



Programação reativa com RxJava
Como criar soluções assíncronas com traços
de programação funcional

Edição 145 :: R\$ 14,90

DEV MEDIA

Automatize a geração de pacotes com o Jenkins

Gerando pacotes em poucos
clics com Ant, Maven e Gradle

Dominando o Spring Boot

Garantia de qualidade e produtividade em seus projetos

MONEY API

Conheça um dos
novos recursos
do Java 9



Web services
RESTful com Jersey
Aprenda a desenvolver
aplicações com serviços

WildFly - Do básico ao
ambiente de produção
Características e configurações
para tunar suas aplicações



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 145 • 2015 • ISSN 1676-8361



MVP

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

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Diogo Souza (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araújo

Distribuição

FC Comercial e Distribuidora S.A.

Rua Teodoro da Silva, 907, Grajaú - RJ

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

Atendimento ao leitor

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

Publicidade

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

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

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

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

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



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

CURSOS ONLINE

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



CONHEÇA ALGUNS DOS CURSOS:

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

Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Sumário

Conteúdo sobre Boas Práticas, Artigo no estilo Solução Completa

06 – Como implementar web services RESTful com Jersey

[Carlos Antônio Martins e Nelio Alves]

Conteúdo sobre Boas Práticas, Conteúdo sobre Novidades

19 – Programação reativa com a biblioteca RxJava

[Rafael Toledo]

Conteúdo sobre Novidades

30 – Money API: Conheça um dos novos recursos do Java 9

[Marlon Silva Carvalho]

Conteúdo sobre Boas Práticas

38 – WildFly - Do básico ao ambiente de produção

[Adriano Schmidt]

Conteúdo sobre Boas Práticas, Artigo no estilo Solução Completa

46 – Dominando o Spring Boot

[Willian Oki]

Conteúdo sobre Boas Práticas, Artigo no estilo Solução Completa

58 – Automatizando a geração de pacotes com o Jenkins

[Rafael Campos Lima]



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



DEVMEDIA

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior
portal para
desenvolvedores
da América
Latina!



20
mil
posts

430
mil
cadastrados

10
milhões de
page-views
por mês

Como implementar web services RESTful com Jersey

Veja neste artigo como desenvolver uma aplicação completa com serviços RESTful em Java

A integração entre aplicações é uma necessidade cada vez mais presente em nosso dia a dia, necessidade esta que motivou a criação de diferentes abordagens para a “integração de dados”. Dentre tais abordagens, temos soluções mais simples, como utilizar um banco de dados compartilhado ou realizar a troca de arquivos, até soluções mais elaboradas, que fazem uso de objetos distribuídos, como COM (*Component Object Model*) e CORBA (*Common Object Request Broker Architecture*).

No entanto, nem a solução COM e nem a solução CORBA conseguiu padronizar de forma satisfatória a integração entre diferentes plataformas (Windows, Linux, entre outros) e diferentes linguagens de programação (PHP, C#, Java, etc.). Diante desse cenário, os web services surgiram como uma ótima solução e com eles tornou-se possível a troca de informações entre sistemas de forma padronizada. Como exemplo, podemos citar as aplicações de comércio eletrônico, que obtêm informações do endereço do cliente através do serviço dos Correios dado um CEP sem conhecer a plataforma onde encontra-se o serviço e em qual linguagem de programação ele foi implementado.

Entre as abordagens existentes atualmente para a implementação de web services estão o protocolo SOAP (*Simple Object Access Protocol*) e o REST (*Representational State Transfer*) e ambos funcionam sobre o protocolo HTTP, protocolo padrão para a transferência ou troca de dados na web. Os web services implementados com SOAP usam um protocolo de mesmo nome para a transferência de mensagens no formato XML sobre o protocolo HTTP e esse XML é constituído basicamente

Fique por dentro

Neste artigo será mostrado como desenvolver uma aplicação Java web com serviços RESTful. Para isso, abordaremos os principais conceitos do estilo arquitetural REST, bem como seus princípios e restrições. Falaremos também sobre a API JAX-RS (ou Java API for RESTful Web Services) e sua implementação de referência: o Jersey. Por fim, ao final deste artigo você estará apto a configurar o ambiente de desenvolvimento e saberá como criar a aplicação propriamente dita, compreendendo as partes do código mais relevantes para esse tipo de implementação.

por um elemento raiz chamado *Envelope* o qual contém dois sub elementos, *Header* e *Body*.

O elemento *Header* é opcional na mensagem e dentro dele são passadas informações de controle como, por exemplo, dados para assinatura digital, IP de origem, dados para autenticação, entre outros. Já o elemento *Body* é obrigatório e contém a informação a ser transportada para seu destino final, como o nome do método que será invocado e o objeto da requisição serializado. Além disso, ele pode conter um elemento opcional chamado *Fault*, usado para carregar as mensagens de status e erros nas transações. Como no protocolo SOAP a transferência de mensagens entre origem e destino se dá somente através de mensagens XML, haverá um processamento considerável tanto na origem quanto no destino para transcrever os elementos de controle, além do necessário para serializar e desserializar os objetos contidos na mensagem, tornando tal protocolo mais lento. Tal fato se torna ainda mais relevante em situações onde há limitações de recursos (hardware e memória, por exemplo), o que indica que a adoção de SOAP nestes casos não é uma escolha adequada.

Por outro lado, os web services REST são mais simples e leves, pois usam somente o protocolo HTTP para a transferência de mensagens, não adotando elementos adicionais de controle e nem elementos adicionais para o envio das mensagens além dos existentes no protocolo HTTP. Outro diferencial é que eles são mais flexíveis, pois não impõem restrições no formato das mensagens, ou seja, o desenvolvedor pode optar pelo formato mais adequado à sua necessidade, como JSON, texto, XML, entre outros. Com isso, aplicações que necessitam de simplicidade na troca de informações e maior flexibilidade no desenvolvimento se beneficiam da arquitetura REST como, por exemplo, serviços remotos acessados via smartphones e PDAs.

A partir do que foi analisado, serão apresentados neste artigo os principais conceitos relacionados à arquitetura REST e demonstrado, na prática, como é simples desenvolver uma aplicação com esta arquitetura. Para isso, será adotada a IDE NetBeans, a IDE Eclipse juntamente com o servidor web Tomcat, JAX-RS e o Jersey.

REST

O REST é um estilo arquitetural para sistemas distribuídos que utiliza o protocolo HTTP e outras tecnologia existentes na web, como XML, JSON, entre outras, para a criação de web services. Esse termo foi citado pela primeira vez por Roy Fielding (um dos criadores do protocolo HTTP) no ano de 2000, em sua tese de doutorado. A ideia por trás do mesmo é que em vez de usar mecanismos complexos como CORBA ou SOAP para a troca de mensagens entre cliente e servidor, utiliza-se o protocolo HTTP, opção mais simples e fácil e que já possui os recursos necessários para tal.

Em uma arquitetura baseada em REST, tudo é tido como recursos (elementos de informação), os quais são identificados por URIs (*Uniform Resource Identifiers*). Um recurso pode ser qualquer informação, seja ela os dados de uma pessoa, de uma tarefa ou mesmo um post em um blog. Por exemplo, a previsão do tempo de uma determinada cidade pode ser considerada um recurso. Na internet, recursos são normalmente representados como uma página HTML, um arquivo de imagem ou mesmo um documento XML, os quais podem ser acessados utilizando uma interface comum baseada nos métodos do protocolo HTTP: GET, PUT, POST e DELETE.

Neste momento é importante destacar que na arquitetura REST os recursos são totalmente desacoplados do formato retornado para o cliente. Assim, um registro do banco de dados no servidor pode ser enviado ao cliente como um arquivo XML, JSON, uma página HTML, entre outras opções. Atualmente, os formatos mais utilizados para representar os recursos são: JSON, XML, páginas HTML, documentos PDF e arquivos JPG, diversidade esta que facilita a integração com diferentes tipos de clientes, como browsers, smartphones, tablets, aplicativos desktop, entre outros.

Ressalta-se ainda que o REST é um estilo arquitetural e não um modelo formalmente definido, como muitos pensam. Note que não existe uma definição detalhada e padronizada de como a troca de mensagens entre cliente e servidor deve ocorrer. Isto varia de

acordo com a interpretação de cada pessoa. Em outras palavras, cada empresa/desenvolvedor pode definir seus serviços com um estilo próprio, embora deva considerar todos os princípios e restrições que a arquitetura descreve, originando, nestes casos, os web services RESTful.

Princípios do REST

Conforme definido por Roy Fielding, o REST possui um conjunto de princípios que descrevem como os padrões da web, como HTTP e URIs, devem ser usados. A argumentação é que a adesão a estes princípios em um projeto faz com que o sistema explore melhor a arquitetura web para benefício próprio. Em resumo, os cinco princípios que norteiam o REST são:

1. Utiliza o conceito de recursos. Um recurso é tudo que possa ser requisitado a um servidor. Por exemplo, um arquivo texto, uma página HTML, uma imagem, etc.;
2. Cada recurso deve possuir um ID para identificá-lo na interação entre as partes envolvidas. O mais comum é utilizar o URI do documento;
3. Uso de padrões como HTTP, HTML ou XML;
4. Um recurso pode ter várias representações, como XML, texto, HTML, JSON, entre outras, as quais são conhecidas como media types;
5. A comunicação entre cliente e servidor é feita através do protocolo HTTP e sem estado (stateless), ou seja, cada requisição deve conter todas as informações necessárias para que o servidor consiga entendê-la e processá-la adequadamente, sem utilizar informações vindas de outras requisições.

Restrições do estilo REST

Segundo Roy Fielding, para que os princípios anteriormente apresentados sejam respeitados, um conjunto de restrições deve ser seguido. Portanto, na iteração entre os componentes dos sistemas, as restrições a seguir devem estar presentes (as cinco primeiras restrições são obrigatórias e a última é opcional):

- 1. Cliente-Servidor:** Ao separar os papéis do cliente (consumidor do serviço) dos papéis do servidor (provedor do serviço), melhoramos a portabilidade da interface do usuário em ambientes multiplataforma e a escalabilidade pela simplificação dos componentes no servidor;
- 2. Stateless (sem estado):** O servidor não deve armazenar informações de contexto para atender as requisições dos clientes;
- 3. Interface Uniforme:** Significa que os recursos gerenciados serão manipulados de maneira uniforme, ou seja, a informação é transferida de forma padronizada para todas as aplicações e não apenas para uma em específico;
- 4. Multicamada:** Permite a composição da arquitetura em camadas hierárquicas, de forma a restringir o comportamento dos componentes, impedindo que cada um “veja” além da camada com a qual ele interage diretamente;
- 5. Cache:** Como muitos clientes acessam um mesmo servidor, e muitas vezes requisitando os mesmos recursos, é necessário que estas respostas possam ser armazenadas em cache, evitando o

Como implementar web services RESTful com Jersey

processamento desnecessário e aumentando significativamente a performance;

6. Code-on-Demand: Permite aumentar as funcionalidades no cliente através da execução de códigos sobre demanda carregados do servidor na forma de applets ou scripts, ou seja, possibilita ao cliente executar alguma lógica atribuída ao servidor.

JAX-RS

O Java dá suporte ao REST através das JSRs 311 e 339. A partir da JSR 311 surgiu a JAX-RS 1.0 e a partir da JSR 339 surgiu a JAX-RS 2.0, as quais são as APIs Java para o desenvolvimento de web services RESTful. Essas APIs disponibilizam um conjunto de anotações, classes, interfaces e métodos que podem ser usados para definir recursos web e ações que podem ser executadas sobre eles. A **Tabela 1** lista as anotações mais importantes da JAX-RS e suas descrições.

Quando a JAX-RS 1.0 foi lançada em 2008, sua especificação não tinha como objetivo a definição de uma API Cliente. Assim, a criação da parte cliente dos serviços desenvolvidos com essa versão deveriam utilizar outras soluções para a implementação das classes que invocassem os serviços RESTful, sendo necessário, ainda, implementar as conversões de objetos para XML e vice-versa. Com o JAX-RS 2.0, essas operações e algumas outras se tornaram mais fáceis de serem realizadas em virtude de tais conversões já estarem disponíveis na API, possibilitando, principalmente, um ganho maior de produtividade no desenvolvimento dos serviços.

Vejamos as novas funcionalidades disponíveis a partir da versão 2.0:

1. API Cliente: A JAX-RS 1.0 era voltada estritamente para o lado servidor e no lado cliente utilizava-se outras soluções (APIs) para consumir os serviços REST. Na versão 2.0 foi adicionado um construtor para clientes (“builder”) a partir do qual uma instância de cliente pode ser obtida chamando o método **newClient()** da classe **ClientBuilder**;

2. Suporte a chamadas assíncronas: Permite ao cliente realizar requisições ao servidor e executar outras operações enquanto aguarda por uma notificação de resposta. Isso é feito através das interfaces **Future** e **InvocationCallback** da API;

3. Hipermídia: Criação das classes **Link** e **Target** para inserir hyperlinks dentro da resposta fornecida pelo servidor para permitir a navegação entre recursos;

4. Novas anotações: Novas anotações foram inseridas, como:

- **@NameBinding:** serve para criar anotações associadas a filters ou interceptors, a qual pode ser usada em métodos ou classes. Por exemplo, se uma anotação associada a um interceptor for aplicada a determinado método, o interceptor será executado a cada chamada àquele método;

- **@Suspended:** essa anotação indica que o método em questão executará chamadas assíncronas;

- **@BeanParam:** essa anotação serve para agrupar um conjunto de parâmetros da requisição em uma classe bean definida pelo desenvolvedor. Isso é feito para evitar que um método tenha um número muito grande de parâmetros em sua assinatura.

5. Validação: Serve para verificar se os dados da requisição obedecem a uma ou mais restrições pré-definidas e é utilizada através de anotações nos parâmetros do método. Por exemplo, a anotação **@NotNull** indica que o parâmetro assim anotado não pode ser nulo;

6. Filtros e Interceptadores: Filters são usados principalmente para modificar ou processar o conteúdo *Header* (cabeçalho) de requisições a métodos ou respostas de métodos. Eles podem ser utilizados tanto no lado cliente quanto no lado servidor. Para a implementação de filters no lado servidor deve-se utilizar as interfaces **ContainerRequestFilters** e **ContainerResponseFilters** presentes na API JAX-RS, sendo que o filter que implementa a primeira será executado antes da invocação do método e o filter que implementa a segunda interface será executado depois da invocação do método. De forma semelhante, para implementação de filters do lado cliente deve-se utilizar as interfaces

Anotação	Descrição
@Path	Esta anotação define a URI relativa a um método ou classe Java presente em um servidor. Por exemplo, podemos anotar uma classe com @Path("Service") e um método dentro dessa classe com @Path("/Service/hello") . Além disso, é possível especificar parâmetros para a URI, como: <code>/Service/hello/{username}</code> , onde username seria o parâmetro passado para o método hello() .
@GET	Essa anotação especifica que o método anotado irá lidar com requisições correspondentes a HTTP GET. Portanto, um método que receba essa anotação processará apenas requisições do tipo GET.
@POST	Essa anotação especifica que o método anotado irá lidar com requisições correspondentes a HTTP POST. Portanto, um método que receba essa anotação processará apenas requisições do tipo POST.
@PUT	Essa anotação especifica que o método anotado irá lidar com requisições correspondentes a HTTP PUT. Portanto, um método que receba essa anotação processará apenas requisições do tipo PUT.
@DELETE	Essa anotação especifica que o método anotado irá lidar com requisições correspondentes a HTTP DELETE. Portanto, um método que receba essa anotação processará apenas requisições do tipo DELETE.
@PathParam	Anotação usada para passar valores como parâmetros para os métodos. Por exemplo: podemos passar um ID para um método para obter um recurso.
@Produces	Anotação usada para definir o formato do recurso (<i>media type</i>) que será produzido pelo método anotado com @GET . Por exemplo: com a opção “text/plain”, o método produzirá para o cliente uma resposta no formato texto.
@Consumes	Anotação usada para definir o formato do recurso (<i>media type</i>) que será consumido pelo método. Por exemplo: com a opção “text/xml”, o método consumirá os dados no formato XML.

Tabela 1. Anotações mais importantes do JAX-RS e suas descrições

ClientRequestFilter e **ClientResponseFilters**, sendo que o filter que implementa **ClientRequestFilter** é executado antes da requisição HTTP ser enviada para o servidor e o filter que implementa **ClientResponseFilters** é executado depois que a mensagem de resposta foi recebida do servidor, mas antes dela ser deserializada. Como exemplo de aplicabilidade, podemos citar a escrita em logging, autenticação, autorização, entre outras funções.

Para mais informações sobre as novas funcionalidades da JAX-RS 2.0, veja o site oficial desta especificação.

Jersey

Como visto anteriormente, de acordo com as JSRs especificadas, implementações são criadas por diferentes fornecedores. Assim acontece com a implementação do JSF, da JPA e não seria diferente para o desenvolvimento de web services RESTful em Java. Com o intuito de possibilitar o desenvolvimento desse tipo de serviço de forma mais fácil e produtiva, foram implementadas soluções como o Jersey, Apache CXF, JBoss RESTEasy, entre outras. Como o Jersey é a implementação de referência da API JAX-RS 2.0, o adotaremos para a construção do nosso exemplo.

Além das opções que devem ser fornecidas de acordo com a especificação, o Jersey adiciona novas características e módulos, de forma a simplificar ainda mais o desenvolvimento, tanto do lado cliente quanto do lado servidor. Para mais informações a respeito, veja a documentação do Jersey no site oficial (veja a seção **Links**).

Outro diferencial é que o Jersey suporta uma variedade de servidores, desde servlet containers (os servidores web) a application servers (os servidores de aplicação). No entanto, quando a aplicação é publicada em um servlet container, como o Apache Tomcat, ela é empacotada em um arquivo WAR e as bibliotecas da API JAX-RS e do Jersey devem estar presentes no mesmo, uma vez que elas não fazem parte do conjunto de bibliotecas do container. Além disso, o servlet container deve fornecer o suporte à especificação servlet 2.5 ou posterior, pois a servlet implementada pelo Jersey, chamada **ServletContainer**, requer tal especificação para funcionamento. Por outro lado, caso seja usado no serviço o suporte a recursos assíncronos da JAX-RS 2.0, a versão 3.0 ou posterior é necessária. Em alguns servidores de aplicação, por outro lado, não existe a necessidade de empacotar as bibliotecas junto à aplicação, uma vez que elas já estão integradas ao servidor, como é o caso do GlassFish.

Descrição do cenário para implementação da parte prática

Após conhecer os principais conceitos relacionados aos serviços RESTful, iniciaremos a parte prática, a qual se baseia no desenvolvimento de uma aplicação que possibilitará ao leitor visualizar como o Jersey funciona, assim como alguns dos seus recursos.

Para exemplificar o uso deste framework, será desenvolvido um chat que, no servidor, terá um serviço RESTful que atenderá as solicitações do cliente e manterá as informações de mensa-

gens trocadas e usuários logados em uma base de dados, e no cliente, teremos duas janelas, sendo uma para o login e outra do chat propriamente dito. Esta última janela conterá um campo para o usuário digitar as mensagens e um botão para enviá-las, um campo para exibir os usuários participantes do chat e outro para exibir todas as mensagens trocadas entre eles. E na janela de login, serão informados o host e a porta do servidor de acesso, bem como um nome para o usuário que realizará a troca de mensagens.

O servidor será construído utilizando a IDE Eclipse, o JDK 7 e o Tomcat 8.0.15 e o cliente será construído utilizando a IDE NetBeans, o JDK 7 e a API Swing. As **Figuras 1** e **2** demonstram as janelas do cliente.

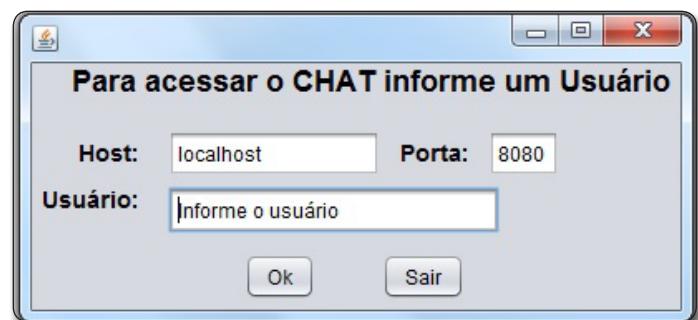


Figura 1. Janela do cliente: Login

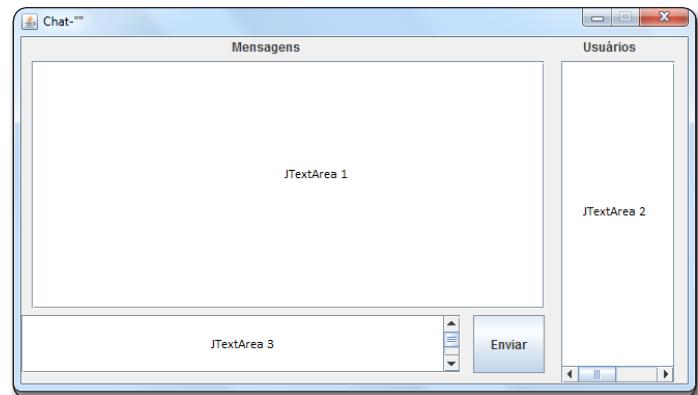


Figura 2. Janela do cliente

Configuração do ambiente

Para desenvolver o chat, faz-se necessário instalar os softwares que auxiliarão na implementação e também obter as bibliotecas que permitirão construir os serviços REST. A configuração do ambiente como um todo se baseia em duas etapas: *Download e instalação das bibliotecas* e *Configuração do ambiente de desenvolvimento*, que foi dividida em outras duas fases: *Configuração do projeto servidor* e *Configuração do projeto cliente*.

Download e instalação das bibliotecas

A lista a seguir traz o conjunto de ferramentas necessárias para o desenvolvimento da parte prática:

Como implementar web services RESTful com Jersey

1. JDK 1.7 ou superior;
2. Eclipse Luna;
3. NetBeans 8.0.2;
4. Tomcat 8.

Além destas, é preciso realizar o download do Jersey que, como vimos, será a solução REST utilizada. Para isso, basta acessar a seção de download na página do projeto e baixar o arquivo *jaxrs-ri-2.17.zip*. Este contém todas as bibliotecas que desejamos para o nosso exemplo.

Configuração do ambiente de desenvolvimento

As configurações no servidor se limitam a criar o projeto do tipo web, informar o servidor que adotaremos (no caso o Tomcat) e também o conjunto de bibliotecas necessárias ao desenvolvimento. Também de modo simples, as configurações da parte cliente se limitam a criar o projeto do tipo desktop e configurar o classpath deste com todas as bibliotecas necessárias ao desenvolvimento.

Em nosso exemplo, o projeto servidor utilizará o Eclipse, enquanto o projeto cliente adotará o NetBeans. A opção pelas duas IDEs não está relacionada a uma limitação técnica, pois ambas atendem muito bem ao propósito deste artigo. A escolha foi de cunho pessoal, uma vez que o NetBeans oferece recursos melhores quando lidamos com a programação para desktop. Portanto, o leitor pode escolher apenas uma delas ou usar as duas, conforme apresentado.

A seguir são apresentados os passos necessários para a configuração dos projetos nas IDEs.

Configuração do projeto servidor

Para configurar o ambiente onde será desenvolvido o servidor da aplicação, será utilizada a IDE Eclipse Luna. Vejamos os passos necessários:

1. No Eclipse, acesse a opção de menu *Window > Preferences*. Na janela que abrir, procure por *Server* e escolha *Runtime Environments*. Em seguida, clique no botão *add* e selecione o Tomcat 8.0.15 para o server runtime e clique em *Ok*;
2. Feito isso, acesse *File > New > Dynamic Web Project*. Na janela que for aberta, nomeie o projeto como “*WSServer*”, escolha o Tomcat 8.0.15 em *Target Runtime*, a opção 3.1 em *Dynamic web module version* e clique em *Next* duas vezes;
3. No final do processo, marque a opção *Generate web.xml deployment descriptor* para que seja criado o arquivo *web.xml* para o projeto e clique em *Finish*. O arquivo *web.xml* servirá para informar o servlet do Jersey, bem como os parâmetros de inicialização do mesmo;
4. Acesse a pasta onde foi realizado o download do arquivo *jaxrs-ri-2.17.zip* e o descompacte. Logo após, será gerada uma pasta chamada *jaxrs-ri*, a qual conterá três subpastas: *api*, *ext* e *lib*;
5. Acesse a subpasta *api* e copie o arquivo *javax.ws.rs-api-2.0.1.jar* para a pasta *WEB-INF/lib*;
6. Acesse a subpasta *lib* e copie os arquivos *jersey-common.jar*, *jersey-container-servlet.jar*, *jersey-container-servlet-core.jar*, *jersey-media-jaxb.jar* e *jersey-server.jar* para a pasta *WEB-INF/lib*;

7. Ao final do processo esta pasta estará configurada conforme a **Figura 3**;
8. Para configuração do classpath da aplicação, clique com o botão direito sobre o projeto e acesse *Build Path > Configure Build Path*. Em seguida, na aba *Libraries*, adicione todos os arquivos com extensão *JAR* presentes na pasta *WEB-INF/lib*;
9. Pronto! O ambiente para desenvolvermos a aplicação do lado servidor já está configurado.

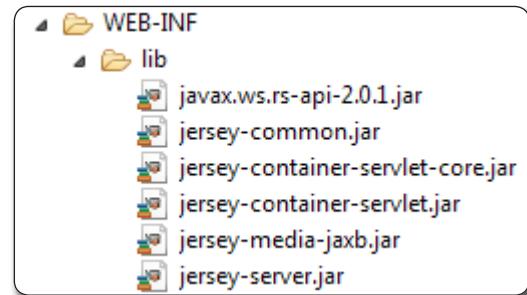


Figura 3. Bibliotecas necessárias para o servidor

Configuração do projeto cliente

Para configurar o ambiente onde será desenvolvido o cliente, execute os seguintes passos:

1. No NetBeans, acesse a opção de menu *Arquivo > Novo Projeto*. Em seguida, escolha em *Categorias* a opção *Java*, em *Projetos, Aplicação Java* e clique em *Próximo*;
2. Informe o nome do projeto como “*WSClient*”, desmarque a opção *Criar Classe Principal* e clique em *Finalizar*;
3. Na aba *Projetos*, clique com o botão direito sobre *WSClient* e escolha *Novo > Pasta*. Em *Nome da Pasta*, informe “*lib*”;
4. Agora, acesse a aba *Arquivos* e verifique se a pasta *lib* foi criada. Confirmada essa questão, navegue até a pasta *jaxrs-ri* (veja o passo 4 do projeto servidor) e copie os arquivos com extensão *.jar* mostrados na **Figura 4** para a pasta *lib*;
5. Retorne para a aba *Projetos*, clique com o botão direito sobre *Bibliotecas* e escolha *Adicionar JAR/Pasta*. Logo após, selecione todos os arquivos com extensão *.jar* da pasta *lib* do projeto e clique em *Abrir*;
6. No final do processo, a pasta conterá as bibliotecas necessárias para execução da aplicação.

Desenvolvendo o chat

Com os ambientes de desenvolvimento prontos, podemos iniciar, de fato, a construção da aplicação. De forma semelhante à configuração, este processo será dividido em duas etapas: *Implementação da aplicação servidor* e *Implementação da aplicação cliente*, o que permitirá uma explicação mais detalhada do cenário proposto.

Em cada uma delas serão criadas as classes e métodos necessários ao funcionamento dos dois lados da aplicação: cliente e servidor. Assim, os passos necessários para a implementação do código e as explicações necessárias serão tratados nos tópicos conforme o grau de relevância de cada ponto.

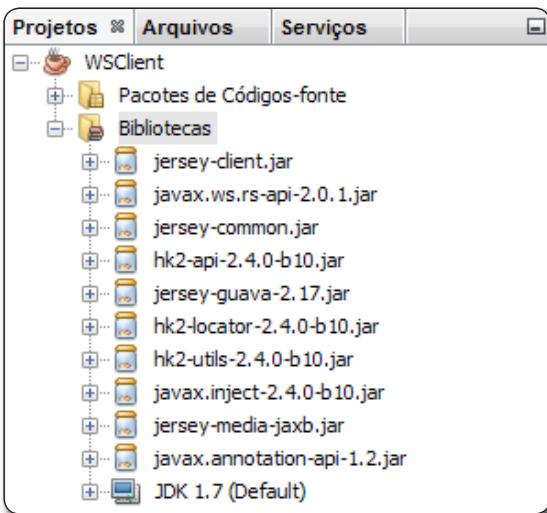


Figura 4. Bibliotecas necessárias para o projeto cliente

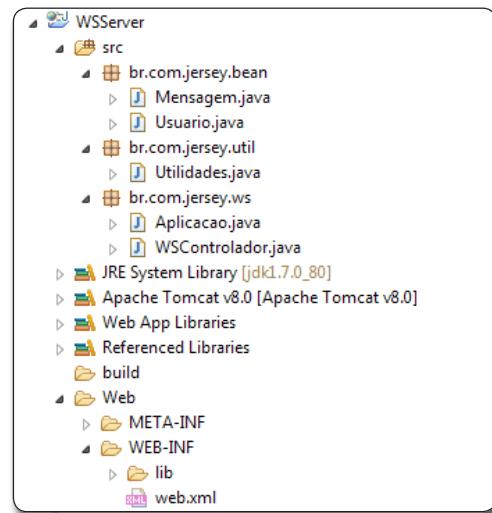


Figura 5. Visão final do projeto servidor

Implementação da aplicação servidor

Vejamos a seguir os passos para implementação do código referente à parte servidor:

1. Crie três pacotes no projeto WSServer, chamados **br.com.jersey.bean**, **br.com.jersey.util** e **br.com.jersey.ws**;
2. Crie a classe **WSControlador** dentro do pacote **br.com.jersey.ws** e a implemente conforme descrito na **Listagem 1**;
3. Crie as classes **Mensagem** e **Usuario** dentro do pacote **br.com.jersey.bean** e as implemente conforme descrito nas **Listagens 2** e **3**, respectivamente;
4. Crie a classe **Utilidades** no pacote **br.com.jersey.util**. Seu código é apresentado na **Listagem 4**;
5. Crie a classe **Aplicacao** no pacote **br.com.jersey.ws**. Seu código é apresentado na **Listagem 5**;
6. Substitua o conteúdo do arquivo **web.xml**, existente no diretório **WEB-INF**, pelo conteúdo da **Listagem 6**;
7. Salve todos os arquivos e clique em **Project > Clean** para que o código criado seja compilado;
8. Publique o projeto no servidor Tomcat clicando sobre o mesmo e escolhendo a opção **Run as > Run on Server**.

Ao final, o servidor conterá todas as classes e métodos necessários para o funcionamento do chat e teremos o projeto com uma situação semelhante à **Figura 5**.

Entendendo o código

Se tudo foi criado corretamente, o serviço RESTful já estará funcionando e poderá ser utilizado. Comecemos então a explicar o código, iniciando nossa análise pela classe principal do servidor: **WSControlador** (vide **Listagem 1**). É nesta classe onde são definidos todos os recursos que podem ser acessados pelo cliente. Na linha 1, através da anotação **@Path("/WSControlador")**, especificamos **WSControlador** como um recurso raiz (*Root Resource Classes*) da aplicação, ou seja, a partir dela o cliente poderá acessar outros recursos no servidor.

Um recurso raiz é uma classe POJO (*Plain Old Java Object*) anotada com **@Path** e que possui pelo menos um método anotado com **@Path** ou algum método anotado com **@GET**, **@PUT**, **@POST** ou **@DELETE**. Em nosso caso, podemos observar que **WSControlador** tem sete recursos que podem ser acessados pelo cliente, como explicita a anotação HTTP existente sobre eles. Vejamos:

- **hello()**: a anotação **@Path** sobre ele indica que esse método responderá pela URI relativa **/WSControlador/hello** e **@GET** sinaliza que ele processará uma requisição do tipo GET. Além disso, **hello()** produzirá como resposta um objeto do tipo **String** que será convertida para o formato texto conforme descrito na anotação **@Produces**;
- **addMensagem()**: a anotação **@Path** sobre ele indica que esse método responderá pela URI relativa **/WSControlador/addMensagem** e **@PUT** sinaliza que ele processará uma requisição do tipo PUT. Além disso, **addMensagem()** receberá como parâmetro um objeto do tipo **Mensagem** que será convertido para o formato XML conforme descrito na anotação **@Consumes**. O objetivo deste método é adicionar mensagens ao chat de forma que os participantes vejam o histórico de mensagens trocadas;
- **addUsuario()**: a anotação **@Path** sobre esse método indica que ele responderá pela URI relativa **/WSControlador/addUsuario** e **@PUT** sinaliza que ele processará uma requisição do tipo PUT devido à anotação. Além disso, **addUsuario()** receberá como parâmetro um objeto do tipo **Usuario**, que será convertido para o formato XML conforme descrito na anotação **@Consumes**. O objetivo deste método é adicionar usuários ao chat de modo que o nome dos participantes seja conhecido;
- **getUsuarios()**: a anotação **@Path** sobre ele indica que esse método responderá pela URI relativa **/WSControlador/getUsuarios** e **@GET** sinaliza que ele processará uma requisição do tipo GET. Além disso, **getUsuarios()** produzirá como resposta uma lista de objetos do tipo **Usuario** que serão convertidos para o formato XML conforme descrito por **@Produces**. A função deste método é obter todos os usuários presentes no chat;

Como implementar web services RESTful com Jersey

Listagem 1. WSControlador: classe que conterá todos os recursos do serviço.

```
01. @Path("/WSControlador")
02. public class WSControlador {
03.
04.     @GET
05.     @Path("/hello")
06.     @Produces("text/plain")
07.     public String hello(){
08.         System.out.println("Dizendo hello ...");
09.         return "Hello World!!! Isto eh um teste\n";
10.    }
11.
12.    @PUT
13.    @Path("/addMensagem")
14.    @Produces("application/xml")
15.    public void addMensagem(Message msg){
16.        Utilidades.add(msg);
17.        System.out.println("Adicionando mensagem:" + msg.getMsg());
18.    }
19.
20.    @PUT
21.    @Path("/addUsuario")
22.    @Produces("text/plain")
23.    public void addUsuario(Usuario user){
24.        Utilidades.adicionaUsuario(user);
25.        //Adiciona a mensagem indicando que o usuário entrou na sala
26.        Message mens = new Message();
27.        mens.setUser("");
28.        mens.setMsg(user.getNome() + " entrou na sala.");
29.        Utilidades.add(mens);
30.        System.out.println("Adicionando user:" + user);
31.    }
32.
33.    @GET
34.    @Path("/getUsuarios")
```



```
35.     @Consumes("application/xml")
36.     public List<Usuario> getUsuarios(){
37.         System.out.println("getUsuarios chamado.");
38.         return Utilidades.getUsuarios();
39.     }
40.
41.     @GET
42.     @Path("/getMensagens")
43.     @Consumes("application/xml")
44.     public List<Message> getMensagens(){
45.         System.out.println("getMensagens chamado.");
46.         return Utilidades.getMessages();
47.     }
48. }
49.
50. @DELETE
51. @Path("/delUsuario")
52. @Consumes("text/plain")
53. public void delUsuario(@QueryParam("user") String user){
54.     Utilidades.removeUsuario(user);
55.     Message mens = new Message();
56.     mens.setUser("");
57.     mens.setMsg(user + " saiu da sala.");
58.     Utilidades.add(mens);
59.     System.out.println("removendo user:" + user);
60. }
61.
62. @DELETE
63. @Path("/delTodos")
64. public void delTodos(){
65.     Utilidades.removeAll();
66.     System.out.println("removendo todas as msg e users");
67. }
68. }
```

- **getMensagens()**: a anotação `@Path` indica que esse método responderá pela URI relativa `/WSControlador/getMensagens` e `@GET` sinaliza que ele processará uma requisição do tipo GET. Além disso, `getMensagens()` produzirá como resposta uma lista de objetos do tipo **Mensagem** que serão convertidos para o formato XML conforme descrito em `@Produces`. A função deste método é obter o histórico de mensagens trocadas;
- **delUsuario()**: a anotação `@Path` sinaliza que esse método responderá pela URI relativa `/WSControlador/delUsuario` e `@DELETE` indica que ele processará uma requisição do tipo DELETE. Além disso, `delUsuario()` receberá como parâmetro um objeto do tipo **String** que será convertido para o formato de texto conforme descrito por `@Consume`. Esse método não retornará nenhuma informação e possui a função de remover um usuário do chat;
- **delTodos()**: a anotação `@Path` indica que esse método responderá pela URI relativa `/WSControlador/delTodos` e `@DELETE` explica que ele processará uma requisição do tipo DELETE. A função deste método é remover todos os usuários presentes no chat.

