o score.

Como escolher o algoritmo mais eficiente com OptaPlanner – Parte 2

Já o gráfico exibido na Figura 5 mostra como o score progride conforme novos movimentos vão sendo aceitos pelos algoritmos. Como o algoritmo Hill Climbing (linha vermelha) não aceita movimentos que deterioraram o score corrente, ele tem

uma progressão sempre ascendente. Em contrapartida, o algoritmo Late Acceptance (linha verde) fica oscilando, uma vez que este considera movimentos que, inclusive, deterioraram o score. É claro que, se mexermos nos parâmetros de cada

algoritmo e executarmos novos benchmarkings (o leitor é incentivado a fazer isso), poderemos melhorar ainda mais os resultados e, inclusive, outros algoritmos poderão se mostrar mais eficientes.

Trocando de algoritmos

Uma vez que temos um candidato a melhor algoritmo/configuração para o nosso caso de uso, podemos alterar a aplicação para substituir o algoritmo de força bruta que definimos inicialmente. Esse procedimento é muito simples de se fazer, pois não requer nenhuma mudança no *design* do modelo de domínio e nem nas regras de score. Basta ajustar o arquivo de configuração do solver. Às vezes, dependendo do algoritmo, é necessário também implementar pequenas porções de código, como comparators. Mas, no geral, a mudança no código Java é mímina.

Como já houve a necessidade de se implementar a classe *GrupoDifficultyComparator* por conta do benchmarking, a única alteração que nos resta a fazer é no arquivo *solverConfig.xml*. Portanto, substitua todo o elemento *<exhaustiveSearch>...</exhaustiveSearch>* pelo conteúdo do elemento *<solver>* referente ao solver benchmark “Tabu Search” (arquivo *localSearch_benchmarkConfig.xml*). Além disso, adicione o elemento *<termination>* para que o algoritmo não fique executando indefinidamente. Uma opção mais simples é copiar a configuração completa do solver Tabu Search da página de relatório do benchmarking e sobreescrivar todo o arquivo *solverConfig.xml*. A Listagem 10 mostra o arquivo *solverConfig.xml* com a alteração aplicada.

Neste momento já poderíamos executar novamente a aplicação para verificarmos o resultado das alterações. Mas, antes disso, vamos fazer mais uma pequena mudança. Repare que o problema sendo construído pela classe **Main** é bastante pequeno e não tiraria muito proveito da nova configuração. Para realmente observarmos as melhorias implementadas, precisamos de um dataset mais desafiador. Para isso, iremos reutilizar um dos arquivos XML que foram usados no benchmarking para criar uma instância do problema. Sendo assim,

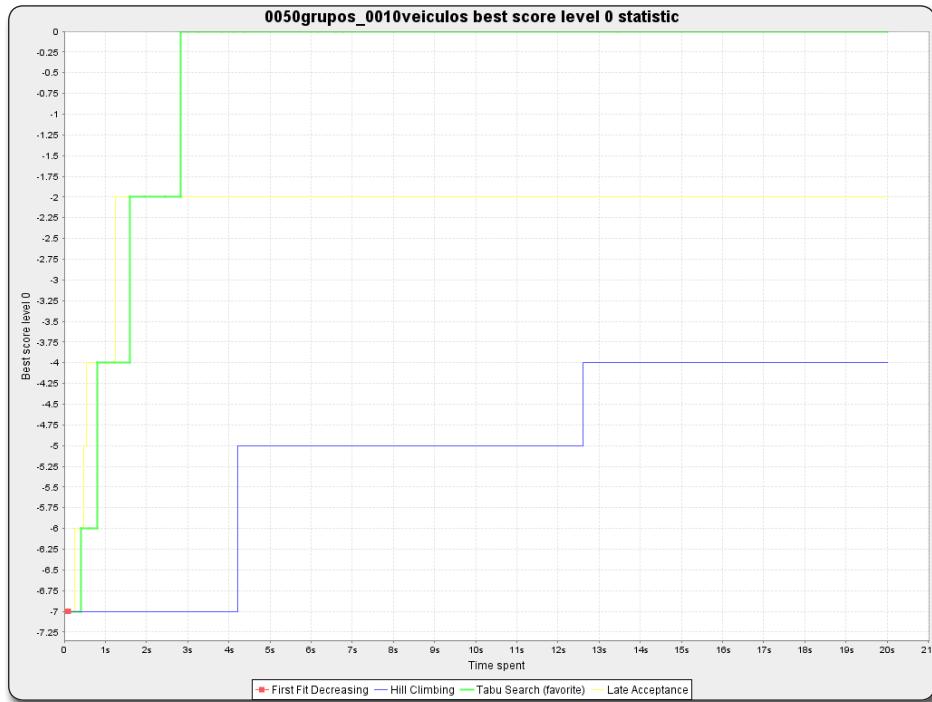


Figura 4. Gráfico mostrando a evolução do melhor score ao longo do tempo

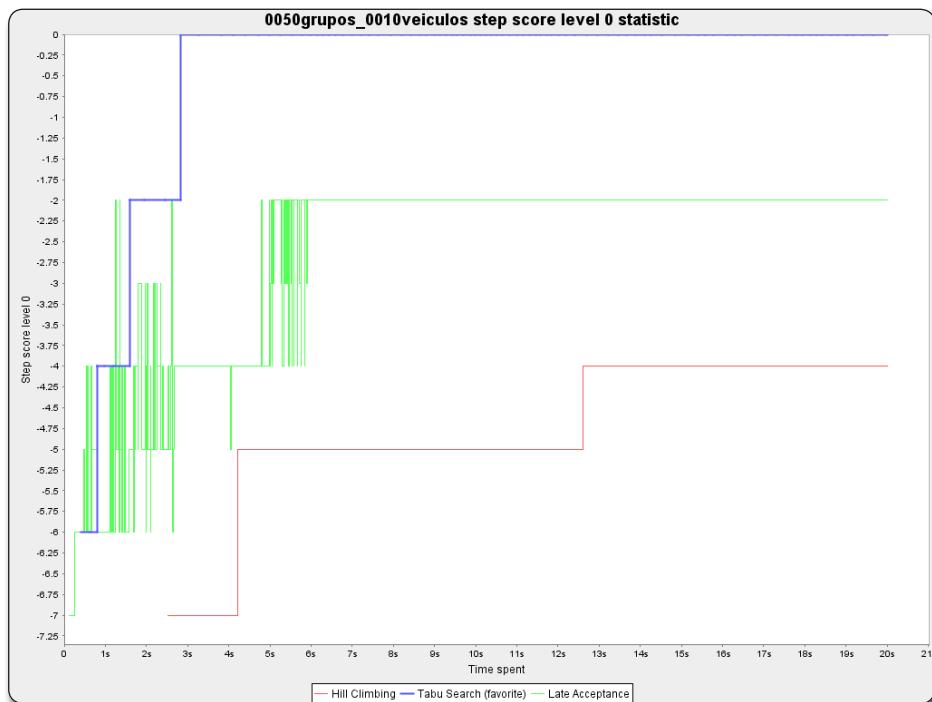


Figura 5. Gráfico mostrando os scores aceitos ao longo do tempo

altere o método **construirProblema()** e utilize o XStream para ler o conteúdo do arquivo. Veja essa mudança na **Listagem 11**.

Listagem 10. Arquivo de configuração solverConfig.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<solver>
    <!-- Modelo de domínio -->
    <solutionClass>br.com.devmedia.agenciaturismo.planejamento.
        AlocacaoPassageiros</solutionClass>
    <entityClass>br.com.devmedia.agenciaturismo.dominio.Grupo</entityClass>

    <!-- Score -->
    <scoreDirectorFactory>
        <scoreDefinitionType>HARD_MEDIUM_SOFT</scoreDefinitionType>
        <scoreDrl>solver/allocacaoPassageirosScoreRules.drl</scoreDrl>
        <initializingScoreTrend>ONLY_DOWN</initializingScoreTrend>
    </scoreDirectorFactory>

    <!-- Algoritmos de otimização -->
    <termination>
        <secondsSpentLimit>20</secondsSpentLimit>
    </termination>

    <constructionHeuristic>
        <constructionHeuristicType>FIRST_FIT_DECREASING
        </constructionHeuristicType>
    </constructionHeuristic>

    <localSearch>
        <acceptor>
            <entityTabuSize>7</entityTabuSize>
        </acceptor>
        <forager>
            <acceptedCountLimit>1000</acceptedCountLimit>
        </forager>
        <unionMoveSelector>
            <changeMoveSelector />
            <swapMoveSelector />
            <pillarChangeMoveSelector />
            <pillarSwapMoveSelector />
        </unionMoveSelector>
    </localSearch>
</solver>
```

Listagem 11. Alteração na classe Main para carregar data set a partir de arquivo.

```
public class Main {

    // outros métodos omitidos

    private static AlocacaoPassageiros construirProblema() {
        XStream xStream = new XStream();
        xStream.setMode(XStream.NO_REFERENCES);
        xStream.processAnnotations(AlocacaoPassageiros.class);
        return (AlocacaoPassageiros) xStream.fromXML(Main.class
            .getResourceAsStream("/data/0050grupos_0010veiculos.xml"));
    }
}
```

Agora sim. Execute novamente a classe **Main** e aguarde alguns segundos para ver o resultado sendo impresso no console. É esperado que a aplicação consiga encontrar uma solução viável para o problema dentro de um curto período de tempo.

Vale ressaltar que não existe um algoritmo “bala-de-prata” que sempre será o mais eficiente para todos os casos de uso imagináveis. Escolher a configuração perfeita também não é uma tarefa muito trivial, uma vez que isso pode requerer vários ciclos de testes e refinamentos. Felizmente existe a ferramenta de benchmarking para ajudar a fazer essa escolha.

Mas, para aqueles que desejam alcançar um resultado bom sem perder muito tempo com experimentações, Geoffrey De Smet (líder do projeto OptaPlanner) recomenda como ponto de partida uma configuração que combina First Fit Decreasing com Late Acceptance. Essa configuração já é suficientemente boa para conseguir alcançar ótimos resultados. Melhor que isso, só mesmo experimentando ajustes e fazendo novos benchmarkings.

Há muitas outras informações que não puderam ser cobertas pelo artigo, mas que valem a pena conhecer. Por isso, não deixe de dar uma olhada na documentação oficial e em outros recursos disponibilizados no site do projeto para aprender um pouco mais sobre esse interessante framework.

Autor



Marcelo A. Cenerino

marcelocenerine@gmail.com

É graduado em Sistemas de Informação e pós-graduado em Desenvolvimento de Software para Web e Computação Ubíqua pela UFSCar. Trabalha com desenvolvimento de software na plataforma Java há mais de seis anos. Possui, entre outras, as certificações SCJA, SCJP, OCPJWCD, OCPJBCD, OCPJWSD e OCMJEA.



Links:

Código-fonte completo do exemplo do artigo no GitHub.

github.com/marcelocenerine/javamagazine-optaplanner

Site do OptaPlanner.

www.optaplanner.org/

Código-fonte do projeto OptaPlanner no GitHub.

github.com/droolsjbpm/optaplanner

Essentials of Metaheuristics.

cs.gmu.edu/~sean/book/metaheuristics/

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Crie sistemas web com JSF e HTML5

Veja neste artigo como empregar recursos do HTML5 em aplicações JSF

A sétima versão do JavaServer Faces (JSF) foi lançada em junho de 2014, junto com a plataforma Java EE 7. Nesta versão, uma das principais novidades é a possibilidade de utilizar recursos do HTML5, tecnologia que busca adicionar maior descrição semântica aos documentos HTML, facilitando assim a interpretação do conteúdo pelos navegadores. Essa inovação traz como grande destaque o uso de um padrão que permite a adaptação do conteúdo para dispositivos móveis, como *smartphones* e *tablets*, otimizar técnicas de SEO, visando melhorar o posicionamento em buscadores, e também o aprimoramento da acessibilidade a pessoas com deficiência, visto que aplicações em HTML5 possibilitam a navegação via atalhos de teclado.

Lançado em outubro de 2014, o HTML5 é a última versão da linguagem de marcação que utilizamos para a construção de páginas web. Dentre as novas funcionalidades, foram incluídas algumas mais simples, como os novos tipos de campos para entrada de dados, e outras mais complexas, como a possibilidade de criação de imagens 2D com a nova tag Canvas.

Dito isso, neste artigo, primeiramente vamos introduzir e revisar os conceitos básicos do JSF, a fim de facilitar o entendimento do restante do artigo. Posteriormente, vamos apresentar as novidades do JSF, focando principalmente no *Pass Through Elements*, que é uma funcionalidade que permite desenvolver as view do JSF em XHTML puro. Além disso, para exemplificar o uso do HTML5 com JSF, será implementada uma aplicação exemplo, na qual um usuário poderá cadastrar e revisar lembretes, que estarão acessíveis tanto pelo computador quanto pelo *smartphone*.

Conceitos básicos

O entendimento dos conceitos básicos do JSF é fundamental para a sequência do artigo, pois as novas funcionalidades dependem de algumas definições originadas nas versões anteriores.

Fique por dentro

Esse artigo é útil para estudantes e profissionais que tenham interesse em utilizar o JavaServer Faces em conjunto com o HTML5. Para isso, inicialmente será apresentado o funcionamento básico do JSF, e depois analisaremos como realizar a integração entre essas tecnologias a partir de um exemplo simples, para que não percamos o foco no tema em estudo. Basicamente, ao adotar duas novas tags do JSF conseguimos reunir o melhor de dois mundos, possibilitando a criação de interfaces limpas e responsivas, dois dos principais pré-requisitos a toda solução web.

Os principais conceitos do JSF que serão importantes para a compreensão deste artigo são:

- **Renderização:** Processo que transforma código Java do servidor em páginas HTML, para que seja possível a leitura das páginas pelo navegador;
- **Facelets:** Linguagem declarativa usada pelo JSF para criação de páginas HTML;
- **Faces-Servlet:** Controlador padrão, que comanda o ciclo de vida do framework JSF;
- **Navegação:** Está relacionado às regras que determinam como serão efetuadas as transições entre as páginas da aplicação Web;
- **ManagedBeans:** Classes Java com anotações do framework JSF usadas nos componentes das páginas web de uma aplicação;
- **View:** Qualquer página JSF declarada com tags do framework e tags HTML.

Além disso, na nova versão do JSF foram adicionados dois recursos que permitem que trabalhemos de forma integrada com o HTML5. São eles:

- **PassThrough Attributes:** Nova tag do JSF que permite a utilização de atributos do HTML5 em componentes JSF;
- **PassThrough Elements:** com esse conceito é possível adicionar a uma tag HTML atributos do JSF. Por exemplo, podemos criar uma tag **button** da seguinte forma: `<button jsf:id="salvar">`. Isso permite desenvolver páginas JSF também com a linguagem HTML, e não apenas com tags JSF.

Nas primeiras versões do JSF, o principal arquivo de configuração era o *faces-config.xml*. Esse arquivo ainda está presente, porém, as últimas versões do framework, incluindo a 2.2, facilitam o desenvolvimento através de annotations, seguindo uma tendência adotada também por outros frameworks, como Hibernate e Spring.

A **Figura 1** apresenta a arquitetura básica de uma aplicação JSF. Nesta arquitetura o principal componente é o FacesServlet, que irá comandar a navegação entre as páginas e suas interações com os ManagedBeans. Lembre-se que o FacesServlet pode encaminhar a requisição diretamente para um XHTML ou pode enviar a requisição para um ManagedBean, dependendo do mapeamento que foi feito no *faces-config.xml*.

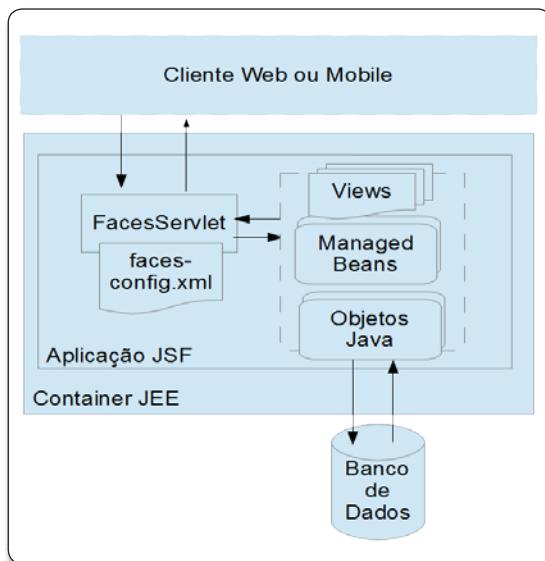


Figura 1. Arquitetura do JSF

O ciclo de vida do JSF, descrito na **Figura 2**, inicia na fase *Restore View*, que verifica de onde veio a requisição, ou chamada HTTP. A segunda fase é a *ApplyRequestValues*, que recupera os valores dos parâmetros dessa chamada. Em seguida, *ProcessValidations* faz as validações definidas para cada um desses parâmetros. Se algum erro for encontrado, é gerada uma resposta com uma mensagem descrevendo esse erro. Caso contrário, é executada a fase *UpdateModelValues*, na qual os valores dos parâmetros da requisição são colocados nos atributos do *ManagedBean*. Já na fase *InvokeApplication* é executado o método do *ManagedBean* que foi chamado, e por último, na fase *RenderResponse*, é feita a renderização da view acessada pelo usuário.

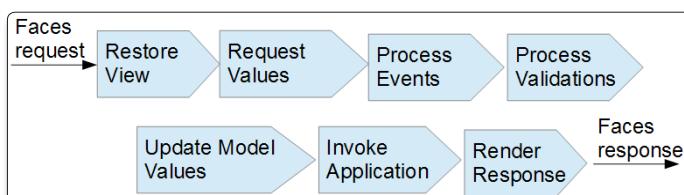


Figura 2. Ciclo de vida

Conhecer o ciclo de vida do JSF é fundamental para entender como o framework trata as requisições do usuário, quais os passos internos que são executados, os motivos que levam à necessidade de certas configurações e assim poder tomar a melhor decisão durante a implementação de suas soluções.

Instalação e configuração

Para o desenvolvimento da nossa aplicação, necessitamos do Eclipse e do Tomcat instalados. Obviamente poderíamos apresentar os exemplos sem essas ferramentas, mas é fundamental utilizar uma ferramenta que apoie a codificação e ter acesso a um servidor que atenda o padrão de referência do JSF (além do Tomcat, podemos citar o GlassFish e o JBoss). Ademais, para simplificar o nosso estudo, vamos utilizar o Maven.

Assim, para fazer uso dos recursos do JSF, devemos incluir duas dependências deste ao projeto. Deste modo, adicione o conteúdo indicado na **Listagem 1** no seu *pom.xml*. Estas dependências estão relacionadas à *jsf-api*, que contém a especificação do framework, e a *jsf-impl*, que contém a implementação padrão fornecida pela Oracle.

Listagem 1. Dependências Maven do JSF.

```

<dependencies>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.2.0</version>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.2.0</version>
  </dependency>
</dependencies>
  
```

Após a configuração do projeto, a primeira tarefa que devemos fazer é configurar o *web.xml*, como ilustrado na **Listagem 2**. Com isso, passamos a utilizar o FacesServlet no controle da aplicação e no gerenciamento das requisições que chegam ao servidor.

Na configuração, a tag **<servlet>** define qual será o servlet do JSF, ou Faces-Servlet, como definido anteriormente. A tag **<servlet-name>**, por sua vez, nomeia o Servlet a fim de facilitar sua identificação e referência no código. Qualquer nome pode ser utilizado, mas o padrão é **FacesServlet**. Logo após, **<servlet-class>** especifica a classe que implementa o servlet do JSF (no caso, **javax.faces.webapp.FacesServlet**). Já a tag **<load-on-startup>** informa que o servlet deve ser instanciado assim que o Tomcat for iniciado. Outra tag importante é a **<servlet-mapping>**, que determina qual o padrão de endereço para que a requisição seja executada pelo JSF. No exemplo da **Listagem 2**, todos os endereços que começam com */faces/* ou que terminam com **.faces*, **.jsf* ou **.xhtml*, terão suas requisições tratadas pelo JSF. Por fim, a tag **<welcome-file-list>** especifica a página inicial da aplicação.

Depois da configuração do FacesServlet, é possível implementar os beans gerenciados (ManagedBean) da aplicação.

Exemplificado na **Listagem 3**, um ManagedBean é uma classe Java gerenciada pelo JSF que possui uma série de anotações para especificar como o framework irá acessar suas propriedades e seus métodos. Nesse código, definimos que o nome do ManagedBean é **usuario**, e os métodos, como **login()**, definem qual será a ação tomada após cada requisição.

Listagem 2. Conteúdo do arquivo web.xml.

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

Listagem 3. Exemplo mais simples de bean gerenciado.

```
@ManagedBean(name="usuario")
@SessionScoped
public class UserBean{

  private String nome, email, usuario, senha;
  private Date dataNascimento;

  public String login() {
    return "sucesso";
  }
}
```

Como ilustrado nesse código, uma das anotações que devem ser utilizadas em um ManagedBean é a que define o seu escopo. O escopo é quem especifica o ciclo de vida de uma instância de um ManagedBean. Do ponto de vista prático, isso decide quanto tempo os valores armazenados em um bean estarão disponíveis. A **Tabela 1** apresenta os quatro tipos de escopos oferecidos pelo JSF.

Annotation	Descrição
@RequestScoped	Define que o bean estará disponível apenas durante o processamento da requisição.
@ViewScoped	Define que o bean estará disponível enquanto o usuário estiver em uma mesma view.
@SessionScoped	Define que o bean estará disponível durante toda a sessão do usuário.
@ApplicationScoped	Define que o bean estará disponível enquanto a aplicação estiver instalada.

Tabela 1. Escopos disponíveis na implementação padrão do JSF

Além do arquivo *web.xml*, o JSF necessita que seja criado o *faces-config.xml*. Ainda que a maioria das configurações possam ser feitas com annotations, alguns detalhes e parâmetros globais devem ser especificados nesse arquivo. O *faces-config.xml* irá definir parâmetros como o idioma padrão da aplicação, a navegação entre as páginas (conforme explicado na seção anterior) e a localização do arquivo de internacionalização, caso exista. A **Listagem 4** apresenta um exemplo de código para o *faces-config.xml*, onde foi criado um recurso (um **resource-bundle**) para armazenar mensagens (neste exemplo, em português e inglês) que serão apresentadas pela aplicação dependendo da configuração de localização do browser.

Listagem 4. Conteúdo do arquivo faces-config.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
  version="2.0">
  <application>
    <locale-config>
      <default-locale>pt_BR</default-locale>
      <supported-locale>en</supported-locale>
    </locale-config>
    <resource-bundle>
      <base-name>com.organizacao.mensagens</base-name>
      <var>msg</var>
    </resource-bundle>
  </application>
</faces-config>
```

O próximo passo no desenvolvimento da aplicação é criar as páginas JSF que irão receber as requisições e exibir as respostas ao usuário.

Para nosso estudo, será desenvolvida uma página JSF responsável por apresentar os dados de um usuário. O código dessa página pode ser visto na **Listagem 5**. Ao analisá-lo, note que utilizamos a tag **<h:outputLabel>** e dentro dela os atributos **rendered**, que define se o componente será renderizado ou não; **binding**, que estabelece a ligação entre essa tag e uma propriedade de um ManagedBean (nesse caso, os atributos **nome**, **dataNascimento** e **email**); e **lang**, que especifica o idioma adotado.

Uma vez revisado como apresentar dados em JSF, podemos avançar um pouco mais e criar um formulário, responsável por enviar informações ao servidor usando tags JSF. A **Listagem 6** apresenta o código correspondente. Nele, encontraremos dois **h:outputText** para renderizar os textos estáticos definidos no atributo **value**. Além disso, as tags **h:inputText** e **h:inputSecret** serão renderizadas como elementos do tipo *input* de um formulário HTML, respectivamente com os tipos **text** e **password**. Finalmente, o **h:commandButton** definirá qual método do ManagedBean será responsável por receber os dados enviados ao servidor (nesse caso, o método **login()**). Esse método retornará a **String** “sucesso”, que é utilizada pela regra de navegação para definir qual será a próxima página a ser apresentada pela aplicação.

Listagem 5. Exemplo de página JSF para apresentação de dados de um usuário.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>JSF 2.2!</title>
</head>
<body>
<h:outputLabel
rendered="true"
binding="#{usuario.nome}"
lang="pt-br">
</h:outputLabel>
<h:outputLabel
rendered="true"
binding="#{usuario.dataNascimento}"
lang="pt-br">
</h:outputLabel>
<h:outputLabel
rendered="true"
binding="#{usuario.email}"
lang="pt-br">
</h:outputLabel>
</body>
</html>
```

Listagem 6. Formulário JSF para envio dos dados de um usuário.

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:c="http://java.sun.com/jsp/jstl/core">
<h:body>
<h:form id="form">
<h1>
<h:outputText value="Usuário: " />
<h:inputText id="inputUsername" value="#{usuario.usuario}" />
</h1>
<h1>
<h:outputText value="Senha: " />
<h:inputSecret id="inputPassword" value="#{usuario.senha}" />
</h1>
<h:commandButton value="Submit" action="#{usuario.submit}" />
</h:form>
</h:body>
</html>
```

Neste exemplo foram adotadas as tags **outputText**, **inputText**, **inputSecret** e **commandButton**, mas existem muitas outras no framework JSF. A **Tabela 2** descreve as mais importantes. O leitor, se familiarizado com HTML, conseguirá notar a semelhança dessas tags com as existentes na linguagem de marcação padrão da web.

Tag	Descrição
h:inputSecret	Renderiza um campo de entrada com o atributo type="password".
h:inputText	Renderiza um campo de entrada com o atributo type="text".
h:inputTextarea	Renderiza um campo de entrada de texto.
h:inputHidden	Renderiza um campo de entrada com o atributo type="hidden".
h:selectBooleanCheckbox	Renderiza um check box.
h:selectManyCheckbox	Renderiza um grupo de check boxes.
h:selectOneRadio	Renderiza um radio button.
h:selectOneListbox	Renderiza um list box.
h:selectManyListbox	Renderiza uma caixa de múltipla escolha.
h:selectOneMenu	Renderiza um combo box.
h:outputText	Renderiza um text.
h:outputFormat	Renderiza uma saída de texto simples.
h:graphicImage	Renderiza uma imagem.
h:outputStylesheet	Inclui uma folha de estilo CSS.
h:outputScript	Inclui um script.
h:commandButton	Renderiza um botão com o atributo type="submit".
h:Link, h:commandLink e h:outputLink	Renderiza um link.
h:panelGrid	Renderiza uma tabela.
h:message	Renderiza uma mensagem para um componente JSF.
h:messages	Renderiza todas as mensagens de todos os componentes JSF.
f:param	Passa parâmetros para um componente JSF.
f:attribute	Passa atributos para um componente JSF.
f:setPropertyActionListener	Coloca um valor em um atributo de um ManagedBean.

Tabela 2. Principais tags do JSF

Integração com HTML5

A nova versão do JSF inclui diversas atualizações para poder utilizar recursos do HTML5. Algumas dessas atualizações, no entanto, são invisíveis para o programador, pois a diferença se concentra na renderização da página, onde algumas tags do JSF são automaticamente transformadas em tags HTML5. Por outro lado, outras novidades do JSF necessitam de alguma intervenção do programador, como o atributo **type** do **h:inputText**, que agora pode receber as opções **text**, **search**, **email**, **url**, **tel**, **range** e **number**. Essas opções facilitam, por exemplo, a criação de códigos para validação dos valores inseridos nos campos de um formulário.

Ainda no JSF 2.2 foi criada a tag **passThroughAttribute**, pois por uma limitação do framework, alguns atributos do HTML5 não podiam ser adicionados diretamente em componentes JSF, como os atributos **required**, **url**, **range** e **placeholder**, empregado na próxima listagem.

A **Listagem 7** mostra um exemplo com a tag **passThroughAttribute** utilizando o **placeholder** em um **h:inputText** e em um **h:inputTextArea**. Esse atributo exibe um texto de explicação enquanto as caixas de texto estiverem vazias. Geralmente é utilizado para evitar o uso de rótulos em formulários e deixar o design mais limpo. A **Listagem 7** mostra também como criar um input do tipo **range** e um input do tipo **url** em tags **h:inputText**. Tais recursos – acrescentadas no HTML5 – serão renderizadas, respectivamente, como uma escala de valores crescentes (no exemplo, uma barra que terá como valor mínimo 1 e máximo 10) e uma caixa de texto que só aceitará valores no formato de uma URL.

Nas versões anteriores, para desenvolver o XHTML, a única opção era usar tags JSF, o que dificultava muito o trabalho de designers e de pessoas que não conhecem bem o JSF. Na atual versão deste framework, o conceito de *Pass Through Elements* permite que parte do código da view seja desenvolvido em XHTML puro, utilizando os atributos JSF diretamente em tags HTML.

Como mostra a **Listagem 8**, podemos utilizar um atributo de uma tag com o namespace **jsf**: e as tags que contenham esses atributos serão transformadas em componentes JSF. No caso desse código, a tag **<input>** será transformada em um **h:inputText** e a tag **<button>** será transformada em um **<h:commandButton>**.

Listagem 7. Código para utilizar o atributo placeholder em um **inputText** e em um **inputTextArea**.

```
<h:inputText id="email" value="#{bean.email}">
  <f:passThroughAttribute name="placeholder" value="Entre com seu email"/>
</h:inputText>

<h:inputTextArea id="descricao" value="#{bean.descricao}">
  <f:passThroughAttribute name="placeholder" value="Entre com seu email"/>
</h:inputTextArea>

<input type="range" jsf:id="relevance" jsf:value="#{notebookBean.notebook.relevance}" min="0" max="10" />

<input type="url" jsf:id="url" placeholder="Entre com uma URL" jsf:value="#{notebookBean.notebook.url}" />
```

Listagem 8. Utilizando Pass Through Elements.

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:jsf="http://xmlns.jcp.org/jsf">
<head jsf:id="head"><title>JSF 2.2</title></head>
<body jsf:id="body">
  <form jsf:id="form">
    <input type="text" jsf:id="name" placeholder="Enter name" jsf:value="#{bean.name}"/>
    <button jsf:action="#{bean.save}">Save</button>
  </form>
</body>
</html>
```

Para melhor demonstrar a integração do JSF com HTML5, criaremos uma aplicação exemplo simples, responsável por cadastrar e listar anotações. A **Listagem 9** mostra o código de uma classe POJO, a ser utilizada em todo o exemplo. Essa classe, de nome **Notebook**, tem os seguintes atributos: **title**, que representa o título da anotação; **text**, que armazenara o texto da anotação; **date**, que representa a data em que a anotação foi feita; e **relevance**, que indica a importância da anotação criada.

Listagem 9. Código do bean **Notebook**.

```
public class Notebook {

  private String title;
  private String text;
  private Date date;
  private int relevance;

  public String getTitle() {
    return title;
  }
  public void setTitle(String title) {
    this.title = title;
  }
  public String getText() {
    return text;
  }
  public void setText(String text) {
    this.text = text;
  }
  public Date getDate() {
    return date;
  }
  public void setDate(Date date) {
    this.date = date;
  }
  public int getRelevance() {
    return relevance;
  }
  public void setRelevance(int relevance) {
    this.relevance = relevance;
  }
}
```

Em seguida, devemos criar o ManagedBean **NotebookMB**, com o escopo **ViewScoped** para gerenciar as ações da tela. Este ManagedBean tem como atributos **notebook** e uma lista de notebooks chamada **notebookList**. Além disso, a classe tem o método **addNotebookToList()**, que implementa a ação de criar um novo objeto do tipo **notebook**, e depois o adiciona à lista de notebooks. Esse método tem retorno do tipo **void**, pois a resposta será enviada para a mesma view. A **Listagem 10** mostra o código da classe **NotebookMB**.

Depois de criado o ManagedBean, devemos codificar a view da tela de cadastro e listagem de notebooks (veja a **Listagem 11**). Para isso, foi utilizado o recurso *Pass Through Elements*. Esta tela tem um componente para cada atributo da classe **Notebook**. Sendo assim, para o atributo **title**, foi utilizado um input com o **type text**.

Para a data, foi utilizado um input com o **type date**, que cria um campo com um calendário. Para o atributo **relevance**, foi criado um input com o **type range**, que cria uma barra numérica com

Listagem 10. Código da classe NotebookMB.

```
@ManagedBean(name="notebookBean")
@ViewScoped
public class NotebookMB {

    private Notebook notebook = new Notebook();
    private List<Notebook> notebookList = new ArrayList<Notebook>();

    public void addNotebookToList() {
        notebook = new Notebook();
        notebookList.add(notebook);
    }

    public Notebook getNotebook() {
        return notebook;
    }

    public void setNotebook(Notebook notebook) {
        this.notebook = notebook;
    }

    public List<Notebook> getNotebookList() {
        return notebookList;
    }

    public void setNotebookList(List<Notebook> notebookList) {
        this.notebookList = notebookList;
    }
}
```

o valor mínimo 0 e máximo 10. E para o atributo **text**, foi criado um **textArea**. Além disso, na mesma tela, foi criado um **dataTable** para listar os notebooks criados.

Na construção do XHTML, em todos os campos de cadastro foi adotado um elemento do HTML5, como nos campos **title** e **text**, onde foi utilizado o atributo **placeholder**, um recurso do HTML5 que exibe um texto inicial que desaparece assim que o usuário clica sobre o componente relacionado. Já nos campos **relevance** e **date**, foram utilizados dois dos novos tipos de entrada disponíveis no HTML5: **date** e **range**. O tipo **range** exibe um slider, com um valor mínimo e máximo definidos, e o tipo **date** mostra um calendário para facilitar a escolha de uma data.

A **Figura 3** apresenta o resultado da view ao ser acessada a partir do browser em um desktop. No campo relacionado ao título da anotação é possível notar o atributo **placeholder** em ação.

Título	Data	Relevância	Texto
Java	24/12/2014	4	A linguagem java...
PHP	26/12/2014	4	O PHP

Figura 3. Tela criada com JSF e HTML5 visualizada no browser

RENOVE JÁ!

Sua assinatura pode estar acabando

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

The Evening Herald
WE'RE HERE FROM THE WORLD EXTRA!

java Magazine
Mobile
.NET
SQL
Delphi
REST e JSON

Crie sistemas web com JSF e HTML5

Agora, para confirmar que o HTML 5 funciona também em dispositivos móveis, garantindo assim responsividade de nossas aplicações web, a **Figura 4** mostra a mesma página exibida na **Figura 3** quando a acessamos a partir de um smartphone.

Com o conceito de Pass Through Elements é possível ainda desenvolver páginas mais complexas, adotando outros recursos do HTML 5, como por exemplo, a nova API para geolocalização.

Listagem 11. XHTML com pass through elements.

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:jsf="http://xmlns.jcp.org/jsf"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
<head jsfid="head">
<title>JSF 2.2</title>
</head>
<body jsfid="body">
<form jsfid="form">
  <input type="text" jsfid="title" placeholder="Digite o título"
    jsf.value="#{notebookBean.notebook.title}" />
  <br><br>
  <input
    type="date" jsfid="date" jsf.value="#{notebookBean.notebook.date}"
    <f:convertDateTime pattern="yyyy-MM-dd"/>
  </input>
  <br><br>
  <input type="range" jsfid="relevance"
    jsf.value="#{notebookBean.notebook.relevance}" min="0" max="10" />
  <br><br>
  <textarea rows="4" cols="50" jsfid="text"
    jsf.value="#{notebookBean.notebook.text}" placeholder="Digite o texto" />
  <br><br>
  <button jsfid="salvar">
    Salvar
    <f:ajax execute="@form" event="click" render="lista"
      listener="#{notebookBean.addNotebookToList}" />
  </button>
</form>
```

A **Listagem 12** mostra o código de uma página que recupera a localização do usuário com HTML5 e passa essa informação, via JavaScript, para os componentes JSF.

Nessa página, as funções JavaScript `getLocation()` e `showPosition()` são responsáveis por recuperar a latitude e a longitude do usuário. Caso o browser não suporte essa API, é mostrada uma mensagem de erro. Caso contrário, a latitude é colocada no `inputText` com `id lat` e a longitude é colocada no `inputText` com `id lon`.

```
</button>
<br><br>
<h: dataTable id="lista" value="#{notebookBean.notebookList}"
  var="notebook" styleClass="order-table"
  headerClass="order-table-header"
  rowClasses="order-table-odd-row,order-table-even-row">
  <h: column>
    <f: facet name="header">Título</f: facet>
    <h: outputText value="#{notebook.title}" />
  </h: column>
  <h: column>
    <f: facet name="header">Data</f: facet>
    <h: outputText value="#{notebook.date}" />
  </h: column>
  <h: column>
    <f: facet name="header">Relevância</f: facet>
    <h: outputText value="#{notebook.relevance}" />
  </h: column>
  <h: column>
    <f: facet name="header">Texto</f: facet>
    <h: outputText value="#{notebook.text}" />
  </h: column>
</h: dataTable>
</form>
</body>
</html>
```

Listagem 12. Exemplo mais avançado de XHTML com Pass Through elements.

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:jsf="http://xmlns.jcp.org/jsf"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
<head jsfid="head">
<title>JSF 2.2</title>
<script>
  function getLocation() {
    if (navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(showPosition);
    } else {
      x.innerHTML = "Geolocation is not supported by this browser.";
    }
  }

  function showPosition(position) {
    var inputLat = document.getElementById('form:latitude');
    inputLat.value = position.coords.latitude;

    var inputLong = document.getElementById('form:longitude');
    inputLong.value = position.coords.longitude;
  }
  getLocation();
</script>
```

```
</head>
<body jsfid="body" onload="">
<form jsfid="form">
  <input type="text" jsfid="lat" placeholder="Digite a latitude"
    jsf.value="#{latLongBean.latitude}" />
  <br><br>

  <input type="text" jsfid="lon" placeholder="Digite a longitude"
    jsf.value="#{latLongBean.longitude}" />