Além disso, temos as classes **Mensagem** e **Usuario**, apresentadas nas Listagens 2 e 3. Em **Mensagem**, temos dois atributos: `user` e `msg`. O atributo `user` guarda o nome do usuário que

digitou a mensagem e o atributo `msg` guarda a mensagem em si. Por sua vez, em **Usuario** temos apenas um atributo, chamado `nome`, que representa o nome do usuário logado. Além disso, ambas as classes são anotadas com `@XmlElement`, para que seja realizada a conversão pela API JAXB de objeto Java para XML e vice-versa. Para mais informações a respeito de `@XmlElement`, veja a documentação do JAXB.

Para armazenar todas as mensagens enviadas e todos os usuários logados, foi criada uma classe utilitária chamada **Utilidades**, conforme expõe o código da Listagem 4. Esta classe servirá como uma base de dados. Ela contém dois atributos `static` do tipo lista: **mensagens** e **usuarios**, e alguns métodos para realizar operações nestes atributos, como adicionar, remover e buscar algum usuário ou mensagem. Por se tratar de uma listagem simples, os comentários sobre a mesma serão dispensados.

Em seguida, temos a classe **Aplicacao**, descrita na Listagem 5, a qual é utilizada para declaração do recurso raiz da aplicação, sendo anotada com `@ApplicationPath("")`. Além disso, ela estende da classe abstrata **Application**, provida pela API JAX-RS. Como podemos verificar na linha 6, o método `add()` adiciona **WSControlador** como recurso raiz no serviço RESTful, e a partir dele o cliente pode acessar outros recursos realizando chamadas a métodos de **WSControlador** conforme a necessidade.

Listagem 2. Mensagem: classe utilizada para representar as mensagens trocadas no chat.

```
01. @XmlRootElement(name = "message")
02. public class Mensagem {
03.
04.     private String user;
05.     private String msg;
06.
07.     public Mensagem() {
08.         super();
09.     }
10.    @XmlElement
11.    public String getUser() {
12.        return user;
13.    }
14.    public void setUser(String user) {
15.        this.user = user;
16.    }
17.    @XmlElement
18.    public String getMsg() {
19.        return msg;
20.    }
21.    public void setMsg(String msg) {
22.        this.msg = msg;
23.    }
24. }
```

Listagem 3. Usuario: classe utilizada para representar os usuários no chat.

```
01. @XmlRootElement(name = "usuario")
02. public class Usuario {
03.
04.     private String nome;
05.     @XmlElement
06.     public String getName() {
07.         return nome;
08.     }
09.     public void setName(String nome) {
10.         this.nome = nome;
11.     }
12. }
```

Já uma classe provider implementa a conversão de objetos para algum media type que não pode ser realizada pela JAX-RS de forma padrão, embora esta especificação forneça suporte a muitos objetos da linguagem Java, tais como **String**, **Integer**, **Float**, entre outros. Assim, tais conversões precisam ser customizadas para que a serialização/desserialização de objetos seja realizada quando requisições e respostas forem processadas. Para isso, a entidade customizada deve implementar a interface **MessageBodyWriter<T>** ou a interface **MessageBodyReader<T>**, existentes na API JAX-RS, sendo que a primeira permite realizar a serialização e a segunda a desserialização de objetos. Além disso, a entidade deve ser adicionada ao contexto da aplicação através da chamada ao mesmo método **add()**, utilizado para a declaração do recuso raiz. Para mais informações sobre como criar classes provider, veja a documentação do Jersey.

Por último, temos o arquivo *web.xml* (vide **Listagem 6**), o qual é responsável pela configuração do servlet do Jersey para que tudo funcione no chat. Assim, no elemento **<servlet-name>** informamos um nome para o servlet, que neste caso será **WSServer**, e em **<servlet-class>** informamos ao container de servlet (neste caso o Tomcat) qual a classe do Jersey ele deve executar quando a aplicação iniciar. Em seguida temos **<init-param>**, onde informamos ao servlet **WSServer** que o mesmo deve iniciar executando a classe **Aplicacao** que, como vimos, define o recurso base como sendo **WSControlador**. Finalmente, em **<url-pattern>**, dentro de **<servlet-mapping>**, definimos a URL que o servlet **WSServer** irá interceptar quando chegar uma requisição do tipo *http://<host:port>/WSServer/* ao servidor.

The advertisement features a large, stylized title 'RENOVE JÁ!' at the top, followed by the subtitle 'Sua assinatura pode estar acabando'. Below this, there's a cartoon illustration of a man with a shocked expression, holding a newspaper that says 'EXTRA!'. To the right, there's a call to action: 'Renovando a assinatura de sua revista favorita você ganha brindes e descontos exclusivos.' At the bottom, several magazine covers are shown, including Java Magazine, Mobile Magazine, .NET Magazine, SQL Magazine, and Club Delphi. The DevMedia logo is in the bottom right corner.

Como implementar web services RESTful com Jersey

Listagem 4. Utilidades: classe utilizada para representar a base de dados da aplicação.

```
01. public class Utilidades {  
02.  
03.     private static List<Message> mensagens = new ArrayList<Message>();  
04.     private static List<Usuario> usuarios = new ArrayList<Usuario>();  
05.  
06.     public Utilidades () {  
07.         super();  
08.     }  
09.  
10.    public static Message getMessage(int index)  
11.    {  
12.        return Utilidades.mensagens.get(index);  
13.    }  
14.  
15.    public static List<Message> getMessages()  
16.    {  
17.        return Utilidades.mensagens;  
18.    }  
19.  
20.    public static void add(Message msg)  
21.    {  
22.        Utilidades.mensagens.add(msg);  
23.    }  
24.  
25.    public static Usuario getUser(Usuario user)  
26.    {  
27.        for(Usuario u : ServiceUtils.usuarios)  
28.        {  
29.            if(u.getNome().equalsIgnoreCase(user.getNome()))  
30.                return u;  
31.        }  
32.        return null;  
33.    }  
34.}
```



```
35.    public static void adicionaUsuario(Usuario user)  
36.    {  
37.        Utilidades.usuarios.add(user);  
38.        return;  
39.    }  
40.  
41.    public static List<Usuario> getUsuarios()  
42.    {  
43.        return Utilidades.usuarios;  
44.    }  
45.  
46.    public static void removeUsuario(String user)  
47.    {  
48.        Iterator<Usuario> itr = Utilidades.usuarios.iterator();  
49.        while(itr.hasNext()) {  
50.            Usuario usuario = itr.next();  
51.            System.out.println("removendo:" + user + "- na fila:" + usuario.getNome());  
52.            if(user.equalsIgnoreCase(usuario.getNome()))  
53.            {  
54.                Utilidades.usuarios.remove(usuario);  
55.                break;  
56.            }  
57.        }  
58.        return;  
59.    }  
60.  
61.    public static void removeAll()  
62.    {  
63.        Utilidades.usuarios.clear();  
64.        Utilidades.mensagens.clear();  
65.        return;  
66.    }  
67. }
```

Listagem 5. Aplicacao: classe utilizada para registrar o recurso raiz no serviço.

```
01. @ApplicationPath("//")  
02. public class Aplicacao extends Application {  
03.     @Override  
04.     public Set<Class<?>> getClasses() {  
05.         Set<Class<?>> classes = new HashSet<Class<?>>();  
06.         classes.add(WSController.class);  
07.         return classes;  
08.     }  
09. }
```

Listagem 6. web.xml: arquivo de configuração da aplicação.

```
<?xml version="1.0" encoding="UTF-8"?>  
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee/  
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"  
id="WebApp_ID" version="3.1">  
    <display-name>WSServer</display-name>  
    <servlet>  
        <servlet-name>WSServer</servlet-name>  
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>  
        <init-param>  
            <param-name>javax.ws.rs.Application</param-name>  
            <param-value>br.com.jersey.ws.Aplicacao</param-value>  
        </init-param>  
        <load-on-startup>1</load-on-startup>  
    </servlet>  
    <servlet-mapping>  
        <servlet-name>WSServer</servlet-name>  
        <url-pattern>/*</url-pattern>  
    </servlet-mapping>  
</web-app>
```

Implementação da aplicação cliente

Vejamos a seguir os passos para a implementação do código referente à parte cliente do Chat:

1. Crie cinco pacotes no projeto **WSClient**, chamados **br.com.jersey.constants**, **br.com.jersey.controller**, **br.com.jersey.bean**, **br.com.jersey.utils** e **br.com.jersey.view**;
2. Crie as classes **Mensagem** e **Usuario** dentro do pacote **br.com.jersey.bean**. O código destas classes é apresentado nas **Listagens 2** e **3**, respectivamente. Como estas classes possuem o mesmo código que as classes de mesmo nome no lado servidor, não as explicaremos novamente;
3. Crie as classes **Main** e **ChatView** dentro do pacote **br.com.jersey.view**. As **Listagens 7** e **8** mostram o código das mesmas;
4. Crie a classe **ServiceControl** no pacote **br.com.jersey.controller** e a implemente conforme o código da **Listagem 9**;
5. Crie a classe **Factory** no pacote **br.com.jersey.utils**. Seu código deve ficar igual ao da **Listagem 10**;
6. Crie a classe **Constantes** no pacote **br.com.jersey.constants** e a implemente conforme o código da **Listagem 11**;
7. Salve todos os arquivos e clique em **Executar > Limpar e Construir Projeto**.

Entendendo o código

Para entendermos o funcionamento do código construído no cliente, começaremos nossa análise pela classe **Main**, apresentada na **Listagem 7**. Esta classe é responsável por obter os dados infor-

mados para o host e porta do servidor onde encontra-se o serviço, o nome do usuário para login no chat (vide **Figura 1**), bem como realizar a verificação de obrigatoriedade no preenchimento deles, configurar as variáveis de conexão e iniciar a janela **ChatView**.

Após o usuário clicar no botão *Ok*, o código que inicia na linha 21 será executado e as informações da janela (usuário, host e porta) serão verificadas e configuradas em **usuario** (vide linha 34), **HOST** e **PORT** (vide linha 36 e 37). Note que a variável **usuario** pertence à própria classe **Main**, enquanto as outras duas são variáveis estáticas da classe **ServiceControl** (explorada mais adiante). Em seguida, na linha 39, é iniciada a janela principal do chat (**Figura 2**), através da criação de uma instância da classe **ChatView** (veja o código na **Listagem 8**), que recebe como parâmetro a variável **usuario**. E logo após, é adicionado o evento

WindowListener (vide linha 49), para capturar os eventos nesta janela. Neste caso, somente o evento **windowClosing** será considerado, porque precisamos remover o usuário do chat quando a janela principal for fechada.

A classe **ChatView**, como verificado, representa a janela do chat propriamente dita. Ela recebe em seu construtor o argumento **usuario**, obtido em **Main**, e configura-o no título da janela (linha 7) e na mensagem (linha 9), representada pelo atributo privado **msg**. Depois é executado o método **addUser()**, da classe **ServiceControl** (linha 11), para adição deste usuário no servidor, bem como são realizadas duas buscas: uma para recuperar todas as mensagens já digitadas, através do método **initMensagens()** (linha 12), e outra para recuperar todos os participantes do bate papo, através do método **initUsers()**, na linha 13.

Listagem 7. Main: classe utilizada para criar a janela de login.

```

01. public class Main extends javax.swing.JFrame {
02.
03.     private String usuario;
04.     //Demais variáveis da classe
05.
06.     public String getUsuario() {
07.         return usuario;
08.     }
09.
10.    public void setUsuario(String usuario) {
11.        this.usuario = usuario;
12.    }
13.
14.    //Método gerado pelo NetBeans
15.    private void initComponents() {...}
16.
17.    private void bntSairActionPerformed(java.awt.event.ActionEvent evt) {
18.        System.exit(0);
19.    }
20.
21.    private void bntOkActionPerformed(java.awt.event.ActionEvent evt) {
22.        //Condições iniciais que verificam a obrigatoriedade de preenchimento dos
23.        //campos host, porta e usuário
24.        if (".".equals(txtUsuario.getText())) {
25.            JOptionPane.showMessageDialog(rootPane,
26.                "Nome do usuário não informado", "Informe usuário", WIDTH);
27.
28.        } else if (".".equals(txtHost.getText())) {
29.            JOptionPane.showMessageDialog(rootPane,
30.                "Nome do Host não informado", "Informe Host", WIDTH);
31.
32.        } else if (".".equals(txtPorta.getText())) {
33.            JOptionPane.showMessageDialog(rootPane,
34.                "Nome da Porta não informado", "Informe Porta", WIDTH);
35.
36.        } else {
37.            //Captura o nome do usuário digitado e seta na variável 'usuario' da classe.
38.            this.setUsuario(txtUsuario.getText());
39.            //Seta o host e a porta para a conexão
40.            ServiceControl.HOST = txtHost.getText();
41.            ServiceControl.PORT = txtPorta.getText();
42.            //inicia a janela do chat com o título igual ao nome do usuário
43.            ChatView view = new ChatView(usuario);
44.            //posiciona a janela no meio da tela
45.            view.setLocation(
46.                ((Toolkit.getDefaultToolkit().getScreenSize().width / 2)
47.                - (this.getWidth() / 2)),
48.                ((Toolkit.getDefaultToolkit().getScreenSize().height / 2)
49.                - (this.getHeight() / 2))
50.            );
51.
52.            //adiciona o evento na janela, o qual será disparado quando
53.            //a mesma for fechada.
54.            view.addWindowListener(new WindowListener() {
55.
56.                public void windowActivated(WindowEvent e) {
57.
58.                }
59.
60.                public void windowClosed(WindowEvent e) {
61.
62.                }
63.
64.                //evento disparado quando a janela for fechada.
65.                //Remove o usuário do chat no servidor
66.                public void windowClosing(WindowEvent e) {
67.                    System.out.println("Removendo:" + usuario);
68.                    ServiceControl.delUser(usuario);
69.                    System.exit(0);
70.
71.                }
72.            });
73.        }
74.
75.        public static void main(String args[]) {
76.
77.            java.awt.EventQueue.invokeLater(new Runnable() {
78.                public void run() {
79.                    new Main().setVisible(true);
80.                }
81.            });
82.        }
83.    }

```

Como implementar web services RESTful com Jersey

Listagem 8. ChatView: classe utilizada para representar a janela do chat.

```
01. public class ChatView extends javax.swing.JFrame {  
02.  
03.     private Message msg;  
04.     //Demais variáveis da classe  
05.  
06.     public ChatView(String usuario) {  
07.         super.setTitle("Chat"+ usuario);  
08.         msg = new Message();  
09.         msg.setUser(usuario);  
10.         Usuario user = new Usuario(usuario);  
11.         ServiceControl.addUser(user);  
12.         initMensagens();  
13.         initUsers();  
14.         initComponents();  
15.     }  
16.  
17.     public void initMensagens() {  
18.         new Thread() {  
19.             @Override  
20.             public void run() {  
21.                 while(true)  
22.                 {  
23.                     try {  
24.                         Thread.sleep(5000);  
25.                         List<Message> list = ServiceControl.getMessages();  
26.                         String mensagem = "";  
27.                         for(Message m : list)  
28.                         {  
29.                             if(m.getUser().equals("")){  
30.                                 // Mensagens de entrada e saída da sala.  
31.                                 //Por exemplo:>-> maria entrou na sala"  
32.                                 mensagem = mensagem + ">->" + m.getMsg() + "\n";  
33.                             }  
34.                             else{  
35.                                 //Mensagem de conversa entre os usuários.  
36.                                 //Por exemplo: -> maria diz: ola !!!  
37.                                 mensagem = mensagem + ">->" + m.getUser() + " diz:  
38.                                 + m.getMsg() + "\n";  
39.                             }  
40.                         txtMensagens.setText(mensagem);  
41.                     } catch (InterruptedException ex) {  
42.                         ex.printStackTrace();  
43.                         JOptionPane.showMessageDialog(  
44.                             null,"Erro ao buscar mensagens", null, WIDTH);  
45.                     }  
46.                 }.start();  
47.  
48.     public void initUsers() {  
49.         new Thread() {  
50.             @Override  
51.             public void run() {  
52.                 while(true)  
53.                 {  
54.                     try {  
55.                         Thread.sleep(5000);  
56.                         List<Usuario> list = ServiceControl.getUsers();  
57.                         System.out.println("List usuario size:" + list.size());  
58.                         String usuarios = "";  
59.                         for(Usuario u : list)  
60.                         {  
61.                             usuarios = usuarios + u.getNome() + "\n";  
62.                         }  
63.                         System.out.println("Usuarios:" + usuarios);  
64.                         txtUsuarios.setText(usuarios);  
65.                     } catch (InterruptedException ex) {  
66.                         ex.printStackTrace();  
67.                         JOptionPane.showMessageDialog(null,"Erro ao buscar mensagens",  
68.                             null, WIDTH);  
69.                     }  
70.                 }  
71.             }.start();  
72.         }  
73.     }  
74.  
75.     //Método gerado pelo NetBeans  
76.     private void initComponents() {...}  
77.  
78.     private void bntSendActionPerformed(java.awt.event.ActionEvent evt) {  
79.         msg.setMsg(txtEnviar.getText());  
80.         txtEnviar.setText("");  
81.         ServiceControl.addMessage(msg);  
82.     }  
83.  
84.     private void bntSendKeyPressed(java.awt.event.KeyEvent evt) {  
85.         if (evt.getKeyCode()==KeyEvent.VK_ENTER){  
86.             msg.setMsg(txtEnviar.getText());  
87.             txtEnviar.setText("");  
88.             ServiceControl.addMessage(msg);  
89.         }  
90.     }  
91.  
92.     private void txtEnviarKeyPressed(java.awt.event.KeyEvent evt) {  
93.         if (evt.getKeyCode()==KeyEvent.VK_ENTER){  
94.             msg.setMsg(txtEnviar.getText());  
95.             txtEnviar.setText("");  
96.             ServiceControl.addMessage(msg);  
97.         }  
98.     }  
99. }
```

Outro ponto a ser observado nesta listagem são os métodos dos eventos adicionados no botão *Enviar* e na área de texto. O evento **bntSendActionPerformed()**, cujo código inicia na linha 78, será executado quando o usuário clicar no botão *Enviar* e adicionará uma mensagem no servidor através da chamada ao método **addMessage()** (linha 81). Por sua vez, os eventos **bntSendKeyPressed()** e **txtEnviarKeyPressed()** serão executados para o envio de mensagens ao servidor sempre que o *Enter* for pressionado e o respectivo elemento (botão ou área de texto) estiver com o foco do cursor.

Dando continuidade à nossa análise, temos a classe **ServiceControl**, cujo código pode ser visto na Listagem 9. Nesta classe temos implementadas todas as funções necessárias para o funcionamento do chat, as quais servirão para a interação com o servidor. São elas:

1. **hello():** serve apenas como um método de teste e retornará para o cliente um objeto do tipo **String**;
2. **addMessage():** serve para adicionar uma mensagem no servidor;
3. **addUser():** serve para adicionar um usuário no servidor;

Listagem 9. ServiceControl: classe utilizada para realizar as chamadas HTTP ao serviço.

```

01. public class ServiceControl {
02.
03.     public static String HOST = "";
04.
05.     public static String PORT = "";
06.
07.     public static String hello() {
08.         Client client = Factory.getInstance();
09.         WebTarget webTarget = client.target("http://localhost:8080");
10.         WebTarget resourceWebTarget = webTarget.path(Contantes.RESOURCE_HELLO);
11.         Response response = resourceWebTarget.request
12.             (MediaType.TEXT_PLAIN_TYPE).get();
13.         if (response.getStatus() != 200) {
14.             throw new RuntimeException("Failed : HTTP error code : "
15.                 + response.getStatus());
16.         }
17.         String output = response.readEntity(String.class);
18.         return output;
19.     }
20.
21.     public static void addMessage(Message msg) {
22.         Client client = Factory.getInstance();
23.         WebTarget webTarget = client.target("http://" + HOST + ":" + PORT +
24.             Contantes.RESOURCE_ROOT);
25.         WebTarget resourceWebTarget = webTarget.path
26.             (Contantes.RESOURCE_ADD_MESSAGE);
27.         resourceWebTarget.request().put(Entity.entity
28.             (msg, MediaType.APPLICATION_XML), Response.class);
29.
30.     }
31.
32.     public static void addUser(Usuario user) {
33.         Client client = Factory.getInstance();
34.         WebTarget webTarget = client.target("http://" + HOST + ":" + PORT +
35.             Contantes.RESOURCE_ROOT);
36.         WebTarget resourceWebTarget = webTarget.path
37.             (Contantes.RESOURCE_ADD_USER);
38.         resourceWebTarget.request().put(Entity.entity
39.             (user, MediaType.APPLICATION_XML), Response.class);
40.
41.     }
42.     public static List<Usuario> getUsers() {
43.         Client client = Factory.getInstance();
44.         WebTarget webTarget = client.target("http://" + HOST + ":" + PORT +
45.             Contantes.RESOURCE_ROOT);
46.         WebTarget resourceWebTarget = webTarget.path
47.             (Contantes.RESOURCE_GET_USERS);
48.         List<Usuario> usuarios = resourceWebTarget.request()
49.             .get(new GenericType<List<Usuario>>(){});
50.         return usuarios;
51.
52.     }
53.     public static List<Message> getMessages() {
54.         Client client = Factory.getInstance();
55.         WebTarget webTarget = client.target("http://" + HOST + ":" + PORT +
56.             Contantes.RESOURCE_ROOT);
57.         WebTarget resourceWebTarget = webTarget.path
58.             (Contantes.RESOURCE_GET_MSG);
59.         List<Message> msgs = resourceWebTarget.request()
60.             .get(new GenericType<List<Message>>(){});
61.         return msgs;
62.
63.     }
64.     public static void removeAll() {
65.         Client client = Factory.getInstance();
66.         WebTarget webTarget = client.target("http://" + HOST + ":" + PORT +
67.             Contantes.RESOURCE_ROOT);
68.         WebTarget resourceWebTarget = webTarget.path
69.             (Contantes.RESOURCE_DEL_ALL);
70.         resourceWebTarget.request().delete();
71.
72.     }
73.
74. }
```

4. **delUser()**: serve para excluir um usuário do servidor;
5. **getUsers()**: serve para buscar todos os usuários existentes no servidor;
6. **getMessages()**: serve para buscar todas as mensagens digitadas pelos usuários existentes no servidor;
7. **removeAll()**: serve para remover todos os usuários existentes no servidor.

Em todas as funções apresentadas podemos observar que os passos executados são semelhantes. O que muda é a URI do recurso e o método HTTP chamado (GET, PUT ou DELETE). Antes de informar a URI para se comunicar com o serviço, no entanto, devemos criar uma instância de **Client** utilizando um dos métodos estáticos da fábrica **ClientBuilder**, pertencente à API do JAX-RS (vide linha 10 da **Listagem 10**). Feito isso, podemos criar uma instância da classe **WebTarget** através da chamada ao método **target()** de **Client** para representar o recurso remoto desejado.

Quando **WebTarget** é configurado com uma URI de recurso raiz, é possível acessar todos os métodos anotados com **@Path** deste

Listagem 10. Factory: classe utilizada para criar as instâncias de Client.

```

01. public class Factory {
02.
03.     private static Client client;
04.
05.     public Factory() {
06.     }
07.     public static Client getInstance() {
08.     {
09.         if(client == null)
10.             client = ClientBuilder.newClient();
11.         return client;
12.     }
13. }
```

recurso através do método **path()** de **WebTarget** e passando como parâmetro a URI relativa ao método remoto. Assim, podemos executar a requisição HTTP para obtenção dos dados de acordo com a anotação utilizada no serviço, como podemos observar nas linhas 20 a 23 da **Listagem 9**. Tal fato pode ser verificado dentro de cada método dessa classe. Por exemplo, dentro do método **addMessage()**, iniciado na linha 19, os passos para criação do cliente

e obtenção do recurso ocorrem basicamente em quatro etapas:

1. A criação de uma instância do cliente JAX-RS, na linha 20;
2. O encapsulamento da URI que representa o contexto raiz da aplicação RESTful, na linha 21;
3. O encapsulamento da URI que representa o recurso desejado dentro do contexto raiz da aplicação RESTful, na linha 22;
4. Na linha 23, a chamada ao método `request()` inicia a requisição HTTP ao recurso encapsulado no passo 3 e, em seguida, a requisição é enviada ao servidor através do método `put()`.

Por fim, foi criada a classe **Constantes** (exposta na **Listagem 11**) para armazenar a URI do contexto raiz da aplicação RESTful e cada recurso dentro dela de forma a concentrar todas as URIs do serviço em uma única classe. Assim, cada variável estática dela representa um recurso ou URI na aplicação. Entretanto, nada impede que sejam utilizados arquivos de propriedades para armazenar as URIs de tais recursos, bem como outras abordagens que se julgar mais convenientes ao projeto.

Implementados todos os artefatos, a aplicação cliente pode ser executada clicando com o botão direito sobre a classe **Main** e escolhendo a opção *Executar Arquivo*. Feito isso, a janela da **Figura 1** será aberta para que as informações necessárias ao funcionamento do chat sejam fornecidas. Logo após, a janela da **Figura 2** será apresentada ao usuário para o envio e recebimento de mensagens. Teremos, portanto, a aplicação funcionando conforme exposto na **Figura 6**, onde temos dois usuários participando do chat: *maria* e *joao*.

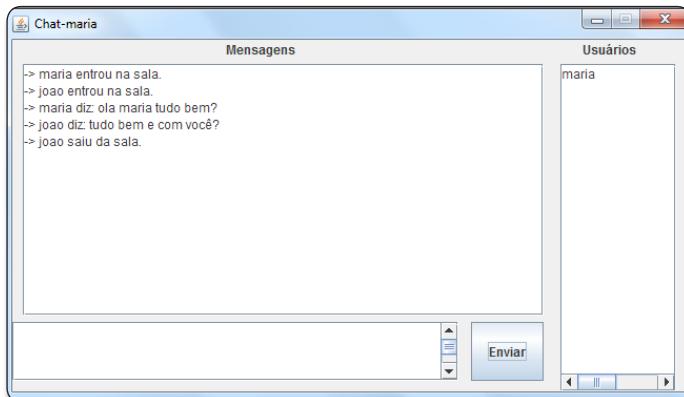


Figura 6. Exemplo da aplicação em funcionamento

Listagem 11. Constantes: classe utilizada para armazenar as URIs disponíveis no serviço.

```
01. public class Constantes {  
02.  
03.     public static final String RESOURCE_ROOT= "/WSServer/WSControlador";  
04.     public static final String RESOURCE_HELLO= "/hello";  
05.     public static final String RESOURCE_ADD_MESSAGEM= "/addMensagem";  
06.     public static final String RESOURCE_ADD_USER= "/addUsuario";  
07.     public static final String RESOURCE_DEL_USER= "/delUsuario";  
08.     public static final String RESOURCE_DEL_ALL= "/delTodos";  
09.     public static final String RESOURCE_GET_USERS= "/getUsuarios";  
10.    public static final String RESOURCE_GET_MSG= "/getMensagens";  
11.  
12. }
```

Com esse estudo pudemos verificar que construir serviços RESTful não é um bicho de sete cabeças. Entretanto, ao adotar esta opção, alguns pontos importantes devem ser considerados, como o uso correto dos verbos HTTP, o tratamento do status de retorno deve utilizar os códigos de resposta descritos no protocolo, bem como é importante fornecer descrições precisas dos erros ocorridos.

Para o leitor que deseja aprofundar-se no assunto, a seção **Links** fornece os subsídios necessários, além de apresentar exemplos de aplicabilidade. E para os leitores interessados em saber mais sobre a importância de dominar os serviços RESTful, vale ressaltar que este padrão arquitetural tem se tornado comum em grande parte das aplicações, principalmente as voltadas para comércio eletrônico e as que oferecem serviços através da computação em nuvem.

Autor



Carlos Antônio Martins

carlosmartinsufu@yahoo.com.br

É formado em Ciência da Computação pela Universidade Federal de Uberlândia (UFU) com especialização em Desenvolvimento Java pelo Centro Universitário do Triângulo (UNITRI). Trabalha na empresa Algar Telecom como Analista de Sistemas e possui sólidos conhecimentos na área de Telecomunicações.



Autor



Nélvio Alves

nelio@iftm.edu.br

É formado em Ciência da Computação pela Universidade Federal de Uberlândia (UFU) com doutorado em Engenharia de Software pela mesma instituição. É professor efetivo do Instituto Federal de Triângulo Mineiro.



Links:

Site oficial do NetBeans.

<https://netbeans.org>

Página de Roy T. Fielding

<http://www.ics.uci.edu/~fielding/>

Site oficial do JAX-RS.

<http://www.oracle.com/technetwork/articles/java/jaxrs20-1929352.html>

Site oficial do Jersey.

<https://jersey.java.net/documentation/latest/index.html>

Site oficial do JAXB.

<http://www.oracle.com/technetwork/articles/javase/index-140168.html>

Site oficial do Eclipse.

<https://eclipse.org/>

Site oficial do Tomcat.

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

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Programação reativa com a biblioteca RxJava

Criando aplicações assíncronas com traços de programação funcional de forma simples e thread safe com Java

A linguagem Java é imperativa e orientada a objetos. Sua versão 8 e a inserção, dentre outros recursos, das expressões lambda e dos streams, trouxe características da programação reativa funcional, abrindo o leque de possibilidades de se programar. Dentre essas possibilidades, destaca-se a facilidade de se trabalhar e estruturar aplicações de forma menos verbosa e mais eficiente.

Programação reativa pode ser definida como um paradigma de desenvolvimento baseado em fluxos assíncronos de dados e na propagação de mudanças nesse fluxo. Isso significa que uma determinada fonte de dados dinâmica ou estática é definida e, posteriormente ou mesmo imediatamente, executada propagando seus resultados através de um fluxo de dados e notificando todos os receptores. Pode-se definir a programação reativa como um paradigma de programação baseado em eventos que podem ser aguardados em diferentes pontos do código servindo como gatilhos para a execução de lógicas específicas.

A programação reativa funcional surgiu como uma ideia no final dos anos 90 e inspirou Erik Meijer, então funcionário da Microsoft, a criar uma forma de se utilizar este paradigma dentro do mundo .NET. Surgiu assim a biblioteca Microsoft Rx (de *Reactive eXtension*), que permite ao desenvolvedor criar aplicações assíncronas e baseadas em eventos de forma simplificada.

Com a Microsoft Rx, pode-se criar Observables, realizar queries utilizando LINQ e gerenciar a concorrência facilmente com Schedulers. Esta abordagem modifica a forma síncrona utilizada para a programação que visa o resultado através de chamadas a métodos e seu retorno. De maneira diferente, utilizando-se a Microsoft Rx, ao solicitar determinado recurso, assim que este estiver pronto, o objeto solicitante é notificado.

Em 2012, a Netflix constatou a necessidade de melhorias em seu sistema, o qual apresentava problemas de escalabilidade, devido à enorme expansão do serviço. A Netflix

Fique por dentro

Este artigo é útil por apresentar os principais componentes e conceitos que envolvem o uso da biblioteca RxJava e da programação reativa, desde a criação de um Observable, simplificação de código usando lambdas e transformações de dados. Também será possível conhecer alguns dos principais operadores do RxJava, como combinar Observables, como ter controle sobre os erros e entender o que são Subscriptions. A partir desse conteúdo o leitor terá uma boa fundamentação para iniciar nesta nova opção que vem ganhando cada vez mais atenção de profissionais que atuam com Java.

então optou por refazer a arquitetura do sistema, visando reduzir a quantidade de chamadas REST realizadas. Assim, em vez de realizar várias chamadas, seria preciso apenas uma, baseada na necessidade do cliente, o que possibilita a melhora do desempenho do sistema.

Para atingir este objetivo foi adotado o paradigma reativo e foi feita a portabilidade do sistema de *Reactive Extensions da Microsoft* para a *Java Virtual Machine*.

Nesse processo de troca de paradigmas, optou-se por focar não na linguagem Java, mas na JVM, buscando desenvolver uma ferramenta para possibilitar a programação reativa que pudesse ser utilizada por qualquer linguagem baseada na mesma JVM, como Scala, Groovy e, claro, Java. Em fevereiro de 2013, Ben Christensen e Jafar Husain, engenheiros da Netflix, mostraram a biblioteca RxJava ao mundo em um post publicado no blog técnico da empresa (veja a seção [Links](#)).