  <button jsfid="salvar">
    Salvar
    <f: ajax execute="@form" event="click"
      listener="#{latLongBean.saveLatLong}" />
  </button>
  <br><br>
</form>
</body>
</html>
```

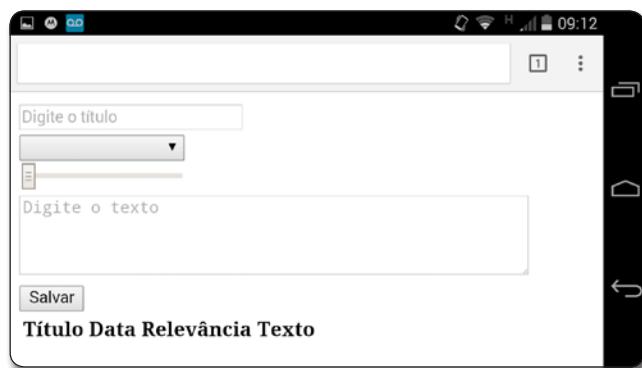


Figura 4. Tela criada com JSF e HTML 5 visualizada em um smartphone

Em seguida, quando o usuário clicar no botão *Salvar*, esses dados serão passados para o ManagedBean *latLongBean*.

Nessa última versão, o principal objetivo das novidades do JSF foi trazer ganhos em produtividade e qualidade, facilitando o trabalho de desenvolvedores e também de outros membros da equipe de desenvolvimento, como o designer e os testers que, não necessariamente, conhecem as tags do framework JSF.

A adição desses novos elementos permite também que os desenvolvedores disfrutem das vantagens originadas pela nova versão do HTML. Visando essa possibilidade, projetos que adotem versões anteriores do JSF podem ter seu código atualizado para explorar esses novos recursos. Dentre as vantagens, destacamos a adaptação do conteúdo das páginas web aos browsers dos dispositivos móveis, otimização do posicionamento em buscadores e o aperfeiçoamento da acessibilidade.

Sendo assim, recomendamos ao leitor que dê continuidade aos estudos sobre o JSF, que segue se reinventando, para assim saber empregar os seus recursos nos próximos projetos.

Autor



Luiz Henrique Zambom Santana

lhzsananta@gmail.com

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

Autor



Eduardo Felipe Zambom Santana

ezambomsantana@gmail.com

É bacharel e mestre em Ciência da Computação pela UFSCar. Possui mais de 10 anos de experiência em programação. Atualmente é professor na Universidade Anhembi Morumbi.

Links:

GitHub do autor, com os códigos apresentados no artigo.

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

Padrão HTML5 no site do W3C.

<http://www.w3.org/TR/html5/>

Implementação de referência do JSF.

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

Site oficial da tecnologia JSF.

<http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Apache JMeter: Testes de software com SoapUI

Compartilhando a responsabilidade pela garantia da qualidade entre as equipes de desenvolvimento e teste

A qualidade tem se tornado um fator de grande importância no mercado, e com isso, exigido que os testes sejam executados cada vez mais cedo dentro do ciclo de desenvolvimento de software.

Para tornar isso viável, é relevante frisar que os testes não devem ficar a cargo exclusivamente de uma equipe dedicada à sua execução. O compartilhamento da responsabilidade entre as equipes de desenvolvimento e teste, por exemplo, propiciaria a identificação ainda mais antecipada dos defeitos, indo de encontro à famosa Regra 10 de Myers, que indica que o custo da correção dos defeitos tende a ser cada vez maior quanto mais tarde ele for descoberto.

Para que a garantia da qualidade do software seja tangível, é possível seguir por diferentes caminhos incorporados às fases do ciclo de vida de desenvolvimento, a saber:

- A equipe de testes pode atuar desde o início do ciclo de vida do desenvolvimento, executando tanto os testes de Caixa Branca quanto os Teste de Caixa Preta;
- A equipe de desenvolvimento pode auxiliar nos testes, executando os testes de Caixa Branca;
- A equipe de testes pode atuar ao final do desenvolvimento do software, focando nos testes de Caixa Preta, e se houver a junção da disponibilidade de tempo com o conhecimento técnico, executar também os testes de Caixa Branca.

Para melhor esclarecer essas atividades, o Teste de Caixa Preta é aquele que tem como alvo verificar se a implementação está de acordo com o que foi especificado. Já o Teste de Caixa Branca busca garantir que o software desenvolvido esteja bem estruturado internamente; portanto, funcionando corretamente.

Dante dos cenários de teste apresentados, poder contar com o auxílio da equipe de desenvolvimento ao fazer a

Fique por dentro

Ao longo desse artigo mostraremos tanto para a equipe de desenvolvimento quanto para a equipe de testes, que trabalhar em conjunto pode ser de grande valia para o ciclo de vida do desenvolvimento de software.

A partir da utilização das ferramentas JMeter e SoapUI nas fases do Processo de Desenvolvimento, é possível executar diversos tipos de testes, sob diferentes perspectivas, sem conflitos, visando exclusivamente a obtenção da qualidade tão almejada.

junção dessas técnicas pode propiciar uma execução mais minuciosa dos testes, possibilitando que sejam descobertos diferentes tipos de erros no software.

É importante frisar que contar com os testes da equipe de desenvolvimento não exclui a necessidade da equipe de testes também executá-los; pelo contrário, podem ser realizados sob diferentes perspectivas. Isso porque enquanto normalmente o desenvolvimento se preocupa em garantir que o código funcione sob a perspectiva do programador com o caso de uso, ele nem sempre consegue e tem a bagagem necessária para pensar em todos os cenários e nuances que a equipe de testes certamente irá avaliar.

Dito isso, serão apresentadas ao longo desse artigo as ferramentas JMeter e SoapUI, tanto sob a perspectiva do desenvolvimento quanto do teste, demonstrando suas utilizações ao longo do ciclo de vida de desenvolvimento, algumas vantagens e as suas principais características.

SoapUI: o canivete Suíço dos testes

A SoapUI, ferramenta *open source* da SmartBear, é uma solução voltada para a execução de testes funcionais. Com interface gráfica simples, facilmente é possível criar e executar diferentes tipos de teste. Além disso, a ferramenta provê suporte à maioria dos protocolos e diferentes tecnologias utilizadas no mercado para desen-

volvimento de aplicações, como: web services SOAP e REST, JMS, JDBC, monitoramento HTTP, além de outras funcionalidades.

O SoapUI é uma ferramenta robusta que permite a maximização do trabalho, levando em consideração fatores como prazo e custo dos projetos, viabilizando assim um ganho expressivo para todo o processo de desenvolvimento.

Com relação à licença, estão disponíveis diferentes tipos. Uma delas é a versão gratuita, que possui recursos limitados. Além dessa, a SmartBear disponibiliza também uma versão completa e paga, possível de ser usada por 14 dias a título de demonstração. Dentro as principais características dispostas nessas versões, podemos destacar:

- Testes Funcionais;
- Testes de Performance;
- Testes de Segurança;
- Execução via linha de comando;
- Simulação de serviços – *Mocking*;
- Depuração de testes;
- Testes dirigidos a dados (*Data driven tests*);
- Análises de cobertura de testes;
- Gerenciamento de ambientes de teste.

O SoapUI apresenta uma vasta gama de possibilidades de testes, especialmente os que envolvem *web services*, que podem abranger desde a comprovação da disponibilidade dos serviços até a certificação de que todas as validações relacionadas à arquitetura de software foram efetivamente desenvolvidas.

Um pouco sobre o JMeter

Para explicar melhor essa ferramenta, apresentamos a seguir algumas das suas características básicas:

- Ferramenta desenvolvida utilizando a linguagem Java;
- Amplo apoio da comunidade, incluindo ajuda através de fóruns de discussão e tutoriais;
- Pode ser usada para testar recursos estáticos e dinâmicos, como servlets, objetos Java e scripts Perl;
- Possui uma interface gráfica que propicia uma baixa curva de aprendizado;
- Permite analisar e reexecutar os testes mesmo quando a aplicação estiver offline;
- Possibilita a simulação de um servidor Proxy, permitindo analisar e obter acesso às informações enviadas em uma requisição HTTP.

Vantagens

Utilizar o JMeter propicia uma série de vantagens ao software. Dentre elas, destacamos:

- Identificar e documentar as condições que fazem com que o sistema deixe de funcionar adequadamente;
- Verificar se há memória disponível no servidor para executar a aplicação;
- Verificar qual o número máximo de conexões possíveis e disponíveis, e com isso averiguar se as solicitações do usuário,

relacionadas à performance, serão efetivamente atendidas com o software desenvolvido;

- Verificar como a aplicação se comporta quando há pouca memória ou espaço em disco disponível.

Por que utilizar essas ferramentas?

Vale notar que estamos vivendo uma rápida evolução em diferentes áreas voltadas para tecnologia, como por exemplo, redes sociais, móveis e *cloud computing*. Com isso, testar aplicações abrangendo esses diferentes nichos de mercado se torna cada vez mais desafiante.

Os desafios enfrentados na busca pela garantia da qualidade do software podem ir desde o tradicional tempo escasso para a execução dos testes, geralmente influenciado pela rápida e contínua demanda por novos softwares, até a definição sobre qual será a abordagem adotada para a realização dos testes.

Diante desse cenário, torna-se mais recorrente e necessário o uso de ferramentas que possam apoiar a busca pela qualidade das aplicações desenvolvidas.

Para auxiliar nesse processo, SoapUI e JMeter, embutidas no ciclo de vida de desenvolvimento podem agregar, no mínimo, qualidade ao software.

Utilizando SoapUI no ambiente de desenvolvimento

Ao desenvolver aplicações, há duas análises comumente realizadas que são conhecidas e bastante difundidas no ganho de qualidade para os sistemas, as análises estática e dinâmica.

Na análise estática, os resultados ajudam o desenvolvedor a encontrar *bugs*, comportamentos inesperados da aplicação, além de possibilitar a codificação utilizando as melhores práticas de programação. Isso porque as ferramentas disponíveis no mercado, voltadas a linguagens orientadas a objetos, possibilitam identificar padrões de projeto empregados no código e até mesmo a detecção de “*Bad Patterns*”.

A título de exemplo, podemos citar a ferramenta de licença paga IBM Rational Software Architect. Através dela é possível realizar a análise estática no código e mostrar, de maneira gráfica, quais foram os “*Bad Patterns*” encontrados. Ferramentas como esta permitem que sejam configuradas métricas para identificar práticas que não devem ser utilizadas na codificação. Além disso, há ainda várias possibilidades de configuração visando a detecção de códigos não desejados, que podem ir desde a localização de uma má identação, até as verificações redundantes de comandos de decisão, como os famosos *if(true)*, *while(true)* e assim por diante.

A análise estática acontece sem a execução do software, isto é, sem a necessidade da aplicação estar no ar, através de um diagnóstico sintático do código, sendo possível certificar se houve ou não uso das melhores práticas durante a criação da aplicação.

Nesse artigo não iremos tratar da análise estática de software, e sim da dinâmica. Como descrito em várias metodologias de desenvolvimento, o software deve passar por diferentes tipos de teste, que vão desde os testes de caixa branca até os testes de caixa preta.

Geralmente, esses testes podem começar em qualquer momento do ciclo de desenvolvimento do software, inclusive no ambiente de desenvolvimento, executados pela própria equipe responsável pela codificação do software.

A execução de alguns testes nesse primeiro momento poderá ser determinante para que a aplicação seja devidamente analisada e corrigida. Em função disso, torna-se importante que esse tempo seja bem identificado e descrito no planejamento do desenvolvimento do software, pois assim o tempo de execução dos testes poderá ser melhor gerenciado, a ponto de prever e mensurar como que alguns deles serão aplicados e distribuídos ao longo do ciclo de vida de desenvolvimento.

É nesse cenário que o SoapUI se enquadra perfeitamente. Uma ampla ferramenta que, devido a sua versatilidade, é capaz de realizar uma série de verificações e testes nas aplicações.

Porém, diante do potencial da ferramenta, é comum que nem todos os recursos sejam utilizados – para exemplificar, em função de limitadores como o prazo. Portanto, é preciso dar prioridade aos testes que serão de fato pertinentes ao funcionamento da sua aplicação. Para se ter uma ideia da dimensão das possibilidades do SoapUI, serão mencionados a seguir alguns recursos da ferramenta:

- Análise de web services (também conhecidos como serviços, termo muito utilizado inclusive nas documentações de arquitetura orientada a serviços – SOA) utilizados pela aplicação:

- Verificar se o *web service* está no ar e respondendo as requisições;
- Certificar que o serviço está apropriado para uso, se não há erros na disponibilização do descriptor do serviço, conhecido como WSDL (*Web Service Description Language*);
- Averiguar se os parâmetros de entrada e saída do serviço estão de acordo com o definido no documento de arquitetura do sistema;
- Examinar se as regras descritas no caso de uso oferecem respostas sem erro de execução;
- Possibilitar a descoberta de serviços que realizam chamadas REST analisando o tráfego de um site ou Proxy (funcionalidade disponível somente na versão PRO);
- Testar a segurança em *web services* com a finalidade de averiguar se há alguma falha na disponibilização dos serviços. Por exemplo, é possível verificar se em uma URL mal formada existe a possibilidade de ataque por SQL injection – técnica que consiste em alterar os parâmetros enviados à URL utilizada para invocar o *web service*;

- Análise de JMS – Java Message Service:

- Validar se as mensagens são trocadas sem erros de execução;
- Verificar se os objetos passados nas mensagens correspondem ao que está definido no documento de arquitetura do projeto;
- Garantir ao desenvolvedor que o canal de comunicação está estabelecido e sem erro de execução.

- Análise de conexão com o banco de dados utilizando JDBC – Java Database Connectivity (disponível na versão PRO):

- Analisar se a conexão com o banco de dados foi bem sucedida;
- Executar e criar consultas SQL;

- Utilização de *plug-ins* para personalização da ferramenta:

- Possibilita que qualquer usuário avançado ou empresa possa desenvolver *plug-ins* para customização do uso da ferramenta;
- Possui uma loja que permite o download desses *plug-ins*, desenvolvidos por qualquer um que queira compartilhar as suas criações;

Há também arquivos que integram a ferramenta, por exemplo, com o NetBeans, Eclipse e IntelliJ IDEA;

- Emissão de relatórios das atividades executadas na ferramenta (disponível na versão PRO):

- O SoapUI permite que diferentes relatórios sejam extraídos a partir da utilização da ferramenta, como relatórios gerados com os resultados dos testes executados;

- Os relatórios da ferramenta podem ser extraídos em vários formatos, como PDF, ODF, RTF, HTML, XLS, CSV e XML.

Utilizando SoapUI no ambiente de teste

A ferramenta SoapUI pode ser utilizada durante todo o ciclo de criação do software, desde os testes funcionais, como o de regressão, até os estruturais, relacionados à avaliação da performance e da segurança da aplicação.

Para adotar o SoapUI no teste de software não é premissa básica ser um desenvolvedor; afinal, trata-se de uma ferramenta de usabilidade fácil, intuitiva e significativamente robusta.

Nos subtópicos a seguir vamos citar alguns dos tipos de teste viáveis através do SoapUI.

Teste funcional

O Teste Funcional, ou Teste de Caixa Preta, é o teste que visa avaliar o comportamento da aplicação através da execução de suas funcionalidades. É aquele que tem como alvo verificar se a implementação está de acordo com o que foi especificado.

Através do SoapUI é possível executar inúmeras técnicas de teste funcional, cada uma ao seu modo, possibilitando uma cobertura mais ampla. Dentre as técnicas disponíveis, destacamos o Teste de Regressão, que ocorre a cada modificação realizada no software. O objetivo é evitar que novos defeitos sejam introduzidos na aplicação.

Testes de desempenho

É projetado para testar o desempenho do software durante a sua execução, visando descobrir situações que levem a uma possível falha. É um tipo de teste destinado a determinar o tempo de resposta, a confiabilidade e a escalabilidade da aplicação sob uma determinada carga de trabalho.

O Teste de Desempenho deve ocorrer implicitamente ao longo de todo o Processo de Teste, porém, só pode ser finalizado efetivamente quando todos os componentes da aplicação estiverem integrados.

O objetivo dele é eliminar possíveis gargalos e estabelecer uma base para os testes de regressão futuros. Ou seja, visa tornar

a aplicação estável, de modo que o Processo de Teste definido possa prosseguir sem maiores problemas. Permite também que o software seja cuidadosamente controlado através da análise dos resultados apresentadas após a execução do teste.

Testes de segurança

O Teste de Segurança tem como meta garantir que o software se comporte adequadamente mediante as mais diversas tentativas ilegais de acesso. Para isso, verificar se todos os mecanismos de proteção embutidos na aplicação de fato a protegerão de acessos indevidos.

É muito comum que as aplicações se tornem alvo de sujeitos que buscam provocar ações que possam prejudicar ou, até mesmo, beneficiar pessoas. Em função de situações como estas, o Teste de Segurança propõe demonstrar se a aplicação faz exatamente o que deve fazer ou se a aplicação não faz o que não deve ser feito.

A execução do Teste de Segurança possibilita que dúvidas sobre prováveis vulnerabilidades do software sejam sanadas. Pode auxiliar também na definição de um plano de contingência, visando determinar qual precaução será tomada contra os possíveis ataques. Além disso, o Teste de Segurança também tem como objetivos:

- Validar os requisitos do software;
- Identificar as funcionalidades críticas de segurança;
- Analisar o risco para determinar todos os fatores que contribuem para sua ocorrência;
- Categorizar o risco em termos de gravidade e probabilidade de ocorrência.

O Teste de Segurança visa garantir a confidencialidade, integridade e disponibilidade das informações tratadas pelo software, sendo que cada um desses itens possui a seguinte definição:

- **Confidencialidade:** é a propriedade que garante se a informação estará disponível apenas para aqueles usuários devidamente autorizados;
- **Integridade:** é a propriedade que garante que, independentemente da situação, a informação não será destruída ou corrompida, permitindo que o software se mantenha com um desempenho próximo ao que foi definido pelo cliente;
- **Disponibilidade:** é a propriedade que garante se o software estará disponível sempre que for necessário, havendo ou não situações anormais de utilização.

Utilizando REST ou SOAP no SoapUI

O SoapUI propicia a execução de diferentes tipos teste em diferentes momentos do ciclo de vida de desenvolvimento do software. Possibilita também que sejam criados projetos de teste baseados no protocolo SOAP ou na arquitetura REST. Para melhor compreender a diferença entre esses tipos de projeto, apresentamos a seguir algumas definições:

- **SOAP:** é um protocolo de transferência de mensagens que determina que o formato utilizado na mensagem enviada seja o XML. Sua aplicação é indicada para uso em ambientes distri-

buídos. O SOAP permite que a interoperabilidade entre diversas plataformas seja possível, desde que o formato obrigatório (XML) seja utilizado. Ao adotar SOAP no desenvolvimento de *web services* é necessário fazer uso de WSDL (*Web Services Description Language*), uma linguagem baseada em XML utilizada para descrever a estrutura das mensagens;

- **REST:** é uma arquitetura cujo protocolo de comunicação é o HTTP. O REST não impõe restrições quanto ao tipo da mensagem e, por isso, a sua maior vantagem é a flexibilidade quanto à escolha da padronização das mensagens, possibilitando que o desenvolvedor opte pelo formato mais adequado. Os formais mais comuns para essa arquitetura são: JSON, XML e texto puro, mas em tese, qualquer um pode ser adotado.

Utilizando o JMeter em ambientes de desenvolvimento e teste

O JMeter tem por finalidade a execução de testes de performance, seja no ambiente de desenvolvimento ou no ambiente de teste. Através dessa ferramenta é possível avaliar resultados de maneira simples, por exemplo, o status das requisições com determinados números de usuários simultâneos.

Os resultados são obtidos de acordo com o componente da ferramenta selecionado para realizar o teste. Para melhor compreensão, segue uma breve descrição de alguns deles:

- **Plan:** é o elemento fundamental para execução de um teste. Tem como objetivo agrupar todos os outros elementos, além de controlar a execução das *Schedule Thread Groups*;
- **Schedule Thread Group:** é o componente responsável por controlar a execução dos testes;
- **Logic Controllers:** os controladores lógicos permitem definir a ordem de solicitação de processamento em uma *Thread*, ou seja, determinam a ordem em que as solicitações dos usuários serão executadas. Por exemplo:

- **Recording Controller:** o JMeter permite a gravação dos testes. Esse componente viabiliza organizar o armazenamento dessas gravações;
- **Simple Controller:** tem como objetivo agrupar os elementos, visando a organização;
- **Throughput Controller:** visa decidir, aleatoriamente, o número absoluto máximo de execuções de determinado grupo de componentes;
- **Random Order Controller:** permite que os testes elaborados sejam executados aleatoriamente.
- **Timers:** deve ser utilizado quando é necessário realizar pausas durante a execução dos testes. Por padrão, o JMeter faz requisições sem pausas;
- **Samplers:** são controladores pré-definidos para requisições específicas. Podem ser customizados através da inserção de *Configuration Elements* e *Assertions*;
- **Assertions:** recurso utilizado para verificar se a resposta obtida na requisição é a esperada;
- **Configuration Elements:** embora não façam requisições (exceto para HTTP Proxy Server), são utilizados para adicionar ou modificar as solicitações feitas pelos *Samplers*.

• **Listeners:** os chamados “ouvintes” funcionam como relatórios. São responsáveis por fornecer acesso às informações obtidas através da execução dos testes pelo JMeter. São eles que permitem a análise visual dos resultados alcançados, seja através de gráficos ou tabelas.

JMeter, analisando a performance das aplicações

Sabemos que, geralmente, o computador utilizado para desenvolvimento de um

software possui uma configuração inferior às máquinas do ambiente de produção. Diante disso, para avaliar o possível comportamento de uma aplicação, se faz necessário simular algumas situações de acordo com o escopo desejado. Por exemplo, um aplicativo com a finalidade de gerenciar os livros de uma escola poderá sofrer alguns picos de utilização e talvez até alguns acessos simultâneos.

Mesmo que seja desconhecida a capacidade total da demanda, vamos supor que,

hipoteticamente haverá um pico de 10 usuários simultâneos realizando o empréstimo de livros. Vamos pressupor ainda que cada evento possua uma quantidade de 15 livros emprestados e que a aplicação seja executada em um sistema *mobile*.

Dessa maneira, conseguimos chegar a um possível cenário com um mínimo de dados básicos que a aplicação precisará para que seja possível certificar sua escalabilidade e disponibilidade:

- 10 usuários simultâneos;
- 15 livros emprestados por evento;
- Desse modo, considerando que cada usuário é responsável por um evento, serão 150 livros emprestados.

O objetivo do teste de performance será analisar se a aplicação está preparada para atender os usuários que utilizarem a aplicação, bem como realizar os empréstimos. Caso os valores sejam suportados, a aplicação estará apta para ser disponibilizada no ambiente de produção.

Ao abrir o JMeter, antes mesmo de qualquer intervenção, a ferramenta exibe dois itens: o *Plano de Teste* e a *Área de Trabalho*, ambos localizados à esquerda, conforme apresenta a **Figura 1**.

O primeiro passo, de acordo com a **Figura 2**, é adicionar um *Grupo de Usuários*, clicando com o botão direito em cima do item *Plano de Teste* e selecionando a opção correspondente.

O objetivo do *Grupo de Usuários* é definir a parametrização dos testes. Conforme a **Figura 3**, em nosso exemplo serão definidos:

- **Número de usuários virtuais (threads):** 10. Nesse campo deve ser informado o número de usuários simultâneos que serão simulados no teste;
- **Tempo de inicialização (em segundos):** 1. É o tempo em que os usuários virtuais (*threads*) serão inicializados. Por exemplo, se informarmos no campo *Número de usuários virtuais* “10” *threads* e o *Tempo de inicialização* estiver definido como “1”, cada *thread* será inicializada a cada 1 segundo;
- **Contador de Iteração: 100.** É onde é informada a quantidade de vezes que os testes serão executados. Caso o objetivo seja executar os testes infinitamente,

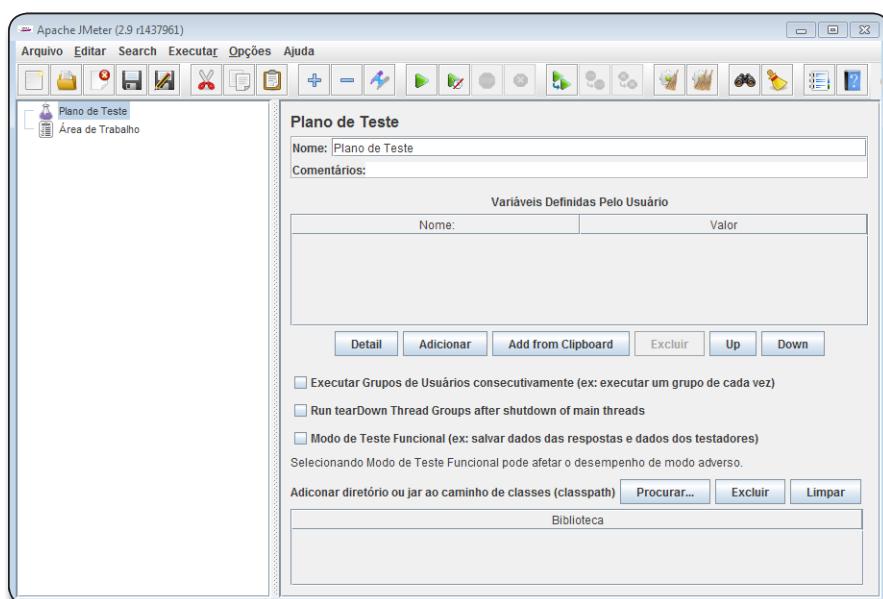


Figura 1. Tela inicial do JMeter

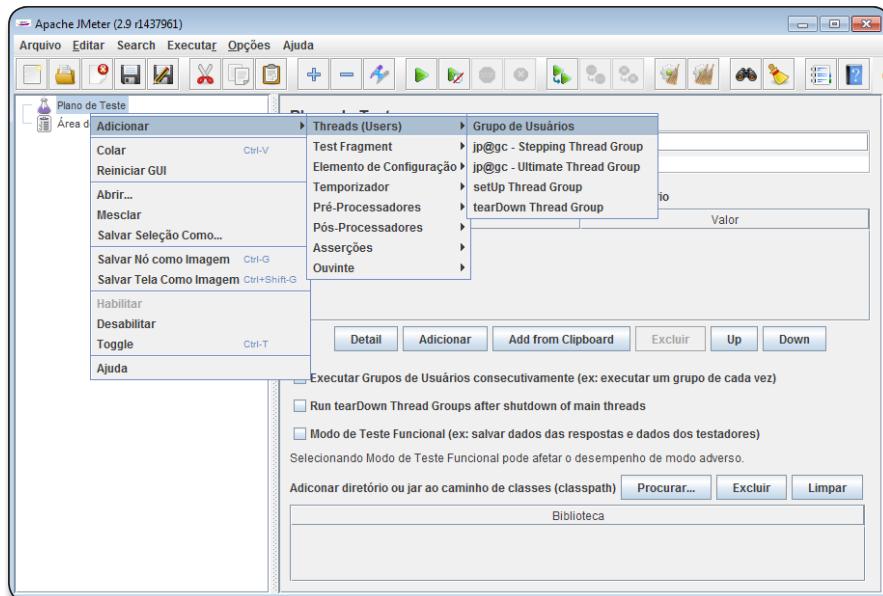


Figura 2. Inserindo um Grupo de Usuários no Plano de Teste do JMeter

basta selecionar o *checkbox* com a opção *Infinito*.

Para dar continuidade ao nosso exemplo, após adicionar o *Grupo de Usuários*, o próximo passo é a inserção de uma *Requisição HTTP*, acionando o botão direito do mouse no item *Área de Trabalho > Adicionar > Testador > Requisição HTTP*, como expõe a **Figura 4**.

Os *Testadores* são controladores pré-definidos para executar requisições específicas. Para exemplificar, segue uma breve descrição sobre alguns dos tipos:

- **Requisição HTTP:** utilizado em nosso exemplo, tem como objetivo permitir que uma solicitação HTTP ou HTTPS seja enviada ao servidor web;
- **Requisição FTP:** possibilita criar requisições usando o protocolo FTP (com autenticação ou não);
- **Requisição LDAP:** permite enviar requisições para um servidor LDAP.

Após adicionar a *Requisição HTTP* é possível especificar manualmente informações detalhadas sobre o teste que está sendo elaborado, por exemplo: o nome do servidor, o caminho da requisição e o número da porta, como demonstrado na **Figura 5**.

É possível também capturar as requisições de um navegador. Para isso, deve ser adicionado o componente *Servidor HTTP Proxy*, que pode ser acessado ao clicar com o botão direito do mouse na Área de Trabalho da ferramenta e navegar pelo caminho: > *Adicionar > Elementos que não são do teste > Servidor HTTP Proxy*.

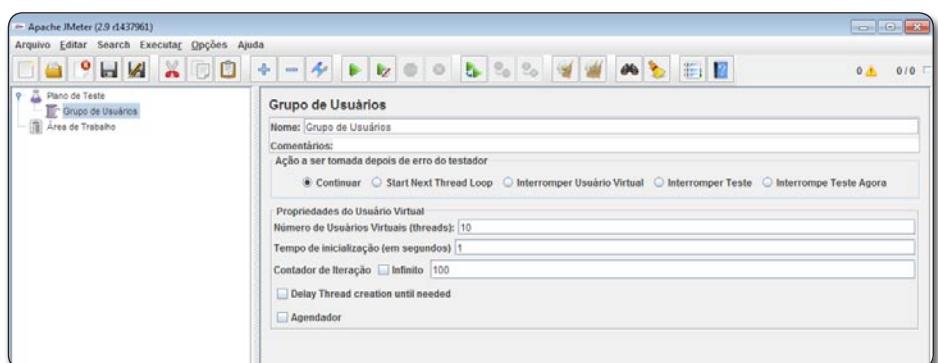


Figura 3. Parametrizando o Grupo de Usuários

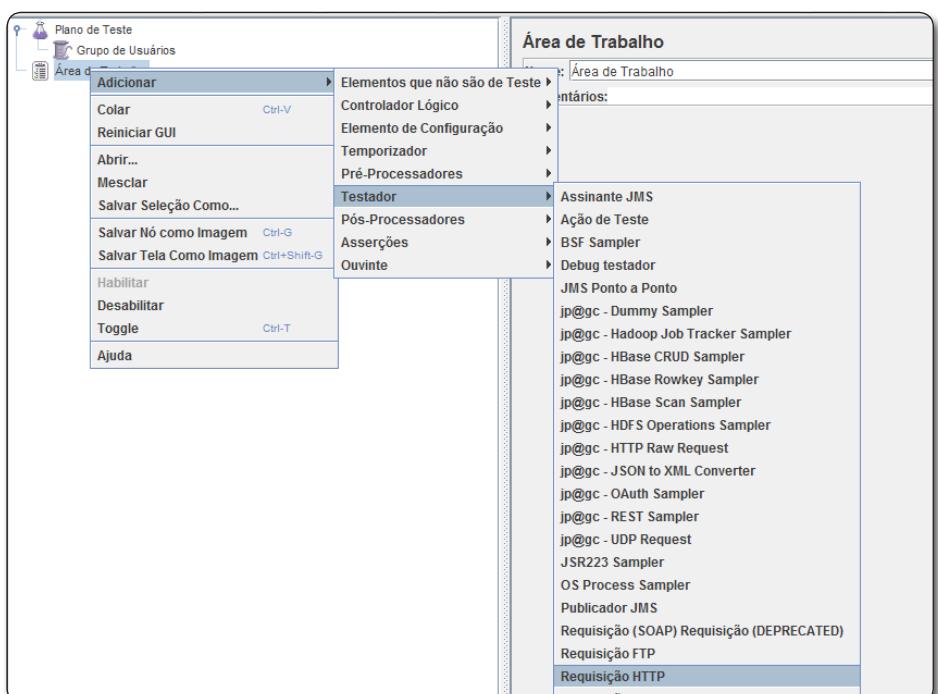


Figura 4. Inserindo uma Requisição HTTP na Área de Trabalho

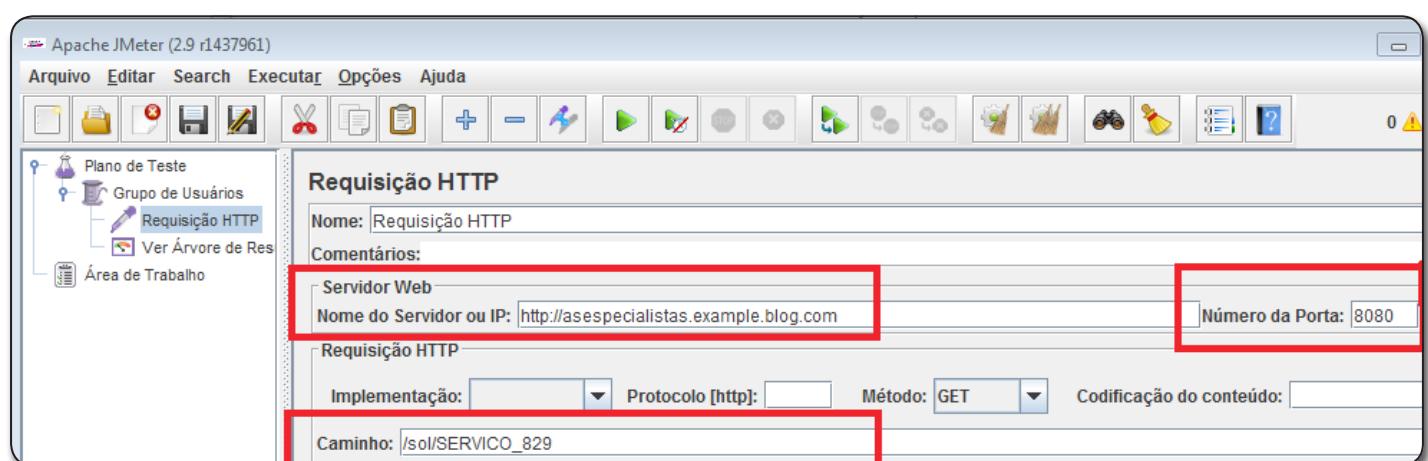


Figura 5. Inserindo os dados na Requisição HTTP

Apache JMeter: Testes de software com SoapUI

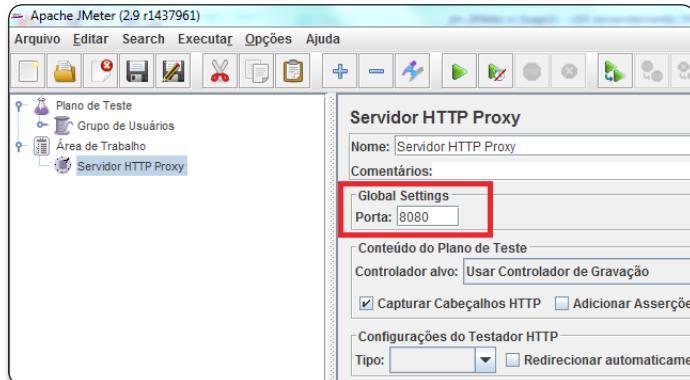


Figura 6. Informando a porta do Servidor HTTP Proxy

E a partir daí, informar uma porta que servirá para comunicação entre a ferramenta e o navegador, como expõe a Figura 6.

O *Servidor HTTP Proxy* é considerado um componente Non-Test, ou seja, não é usado na execução dos testes; ele apenas apoia na gravação, e tem como objetivo capturar todas as ações, visando agilizar a realização dos testes.

O passo seguinte é configurar o navegador, apontando para o caminho “HTTP” do proxy e para a “Porta” que foi informada no *Servidor HTTP Proxy*, para viabilizar a comunicação entre o JMeter e o Browser, conforme apresentado na Figura 7.

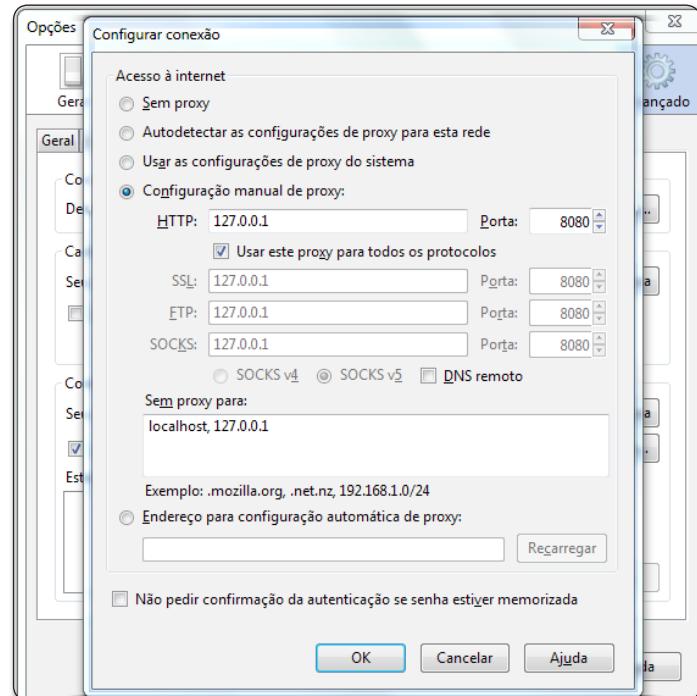


Figura 7. Configurando o Proxy no Navegador

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

Após as configurações no browser, a próxima etapa é acionar o comando *Iniciar* no componente *Servidor HTTP Proxy*. Desse modo, ao acessar a aplicação através do navegador, todas as ações e requisições serão capturadas (veja a **Figura 8**), ou seja, todas as chamadas HTTP realizadas serão gravadas.

A partir desse momento, seja adicionando uma Requisição HTTP ou um Servidor HTTP Proxy, a próxima etapa é adicionar um Ouvinte, componente que tem como objetivo apresentar os resultados da execução dos testes. Os resultados podem ser exibidos em árvores, tabelas, gráficos ou simplesmente gravados em um arquivo. Para isso, é necessário adicionar ao *Plano de Teste* um dos *Ouvintes* disponíveis. Cada um, à sua maneira, apresentará o resultado de uma forma diferenciada.

Para o nosso caso, será utilizado o tipo *Ver Árvore de Resultados*, conforme demonstra a **Figura 9**. Esse *Ouvinte* exibe em uma árvore o resultado detalhado do teste executado, de acordo com cada item da requisição. Ele fornece também uma funcionalidade de pesquisa, que permite procurar rapidamente as respostas dos itens mais relevantes.

Dante de toda a parametrização realizada, o último passo é executar o teste elaborado, acionando a seta verde na barra de comandos da ferramenta. Feito isso, será possível visualizar o resultado nos ouvintes que definimos, conforme demonstra a **Figura 10**.

Para melhor avaliar o resultado do teste, além dos ouvintes (relatórios), é interessante observar o comportamento do servidor onde está hospedada a aplicação, para que finalmente seja possível responder ao questionamento inicial, ou seja, se diante do cenário apresentado é possível que 10 usuários simultâneos consigam efetuar o empréstimo de 150 livros.



Figura 8. Exemplo de captura do Servidor HTTP Proxy

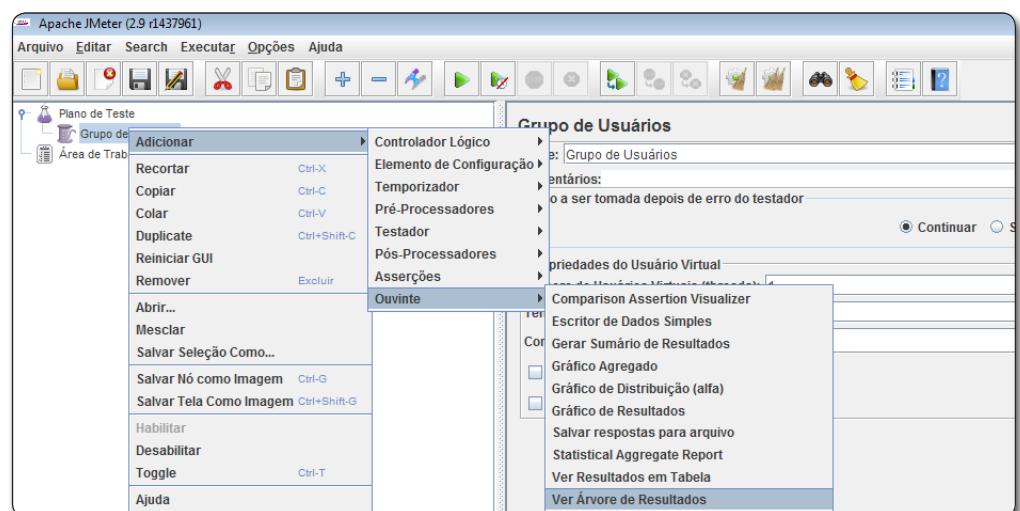


Figura 9. Inserindo um Ouvinte

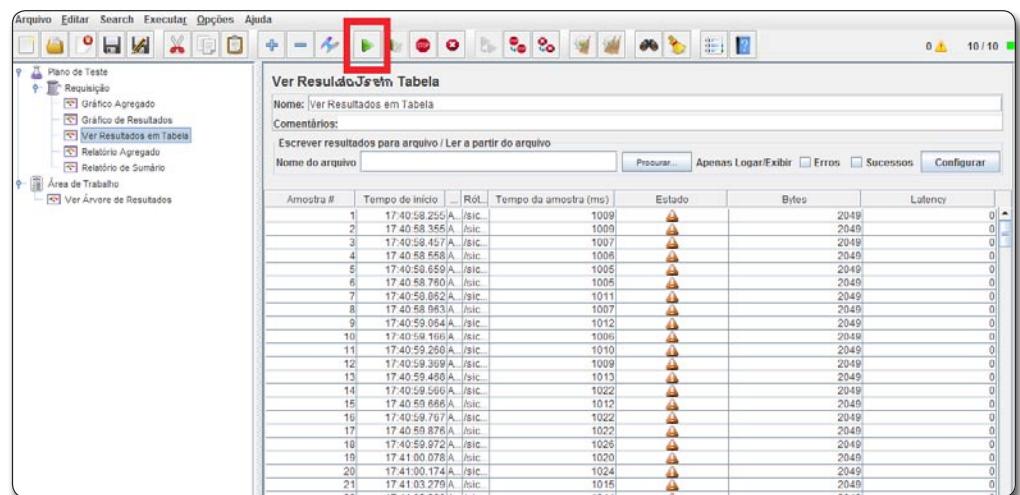


Figura 10. Resultado do teste

Apache JMeter: Testes de software com SoapUI

Pregar a utilização de ferramentas destinadas a testar aplicações para a equipe de desenvolvimento não é nenhum absurdo. Afinal, qualquer ação voltada para incorporar a responsabilidade pela qualidade do software em todas as áreas envolvidas no ciclo de vida de desenvolvimento deste é de grande valia.

Seria fantástico se todo desenvolvedor tivesse em mente a importância de garantir que o código entregue ao teste pelo menos funcione. Ao realizar um teste simples antes de disponibilizar a aplicação, muitos problemas podem ser evitados, como retrabalho, cronograma apertado e produtividade baixa.

Outro problema que pode ser evitado, caso a equipe de desenvolvimento se predisponha a garantir a qualidade do software, é a performance. Um projeto de teste que utiliza o JMeter poderia, com certa facilidade, detectar grandes disparidades entre o que foi especificado e o que foi desenvolvido, o que faria valer a pena qualquer tempo gasto nessa etapa.

Para finalizar com os exemplos, imagine o tempo que seria economizado se o desenvolvimento entregasse à equipe de testes *web services* ativos e funcionais? Note que para alcançar tal feito, bastaria a elaboração de um simples projeto de teste utilizando o SoapUI.

Trabalhando sob diferentes perspectivas e sendo utilizados tanto no desenvolvimento quanto no teste, o JMeter e o SoapUI podem ter diferentes funções e finalidades, mas em suma, um objetivo único: a garantia da qualidade do software entregue ao usuário.

Autora



Fabiana Alencar

fabianabyte@gmail.com - @FabianaByte

Trabalha no desenvolvimento de aplicações Java há mais de oito anos. Bacharel em sistemas de informação, ITIL V3 Foundation.



Autora



Renata Eliza

renataeliza@gmail.com - @RenataEliza

Atua na área de Teste de Software há mais de oito anos.

Tecnóloga em Processamento de Dados, MBA em Teste de Software e ISTQB Certified Tester Foundation Level.

Mantém o blog www.asespecialistas.blog.com.



Links:

Automação de testes em web services com SoapUI.

<http://www.qualister.com.br/blog/automacao-de-testes-em-webservices>

Destilando JMeter I: Introdução e Conceitos.

<http://www.bugbang.com.br/destilando-jmeter-i-introducao-e-conceitos/>

Loadrunner vs JMeter – feature comparison.

<http://www.perftesting.co.uk/loadrunner-vs-jmeter-feature-comparison/2012/06/11/>

Transferência de estado representacional – REST – via HTTP.

<http://javascriptbrasil.com/2012/03/29/transferencia-de-estado-representacional-rest-via-http/>

WSDL – O que é? Pra que serve? Onde utilize?

<http://fabriciosanchez.com.br/2/wsdl-o-que-e-pra-que-servi-onde-utilizo/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Como trabalhar com tarefas assíncronas em Java EE

Aprenda a utilizar os recursos assíncronos da plataforma Java EE

ESTE ARTIGO É DO TIPO MENTORING

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

A comunicação entre sistemas e métodos, dentro da área da computação, pode ser considerada um dos tópicos mais importantes na arquitetura e desenvolvimento de software, propiciando inúmeras alternativas para a integração de aplicações. Essas alternativas, por sua vez, se dividem em dois paradigmas principais, assíncrono e síncrono, determinando o comportamento que essa comunicação deve ter.

O uso correto de cada um desses paradigmas é de extrema importância, ainda mais dentro de uma aplicação corporativa. Sendo devido aos ganhos de performance ou pela confiabilidade de resposta, a utilização do paradigma correto, de acordo com sua necessidade, é um fator importantíssimo na hora de criar um sistema ou uma nova funcionalidade.

Além disso, a utilização desses dois tipos em conjunto permite uma maior paralelização do código, podendo diminuir o impacto de tarefas demoradas e custosas em sua aplicação, trazendo uma maior qualidade ao software ao final do desenvolvimento.

O processamento assíncrono é comumente empregado por desenvolvedores através do uso da interface **Runnable**, que permite a criação de threads em paralelo, ou através de beans de mensagens, os MDBs. Essas estratégias, apesar de extremamente úteis, muitas vezes não atingem o paralelismo desejado devido à dificuldade de implementação e a alta complexidade do código, características indesejadas para qualquer solução.

Fique por dentro

Este artigo abordará os recursos disponibilizados pela Java EE para executar métodos em EJBs assíncronos. Esse tipo de estratégia permite que os desenvolvedores consigam criar processos independentes uns dos outros e também viabiliza uma maior paralelização da carga de trabalho dentro de sua aplicação. Portanto, esse artigo é muito importante para desenvolvedores que buscam melhorar o desempenho de suas soluções e entender um pouco mais as ferramentas que o Java disponibiliza para executar tarefas concorrentes e distribuídas.

Com o intuito de apresentar uma solução prática da utilização desses recursos, iremos demonstrar uma situação real onde métodos assíncronos se aplicam, abordando as alternativas que temos para conseguir executar essas tarefas.

Portanto, pretendemos demonstrar nesse artigo as vantagens e desvantagens de utilizarmos uma abordagem mais comum, e também apresentar os recursos que a especificação da Java EE 6 trouxe para o desenvolvimento e criação de tarefas e EJBs assíncronos, explicando o motivo desses recursos serem mais adequados na implementação de um processo desse tipo.

Além disso, iremos colocar essas ideias em prática, analisando uma situação real em que a utilização de EJBs assíncronos se faz necessária e mostrar os principais ganhos desse tipo de abordagem, aplicando ela dentro de um projeto exemplo que iremos desenvolver ao longo do artigo.

Os paradigmas e suas diferenças

Antes de começarmos a desenvolver esses exemplos, precisamos entender a diferença de cada um dos paradigmas de comunicação apresentados anteriormente e suas principais diferenças, fatores

Como trabalhar com tarefas assíncronas em Java EE

essenciais para sabermos quando e onde devemos empregar cada um deles.

Iniciando pelo paradigma síncrono, esse tipo de comunicação determina que o método chamador deve aguardar a execução do método chamado antes de continuar a execução de seu código. Esse tipo de comunicação é o tipo padrão de chamadas de método em Java e acontece toda vez que um método chama outro para executar.

O funcionamento desse tipo de chamada, por sua vez, pode ser explicado através do conceito de **pilha de execução** ou **stack** da thread. Cada vez que um método é chamado, suas variáveis, parâmetros e classes são empilhados e, uma vez que esse mesmo método chega ao final de sua execução, esses objetos que foram empilhados são desempilhados, passando a execução ao método chamador, ou seja, ao método que estava na posição inferior da pilha.

Na **Figura 1** apresentamos uma representação gráfica de como funciona a pilha de execução de métodos síncronos. Conforme podemos observar, o exemplo em funcionamento é baseado em três métodos. O primeiro deles, o método **main()**, começa sua execução ao iniciar a thread e é colocado na pilha. Sua execução continua até a chamada do **metodoA()**, quando o processo do método **main()** é interrompido, o **metodoA()** é colocado no topo da pilha (neste caso, uma posição acima) e inicia seu processamento.

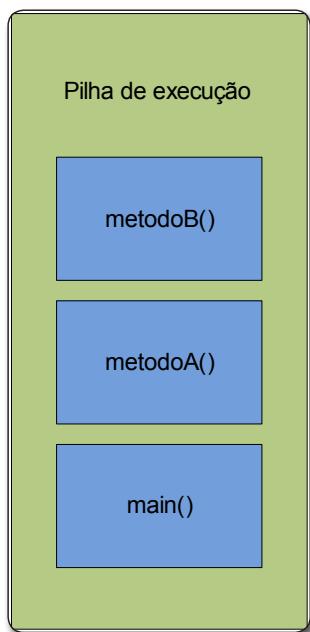


Figura 1. Exemplo do funcionamento da pilha de execução

Ao chamar o **metodoB()**, o **metodoA()** também tem sua execução interrompida, dando início à execução do **metodoB()**, que, no estado atual do exemplo descrito, é o método que está ocupando o processador.

Por fim, analisando o exemplo da figura, é fácil perceber que sempre o método no topo da pilha é o que está sendo executado

pela JVM, enquanto os outros ficam aguardando o fim da execução dos métodos acima. Também é interessante ressaltar que, para cada thread, a JVM cria uma pilha de execução, ou seja, podemos ter múltiplas threads rodando em paralelo dentro de nossa JVM.

Comunicação assíncrona dentro do Java

Esse modelo síncrono, apesar de ser o mais utilizado e atender quase todas as necessidades de um sistema, traz algumas limitações. A primeira delas é o fato de que, uma vez que cada método aguarda o outro para executar, o tempo de execução de um processo síncrono se torna muito mais lento que de um processo assíncrono.

Além disso, fica muito mais complicado atingirmos um paralelismo eficiente a partir de chamadas de métodos síncronos, uma vez que essa espera dificulta a execução de processos ao mesmo tempo.

Visando resolver esses problemas, a comunicação assíncrona entre métodos permite que um método chame o outro, do mesmo modo que a forma síncrona, porém, permite que a execução do método chamador continue em paralelo com a do método chamado, ou seja, a chamada de outro método não impede a execução do método corrente.

Na **Figura 2** apresentamos a abordagem mais tradicional, dentro do Java, para a criação de processos assíncronos: a criação de uma **thread** separada para cuidar da execução assíncrona.

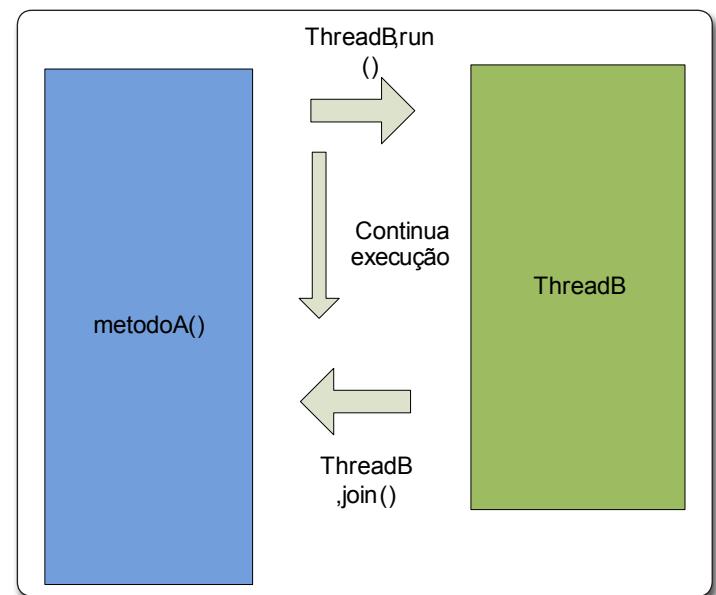


Figura 2. Exemplo de thread para comunicação assíncrona

No exemplo da comunicação síncrona, cada thread possui sua pilha de execução. Portanto, para atingir uma comunicação assíncrona, a forma mais simples de iniciar um novo processo em paralelo ao corrente é iniciar uma nova thread.

Nesse exemplo, vemos que o **metodoA()** inicia uma thread a partir da classe denominada **ThreadB**, que implementa a interface **Runnable**. Ao chamar o método **run()** dessa classe, uma nova thread juntamente com sua pilha de execução, é alocada na JVM e iniciada em paralelo, independente da pilha da thread em que o **metodoA()** se encontra.

Esse tipo de comunicação permite que o **metodoA()** possa executar outras tarefas enquanto a **ThreadB** executa, e quando necessitar dos dados que estão sendo processados por essa thread, recuperar os resultados chamando o método **join()**. Dessa forma atingimos uma comunicação independente e totalmente assíncrona entre ambos os processos.

Quando usar cada um?

Agora que introduzimos os dois paradigmas de comunicação que queremos focar nesse artigo, precisamos entender quando e como utilizar cada um deles.

De um lado, a comunicação síncrona traz uma proposta mais simplificada e com uma confiabilidade maior, tanto no fluxo de execução como no tratamento de erros, uma vez que, por se tratar de um fluxo único de execução, não existem muitos cenários alternativos que exigem uma atenção extra do desenvolvedor na hora de sua execução.

Porém, ainda assim, existem cenários que esse tipo de abordagem simplesmente não é suficiente. Sistemas de grande porte sempre trazem consigo tarefas pesadas e custosas computacionalmente e, com isso, a necessidade de uma paralelização de código e criação de processos batches.

Em casos como esse, podemos utilizar o paradigma assíncrono e a criação de tarefas paralelas para nos ajudar a criar um ambiente mais performático e confiável. No entanto, cada situação exige uma abordagem diferente. Assim, pretendemos demonstrar nos próximos tópicos a diferença entre cada uma das abordagens mais populares para a criação de tarefas assíncronas e, também, apresentar os novos recursos da Java EE 6 que trouxeram excelentes soluções para esse tipo de problema.

Criando nosso projeto exemplo

Como base para exemplificar essas situações, vamos começar criando um projeto exemplo que servirá de referência para demonstrar as principais técnicas de comunicação assíncrona.

O primeiro passo para isso é realizar o download de um servidor capaz de prover todas as bibliotecas da Java EE 6, permitindo o uso das novas funcionalidades assíncronas que essa versão provê. Como escolha para esse artigo, decidimos adotar o servidor WildFly, considerado o sucessor do famoso JBoss AS (veja o endereço para download na seção **Links**).

Uma vez feito o download do WildFly, para realizar a instalação basta descompactá-lo em alguma pasta de seu computador e, dentro de sua IDE, criar a configuração de um novo servidor e identificar, dentro das propriedades, o caminho do diretório em que os arquivos foram descompactados. No caso de nosso exemplo, iremos utilizar a IDE Eclipse Kepler, juntamente

com o plugin do JBoss Tools, para podermos configurar nosso servidor.

Por padrão, o servidor WildFly não possui todas as funcionalidades da Java EE, como os MDBs e outros recursos, habilitados. Como em nosso exemplo iremos demonstrar um pouco do uso de MDBs, precisamos alterar as configurações de nosso servidor para permitir que, uma vez ele seja iniciado, esse tipo de tecnologia esteja disponível.

Nota

A instalação e configuração do plugin do JBoss, juntamente com a configuração do servidor na IDE, podem ser visualizadas com mais detalhes no endereço indicado na seção **Links**.

Para isso, na pasta onde foi descompactado o WildFly, exclua o arquivo **standalone.xml**, que possui o perfil padrão do WildFly, e renomeie o arquivo **standalone-full.xml**, que determina um perfil em que todas as funcionalidades da Java EE estão habilitadas, para **standalone.xml**. Dessa forma o servidor irá utilizar o arquivo correto na sua inicialização e não teremos problemas com nenhuma das tecnologias que utilizaremos.

Configurado o servidor, podemos agora criar o nosso projeto na IDE. Para isso, basta clicarmos em **New > Dynamic Web Project**, nomearmos nosso projeto e escolher, como runtime de nossa aplicação, o servidor recém-configurado do WildFly 8. Na **Figura 3** mostramos a configuração do projeto exemplo.

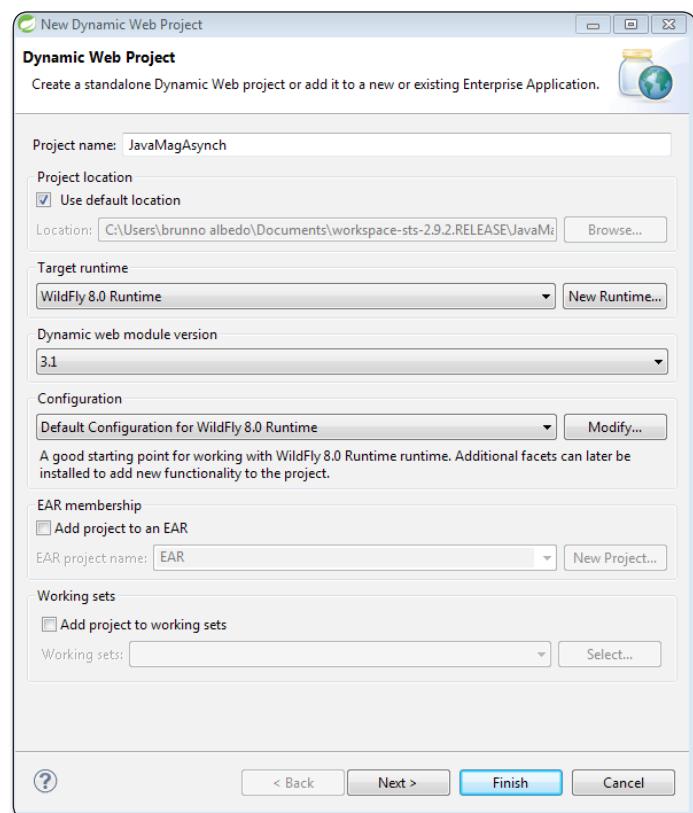


Figura 3. Configuração de nosso projeto

Como trabalhar com tarefas assíncronas em Java EE

Estratégias para criação de tarefas assíncronas

Uma vez criado o projeto, podemos iniciar a discussão sobre algumas das estratégias mais utilizadas por desenvolvedores para implementar tarefas assíncronas em seus projetos, exemplificando as vantagens e desvantagens de cada uma delas.

O uso de threads

A primeira das estratégias que iremos discutir é o uso da interface **Runnable** para atingir um comportamento assíncrono. Esse tipo de abordagem é uma das soluções mais primitivas para se conseguir um efeito assíncrono, uma vez que, ao implementar essa interface, a classe Java permite que seu método seja invocado em uma thread separada da thread corrente.

Graças a isso, conforme exemplificamos anteriormente, os métodos invocados por essa nova thread serão colocados em uma pilha de execução à parte e, consequentemente, serão executados assincronamente.

Na **Listagem 1** apresentamos um exemplo da classe **ExemploRunnable**, que implementa a interface **Runnable**, e onde iremos imprimir uma mensagem na tela após um período de espera de 10 segundos.

Como sabemos, a interface **Runnable** exige que o método **run()** seja implementado. Esse método será responsável por criar, quando invocado, uma nova thread Java independente da execução da thread que originou a requisição.

Para executarmos esse exemplo, basta rodarmos o método **main()**, também descrito na classe **ExemploRunnable**, que irá executar a comunicação assíncrona com o método **run()** de nossa classe. Para observarmos o comportamento dessa comunicação, basta analisarmos a saída de dados no console, que demonstra que a thread do método **main()** termina antes que a thread que iniciamos no método **run()**. Entre as mensagens impressas, esse comportamento é visto na mensagem “Terminei a thread principal”, impressa antes das linhas “Terminou de rodar a thread separada!”, o que indica claramente que uma thread se torna independente da outra.

Identificando o fim da execução das threads

Além da possibilidade de executar outras threads de forma assíncrona, a interface **Runnable** permite que, ao iniciarmos uma thread, seja possível identificar quando esse processo em paralelo termina: através do método **join()**.

Isso possibilita que, uma vez que o método executado de forma assíncrona termine, a thread original, que também está executando, possa identificar essa conclusão e utilizar os dados processados.

Na **Listagem 2** demonstramos a aplicação do método **join()** da classe **Thread**, implementado justamente para esse tipo de comportamento. Nesse exemplo, criamos a classe **ExemploRunnable**, onde iniciamos diversas threads e, logo em seguida, recuperamos os resultados de seus processamentos com o uso desse método.

A implementação dessa classe demonstra como podemos usar o método **join()** para que o processamento principal aguarde a execução de uma thread, garantindo que seu processamento só continue após a thread na qual o **join()** foi invocado termine.

Listagem 1. Código da classe **ExemploRunnable**, demonstrando o uso da interface **Runnable** para criação de threads.