Desde então, a RxJava vem recebendo muita atenção e sendo mantida pela comunidade de forma efetiva. Inclusive, após algum tempo de desenvolvimento, foi criada uma organização para manter os diversos ports das RXs para as mais diversas plataformas e linguagens, hoje mantida sobre o nome de *Reactive Extensions* (veja a seção [Links](#)).

A RxJava possui características interessantes como facilidade de controlar concorrência, de forma a melhorar o uso do poder de processamento dos seus servidores; facilidade na execução

condicional de tarefas assíncronas; melhoria quanto a evitar o problema de uso de vários callbacks (evitar o uso excessivo do padrão Observer) e abordagem reativa.

Principais componentes da RxJava

Para fazer um bom uso da biblioteca RxJava, é necessário entender o que são e quais os papéis dos principais componentes que a compõem. Nesta seção, vamos conhecer os Observables, produtores de dados, e os Observers e Subjects, que possuem o papel de consumir e responder a esses dados.

Observable

Vamos imaginar o seguinte cenário: quando precisamos executar alguma tarefa assíncrona no Java, de baixa complexidade, quais saídas temos? Podemos utilizar Threads, Future ou algo do tipo. Porém, quando o nível de complexidade da tarefa aumenta, essas saídas tendem a se tornar complexas, confusas e difíceis de gerenciar.

Os Observables do RxJava foram projetados para suprir essa demanda. Algumas das suas principais características são flexibilidade e facilidade de uso, podendo ser encadeados, trabalhar com dados isolados ou com sequências de dados. Tanto para os casos onde a necessidade é de se emitir um valor isoladamente, quanto para uma série ou mesmo para um fluxo infinito de valores, o Observable pode ser utilizado.

O ciclo de vida de um Observable é composto de três eventos:
1. **onNext()**: disparado para notificar o Observer sobre um novo dado no fluxo do Observable;
2. **onCompleted()**: disparado para notificar o Observer de que não há mais dados a serem enviados do Observable;
3. **onError()**: disparado para notificar o Observer de que um erro ocorreu.

Para melhor entendimento do Observable, podemos dizer que ele é um tipo de Iterator assíncrono. Neste contexto, o método **onNext()** poderia ser o método **next()** do Iterator, enquanto o **onCompleted()** seria análogo ao **!hasNext()** e o **onError()** corresponderia ao caso onde o Iterator lança uma exceção, indicando que o fluxo foi interrompido e não há novos dados.

Do ponto de vista da emissão de dados, os Observables podem ser de dois tipos diferentes, “quentes” ou “frios”. Os Observables “quentes” (ou hot), são aqueles que começam o fluxo de emissão de itens imediatamente após serem criados. Desta forma, qualquer Observer que comece a observar este Observable pode começar a receber dados desse fluxo a partir de qualquer ponto. Em contrapartida, os Observables “frios” (ou cold) aguardam até que haja uma assinatura, através do método **subscribe()**, para que ele comece a produzir seu fluxo de dados.

Observers e Subjects

No RxJava, um Observer se inscreve através do método **subscribe()** para receber eventos de um Observable. A partir desta inscrição, o Observer reage a qualquer item ou sequência de itens emitida pelo Observable. No contexto do RxJava, o Observer é

uma interface que pode ser implementada por qualquer classe que precise receber eventos de um Observable.

Um Subject é um objeto coringa dentro da API do RxJava, já que pode ser um Observable e um Observer ao mesmo tempo. O Subject age como uma espécie de funil, podendo se inscrever em um Observable, agindo dessa forma como um Observer e, ao mesmo tempo, pode emitir novos itens ou simplesmente emitir os itens recebidos do Observable. Sendo também um Observable, outros Observers podem se inscrever para receber seus elementos.

Integrando a RxJava

Podemos facilmente integrar o RxJava, atualmente na versão 1.0.14, ao nosso projeto. Utilizando-se o Maven para gerenciar nossas dependências, basta modificar o arquivo **pom.xml**, conforme código mostrado na **Listagem 1**.

Listagem 1. Adicionando a RxJava ao projeto com Maven.

```
<dependency>
<groupId>io.reactivex</groupId>
<artifactId>rxjava</artifactId>
<version>1.0.14</version>
</dependency>
```

Caso esteja sendo utilizado o Gradle, a dependência pode ser adicionada conforme o código:

```
compile 'io.reactivex:rxjava:1.0.14'
```

Criando um Observable

Para iniciarmos a parte prática e vermos como funcionam os componentes do RxJava e seus principais operadores, criaremos o componente **Observable**, responsável por emitir valores ou itens que serão manipulados ao longo do artigo. Na **Listagem 2** podemos ver o seu código, que neste exemplo produzirá um fluxo de apenas um item e encerrará logo em seguida.

Listagem 2. Criando um Observable.

```
Observable<String> meuObservable = Observable.create(
    new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> subscriber) {
            subscriber.onNext("Hello World");
            subscriber.onCompleted();
        }
    }
);
```

Nosso **Observable** simplesmente emite uma mensagem “Hello World” e então conclui a sua execução. Com este **Observable** em mãos, vamos criar um **Subscriber** que possa consumir este conteúdo.

O **Subscriber** mostrado na **Listagem 3** imprimirá qualquer mensagem recebida pelo fluxo de dados do **Observable** no console. No entanto, caso ocorra algum erro ou quando o **Observable** completar a sua execução, nenhum comportamento será executado. Isso pode ser verificado ao analisar o código dos métodos

onCompleted() e **onError()**. Por fim, vale destacar que o parâmetro recebido pelo método **onError()** é um objeto do tipo **Throwable**, que pode englobar tanto classes que estendem **Exception** quanto classes que estendem **Error**.

Listagem 3. Criando um Subscriber.

```
Subscriber<String> meuSubscriber = new Subscriber<String>() {  
    @Override  
    public void onNext(String value) { System.out.println(value); }  
    @Override  
    public void onCompleted() {}  
    @Override  
    public void onError(Throwable error) {}  
};
```

Com os dois objetos criados, podemos fazer o **Subscriber** se inscrever no **Observable**, conforme o código:

```
meuObservable.subscribe(meuSubscriber);
```

O resultado desta operação será a impressão no console do valor "Hello World". Quando a inscrição é realizada, **meuObservable** chama o método **onNext()** do **Subscriber**, seguido do método **onCompleted()**. Podemos criar um outro exemplo, desta vez utilizando uma sequência de números (vide **Listagem 4**), para entendermos melhor o funcionamento de um **Observable**. Nesse caso, produziremos não apenas um, mas uma sequência de três valores para, somente então, concluirmos a produção de elementos do **Observable**, através do método **onCompleted()**.

Tendo o **Observable** pronto, no exemplo apresentado na **Listagem 5** fazemos uso da interface **Observer** (em vez da classe abstrata **Subscriber**), que também pode ser adotada para o propósito de assinarmos **Observables**.

Nesse código, temos um **Observable** que emite três valores inteiros (0, 1 e 2) e então é encerrado. No processo da inscrição, passamos como parâmetro um objeto **Observer** (uma classe anônima), que será notificado quando os eventos do **Observable** ocorrerem.

Outra forma que temos para construir **Observables** é a partir de dados que já possuímos. Por exemplo: imagine que temos uma coleção de objetos que desejamos utilizar na forma de um **Observable**, emitindo os itens um a um. Para que isso seja possível, fazemos uso do método **from()** da classe **Observable**, conforme apresentado na **Listagem 6**.

O método **from()** cria um **Observable** a partir de uma lista ou array e emite cada um dos objetos contidos no conjunto. Também é possível criar um **Observable** a partir de um parâmetro **Future**, especificando inclusive um timeout (que consequentemente realizará uma chamada ao método **onError()** do **Observer**).

Outra possibilidade de criar um **Observable** é a partir de um único valor, com o método **just()**. Este caso é bastante comum quando já possuímos código escrito de maneira imperativa e gostaríamos de reimplementá-lo de forma reativa.

Veja um exemplo com o método **just()**:

```
Observable<String> observable = Observable.just("Hello World");
```

O método **just()** ainda possui versões sobrecarregadas com até nove parâmetros, podendo adicionar vários valores que serão emitidos na mesma ordem em que são passados.

Listagem 4. Observable imprimindo números.

```
Observable<Integer> meuObservable = Observable.create(  
    new Observable.OnSubscribe<String>() {  
        @Override  
        public void call(Subscriber<? super String> subscriber) {  
            for (int i = 0; i < 3; i++) {  
                subscriber.onNext(i);  
            }  
            subscriber.onCompleted();  
        }  
    }  
);
```

Listagem 5. Utilizando a interface Observer.

```
meuObservable.subscribe(new Observer<Integer>() {  
    @Override  
    public void onCompleted() {  
        System.out.println("Observable completado");  
    }  
    @Override  
    public void onError(Throwable e) {  
        System.err.println("Ocorreu algum erro no Observable");  
    }  
    @Override  
    public void onNext(Integer value) {  
        System.out.println("O observable retornou " + value);  
    }  
});
```

Listagem 6. Método from() do Observable.

```
List<String> linguagens = new ArrayList<>();  
linguagens.add("Java");  
linguagens.add("Groovy");  
linguagens.add("Scala");  
Observable<String> observable = Observable.from(linguagens);  
Subscription sub = observable.subscribe(new Observer<String>() {  
    @Override  
    public void onCompleted() {  
        System.out.println("Observable completado");  
    }  
    @Override  
    public void onError(Throwable e) {  
        System.err.println("Ocorreu algum erro no Observable");  
    }  
    @Override  
    public void onNext(String value) {  
        System.out.println("O observable retornou " + value);  
    }  
});
```

Simplificando o código

Em alguns casos, como no exemplo dado na **Listagem 3**, não queremos nos preocupar com todos os três métodos da interface **Observable**. Pensando nisso, o RxJava nos fornece maneiras de evitar todo esse boilerplate e nos preocuparmos apenas com o callback desejado.

Para verificar essa opção, tomemos como exemplo um **Observable** simples, que simplesmente emite uma **String**, conforme o código:

```
Observable<String> meuObservable = Observable.just("Hello World");
```

Vamos então imaginar que, nesse caso, só precisamos nos preocupar com o callback **onNext()**. Queremos desconsiderar qualquer coisa que ocorra no **onCompleted()** e no **onError()**. Para isso, o RxJava nos fornece a interface **Action1**, conforme demonstra a **Listagem 7**.

Listagem 7. Interface Action1 da RxJava.

```
Action1<String> onNextAction = new Action1<String>() {
    @Override
    public void call(String value) {
        System.out.println(value);
    }
};
```

Esta interface nos permite definir separadamente cada uma das partes do callback. Através dela, podemos especificar cada um dos métodos de um **Subscriber** em objetos diferentes, como ilustrado no código:

```
meuObservable.subscribe(onNextAction, onErrorAction, onCompleteAction);
```

Porém, como neste caso só nos importa o callback do método **onNext()**, podemos passar somente um objeto ao método **subscribe()**, conforme o código a seguir. Para este exemplo, estamos passando um objeto que implementa a interface **Action1**, contudo, é possível que uma classe anônima também seja utilizada neste contexto.

```
meuObservable.subscribe(onNextAction);
```

Para melhorar a legibilidade do nosso código, podemos agrupar o processo de criação do Observable e encadear a chamada ao método de inscrição. Assim, teremos o código apresentado na **Listagem 8**. Por fim, utilizando o recurso de lambdas do Java 8, podemos deixar essa sintaxe ainda mais limpa e agradável, conforme exposto a seguir:

```
Observable.just("Hello World")
    .subscribe(value -> System.out.println(value));
```

Nota

Caso esteja em algum ambiente onde não possa utilizar Java 8 por algum motivo, podemos fazer uso do Retrolambda, que nos permite utilizar lambdas e outras vantagens do Java 8 em ambientes rodando Java 5, 6 ou 7. Vale a pena experimentar (veja a seção [Links](#)).

Transformações de dados

Uma das grandes vantagens de se utilizar a abordagem reativa na criação de aplicações é a possibilidade de transformar os dados de uma forma mais objetiva. Com esse recurso, funcionalidades como filtros, remoção de valores nulos, transformações de Strings, dentre outras, tornam-se muito mais simples de serem realizadas.

Para este exemplo, vamos supor que necessitamos de concatenar uma String "Hello World" com a hora atual do sistema. Em uma primeira abordagem, poderíamos realizar o processo de concatenação diretamente na origem de dados do **Observable**, conforme o código:

```
Observable.just("Hello World " + new Date()).subscribe(value -> System.out
    .println(value));
```

Essa abordagem funciona, mas somente em casos onde se possui controle sobre o fluxo de dados. No entanto, em casos onde isso não ocorre (quando, por exemplo, temos um fluxo vindo da rede ou de um arquivo), essa abordagem se tornaria inválida. Neste caso, poderíamos realizar esta operação no **Subscriber**, de acordo com o código:

```
Observable.just("Hello World")
    .subscribe(value ->
        System.out.println(value + " " + new Date()));
```

Listagem 8. Encadeamento de Objetos.

```
Observable.just("Hello World")
    .subscribe(new Action1<String>() {
        @Override
        public void call(String value) {
            System.out.println(value);
        }
});
```

Esta solução funciona, mas é pouco satisfatória semanticamente dentro da programação reativa, pois o **Subscriber** deve ter o papel apenas de reagir a um fluxo, não de transformá-lo. Para o papel de transformação dos dados de um fluxo, utilizamos os operadores.

Para introduzir este conceito, vamos utilizar o operador **map()**, que é utilizado para transformar os dados de um fluxo no RxJava. Primeiramente, vamos conhecer a sua implementação sem lambdas, para um melhor entendimento da sua responsabilidade, conforme apresentado na **Listagem 9**.

Com a simplificação por lambdas, temos o resultado mostrado a seguir, que gera um código muito menor e mais limpo:

```
Observable.just("Hello World")
    .map(value -> value + " " + new Date())
    .subscribe(value -> System.out.println(value));
```

Uma das características mais interessantes do operador **map()** é que ele não precisa obrigatoriamente retornar um fluxo do mesmo tipo que está recebendo. Através de generics, ele especifica o tipo de dado recebido e o tipo de dado retornado. Poderíamos, neste caso, não retornar a **String** em si, mas seu tamanho (vide **Listagem 10**).

Ademais, como o método **map()** retorna outro **Observable**, é possível concatenar chamadas da forma como for necessária no código, como mostrado na **Listagem 11**.

Alterando um Observable

Com o RxJava, podemos aplicar algumas alterações sobre um **Observable**, modificando a forma como ele emite seu fluxo de conteúdo através de operadores. Deste modo, é possível customizar a frequência ou mesmo a quantidade de itens emitidos originalmente pelo **Observable**.

Listagem 9. Utilizando o operador map().

```
Observable.just("Hello World")
    .map(new Func1<String, String>() {
        @Override
        public String call(String value) {
            return value + " " + new Date();
        }
    })
    .subscribe(value -> System.out.println(value));
```

Listagem 10. Operador map() retornando um tipo diferente do recebido pelo Observable.

```
Observable.just("Hello World")
    .map(value -> value.length())
    .subscribe(value -> System.out.println(value));
```

Listagem 11. Operador map() com chamadas concatenadas.

```
Observable.just("Hello World")
    .map(value -> value.length())
    .map(intValue -> String.valueOf(intValue))
    .subscribe(value -> System.out.println(value));
```

A seguir, veremos os principais operadores do RxJava e suas aplicações.

repeat()

O operador **repeat()**, presente na classe **Observable**, permite, como o nome diz, repetir os itens emitidos por um Observable por uma quantidade de vezes determinada pelo número inteiro passado como parâmetro, conforme mostrado na **Figura 1** e nesse código:

```
Observable.just(1, 2, 3)
    .repeat(2)
    .subscribe(observer);
```

Este **Observable** gerará a sequência 1, 2 e 3 duas vezes, notificando o **Observer** através do método **onNext()**. Após a emissão dos seis elementos, o método **onCompleted()** do **Observer** será chamado, encerrando o fluxo de dados.

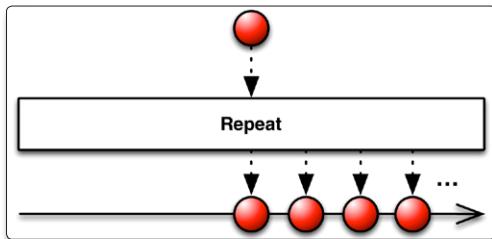


Figura 1. Fluxo no método repeat()

Listagem 12. Observable emissor de String.

```
public Observable<String> getObservableString() {
    return Observable.create(subscriber -> {
        if (subscriber.isUnsubscribed()) return;
        System.out.println("Operação realizada");
        subscriber.onNext("Mensagem");
        subscriber.onCompleted();
    });
}
```

defer()

O operador **defer()** tem como funcionalidade fazer com que um **Observable** só comece a emitir itens quando um objeto se inscreve para receber seu fluxo de dados. Como exemplo, vamos analisar o **Observable** apresentado na **Listagem 12**.

Apesar deste **Observable** ser bastante simples, ele servirá para ilustrar o uso do **defer()**, como apresentado no código a seguir, que mostra o **Observable** criado, mas como ainda não foi inscrito por nenhum **Observer**, nenhuma mensagem será exibida no console.

```
Observable<String> deferredObservable = Observable.defer(this::getObservableString);
```

Somente após a inscrição no **Observable** é que teremos tanto a mensagem “Operação realizada” quanto a mensagem “Mensagem” mostradas no console de nossa aplicação, conforme o código:

```
deferredObservable.subscribe(value -> System.out.println(value));
```

Na **Figura 2** é ilustrada a estrutura do operador **defer()**.

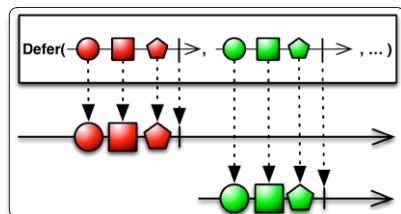


Figura 2. Estrutura do operador defer()

range()

O método **range()** nos permite criar um Observable que emite elementos dentro de uma faixa de valores definida pelos parâmetros recebidos em sua chamada. Estes valores são emitidos um a um até que a quantidade de repetições definida seja satisfeita. Vejamos um exemplo:

```
Observable.range(100, 5).subscribe(value -> System.out.println(value));
```

O primeiro parâmetro indica o valor inicial a ser emitido e o segundo parâmetro, a quantidade de incrementos a serem executados (sempre somando-se 1 ao valor inicial), indicando quantos valores serão emitidos no fluxo do **Observable** após a emissão do valor inicial. A representação desse comportamento é demonstrada na **Figura 3**.

interval()

O método **interval()** é bastante útil para os casos onde é necessário realizar algum tipo de pooling em algum recurso. Com ele é possível configurar o intervalo entre as chamadas ao **onNext()**, bem como um delay inicial, caso desejado.

No exemplo de código a seguir, temos um caso onde o método **onNext()** é chamado a cada dois segundos, recebendo um valor **Long** que indica o número da execução:

Programação reativa com a biblioteca RxJava

```
Observable.interval(2, TimeUnit.SECONDS).subscribe(System.out::println);
```

Na **Figura 4** é mostrada a estrutura do método **interval()**.

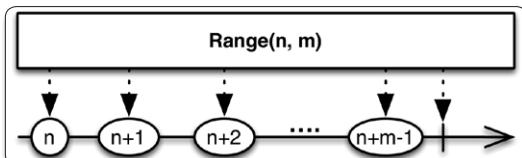


Figura 3. Estrutura do método range()

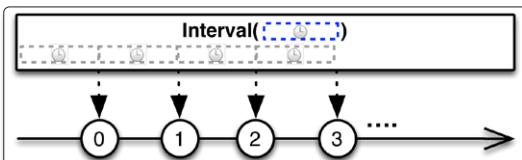


Figura 4. Estrutura do método interval()

timer()

Semelhante ao **interval()**, o método **timer()** define um delay para emitir um elemento e se encerra logo em seguida. Útil para casos onde uma ação deve ser agendada para ser executada após determinado tempo, porém executada apenas uma única vez. Após a emissão do elemento, o Observable é concluído. Veja o código da **Listagem 13**. Na **Figura 5** é mostrada a estrutura do método **timer()**.

Listagem 13. Exemplo com o método timer().

```
Observable.timer(2, TimeUnit.SECONDS)
    .subscribe(
        System.out::println, // onNext() -> irá imprimir 0
        System.out::println,
        () -> System.out.println("Completed") // onCompleted()
```

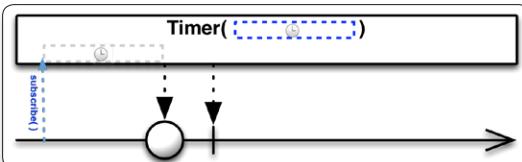


Figura 5. Estrutura do método timer()

Transformando fluxos de dados

Uma das grandes vantagens da utilização da programação reativa é a possibilidade de transformar os dados antes de serem entregues de fato ao componente receptor. Na seção anterior, conhecemos alguns operadores que alteram o fluxo de dados quanto a sua quantidade de elementos ou frequência, ou ainda condicionando em relação à quando serão entregues.

Para os próximos exemplos, vamos simular um cenário onde temos um método que realiza uma busca em um banco de dados e retorna uma série de URLs favoritadas. Este método poderia ter uma assinatura que retorna um **Observable** de uma lista de **Strings**, como demonstrado a seguir:

```
Observable<List<String>> getBookmarkedPages();
```

A partir disso, caso fossemos consumir este Observable, teríamos algo semelhante ao código apresentado na **Listagem 14**.

Esta solução, apesar de funcionar, não é satisfatória, já que exige que um looping seja escrito para que qualquer tipo de transformação seja realizado sobre os dados do **Observable**.

Uma maneira de realizar algum tratamento individualmente em cada um dos elementos da lista emitida pelo Observable do método **getBookmarkedPages()**, seria utilizando o método **from()**, que já foi visto anteriormente. Veja um exemplo na **Listagem 15**.

Porém, essa opção resultou em um código bastante confuso e difícil de entender. Para solucionar esse problema, podemos fazer uso do operador **flatMap()**, que permite criar um **Observable** a partir de outro **Observable**. Dessa forma, teríamos algo como o mostrado na **Listagem 16**.

O operador **flatMap()** recebe um objeto do tipo **Func1**, que através dos generics que especifica em sua declaração, define o tipo recebido e o retornado. O primeiro, indica que tipo de dados o **Observable** recebe (em nosso exemplo, uma lista de **String**), e o segundo, diz que tipo de dado será retornado após a transformação (no nosso caso, um **Observable** de **Strings**).

No método **call()** é onde realizaremos qualquer tipo de transformação que desejarmos. Na **Listagem 16** transformamos a lista em um novo **Observable** que retornará seus elementos um a um. Com a notação de lambdas, podemos deixar este código ainda mais limpo, como o trecho a seguir:

```
getBookmarkedPages()
    .flatMap(bookmarks -> Observable.from(bookmarks))
    .subscribe(bookmark -> System.out.println(bookmark));
```

E com **method reference**, podemos sintetizar ainda mais esta chamada, resultando em:

```
getFavoriteUrls()
    .flatMap(Observable::from)
    .subscribe(System.out::println);
```

Apesar de parecer um pouco complexo no início, este código simplesmente cria um **Observable** a partir do **Observable** retornado pelo método **getFavoriteUrls()**, e este **Observable** resultante é que será inscrito pelo método **subscribe()**. O funcionamento do operador **flatMap()** pode ser verificado na **Figura 6**.

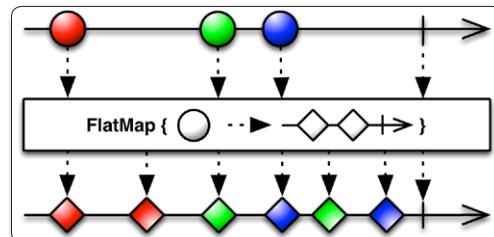


Figura 6. Estrutura do operador flatMap()

Listagem 14. Consumindo Observable de lista de String.

```
getBookmarkedPages()
.subscribe(bookmarks -> {
    for (String bookmark : bookmarks) {
        System.out.println(bookmark);
    }
});
```

Listagem 15. Utilizando método from() do Observable.

```
getBookmarkedPages()
    .subscribe(bookmarks -> {
        Observable.from(bookmarks)
            .subscribe(bookmark -> System.out.println(bookmark));
    });
}
```

Listagem 16. Utilizando operador flatMap().

```
getBookmarkedPages()
    .flatMap(new Func1<List<String>, Observable<String>>() {
        @Override
        public Observable<String> call(List<String> bookmarks) {
            return Observable.from(bookmarks);
        }
    })
    .subscribe(bookmark -> System.out.println(bookmark));
```

Com o encadeamento de chamadas, é possível criar praticamente qualquer grupo de dados transformados, inclusive filtrando valores desnecessários. Imaginemos o caso onde, por exemplo, a lista poderia conter valores nulos. Para filtrar este tipo de dados, poderíamos fazer uso do operador `filter()`. Com este operador conseguimos estabelecer uma determinada condição para que os valores que não a satisfaça não sejam emitidos pelo Observable. A **Figura 7** mostra a estrutura do operador `filter()`.

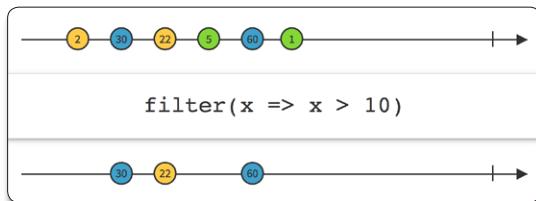


Figura 7. Estrutura do operador filter()

Caso desejemos filtrar os valores nulos, basta encadear o uso do `filter()` à chamada, como demonstra a [Listagem 17](#).

Note que o método recebe como parâmetro um objeto `Func1`, que recebe um item do fluxo emitido pelo `Observable` e um `Boolean` indicando se a condição foi satisfeita, fazendo com que o elemento seja devidamente filtrado. Já com o uso de lambdas, temos como resultado o código mostrado na [Listagem 18](#).

Conhecendo outros operadores

Após entender o papel de um operador no RxJava e conhecer os principais, vamos ver outros operadores não tão comuns, porém bastante úteis. Um deles é o operador **take()**, que cria um novo **Observable**, retornando em seu fluxo a quantidade de elementos definida como parâmetro. Uma ilustração do funcionamento desse operador é apresentada na **Figura 8**.

Listagem 17. Utilizando os operadores flatMap() e filter().

```
getFavoriteUrls()
    .flatMap(Observable::from)
    .filter(new Func1<String, Boolean>() {
        @Override
        public Boolean call(String s) {
            return s != null;
        }
    })
    .subscribe(System.out::println);
```

Listagem 18. Utilizando operadores flatMap() e filter() com lambdas.

```
getFavoriteUrls()  
    .flatMap(Observable::from)  
    .filter(s -> s != null)  
    .subscribe(System.out::println)
```

Na estrutura mostrada nesta imagem, assim que o segundo elemento é recebido, o Observable é completado. Em código, teríamos algo como o apresentado a seguir:

```
Observable.just(1, 2, 3, 4)  
.take(2)  
.subscribe(System.out::println);
```

De forma semelhante, o RxJava também nos fornece os operadores `takeFirst()` e `takeLast()`, que nos permitem obter os primeiros ou últimos n elementos de um Observable, retornando-os. No caso específico do `takeLast()`, é importante salientar que ele somente cria seu Observable quando o outro Observable conclui a sua emissão de elementos. A **Figura 9** ilustra esta situação.

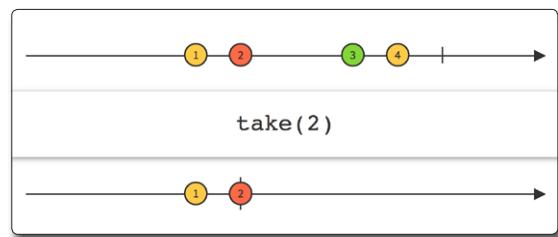


Figura 8. Estrutura do operador take()

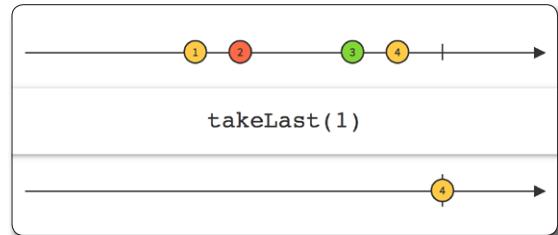


Figura 9. Estrutura do operador takeLast().

Para casos onde temos um fluxo de dados com muitos dados repetidos, como um sensor de temperatura ou um socket que envia dados continuamente, podemos fazer uso do operador `distinct()`, que filtra os elementos repetidos de um fluxo. Veja o código a seguir:

Programação reativa com a biblioteca RxJava

```
Observable.just(1, 2, 2, 1, 3)
    .distinct()
    .subscribe(System.out::println);
```

O resultado deste trecho é a impressão do valor 123.

Principalmente na programação embarcada, quando necessitamos de algum código que emita valores originados em um sensor, é comum filtrarmos os valores para que, por exemplo, o Observable somente notifique a interface quando um valor diferente do anterior for exibido. Para estes casos, o RxJava nos fornece o operador **distinctUntilChanged()**. Ele filtra elementos repetidos em sequências em um Observable.

Imagine o caso de um sensor de temperatura que emite a temperatura atual. Só necessitamos atualizar a informação em um visor ou na tela quando há alguma mudança. O comportamento deste operador pode ser visto na **Figura 10**.

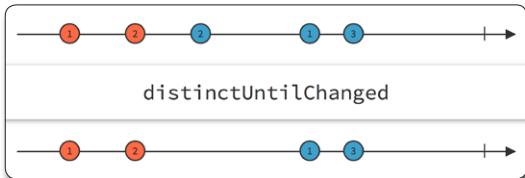


Figura 10. Estrutura do operador distinctUntilChanged()

Em código, poderíamos implementar este operador como mostrado a seguir:

```
Observable.just(1, 1, 2, 2, 1, 3)
    .distinctUntilChanged()
    .subscribe(System.out::println);
```

Este código imprime no console a saída 1213.

Outros operadores que funcionam de forma semelhante, filtrando porções dos itens de um Observable, são o **first()** e **last()**, que não recebem argumentos e retornam o primeiro e o último elementos de um fluxo, respectivamente.

Neste mesmo leque de operadores que filtram porções lineares de um fluxo, temos o **skip()**, o qual recebe um parâmetro inteiro que indica quantos itens devem ser ignorados em um determinado Observable, e então, começa a emitir itens no fluxo do Observable resultante. E temos também o **skipLast()**, que tem comportamento semelhante ao **skip()**, porém, ignorando os últimos n elementos de um Observable, como mostrado na **Figura 11**.

Para casos onde temos um fluxo de dados contínuo com pouca variação de valores, ou mesmo casos onde uma atualização em tempo real poderia causar um overhead muito grande no sistema, o RxJava nos fornece o operador **sample()**, que cria um Observable que emite o último valor originado no fluxo do Observable original após uma faixa de tempo definida.

Para analisar essa situação, imaginemos um Observable que retorne números de forma aleatória, conforme a **Listagem 19**.

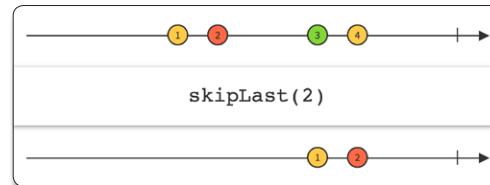


Figura 11. Estrutura do operador skipLast()

Listagem 19. Observable que retorna números aleatórios.

```
public Observable<Long> randomObservable() {
    return Observable.create(subscriber -> {
        Random random = new Random(System.currentTimeMillis());
        while (true) {
            subscriber.onNext(random.nextLong());
        }
    });
}
```

Neste cenário, podemos, através do operador **sample()**, transformar este Observable em um outro, que retorne um valor aleatório a cada 10 segundos:

```
randomObservable()
    .sample(10, TimeUnit.SECONDS)
    .subscribe(System.out::println);
```

Outro operador relacionado ao **sample()** é o **debounce()**. Este operador geralmente é utilizado para filtrar itens emitidos em um curto período de tempo (timer), evitando que o fluxo de dados final contenha elementos indesejados. Com o uso deste operador, caso um item seja emitido antes que o tempo de timeout seja atingido, a emissão do elemento mais antigo é cancelada e o timer reiniciado. Somente quando o tempo total do timer é decorrido e nenhuma outra emissão ocorreu é que o item será publicado no fluxo de dados filtrado resultante do **debounce()**.

Exemplos clássicos de uso do operador **debounce()** são os casos onde temos uma caixa de busca e precisamos buscar sugestões relacionadas. Em vez de realizarmos uma requisição a cada letra inserida (que muitas vezes é custosa, para algum serviço, ou mesmo uma consulta a uma base de dados), estipulamos uma fração de tempo que nos indica que o usuário terminou de inserir a palavra que deseja pesquisar.

Um exemplo de código que demonstra esse comportamento pode ser visto a seguir:

```
getInputTextObservable()
    .debounce(400, TimeUnit.MILLISECONDS)
    .subscribe(s -> performRequest(s));
```

Como podemos verificar, o RxJava possui uma infinidade de operadores, para os mais diversos propósitos. Nesta seção foi possível conhecer alguns deles e suas respectivas utilizações. A lista completa de todos os operadores disponíveis, seguidos de descrições e diagramas ilustrativos, é encontrada na wiki do projeto no GitHub (veja a seção **Links**).

Combinando Observables

Ao construir aplicações assíncronas, podem ocorrer situações em que temos mais de uma fonte de dados, porém um ponto em comum onde esses dados são tratados de forma conjunta, ou seja, múltiplas origens, mas apenas um fim. Para situações como esta, o RxJava nos fornece uma série de operadores que se encarregam da função de unir Observables para nós.