```
public class ExemploRunnable implements Runnable{  
    @Override  
    public void run() {  
        try {  
            System.out.println("Rodando a thread separada!");  
            Thread.sleep(10000);  
            System.out.println("Terminou de rodar a thread separada!");  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Iniciando a thread principal!");  
        new Thread(new ExemploRunnable()).start();  
        System.out.println("Iniciei a primeira thread separada!");  
        new Thread(new ExemploRunnable()).start();  
        System.out.println("Iniciei a segunda thread separada!");  
        System.out.println("Terminei a thread principal");  
    }  
}
```

Listagem 2. Exemplo de uso do método **join()**.

```
public class ExemploRunnable implements Runnable{  
    private static int contador=0;  
  
    @Override  
    public void run() {  
        try {  
            System.out.println("Rodando a thread separada!");  
            contador++;  
            Thread.sleep(10000);  
            System.out.println("Terminou de rodas a thread separada!");  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread = new Thread(new ExemploRunnable());  
        Thread thread2 = new Thread(new ExemploRunnable());  
        System.out.println("Iniciando a thread principal!");  
        thread.start();  
        System.out.println("Iniciei a primeira thread separada!");  
        System.out.println("Esperando a primeira thread terminar");  
        thread.join();  
        System.out.println("O contador eh "+ contador);  
        System.out.println("Esperando a segunda thread terminar");  
        thread2.join();  
        System.out.println("O contador eh "+ contador);  
        System.out.println("Terminei a thread principal");  
    }  
}
```

Sendo assim, ao executar o método **main()** dessa classe, podemos observar no console que as mensagens serão impressas conforme as threads iniciadas forem terminando de executar, permitindo com isso um maior controle do fluxo de execução e uma garantia de que os processamentos executados realmente terminaram seu processamento.

Porém, apesar de todas essas garantias, o uso de threads puras em Java traz duas grandes dificuldades: a primeira delas está relacionada à implementação do comportamento assíncrono e o paralelismo, exigindo um cuidado extremo do desenvolvedor ao tratar exceções. Devido a isso, em casos muito complexos, em que um alto nível de confiabilidade é necessário fazer o uso de threads puras em Java se torna inviável.

Além disso, a utilização de recursos compartilhados entre threads se torna bastante complexo, uma vez que se faz necessário a implementação de variáveis estáticas ou a passagem de parâmetros para que uma thread possa acessar o resultado e dados de outra. Isso, tanto do ponto de vista prático, como do ponto de vista do design do código, não é nada prático, podendo trazer uma dificuldade ainda maior ao time de desenvolvimento na hora de realizar alguma manutenção.

Segurança e confiabilidade com o uso de MDBs

Caso o desenvolvimento com o uso de threads puras não seja feito com muito cuidado, podemos ter diversos problemas de confiabilidade. Isso se deve pela própria natureza de um processo assíncrono, onde a tarefa iniciada em paralelo tem um fluxo totalmente independente da tarefa principal. Dessa forma, qualquer erro que aconteça na tarefa paralela se torna praticamente invisível ao processo principal em execução e, consequentemente, muito difícil de ser tratado.

Para resolver isso, é necessário que sejam implementadas diversas estratégias de recuperação de falhas, como realizar novas tentativas ou executar um rollback, o que acaba gerando mais complexidade para o código.

Como alternativa a esse problema, uma estratégia bastante utilizada e que traz uma solução mais segura para se conseguir um comportamento assíncrono dentro do Java é o uso de Message Driven Beans, ou MDBs.

Os MDB são, por definição, beans em Java que respondem a notificações por mensagem, ou seja, executam uma determinada ação quando recebem uma mensagem. Esse tipo de bean é bastante usado para o envio de notificações entre sistemas e tem, entre suas diversas características, a capacidade de gerenciar a transação de uma mensagem, ou seja, cuidar de todo o fluxo de processamento de uma mensagem, evitando que alguma mensagem se perca devido a qualquer erro.

Além disso, esses beans funcionam de modo assíncrono, isto é, o cliente que envia a mensagem não precisa esperar uma resposta para continuar seu processo, podendo ambos os processos rodarem paralelamente.

Na **Listagem 3** apresentamos a classe **ExemploMDB**, que implementa uma lógica de processamento de mensagens a partir do uso da tecnologia MDB.

Nessa listagem podemos observar que, para definir um MDB precisamos, em primeiro momento, utilizar a anotação **@MessageDriven** para indicar o tipo do bean que desejamos criar e algumas configurações do mesmo, como o tipo e nome do destino (nesse caso utilizamos uma fila com o nome **queue/test**), e o

tipo de confirmação de recebimento, que no exemplo está como automática.

Em segundo lugar, implementamos a interface **MessageListener**, que define um MDB passivo que aguarda uma mensagem para ser ativado. Nessa implementação, o método responsável por aguardar esse estímulo é o **onMessage()**, que recebe a mensagem como parâmetro e irá, dentro do código, imprimir o conteúdo dessa mensagem no console.

Listagem 3. Código da classe ExemploMDB.

```
@MessageDriven(activationConfig = {  
    @ActivationConfigProperty(  
        propertyName = "destinationType", propertyValue = "javax.jms.Queue"),  
    @ActivationConfigProperty(  
        propertyName = "destination", propertyValue = "java:/queue/test" )})  
public class ExemploMDB implements MessageListener {  
  
    @Override  
    public void onMessage(Message message) {  
        TextMessage tm = (TextMessage) message;  
        try {  
            System.out.println("Mensagem recebida " + tm.getText());  
        } catch (JMSException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Uma vez criado nosso MDB, agora iremos definir o cliente que irá enviar as mensagens para serem processadas em paralelo pelo Bean de Mensagens. Na classe **ServletClientMDB**, apresentada na **Listagem 4**, temos o código responsável por enviar essas mensagens. Note que ela ainda implementa a interface **HttpServlet**, permitindo que, uma vez instanciada pelo servidor, o cliente possa fazer requisições ao método **doGet()** através de chamadas HTTP desse tipo (GET).

Como podemos verificar, a Servlet de envio da mensagem é criada como uma Servlet tradicional, implementando, dentro do método **doGet()**, responsável por atender requisições HTTP do tipo GET, a lógica para envio de mensagens de texto para o MDB anteriormente definido.

Essa lógica é composta basicamente pela criação de uma conexão com nossa fila **queue/test** e o envio, através dessa conexão, de uma mensagem de texto com o conteúdo “Java Magazine”, definido através do método **createTextMessage()**. É importante notar que, uma vez a mensagem enviada, nossa Servlet não se preocupa em gerenciar se a mensagem foi entregue ou não, ficando como uma responsabilidade e funcionalidade do próprio servidor (nesse caso o WildFly) de entregar e, caso ocorra alguma falha, tentar novamente o envio da mensagem para evitar perda de informação.

Apesar de todos esses benefícios relacionados à segurança e garantia de entrega da mensagem ao destino, os MDBs apresentam algumas dificuldades ao serem aplicados em algumas situações. A primeira das dificuldades é que, para se conseguir uma resposta

Como trabalhar com tarefas assíncronas em Java EE

do processamento da mensagem, é necessário a criação de filas temporárias e outras estratégias que acabam se tornando bastante complexas para simplesmente conseguir um retorno assíncrono do método invocado.

Além disso, o MDB exige toda a estrutura e gerenciamento de um Application Server completo, o que se torna extremamente exagerado para pequenas tarefas que não necessitam de tantas features e, simplesmente, querem uma comunicação assíncrona simplificada.

Nota

O uso de MDBs possui diversas outras funcionalidades e utilidades, principalmente relacionadas à integração entre diferentes sistemas em uma arquitetura SOA (Arquitetura Orientada a Serviços), que vão além das apresentadas nos exemplos desse artigo. Para saber um pouco mais sobre essa tecnologia, adicionamos alguns endereços na seção **Links** com mais exemplos e definições do uso de MDBs.

Listagem 4. Implementação do cliente MDB ServletClientMDB.

```
@WebServlet("/clientMDB")
public class ServletClientMDB extends HttpServlet {

    @Resource(mappedName = "java:/queue/test")
    Queue queue;

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        Context ic = null;
        ConnectionFactory cf = null;
        Connection connection = null;

        try {
            ic = new InitialContext();
            cf = (ConnectionFactory)ic.lookup("/ConnectionFactory");

            connection = cf.createConnection();
            Session session = connection.createSession(
                false, Session.AUTO_ACKNOWLEDGE);
            MessageProducer publisher = session.createProducer(queue);

            connection.start();

            TextMessage message = session.createTextMessage("Java Magazine");
            publisher.send(message);

            out.println("Enviando mensagem");

        } catch (Exception exc) {
            exc.printStackTrace();
        }
        finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Com a finalidade de propiciar uma forma simples, confiável e fácil de ser implementada para a comunicação assíncrona, foi criada, na plataforma Java EE 6, a anotação **@Asynchronous**.

Métodos assíncronos com Java EE 6

A introdução de métodos assíncronos na Java EE 6 abriu a possibilidade aos desenvolvedores de implementar e desenvolver serviços com o uso de comunicação assíncrona sem muito esforço, deixando toda a parte de gerenciamento e controle das threads para o próprio servidor.

Sua utilização se dá a partir da anotação **@Asynchronous**, empregada em métodos de qualquer EJB (Stateless ou Stateful), permitindo que, quando a invocação do método anotado seja feita, essa comunicação seja de forma assíncrona, ou seja, o método que o chama não necessita aguardar sua execução para continuar seu processamento.

Nota

Para saber um pouco mais sobre EJBs e suas utilidades, adicionamos na seção **Links** um endereço que traz um conteúdo que explica o básico sobre Stateless e Stateful beans.

Além disso, a Java EE 6 também trouxe a interface **Future<V>**. Essa interface, normalmente utilizada como tipo de retorno de métodos assíncronos, permite que se obtenha o resultado de um método desse tipo no momento que se achar necessário, podendo ser obtido muito depois da invocação do método.

Para demonstrar o uso de ambas as funcionalidades, na **Listagem 5** apresentamos a implementação de um EJB com dois métodos assíncronos: um que não possui retorno e outro que retorna um **Boolean**, através da interface **Future**.

Essa listagem apresenta um pouco do funcionamento da anotação **@Asynchronous**. Como podemos ver, ambos os métodos possuem um funcionamento semelhante, imprimindo uma mensagem no console após um tempo de espera.

Listagem 5. Implementação de um EJB assíncrono.

```
@Stateless
public class EJBAsynch {

    @Asynchronous
    public void metodoSemRetorno() throws InterruptedException{
        System.out.println("Inicieie!");
        Thread.sleep(10000);
        System.out.println("Terminei!");
    }

    @Asynchronous
    public Future<Boolean> metodoComRetorno() throws InterruptedException{
        System.out.println("Inicieei com retorno!");
        Thread.sleep(10000);
        System.out.println("Terminei com retorno!");
        return new AsyncResult<>(Boolean.TRUE);
    }
}
```

A única diferença está no método **metodoComRetorno()**, onde devolvemos um **AsyncResult**, ao contrário do outro, que não possui retorno.

A classe **AsyncResult**, que implementa a interface **Future**, permite que devolvamos, como retorno do método **metodoComRetorno()**, uma referência assíncrona a um objeto, que no nosso exemplo é um **Boolean**. Essa referência é armazenada dentro do retorno **AsyncResult** e é devolvida imediatamente ao método chamador, não esperando nenhum processamento e garantindo, dessa forma, a comunicação assíncrona.

Com a referência desse resultado em mãos, o método que iniciou a chamada fica livre para executar as tarefas que achar necessário, enquanto a tarefa assíncrona fica rodando. Por fim, caso seja necessário, esse mesmo método pode recuperar o objeto booleano contido dentro do **AsyncResult**, ao chamar o método **get()** dessa classe, permitindo o acesso ao resultado que foi processado em paralelo. Voltando ao nosso exemplo, ao invocar a função **get()**, teremos acesso ao valor **true**, retornado no final do método **metodoComRetorno()**.

Portanto, podemos dizer que a interface **Future** (e consequentemente sua implementação **AsyncResult**) funciona como uma espécie de **wrapper**, sendo responsável por devolver o objeto retornado pela tarefa assíncrona ou, caso a tarefa não tenha terminado, esperar o seu fim para então devolver o resultado.

Para apresentar o uso dessa interface, na **Listagem 6** apresentamos a classe **ServletEJBAsynch**, responsável por fazer a chamada aos métodos de nossa classe **EJBAsynch** e demonstrar o funcionamento da invocação de um método assíncrono.

Listagem 6. Servlet que invocará nossos métodos assíncronos.

```
@WebServlet("/clientEjb")
public class ServletEJBAsynch extends HttpServlet {

    @EJB
    private EJBAsynch asynch;

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        try {
            System.out.println("Iniciando servlet! Invocando método assíncrono!");
            asynch.metodoSemRetorno();
            System.out.println("Invocando método assíncrono com retorno!");
            Future<Boolean> retorno = asynch.metodoComRetorno();
            System.out.println("Invocados ambos métodos!");
            System.out.println("O retorno é " + retorno.get());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Para rodarmos essa servlet basta subirmos nosso processo dentro do WildFly e acessar a URL <http://localhost:8080/ArtigoAsync/client-Ejb>. Feito isso, é interessante notar que os métodos assíncronos serão invocados através do EJB que injetamos na classe, com o uso da anotação **@EJB**, e que as chamadas não aguardam a finalização do método invocado para serem concluídas.

Também é importante notar que o único momento em que a servlet espera pela execução das funções invocadas é quando chamamos o método **get()**, da interface **Future**, que é responsável por recuperar o resultado de uma tarefa assíncrona. Conforme mencionado na explicação da listagem anterior, esse método sempre aguarda a tarefa assíncrona terminar para retornar o resultado. Com isso, é possível garantir que a tarefa assíncrona chegou ao fim e, consequentemente, que o resultado final, que será devolvido ao método chamador, foi completamente processado.

Como vantagens do uso da anotação **@Asynchronous**, temos dois pontos principais: o primeiro é a segurança que o uso de EJBs traz para o sistema, deixando o controle de transação como responsabilidade do servidor – uma tarefa extremamente complexa e sujeita a falhas se deixada para ser implementada manualmente. Em segundo lugar, ao aliar o uso desse tipo de método com a interface **Future**, fica muito fácil extraír os resultados de um processo assíncrono, abrindo caminho para diversos usos desses dois recursos em projetos reais.

Em projetos reais, a comunicação assíncrona é extremamente importante em processos batch e em serviços de notificação, principalmente pelo fato desses serviços não exigirem uma resposta dos métodos invocados. Alguns exemplos de projetos que utilizam essa tecnologia são serviços de e-mail, SMS, Push para celulares e outras plataformas de troca de mensagens.

Além disso, graças à interface **Future** e à possibilidade de recuperar o retorno dessas tarefas assíncronas, a criação de tarefas concorrentes através da anotação **@Asynchronous**, da Java EE 6, permite que os desenvolvedores construam processos paralelizados com mais facilidade, sendo o uso dessa interface extremamente útil em casos que uma tarefa bastante custosa precisa ser quebrada em tarefas menores.

Usando tarefas assíncronas para concorrência entre processos

Para apresentar um exemplo simplificado do conceito de paralelismo entre threads, iremos introduzir nesse artigo como podemos usar métodos assíncronos, especificamente aplicando a anotação **@Asynchronous**, para construir processos concorrentes. No nosso caso, iremos processar um determinado dado através de diversas tarefas em paralelo.

A adoção desse tipo de estratégia permite que uma tarefa custosa seja executada paralelamente e, devido a isso, seja concluída muito mais rapidamente. Para demonstrar um exemplo dessa abordagem, na **Listagem 7** exibimos a classe **AsyncParalelo**, que usaremos em nosso exemplo de concorrência.

Esse exemplo se baseará num cenário fictício em que, a partir de um conteúdo textual, representado em nosso exemplo por uma lista de Strings, precisamos realizar um replace de certas palavras por outras. Esse tipo de situação é bastante comum em sistemas de publicação de anúncios, onde certas palavras precisam ser removidas para evitar que conteúdos impróprios possam ser publicados indevidamente.

Como podemos observar nessa listagem, a classe **AsyncParalelo** simplesmente implementa um método assíncrono responsável

Como trabalhar com tarefas assíncronas em Java EE

por fazer o replace de todas as ocorrências da palavra **Java** em uma lista de Strings. Essa classe também faz uso da interface **Future**, que permitirá que, assim que a tarefa assíncrona terminar, o método principal possa recuperar a lista de Strings substituídas.

Listagem 7. Classe responsável por processar paralelamente uma atividade de replace de String.

```
@Stateless  
public class AsyncParalelo {  
  
    @Asynchronous  
    public Future<List<String>> replaceLista(List<String> lista){  
        List<String> listFinal = new ArrayList<>();  
        for(String palavra:lista){  
            listFinal.add(palavra.replace("Java","Magazine"));  
        }  
        return new AsyncResult<List<String>>(listFinal);  
    }  
}
```

Agora, com nosso método assíncrono pronto, vamos criar a Servlet que irá executar esse método. Sua função principal será, através das funcionalidades da Java EE 6, iniciar várias threads em paralelo no momento que o usuário fizer uma requisição à URL

startAsync e, em cada uma dessas threads, executar um processo para substituição de palavras dentro de uma lista de strings. A implementação dessa classe, que chamamos de **StartAsync**, pode ser vista na **Listagem 8**.

Essa listagem nos mostra como é simples realizar uma tarefa concorrente utilizando as novas soluções presentes na Java EE 6. Na implementação da classe **StartAsync**, criamos um código onde, dentro do método **doGet()**, construímos uma lista com 99 Strings, representando nosso conteúdo textual com as palavras que deverão ser substituídas.

Uma vez criada essa lista, separamos a mesma em duas listas menores, possibilitando que cada uma seja processada em paralelo pelo nosso método assíncrono, através de duas tarefas concorrentes. Nesse ponto, é interessante observar que, devido à comunicação assíncrona, ambas as tarefas são iniciadas ao mesmo tempo e irão rodar paralelamente, ao contrário da situação que teríamos se rodássemos essa tarefa de modo síncrono, quando teríamos que esperar a finalização de uma tarefa para iniciar a outra.

Por fim, pegamos o resultado de ambas as tarefas e, para demonstrar que a tarefa foi concluída com sucesso, imprimimos o tamanho da lista final processada, terminando assim nosso exemplo de tarefas concorrentes utilizando métodos assíncronos.

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

Esse exemplo nos mostra uma aplicação prática de métodos assíncronos que pode ser adotada para o processamento de textos e remoção de palavras “proibidas” em conteúdos digitais. Esse mesmo conceito de comunicação e paralelismo pode ser empregado em diversas outras tarefas, como consultas custosas em bancos dados (projetando pequenas queries em paralelo ao invés de uma única query muito grande) e, também, em processos que envolvem cálculos e formulação de resultados matemáticos, nos quais muitas vezes a execução em uma só tarefa não é possível devido ao alto tempo para processamento.

Listagem 8. Servlet que implementa a chamada concorrente da classe AsyncParalelo.

```
@WebServlet("/startAsync")
public class StartAsync extends HttpServlet {

    @EJB
    private AsyncParalelo asyncTask;

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        try {
            List<String> listaASerProcessada = new ArrayList<>();
            for (int i = 0; i < 100; i++) {
                listaASerProcessada.add("Java");
            }
            // Em nossa tarefa, vamos quebrar o processamento da lista em 2
            List<String> parte1 = listaASerProcessada.subList(0, 50);
            List<String> parte2 = listaASerProcessada.subList(50, 99);
            System.out.println("Iniciando tarefas assíncronas");
            Future<List<String>> processamento1 = asyncTask
                .replaceLista(parte1);
            Future<List<String>> processamento2 = asyncTask
                .replaceLista(parte2);
            System.out.println("Finalizou chamadas");
            System.out.println("Pegando resultados");
            parte1 = processamento1.get();
            parte2 = processamento2.get();
            List<String> listaProcessada= new ArrayList<>(parte1);
            listaProcessada.addAll(parte2);
            System.out.println("O tamanho da lista processada eh "
                + listaProcessada.size());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

Situações e aplicações de métodos assíncronos

Os novos recursos da Java EE 6 trouxeram para o Java um novo aliado na implementação de serviços assíncronos e de tarefas concorrentes, facilitando o trabalho de desenvolvedores ao construir sistemas eficientes e performáticos.

Porém, não podemos simplesmente ignorar as outras estratégias, como o MDB e o simples uso de Threads, para implementar tarefas assíncronas. Como desenvolvedores, é importante analisarmos o problema que temos à nossa frente e pensarmos na melhor maneira de solucioná-lo.

Como exemplo de tais situações, MDBs são extremamente úteis para o envio de notificações entre sistemas remotos. Já o uso de threads é bastante recomendado quando é necessária a criação de sistemas desktop ou sistemas que não possuem uma infraestrutura com um servidor de aplicação como base.

Portanto, a utilização da anotação **@Asynchronous** pode ser bastante prática, mas é sempre aconselhável estudar o contexto do projeto e da situação a ser enfrentada para assim verificar se a opção pelo uso desse recurso, ao invés do MDB e da interface **Runnable**, será adequado para a solução de seu problema e será a melhor opção para a construção de uma solução assíncrona.

Autor



Bruno F. M. Attorre

brattorre@gmail.com

Trabalha com Java há três anos. Apaixonado por temas como Inteligência Artificial, ferramentas open source, BI, Data Analysis e Big Data, está sempre à procura de novidades tecnológicas na área. Possui as certificações OCJP, OCWCD e OCEEJBD.



Links:

Link para download do WildFly 8.

<http://download.jboss.org/wildfly/8.1.0.Final/wildfly-8.1.0.Final.zip>

Página com instruções de instalação do JBoss Tools e do WildFly 8.

<http://www.mastertheboss.com/wildfly-8/configuring-eclipse-to-use-wildfly-8>

Exemplos de uso do MDB.

<https://docs.oracle.com/javaee/6/tutorial/doc/bnbpk.html>

Tutorial sobre EJBs.

<http://docs.jboss.org/ejb3/app-server/tutorial/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



DevOps: Como criar ambientes virtuais para testes

Conheça uma maneira eficiente de criar ambientes complexos com virtualização e provisionamento

Entregar software com qualidade sempre foi um desafio muito grande para a maioria das empresas de Tecnologia da Informação (TI). Desafio esse que está diretamente relacionado aos riscos existentes no processo de Engenharia de Software, que vão desde a análise até a distribuição da nova versão.

O processo de distribuição de uma nova versão no ambiente de produção muitas vezes é considerado algo simples e sem muitos riscos. Porém, esse processo pode se tornar catastrófico se as etapas de testes não forem executadas em um ambiente bastante similar ao ambiente no qual o software será executado.

Muitas vezes o ambiente de testes não representa o ambiente de produção. Isto pode ocorrer devido à complexidade do projeto, elevado custo ou até mesmo uma distância entre a equipe de desenvolvimento e a equipe de infraestrutura.

Na tentativa de estreitar a relação entre as equipes de desenvolvimento e infraestrutura surgiu o movimento DevOps, que trouxe com ele muitos facilitadores para contornar esses problemas e garantir uma distribuição com testes mais próximos ao ambiente de produção e com menos riscos.

Através da virtualização é possível criar ambientes mais próximos à produção, sem a necessidade de dispor de uma grande quantidade de máquinas físicas. Além do ambiente similar à produção, é muito interessante poder recriar todo ambiente virtual de uma maneira rápida e sem esforço, viabilizando deste modo testes mais completos do software antes de distribui-lo.

Fique por dentro

O desenvolvimento de software está cada vez mais complexo, e um dos motivos para isto é a grande variedade de requisitos não funcionais inseridos no escopo de desenvolvimento do software. Em virtude disso, cada vez mais o processo de instalação e configuração do ambiente de testes e homologação se torna custoso. Na maioria das vezes, para conseguir um ambiente similar ao de produção, é necessária uma grande estrutura de servidores, que muitas vezes não estão disponíveis.

Deste modo, neste artigo serão apresentados os conceitos básicos de virtualização e provisionamento, bem como as ferramentas que podem ser utilizadas para criar um ambiente virtualizado similar ao de produção de forma rápida e eficiente, facilitando os testes em cenários mais próximos da realidade.

Virtualização

Nos últimos anos o assunto Virtualização tem ganhado muito destaque, mas o conceito de virtualização foi criado no final dos anos 60 e início dos anos 70 pela IBM.

O intuito da virtualização é abstrair os hardwares, criando um grande conjunto de recursos lógicos (CPU, memória, rede, armazenamento, entre outros), permitindo assim a definição de máquinas virtuais com configurações e sistemas operacionais distintos em uma única máquina física.

A máquina física, conhecida como **host**, é responsável por fornecer uma plataforma virtual para a execução dos sistemas operacionais “convidados” (**guest**). Esta plataforma virtual será criada a partir dos recursos da máquina **host**, ficando assim limitada à capacidade dos recursos disponíveis.

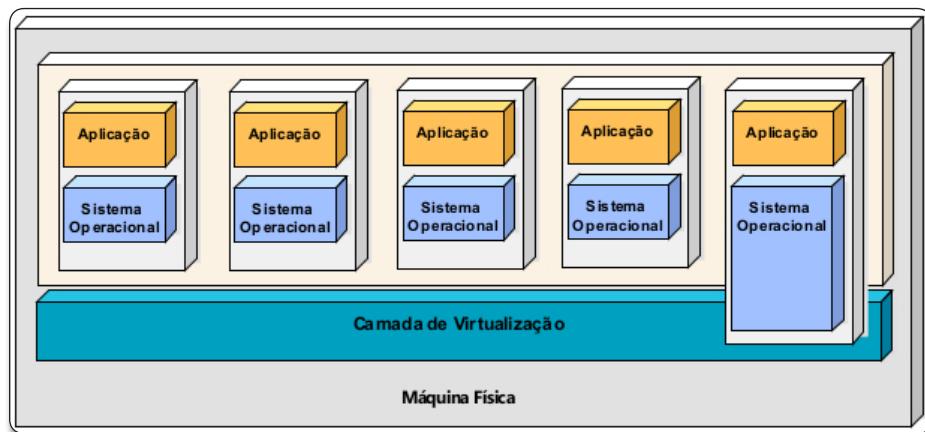


Figura 1. Hypervisor – Native ou Bare Metal

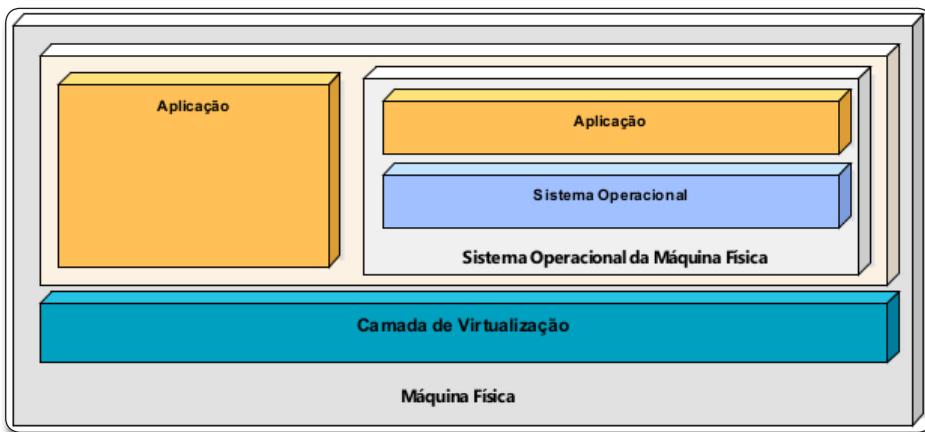


Figura 2. Hypervisor – Hosted

Hypervisor

O Hypervisor, ou Virtual Machine Monitor (VMM), é um dos principais componentes da virtualização. A sua função é criar a plataforma virtual na máquina **host**, viabilizando deste modo a instalação de vários sistemas operacionais **guest** que irão compartilhar os recursos da máquina **host**.

Os dois tipos mais comuns de Hypervisor são:

- **Native ou Bare Metal:** software de virtualização executado diretamente no hardware **host**, sendo responsável por gerenciar os recursos e o sistema operacional **guest** (vide **Figura 1**). Neste caso, o sistema operacional **guest** é executado um nível acima do Hypervisor, o que na maioria das vezes, representa um melhor desempenho. Exemplos de soluções desse tipo: VMware ESX, Xen, Hyper-V, etc.

- **Hosted:** software de virtualização executado sobre o sistema operacional da máquina

host (vide **Figura 2**). Neste caso, existem duas camadas de software entre o hardware da máquina física e o sistema operacional **guest**. Exemplos de soluções desse tipo: KVM, Parallels e Oracle VM VirtualBox.

Oracle VM VirtualBox

O Oracle VM VirtualBox é um software de virtualização que permite a criação de máquinas virtuais (**guest**) que são executadas sobre o sistema operacional da máquina física (**host**). Ele pode ser utilizado em ambientes corporativos ou domésticos, disponibilizando uma série de funcionalidades interessantes para a criação de máquinas virtuais. Com o intuito de atender uma grande quantidade de usuários, esta solução pode ser instalada nos principais sistemas operacionais do mercado, além de possuir uma lista extensa de sistemas operacionais que poderão ser utilizados nas máquinas virtuais.

As máquinas virtuais podem ser criadas através de uma interface gráfica amigável disponibilizada pelo VirtualBox. Esse é um processo bem simples e fornece uma série de opções para a personalização dos hardwares e sistema(s) operacional(is) da(s) máquina(s) sendo criada(a). Mesmo com toda a simplicidade, algumas vezes é mais interessante criar máquinas virtuais através de um processo automatizado, evitando assim a necessidade de uma configuração ou inicialização manual. Uma das maneiras de automatizar o processo de configuração e inicialização dessas máquinas é utilizar uma ferramenta chamada Vagrant, que faz uso de um SDK disponibilizado pelo VirtualBox para interagir com a plataforma virtual.

Vagrant

O Vagrant é um software utilizado para criar e configurar ambientes virtuais de desenvolvimento, porém ele não é responsável pelo processo de virtualização. Este processo é delegado para uma ferramenta especializada em virtualização, denominada provedor. Um exemplo de provedor do Vagrant é o VirtualBox, porém existem outros provedores disponíveis, como o VMware Fusion e até mesmo uma instância EC2 na Amazon.

Para criar um ambiente virtual no Vagrant é necessária uma imagem de um determinado sistema operacional, conhecida como **box**. Os **boxes** têm por objetivo agilizar a criação dos ambientes virtuais e são vinculados diretamente a um determinado provedor suportado pelo Vagrant, ou seja, um **box** do Vagrant vinculado ao VirtualBox não poderá ser usado com os demais provedores suportados pelo Vagrant.

A configuração do ambiente virtual, contendo o nome do **box** a ser utilizado, é feita em um único arquivo, chamado **Vagrantfile**. Através desse arquivo é possível recriar com muita facilidade o ambiente virtual, na mesma máquina física ou em outras máquinas físicas que possuam o Vagrant e pelo menos um provedor instalados.

A **Listagem 1** demonstra um exemplo de um **Vagrantfile** mínimo para a criação de uma máquina virtual utilizando um **box** com o sistema operacional Linux, distribuição Ubuntu 14.04.

DevOps: Como criar ambientes virtuais para testes

Listagem 1. Arquivo mínimo de inicialização de uma máquina virtual com Vagrant.