Uma das aplicações mais comuns da junção de Observables acontece em sistemas nos quais determinada ação envolve, por exemplo, duas requisições a uma API REST, onde pode ser necessário realizar um login e outra requisição para obter os dados de determinado usuário, ou mesmo suas configurações de ambiente.

merge()

O operador **merge()**, como o próprio nome indica, transforma dois ou mais Observables em um só, reunindo os elementos emitidos pelos Observables em um único fluxo resultante, como mostrado no código a seguir e na **Figura 12**:

```
Observable.merge(meuObservable, outroObservable)
    .subscribe(onNextAction);
```

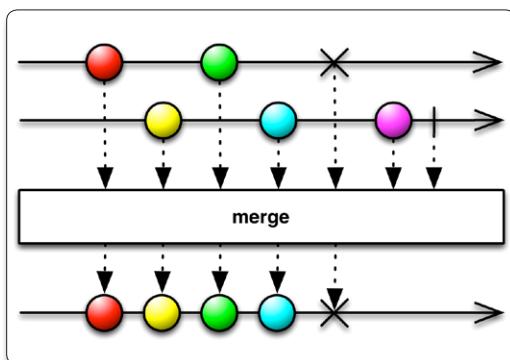


Figura 12. Estrutura do operador merge()

Outra forma de se unir dois Observables é através do método **mergeWith()**, que permite encadear o processo a partir de um Observable já existente. Verifique este exemplo:

```
meuObservable.mergeWith(outroObservable)
    .subscribe(onNextAction);
```

Se no processo de emissão de itens no fluxo algum dos dois Observables lançar uma exceção, o Observable resultante é interrompido e o método **onError()** é chamado. Caso você deseje que os outros Observables continuem enviando itens e o erro seja reportado apenas quando todos os Observables concluírem sua execução, você pode fazer uso do operador **mergeDelayError()**, cuja estrutura é mostrada na **Figura 13**.

zip()

O operador **zip()** funciona de forma semelhante ao **merge()**, porém emitindo os itens de forma agrupada. Deste modo, em um

caso onde temos dois Observables, ele aguarda que um item seja emitido nos dois fluxos para somente então gerar um objeto no fluxo do Observable resultante (**Figura 14**).

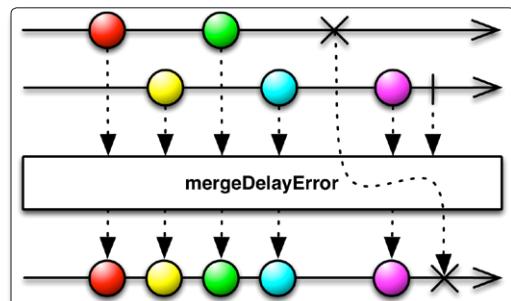


Figura 13. Estrutura do operador mergeDelayError()

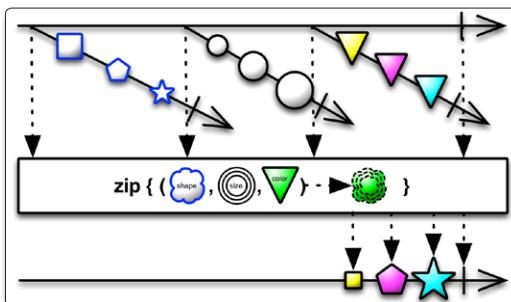


Figura 14. Estrutura do operador zip()

Nota

Neste caso, vale salientar que o tipo de dado fornecido pelos Observables deve ser o mesmo.

Em sua implementação, o operador **zip()** recebe também um objeto **Func2**, que deve ensinar o Observable a agrupar os resultados. No exemplo dado na **Listagem 20**, temos **meuObservable**, que retorna Strings em seu fluxo, e **outroObservable**, que retorna valores inteiros. O Observable resultante é definido para retornar Strings, através do terceiro generic passado para o objeto **Func2**.

Listagem 20. Utilizando o operador zip().

```
Observable.zip(meuObservable, outroObservable,
    new Func2<String, Integer, String>() {
    @Override
    public String call(String string, Integer integer) {
        return string + integer;
    }
}).subscribe();
Com o uso de lambdas, esse código pode ser simplificado a:
Observable.zip(meuObservable, outroObservable,
    (string, integer) -> string + integer)
    .subscribe();
```

combineLatest()

Com o operador **combineLatest()**, a cada valor emitido por um dos Observables envolvidos, é gerado um valor no fluxo resultante com o último valor emitido por cada um dos Observables.

Na **Figura 15** podemos verificar o seu funcionamento.

Neste caso, o Observable resultante só começa a emitir objetos em seu fluxo quando todos os Observables envolvidos já emitiram pelo menos um item. Em sua definição, ele também faz uso de um objeto **Func2** para definir como os objetos resultantes serão construídos.

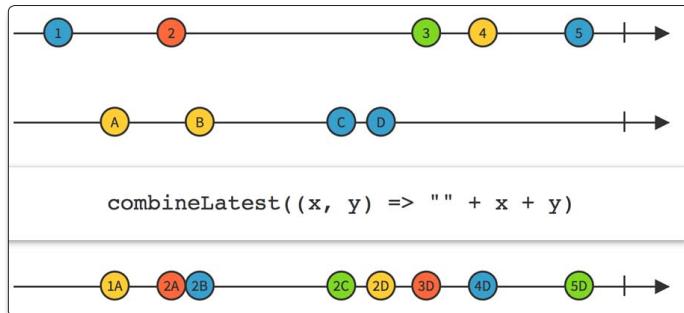


Figura 15. Estrutura do operador `combineLatest()`.

Schedulers

Como citado anteriormente, uma das grandes vantagens da programação reativa é a sua facilidade de trabalhar de forma assíncrona. Essa forma assíncrona de executar a lógica é controlada no RxJava através de um mecanismo chamado Scheduler, mecanismo esse que nos fornece uma maneira simples de criar rotinas concorrentes sem termos que nos preocupar com implementações, sincronização, threads, limitações de plataforma e várias complexidades geralmente relacionadas ao paralelismo de execução.

O RxJava oferece em sua API cinco tipos de Schedulers, analisados a seguir:

1. **Schedulers.io()** é utilizado para operações de I/O, como consultas em disco ou mesmo requisições que envolvem a rede. Ele é implementado em cima de um pool de threads com tamanho dinamicamente ajustado dependendo da necessidade da aplicação;
2. **Schedulers.computation()** é o Scheduler padrão nos Observables que não executam operações de I/O. Ele é automaticamente utilizado em operadores como **buffer()**, **debounce()**, **delay()**, **interval()**, **sample()** e **skip()**;
3. **Schedulers.immediate()** permite iniciar o processamento de forma imediata na thread atual. É utilizado por padrão nos operadores **timeout()**, **timeInterval()** e **timestamp()**;
4. **Schedulers.newThread()**, como o próprio nome indica, realiza a execução iniciando uma nova thread incondicionalmente;
5. **Schedulers.trampoline()**, por sua vez, enfileira a execução para ser feita assim que possível na thread atual. É utilizado em operadores como **retry()** e **repeat()**.

Para configurarmos a thread de execução de um Observable, fazemos uso de seu método **subscribeOn()**, passando como parâmetro o Subscriber desejado.

Temos também o método **observeOn()**. Este método indica a thread para a qual o resultado de um Observable será entregue. Para ambientes onde temos problemas de manipulação de resul-

tados em outras threads (como o Android), este método se torna um grande facilitador.

Inclusive, falando um pouco de Android, temos na plataforma a restrição de que somente a thread de interface pode manipular os itens da tela. Para contornar essa questão, temos uma extensão da RxJava, chamada RxAndroid (atualmente na versão 1.0.1), que nos fornece uma classe chamada **AndroidSchedulers**. Esta classe, convenientemente, possui um método estático chamado **mainThread()**, que permite que os resultados do processamento de um Observable, ou mesmo a emissão de itens assíncronos de um fluxo, ocorra na thread correta.

Assim, uma chamada encadeada nesse caso resultaria em algo como o código mostrado na **Listagem 21**.

Controle de erros

Para entendermos melhor como funciona o tratamento de erros no RxJava, vamos tomar o trecho de código da **Listagem 22** como exemplo, onde os métodos chamados pelo operador **map()** podem lançar exceções.

Listagem 21. Observable com chamada encadeada de métodos.

```
observableDeProcessamentoPesado()
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(atualizarTela(value));
```

Listagem 22. Observable com alguns métodos.

```
Observable.just(1, 2, 3, 4)
    .map(value -> metodoPerigoso(value))
    .map(value -> metodoMaisPerigosoAinda(value))
    .subscribe(new Subscriber<String>() {
        @Override
        public void onNext(String s) {
            System.out.println(s);
        }

        @Override
        public void onComplete() {
            System.out.println("Completed!");
        }

        @Override
        public void onError(Throwable e) {
            System.err.println("Ouch!");
        }
   });
```

Neste caso, temos dois métodos, **metodoPerigoso()** e **metodoMaisPerigosoAinda()**, que podem lançar uma exceção. No RxJava, caso uma exceção ocorra em qualquer ponto da execução de um Observable, toda a execução deste Observable é interrompida, e o método **onError()** do Subscriber ou Observer é chamado. Na **Listagem 22** se uma exceção for lançada pelo **metodoPerigoso()**, o **metodoMaisPerigosoAinda()** não será executado. Também é possível perceber que o método **onError()** é chamado em caso de exceções em qualquer ponto. Esta é uma grande vantagem para momentos em que os processamentos somente fazem sentido

juntos, e devem ser executados em sua totalidade. Outra vantagem é o tratamento de erros centralizado, que permite um código mais sucinto e organizado.

Subscriptions

Um último ponto que deve ser mencionado é o objeto **Subscription**. Caso você tenha explorado um pouco mais os Observables, deve ter notado que toda vez que um Observable é inscrito por um Subscriber ou Observer, é retornado um objeto do tipo **Subscription**.

O objeto **Subscription** representa o elo de ligação entre o nosso Observable e um Subscriber, conforme mostra o código a seguir:

```
Subscription subscription = Observable.just("Hello World")
    .subscribe(value -> System.out.println(value));
```

Através desse objeto, é possível encerrar a execução de um Observable imediatamente, realizando-se uma chamada ao método **unsubscribe()**.

O conceito de programação reativa já existe há bastante tempo, porém só acabou sendo evidenciado no mundo Java com o Java 8. O próprio uso de streams em coleções já fornece ares de programação funcional e reativa ao Java, deixando a sintaxe mais limpa com o uso dos lambdas.

Nesse contexto o RxJava vem se tornando cada vez mais uma saída viável para adotar este paradigma nas aplicações, sejam elas corporativas ou até mobile (dentro do Android) e sem a limitação de versão, podendo funcionar em qualquer versão do Java a partir do JDK 5.

Soluções assíncronas e real time são apenas algumas das possíveis aplicações da RxJava, onde paralelismo e concorrência são desafios reais a serem superados.

Com a comunidade ReactiveX crescendo a cada dia com novos ports para outras plataformas, esta é a hora certa de começar a experimentar!

Nota

Ainda falando de Android, vale aqui um adendo sobre a popularidade que o RxJava vem tendo dentro da comunidade. Ele vem sendo utilizado para adicionar traços de programação reativa desde binding de views até requisições de rede. Inclusive a biblioteca Retrofit, desenvolvida pela Square (uma das bibliotecas mais utilizadas para o consumo de serviços REST no Android), possui integração nativa com RxJava, transformando automaticamente o resultado das requests em Observables. Ademais, outros projetos como RxLifecycle (ciclo de vida reativo de Activities) e RxBinding (binding reativo de views) vêm ganhando cada vez mais relevância dentro da comunidade Android.

Autor



Rafael Toledo

rafaeltoledo@gmail.com – <http://www.rafaeltoledo.net>

Desenvolvedor Android na Concrete Solutions, tem experiência em mais de 15 projetos no Brasil e no exterior. Possui também experiência em desenvolvimento Java desde 2009. É formado em Sistemas de Informação e pós-graduado em Desenvolvimento Ágil para Web e Mobile. Escreve esporadicamente em seu blog pessoal que possui mais de 15.000 visitas mensais.



Links:

The Netflix Tech Blog.

<http://techblog.netflix.com/2013/02/rxjava-netflix-api.html>

Reactive Extensions.

<https://reactivex.io>

Retrolambda.

<https://github.com/orfjackal/retrolambda>

RxJava Operators.

<http://reactivex.io/documentation/operators.html>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Money API: Conheça um dos novos recursos do Java 9

Aprenda nesse artigo o que vem de novo na Money API e como ela facilitará a manipulação de valores monetários em suas aplicações

Você já precisou criar uma aplicação na qual existe a necessidade de manipulação de valores monetários? Dois bons exemplos de aplicações deste tipo são os e-commerce e sistemas de frente de loja, nos quais precisamos constantemente somar, subtrair, dividir e multiplicar valores em dinheiro, sem contar a necessidade de converter valores de real para outra moeda. Para dificultar um pouco mais, precisamos também imprimir esses valores de forma amigável para o usuário.

Caso ainda não o tenha feito, pergunte para quem já precisou desenvolver estes tipos de sistemas e a resposta provavelmente será que o Java deveria fornecer uma forma mais simples e elegante de tratar valores monetários. Embora aparente ser um problema simples, existem diversos aspectos com os quais devemos nos preocupar e que nos sobrecarregam.

Atualmente, a prática mais comum entre os desenvolvedores consiste em criar um campo do tipo **Double**, **Float** ou **BigDecimal** para armazenar estes valores. Contudo, como veremos no decorrer deste artigo, estas não são práticas recomendáveis, assim como também não é aconselhável utilizar os tipos **int** e **long** para esta tarefa. Contudo, todos estes problemas estão prestes a desaparecer com o lançamento da Money API, que virá junto com a nona versão do Java, com lançamento previsto para o ano de 2016.

A Money API é representada pela JSR 354 e vem para simplificar esta árdua tarefa. Deste modo, dedicaremos todo este artigo para analisar as motivações que levaram à criação desta nova JSR, assim como a instalação da Mo-

Fique por dentro

Criar sistemas que manipulam valores monetários na plataforma Java sempre foi motivo de dor de cabeça para os desenvolvedores. Quem já precisou criar uma aplicação de e-commerce, por exemplo, já notou que as técnicas disponíveis no momento sempre nos levam a problemas difíceis de serem superados ou a um grande trabalho manual de codificação de classes próprias que visam manipular esses valores de forma satisfatória. É neste cenário que surge a Money API, solução que visa facilitar enormemente esta tarefa, trazendo uma API simples e intuitiva de ser usada. Embora seu lançamento esteja previsto apenas na nona versão do Java, desde já podemos usá-la e nos antecipar a estas mudanças, pois a JSR e sua implementação de referência já estão disponíveis. Após ler este artigo e entender como esta API vai facilitar substancialmente sua vida, não temos dúvidas que você irá baixá-la agora mesmo e começar a utilizá-la em seu próximo projeto!

neta (implementação de referência desta JSR) e diversos exemplos de código demonstrando como usar esta nova API desde já, sem a necessidade de aguardar o lançamento do Java 9.

A vida antes da Money API

Antes de começarmos a analisar esta nova API, vamos relembrar como manipulamos valores monetários nas atuais versões do Java. Para isso, suponha que estamos projetando uma aplicação de frente de loja, e por ser comum a soluções desse tipo, precisaremos de uma classe chamada **Produto** que, além de outros atributos, necessita de um campo para armazenar o preço.

Ao codificar esse atributo, logo pensamos em especificá-lo como sendo do tipo **Double** ou **Float**. Mas, será esta uma boa prática? Em seu livro Money API (ver seção **Links**), Otávio Santana, um dos responsáveis por essa JSR, nos mostra que esta não é uma prática recomendável quando precisamos de valores precisos. Este argumento também é sustentado por Joshua Bosch, em seu livro *Effective Java*: os tipos **Double** e **Float** foram criados para cálculos científicos e de engenharia. Quando os usamos para representar moeda, surgem alguns problemas difíceis de serem superados.

Por exemplo, suponha que o seu produto custe R\$ 1,03, mas você resolveu dar um desconto de R\$ 0,42 para um cliente. Qual valor você espera que esse atributo tenha após realizada esta subtração? Exatos 0.61, correto? Mas, caso você resolva imprimir este atributo no console, verá que o valor impresso será 0.6100000000000001, conforme demonstra o código da **Listagem 1**. Trata-se de uma aproximação e não o valor exato que queríamos.

Você pode argumentar que basta arredondar este valor e estará tudo resolvido! Mas, isto nem sempre irá funcionar, conforme o código da **Listagem 2** nos mostra. Observe nesta listagem que usamos apenas valores simples, com apenas uma casa decimal, para fazer os cálculos, e mesmo assim o Java imprime no console o valor 0.3999999999999997, que obviamente não era o que esperávamos. O correto seria termos impresso o valor 0.40.

Ainda assim, você pode arredondar este valor manualmente. Contudo, observe que usar esta estratégia não é tão interessante, pois você terá que arredondar os resultados de todas as operações que realizar, o que é muito ineficiente e sujeito a erros, uma vez que o processo de arredondamento pode ser diferente em cada moeda.

Listagem 1. Primeiro problema ao usar Double ou Float para valores monetários.

```
package br.com.devmedia;

public class DoubleFloat {

    public static void main(String...args) {
        System.out.println(1.03 - 0.42); // Imprime 0.6100000000000001
    }
}
```

Listagem 2. Arredondar ou usar valores exatos também não funciona.

```
package br.com.devmedia;

public class ArredondarNaoFunciona {

    public static void main(String...args) {
        System.out.println(1.0 - 0.1 - 0.2 - 0.3); // 0.3999999999999997
    }
}
```

Muro das lamentações

Então, como podemos resolver esses problemas com os tipos **Double** e **Float** com as versões atuais do Java? A primeira resposta possível está no tipo **BigDecimal**, já que com ele não temos

mais os problemas de arredondamento nos cálculos aritméticos, conforme vimos na seção anterior. Contudo, caímos em outra situação um pouco desagradável, pois por se tratar de um objeto e não um tipo primitivo, você não poderá realizar operações usando os operadores +, -, * e etc. Pior, este objeto não possui sequer métodos para facilitar estes cálculos, como **add()**, **multiply()**, **subtract()** ou **divide()**.

Como última cartada, podemos tentar usar os tipos **int** e **long**, tratando os valores monetários sempre como centavos. Desta forma, no lugar de guardarmos R\$ 1,00 no atributo, guardaríamos 100 centavos. Entretanto, embora pareça uma boa solução, ainda não é 100% efetiva, uma vez que ainda temos o problema da representação desse valor com casas decimais, que terá que ser tratado de forma manual.

Outro problema que enfrentamos ao usar **Double**, **Float**, **int**, **long** ou **BigDecimal** trata-se da questão da representação da moeda, ou seja, se o valor está em Reais, Dólares, Libras, etc. Por exemplo, não faz sentido somar R\$ 2,00 (dois reais) a \$2.00 (dois dólares), pois são duas unidades monetárias diferentes.

Além disso, temos que imprimir esses valores para a visualização do usuário do seu sistema, que também é de extrema importância. Caso seu sistema seja feito para funcionar apenas com uma moeda, você talvez não encontre muitas barreiras. No entanto, imagine um sistema feito para funcionar na Web e receber pagamentos oriundos de todos os cantos do mundo! Para um chinês, precisamos exibir o valor formatado conforme as regras da sua moeda local. Vale ressaltar ainda que existe a questão da conversão entre moedas, que você precisará realizar por conta própria.

Joda-Money

Cabe destacar rapidamente a biblioteca Joda Money. Esta é uma biblioteca escrita pelo mesmo pessoal que criou o tão famoso Joda Time, que visa facilitar o uso e tratamento de datas em Java. Caso você já tenha utilizado a Joda Money, notará que a Money API tem muitas similaridades e isto não é mera coincidência, pois o criador dessa biblioteca também trabalhou nesta JSR como expert no assunto. Portanto, caso você já domine essa biblioteca, verá que a transição para a Money API será bastante simples.

Moneta

Antes de iniciarmos com os exemplos práticos, vamos criar um projeto e prepará-lo para usar a Moneta, implementação de referência da Money API. Em primeiro lugar, usando a IDE de sua preferência, crie um projeto Java com suporte para o Maven. Em seguida, adicione as linhas a seguir no seu arquivo *pom.xml*, dentro da seção *dependencies*:

```
<dependency>
<groupId>org.javamoney</groupId>
<artifactId>moneta</artifactId>
<version>1.0</version>
</dependency>
```

Por último, não se esqueça de pedir para o Maven baixar as dependências. Pronto!

Unidades Monetárias

O primeiro passo ao utilizarmos a Money API é obter uma instância de uma classe que representa uma moeda em especial, como o Real ou Dólar. Na Money API, uma moeda é representada através de uma instância da classe `javax.money.CurrencyUnit`, sendo que podemos obter instâncias desta classe de duas formas, conforme apresentado na **Listagem 3**.

No primeiro exemplo, invocamos o método estático `getCurrency()` da classe `javax.money.Monetary`, passando como parâmetro uma `String` contendo os três caracteres que representam uma moeda. Assim, caso queiramos uma instância que representa o Real brasileiro, podemos passar o texto “BRL” ou, caso queiramos o Dólar americano, passamos o texto “USD”.

A segunda forma de obter uma instância desta classe consiste em passar uma instância de `Locale` para este mesmo método. Observe que esta opção é de extrema utilidade quando estamos desenvolvendo sistemas Web ou que devem funcionar em máquinas de pessoas do mundo inteiro, uma vez que a Money API lhe dará uma instância de `CurrencyUnit` apropriada para uso na máquina do usuário.

Listagem 3. Obtendo uma unidade monetária.

```
package br.com.devmedia;

import javax.money.CurrencyUnit;
import javax.money.Monetary;
import java.util.Locale;

public class Money {
    public static void main(String[] args) {
        CurrencyUnit currencyUnit = Monetary.getCurrency("BRL");
        System.out.println(currencyUnit);

        CurrencyUnit currencyUnitLocale = Monetary.getCurrency
            (Locale.getDefault());
        System.out.println(currencyUnitLocale);
    }
}
```

Valores monetários

Com relação ao exemplo da **Listagem 3**, note que a classe `CurrencyUnit` trata uma unidade monetária, mas não trata valores monetários. E qual a diferença entre ambos? Por exemplo, o Real brasileiro é uma unidade monetária, enquanto o valor 10 reais e 20 centavos é um valor monetário representado na unidade monetária chamada Real brasileiro.

Para tratarmos estes valores na Money API, precisamos de uma instância da interface `javax.money.MonetaryAmount`, que é implementada por três classes na Moneta:

1. org.javamoney.moneta.Money: Esta é a implementação padrão da interface `MonetaryAmount` dentro da Moneta. Internamente, ela representa o valor monetário como um `BigDecimal`;

2. org.javamoney.moneta.RoundedMoney: De forma similar à classe `Money`, a `RoundedMoney` também representa o valor monetário usando um `BigDecimal`. Contudo, ela também trabalha com uma instância de `MonetaryOperator`, que é uma operação a ser executada após cada operação realizada no valor monetário.

Por exemplo, podemos passar um `MonetaryOperator` que arredonda o valor após alguma operação aritmética ser realizada;

3. org.javamoney.moneta.FastMoney: Esta classe representa o valor monetário como um tipo primitivo `long` e, por este motivo, é a que possui melhor desempenho para realizar operações. Conforme descrito por Otávio Santana, em seu livro, ela chega a ser quinze vezes mais rápida que as outras duas implementações. Porém, esta classe não é indicada para representar valores que contenham mais de cinco casas decimais, já que sua precisão é bastante limitada.

Todas as três classes mencionadas possuem os mesmos métodos estáticos que nos permitem criar instâncias delas próprias. Na **Listagem 4**, por exemplo, usamos o método `of()`, da classe `Money`, para este propósito. Este método recebe dois parâmetros: o valor que desejamos e a unidade monetária em que este valor será representado. Caso queiramos evitar a criação de uma instância de `CurrencyUnit`, temos a opção de usar o mesmo método `of()`, mas passando uma `String` contendo a unidade monetária desejada como segundo parâmetro, conforme a **Listagem 5**.

Listagem 4. Obtendo uma unidade monetária usando uma instância de `CurrencyUnit`.

```
package br.com.devmedia;

import org.javamoney.moneta.Money;

import javax.money.CurrencyUnit;
import javax.money.Monetary;
import javax.money.MonetaryAmount;

public class MoneyExemplo {

    public static void main(String[] args) {
        CurrencyUnit currencyUnit = Monetary.getCurrency("BRL");
        MonetaryAmount amount = Money.of(10, currencyUnit);
        System.out.println(amount);
    }
}
```

Listagem 5. Obtendo uma unidade monetária sem `CurrencyUnit`.

```
package br.com.devmedia;

import org.javamoney.moneta.Money;
import javax.money.MonetaryAmount;

public class MoneyExemplo {

    public static void main(String[] args) {
        MonetaryAmount amount = Money.of(10, "BRL");
        System.out.println(amount);
    }
}
```

Além destas opções, existem mais dois métodos estáticos nestas classes que nos permitem criar instâncias de **MonetaryAmount**:

1. O método **zero()**, que recebe como parâmetro uma instância de **CurrencyUnit** e retorna um **MonetaryAmount** que corresponde ao valor zero na unidade informada;
2. O método **ofMinor()**, que recebe um número e a unidade monetária, retornando uma instância de **MonetaryAmount** que representa o valor informado em centavos, seguindo as regras estabelecidas pela unidade monetária escolhida.

Quanto à classe **RoundedMoney**, você notará que ela possui os mesmos métodos de **Money** e **FastMoney**, entretanto, com assinaturas um pouco diferentes, já que temos um parâmetro a mais: uma instância de **MonetaryOperator**! Nas próximas seções analisaremos o uso destas duas classes.

Operações aritméticas

Uma instância de **MonetaryAmount** seria de muito pouca utilidade, caso não pudéssemos realizar operações aritméticas com ela, concorda? E, obviamente, o time da Money API não deixou os detalhes para segundo plano e implementou diversos métodos que nos permitem adicionar, subtrair, multiplicar e dividir valores monetários. Esses métodos, entretanto, nunca alteram diretamente o objeto em si, pois eles são objetos imutáveis. Em vez de alterar a própria instância, eles retornam um novo objeto representando o resultado da operação realizada.

Analizando a **Listagem 6** temos quatro exemplos de operações aritméticas em sequência. Inicialmente, criamos duas instâncias de **Money** usando o método **of()**, conforme visto na seção anterior. A primeira instância representa o valor R\$ 5,00 e está armazenada na variável chamada **cincoReais**, enquanto a segunda instância representa o valor R\$ 15,00 e está armazenada na variável **quinzeReais**.

Em seguida, somamos esses dois valores usando o método **add()** do objeto **quinzeReais** e guardamos o resultado na variável **vinteReais**. Feito isso, usamos o método **subtract()** do objeto **quinzeReais** para realizar uma operação de subtração. Finalizando, temos um exemplo de como multiplicar utilizando o método **multiply()** e dividir, usando o método **divide()**. Note que ambos recebem um valor, representado pela classe **Number**, que será usado para multiplicar e dividir, respectivamente.

Além das operações mais comuns, vistas anteriormente, também temos duas operações de sinais e uma operação de resto. As duas operações de sinais são representadas pelos métodos **negate()** e **plus()**, enquanto a operação de resto é representada pelo método **remainder()**. Através da **Listagem 7** podemos ver alguns exemplos de uso destes métodos. Observe que primeiro criamos um valor monetário que representa 10 reais e atribuímos à variável **dezReais**. Logo após, chamamos o método **negate()** e imprimimos no console o valor obtido, que será exibido como BRL -10.

Prosseguindo, chamamos o método **negate()** e o método **plus()** em sequência e obtemos o valor BRL -10 impresso de novo. Esse valor parece estranho para você? Ele não deveria imprimir BRL 10?

Não, não deveria, pois aplicar o sinal de positivo em um valor negativo sempre dá um valor negativo. Caso você queira voltar para o valor positivo, deve chamar duas vezes seguidas o método **negate()**, conforme fizemos logo em seguida nesta mesma listagem.

Listagem 6. Operações aritméticas de soma, subtração, multiplicação e divisão.

```
package br.com.devmedia;

import org.javamoney.moneta.Money;

import javax.money.CurrencyUnit;
import javax.money.Monetary;
import javax.money.MonetaryAmount;

public class Aritmetica {

    public static void main(String...args) {
        CurrencyUnit currency = Monetary.getCurrency("BRL");

        MonetaryAmount quinzeReais = Money.of(15, currency);
        MonetaryAmount cincoReais = Money.of(5, currency);

        MonetaryAmount vinteReais = quinzeReais.add(cincoReais);
        MonetaryAmount dezReais = quinzeReais.subtract(cincoReais);
        MonetaryAmount vinteECincoReais = cincoReais.multiply(5);
        MonetaryAmount dozeReaisECinquenta = vinteECincoReais.divide(2);
    }
}
```

Listagem 7. Operações aritméticas de sinais e resto.

```
package br.com.devmedia;

import org.javamoney.moneta.Money;

import javax.money.CurrencyUnit;
import javax.money.Monetary;
import javax.money.MonetaryAmount;

public class Aritmetica {

    public static void main(String...args) {
        CurrencyUnit currency = Monetary.getCurrency("BRL");

        MonetaryAmount dezReais = Money.of(10, currency);
        System.out.println(dezReais.negate()); // Imprime BRL -10
        System.out.println(dezReais.plus()); // Imprime BRL -10
        System.out.println(dezReais.negate().negate()); // Imprime BRL 10
    }
}
```

Concluindo esta seção temos a **Listagem 8**, onde usamos o método **remainder()** para obter o resto da divisão do valor dez reais com quatro.

Operações monetárias

Na seção anterior vimos como realizar as principais operações aritméticas em objetos do tipo **MonetaryAmount**, contudo, não vimos como realizar outras operações monetárias, tais como: quanto é dez por cento de cem reais? Quantos centavos há no valor R\$ 2,35? O time da Money API pensou também nesse detalhe e

Money API: Conheça um dos novos recursos do Java 9

resolveu este problema criando a interface **MonetaryOperator**, que contém apenas um método, chamada **apply()**. Este método recebe apenas um parâmetro, do tipo **MonetaryAmount**, e retorna um valor do mesmo tipo, que representa o valor após a operação.

Listagem 8. Uso do método remainder().

```
package br.com.devmedia;

import org.javamoney.moneta.Money;

import javax.money.CurrencyUnit;
import javax.money.Monetary;
import javax.money.MonetaryAmount;

public class Aritmetica {

    public static void main(String...args) {
        CurrencyUnit currency = Monetary.getCurrency("BRL");

        MonetaryAmount dezReais = Money.of(10, currency);
        System.out.println(dezReais.remainder(4));
    }
}
```

Entretanto, você não precisa se preocupar em implementar esta interface caso precise de algumas operações comuns, como obter a parte decimal ou o percentual de um valor, pois a Moneta já nos provê algumas implementações padrão, que se encontram na classe **org.javamoney.moneta.function.MonetaryUtils**.

Vamos começar analisando, inicialmente, como obter 10% de um valor monetário qualquer. Para isso, observe a **Listagem 9**, onde criamos uma instância de **FastMoney** e, logo em seguida, chamamos o método **with()**, passando como parâmetro uma instância de **MonetaryOperator** obtida através da chamada ao método estático **percent()** da classe **MonetaryUtil**, que recebe como parâmetro um número representando a porcentagem. Neste exemplo, será impresso no console o valor BRL 1.00000, que corresponde a 10% de 10 reais.

Quanto ao valor impresso no exemplo anterior, você notou que este valor contém cinco casas decimais? Isto ocorre porque uma instância de **FastMoney** tem uma precisão máxima de cinco casas decimais, conforme mencionamos nas seções anteriores.

Suponha agora que você tem um valor monetário, mas gostaria de obter somente os centavos que compõem este valor. Por exemplo, do valor R\$ 10,22, você gostaria de obter R\$ 0,22. Como fazer isso? Basta usar o método estático **minorPart()** da classe **MonetaryUtil**, conforme a **Listagem 10**.

Após usarmos alguns dos operadores fornecidos pela própria Moneta, vamos criar uma implementação própria para exemplificar o uso da interface **MonetaryOperator**. Para termos um exemplo que se aproxime mais da vida real, suponha que estamos criando uma aplicação para uma loja que vende roupas femininas. A dona da loja nos pediu que calculássemos um desconto de 10% para toda compra que seja acima de R\$ 1.000,00. Como podemos fazer isso? Com um **MonetaryOperator**.

Listagem 9. Obtendo 10% de um valor monetário.

```
package br.com.devmedia;

import org.javamoney.moneta.FastMoney;
import org.javamoney.moneta.function.MonetaryUtil;

import javax.money.CurrencyUnit;
import javax.money.Monetary;
import javax.money.MonetaryAmount;

public class Operacoes {

    public static void main(String... args) {
        CurrencyUnit currency = Monetary.getCurrency("BRL");
        MonetaryAmount dezReais = FastMoney.of(10, currency);
        System.out.println(dezReais.with(MonetaryUtil.percent(10)));
    }
}
```

Listagem 10. Obtendo apenas os centavos de um valor monetário.

```
package br.com.devmedia;

import org.javamoney.moneta.FastMoney;
import org.javamoney.moneta.function.MonetaryUtil;

import javax.money.CurrencyUnit;
import javax.money.Monetary;
import javax.money.MonetaryAmount;

public class Operacoes {

    public static void main(String... args) {
        CurrencyUnit currency = Monetary.getCurrency("BRL");
        MonetaryAmount minorPart = FastMoney.of(10.22, currency);
        System.out.println(minorPart.with(MonetaryUtil.minorPart()));
    }
}
```

Inicialmente, precisamos criar uma classe que implemente a interface **MonetaryOperator** e, em seguida, definir o corpo do método **apply()** de forma que ele dê o desconto que desejamos. Esta classe encontra-se na **Listagem 11** com o nome de **Desconto**. Note que ela recebe dois parâmetros em seu construtor: o valor mínimo para receber o desconto e a porcentagem do desconto. Uma vez que temos esses valores armazenados, nós os usamos dentro do método **apply()** para calcular o desconto a ser dado no valor passado como parâmetro para este método, retornando-o logo em seguida.

Nota

A próxima versão da Moneta, que será lançada em breve, não terá mais a classe **MonetaryUtil**, conforme usamos neste artigo. Ela será substituída pela classe **MonetaryOperators**, contudo, os métodos continuarão os mesmos, apenas com a adição do método **rounding()**, cujo objetivo é obter valores arredondados.

RoundedMoney

Uma vez entendido como funciona um **MonetaryOperator**, podemos analisar a classe **RoundedMoney** da Moneta com mais

detalhes. Você notará nos exemplos desta seção que sempre que criamos uma instância desta classe, passamos para ela uma instância de **MonetaryOperator**, que será executada toda vez que uma operação for realizada.

Para entender melhor seu funcionamento vamos observar a **Listagem 12**, na qual usamos a classe **Desconto**, criada nas seções anteriores. Iniciamos esta listagem criando uma instância de **Desconto**, informando para ela que o valor mínimo para dar o desconto é de R\$ 200,00 e que o valor do desconto será de 10%. Em seguida, criamos uma instância de **RoundedMoney** usando o método estático **of()**, passando um terceiro parâmetro, que é a ins-

Listagem 11. Criando nosso próprio MonetaryOperator.

```
package br.com.devmedia;

import org.javamoney.moneta.function.MonetaryUtil;

import javax.money.MonetaryAmount;
import javax.money.MonetaryOperator;

class Desconto implements MonetaryOperator {
    private Double valorDesconto;
    private MonetaryAmount valorMinimo;

    public Desconto(MonetaryAmount valorMinimo, Double valorDesconto) {
        this.valorDesconto = valorDesconto;
        this.valorMinimo = valorMinimo;
    }

    public MonetaryAmount apply(MonetaryAmount monetaryAmount) {
        MonetaryAmount result = null;

        if (monetaryAmount.isGreaterThanOrEqualTo(valorMinimo)) {
            MonetaryAmount desconto = monetaryAmount.with(MonetaryUtil.
                percent(valorDesconto));
            result = monetaryAmount.subtract(desconto);
        }

        return result;
    }
}
```

Listagem 12. Exemplo de uso de RoundedMoney.

```
package br.com.devmedia;

import org.javamoney.moneta.FastMoney;
import org.javamoney.moneta.Money;
import org.javamoney.moneta.RoundedMoney;

import javax.money.MonetaryAmount;

public class Rounded {

    public static void main(String... args) {
        MonetaryAmount valorMinimo = Money.of(200, "BRL");
        Double valorDesconto = 10D;
        Desconto desconto = new Desconto(valorMinimo, valorDesconto);

        MonetaryAmount money = RoundedMoney.of(100, "BRL", desconto);
        System.out.println(money);

        money = money.add(FastMoney.of(300, "BRL"));
        System.out.println(money);
    }
}
```

tância de **Desconto** que acabamos de criar. Logo após, criamos um **MonetaryAmount** no valor de R\$ 100,00 e imprimimos seu valor no console. Note que será impresso o valor BRL 100 no console.

Prosseguindo, adicionamos R\$ 300,00 e voltamos a imprimir o valor. Desta vez, no entanto, o valor impresso não será R\$ 400,00, pois nossa classe **Desconto** será executada, uma vez que esse valor supera o valor mínimo para dar desconto (200 reais). Portanto, será impresso o valor R\$ 360,00, que corresponde a 10% de desconto no valor de R\$ 400,00.

Formatação e exibição

Até este momento, sempre que precisávamos exibir um valor monetário, o imprimíamos diretamente no console, sem nenhum tratamento. Você notou que esses valores eram sempre impressos no formato BRL + Valor? Lembre-se que os caracteres BRL referem-se ao código da nossa moeda. Contudo, obviamente, este formato não é interessante para exibição ao usuário, uma vez que nossa moeda é normalmente exibida usando o símbolo R\$. Então, como fazemos para imprimir dessa forma?

A resposta está nas classes **AmountFormatQuery**, **AmountFormatQueryBuilder** e **MonetaryAmountFormat**. Usando essas três classes em conjunto, podemos formatar valores monetários da forma que desejarmos.

Vejamos o exemplo da **Listagem 13**, onde iniciamos criando uma instância de **FastMoney** com o valor de dez reais. Em seguida, para exibir este valor formatado, precisamos de uma instância de **AmountFormatQuery**. Para esta tarefa, usamos a classe **AmountFormatQueryBuilder**, que é responsável por criar instâncias de **AmountFormatQuery**. Desta classe, chamamos o método estático **of()**, que recebe como parâmetro um **Locale**.

Listagem 13. Exemplo de formatação de dinheiro.

```
package br.com.devmedia;

import org.javamoney.moneta.FastMoney;
import org.javamoney.moneta.format.CurrencyStyle;

import javax.money.MonetaryAmount;
import javax.money.format.AmountFormatQuery;
import javax.money.format.AmountFormatQueryBuilder;
import javax.money.format.MonetaryAmountFormat;
import javax.money.format.MonetaryFormats;
import java.util.Locale;

public class Formatacao {

    public static void main(String[] args) {
        MonetaryAmount amount = FastMoney.of(10, "BRL");

        AmountFormatQuery query = AmountFormatQueryBuilder.of(
            Locale.getDefault()).set(CurrencyStyle.SYMBOL).build();
        MonetaryAmountFormat format =
            MonetaryFormats.getAmountFormat(query);

        System.out.println(format.format(amount));
    }
}
```

Neste caso, estamos pedindo ao **AmountFormatQueryBuilder** para criar um **AmountFormatQuery** conforme as peculiaridades do **Locale** informado.

Logo após, definimos o estilo de exibição do valor monetário. Em nosso exemplo, pedimos para formatá-lo com o símbolo da moeda, usando a enumeração **CurrencyStyle.SYMBOL**. Finalizamos chamando o método **build()** para concluir a criação de uma instância de **AmountFormatQuery**, e que será passada como parâmetro para o método **getAmountFormat()** de **MonetaryFormats**.

Agora que temos uma instância de **MonetaryFormat**, podemos chamar o método **format()**, passando como parâmetro o **MonetaryAmount** que desejamos formatar. Em nosso exemplo, note que será impresso o valor R\$ 10,00.

O enum **CurrencyStyle** ainda possui mais três opções:

1. **CurrencyStyle.CODE**: Exibe o código da moeda, no caso do real brasileiro, será impresso "BRL 10,00";
2. **CurrencyStyle.NAME**: Exibe o nome da moeda, no caso do real brasileiro, será impresso "Real brasileiro 10,00";
3. **CurrencyStyle.NUMERIC_CODE**: Exibe o código numérico da moeda, no caso do real brasileiro, será impresso "986 10,00".

Conversão entre moedas

Vejamos agora como podemos fazer a conversão entre tipos de moedas, uma vez que a Money API não nos permite fazer operações com moedas de países diferentes. Por exemplo, o código da **Listagem 14** lançará uma exceção **javax.money.MonetaryException**, pois estamos tentando somar um valor em Real com outro valor em Dólar americano. Na Money API, todas as operações precisam ser realizadas sempre em moedas de um mesmo tipo.

Então, como posso converter um valor em Real para Dólar ou vice-versa? Para isto, a Money API nos fornece a interface **ExchangeRateProvider**, que já vem com várias implementações padrão na Moneta. Entretanto, caso você queira implementar sua própria forma de conversão, basta implementar esta interface e executar as operações necessárias para fazer a conversão. Em nossos exemplos, usaremos as implementações já providas pela Moneta, que são:

1. **ExchangeRateProvider.ECB**: Obtém o valor de conversão da moeda no Banco Central Europeu;
2. **ExchangeRateProvider.IMF**: Obtém o valor de conversão da moeda no Fundo Internacional Monetário;
3. **ExchangeRateProvider.IMF_HIST**: Obtém o valor de conversão da moeda de uma data específica no Fundo Internacional Monetário;
4. **ExchangeRateProvider.ECB_HIST90**: Obtém o valor de conversão da moeda nos últimos 90 dias no Banco Central Europeu;
5. **ExchangeRateProvider.ECB_HIST**: Obtém o valor de conversão da moeda de uma data específica a partir de 1999 no Banco Central Europeu.

Como você já deve ter percebido, para usar os provedores de cotação listados, precisaremos de uma conexão ativa com a internet,

pois será necessário realizar uma consulta às bases de dados destes provedores para obter a cotação atual de cada moeda.

Para entender melhor o uso desta API analisaremos a **Listagem 15**, onde verificamos quanto corresponde o valor de um dólar em reais, conforme a cotação do dia fornecida pelo Banco Central Europeu (ECB). Para esta tarefa, iniciamos obtendo uma instância de **ExchangeRateProvider** chamando o método estático **getExchangeRateProvider()** de **MonetaryConversions**. Este método aguarda como parâmetro uma **String** que informa qual o provedor desejado, contudo, você não precisa se preocupar em memorizar esses valores, pois para isso existe o **enum** chamado **ExchangeRateType**.

Listagem 14. Exemplo de erro na conversão de moedas diferentes.

```
package br.com.devmedia;

import org.javamoney.moneta.FastMoney;

import javax.money.MonetaryAmount;

public class ExchangeError {

    public static void main(String[] args) {
        MonetaryAmount real = FastMoney.of(10, "BRL");
        MonetaryAmount dolar = FastMoney.of(10, "USD");

        real.add(dolar);
    }
}
```

Listagem 15. Exemplo de uso da conversão de moedas.

```
package br.com.devmedia;

import org.javamoney.moneta.ExchangeRateType;
import org.javamoney.moneta.FastMoney;

import javax.money.CurrencyUnit;
import javax.money.Monetary;
import javax.money.MonetaryAmount;
import javax.money.convert.CurrencyConversion;
import javax.money.convert.ExchangeRateProvider;
import javax.money.convert.MonetaryConversions;

public class Exchange {

    public static void main(String... args) {
        ExchangeRateProvider exchangeRateProvider = MonetaryConversions
            .getExchangeRateProvider(ExchangeRateType.ECB);

        MonetaryAmount dolar = FastMoney.of(1, "USD");
        CurrencyUnit currencyReal = Monetary.getCurrency("BRL");

        CurrencyConversion currencyConversion = exchangeRateProvider
            .getCurrencyConversion(currencyReal);
        MonetaryAmount real = currencyConversion.apply(dolar);

        System.out.println(real);
    }
}
```

Após termos uma instância do provedor da ECB, precisamos criar um **MonetaryAmount** que represente o valor 1 dólar, conforme já vimos no decorrer deste artigo, para que possamos convertê-lo de dólar para real. Neste exemplo, usamos a classe **FastMoney** chamando o método **of()** e passando como parâmetro o valor 1 e o tipo monetário USD.

Em seguida, precisamos de um conversor de dólar para real, que na Money API é representado pela classe **CurrencyConversion**, obtida a partir do método **getCurrencyConversion()** da instância de **ExchangeRateProvider** que criamos previamente. Observe que este método aguarda uma instância de **CurrencyUnit** como parâmetro e que informa para qual unidade será realizada a conversão.

Com uma instância de **CurrencyConversion** em mãos, podemos chamar seu método **apply()** passando como parâmetro uma instância de **MonetaryAmount** que representa o valor a ser convertido. Note que este exemplo retornará um valor diferente conforme a cotação da moeda do dia. Para realizar a conversão para outras moedas, basta usar uma **CurrencyUnit** diferente, conforme suas necessidades.

Enquanto analisamos os recursos da Money API o Java 9 continua sendo planejado e implementado pela Oracle e comunidade.

Já para começar a explorar as novas opções, o primeiro passo é testar os recursos da Money API e avaliar na prática como a codificação de requisitos mais trabalhosos, como a conversão de valores monetários, se torna mais simples.

Autor



Marlon Silva Carvalho

marlon.carvalho@gmail.com

Programador poliglota há mais de 20 anos, é fã de programação para dispositivos móveis e 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](http://marlonscarvalho) ou acesse sua página pessoal em <http://marlon.co/>.



Links:

Página da JSR 354.

<https://www.jcp.org/en/jsr/detail?id=354>

Página da implementação de referência Moneta.

<http://javamoney.github.io/ri.html>

eBook sobre a Money API de Otávio Santana.

<http://otaviojava.gitbooks.io/money-api/>

Página do projeto Joda Money.

<http://www.joda.org/joda-money/>

Site do projeto Java 9.

<http://openjdk.java.net/projects/jdk9/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



WildFly - Do básico ao ambiente de produção

Aprenda conceitos, características e configurações para trabalhar com este servidor de aplicações e colocá-lo em produção

Aplataforma Java EE (*Java Platform Enterprise Edition*) é um padrão amplamente adotado no desenvolvimento de softwares corporativos, pois oferece uma solução robusta, portável, escalável e que atende demandas de acesso, transações, segurança e outras necessidades que uma aplicação de grande porte possa ter. Desenvolvido através do *Java Community Process*, este padrão conta ainda com contribuições de organizações comerciais e open source, além de experts da área.

Para utilizar esta plataforma, no entanto, é preciso de um servidor de aplicações, que vai prover a infraestrutura necessária para rodar a aplicação. Felizmente, hoje em dia existe um leque com boas opções de servidores homologados e certificados para o Java EE 7 – versão mais atual desta tecnologia. Apesar das opções, este estudo será focado em apenas uma delas, o WildFly, solução que substitui o *JBoss Application Server* e que está extremamente leve, rápida e é amplamente utilizada no Brasil e no mundo.

Portanto, este artigo abordará tópicos como história, diferença entre as versões AS, EAP e WildFly, funcionalidades disponíveis, características, clusterização, load balancer, assim como tópicos sobre segurança, tuning e microserviços.

De onde vem o WildFly?

O WildFly, que antes se chamava JBoss AS, é um servidor de aplicações open source, escrito em Java, baseado nos padrões definidos pela especificação Java EE e mantido pela comunidade e pela empresa Red Hat.

Para contextualizar, seguem alguns pontos importantes na história do WildFly:

- **1999:** Foi lançada a primeira versão, com o nome EJBoss (*Enterprise Java Beans in Open Source System*). Era um servidor que atendia à especificação EJB 1.0;

Fique por dentro

Este artigo é útil para desenvolvedores que têm interesse em conhecer mais sobre o WildFly, para que possam resolver mais facilmente possíveis problemas durante o desenvolvimento, e também para profissionais de infraestrutura e devops, que utilizam ou têm interesse em utilizar o WildFly em produção. Para isso, serão apresentados conceitos e características importantes para se ter um ambiente com alta disponibilidade e escalável adotando cluster e load balancer com segurança e performance. Assim, o leitor irá adquirir conhecimentos que lhe permitirão tomar decisões sobre o uso ou não deste servidor de aplicações e como configurá-lo de acordo com suas necessidades.

- **2000:** Na versão 1.0, o nome passou a ser JBoss Application Server ou JBoss AS;
- **2011:** Foi lançada o JBoss AS 7.0, quando houve a maior alteração de todas as versões. A arquitetura do JBoss foi remodelada em módulos utilizando OSGi, o que fez o servidor de aplicações se tornar extremamente mais rápido;
- **2013:** O projeto foi renomeado para WildFly, pois já existiam vários outros projetos com o nome JBoss e também para não confundir com a versão comercial do servidor de aplicação, chamada JBoss EAP. Assim sendo, não existe JBoss AS 8, e sim WildFly 8;
- **2015:** Foi lançado o WildFly 9 e também liberada uma versão beta do WildFly 10.

Comunidade

Uma vantagem de utilizar o WildFly é poder contar com a geração de conteúdo e suporte da comunidade. Existem inúmeros blogs no Brasil falando constantemente sobre características, resolução de problemas, montando ambientes em diversos cenários e apresentando as novidades.

Também há um grupo de usuários chamado JBUG:Brasil, onde o fórum é bastante ativo e as dúvidas postadas, em português,

são rapidamente respondidas. Caso tenha interesse, você pode fazer seu cadastro no JBUG e postar suas dúvidas, assim como responder dúvidas de outras pessoas.

Nota

Se sua empresa necessita de suporte e também quer ser atendida rapidamente caso encontre algum bug no servidor de aplicações, a Red Hat oferece a versão comercial, com direito a suporte e bug fixes, chamada JBoss EAP (JBoss Enterprise Application Platform).

Perfis e modos de operação

Para trabalhar com o WildFly é importante conhecer os perfis existentes, que são os conjuntos de tecnologias disponíveis para uma aplicação deployada no servidor, e os modos de operação standalone e domain, que determinam a estrutura de administração e de configurações do ambiente.

Perfis

Segundo a especificação Java EE, o WildFly possui diferentes conjuntos de tecnologias que podem ser utilizadas em uma aplicação. Estes conjuntos são chamados de perfis e possibilitam que uma aplicação mais simples possa fazer uso de um perfil que oferece menos tecnologias, com o objetivo de carregar apenas o que vai precisar.

O WildFly possui quatro perfis e todos eles podem ser customizados, caso necessário, para adicionar, alterar ou remover determinada tecnologia. A seguir são apresentadas as quatro opções:

- **web:** Esse é o perfil default e não possui algumas tecnologias como mensageria (JMS) e batch, além de não permitir a criação de um cluster;
- **full:** Caso precise utilizar JMS ou alguma tecnologia não existente no perfil web, deve-se adotar este perfil;
- **ha:** Para montar um ambiente de alta disponibilidade criando um cluster de servidores WildFly, utilize o perfil "ha" (de "high availability");
- **full-ha:** Caso precise de tecnologias não existentes no perfil web e seja necessário montar um cluster, então opte por este perfil.

Para fazer uso de um perfil diferente do default, basta passar como parâmetro, na inicialização do WildFly, o comando `--server-config=standalone-<nome do perfil>.xml`, pois na pasta `configuration`, além do arquivo `standalone.xml`, que é o arquivo de configurações do perfil default, existe um arquivo para cada tipo de perfil, devendo esse arquivo ser informado na inicialização do servidor através do parâmetro supracitado.

Modo Standalone e Modo Domain

Imagine uma empresa que tenha 15 servidores WildFly rodando em cluster e em um determinado momento foi alterada a senha do banco de dados. Ao fazer isso, normalmente teríamos que acessar cada um dos servidores para realizar a atualização. Pensando em evitar esse trabalho repetitivo, é possível automatizar essa tarefa em uma ferramenta como o Jenkins, mas você tem a possibilidade

de ter uma instância do WildFly gerenciando todas as demais, e assim, qualquer configuração para qualquer máquina pode ser feita em um só lugar. Este modo de operação onde a administração é centralizada é chamado de modo domain.

O modo standalone, por sua vez, é a opção mais utilizada em ambientes de desenvolvimento e também em ambientes de produção com um ou poucos servidores WildFly, pois com um menor número de servidores, é simples realizar as configurações máquina por máquina. Contudo, se tiver muitas máquinas, é recomendado usar o modo domain para centralizar a administração.

Configurando o WildFly

Antes de 2011, até a versão 6 do JBoss AS, havia inúmeros arquivos XML que podiam ser modificados para realizar uma configuração. Por exemplo: a configuração do banco de dados ficava em um arquivo, e para trocar a porta HTTP do JBoss você tinha que alterar outro arquivo. Agora, tudo está centralizado em um único local, em um único arquivo (`standalone.xml`), e para facilitar, pode-se alterá-lo via admin console ou via linha de comando com o JBoss CLI.

Direto no XML

O arquivo `standalone.xml`, localizado na pasta `configuration`, é dividido em categorias (também chamadas de `subsystems`) como datasource, logging, ejb e security, de forma a organizar as configurações e facilitar a busca pelo lugar exato para conferir uma informação ou alterar o que for necessário. Assim, para alterações relacionadas a banco de dados como, por exemplo, conexões, senhas, pool, entre outros parâmetros, deve-se procurar o subsystem `datasource`.

Vale ressaltar que caso utilize um perfil que não seja o `default`, deve-se trabalhar com o arquivo de configurações do perfil em questão. Por exemplo, para o perfil `full-ha`, deve-se utilizar o arquivo `standalone-full-ha.xml`, que é semelhante ao `standalone.xml`, mas com mais subsystems. Note que há um arquivo para cada perfil.

No modo domain, por sua vez, deve-se alterar o `domain.xml`. Este arquivo é maior e possui todos os perfis dentro dele. É neste local que devemos configurar os grupos de servidores que farão parte do seu domínio e definir qual é o perfil de cada grupo.

Interface web: admin-console

Você também pode realizar qualquer configuração via web console. Para isso é necessário criar um usuário através do script `add-user.bat` (no Windows) ou `add-user.sh` (no Linux), que ficam na pasta `bin` da instalação do WildFly.

Além de realizar configurações, consegue-se monitorar algumas informações do seu WildFly, visualizar e baixar os logs, fazer deploy, entre outras funcionalidades, tudo de forma simples, pois nas últimas versões houve muitas melhorias na interface do admin console a fim de facilitar e melhorar a experiência do usuário.

CLI – Command Line Interface

Com o WildFly, qualquer configuração que pode ser feita via XML, ou via admin console, pode ser feita via linha de comando, através do CLI (*Command Line Interface*).

O CLI é extremamente útil para que uma alteração efetuada em um ambiente seja exatamente a mesma realizada em outro. Por exemplo, ao alterar determinada linha no arquivo XML no seu ambiente local e solicitar que outra pessoa altere essa mesma linha no XML em outro ambiente pode gerar algum erro. Com o CLI basta enviar o comando que deve ser executado no outro ambiente. Após isso, a alteração terá sido realizada sem a necessidade de abrir qualquer arquivo e talvez mexer em outra coisa que não deveria ter sido modificada.

Outro diferencial é que podemos utilizar o CLI para automatizar tarefas como alterações de configurações ou deploys em ferramentas de integração contínua como o Jenkins.

Para acessar o CLI, vá até a pasta *bin* do WildFly e digite *jboss-cli.bat -connect* no Windows ou *./jboss-cli.sh -connect* no Linux. Caso deseje configurar um WildFly remoto, é possível acessá-lo via CLI passando IP e porta como parâmetro. Por fim, a partir do WildFly 9 consegue-se utilizar o CLI mesmo se o WildFly não estiver rodando.

Outras formas

Além das opções supracitadas, também pode-se configurar ou obter dados de configuração do WildFly utilizando a API Java do WildFly, acessando via API REST ou ainda via JMX, porém, esses três caminhos não são tão comuns e não serão detalhados aqui. No entanto, é interessante saber que eles existem.

Construindo um ambiente de produção

Para configurar um ambiente de produção é importante definir o perfil utilizado e também se ele será executado no modo stand-alone ou no modo domain. Além disso, pode ser preciso configurar um load balancer ou até um cluster, assim como definir onde esse ambiente ficará hospedado. Estes assuntos serão abordados nos tópicos a seguir.

Load balancer através de um servidor HTTP

O que um servidor HTTP faz pode ser explicado de forma simples. Ele espera requisições de web browsers (clientes) e retorna os dados requisitados. No entanto, ele também pode fazer cache de sua aplicação, aumentar a segurança do seu ambiente, permitir uma arquitetura de alta disponibilidade, entre outras possibilidades. Sabendo disso, caso queira ter mais de uma instância do WildFly rodando com a sua aplicação a fim de garantir performance, escalabilidade e alta disponibilidade, pode-se utilizar o Apache HTTP Server (um dos mais populares HTTP Servers) com o *mod_jk* ou com o *mod_cluster* (projeto da Red Hat) para realizar o load balancer, ou seja, para distribuir os acessos entre suas instâncias WildFly, como demonstra a **Figura 1**.

Tanto o *mod_jk* quanto o *mod_cluster* são bibliotecas que permitem realizar o load balancer. Quem já trabalhou com load balancer nos últimos anos deve conhecer o *mod_jk*, porém, o *mod_cluster*,

que é um novo projeto da Red Hat e é uma excelente opção em questões de simplicidade de configuração, desempenho e flexibilidade, já vem sendo muito utilizado em conjunto com o WildFly.

Note que um servidor HTTP também é um importante componente de segurança, pois funciona como um proxy reverso, ou seja, tudo que entrará na sua aplicação passará por ele. Assim, ele representa uma camada a mais que deverá ser acessada antes de um possível invasor chegar ao servidor onde está seu WildFly ou banco de dados.

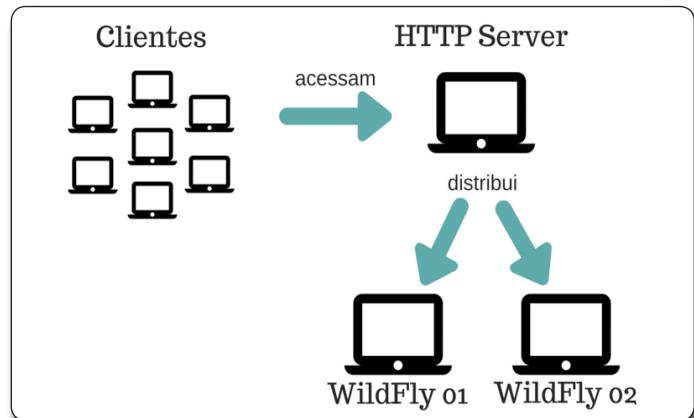


Figura 1. Ambiente com Load Balancer

Nota

A partir da versão 9 já é possível utilizar o próprio WildFly como servidor HTTP, substituindo assim o Apache HTTP Server. A arquitetura mostrada na **Figura 1** seria a mesma. Isto é viável devido aos avanços do Undertow, novo web server do WildFly que substituiu o JBoss Web Server (que é um fork do Tomcat). Essa nova feature ainda não é tão comum em ambientes de produção por ser recente, mas já está funcional

Outra vantagem em utilizar um servidor HTTP fazendo load balancer é que podemos atualizar sua aplicação sem tirá-la do ar. Caso tenha um Apache distribuindo seus acessos para dois servidores WildFly, pode-se parar um deles e todos os seus usuários continuarão acessando normalmente a aplicação, pois ainda há um WildFly rodando. Esse é o momento para atualizar a aplicação no WildFly parado e subi-lo novamente. Quando o WildFly atualizado subir, pode-se parar o outro e atualizá-lo seguindo o mesmo procedimento.

Ao efetuar a atualização da forma citada, no entanto, os usuários que estiverem utilizando a aplicação no momento podem perder sua sessão, tendo que se logar no sistema novamente, o que pode levar à perda de algum dado que estava sendo salvo em um EJB Stateful, por exemplo. Para que isso não aconteça, tem-se que criar um cluster para replicar a sessão dos usuários entre os servidores WildFly.

Cluster

Caso tenha uma aplicação onde seja importante que o usuário não perca sua sessão e tenha que logar novamente, você precisará

implementar um cluster, pois isto possibilitará a replicação de sessão entre servidores, o que permite que se um servidor cair, seu usuário continue logado e não perceba a queda, pois os dados estarão replicados entre os outros servidores.

Para montar um cluster, é necessário configurar o load balancer com um HTTP Server para distribuir os acessos entre os servidores sem mudar a URL que o usuário utiliza para acessar a aplicação, e também usar o perfil *ha* ou *full-ha* para viabilizar a comunicação entre os servidores. Vale ressaltar, no entanto, que o modo de operação não influencia, ou seja, pode-se montar um cluster tanto com o modo standalone, quanto com o modo domain.

O WildFly utiliza a biblioteca JGroups para encontrar e se comunicar, através de multicast (vide **BOX 1**), com os outros membros do cluster e também utiliza o Infinispan, que é uma plataforma dedatagrid baseada na JSR-107 (JSR relacionada a cache), para armazenar e manter sincronizados os dados da aplicação replicados no cluster.

BOX 1. Multicast

Conforme Marchioni (2013), multicast é o protocolo onde os dados são transmitidos simultaneamente a um grupo de hosts que estão dentro da mesma rede de multicast, semelhante a um canal de televisão, onde a imagem e som são transmitidos apenas àqueles que estiverem sintonizados em determinada frequência.

Montar um ambiente em cluster pode até ser algo simples, porém, adiciona certa complexidade em seu ambiente, tornando-o um pouco mais difícil de manter, e isto também pode gerar outros problemas que não seriam encontrados em um ambiente sem cluster (uma classe que não implementa *Serializable* e está na sessão ou algum lookup de um serviço EJB que da forma que foi implementado não funciona em cluster). Portanto, é sempre interessante analisar a relação custo x benefício.

Onde rodar?

Você pode deixar seu ambiente rodando em uma infraestrutura interna de sua empresa ou mantê-lo na nuvem, em serviços como o Digital Ocean, Amazon EC2 ou OpenShift. Através do Digital Ocean e do Amazon EC2, que são provedores IaaS (proveem infraestrutura como serviço), tem-se uma máquina onde é possível instalar o WildFly e configurar tudo como desejar. Já no OpenShift, que é um PaaS (oferece uma plataforma como serviço), é fornecida uma máquina já configurada com o WildFly instalado e com ferramentas que lhe auxiliarão na configuração e manutenção do ambiente.

Como um diferencial, com o OpenShift pode-se deployar a aplicação utilizando um banco de dados como PostgreSQL ou MongoDB em poucos minutos, e para fazer deploy basta realizar um push no repositório Git do próprio OpenShift. Já na Amazon EC2 e no Digital Ocean tem-se como vantagens a flexibilidade e o controle, pois teremos total acesso à máquina, podendo instalar e configurar o que quiser sem necessariamente estar preso a algum padrão ou especificidade de um serviço PaaS.

E a segurança? Por onde eu começo?

Nas primeiras versões do JBoss era necessário realizar algumas configurações básicas de segurança antes de colocá-lo em produção, mas muitas pessoas não faziam e deixavam seus servidores vulneráveis a acessos externos maliciosos. Nas últimas versões, por outro lado, ao fazer o download do WildFly, descompactá-lo e iniciá-lo sem alterar qualquer configuração, você vai perceber que:

- É possível acessar o WildFly (pelo browser) apenas da máquina onde ele foi iniciado;
- Ao acessar o admin console, verifica-se que não é possível visualizar as configurações do servidor, pois é preciso criar um usuário administrador antes;
- E ao criar o usuário e tentar digitar uma senha simples, o admin será alertado para criar uma senha mais difícil de ser adivinhada.

Isso tudo serve para evitar problemas de segurança como acessos indevidos ou descoberta de senhas através de força bruta caso alguém apenas faça o download do WildFly, realize o deploy de uma aplicação e já a coloque no ar. Contudo, mesmo com essas validações padrão, existem inúmeros cuidados que devem ser tomados para aumentar a segurança do seu ambiente. A seguir serão listados alguns itens que devem ser verificados.

Nunca inicie como root

Você nunca deve iniciar um servidor de aplicações com o usuário *root*, a não ser que tenha total consciência do que está fazendo e tenha um motivo para isso. É aconselhável criar um usuário específico para iniciar o WildFly ou usar um usuário com direitos limitados, com acesso apenas à pasta onde está instalado o servidor.

Isso é recomendado porque uma aplicação Java consegue executar comandos do sistema operacional para manipular diretórios e arquivos. Assim sendo, se o leitor iniciar seu WildFly com um usuário com permissões de *root* e alguém conseguir deployar uma aplicação utilizando estes recursos do Java de executar comandos do sistema operacional, a pessoa poderá obter ou excluir qualquer arquivo no servidor.

Acesso externo

Verifique se alguma seção administrativa ficou aberta para acesso externo. Faça testes tentando acessar de outra máquina o WildFly via CLI e também tentando entrar no admin-console via browser e veja se por acaso ficou alguma seção visível para outras máquinas. Deixar o admin console exposto e ainda com uma senha fácil de ser descoberta não é uma boa ideia.

As configurações para não expor as seções administrativas podem ser feitas de duas formas:

- Na sua infraestrutura, liberando para acesso externo do seu servidor ou da sua rede apenas as portas relativas à aplicação e não as portas administrativas;
- Diretamente no WildFly, na seção **interfaces** do *standalone.xml*, onde configura-se o que deseja-se disponibilizar para acesso

externo como **public**, e o que não é recomendável ser liberado como **management**.

O que vai ser liberado pode variar se a configuração estiver definida com um servidor HTTP na frente, se utiliza HTTPS ou não, entre outras possibilidades. No entanto, independentemente do seu contexto, é válido realizar os testes de acesso às seções administrativas e, se detectada alguma brecha de segurança, corrigi-la.

Atualizações

Sempre busque manter seu WildFly atualizado, assim como seu sistema operacional, o Java e, caso utilize, seu servidor HTTP também. Versões mais recentes dos softwares geralmente acrescentam melhorias de segurança. Deste modo, mantê-los atualizados é uma excelente forma de diminuir vulnerabilidades de seu ambiente.

Role Based Access Control

A partir do WildFly 8 é possível fazer o controle de acesso à parte administrativa baseado em papéis. Assim sendo, pode-se, por exemplo, definir que um usuário com um papel que poderá apenas visualizar o estado atual do servidor e reiniciá-lo se necessário, enquanto outro usuário pode ter permissões para alterar configurações e fazer deploys, bem como ter um usuário com permissão total. Existem sete papéis com permissões diferentes que podem ser utilizados conforme sua necessidade e ainda pode-se criar outros papéis combinando dois ou mais papéis existentes. Para mais detalhes, consulte o artigo sobre RBAC (*Role Based Access Control*) na documentação do WildFly, indicada na seção **Links**.

Log de Auditoria

Caso tenha vários usuários acessando o admin console, vale a pena utilizar o novo sistema de auditoria, que garante que qualquer alteração no modelo de gerenciamento seja registrada em um log para análise posterior, se necessário. Dessa forma, pode-se obter informações de todas as mudanças realizadas no admin console, como o usuário responsável pela alteração e o horário. Isso pode ajudar a compreender ou evitar alterações indevidas. Para utilizar esta funcionalidade deve-se ativá-la na seção **audit-log** do *standalone.xml*, conforme a **Figura 2**, e acompanhar o log gerado na pasta *data* do WildFly.

Um pouco de tuning

Existem inúmeras configurações que podem ser feitas na sua aplicação ou infraestrutura que vão melhorar a performance da sua solução, mas quando se fala em tuning, é interessante primeiro identificar os gargalos, ou seja, os pontos que mais estão utilizando recursos computacionais como memória ou processamento. Um gargalo pode ser:

- Uma tela do sistema que é acessada muitas vezes e faz inúmeras requisições;
- Uma consulta ao banco de dados que traz muitos registros;
- Um pool de EJB muito pequeno;
- Entre outros.

```
<audit-log>
    <formatters>
        <json-formatter name="json-formatter"/>
    </formatters>
    <handlers>
        <file-handler name="file" formatter="json-formatter" relative-to="jboss-server">
            <log-boot="true" log-read-only="false" enabled="true">
                <handlers>
```

Figura 2. Habilitando o log de auditoria

Essa visão holística para identificar os pontos mais críticos é importante para saber onde e como atuar e não simplesmente aumentar a memória da máquina. Trabalhar nos gargalos de sua aplicação será mais eficiente para melhorar a performance. A seguir será mostrado como identificar os pontos críticos e algumas dicas de ações que podem ser tomadas.

Testes de desempenho e monitoração

Existem duas atividades que devem ser feitas constantemente que ajudam a identificar gargalos: testar e monitorar. Você pode realizar testes de desempenho utilizando JMeter ou outras ferramentas e através dos testes simular, por exemplo, mil usuários acessando uma mesma tela em um minuto e monitorar como sua aplicação se comporta.

Durante os testes também é interessante monitorar a aplicação com ferramentas como VisualVM, JProfiler, New Relic, Datadog, entre outras, para obter mais informações de como ela está lidando com os acessos simulados. Estas ferramentas ainda podem ser utilizadas para monitoramento constante da sua aplicação.

Através de testes e monitoração pode-se verificar, por exemplo, que existem milhares ou milhões de instâncias de uma classe e que isso está consumindo muita memória, identificar que estas instâncias estavam sendo criadas dentro de um loop **for**, e que simplesmente alterar o algoritmo melhorará consideravelmente o uso de memória pela sua aplicação.

Memória

A fim de obter um melhor uso da memória RAM, é importante configurar adequadamente as variáveis **-Xms** e **-Xmx** para o seu WildFly. O **-Xms** define o quanto de memória será alocado inicialmente para ser utilizado. Essa quantidade pode aumentar automaticamente, de acordo com a necessidade da aplicação, até o quanto foi definido em **-Xmx**. Em ambientes de produção é recomendável manter esses dois valores iguais, para que não seja preciso alocar mais memória depois que a aplicação estiver no ar.

Além disso, não é indicado colocar um valor muito alto para **-Xms** e **-Xmx**, pois isso pode prejudicar a performance, já que o Garbage Collector funciona melhor com valores não tão altos. Portanto, é mais apropriado ter quatro WildFlys com 4GB do que um com 16GB.

Geralmente são especificados valores entre 1GB e 4GB para estes parâmetros de memória, variando de acordo com a infraestrutura disponível e com a aplicação. Para descobrir o valor mais adequado é necessário realizar testes de desempenho e monitorar como a

aplicação se comporta com diferentes valores de `-Xms` e `-Xmx` para identificar o valor ideal.

Em casos específicos onde a aplicação vai utilizar mais memória do que o comum como, por exemplo, em um sistema que possui relatórios que envolvam muito processamento de dados ou em um projeto que não foi desenvolvido aplicando boas práticas de programação, é preciso realizar uma configuração com valores maiores de `-Xms` e `-Xmx` e especificar outros ajustes na JVM para obter um melhor desempenho. Para entender melhor a JVM e como configurá-la para aprimorar sua performance, leia o artigo “JVM Performance Optimization”, indicado na seção **Links**.

Java 8

O Java 8 trouxe inúmeras novidades, como as APIs de Datas, Streams e as expressões lambda, mas além destas ele trouxe também um Garbage Collector renovado e um melhor uso de memória e processamento. Inspirado em alguns pontos da JVM *JRockit Mission Control*, que sempre teve foco em alta performance, o Java 8 fornece um melhor aproveitamento dos recursos computacionais. Assim, caso sua aplicação esteja em uma versão anterior do Java, uma simples atualização de versão pode tornar sua aplicação mais performática.

Caso tenha alguma aplicação rodando em Java 6 e por algum motivo não possa migrar para o Java 8, faça testes com a JRockit e

veja a diferença. Sua aplicação provavelmente ficará visivelmente mais rápida. Inclusive aplicações com Java 7 poderiam ser migradas para o JRockit (que é Java 6) para melhorar a performance. Porém, perderia os recursos do Java 7. Assim sendo, o ideal seria migrá-la para o Java 8.

Logging

As configurações default de log do WildFly são ótimas para o ambiente de desenvolvimento, mas para o ambiente de produção é recomendado diminuir a quantidade de log gerado, pois a escrita e leitura de dados no disco (também chamada de I/O) consome muito processamento.

No WildFly podemos configurar quatro níveis de log:

- **ERROR:** Exibe apenas erros do sistema ou da aplicação. Esse é o nível que menos exibe logs.
- **WARN:** Mostra situações que podem gerar um erro ou que necessitam análise, além de exibir tudo que o nível ERROR captura;
- **NFO:** Adiciona ao nível WARN informações sobre eventos que acontecem na aplicação;
- **DEBUG:** É o nível que mais gera logs, registrando de forma refinada o que acontece no sistema. É muito útil durante o desenvolvimento ou para ter mais detalhes para encontrar a causa de algum erro;

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

Dessa forma e tendo em vista diminuir o processamento de I/O, é útil diminuir o log gerado pelo WildFly quando sua aplicação já estiver estável, alterando para isso o subsystem **logging** para os níveis WARN ou ERROR. Lembre-se, no entanto, que ao diminuir a quantidade de log consegue-se uma melhoria na performance, mas recebe-se menos informações que podem ser úteis ao analisar algo que aconteceu no sistema.

O log, por padrão, é gerado no console (terminal de comandos), onde o WildFly foi iniciado, e também no arquivo de log *server.log*. Para o ambiente de produção você pode, no subsystem de **logging**, desativar a geração de log no console adicionando **enabled=false** na tag **console-handler** (vide Figura 3) a fim de diminuir o I/O.

```
<subsystem xmlns="urn:jboss:domain:logging:2.0">
  <console-handler name="CONSOLE" |enabled="false">
    <level name="INFO"/>
    <formatter>
      <named-formatter name="COLOR-PATTERN"/>
    </formatter>
  </console-handler>
```

Figura 3. Desabilitando o log no console

Ademais, verifique se no código da sua aplicação também não está sendo gerado muito log desnecessário. Gerar log em um código que é executado muitas vezes, como em um filtro HTTP, que é chamado a cada requisição, pode tornar sua aplicação mais lenta. Pense nisso!

Tuning em cada tecnologia

Ainda com o intuito de melhorar a performance da aplicação, recomenda-se realizar tuning para cada tecnologia que estiver utilizando, por exemplo:

- Adotar uma versão mais recente do JSF pode trazer um melhor uso de memória;
- Otimizar imagens para que fiquem mais leves, minificar arquivos JavaScript e CSS, entre outras estratégias de performance front-end certamente melhorarão a performance;
- Aumentar o pool de EJBs pode permitir mais usuários acessando simultaneamente a aplicação;
- Analisar os mapeamentos e queries no JPA também pode trazer inúmeros benefícios.

Esse tipo de análise pode ser feito em todas as camadas da aplicação e para todas as tecnologias empregadas, pois por mais que exista um gargalo que gere o maior impacto na performance, cada melhoria realizada influencia positivamente na aplicação como um todo.

WildFly como microserviço

Um dos termos mais discutidos nos últimos anos em conferências e blogs de desenvolvimento de software é a arquitetura de microserviços. Ela representa uma maneira de projetar aplicações como suítes de pequenos serviços independentes,

que rodam em processos separados se comunicando com mecanismos leves como REST. Para mais informações a respeito, veja o artigo “Microservices”, de Martin Fowler e James Lewis, indicado na seção **Links**.

Um exemplo de microserviço seria uma consulta a um banco de dados, disponibilizada via REST, onde todo o código necessário para isso estaria empacotado em um único JAR, que podemos rodar através de um comando *java -jar*, em vez de ter um WAR com inúmeros serviços rodando em um servidor de aplicações.

Através do projeto WildFly Swarm conseguimos construir microserviços em Java com poucas linhas de código e colocá-los no ar utilizando poucos recursos computacionais. O Swarm é um projeto novo, lançado em 2015, mas já é possível encontrar inúmeros exemplos e tutoriais na Internet.

Além deste, existem outros projetos Java, há mais tempo no mercado, que permitem a criação de microserviços. Este é o caso do Grizzly e do Spring Boot. Uma das vantagens do Swarm, no entanto, é que você utilizará as mesmas tecnologias do WildFly, podendo assim ser mais fácil o desenvolvimento caso domine essas tecnologias ou queira aumentar a compatibilidade com os demais projetos que já possuir rodando neste servidor de aplicações.

Outras novidades

Além dos microserviços e outros assuntos já abordados neste artigo, há muita coisa nova nas últimas versões do WildFly. Nos tópicos a seguir serão destacadas algumas delas.

Alterar portas ficou muito mais simples

Imagine que seu ambiente tenha um WildFly rodando em uma máquina e, por algum motivo, é solicitada mais uma instância do servidor de aplicações na mesma máquina. Se o admin subir outra instância sem os ajustes necessários, acontecerão vários conflitos de portas. Em versões anteriores, para contornar esse problema, era preciso acessar e reconfigurar vários arquivos.

A partir do JBoss AS 7 todas as configurações de porta também estão centralizadas no *standalone.xml*. Para facilitar ainda mais, podemos simplesmente passar um parâmetro na inicialização com o valor que gostaríamos de incrementar em todas as portas: *-Djboss.socket.binding.port-offset=200*. Dessa forma, todas as portas serão incrementadas em 200, ou seja, a porta 8280 passará a ser utilizada no lugar da porta 8080, a 10190 no lugar de 9990 e assim por diante.

Aplicar patches de correção e de atualização

Uma das preocupações da equipe do WildFly é com a atualização do servidor depois de instalado. Deste modo, as novas versões estão sempre trazendo melhorias ao processo de aplicação de patches, tornando-o cada vez mais simples e com menos impactos, podendo até ser realizado remotamente.

Usando JavaScript no back-end

A partir do WildFly 10 será possível escrever código JavaScript para o back-end e disponibilizá-lo via REST de uma maneira seme-

lhante ao funcionamento do NodeJS. Isso se tornou viável por causa da engine de JavaScript chamada Nashorn, presente no JDK 8.

Assim, com poucas linhas você poderá, por exemplo, escrever em JavaScript um serviço REST que acesse diretamente um banco de dados e retorne os dados solicitados em formato JSON, conforme demonstra a **Listagem 1**.

Listagem 1. Código JavaScript no lado Servidor com WildFly 10.

```
$undertow
.onGet("/rest/members",
{headers: {"content-type": "application/json"}},
['jndi:java:jboss/datasources/ExampleDS', function ($exchange, db) {
    return db.select("select * from member");
}]);
```

Suporte ao HTTP/2

A nova versão do protocolo HTTP, desenvolvida com foco em performance, já é suportada pelo WildFly. Infelizmente, nem todos os browsers já suportam esse protocolo, mas ainda assim, é interessante ver esse trabalho em andamento, pois o HTTP/2 será um grande avanço para a web.

O WildFly é um servidor de aplicações robusto que atende desde pequenas aplicações até produtos que exigem alta disponibilidade, performance e segurança. Para se destacar ainda mais em sua categoria, configurá-lo e utilizá-lo é simples e sua performance torna ainda mais produtivo o trabalho de desenvolvedores e administradores de ambiente.

A partir deste artigo o leitor está apto a dar os primeiros passos com o WildFly em seus projetos e inclusive colocá-lo em um ambiente de produção. Ademais, caso precise de mais detalhes para implementar um cenário mais complexo, você já conhece o necessário para ter um norte em seu planejamento.

Além do que foi abordado, o WildFly tem inúmeras outras características e opções que podemos utilizar de acordo com o contexto da aplicação. Para conhecer tudo isso, existe muito material na internet, ótimos livros e também o grupo de usuários JBUG:Brasil, que pode lhe auxiliar em suas dúvidas, desde as mais simples até as mais complexas.

Autor



Adriano Schmidt

adriano@localhost8080.com.br – www.localhost8080.com.br

Arquiteto de Software e UX Strategist, programador Java desde 2007 e trabalha com JBoss desde a versão 3, formado em Administração de Empresas, fazendo MBA em Marketing Digital na FGV e é ator de teatro como hobby. Além disso, testa sistemas em uma perspectiva de Usabilidade e UX após beber muita cerveja no www.oUsuarioEstaBebado.com



Links:

Authorizing management actions with Role Based Access Control.

<https://docs.jboss.org/author/display/WFLY9/RBAC>

Using Server-Side JavaScript with WildFly.

<http://wildfly.org/news/2015/08/10/javascript-Support-In-Wildfly/>

JVM Performance Optimization - Eva Andreasson.

<http://www.javaworld.com/article/2078623/core-java/jvm-performance-optimization-part-1-a-jvm-technology-primer.html>

Microservices. Martin Fowler e James Lewis.

<http://martinfowler.com/articles/microservices.html>

WildFly 8 – Administration Guide. Marchioni, Francesco – 2014.

<http://itbuzzpress.com/ebooks/wildfly-8-book.html>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Dominando o Spring Boot

Conheça a nova ferramenta do ecossistema Spring que garante ganho de produtividade, qualidade e satisfação em seus novos projetos

Há pouco mais de um ano – em abril de 2014 – a Spring IO disponibilizava a primeira versão não-beta do Spring Boot, a 1.0.0, depois de mais de dezoito meses de desenvolvimento e maturação. Desde então, o projeto tem evoluído em grande velocidade e, na data de escrita deste artigo, já se encontra na versão 1.2.5, com a 1.3.x batendo a porta.

O Spring Boot é um projeto-chave para a Spring IO, mas, por ser recente, muitos desenvolvedores ainda não tiveram um primeiro contato com ele e muitos dos já introduzidos ainda desconhecem algumas ou várias de suas possibilidades e recursos. É uma ferramenta de alavancagem de produtividade e qualidade. Assim, se você é um desenvolvedor que lida com aplicações baseadas em Spring, deve ao menos conhecê-la.

Ao ler este artigo você entenderá as motivações por trás desse projeto, assim como sua importância, não só para a Spring IO, como para a indústria de software atual. Também irá aprender, se já não sabe, como dar seus primeiros passos no uso do Spring Boot, assim como orientá-lo em casos excepcionais às suas configurações pré-fixadas. Portanto, mesmo àqueles que não estão envolvidos com o Spring em seu dia a dia, ou o fazem muito esporadicamente, este artigo trará muita informação útil, de uma forma ou de outra.

Com a finalidade de exercício dos conceitos, mecanismos e pontos apresentados, este artigo também traz um pequeno projeto web que expõe um serviço REST de consulta de aeroportos pelo mundo através dos critérios país, cidade ou código da *International Air Transport Association*, IATA. É uma aplicação um tanto mais complexa que um 'Hello World!', mas simples ao ponto de não perdermos de vista nosso foco, a compreensão e uso do Spring Boot.

Para construir e executá-la é preciso apenas do JDK, do Maven e estar conectado à Internet, para que o

Fique por dentro

Ao ler este artigo você irá saber como o Spring Boot se encaixa no contexto histórico do Java EE e Spring e também entender as relações entre eles, além de conhecer as motivações por trás desse projeto da Spring IO. Também irá aprender como construir uma aplicação com o Spring Boot e como customizá-la de acordo com suas necessidades. A partir disso, você poderá adotar em seus projetos uma solução que reduzirá a quantidade de configurações e opinará pelas tecnologias a serem utilizadas visando a alta produtividade.

mesmo descarregue as dependências do projeto. Apesar de ser inteiramente funcional com o Java 6 e 7, replicamos aqui a recomendação do documento de referência do Spring Boot em utilizá-lo com o Java 8 – como veremos, entretanto, configurá-lo a uma ou outra versão é tão simples como escrever o número dela. Quanto ao Maven, é necessário que seja a versão 3.2+. É possível também utilizar o Gradle (1.12+), mas como o primeiro é mais conhecido e já faz parte do dia a dia da maior parte dos desenvolvedores, nosso projeto irá adotá-lo.

Apesar de código e recursos em arquivo não somarem muito mais que uma dezena de itens, recomenda-se o uso de uma IDE, preferencialmente o Spring Tool Suite (STS) – mas só pelo fato de ele possuir uma integração natural com componentes Spring, como o próprio Boot, o que facilita a execução e depuração com o mínimo esforço.

Para entendermos por que o Spring Boot foi criado, precisamos conhecer um pouco da história do Spring Framework. E para que possamos entender, de fato, o Spring Framework, precisamos conhecer um pouco da história do J2EE, Java 2 – Enterprise Edition. Comecemos, então, a discorrer sobre essas histórias e suas relações, a começar pelo J2EE. É provável que alguns dos leitores já saibam delas, mas vale a pena ver de novo.

J2EE, Java 2 – Enterprise Edition

O J2EE, parte da plataforma Java, é similarmente uma plataforma, mas específica para o desenvolvimento de aplicações Java em arquitetura de multicamadas, modular e distribuída, executadas em servidores de aplicações. Ela não é exatamente um produto, mas um conjunto de especificações, originalmente desenvolvido pela Sun Microsystems (casa original do Java, hoje na Oracle, como sabemos), e lançado evolutivamente em versões. A partir da versão 1.3 a especificação da plataforma passou ao domínio da JCP (*Java Community Process*), que a controla através de JSRs (*Java Specification Requests*), como a JSR-58, que especifica o J2EE 1.3 (2001), a JSR-151, que especifica o J2EE 1.4 (2002), e assim por diante.

O Java EE – passemos a utilizar a sigla atual a partir daqui – inclui diversas especificações de API, como o JDBC, e-mail, JMS, web services, RMI, JTA, JPA, JSF, JMX, JNDI, manipulação de XML, etc. e como usá-las de maneira coordenada. Há também especificações únicas a seus componentes, como EJB, servlets, portlets e diversas tecnologias de serviços web. Todas essas especificações estão interligadas com o propósito de permitir a criação de aplicações corporativas – termo comum que usamos para designar uma aplicação Java EE – portáveis entre os servidores de aplicação, os quais disponibilizam todas as especificações como infraestrutura de maneira uniforme, além de garantir escalabilidade, concorrência e gerenciamento dos componentes e aplicações entregues (*deployed*) neles. O objetivo é permitir ao desenvolvedor manter o máximo possível de seu foco na construção do negócio de sua aplicação e alavancar produtividade e qualidade. Observe na **Figura 1** a visão geral do Java EE. A **Figura 2** apresenta uma visão mais detalhada e integrada.

Java EE – Do discurso à realidade

Não há como negar que o Java EE tenha sido e seja um sucesso. Diferentemente do CORBA, *Common Object Request Broker Architecture*, da OMG, e do DCOM/COM+, da Microsoft, – só para citar duas das propostas próximas em intenção e modelo de mais destaque na época do surgimento do Java EE – sua adoção pelo mercado e pelos desenvolvedores foi significativamente maior. Além de contar com a força crescente da plataforma Java à época, a estratégia de delegar a especificação comunitária à JCP pela Sun Microsystems e seu foco em dominar, ou ao menos estar entre os mais importantes nomes do mercado de soluções para o desenvolvimento e execução de aplicações corporativas, impulsionaram a consolidação e evolução autossustentável da plataforma Java EE.

Não há também como negar que o Java EE tenha tido muitos problemas em seus primeiros anos. Todos ou boa parte deles vêm sendo tratados a cada nova versão das diversas especificações. De qualquer forma, no entanto, o desconforto causado deu espaço a novos modelos de desenvolvimento, manutenção e execução de aplicações corporativas, como é o caso do Spring Framework.

O desconforto deveu-se principalmente a três fatores. O primeiro deles é que apesar de todo o cuidado e visão das especificações, o como elas eram implementadas e coordenadas em um servidor de aplicações Java EE ficou bastante vago e até mesmo ausente

em alguns aspectos nas primeiras versões, afetando definições de segurança, representação de campos CMP (*Container Managed Persistence*) na classe de bean, ordem de deployment de módulos, para citar alguns. As diversas implementações tentaram endereçar da melhor maneira possível algumas das lacunas de definição citadas. Alguns exemplos comerciais e open source de servidores de aplicação são:

- IBM – WebSphere;
- WebLogic/BEA – WebLogic;
- Adobe – JRun;
- JBoss/Red Hat – JBoss AS (recentemente substituído pelo WildFly);
- Apache – Geronimo, TomEE;
- Object Web – JOnAS;
- Caucho Technology – Resin.



Figura 1. Visão geral da pilha Java EE

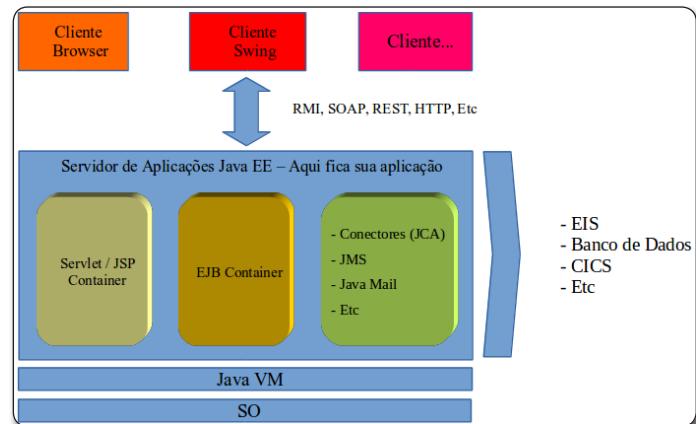


Figura 2. Visão integrada da pilha Java EE

Essa quantidade de opções, apesar de vibrante, afetava um dos pilares conceituais do Java EE: a portabilidade. Cada produto tinha seu conjunto de ferramentas e especializações de configurações e o resultado final é que o único artefato portável era o código da aplicação. Iniciou-se, então, um programa de certificação para servidores de aplicação, mas sua adoção foi lenta.

O JBoss AS, por exemplo, só se tornou certificado a partir de sua versão 4. Apesar de ser ideal estar 100% fiel à especificação, é mais importante manter a base de usuários e clientes que simplesmente ter um selo de conformidade.

O código de uma aplicação Java EE, vale a pena frisar, talvez seja sua parte mais descomplicada e que consuma menos tempo no flu-

xo de entrega, ao contrário de sua configuração, empacotamento e deployment. Lembremos que a intenção de todas as especificações é liberar a aplicação de todo código de infraestrutura, como segurança, transação, resolução de nomes, camada de persistência, detalhes de mensageria, etc., os quais devem ser disponibilizados e coordenados pelo servidor de aplicações. Obviamente, a aplicação necessita configurar o uso desses componentes junto ao servidor. Na prática, há muito que se configurar através, principalmente, de descritores em XML.

Desde o início soube-se da complexidade da configuração no Java EE. Por isso mesmo foram formulados guias para o ambiente de desenvolvimento e entrega de aplicações nos quais existiam diversos papéis além do desenvolvedor, como o do empacotador, entregador e administrador. Não é difícil imaginar que poucas empresas possuíam ou possuem tal estruturação. Na vida real os papéis acabam se acumulando nos desenvolvedores, afetando outro pilar conceitual do Java EE: o foco no negócio.

O segundo fator de desconforto do Java EE, como já foi de certa forma delineado, é a configuração complexa – e como foi ressaltado no parágrafo anterior, diferindo conforme o produto utilizado. Também foi ressaltado no parágrafo anterior que na prática o papel de configuração acaba se acumulando no desenvolvedor. Por contraditório que seja ou não, quando elas afetam ambientes de produção ou qualquer outro ambiente sensível, a responsabilidade é das equipes de operação, cujo modus operandi difere fundamentalmente dos das equipes de desenvolvimento. Essa impedância histórica entre essas áreas acaba afetando outro pilar conceitual do Java EE: a produtividade/agilidade. Curiosamente essa tensão vem alavancado modelos de Desenvolvimento versus Operacionalização, que hoje chamamos de DevOps. Também é interessante observar que o termo *Convention Over Configuration* – convenção sobre configuração, apesar de não estar exclusivamente ligado ao Java EE, recebeu um novo acento desde então.

O terceiro fator de desconforto é o sobrepeso. Uma aplicação Java EE pode contar com todas as facilidades expostas pelo servidor de aplicações, mas ela precisa? Na prática, observamos que muito poucas utilizam todas as facilidades. A imensa maioria precisa apenas do Servlet Container e da camada de persistência através de JPA/JDBC. Eventualmente utiliza mensageria por JMS para implementar padrões assíncronos. Utilizar o JAAS para segurança é muitas vezes muito complicado para os requisitos da aplicação e o desenho acaba optando por mecanismos mais leves e de simples uso para tal. Enfim, a aplicação ainda é corporativa, mas utiliza apenas uma pequena parte das facilidades disponíveis. Apesar disso, sendo executada em um servidor Java EE, irá pagar por todos os ônus associados. Esse é o motivo de encontrarmos boa parte dessas aplicações sendo executadas em produtos que cumprem apenas parte das especificações, como os Servlet Containers Tomcat e Jetty, e produtos exclusivos de mensageria com JMS e ESB (*Enterprise Service Bus*), como o Mule, e o WSO2. Assim, as aplicações cumprem seus requisitos em ambientes costumeiramente mais simples de implantar e gerenciar, tanto para as equipes de desenvolvimento quanto operações.

Apesar dos desconfortos citados, como foi dito inicialmente, o Java EE os tem tratado e vem evoluindo a cada nova versão. E mesmo com todos eles, não há como negar que desenvolver uma aplicação corporativa com o Java EE seja muitíssimo mais simples e gerenciável que com CORBA, por exemplo.

Spring Framework

Em novembro de 2002, Rod Johnson publicou pela editora Wrox, o livro: *Expert One-on-One J2EE Design and Development*. O objetivo de seu trabalho foi: 1) apontar o quanto complexo se tornava o desenvolvimento e manutenção da maioria das aplicações corporativas quando se seguia ortodoxamente o guia do J2EE; e 2) apresentar uma alternativa que fosse mais simples, menos custosa, com tempo de entrega de mercado mais realista, mais gerenciável e com melhor performance. Acompanhavam o texto mais de 30.000 linhas de código em Java de um framework inicialmente batizado como Interface21 e futuramente renomeado como Spring Framework. Muitos dos conceitos e mecanismos fundamentais do Spring já estavam nessas linhas, a saber: um container de IoC – *Inversion of Control*, ou Inversão de Controle – com um BeanFactory, um ApplicationContext e um DI, ou *Dependency Injection* – Injeção de Dependência – capaz e funcional, os primórdios do Spring MVC com Controller, HandlerMapping e associados, o JdbcTemplate e o conceito de acesso a dados agnóstico à tecnologia. Esses componentes são o básico para a implementação de uma aplicação web CRUD utilizando-se o Spring. Citando Rod Johnson, o nome Spring *foi inspirado pelo fato de se buscar uma Primavera, novos ares, ao desenvolvimento Java EE sob o Inverno de sua abordagem tradicional*.

Diante de tudo isso, o Spring Framework não levou muito tempo para conquistar o público. Não foram só as dificuldades inerentes ao desenvolvimento e manutenção do Java EE tradicional que o impulsionaram. O conceito e mecanismos de DI são extremamente poderosos, pois atuam sobre um dos principais vilões do desenvolvimento de software moderno, o acoplamento. Desacoplar significa simplificar, ampliar extensibilidade e modularidade, melhorar o processo de testes, reduzir impactos negativos, aumentar produtividade e qualidade. Coincidência ou não, esses atributos são muito alinhados às metodologias de desenvolvimento ágil e evolução contínua. Ou seja, houve uma conjunção de fatores que alavancaram exponencialmente a aceitação e uso do Spring como framework de desenvolvimento Java EE não tradicional.

Categoricamente, hoje, o Spring é uma plataforma Java que disponibiliza, de forma ordenada e comprehensiva, uma infraestrutura para o desenvolvimento de aplicações corporativas. O Spring cuida da infraestrutura e você, desenvolvedor, de sua aplicação/negócio. Muito parecido com o proposto pelo Java EE tradicional, não? A diferença está em como o Spring faz. Em uma única expressão, ele não é intrusivo. Ao contrário, podemos desenvolver toda a aplicação utilizando apenas POJO – Plain Old Java Object, classes simples. Não há necessidade de se implementar ou estender nenhum artefato alienígena aos nossos requisitos, como EJB. O Spring simplesmente aplica serviços corporativos (JPA, JTA,

JMX, JNDI, JMS, etc.) conforme solicitemos. Para realizar esse feito, ele se utiliza do container de IoC e também de mecanismos de AOP – *Aspect Oriented Programming*. Isso nos possibilita, citando seu próprio manual de referência:

- Executar um método Java num contexto transacional de banco de dados sem ter que conhecer ou manipular de forma alguma JTA;
- Executar um método Java local que acesse um remoto sem ter que conhecer ou manipular de forma alguma RMI, por exemplo;
- Utilizar-se de mecanismos de gerenciamento sem ter que conhecer ou manipular de forma alguma JMX;
- Tornar um método Java local em um handler de mensageria sem ter que conhecer ou manipular de forma alguma JMS;
- Entre outras possibilidades.

Reforçando, a única coisa que codificamos são POJOs. Todo o resto ligamos, como se brincássemos com Lego.

O Spring pretende ser uma solução leve e one-stop-shop, ou seja, uma caixa de ferramentas completa para desenvolvimento Java EE, como ilustra a **Figura 3**. Entretanto, ele é extremamente modular em si, permitindo utilizar apenas o que necessitamos. Portanto, pode-se empregar o container IoC com qualquer framework web, inclusive o Spring MVC, ou apenas suas abstrações para persistência, por exemplo.

Tipicamente, uma aplicação com o Spring é desenvolvida para executar em simples Servlet Containers, como o Tomcat ou Jetty, mas nada nos impede de a termos em um servidor de aplicações e até mesmo *ligar* componentes Spring a uma aplicação Java EE tradicional.

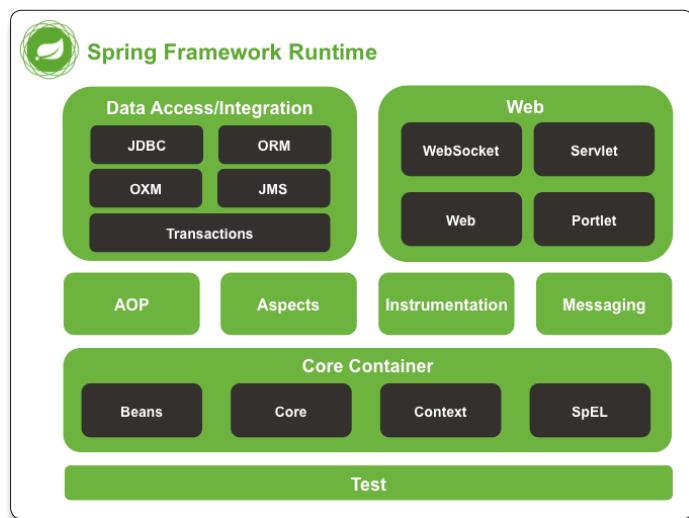


Figura 3. Runtime do Spring Framework

Spring – Do discurso à realidade

Inicialmente, o Spring Framework possuía seu container de IoC e seus mecanismos para executar a carga – o bootstrapping – do contexto dos Beans, inclusive nossos POJOs, conforme configurado em arquivos XML que definem os mesmos. Também possuía uma série de facilidades e mecanismos para persistência de dados, leitura de recursos e MVC. Em suma,

disponibilizava o modelo e ferramentas para o desenvolvimento de aplicações Java EE web de uma forma descomplicada, eficiente (e até mesmo prazerosa para aqueles que conheciam o trabalho árduo de se desenvolver o equivalente com o Java EE tradicional).

Rod Johnson, Juergen Hoeller e Yann Caroff decidiram então fundar uma empresa, a SpringSource, centrada no projeto open source do Spring Framework. A comunidade e adesão crescentes o impulsionaram a uma evolução rápida e marcante, sempre mantendo o espírito Java EE descomplicado de sua origem. Gradativamente foram sendo adicionadas novas funcionalidades e mecanismos para segurança, como o Spring Security, SOAP web services, com o Spring WS, padrões de integração corporativa, com o Spring Integration, infraestrutura de serviços em batch, com o Spring Batch, entre tantos outros. Além disso, os componentes preexistentes, como o Spring MVC, foram se tornando mais e mais inteligentes e completos. Ao final de alguns anos o Spring Framework se tornou um tipo de ecossistema de desenvolvimento Java EE.

Seu poder de penetração como novo modelo de desenvolvimento corporativo fez com que a empresa VMware comprasse a SpringSource em 2009 por cerca de 420 milhões de dólares. Nos dias de hoje o Spring está sob o nome da Pivotal, junção criada pela VMware, EMC Corporation e GE para manter todos os produtos de software para aplicações, como o Spring. Apesar de sempre ter estado sob a direção de alguma empresa, o Spring é e sempre será um projeto open source, sob licença Apache 2.0 e intrinsecamente comunitário.

Esse ecossistema, hoje denominado Spring IO, é um conjunto de projetos contendo o Spring Boot, Spring Framework, Spring XD, Spring Cloud, Spring Data, Spring Integration, Spring Batch, Spring Security, Spring HATEOAS, Spring Social, Spring AMQP, Spring Mobile, Spring for Android, Spring Web Flow, Spring Web Services, Spring LDAP, Spring Session, Spring Roo e alguns ainda por vir, como o Spring Stethemachine. Ufa, quanto projeto! Essa quantidade de soluções se deve ao modus operandi do Spring, que abraça novos desafios no desenvolvimento EE conforme eles surjam, através da incubação e maturação de novos projetos e componentes.

Esse aspecto pragmático e eclético está no cerne da primeira das duas principais críticas ao Spring: a falta de uma concepção coesa em detrimento de uma coleção inconsistente de soluções de melhores práticas a aspectos diversos do desenvolvimento Java EE. A segunda crítica é o excesso de dependência de configuração em XML, que consome muito mais tempo do desenvolvedor que o próprio desenvolvimento de código. Essas duas críticas são pertinentes. A crítica com relação à coesão talvez seja exagerada, mas não há como negar que o espírito do Spring tenha muito mais compromissos com eficiência e pragmatismo do que com concisão conceitual. Isso não vai, a princípio, mudar. Com relação à dependência de muita configuração em XML, trata-se do desdobramento natural da diversidade e abrangência do próprio ecossistema. É necessá-

rio configurar o MVC, o Security, o Integration, seus próprios Beans, etc. É necessário dizer ao Spring como ele irá carregar o contexto, enfim, e quais os mapeamentos e ligações entre os componentes, entre outras coisas.

No entanto, assim como o Java EE tradicional, a cada versão o Spring busca simplificar o trabalho com configuração. Uma das medidas foi o uso do JavaConfig desde a versão 3, mecanismo pelo qual as definições em XML passam a ser substituídas por anotações em classes de configuração. Mesmo com o JavaConfig (vide **BOX 1**), ainda há a necessidade de configurar tanto quanto sejam os recursos utilizados. Por isso o Spring também endereça de forma cada vez mais inteligente o aspecto *Convention Over Configuration*, assumindo muitas coisas através da presença de uma ou outra dependência no classpath da aplicação ou várias opções de comportamento default para o MVC, mapeamentos e etc., sempre, obviamente, deixando a cargo do desenvolvedor seguir ou fugir dessas configurações default. O resultado prático é que é muito mais simples desenvolver a mesma aplicação com o atual Spring 4 que com o Spring 1, 2 ou 3, basicamente porque o desenvolvedor perde muito menos tempo com configuração.

BOX 1. JavaConfig

Spring Java Config, ou JavaConfig, é o mecanismo que nos possibilita configurar o Spring IoC diretamente de nossas classes Java por meio de metadados na forma de annotations – presentes no Java desde sua versão 5 –, não mais dependendo de arquivos de definição de beans em XML. As principais vantagens em se utilizar o JavaConfig são:

- Mecanismo objeto orientado para a injeção de dependência, o que significa que podemos tirar proveito de reuso, herança e polimorfismo em nosso código de configuração.
- Controle completo de instânciação e injeção de dependência, o que significa que mesmo dependências muito complexas podem ser tratadas de forma elegante e clara

Ainda assim, o Spring perde em agilidade se desenvolvemos a mesma aplicação com soluções paralelas mais recentes, como o Ruby On Rails, Django, Play e Node.js, para citar algumas, que apesar de não serem tão abrangentes e generalistas (ainda), atuam no mesmo espaço de desenvolvimento EE. Ciente de seus próprios problemas e tentando se equiparar em agilidade e ferramental a esses paralelos, ou mesmo incorporar boas ideias deles, o Spring desenvolveu ferramentas como o Spring Roo, porém sua aceitação foi muito baixa.

Spring Boot

O Spring Boot, de forma diferente do Roo (acima de tudo não há geração de código), busca solucionar a complexidade da inicialização e gerenciamento de dependências de um projeto com Spring, além de tratar de maneira coesa e eficiente a questão da configuração, fazendo uso extensivo de *Convention Over Configuration*.

Essencialmente, o Spring Boot pode ser considerado um plugin para a ferramenta de building, seja ela o Maven ou o Gradle. Seus principais objetivos são gerenciar dependências

de maneira opinativa e automática, e simplificar a execução do projeto em tempo de desenvolvimento e depuração. Com o comando `mvn spring-boot:run`, por exemplo, o Maven irá executar sua aplicação na porta 8080. Isso é bastante similar ao plugin Maven do Jetty, que se tornou uma forma popular de se desenvolver e testar uma aplicação web utilizando Spring ou outros frameworks. O Spring Boot também possui a funcionalidade de empacotamento da sua aplicação em um JAR executável contendo todas as dependências necessárias, inclusive o Servlet Container, seja ele o Tomcat, Jetty ou mesmo Undertow, apesar de ainda ser possível empacotar um WAR da forma tradicional.

O principal benefício do Boot, entretanto, é a configuração de recursos baseada no que se encontra no classpath. Se o POM de seu Maven inclui a dependência do JPA e o driver do PostgreSQL, ele irá criar uma unidade de persistência baseada no PostgreSQL. Se adicionarmos alguma dependência web, iremos perceber que o Spring MVC assumirá configurações default e dependências com relação a diversos aspectos, como a tecnologia de apresentação (o default é o Thymeleaf), o mapeamento de recursos e marshalling de JSON (o default é o Jackson) e/ou XML (o default é o JAXB 2) para o tratamento de dados de requisição e resposta, que necessitamos em uma aplicação REST, por exemplo.

O Spring Boot é tudo acerca de opinião, ou seja, a menos que especifiquemos explicitamente, ele irá trazer dependências pre-definidas, assim como instanciar e injetar beans predefinidos conforme o tipo de aplicação que estamos construindo (iremos ver como funciona esse mecanismo em nossa aplicação exemplo). Podemos não concordar com algumas opiniões como, por exemplo, a versão de dada dependência ou o Servlet Container default, mas ao menos ele as tem e, caso insistamos com a nossa, ele não se opõe. Por exemplo, se nosso POM declara o uso de persistência com o Spring Data, mas não declaramos nada mais, será assumido que queremos utilizar JPA e Hibernate para a camada de persistência. Se estamos implementando uma aplicação web e não declaramos nada em específico, o Boot irá assumir que usaremos o Thymeleaf como view resolver.

Em suma, o que o Spring Boot faz principalmente, de maneira muito elegante, é nos livrar de qualquer preocupação desnecessária com dependências e configurações que são as mesmas em 99% dos casos. E se o caso está no 1% restante, ele acata nossa opinião com a mesma elegância, como veremos em nosso projeto.

Spring Boot – Perspectivas

Não há como negar a crescente tendência do desenvolvimento de aplicações corporativas estarem migrando para outras linguagens e modelos além do Java EE. Independentemente disso, há muita lenha a queimar com o Java, seja com o tradicional Java EE ou com o Spring. Esse último, entretanto, precisava impor alguma ordem e coesão à sua própria criatividade, e o Spring Boot é essa ordem e coesão. Mais ainda: o modelo de empacotamento em standalone JAR

casa-se muito bem com as unidades de execução desejáveis em uma arquitetura de microserviços e conteinerização que vem tomando tanto corpo ultimamente. Em outras palavras, o Boot não só clareia o horizonte do Spring, mas também o energiza significativamente. Por isso é um projeto tão importante para o Spring e consequentemente para o Java EE, sendo recomendado que desenvolvedores o entendam e façam uso dele sempre que possível.

Desenvolvendo uma aplicação para consulta de aeroportos

Para demonstrar o Spring Boot na prática, iremos desenvolver uma pequena aplicação para consulta de aeroportos que faz uso de uma base de dados gratuita mantida pela OpenFlights – uma ferramenta web (vide [Links](#)) que oferece funcionalidades de busca, estatísticas e rotas de voos comerciais pelo mundo. Basicamente, essa base contém o nome de dado aeroporto, seus códigos IATA e ICAO, coordenadas geográficas, fuso, cidade e país. Nossa aplicação web de serviços REST irá fornecer no formato JSON:

- Os detalhes de dado aeroporto a partir de seu código IATA;
- Uma lista de detalhes de aeroportos a partir de determinada cidade;
- Uma lista de detalhes de aeroportos a partir de determinado país.

Ao executá-la, poderemos acessar seus serviços através do cURL ou um navegador, como ilustra a [Listagem 1](#).

Ferramentas e ambiente para a codificação e execução do projeto

Para a execução, depuração e codificação do projeto, certifique-se que seu ambiente atenda os seguintes requisitos:

- Maven 3.2+ instalado;
- JDK 6+ instalado;
- Conexão com a Internet, pois o Maven irá descarregar as dependências (a menos que você já as possua em seu repositório local);
- O IDE de sua preferência – não é obrigatório, mas ajuda a visualizar os artefatos e código.

O código fonte completo do projeto pode ser obtido na área de download desta edição. Após realizar o download, desempacote-o e execute conforme a [Listagem 2](#) (foi utilizado um ambiente Linux aqui, mas não difere muito para o Windows).

Após a aplicação subir, experimente acessar as URLs conforme a [Listagem 1](#), com o cURL ou seu navegador. Você irá obter uma saída como a da [Listagem 3](#).

Entendendo nossa aplicação

Nossa aplicação possui três serviços REST, que executam a busca de aeroportos pelos critérios cidade, país ou código IATA. Após a busca, uma lista contendo um ou mais aeroportos é retornada em JSON, caso haja resultado; caso contrário, o conteúdo da resposta é vazio. Observe que cada um dos serviços REST é atendido por uma controladora. Para a busca por cidade, temos a controladora CitySearch; por país, a CountrySearch; e por código IATA, a IataCodeSearch. Poderíamos ter uma única

Nota

Para os que não conhecem, cURL é um programa de linha de comando que utilizamos, principalmente em ambientes Unix/Linux, mas também em ambientes Windows, para efetuar GET, POST, PUT, etc. em dada URL. Além desta opção, também é possível acessar os serviços de nossa aplicação pelo navegador, como o Chrome. Neste caso, no entanto, não é necessário adicionar o código de espaço à URL, como no trecho "sao%20paulo" no exemplo da [Listagem 1](#), pois o próprio navegador se encarregará disso.

controladora para todos os serviços, mas optamos pela divisão aqui, por maior clareza e separação de responsabilidades.

As controladoras são especificadas de modo mais detalhado a seguir:

- **CitySearch** – Retorna os aeroportos por cidade em JSON. Para isso, atende à URI `/openflights/city/{cidade}`, onde `{cidade}` é o nome da cidade desejada. No serviço, não há restrições de escrita com letras maiúsculas e/ou minúsculas, ou seja, Brasilia, brasilia, BRASILIA, etc., resultam na mesma pesquisa para a cidade de Brasília. Caso não haja aeroportos, nada será retornado;
- **CountrySearch** – Semelhante a **City-Search**, mas retorna os aeroportos por país e atende à URI `/openflights/country/{pais}`;
- **IataCodeSearch** – Retorna um aeroporto caso exista seu código IATA na base de dados. Atende à URI `/openflights/iata/{iataCode}`.

As controladoras utilizam a interface **FlightService** para realizar a pesquisa. Essa interface existe apenas para tornar abstrata a fonte de dados, pois não importa à camada de controle se os dados vêm de um banco de dados, um serviço externo ou um arquivo. **OpenFlightService** é a implementação real

Listagem 1. Exemplos de uso da aplicação através do cURL, supondo que a mesma esteja à porta 8080.

```
curl http://localhost:8080/openflights/iata/gru
curl http://localhost:8080/openflights/city/rio%20de%20janeiro
curl http://localhost:8080/openflights/country/brazil
```

Listagem 2. Executando o projeto.

```
$ cd /tmp
$ tar xzvf airportapp.tar.gz
$ cd airportapp
$ mvn spring-boot:run
```

Listagem 3. Listando os aeroportos cadastrados para o Rio de Janeiro.

```
$ curl http://localhost:8080/openflights/city/rio%20de%20janeiro
[{"id":2451,"name":"Campo Delio Jardim De Mattos","city":"Rio De Janeiro","country":"Brazil","iataCode":"","icaoCode":"SBAF","latitude":-22.875083,"longitude:-43.384708,"altitude":110.0,"offsetFromUTC":-3.0,"dstCode":"S","timezone":"America/Sao_Paulo"}, {"id":2492,"name":"Galeao Antonio Carlos Jobim","city":"Rio De Janeiro","country":"Brazil","iataCode":"GIG","icaoCode":"SBGL","latitude":-22.808903,"longitude":-43.243647,"altitude":28.0,"offsetFromUTC":-3.0,"dstCode":"S","timezone":"America/Sao_Paulo"}, {"id":2544,"name":"Santos Dumont","city":"Rio De Janeiro","country":"Brazil","iataCode":"SDU","icaoCode":"SBRJ","latitude":-22.910461,"longitude":-43.163133,"altitude":11.0,"offsetFromUTC":-3.0,"dstCode":"S","timezone":"America/Sao_Paulo"}, {"id":2546,"name":"Santa Cruz","city":"Rio De Janeiro","country":"Brazil","iataCode":"STU","icaoCode":"SBSC","latitude":-22.93235,"longitude":-43.719092,"altitude":10.0,"offsetFromUTC":-3.0,"dstCode":"S","timezone":"America/Sao_Paulo"}]
```

de **FlightService** e se utiliza, como se pode notar no código, do repositório (abstração de repositório de dados feita com o Spring Data) **AirportRepo** para efetivamente trabalhar com o banco de dados, buscando aeroportos por cidade, país ou código IATA. As **Figuras 4, 5 e 6** apresentam os diagramas de sequência descrevendo o fluxo de dados da aplicação.

A **Figura 7** mostra a estrutura do projeto e seus arquivos conforme apresentados no Eclipse ou STS. Se você os abrir, irá perceber que nosso (pouco) código está concentrado em apenas cinco classes: as três controladoras (**CitySearch**, **CountrySearch** e **IataCodeSearch**), uma classe de entidade JPA (**Airport**) e a implementação de **FlightService**, **OpenFlightService**. Perceba como, graças ao Spring (MVC e Data) e Spring Boot, estamos focados quase que exclusivamente em nosso negócio.

Iniciando o projeto

Como com qualquer projeto Maven, iniciamos por seu POM. O *pom.xml* do projeto que você descarregou é diferente do apresentado na **Listagem 4**. O original é a versão final, na qual iremos chegar com todas as customizações. O *pom.xml* da listagem não possui as seguintes opções, ou “opiniões”, que faremos prevalecer às default do Spring Boot:

- Iremos utilizar o Jetty 9 como Servlet Container – o default é o Tomcat;
- Iremos utilizar o HikariCP (vide **BOX 2**) como gerenciador do pool de conexões JDBC – o default é o do Tomcat.

BOX 2. HikariCP

O HikariCP é a implementação de um pool de conexões JDBC criado em 2013 que tem ganhado destaque ultimamente por sua performance, confiabilidade e tamanho reduzido – cerca de 90Kb apenas.

A análise desse POM irá nos mostrar muito do trabalho do Spring Boot. Comecemos observando a declaração do projeto pai, das linhas 10 a 14. Ao declararmos o **spring-boot-starter-parent** como pai, teremos à mão toda a gestão de dependências do Spring Boot, o que nos libera de declará-las e mantê-las manualmente. As dependências cujos nomes começam com **spring-boot-starter** são especiais, pois não se tratam de um artefato único, mas um conjunto de artefatos. Por exemplo, o **spring-boot-starter** traz consigo:

- **spring-boot.jar**;
- **spring-context.jar**.
- **spring-boot-autoconfigure.jar**;
- **spring-boot-starter-logging.jar**;
- **jcl-over-slf4j.jar**;
- **jul-to-slf4j.jar**;
- **log4j-over-slf4j.jar**;
- **logback-classic.jar**;
- **logback-core.jar**.

Uma forma de visualizar a árvore de dependências do projeto é utilizar o plugin **dependency** do próprio Maven. A partir disso,

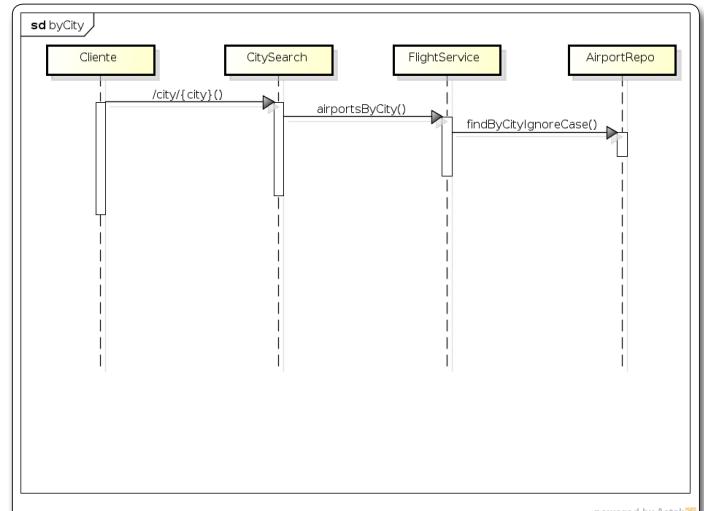


Figura 4. Fluxo da controladora CitySearch

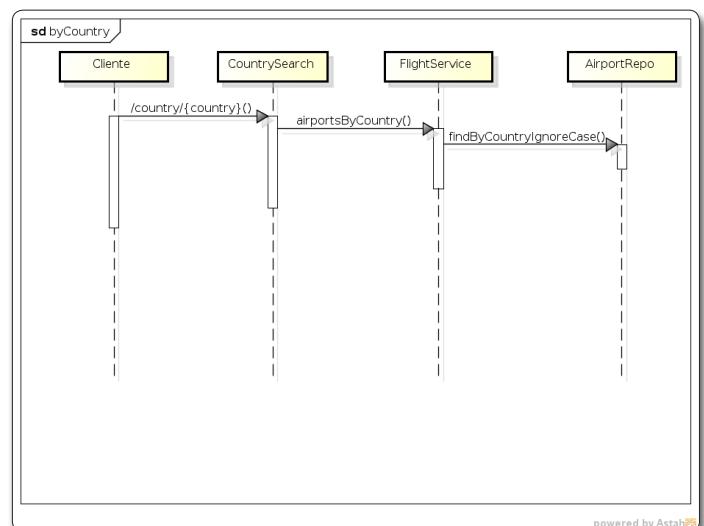


Figura 5. Fluxo da controladora CountrySearch.

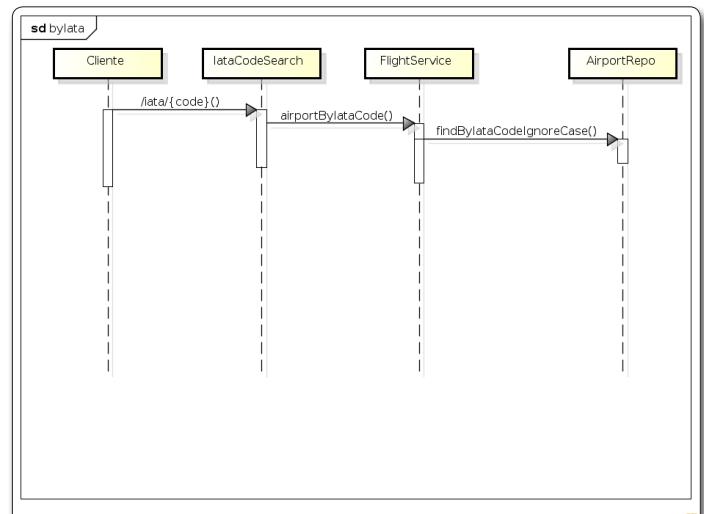


Figura 6. Fluxo da controladora IataCodeSearch

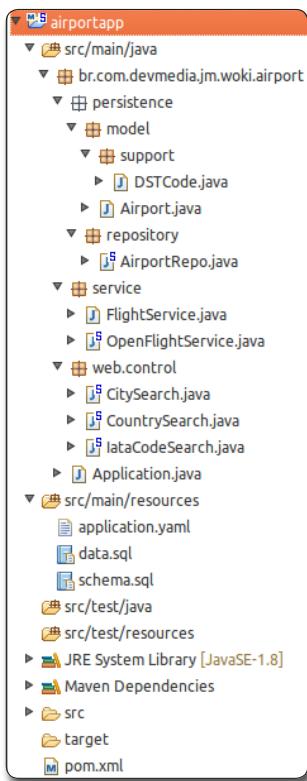


Figura 7. Projeto airportapp e seus arquivos no Eclipse

execute o comando `mvn dependency:tree` e você terá em uma árvore todas as dependências do projeto.

A dependência `h2` mostra outro aspecto do gerenciamento de dependências do Boot. Apesar de não estar sob nenhum **spring-boot-starter**, é uma de suas dependências gerenciadas e a prova disso é que não necessitamos declarar sua versão. Todas as dependências e suas versões – para cada versão do Spring Boot – estão, obviamente, documentadas em seu manual de referência. Sendo assim, antes de declarar qualquer uma, é bom consultar essa documentação. Não que não se possa declará-la independentemente, mas porque é muito melhor deixá-lo gerenciá-las.

Caso desejemos depender de uma versão diferente da declarada no Boot, basta adicionar a propriedade `${artifactId}.version` ao conjunto de propriedades do projeto. Por exemplo, a versão de H2 declarada no Spring Boot 1.2.5 é a 1.4.187. Supondo que haja a 1.4.188 e temos interesse em utilizá-la, basta adicionar a propriedade `<h2.version>1.4.188</h2.version>` e a mesma será acatada. Por falar em propriedades, observe na linha 18 a definição da versão Java que queremos adotar. Se fosse necessário o Java 1.7, bastaria alterar o valor de `java.version` para 1.7.

Além do gerenciamento de dependências, o Spring Boot também é um plugin, e como tal, disponibiliza três goals:

- **run** – executa a aplicação, seja ela web ou não;
- **repackage** – empacota a aplicação em um JAR executável;
- **help** – apresenta mensagens de ajuda de uso do plugin. Basicamente, descreve o uso dos dois goals anteriores.

Listagem 4. POM inicial do projeto airportapp.

```

01 <project xmlns="http://maven.apache.org/POM/4.0.0"
02   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
04   http://maven.apache.org/xsd/maven-4.0.0.xsd">
05   <modelVersion>4.0.0</modelVersion>
06   <groupId>br.com.devmedia.jm.woki</groupId>
07   <artifactId>airportapp</artifactId>
08   <version>0.0.1-SNAPSHOT</version>
09
10  <parent>
11    <groupId>org.springframework.boot</groupId>
12    <artifactId>spring-boot-starter-parent</artifactId>
13    <version>1.2.5.RELEASE</version>
14  </parent>
15  <properties>
16    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
17    <project.reporting.outputEncoding>UTF-8
18    </project.reporting.outputEncoding>
19  </properties>
20  <dependencies>
21    <!-- Core -->
22    <dependency>
23      <groupId>org.springframework.boot</groupId>
24      <artifactId>spring-boot-starter</artifactId>
25    </dependency>
26    <!-- Web application -->
27    <dependency>
28      <groupId>org.springframework.boot</groupId>
29      <artifactId>spring-boot-starter-web</artifactId>
30    </dependency>
31    <!-- Spring Data JPA -->
32    <dependency>
33      <groupId>org.springframework.boot</groupId>
34      <artifactId>spring-boot-starter-data-jpa</artifactId>
35    </dependency>
36    <!-- Embedded DB: H2 -->
37    <dependency>
38      <groupId>com.h2database</groupId>
39      <artifactId>h2</artifactId>
40      <scope>runtime</scope>
41    </dependency>
42  </dependencies>
43 </project>

```

Para executar a aplicação, basta invocar o Maven com `mvn spring-boot:run`. Observe que com esse comando estamos solicitando ao Maven que utilize o plugin `spring-boot` e invoque o goal `run` do mesmo.

Customizando o projeto

Para demonstrar como customizar o projeto, vamos “brincar” um pouco com o Spring Boot rejeitando duas de suas opiniões. Sendo assim, suponha que nosso projeto tenha como requisito não funcional o uso do Jetty como Servlet Container e do HikariCP como gerenciador de pool de conexões JDBC. Impor nossa opinião, neste caso, traduz-se em alterar o POM de nosso projeto.

Para utilizar o Jetty, por exemplo, devemos excluir o `spring-boot-starter-tomcat` do `spring-boot-starter-web`, pois essa é a opinião original, e adicionar a dependência `spring-boot-starter-jetty`, como

demonstra a **Listagem 5**, que traz o trecho do POM alterado.

Para fazer uso do HikariCP, o procedimento é parecido. Primeiro, excluímos o tomcat-jdbc como dependência de **spring-boot-starter-data-jpa**, e depois adicionamos a dependência do HikariCP, como mostra o trecho presente na **Listagem 6**.

Neste momento, ao executar `mvn spring-boot:run` você poderá observar pela saída da aplicação que o Servlet Container mudou para Jetty e que o gerenciador do pool de conexões JDBC passou a ser o HikariCP.

Listagem 5. Alterando opiniões do Spring Boot – de Tomcat para Jetty.

```
26 <!-- Web application -->
27 <dependency>
28   <groupId>org.springframework.boot</groupId>
29   <artifactId>spring-boot-starter-web</artifactId>
30   <exclusions>
31     <exclusion>
32       <artifactId>spring-boot-starter-tomcat</artifactId>
33     <groupId>org.springframework.boot</groupId>
34   </exclusion>
35 </exclusions>
36 </dependency>
37 <dependency>
38   <groupId>org.springframework.boot</groupId>
39   <artifactId>spring-boot-starter-jetty</artifactId>
40 </dependency>
```

Listagem 6. Alterando opiniões do Spring Boot – de Tomcat-JDBC para HikariCP.

```
41 <!-- Spring Data JPA -->
42 <dependency>
43   <groupId>org.springframework.boot</groupId>
44   <artifactId>spring-boot-starter-data-jpa</artifactId>
45   <exclusions>
46     <exclusion>
47       <artifactId>tomcat-jdbc</artifactId>
48     <groupId>org.apache.tomcat</groupId>
49   </exclusion>
50 </exclusions>
51 </dependency>
...
58 <dependency>
59   <groupId>com.zaxxer</groupId>
60   <artifactId>HikariCP</artifactId>
61   <scope>runtime</scope>
62 </dependency>
63 </dependencies>
```

Nota

Para utilizar o goal repackage do `spring-boot` é necessário acompanhá-lo do goal `install` do Maven, como no exemplo: `mvn clean install spring-boot:repackage`.

Listagem 7. Nossa classe de entrada para aplicações que fazem uso do Spring Boot.

```
01 @SpringBootApplication
02 public class Application {
03   public static void main(String... args) {
04     SpringApplication.run(Application.class, args);
05   }
06 }
```

Na linha 4 dessa listagem invocamos o método `run()` e informamos como argumentos nossa própria classe e os argumentos passados para a aplicação. Esse método aceita uma classe que seja fonte de configuração (contenha a anotação `@Configuration`) e parâmetros repassados da linha de comando. Essa é a variação mais recomendada de `SpringApplication.run()` para se inicializar uma aplicação com o Spring Boot. Há outras, mas essa opção induz à utilização de uma única classe de entrada de configuração (que pode, obviamente, carregar outras em cascata), tornando o código mais legível e facilitando a manutenção. “Um momento! Isso significa que nossa classe é o ponto de entrada de configuração? Mas ela não está anotada com `@Configuration` nem está configurando nada! ” – pode-se argumentar. Acontece que ela é e está configurando tudo em nossa aplicação. A anotação `@SpringBootApplication` é uma anotação de conveniência que contém as seguintes anotações do Spring: `@Configuration`, `@EnableAutoConfiguration` e `@ComponentScan`. Essas duas últimas, basicamente, dizem ao inicializador do Spring: “Busque e instancie todo bean anotado deste pacote para frente”. Com “bean anotado” estamos nos referindo a classes anotadas com `@Configuration` e métodos que retornam `@Bean`, e classes anotadas com `@Service`, `@Component`, `@Controller` e `@RestController`, por exemplo. Esse processo de autoconfiguração e busca de componentes, como se sabe, não é especificamente do Boot, e sim do próprio Spring Framework.

Voltando à nossa aplicação, nossa classe `Application` não configura nada, mas dispara a busca por beans a partir de seu pacote. Assim, serão encontradas e devidamente inicializadas as classes e interfaces:

- `br.com.devmedia.jm.woki.airport.persistence.model.Airport` – entidade JPA (Spring Data/JPA);
- `br.com.devmedia.jm.woki.airport.persistence.repository.AirportRepo` – repositório (Spring Data/JPA);
- `br.com.devmedia.jm.woki.airport.service.OpenFlightService` – implementação da interface FlightService (Spring Core);
- `br.com.devmedia.jm.woki.airport.web.control.*` – controladoras, mais especificamente, controladoras dos nossos serviços REST (Spring MVC).

Entendendo o código

Até aqui, vimos como criar e customizar o POM de um projeto com o Spring Boot, e vimos também como executá-lo em tempo de desenvolvimento com o goal `run`. De agora em diante, vamos entender o código que traz a aplicação airportapp à vida. Começemos pela classe de entrada, ou seja, aquela a partir da qual o Spring Boot irá inicializar todos os beans necessários. No caso do nosso exemplo, a classe `br.com.devmedia.jm.woki.airport.Application`, apresentada na **Listagem 7**.

O Boot sabe que se trata da classe de entrada porque ela possui o método `main()` e pela invocação de `SpringApplication.run()`. O nome da classe ou seu pacote não importam.

Deste modo serão inicializados todos os componentes de nossa aplicação. Como podemos notar, trata-se de um processo de bootstrapping normal do Spring, mas aqui orquestrado pelo Spring Boot. Note também que não há qualquer anotação ou dependência do Boot em nenhum desses componentes. São artefatos usuais do Spring MVC e Spring Data/JPA, como mostram de maneira reduzida a **Listagem 8**, com uma das controladoras, a **Listagem 9**, com a entidade **Airport**, a **Listagem 10**, com o repositório **AirportRepo**, e a **Listagem 11**, com a implementação de **FlightService**, **OpenFlightService**.

Propriedades da aplicação

As propriedades da aplicação não ficam em seu POM nem em código, mas em um arquivo próprio. É possível parametrizar muitos aspectos e comportamentos da aplicação através desse arquivo. O nosso, entretanto, configura apenas os seguintes:

- Facilidade de logging da aplicação – apenas especificamos o nível de logging, aceitando o default de uma aplicação Spring Boot, ou seja, o uso do logback tendo como saída o console;
- Porta TCP em que o Jetty irá atender as requisições – apesar de a default ser 8080 e não necessitarmos especificá-la caso seja essa a intenção, de forma ilustrativa a definimos em nosso arquivo de propriedades como 8080;
- Contexto raiz da aplicação – configuramos `/openflights` para o nosso caso. Na ausência dessa configuração, o contexto raiz será simplesmente `/`;
- Criação e carga inicial de dados do banco de dados em memória – nosso arquivo de propriedades dá um nome à plataforma de banco de dados em uso. Esse nome é utilizado pelo Spring JDBC para a criação e carga da base, conforme veremos em breve.

O nome desse arquivo é, por default, `application.properties` ou `application.yaml` – sim, o Spring Boot também entende configurações no formato YAML (veja o **BOX 3**).

Caso nenhum `application.properties/yaml` seja encontrado, o Boot assumirá valores default sempre que possível. Nossa aplicação, no entanto, possui o arquivo `application.yaml`, conforme a **Listagem 12**.

Note que entre as linhas 1 e 4 estamos configurando nossos níveis de logging. Já as linhas de 5 a 7 configuram a porta e o contexto raiz da aplicação no Servlet Container. As linhas 8 a 13, por sua vez, configuram a geração de DDL pelo Hibernate e a plataforma, ou tipo de banco de dados, de nosso Datasource. Em nosso caso não queremos que o Hibernate execute qualquer DDL a partir de nossas entidades e indicamos que usaremos o H2. Iremos explicar melhor essa última configuração em breve. Antes é preciso notar que todas as configurações da **Listagem 12** possuem significado especial para o Boot e são usadas para parametrizar componentes por ele gerenciados. Além das listadas, há várias outras que podem ser verificadas no documento de referência do Spring Boot, assim como também podemos ter nossas próprias configurações aqui, desde que não conflitem

Listagem 8. Código do controller CitySearch.

```
01 @RestController
02 @RequestMapping(value="/city")
03 public class CitySearch {
04     private static final Logger logger = LoggerFactory.getLogger(CitySearch.class);
05
06     @Autowired
07     private FlightService flightService;
08
09     @RequestMapping(value="/{cityName}", method=RequestMethod.GET)
10    public List<Airport> search(@PathVariable String cityName) {
11        List<Airport> retval = flightService.airportsByCity(cityName);
12        if(retval != null) {
13            logger.info("Aeroportos para a cidade {}: {}", cityName, retval.size());
14        } else {
15            logger.info("Aeroportos para a cidade {}: 0", cityName);
16        }
17        return retval;
18    }
19}
```

Listagem 9. Código da entidade JPA Airport.

```
01 @Entity(name="airport")
02 @SuppressWarnings("serial")
03 public class Airport implements Serializable {
04     @Id @GeneratedValue
05     private long id;
06     private String name;
07     private String city;
08     private String country;
09     @Column(name="iatacode")
10    private String iataCode;
11    @Column(name="icaoode")
12    private String icaoCode;
13    private double latitude;
14    private double longitude;
15    private double altitude;
16    @Column(name="offsetutc")
17    private double offsetFromUTC;
18    @Column(name="dstcode")
19    @Enumerated(EnumType.STRING)
20    private DSTCode dstCode;
21    private String timezone;
```

Listagem 10. Código do repositório Spring Data/JPA AirportRepo.

```
01 public interface AirportRepo extends JpaRepository<Airport, Long> {
02     List<Airport> findByCityIgnoreCase(String city);
03     List<Airport> findByCountryIgnoreCase(String country);
04     Airport findByIataCodeIgnoreCase(String iataCode);
05 }
```

Listagem 11. Código da classe de serviço OpenFlightService.

```
01 @Service
02 public class OpenFlightService implements FlightService {
03     @Autowired
04     private AirportRepo airportRepository;
05
06     @Override
07     public List<Airport> airportsByCity(String city) {
08         return airportRepository.findByCityIgnoreCase(city);
09     }
10
11     @Override
12     public List<Airport> airportsByCountry(String country) {
13         return airportRepository.findByCountryIgnoreCase(country);
14     }
15
16     @Override
17     public Airport airportByIataCode(String iataCode) {
18         return airportRepository.findByIataCodeIgnoreCase(iataCode);
19     }
20 }
```

em nomes com as reservadas. Por exemplo, supondo que o projeto airportapp precisasse de uma propriedade contendo, digamos, o caminho de dado arquivo, poderíamos utilizar algo como:

```
airportapp:  
  data-file: /tmp/data.dat
```

Listagem 12. Configurações em application.yaml.

```
01 logging:  
02   level:  
03     org.springframeworkframework: INFO  
04     br.com.devmedia.jm.woki.airport: INFO  
05 server:  
06   port: 8080  
07   contextPath: /openflights  
08 spring:  
09   jpa:  
10     hibernate:  
11       ddl-auto: none  
12   datasource:  
13     platform: h2
```

BOX 3. YAML

O YAML é uma linguagem para serialização de dados derivado do JSON. Seus objetivos principais são expressar informação de forma humanamente legível, portabilidade entre linguagens de programação e suporte a estruturas de dados nativas em linguagens modernas como o C/C++, Java, Ruby, Perl, Python, JavaScript, entre outras.

Na prática, o que estaria em um arquivo de propriedades como:

```
application.name=MyApplication  
application.version=1.0.0
```

pode ser expresso dessa forma em YAML:

```
application:  
  name: MyApplication  
  version: 1.0.
```

Esse mecanismo de externalização de configuração não é específico do Spring Boot, e já existe, inclusive, no próprio Spring Framework. O Boot adicionou, como diferencial, uma série de propriedades reservadas para simplificar a configuração da aplicação, conforme vimos.

Retornando à configuração de geração de DDL e especificação de banco de dados, entre as linhas 8 e 13, percebemos que a criação do banco de dados – geração de DDL – pelo Hibernate está desabilitada. O motivo disso é que em nossa aplicação, de maneira ilustrativa, usamos outra forma para a criação de nosso banco e também sua carga de dados. Queremos criá-lo e populá-lo de maneira explícita através de arquivos SQL. Para isso, o projeto contém dois arquivos SQL:

- *src/main/resources/schema-h2.sql* – contém o DDL para a criação da tabela airport;

- *src/main/resources/data-h2.sql* – contém o DML para a carga da tabela airport com dados.

A **Listagem 13** mostra o conteúdo do arquivo *schema-h2.sql* e a **Listagem 14**, de forma reduzida, o de *data-h2.sql*.

E como o schema está sendo criado e os dados populados? Através do Spring JDBC, que possui uma facilidade de inicialização de Datasource e o Spring Boot a habilita por default. Deste modo, sempre que houver os arquivos *schema.sql* e/ou *data.sql* na raiz do classpath, eles serão executados pelo Boot através do Spring JDBC.

Listagem 13. DDL de nosso banco de dados.

```
01 create table airport (  
02   id integer not null generated always as identity (start with 1, increment by 1),  
03   name varchar(256) not null,  
04   city varchar(256),  
05   country varchar(256) not null,  
06   iatacode varchar(3),  
07   icaocode varchar(4),  
08   latitude double,  
09   longitude double,  
10   altitude double,  
11   offsetutc integer,  
12   dstcode char(1),  
13   timezone varchar(128)  
14 );  
15  
16 create index iata_unq on airport(iatacode);  
17 create index city_idx on airport(city);  
18 create index country_idx on airport(country);
```

Listagem 14. Dados para popular nossa tabela airport.

```
01 insert into airport (name,city,country,iatacode,icaocode,latitude,  
longitude,altitude,offsetutc,dstcode,timezone) values ('Goroka','Goroka',  
Papua New Guinea,'GKA','AYGA',-6.081689,145.391881,5282,10,'U',  
Pacific/Port_Moresby);  
02 insert into airport (name,city,country,iatacode,icaocode,latitude,  
longitude,altitude,offsetutc,dstcode,timezone) values ('Madang','Madang',  
Papua New Guinea,'MAG','AYMD',-5.207083,145.7887,20,10,'U',Pacific/  
Port_Moresby);  
...
```

Adicionalmente, o Boot irá carregar os arquivos *schema-\${platform}.sql* e *data-\${platform}.sql*, se presentes, onde \${platform} corresponde ao conteúdo da propriedade:

```
spring.datasource.platform
```

ou, em YAML:

```
spring:  
  datasource:  
    platform
```

Note que em nossa aplicação não fazemos uso de *schema.sql* nem *data.sql*. O intuito disso é mostrar como se utilizar da facilidade adicional do Spring Boot/Spring JDBC para carregar

arquivos específicos por plataforma. Note que utilizamos em nossas propriedades o nome **h2** para a plataforma. Entretanto, poderia ser qualquer outro nome válido, como h2db, desde que os arquivos de DDL e DML estivessem nomeados como *schema-h2db.sql* e *data-h2db.sql*.

Vale ressaltar ainda que a localização desses arquivos pode mudar configurando-se as propriedades **spring.datasource.schema** e **spring.datasource.data**. Também podemos desabilitar essa facilidade com **spring.datasource.initialize=false**. Outra configuração possível é **spring.datasource.continueOnError=true**, mas não é aconselhável, pois o comportamento default é não permitir que a aplicação inicie sem a carga de dados. É importante notar que essa facilidade na criação do schema conflita com a similar do JPA/Hibernate, **ddl-auto**, que por default é **create-drop**. Por isso é preciso configurar a propriedade **spring.jpa.hibernate.ddl-auto** como **false** para evitar esse conflito. Por outro lado, como a carga de dados é uma facilidade específica do Spring JDBC, podemos ter um cenário em que executamos o DDL pelo Hibernate e carregamos os dados pelo Spring JDBC – o que não é o caso de nossa aplicação.

No decorrer do artigo vimos que o Spring Boot é o componente da plataforma, ou ecossistema, Spring IO que objetiva simplificar, consolidar e reenergizar o Spring. A simplificação deriva da visão opinativa sobre dependências e autoconfiguração. Já a consolidação refere-se à agregação coordenada de componentes, dependências e boas práticas sob os diversos artefatos ***-starter-***. O cumprimento desses dois objetivos pode ser averiguado quando comparamos o tamanho e simplicidade do POM de nosso projeto de exemplo com um que descreva as mesmas dependências sem o uso do Spring Boot. A diferença é da grandeza de algumas dezenas de linhas para projetos simples e pequenos como esse! Mais do que essa diferença, no entanto, o que conta mesmo é a coesão e clareza obtidas, adjetivos que têm grande impacto no tempo de desenvolvimento e manutenção do software. Outro impacto, também indireto, está na qualidade do software. Não se fala aqui exatamente de QA, testes, mas da capacidade do desenvolvedor atingir o máximo possível na codificação do negócio, e não da infraestrutura. Quanto maior essa capacidade, maior a qualidade inerente do trabalho final. Por falar em testes, nosso projeto não possui nenhum e isso de forma alguma é recomendável, mas o objetivo aqui foi de focar o uso básico e customizações. O Spring Boot também possui um ***-starter-*** específico para testes, o **spring-boot-starter-test**, que já traz os artefatos de teste do Spring Test, alguns específicos do Boot que ficam no próprio **spring-boot**, assim como dependências como o JUnit, Hamcrest e Mockito.

O último objetivo abordado, reenergização, é decorrência natural do alcance dos dois primeiros. Há alguns anos era divertido desenvolver com o Spring Framework, e também havia muita sensação de alívio para os que saíram do Java EE tradicional para um projeto com tal tecnologia. Com o decorrer dos anos o número de componentes cresceu, os que já existiam evoluíram e, na prática, o setup de um projeto com o Spring se tornou oneroso e per-

deu muito da sensação de menos trabalho, menos configurações. O Spring Boot, conforme vimos inclusive com o nosso projeto exemplo, responde justamente a esse desconforto, e de uma maneira elegante, simples e não intrusiva.

O nosso projeto exemplo ilustrou como iniciar e customizar alguns aspectos do Spring Boot, mostrando que, apesar de sua visão opinativa, ele não apresenta obstáculos quando resolvemos sobrepor uma ‘opinião’ ou outra.

Apesar de muitas facilidades interessantes, como monitoramento, uso de profiles, integração com bancos de dados SQL ou No-SQL, integração com outros projetos do Spring IO, como o MVC, Security, Integration, Batch e outras não terem sido abordadas aqui pelo escopo do artigo, vale a pena dar uma olhada no documento de referência do Spring Boot e explorá-las!

Por último, é importante dizer que na vida real são muito raras as oportunidades de migrar aplicações legadas para o Spring, mas se você se deparar com uma, não tenha dúvida: utilize o Spring Boot! A mesma diretriz vale a novas aplicações onde se decidiu utilizar o Spring. Portanto, não perca tempo e adicione riscos a seu software gerenciando dezenas, centenas de dependências e arquivos de configuração manualmente, quando você tem à disposição uma ferramenta como o Boot.

Autor



Willian Oki

willian.oki@gmail.com

Trabalha com desenvolvimento de software desde 1997. Já desenvolveu em C/C++ e Perl, mas especializou-se em arquitetura e desenvolvimento Java EE desde 2005. Atuou principalmente no setor financeiro e de telecomunicações e atualmente é engenheiro de software na Adyen Latin America.



Links:

Página do projeto Spring IO.

<http://spring.io>

Documento de referência do Spring Boot.

<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle>

Documentação oficial do Java EE, Oracle.

<http://docs.oracle.com/javaee>

Página do projeto Jetty.

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

Página do projeto HikariCP.

<http://brettwooldridge.github.io/HikariCP>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Automatizando a geração de pacotes com o Jenkins

Aprenda neste artigo a gerar pacotes em poucos cliques com o Jenkins utilizando as principais soluções de build, como Ant, Maven e Gradle

Dependendo da empresa em que você trabalha, o processo de geração de pacotes de uma aplicação pode ir do mais simples, como a execução de um único comando de build, até o mais complexo, envolvendo diversas equipes e muitos processos de controle. Com o passar do tempo, muitos perceberam a importância dessa fase do projeto e hoje temos várias ferramentas de build que facilitam a geração de pacotes, assim como o próprio deploy. No Java, as ferramentas mais famosas são o Ant, Maven e Gradle, cada uma com suas vantagens e desvantagens. O Maven e o Gradle são, hoje em dia, os mais utilizados, visto que além de gerenciar o projeto, também controlam as dependências. Mas independentemente de qual das três será utilizada, o Jenkins consegue se comunicar com ela e orquestrar todos os passos necessários para criar o pacote de uma aplicação.

O Jenkins C.I. é uma aplicação web de Integração Contínua que pode ser instalada em qualquer máquina e serve, principalmente, para executar os testes e criar os artefatos de um projeto de software. Ela nasceu a partir de outra solução bem conhecida, o Hudson, que começou como uma ferramenta open source, mas, em 2010, a Oracle passou a ser a responsável pelo seu desenvolvimento. Neste momento, alguns programadores da equipe aceitaram ir para a Oracle continuar com o projeto, porém, outros, em 2011, decidiram seguir por outro caminho e evoluir a ferramenta de forma open source, adotando o nome Jenkins. Hoje, ambas são bastante utilizadas no mercado e fáceis de usar, pois com

Fique por dentro

Este artigo analisa e discute as vantagens de utilizar um dos servidores de integração contínua mais famosos para automatizar a geração e o deploy dos pacotes de um sistema, o Jenkins. Com exemplos práticos, aprenda a identificar o melhor modelo de automatização para o seu projeto, além de criar e configurar jobs no Jenkins utilizando o Ant, Maven ou o Gradle. Deste modo será possível reduzir consideravelmente o trabalho repetitivo tão comum em todos os projetos, possibilitando ao desenvolvedor manter a atenção na implementação das regras de negócio e nos testes.

apenas alguns cliques você, seu chefe, um analista de teste ou de implantação pode criar um pacote do sistema sem a necessidade de executar comandos no console ou abrir uma IDE.

A Integração Contínua representa um conjunto de práticas que permitem que um sistema seja compilado, testado e construído em um ambiente separado e independente do de desenvolvimento. Ela permite que as mudanças que acontecem no código sejam integradas ao sistema mais rapidamente e de forma contínua, pois possui ferramentas, como o Jenkins e o Hudson, que automatizam as tarefas de todo esse processo.

As vantagens de não gerar os pacotes do projeto manualmente são várias, principalmente quando as equipes envolvidas no sistema são grandes e o ciclo de entregas é curto. No entanto, mesmo quando a equipe é pequena, a automatização ainda pode executar tarefas repetitivas e que tomariam tempo do desenvolvedor, permitindo que ele foque no que realmente importa.

O Jenkins, por exemplo, possibilita que você gere pacotes a cada intervalo predeterminado de tempo, a cada commit no código ou simplesmente quando desejar, ou seja, você define a maneira que melhor se adequa na sua empresa.

Mas como nem tudo é perfeito, automatizar também tem suas desvantagens. Quando você automatiza processos, algumas pessoas da sua equipe e da sua empresa podem não ficar felizes, justamente por pensarem que estão “tirando” o trabalho delas. Isso acontece com frequência em empresas que já têm um fluxo de deploy estabelecido onde, por exemplo, os desenvolvedores enviam o pacote para outra equipe, responsável por implantá-los nos servidores. Provavelmente, obstáculos surgirão para que a automatização não aconteça, entretanto, é importante ter consciência de que esse esforço valerá a pena e que ninguém precisa ser mandado embora por causa do Jenkins. A partir dela as equipes de implantação poderão se dedicar mais à manutenção dos servidores e escalabilidade do sistema, planejando maneiras mais eficientes e rápidas de fazer o deploy da aplicação.

Outro fator que atrapalha a automatização desse processo é a falta de confiança da empresa em seus desenvolvedores. Algumas não dão acesso aos programadores para instalarem aplicações nos ambientes de teste, muitas vezes desmotivando qualquer iniciativa que busque facilitar o processo. Felizmente, todos esses problemas podem ser resolvidos com uma boa análise das necessidades do sistema, dos ambientes e dos usuários que, no fim, só querem ver as funcionalidades que eles pediram o mais rápido possível no ar.

A partir do que foi comentado, vamos abordar neste artigo a situação de uma empresa que deseja aprimorar seu processo de geração de pacotes, analisando para isso como ela trabalha atualmente e que opções temos disponíveis para alcançar a automatização desta etapa.

Analisando um processo manual de geração de pacotes

Suponha que você trabalha numa empresa que tenha um sistema web em Java de compra coletiva. A empresa tem equipes de desenvolvimento, QA, implantação e de negócios, sendo esta última a que conversa com os clientes e cria os requisitos do sistema. Vocês utilizam Scrum, com sprints de três semanas, e SCM Subversion 1.7 como repositório do código, onde o trunk contém o código que está em desenvolvimento e, para cada entrega de versão, uma branch é criada. Lá também tem três ambientes com o sistema instalado: dev, QA e produção, cada um com seu banco de dados MySQL e o servidor Tomcat. Além disso, scripts de banco são criados apenas para inserir configurações, já que o sistema utiliza o Hibernate para gerenciar as criações e atualizações de tabelas.

SCM é o acrônimo para *Source Code Management* e consiste na gestão do armazenamento e das alterações realizadas em um software. Para a gestão de alterações no código, existem ferramentas de controle de versão como o Subversion (SVN), CVS, Git, entre outros, que automatizam o armazenamento, recuperação, log e merge das mudanças.

Voltando ao exemplo, todo o processo de geração e deploy do pacote do sistema é feito de forma manual, dependendo completamente das pessoas da equipe. Neste cenário foi definido que, no fim da Sprint, determinado desenvolvedor deve gerar o pacote a partir do trunk em sua própria máquina e colocá-lo em um servidor FTP para que um analista de implantação possa instalá-lo no ambiente de QA e, caso houvesse script de banco, ele também possa executá-lo. Já a instalação no ambiente de dev é de responsabilidade da equipe de desenvolvimento, que normalmente gera e instala o pacote poucos dias antes de acabar a sprint, para validar e fazer alguns testes integrados antes de ir para QA. Depois de uma semana de testes no ambiente de QA e de tudo funcionando, os analistas de QA solicitam à equipe de implantação que instale o mesmo pacote no ambiente de produção e o processo termina.

Talvez você esteja pensando que o processo definido pela empresa está muito bom, e que atende perfeitamente às necessidades de todos os envolvidos nos projetos. Infelizmente, o que foi descrito é o fluxo perfeito e que com o tempo acabará se tornando raro. Normalmente, quando o pacote vai para o QA, bugs são encontrados e novos pacotes com correções devem ser gerados. Assim, o deploy pode demorar ainda mais, visto que primeiro o desenvolvedor precisa enviar um e-mail solicitando à equipe de implantação uma nova instalação em QA. Depois esta precisa avisar o horário de deploy para a equipe de QA, para evitar que alguém esteja testando algo no momento da instalação, e aí sim, finalmente, instala-se o novo pacote no servidor. Ressalta-se ainda que durante a fase de testes essa situação se repetirá diversas vezes, sempre tomando tempo de várias pessoas, que neste cenário estão envolvidas em todas as partes do processo.

Outro problema que podemos perceber é que, como o processo de instalação é trabalhoso, os próprios desenvolvedores acabam utilizando menos o ambiente de dev (também chamado de ambiente de integração), já que alguém terá de parar para gerar o pacote e fazer o deploy. Diante disso, a equipe de QA sofre com a espera de todo o fluxo de deploy quando precisa retestar a correção de um bug. Já os implantadores ficam recebendo várias requisições de instalação de pacotes em QA e ainda precisam gerenciar o melhor horário para fazê-las. Isso sem contar que eles ainda têm o ambiente de produção para cuidar.

Agora, será que precisa ser assim mesmo? Por que não gastar um tempo conversando com as equipes para definir um processo mais eficiente e menos manual de entrega de pacotes?

Definindo um processo automatizado de entrega dos pacotes

A partir desse cenário, em uma reunião no fim da sprint, alguém surge com a ideia de automatizar o processo de geração e deploy dos pacotes da empresa. Todos gostam e ficam animados, mas por onde começar?

Primeiro, deve-se analisar como a equipe de desenvolvimento tem gerenciado o versionamento do código. No nosso caso, uma sugestão para a equipe de dev seria manter o trunk como o código mais atual do sistema, onde as estórias das sprints são

desenvolvidas, nomeando essa versão com um número seguido pela palavra "SNAPSHOT", por exemplo, "1.0.0-SNAPSHOT". Esse padrão de nomenclatura ficou bem famoso com o uso do Maven, mas se você adota o Ant ou qualquer outra ferramenta de build, também pode utilizá-lo. Quando o que foi desenvolvido no trunk for entregue, uma nova branch do trunk deve ser criada, com o nome indicando a versão final, por exemplo, "1.0.0-RELEASE" ou simplesmente "1.0.0".

Os pacotes para QA e produção sempre serão gerados a partir dessa branch, até que outra branch com uma nova versão seja criada, como "1.1.0", "1.2.0" e assim por diante. Já os pacotes que vão para o ambiente de dev são gerados com base no código que está no trunk e os pacotes que corrigem bugs da versão de QA e produção são gerados com a branch correspondente, sendo que é interessante sempre incrementar o número da versão conforme os pacotes forem gerados, por exemplo, "1.0.1", "1.0.2", etc. Ademais, deve-se lembrar que as correções feitas em qualquer branch também precisam ser feitas no trunk.

Um ponto importante é que a equipe de QA deve estar atenta à versão que está testando, sendo necessário informá-la no momento em que for abrir um bug. Por fim, a equipe de dev precisa informar no bug o número da versão do código em que ele será corrigido.

Definido isto, já temos o que será feito, mas ainda não temos o como. Felizmente, esta parte é a mais simples, pois todo esse controle pode ser feito no Jenkins, que será o responsável por gerar, enviar e fazer o deploy dos pacotes nos servidores. O Jenkins, basicamente, executa tarefas que são configuradas em seus Jobs, e cada job, buscando simplicidade e facilidade de configuração, pode se integrar a outras ferramentas, como Ant, Maven ou Gradle, o que explicita uma tão importante característica dos ambientes de CI, a flexibilidade.

Definindo o "como"

Como já mencionado, na nossa empresa temos os ambientes de dev, QA e produção, cada um com regras e propósitos diferentes. Utilizando o Jenkins, podemos criar uma maneira mais genérica e fácil de trabalhar com todos eles.

Para otimizar ao máximo o uso do ambiente de dev pelos desenvolvedores, podemos definir que a cada commit no trunk o Jenkins irá gerar um pacote, sempre executando antes todos os testes e, logo em seguida, já efetuando o deploy no servidor de dev. Dessa forma, teremos o ambiente de dev atualizado – com a versão mais recente do trunk – e pronto para qualquer teste integrado que venha a ser executado, por exemplo.

Quando o ciclo de desenvolvimento acabar e uma nova release precisar ser gerada, o Jenkins também se encarregará dessa atividade, bastando ao desenvolvedor informar o nome da nova versão que será gerada a partir do trunk e clicar em um botão. Já no fluxo de correção de bugs, que ocorre em cima do código da branch, a geração de uma nova versão corrigida (também chamada de patch) também passa a ser tarefa do Jenkins, sendo necessário apenas informar o nome da versão com a correção.

No ambiente de QA, a definição de qual versão será instalada fica completamente a cargo dos próprios analistas. No próprio Jenkins, eles selecionariam qual versão desejam que seja instalada e pronto. Em alguns minutos um novo deploy é feito no ambiente de QA, sem precisar mandar e-mails e nem esperar por ninguém.

Por último, mas não menos importante, o ambiente de produção continua sob responsabilidade da equipe de implantação, mas que agora deixa de baixar os pacotes de servidores FTP. Agora, basta acessar o Jenkins, selecionar a versão desejada, informar usuário e senha do Tomcat, para segurança dos deploys, e clicar em um botão.

Bem mais simples, não? Agora vamos ver como configurar o Jenkins para executar todas as tarefas que discutimos em cada um dos ambientes nas fases de desenvolvimento, QA e entrega em produção.

Configurando o projeto

Primeiramente, é preciso identificar qual ferramenta de build sua aplicação está utilizando. Para abranger as mais famosas, mostraremos um mesmo exemplo de sistema de compras coletivas em três projetos, cada um com uma ferramenta de build/gestão de dependências diferente (Ant, Maven e Gradle).

Além disso, em cada projeto, alguns plugins ou comandos foram incluídos nos arquivos de configuração das ferramentas, já que só os mais comuns são definidos por padrão. Isso é necessário porque os três projetos precisam ter configuradas todas as tarefas que utilizaremos no Jenkins funcionando sem problemas, pois este simplesmente orquestrará a execução das tarefas desses plugins. Vejamos como ficaria cada um dos projetos.

Ajustando o projeto com o Ant

Um dos projetos do sistema de compra coletiva será configurado para adotar o Ant como ferramenta de build. Considerando que temos todo o código da aplicação pronto, para manter o foco no escopo do artigo, o primeiro passo é criar o arquivo *build.xml*, como pode ser visto nas [Listagens 1, 2 e 3](#). Por estar no formato XML, ele pode ficar um pouco verboso, porém mais fácil de ler. Dividimos o arquivo em três listagens devido a sua extensão e para que possamos explicar com mais detalhes cada parte do mesmo.

Neste arquivo estão definidas as duas principais tarefas que o Jenkins executará para gerar e instalar os pacotes: *release* e *deploy*. Essas tarefas receberão parâmetros como o nome do usuário e senha do Tomcat, mas o mais importante é o nome do ambiente em que o pacote deverá ser instalado. Esse parâmetro terá o nome de *profile*, que é um nome bastante utilizado para essa finalidade.

Na [Listagem 1](#), entre as linhas 18 e 25, são definidos os tipos de tarefas que não vêm por padrão no Ant: as tarefas que trabalham com o SVN (*svnantlib*), as utilitárias (*antlib*) e as tarefas de *deploy* e *undeploy* do Tomcat. Para que tudo funcione corretamente, esses tipos precisam que algumas bibliotecas estejam no classpath do Ant. Portanto, você precisará baixá-

Listagem 1. Configurando as propriedades do projeto no build.xml – Parte 1.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <!DOCTYPE xml>
03 <project name="compra" default="generate-war" basedir="">
04
05 <property name="project.name" value="compra"/>
06 <property name="build.directory" value="build-ant"/>
07 <property name="source.root" value="https://compra.com.br/repos/compra"/>
08 <property name="tomcat.manager.url"
09   value="http://localhost:8090/manager/text"/>
10 <property name="tomcat.username" value="tomcat"/>
11 <property name="tomcat.password" value="tomcat"/>
12 <loadproperties srcFile="${basedir}/build.properties"/>
13 <path id="ant.classpath">
14   <fileset dir="${basedir}/ant-jars">
15     <include name="**/*.jar"/>
16   </fileset>
17 </path>
18 <typedef resource="org/tigris/subversion/svnant/svnantlib.xml"
19   classpathref="ant.classpath"/>
20 <svnSetting id="svn.settings" username="${svn.username}"
21   password="${svn.password}">
22   javahl="false" svnkite="true" failonerror="true"/>
23 <taskdef resource="net/sf/ant/contrib/antlib.xml" classpathref="ant.classpath"/>
24 <taskdef name="deploy" classname="org.apache.catalina.ant.DeployTask"
25   classpathref="ant.classpath"/>
```

las da internet e colocá-las na pasta *ant-jar* do seu projeto. As bibliotecas são arquivos JAR que são dependências dos tipos de tarefas que utilizaremos no Ant para gerar e fazer o deploy de pacotes. A **Figura 1** mostra a estrutura do projeto Ant e as bibliotecas que devem ficar na pasta *ant-jars*.

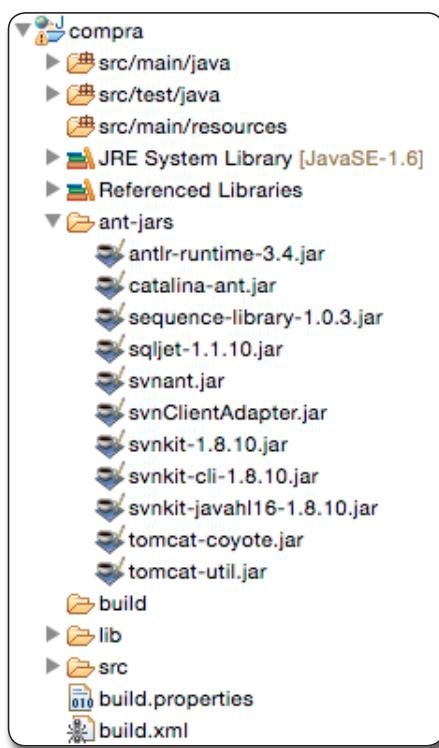


Figura 1. Projeto com o Ant configurado

```
26   <taskdef name="undeploy" classname="org.apache.catalina.ant.UndeployTask"
27     classpathref="ant.classpath"/>
28   <if>
29     <equals arg1="${profile}" arg2="dev"/>
30     <then>
31       <property name="tomcat.manager.url"
32         value="http://dev.compra.com.br:8090/manager/text"/>
33     </then>
34   </if>
35   <if>
36     <equals arg1="${profile}" arg2="qa"/>
37     <then>
38       <property name="tomcat.manager.url"
39         value="http://qa.compra.com.br:8090/manager/text"/>
40     </then>
41   </if>
42   <if>
43     <equals arg1="${profile}" arg2="prod"/>
44     <then>
45       <property name="tomcat.manager.url"
46         value="http://compra.com.br:8090/manager/text"/>
47     </then>
48   </if>
```

Ainda neste arquivo, entre as linhas 27 e 44 está a lógica para definir a URL do manager do Tomcat (que é a interface gráfica para gerenciar os pacotes instalados na aplicação) em que será feito o deploy. Note que três comandos de condição são utilizados para verificar qual URL utilizar dependendo do nome do ambiente, que virá no parâmetro **profile**.

Já na **Listagem 2**, o target (ou alvo) *generate-war*, iniciado na linha 46, é composto por um conjunto de tarefas que têm o objetivo de compilar as classes Java e criar o arquivo do pacote, o arquivo WAR. A tag *<path>*, na linha 67, define uma identificação para as pastas onde estão as bibliotecas do projeto. Assim o Java pode referenciá-la com o parâmetro *classpathref* no comando de compilação dos arquivos, na linha 76. Na linha 81, por sua vez, temos a task *war*, que cria o arquivo do pacote propriamente dito, um arquivo *zip*, contendo os arquivos do projeto, mas com a extensão WAR, o padrão para as aplicações web em Java. Em seguida, nas linhas 92 e 97, os targets *undeploy* e *deploy*, chamam, respectivamente, as tarefas de remoção e instalação de um pacote no Tomcat, que recebem como parâmetros os valores das propriedades definidas na **Listagem 1**.

Antes da tarefa de *deploy* (linha 97) ser executada, o Ant chama os targets *generate-war* e *undeploy*. Voltando à linha 60, a tag *replace* serve para substituir as ocorrências de \${project.version} no arquivo *index.jsp* pelo número da versão que está sendo gerada.

A **Listagem 4** apresenta uma sugestão de como poderia ficar o arquivo *index.jsp* com a variável relacionada à versão do projeto e que no momento da geração do pacote será substituída pelo número da versão desejada.

Automatizando a geração de pacotes com o Jenkins

Na **Listagem 3**, a partir da linha 103, é definida a tarefa `release`, responsável por criar uma tag no SVN com o código fonte da versão que será gerada no momento de sua execução. Essa atividade é conhecida como fechamento da release. Observe que ela recebe um parâmetro chamado `branch.name`, que se

não for informado no momento da execução pelo Jenkins, cria uma branch a partir do trunk e uma tag com o mesmo nome da branch. Caso contrário, ela não cria uma nova branch, mas apenas uma nova tag a partir do código que está na branch informada no parâmetro.

Listagem 2. Configurando a task de geração e deploy de pacote no build.xml – Parte 2.

```
46  <target name="generate-war">
47    <copy todir="${build.directory}/webapp">
48      <fileset dir="src/main/webapp">
49        <exclude name="**/web.xml"/>
50      </fileset>
51    </copy>
52
53  <copy todir="${build.directory}/classes">
54    <fileset dir="src/main/resources">
55      <exclude name="**/*.java"/>
56    </fileset>
57  </copy>
58
59  <echo message="replace properties on files"/>
60  <replace file="${build.directory}/webapp/index.jsp" value="">
61    <propertyFile="${basedir}/build.properties">
62      <replacefilter property="project.version">
63        <replacetoken><![CDATA[${project.version}]]</replacetoken>
64      </replacefilter>
65    </replace>
66
67  <path id="compile.classpath">
68    <fileset dir="lib">
69      <include name="*.jar"/>
70    </fileset>
71    <fileset dir="/usr/local/Tomcat/lib">
72      <include name="*.jar"/>
73    </fileset>
74  </path>
75
76  <javac destdir="${build.directory}/classes" srcdir="src/main/java"
77    classpathref="compile.classpath" includeantruntime="false" nowarn="true"
78    source="1.6" target="1.6" debug="true" compiler="javac1.6"
79    encoding="UTF-8" memoryMaximumSize="256m" fork="yes" />
80
81  <war destfile="${build.directory}/${project.name}-${project.version}.war"
82    webxml="src/main/webapp/WEB-INF/web.xml">
83    <fileset dir="${build.directory}/webapp"/>
84    <lib dir="lib"/>
85    <classes dir="${build.directory}/classes"/>
86  </war>
87
88  <delete dir="${build.directory}/classes"/>
89  <delete dir="${build.directory}/webapp"/>
90 </target>
91
92  <target name="undeploy">
93    <undeploy failonerror="no" url="${tomcat.manager.url}"
94      username="${tomcat.username}"
95      password="${tomcat.password}" path="/${project.name}"/>
96  </target>
97
98  <target name="deploy" depends="generate-war,undeploy">
99    <deploy url="${tomcat.manager.url}" username="${tomcat.username}"
100      password="${tomcat.password}" path="/${project.name}"
101      war="file:${build.directory}/${project.name}-${project.version}.war"/>
102</target>
```

Listagem 3. Configurando a tarefa de fechamento de release no build.xml – Parte 3.

```
103 <target name="release">
104   <property name="source.url" value="${source.root}/trunk"/>
105   <if>
106     <isset property="branch.name"/>
107     <then>
108       <property name="source.url"
109         value="${source.root}/branches/${branch.name}"/>
110     </then>
111   </if>
112   <echo message="update version to ${release.version}"/>
113   <echo file="${basedir}/build.properties"
114     append="false">project.version=${release.version}</echo>
115
116   <echo message="commit release version ${release.version}"/>
117   <svn refid="svn.settings">
118     <commit message="Release Version: ${release.version}">
119       <fileset dir="${basedir}">
120         <include name="build.properties"/>
121       </fileset>
122     </commit>
123   </svn>
124
125   <if>
126     <not>
127       <isset property="branch.name"/>
128     </not>
129     <then>
130       <echo message="create release branch ${release.version}"/>
131       <svn refid="svn.settings">
132         <copy srcUrl="${source.url}"
133           destUrl="${source.root}/branches/${release.version}"
134           message="${release.version}"/>
135       </svn>
136     </then>
137   </if>
138   <echo message="create release tag ${release.version}"/>
139   <svn refid="svn.settings">
140     <copy srcUrl="${source.url}" destUrl="${source.root}/tags/${release.version}"
141       message="${release.version}"/>
142   </svn>
143
144   <echo message="update new version to ${dev.version}"/>
145   <echo file="${basedir}/build.properties" append="false">
146     project.version=${dev.version}</echo>
147   <echo message="commit new version ${dev.version}"/>
148   <svn refid="svn.settings">
149     <commit message="Development Version: ${dev.version}">
150       <fileset dir="${basedir}">
151         <include name="build.properties"/>
152       </fileset>
153     </commit>
154   </svn>
155 </target>
156 </project>
```

A utilidade de criar uma branch no momento de fechar uma release é possibilitar que correções sejam feitas nessa versão e novos pacotes sejam gerados sem levar as alterações que estão sendo feitas no trunk. Assim a equipe de dev pode continuar trabalhando no trunk e o código da versão que está em produção estará separado. Os códigos que estão nas tags é que serão instalados nos servidores. Cada tag possui o código final de cada versão e não devem ser alteradas depois de criadas.

Por último, é necessário também criar um arquivo chamado *build.properties*, como mostrado na **Listagem 5**. Este serve para definir as propriedades que as tarefas do arquivo *build.xml* poderão utilizar. Esse arquivo será aproveitado pela tarefa *release* para armazenar automaticamente na propriedade **project.version** o valor da versão do projeto que está sendo trabalhada.

Listagem 4. Código do arquivo Index.jsp.

```
<%@ page language="java" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
 <head>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
 <title>Compra</title>
 </head>
 <body>
 <div>Versão: ${project.version}</div>
 </body>
</html>
```

Listagem 5. build.properties para o Ant.

```
project.version=1.0.0-SNAPSHOT
```

Ajustando o projeto com o Maven

O segundo projeto do sistema de compras coletivas será configurado com o Maven. Esta ferramenta foi criada depois do Ant e já vem com mais recursos pré-definidos, como as tarefas de compilação, execução das classes de testes, criação do pacote, entre outras, que seguem uma convenção e criam um ciclo de vida bem claro e simples do que se precisa fazer em cada fase de um projeto.

No Maven, por padrão, o arquivo de configuração se chama *pom.xml*. As **Listagens 6, 7 e 8** demonstram como ficaria este arquivo. Observe, mais uma vez, que o dividimos em três partes. Vamos a uma análise mais detalhada.

Como podemos notar na **Listagem 6**, após o trecho que define as informações iniciais do projeto, entre as linhas 20 e 47 declaramos as dependências, como é o caso da API do JSP, JSTL e o JUnit. Estas serão gerenciadas pelo Maven, um importante recurso que não temos no Ant.

Analisando agora a **Listagem 7**, entre as linhas 49 e 128, estão definidos os *profiles* para cada um dos ambientes em que será feita a instalação da aplicação, e diferentemente do que foi feito no projeto com o Ant, não é necessário colocar tags condicionais por ambiente, já que por padrão o Maven entende o conceito de *profiles*.

Além disso, em cada *profile* podem ser definidas configurações específicas dos plugins declarados no pom como, por exemplo, acontece com o *tomcat7-maven-plugin* (vide **Listagem 8**). Dependendo do valor da propriedade **profile** no momento da execução do comando de deploy, o plugin do Tomcat utilizará diferentes valores para a URL do manager, username e password.

Ainda na **Listagem 7**, a linha 53 marca o profile *default* como o padrão. Assim, se nenhum profile for informado na execução de

Listagem 6. Configurando as propriedades e dependências do projeto no pom.xml – Parte 1.

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
02   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
04   http://maven.apache.org/xsd/maven-4.0.0.xsd">
05
06   <modelVersion>4.0.0</modelVersion>
07   <groupId>br.com.devmedia</groupId>
08   <artifactId>compra</artifactId>
09   <version>1.0.0-SNAPSHOT</version>
10   <packaging>war</packaging>
11   <name>compra</name>
12   <properties>
13     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
14   </properties>
15   <scm>
16     <connection>scm:svn:https://compra.com.br/repos/compra/trunk
17     </connection>
18     <url>https://compra.com.br/repos/compra/trunk</url>
19   </scm>
20   <dependencies>
21     <!-- Servlet -