```
# -*- mode: ruby -*-
# vi: set ft=ruby:

# Versão da API, não deve ser alterada
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # É obrigatório informar o box que será utilizado
  config.vm.box = "trusty"
end
```

O Vagrantfile pode ser criado automaticamente através do comando *vagrant init*. Este comando possui a opção para informar o box que será utilizado para criação da máquina virtual, a saber: *vagrant init <URL ou nome do novo box>*.

Para facilitar, existe uma série de boxes prontos (veja a seção **Links**) com uma grande variedade de sistemas operacionais e distribuições. Além disso, o Vagrant permite criar boxes personalizados de acordo com a necessidade.

Criado o Vagrantfile, é necessário executar o comando *vagrant up* para criar a máquina virtual, como demonstra a **Figura 3**. Uma das opções desse comando é informar qual provedor será utilizado. Para isso, deve-se informar a opção *--provider*. Caso ela seja omitida, o provedor padrão será utilizado; neste caso, o VirtualBox.

```
[leocomelli] ~/Dev/envs/devmedia/vagrant-sample$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
--> default: Importing base box 'trusty-64'...
--> default: Matching MAC address for NAT networking...
--> default: Setting the name of the VM: vagrant-sample_default_1412821268859_37481
--> default: Clearing any previously set forwarded ports...
--> default: Clearing any previously set network interfaces...
--> default: Preparing network interfaces based on configuration...
--> default: Adapter 1: nat
--> default: Forwarding ports...
--> default: 22 => 2222 (adapter 1)
--> default: Booting VM...
--> default: Waiting for machine to boot. This may take a few minutes...
--> default: SSH address: 127.0.0.1:2222
--> default: SSH username: vagrant
--> default: SSH auth method: private key
--> default: Warning: Connection timeout. Retrying...
--> default: Warning: Remote connection disconnect. Retrying...
--> default: Machine booted and ready!
--> default: Checking for guest additions in VM...
--> default: Mounting shared folders...
--> default: /vagrant => /Users/leocomelli/Dev/envs/devmedia/vagrant-sample

[leocomelli] ~/Dev/envs/devmedia/vagrant-sample$
```

Figura 3. Criando um novo ambiente virtual com Vagrant

Ao término da execução do comando para criar/inicializar a máquina virtual, a mesma poderá ser acessada através do protocolo SSH. Para isso, deve-se executar o comando *vagrant ssh*, que irá criar uma sessão SSH, permitindo então a interação com a máquina virtual (veja a **Figura 4**).

A partir da sessão SSH é possível instalar e configurar novos pacotes de acordo com a necessidade do projeto. Além do comando para criar uma sessão SSH com a máquina virtual, o Vagrant possui um comando que destrói a máquina virtual. Para isso, basta encerrar a sessão SSH através do comando *exit*, e em seguida, a partir da máquina física, executar o comando *vagrant destroy*.

O processo de destruir a máquina virtual irá encerrar a mesma e apagar todos os arquivos de configuração utilizados pelo Vagrant. Caso haja a necessidade de apenas parar ou suspender a máquina virtual para que a mesma possa ser utilizada posteriormente, basta utilizar os comandos *vagrant halt* ou *vagrant suspend*, respectivamente.

O Vagrantfile exibido na **Listagem 1** é a configuração mínima. Como um dos seus diferenciais, o Vagrant disponibiliza uma série de configurações para as máquinas virtuais, e estas são divididas em três namespaces, apresentados na **Tabela 1**.

Nota

O provedor padrão do Vagrant pode ser alterado. Para isso, é necessário atribuir o nome do provedor à variável de ambiente **VAGRANT_DEFAULT_PROVIDER**. Por exemplo, para utilizar o provedor VMware Fusion como padrão, o valor **vmware_fusion** deverá ser atribuído à variável mencionada.

```
[leocomelli] ~/Dev/envs/devmedia/vagrant-sample$ vagrant ssh
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-36-generic x86_64)

 * Documentation: https://help.ubuntu.com/
 * System information as of Thu Oct  9 02:21:29 UTC 2014
   System load:  0.84          Processes:      91
   Usage of /:  2.7% of 39.34GB  Users logged in:    0
   Memory usage: 17%           IP address for eth0: 10.0.2.15
   Swap usage:  0%
 * Graph this data and manage this system at:
   https://landscape.canonical.com/
 * Get cloud support with Ubuntu Advantage Cloud Guest:
   http://www.ubuntu.com/business/services/cloud
 0 packages can be updated.
 0 updates are security updates.

vagrant@vagrant-ubuntu-trusty-64:~$
```

Figura 4. Acessando o novo ambiente virtual

Namespace	Descrição
config.vm	Configuração da máquina virtual gerenciada pelo Vagrant.
config.ssh	Configuração que o Vagrant utilizará para acessar a máquina virtual através do SSH.
config.vagrant	Configuração do comportamento do Vagrant.

Tabela 1. Configurações do Vagrant

Na maioria das vezes, as configurações utilizadas fazem parte do namespace **config.vm**. Como mencionado, neste namespace estão as configurações que atuam diretamente na máquina virtual, como: o box que será utilizado, redirecionamento de portas entre a máquina **host** e a máquina **guest**, sincronização de diretórios, recursos de hardware, entre outros.

A **Listagem 2** demonstra um Vagrantfile com mais opções de configuração, e a **Tabela 2** explica cada uma dessas opções.

Além das opções de configuração mencionadas, uma bem interessante é a **config.vm.provision**, que permite a utilização de um provisionador durante a criação da máquina virtual. O Vagrant dá suporte a vários provisionadores, como: Chef, Puppet e Ansible.

Provisionamento

Assim como a virtualização, o provisionamento ganhou muito espaço com a chegada da “era cloud”. O provisionamento é uma maneira de automatizar os processos de configuração de estações de trabalho e servidores.

Note que analisamos uma maneira simples e rápida de criar máquinas virtuais com a ajuda do Vagrant (e o provedor VirtualBox). Ao vincular a virtualização com o provisionamento há um ganho na agilidade da criação de um novo ambiente, pois o provisionamento automatiza a instalação e configuração das ferramentas quando for necessário recriar a máquina virtual.

As ferramentas de provisionamento disponíveis atualmente utilizam o conceito de receitas para executar as tarefas necessárias no computador de destino, computador este que pode ser uma máquina física, uma instância EC2 na Amazon ou uma máquina virtual local criada através do Vagrant.

Nota

O Vagrant Cloud é um espaço que permite o compartilhamento de ambientes virtuais do Vagrant com outros usuários. É um local para buscar por um box criado por outro usuário ou publicar um box criado por você. Este compartilhamento de boxes tem como principal objetivo agilizar ainda mais a criação de ambientes virtuais.

Listagem 2. Arquivo personalizado de inicialização de um ambiente virtual com Vagrant.

```
# -*- mode: ruby -*-
# vi: set ft=ruby:

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "trusty"
  config.vm.box_url = "https://cloud-images.ubuntu.com/vagrant/trusty/...box"
  config.vm.network "forwarded_port", guest: 8080, host: 8080
  config.vm.synced_folder "./devmedia", "/devmedia"
  config.vm.provider "virtualbox" do |vb|
    vb.customize ["modifyvm", :id, "--memory", "2048"]
  end
end
```

A grande vantagem das ferramentas de provisionamento está no fato que elas fornecem uma série de facilidades que auxiliam a escrita e o controle da execução das receitas, que anteriormente eram criadas através de shell script ou comandos DOS.

Ansible

O Ansible é uma das opções de ferramentas de provisionamento. Com ele é possível configurar sistemas, distribuir versões de software e orquestrar tarefas avançadas relacionadas à infraestrutura.

Assim como as demais ferramentas de provisionamento, o Ansible também utiliza o conceito de receitas, que são conhecidas como playbooks e são escritas em arquivos no formato YAML. Este formato foi pensado para não ser uma linguagem de marcação, e sim algo amigável para a serialização de dados, permitindo uma fácil leitura e escrita.

Dentre todas as receitas que podem ser criadas, uma receita útil em qualquer projeto Java é a instalação do JDK, que pode ser utilizada para fazer a instalação em um servidor ou mesmo em uma estação de trabalho.

Listagem 3. Receita Ansible para instalar o JDK 8 (java.yml).

```
- hosts: webservers
  sudo: yes

  tasks:
    - name: Adicionar repositório do Java 8
      apt_repository:
        repo: 'ppa:webupd8team/java'
        state: present

    - name: Aceitar automaticamente a licença Oracle
      shell: echo debconf shared/accepted-oracle-license-v1-1 select true | sudo debconf-set-selections

    - name: Instalar o JDK 8
      apt:
        name: oracle-java8-installer
        state: latest
        update-cache: yes
        force: yes
```

A receita apresentada na **Listagem 3** está dividida em três tarefas. A primeira adiciona o repositório que contém o pacote do Java 8, a segunda aceita licença da Oracle – item obrigatório para

Configurações	Descrição
config.vm.box	Configura qual box será utilizado. O valor deve ser o nome de um box instalado ou um nome abreviado de um box do VagrantCloud.
config.vm.box_url	Configura as opções de endereços para buscar por um box. Caso o box já esteja instalado ou esteja disponível no VagrantCloud, esta opção pode ser omitida.
config.vm.network	Configura o redirecionamento de portas entre a máquina física e a máquina virtual. Na configuração realizada na Listagem 2 , toda requisição na porta 8080 da máquina física será redirecionada para a porta 8080 da máquina virtual.
config.vm.synced_folder	Configura os diretórios que serão sincronizados entre a máquina física e a máquina virtual. Por padrão, o Vagrant cria um diretório sincronizado entre o diretório da máquina física em que foi criada a máquina virtual e o /vagrant na máquina virtual.
config.provider	Configura os recursos na máquina virtual de acordo com o provedor utilizado.

Tabela 2. Detalhamento de algumas configurações do Vagrant

permitir a instalação do pacote –, e a terceira tarefa é responsável pela instalação do pacote.

Nessas três tarefas foram usados módulos disponíveis no Ansible. Na primeira tarefa, por exemplo, o módulo `apt_repository` é responsável por adicionar ou remover um repositório APT nas distribuições Debian ou Ubuntu. Já a opção `state` indica ao Ansible se o repositório deve ser adicionado (`present`) ou removido (`absent`) da lista de repositórios.

Na tarefa seguinte, o módulo `shell` executa o comando de aceite da licença Oracle. E na última tarefa, o módulo `apt` gerencia o pacote Java a ser instalado. Assim como no módulo `apt_repository`, algumas opções foram utilizadas, a saber: `state`, para garantir que a última versão do pacote seja instalada; `update-cache`, que executa o comando para atualizar a lista de pacotes disponíveis nos repositórios da APT adicionados através do `apt_repository`; e a opção `force`, para obrigar a instalação do pacote.

Além dos módulos mencionados, o Ansible disponibiliza uma série de outros módulos que auxiliam na criação de novas receitas. Essas receitas podem ser executadas em uma ou mais máquinas, que são definidas através do inventário (`inventory`), como exibido na **Listagem 4**. O inventário é utilizado pela receita para indicar quais as máquinas deverão ser provisionadas, conforme exibido na primeira linha da receita na **Listagem 3**. Neste caso, apenas as máquinas que pertencem ao grupo `webservers` serão provisionadas (para provisionar todas, utilize `all`). Em seguida, através da definição da opção `sudo: yes`, a receita informa que todas as tarefas deverão ser executadas como super-usuário.

Para provisionar as máquinas descritas no inventário exposto na **Listagem 4**, é necessário executar o comando `ansible-playbook -i hosts java.yml`, onde o arquivo `hosts` é o arquivo de inventário exibido na **Listagem 4**, e `java.yml` é a receita exibida na **Listagem 3**.

Listagem 4. Arquivo de inventário do Ansible (`hosts`).

```
[local]
127.0.0.1

[webservers]
web1.devmedia.com.br

[dbservers]
db1.devmedia.com.br
db2.devmedia.com.br
```

Nota

A execução das receitas na máquina física ou virtual de destino é realizada através do protocolo SSH. Por padrão, o Ansible considera que a autenticação por chaves SSH será utilizada.

Nota

A autenticação por chaves SSH permite que os usuários se conectem nos servidores através do protocolo SSH sem a necessidade de informar a senha. Este modelo de autenticação tem algumas vantagens sobre o modo convencional de autenticação, como: não transmitir a senha através da rede no momento do login e eliminar o risco de ataque por força bruta para descobrir a senha.

Ambiente virtual

Nos tópicos anteriores foram abordados conceitos básicos de virtualização e provisionamento. A partir desses conceitos será possível entender melhor os passos para a criação de um ambiente virtual.

Um ambiente virtual pode ser utilizado, por exemplo, para validar algumas decisões relacionadas a um ambiente clusterizado (estratégia de compartilhamento e/ou armazenamento da sessão, os mecanismos de平衡amento de carga, entre outros). Muitas vezes, ambientes como este só estão disponíveis no ambiente de produção, criando muita dificuldade para realizar tais validações. Nesses casos, uma das alternativas que os desenvolvedores possuem é recorrer a um ambiente virtual que seja similar ao de produção.

Para reproduzir o cenário descrito será criado um ambiente composto por duas instâncias do servlet container Jetty e um servidor HTTP Nginx, para realizar o balanceamento de carga entre os nós, conforme exibido na **Figura 5**.

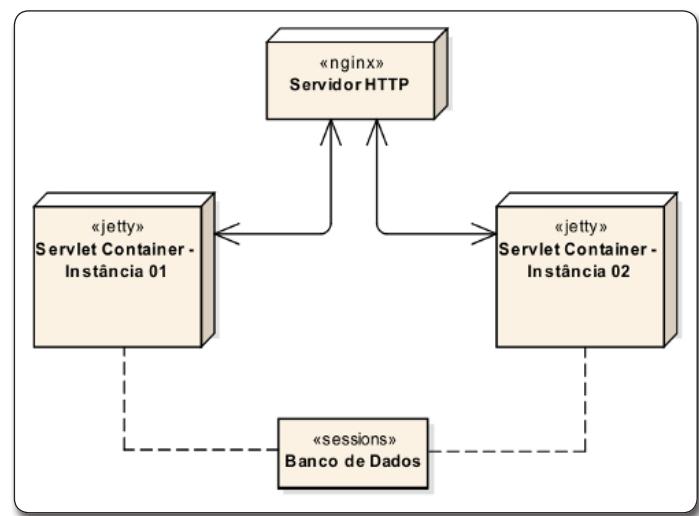


Figura 5. Estrutura de servidores do ambiente virtual

Para contemplar toda a necessidade do cenário descrito serão criadas algumas receitas. Receitas estas que devem respeitar uma determinada ordem para contemplar a configuração de todo o ambiente virtual.

Dito isso, o primeiro passo é criar uma receita para instalar e configurar o MySQL, pois a configuração do compartilhamento de sessões do Jetty depende de algumas informações do banco de dados. Logo em seguida, será criada a receita do Jetty, e posteriormente a receita do Nginx, que precisa de algumas informações relacionadas aos nós do servlet container para configurar o balanceamento de carga.

MySQL, armazenando as sessões

Neste cenário, será utilizado o banco de dados MySQL para armazenar as sessões que serão compartilhadas entre as instâncias do cluster. Para facilitar a criação da máquina virtual com o MySQL, será definida uma receita responsável por instalar e configurar o MySQL, além de criar o banco de dados e o usuário necessário.

Listagem 5. Receita para instalar e configurar o MySQL (mysql.yml)

```
- hosts: all
  sudo: yes

  tasks:
    - name: Instalar Pacotes MySQL
      apt:
        pkg: "{{ item }}"
        state: present
      with_items:
        - mysql-server
        - python-mysqldb

    - name: Liberar acesso ao MySQL
      lineinfile:
        dest: /etc/mysql/my.cnf
        regexp: '^bind-address(.*)= 127\.0\.0\.1'
        line: 'bind-address = 0.0.0.0'

    - name: Reiniciar o MySQL
      service:
        name: mysql
        state: restarted

    - name: Criar banco de dados (sessions)
      mysql_db:
        name: sessions
        state: present

    - name: Criar usuário
      mysql_user:
        name: root
        password: secret
        priv: "%.*:ALL,GRANT"
        host: "%"
        state: present
```

Na **Listagem 5** a primeira tarefa é responsável por instalar os pacotes necessários para a utilização do MySQL. A segunda tarefa, por sua vez, utiliza o módulo do Ansible `lineinfile` para alterar o conteúdo do arquivo de configuração do MySQL (`my.cnf`) para liberar o acesso remoto, e para recarregar o arquivo alterado, a tarefa seguinte reinicia o serviço.

Esta receita tem como objetivos instalar e configurar o MySQL. Para simplificar a configuração, o Ansible disponibiliza os módulos `mysql_db` e `mysql_user`, que fornecem facilidades para o gerenciamento dos bancos de dados e usuários. Para o funcionamento destes, é necessária a instalação do pacote `python-mysqldb`, conforme realizado na primeira tarefa.

Jetty, o servlet container

O Jetty é um servidor HTTP e um servlet container open source mantido pela Fundação Eclipse há muitos anos. Pelo fato de ser open source, tem como uma das suas vantagens uma comunidade muito forte, além de garantir uma boa performance, baixo tempo de carregamento da página, alto throughput e baixo consumo de memória.

Um recurso necessário em qualquer servlet container que será utilizado em um ambiente de alta disponibilidade é o compartilhamento de sessões HTTP. O Jetty atende a esta necessidade com-

partilhando as sessões através de um banco de dados, que deve ser acessível por todas as instâncias que fazem parte do cluster. Os responsáveis pela configuração desse recurso são o gerenciador de identificador da sessão e o gerenciador de sessão.

Neste contexto, o gerenciador de identificador da sessão deve ser configurado em cada instância do Jetty, garantindo assim a existência de um único identificador para todas as aplicações hospedadas na instância.

Listagem 6. Trecho da configuração do gerenciador de identificador da sessão (jetty.xml.j2).

```
<Set name="sessionIdManager">
<New id="jdbcidmgr" class="org.eclipse.jetty.server.session.JDBCSessionIdManager">
<Arg>
<Ref id="Server"/>
</Arg>
<Set name="workerName">{{ node_name }}</Set>
<Call name="setDriverInfo">
<Arg>com.mysql.jdbc.Driver</Arg>
<Arg>jdbc:mysql://{{ mysql_ip }}:3306/sessions?
user=root&password=secret</Arg>
</Call>
<Set name="scavengelInterval">60</Set>
</New>
</Set>
<Call name="setAttribute">
<Arg>jdbcIdMgr</Arg>
<Arg>
<Ref id="jdbcidmgr"/>
</Arg>
</Call>
```

Na **Listagem 6** é possível visualizar as configurações do gerenciador de identificador da sessão. Neste exemplo, será utilizado o banco de dados MySQL para armazenar as sessões que serão compartilhadas entre as instâncias do cluster. Na mesma listagem, também é possível identificar o uso de variáveis do Ansible, definidas dentro de duas chaves, conforme a sintaxe: `{} var {}`. Estas variáveis serão substituídas pelos valores atribuídos a elas durante a execução.

O gerenciador de sessão, por sua vez, deve ser configurado conforme a **Listagem 7**. Ele é responsável por manipular o ciclo de vida da sessão (criar/atualizar/invalidar/expirar) de uma aplicação web. Como mencionado, deve existir apenas um gerenciador de sessão por aplicação, e para garantir isso a configuração deste deve ser realizada no arquivo XML de contexto da aplicação.

Os arquivos de configuração do gerenciador de identificador da sessão (`jetty.xml.j2`) e do gerenciador da sessão (`context.xml.j2`) serão utilizados pela receita responsável por instalar e configurar o Jetty. Como esta receita é um pouco extensa, com o intuito de facilitar a explicação de todas as etapas, ela será apresentada em três partes: pacotes básicos, Jetty e configuração do cluster.

As duas primeiras tarefas dessa receita (apresentadas na **Listagem 8**) foram explicadas com mais detalhes anteriormente, na **Listagem 3**. A terceira tarefa também foi mencionada na mesma listagem, porém naquela ocasião a opção `with_items` não foi utilizada.

DevOps: Como criar ambientes virtuais para testes

Essa opção indica que o Ansible irá iterar por todos os itens (`oracle-java8-installer` e `unzip`) definidos na tarefa e irá utilizá-los como parâmetro para o módulo `apt`.

Listagem 7. Trecho da configuração do gerenciador de sessão (context.xml.j2)

```
<Ref name="Server" id="Server">
<Call id="jdbclDMgr" name="getAttribute">
<Arg>jdbclDMgr</Arg>
</Call>
</Ref>
<Set name="sessionHandler">
<New class="org.eclipse.jetty.server.session.SessionHandler">
<Arg>
<New id="jdbcmgr" class="org.eclipse.jetty.server.session.JDBCSessionManager">
<Set name="sessionIdManager">
<Ref id="jdbclDMgr"/>
</Set>
</New>
</Arg>
</New>
</Set>
```

Listagem 8. Receita para instalar e configurar o Jetty (jetty.yml) – Parte 1 de 3.

```
- hosts: all
sudo: yes
tasks:
- name: Adicionar repositório do Java 8
apt_repository:
repo: 'ppa:webupd8team/java'
state: present

- name: Aceitar automaticamente a licença Oracle
shell: echo debconf shared/accepted-oracle-license-v1-1 select true | sudo debconf-set-selections

- name: Instalar Pacotes
apt:
pkg: "{{ item }}"
update-cache: yes
state: present
with_items:
- oracle-java8-installer
- unzip

...
```

Listagem 9. Receita para instalar e configurar o Jetty (jetty.yml) – Parte 2 de 3.

```
- hosts: all
sudo: yes
tasks:
...
- name: Baixar Jetty
get_url:
url: http://eclipse.org/downloads/download.php?file=/jetty/stable-9/dist/jetty-distribution-9.2.3.v20140905.zip&r=1
dest: /tmp/jetty.zip

- name: Descompactar Jetty
unarchive:
src: /tmp/jetty.zip
dest: /opt
copy: no

- name: Renomear diretório do Jetty
shell: sudo mv /opt/jetty-distribution-9.2.3.v20140905 /opt/jetty
...
```

A segunda parte da receita, exibida na **Listagem 9**, é responsável por baixar o Jetty através do módulo `get_url` e salvar o arquivo no diretório `/tmp`. Em seguida, utilizando o módulo `unarchive` o Ansible irá descompactar o arquivo. A opção `copy: no` informa ao Ansible que o arquivo a ser descompactado já está na máquina remota, caso contrário o módulo iria copiar da máquina host para a máquina remota e depois descompactá-lo. Logo depois, o diretório descompactado do Jetty é renomeado através do módulo `shell`.

Por fim, a **Listagem 10** apresenta a última da parte da receita de configuração do Jetty e irá configurar o cluster, lembrando que as sessões serão persistidas em um banco de dados MySQL compartilhado entre as instâncias.

Nota

O módulo `unarchive` é capaz de descobrir qual o tipo da compactação automaticamente, porém é necessário que a ferramenta de descompactação esteja instalada no sistema operacional. Foi devido ao uso deste módulo que o pacote `unzip` foi instalado na **Listagem 7**.

Listagem 10. Receita para instalar e configurar o Jetty 3/3 (jetty.yml)

```
- hosts: all
sudo: yes
tasks:
...
- name: Copiar driver jdbc para o Jetty
copy:
src: files/mysql-connector-java-5.1.32-bin.jar
dest: /opt/jetty/lib/ext

- name: Copiar o jetty.xml
template:
src: templates/jetty.xml.j2
dest: /opt/jetty/etc/jetty.xml

- name: Copiar arquivo de contexto da aplicação
template:
src: templates/context.xml.j2
dest: "/opt/jetty/webapps/{{ app_name }}.xml"

- name: Copiar aplicação para o webapps
copy:
src: "files/{{ app_name }}.war"
dest: /opt/jetty/webapps

- name: Iniciar o Jetty
shell: /opt/jetty/bin/jetty.sh start
```

Para permitir a conexão com o banco de dados MySQL onde serão armazenadas as sessões, o Jetty precisa do driver JDBC deste banco. Este driver já deve existir na máquina local, conforme a estrutura de diretórios e arquivos exibida mais à frente. Com isso, a única etapa necessária é copiá-lo para a máquina virtual, no diretório de bibliotecas externas do Jetty.

As duas tarefas seguintes são responsáveis por copiar os arquivos de configuração do cluster do Jetty para os locais corretos. A primeira delas irá copiar o arquivo `jetty.xml` para a máquina remota.

O conteúdo deste arquivo é o padrão das instalações do Jetty, adicionando apenas o conteúdo mostrado na **Listagem 6**. Para realizar a cópia é usado o módulo **template** do Ansible, que é responsável por copiar o arquivo para o diretório informado e substituir as variáveis contidas no mesmo. A variável `{{ app_name }}` é usada pelo módulo **template** para nomear o arquivo de acordo com o nome da aplicação, permitindo assim o uso da mesma receita para publicar diferentes aplicações.

A terceira tarefa irá copiar o arquivo de contexto da aplicação que contém a configuração do gerenciador de sessão (**Listagem 7**), e as duas últimas tarefas são responsáveis por publicar a aplicação e iniciar o servlet container.

Ao término da execução da receita, a máquina remota terá o Jetty rodando e compartilhando as sessões com outras instâncias do cluster através de um banco de dados, além de uma aplicação já publicada e disponível para acesso.

Agora, esta receita pode ser executada em uma série de máquinas virtuais (ou físicas), criando um grande cluster. No entanto, é necessário ainda um balanceador de carga para encaminhar as requisições para as instâncias do cluster.

Nginx, o balanceador de carga

O Nginx é um servidor HTTP e proxy reverso que se tornou muito popular devido à sua agilidade para atender as requisições HTTP, flexibilidade e fácil configuração. Por ser um servidor de proxy reservo, como já citado, ele pode ser empregado para fazer o平衡amento de carga dos servidores.

O balanceamento de carga é uma alternativa para otimizar a utilização de recursos, maximizar o rendimento, reduzir a latência e diminuir possíveis falhas. Felizmente todas essas vantagens podem ser obtidas com o Nginx, pois ele possui uma maneira simples de configurar tal recurso para distribuir o tráfego de acordo com um dos três mecanismos analisados a seguir:

- **round-robin**: nesta opção é criada uma fila circular de servidores disponíveis. Quando uma nova requisição é recebida, ela é direcionada para o primeiro servidor disponível. Após atender a requisição, esse servidor é enviado para o final da fila e aguarda a sua vez de atender outra requisição;
- **least-connected**: com esta opção as requisições recebidas pelo balanceador de carga serão direcionadas para o servidor com menos conexões ativas;
- **ip-hash**: nesta opção o servidor que irá atender a requisição é definido a partir do endereço IP do cliente, usando uma função de espalhamento para distribuir a carga entre os servidores de acordo com a capacidade de cada um.

Quando nenhum mecanismo de balanceamento de carga é definido, por padrão o Nginx adota o round-robin.

Na **Listagem 11** podemos verificar a configuração do balanceador de carga através da diretiva **upstream** do Nginx. Esta diretiva utiliza o nome ou endereço IP dos servidores para definir um grupo que poderá receber as requisições do balanceador. Ainda na **Listagem 11**, observe que o grupo contendo os nomes ou

endereços IPs foi criado a partir de uma estrutura de repetição do Ansible que percorre uma lista de servidores definida na variável **nodes**.

De acordo com esta configuração, todas as requisições recebidas pelo balanceador de carga serão encaminhadas para o grupo de instâncias **app-example** definido na diretiva **upstream**, e o Nginx irá distribuir as requisições conforme o mecanismo selecionado, neste caso, **round-robin**.

Listagem 11. Configuração do balanceador de carga (app-example.j2).

```
upstream app-example {  
    {% for node in nodes %}  
        server {{ node }};  
    {% endfor %}  
}  
  
server {  
    listen 80;  
    server_name localhost;  
  
    location / {  
        proxy_pass http://app-example;  
    }  
}
```

Listagem 12. Receita para instalar e configurar o Nginx (nginx.yml).

```
- hosts: all  
  sudo: yes  
  
  tasks:  
    - name: Instalar Nginx  
      apt:  
        pkg: nginx  
        state: present  
  
    - name: Configurar Nginx  
      template:  
        src: templates/app-example.j2  
        dest: /etc/nginx/sites-enabled/default  
  
    - name: Reiniciar o Nginx  
      service:  
        name: nginx  
        state: restarted
```

A instalação e configuração do Nginx é baseada na receita exibida na **Listagem 12**. Como todos os módulos do Ansible demonstrados nesta receita já foram adotados anteriormente, os mesmos não serão detalhados novamente. O fluxo de execução desta receita é: realizar a instalação do Nginx e suas dependências; copiar o arquivo com a configuração do balanceamento de carga (**Listagem 11**) para o diretório do Nginx; e finalmente, reiniciar o serviço do Nginx.

Vagrant e Ansible, trabalho em conjunto

Para obter os benefícios da virtualização e provisionamento durante a construção de um ambiente virtual, podemos empregar o Vagrant e o Ansible em conjunto. Através dessas ferramentas, a criação e provisionamento das máquinas utilizadas no ambiente virtual se torna rápida e eficiente.

DevOps: Como criar ambientes virtuais para testes

Listagem 13. Vagrantfile, arquivo único para criação do ambiente.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|


# Variaveis =====
ip_prefix = "192.168.0.12"
mysql_ip = "192.168.0.120"
number_of_instances = 2
instance_ips = []

# MySQL =====
config.vm.define "mysql" do |mysql|
  mysql.vm.box = "trusty-64"
  mysql.vm.box_url = "http://goo.gl/iUfnh"
  mysql.vm.hostname = "mysql"
  mysql.vm.network :private_network, ip: mysql_ip

  mysql.vm.provision "ansible" do |ansible|
    ansible.extra_vars = {
      ansible_ssh_user: "vagrant"
    }
    ansible.playbook = "mysql.yml"
  end
end

# Jetty =====
1.upto(number_of_instances) do |num|
  name = ('jetty-' + num.to_s).to_sym
  config.vm.define name do |node|
    node.vm.box = "trusty-64"
  end

  node.vm.box_url = "http://goo.gl/iUfnh"
  node.vm.hostname = name
  node.vm.network :private_network, ip: ip_prefix + num.to_s

  node.vm.provision "ansible" do |ansible|
    ansible.extra_vars = {
      ansible_ssh_user: "vagrant",
      node_name: name,
      app_name: "app-example",
      mysql_ip: mysql_ip,
      ansible_ssh_user: "vagrant"
    }
    ansible.playbook = "jetty.yml"
  end
  instance_ips.push(ip_prefix + num.to_s << ":8080")
end

# Nginx =====
config.vm.define "nginx" do |nginx|
  nginx.vm.box = "trusty-64"
  nginx.vm.box_url = "http://goo.gl/iUfnh"
  nginx.vm.hostname = "nginx"
  nginx.vm.network "forwarded_port", guest: 80, host: 8000

  nginx.vm.provision "ansible" do |ansible|
    ansible.extra_vars = {
      ansible_ssh_user: "vagrant",
      nodes: instance_ips
    }
    ansible.playbook = "nginx.yml"
  end
end
end
```

Dito isso, o primeiro passo é criar o Vagrantfile para informar as configurações das máquinas virtuais, além das receitas que serão adotadas no momento em que as máquinas virtuais forem criadas.

O Vagrantfile apresentado na **Listagem 13** é um pouco mais complexo do que os exibidos nas **Listagens 1 e 2**. Note que além de fazer uso das opções de provisionamento, também foi empregado o recurso de multimáquinas do Vagrant. Este recurso deve ser adotado quando existe a necessidade de criar máquinas virtuais que precisam interagir entre si.

Na parte inicial do arquivo foram definidas algumas variáveis que serão utilizadas para especificar o prefixo dos IPs das máquinas virtuais, o endereço IP adotado pelo MySQL, a quantidade de instâncias disponíveis no cluster do Jetty e os IPs das instâncias que o Nginx deve considerar no momento do balanceamento de carga.

Em seguida, existem três trechos de código que serão responsáveis por definir as máquinas virtuais do MySQL, Jetty e Nginx. As especificações em comum direcionam o Vagrant sobre qual o nome do box que deverá ser utilizado. Se esse box ainda não foi importado para o Vagrant, então o box será baixado e importado através do valor definido para o **box.url**. Além da definição do box, cada máquina virtual terá um nome único.

A máquina virtual do MySQL será provisionada de acordo com a receita **mysql.yml** (**Listagem 5**) e fará uso de um endereço IP privado. Este endereço será definido uma única vez através da variável **mysql_ip**. É a partir deste endereço que torna-se possível

```
node.vm.box_url = "http://goo.gl/iUfnh"
node.vm.hostname = name
node.vm.network :private_network, ip: ip_prefix + num.to_s

node.vm.provision "ansible" do |ansible|
  ansible.extra_vars = {
    node_name: name,
    app_name: "app-example",
    mysql_ip: mysql_ip,
    ansible_ssh_user: "vagrant"
  }
  ansible.playbook = "jetty.yml"
end
instance_ips.push(ip_prefix + num.to_s << ":8080")
end

# Nginx =====
config.vm.define "nginx" do |nginx|
  nginx.vm.box = "trusty-64"
  nginx.vm.box_url = "http://goo.gl/iUfnh"
  nginx.vm.hostname = "nginx"
  nginx.vm.network "forwarded_port", guest: 80, host: 8000

  nginx.vm.provision "ansible" do |ansible|
    ansible.extra_vars = {
      ansible_ssh_user: "vagrant",
      nodes: instance_ips
    }
    ansible.playbook = "nginx.yml"
  end
end
end
```

a comunicação com as demais máquinas virtuais do ambiente.

A configuração das máquinas virtuais do Jetty é um pouco mais complexa. Visando uma maior flexibilidade para aumentar e diminuir a quantidade de instâncias, esta configuração foi feita em uma estrutura de repetição do Vagrant, onde o número de instâncias do cluster é definido em uma variável. Assim como na definição da máquina virtual do MySQL, será atribuído um endereço IP privado para cada máquina, e este endereço IP será composto pelo valor atribuído à variável **ip_prefix** concatenado com o índice da instância na iteração. A receita utilizada para criar as máquinas virtuais do Jetty será a especificada em **jetty.yml** (veja as **Listagens 8, 9 e 10**), receita esta que precisa de algumas variáveis que serão definidas pelo Vagrant e enviadas para o Ansible através da opção **ansible.extra_vars**.

Logo após a definição das máquinas virtuais do Jetty, existe a definição de uma única máquina com o Nginx para fazer o balanceamento de carga. Ao invés de atribuir um endereço IP para a máquina virtual, neste caso, será feito um redirecionamento de porta entre a máquina física (**host**) e a máquina virtual (**guest**). Este redirecionamento permite acessar a máquina responsável pelo平衡ador de carga utilizando o endereço IP da máquina física. Esta receita utilizará uma variável chamada **instance_ips**, que armazena todos os endereços IPs das máquinas virtuais que fazem parte do cluster Jetty.

Após a definição de todas as máquinas, é possível identificar através da opção **ansible.playbook** quais receitas serão utiliza-

das para cada uma das máquinas virtuais. Como algumas dessas receitas referenciam arquivos de configuração e instalação em diretórios específicos, para que a criação do ambiente virtual seja concluída com sucesso, a estrutura de diretórios para a criação do ambiente virtual deve ser semelhante à exposta na **Listagem 14**.

Listagem 14. Estrutura de diretórios para criar o ambiente virtual.

```

files
├── app-example.war
├── mysql-connector-java-5.1.32-bin.jar
├── app-example.j2
├── context.xml.j2
└── jetty.xml.j2

jetty.yml
mysql.yml
nginx.yml
templates

Vagrantfile

```

Para criar o ambiente virtual com as duas instâncias do Jetty em cluster, o banco de dados MySQL para armazenar as sessões e o Nginx para fazer o平衡amento de carga, basta executar o comando: `vagrant up --provision`. A **Figura 6** exibe um trecho da saída deste comando, referente ao provisionamento da primeira máquina virtual com o servlet container Jetty.

Após a conclusão da criação das máquinas virtuais e do provisionamento, o estado de cada máquina virtual criada pelo recurso multimáquinas pode ser listado através do comando `vagrant status`, conforme demonstra a **Figura 7**.

Em um ambiente multimáquinas do Vagrant, todos os comandos Vagrant para criar, acessar, destruir, suspender ou recarregar uma máquina virtual serão executados em todas as máquinas virtuais. Para executar o comando em apenas uma máquina virtual, é necessário informar o nome desta, por exemplo: `vagrant ssh jetty-1`.

Validando software através do ambiente virtual

Para avaliar o ambiente criado de uma maneira simples, será implementada uma aplicação exemplo que armazena um identificador na sessão. Desta maneira será possível validar o ambiente virtual, pois o balanceador de carga irá distribuir as requisições entre as instâncias do Jetty e a sessão será compartilhada entre as instâncias através do banco de dados.

O servlet exibido na **Listagem 15** irá gerar um identificador composto pelo nome da instância do Jetty que respondeu a requisição do cliente e a data/hora dessa requisição. Este identificador será adicionado na sessão para permitir identificar a instância do Jetty que respondeu a primeira requisição do cliente. Além disso, será incluído na resposta para o cliente o nome e o endereço IP da instância que atendeu a atual requisição.

Na **Figura 8** é possível visualizar através do **Server ID** que a primeira requisição desse cliente foi direcionada para a instância jetty-2. Já a segunda requisição foi redirecionada para a instância jetty-1.

```

--> jetty-1: Running provisioner: ansible...
PLAY [all] ****
GATHERING FACTS ****
ok: [jetty-1]

TASK: [Adicionar repositório do Java 8] ****
changed: [jetty-1]

TASK: [Aceitar automaticamente a licença Oracle] ****
changed: [jetty-1]

TASK: [Instalar Pacotes] ****
changed: [jetty-1] => (item=oracle-java8-installer,unzip)

TASK: [Baixar Jetty] ****
changed: [jetty-1]

TASK: [Descompactar Jetty] ****
changed: [jetty-1]

TASK: [Renomear diretório do Jetty] ****
changed: [jetty-1]

TASK: [Copiar driver jdbc para o Jetty] ****
changed: [jetty-1]

TASK: [Copiar o jetty.xml] ****
changed: [jetty-1]

TASK: [Copiar arquivo de contexto da aplicação] ****
changed: [jetty-1]

TASK: [Copiar aplicação para o webapps] ****
changed: [jetty-1]

TASK: [Iniciar o Jetty] ****
changed: [jetty-1]

PLAY RECAP ****
jetty-1 : ok=12    changed=11   unreachable=0    failed=0

```

Figura 6. Execução da receita jetty.yml

```

[leocomelli] ~/Dev/envs/devmedia/virtual-env$ vagrant status
Current machine states:

mysql                running (virtualbox)
jetty-1              running (virtualbox)
jetty-2              running (virtualbox)
nginx               running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.

[leocomelli] ~/Dev/envs/devmedia/virtual-env$ 

```

Figura 7. Estado das máquinas virtuais após a criação e provisionamento



Figura 8. Acesso à aplicação de exemplo

DevOps: Como criar ambientes virtuais para testes

```
mysql> use sessions;
Database changed
mysql> select sessionId, virtualHost, lastNode, accessTime, lastAccessTime, createTime, cookieTime, expiryTime from JettySessions;
+-----+-----+-----+-----+-----+-----+-----+-----+
| sessionId | virtualHost | lastNode | accessTime | lastAccessTime | createTime | cookieTime | expiryTime |
+-----+-----+-----+-----+-----+-----+-----+-----+
| jetty-11s9ri2kwun3kcqls32501dno0 | 0.0.0.0 | jetty-1 | 1412824624867 | 1412824515074 | 1412824511724 | 0 | 1412826424867 |
| jetty-11wphtp1bh51yd15wbeur8lvszo | 0.0.0.0 | jetty-2 | 1412824520371 | 1412824520371 | 1412824520371 | 0 | 1412826320419 |
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Figura 9. Sessões armazenadas no MySQL

Nota

As informações adicionadas na sessão e na requisição do cliente são essenciais para identificar o compartilhamento de sessão entre as instâncias do Jetty.

Listagem 15. Servlet da aplicação de exemplo.

```
@WebServlet(urlPatterns = "/")
public class AppServlet extends HttpServlet{
    private static final long serialVersionUID = 7694214469295240550L;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        req.setAttribute("serverName", req.getLocalName());
        req.setAttribute("serverIp", getLocalHost().getHostName());

        HttpSession session = req.getSession();
        if(session.getAttribute("serverId") == null){
            session.setAttribute("serverId", getLocalHost().getHostName() + "-"
                + LocalDate.now());
        }

        RequestDispatcher rd = req.getRequestDispatcher("index.jsp");
        rd.forward(req, resp);
    }
}
```

Como as sessões estão sendo compartilhadas através de um banco de dados, a segunda requisição manteve o valor atribuído ao **Server ID** na primeira requisição.

O armazenamento das sessões no banco de dados pode ser visualizado na **Figura 9**, que exibe o resultado de uma rápida consulta ao MySQL. Entre as colunas interessantes da tabela *JettySession* estão: o identificador da sessão, a última instância que atendeu a requisição, o timestamp da criação, o último acesso e a expiração da sessão.

Ter conhecimento para navegar por várias áreas de um projeto de software é muito interessante. Com o destaque que DevOps tem tido no mercado atual, conseguir unir desenvolvimento e infraestrutura pode proporcionar uma melhor execução de uma tarefa ou projeto, tendo em vista a grande ligação que existe entre essas áreas.

Além de utilizar a virtualização e provisionamento para criar ambientes virtuais de validação e testes, essas ferramentas podem facilitar a execução de projetos em ambientes diferentes, por exemplo: em um projeto que suporta múltiplos bancos de dados, é possível definir uma máquina virtual no Vagrant e uma receita no Ansible para cada banco de dados, viabilizando assim a criação

das máquinas virtuais com determinado banco de dados apenas quando necessário.

Os conceitos e ferramentas descritos neste artigo permitem a criação de ambientes virtuais mais simples ou até complexos para serem utilizados em diversos testes de validação de software. Note que a partir deles, independentemente do nível de complexidade do ambiente de produção, é totalmente viável criar um ambiente virtual respeitando a arquitetura de cada projeto e exigindo uma quantidade menor de recursos.

Por fim, é interessante destacar que as receitas criadas e utilizadas para a criação de um ambiente virtual de testes podem ser utilizadas em produção, buscando automatizar a criação do ambiente de produção nas máquinas físicas ou em instâncias de serviços cloud, como o EC2 da Amazon.

Autor



Leonardo Comelli

leonardo.comelli@gmail.com – <http://leocomelli.com.br>

Graduado em Ciências da Computação e Especialista em Banco de Dados. Atualmente trabalha como Líder Técnico de Transição e Transformação na HP.



Links:

Site da ferramenta VirtualBox

<https://www.virtualbox.org>

Página do projeto Vagrant

<https://www.vagrantup.com>

Página principal do Vagrant Boxes

<http://www.vagrantbox.es>

Página do projeto Ansible

<http://www.ansible.com>

Documentação sobre os módulos Ansible.

<http://docs.ansible.com/modules.html>

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



DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

FORMAÇÃO DESENVOLVEDOR **JAVA**

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

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer

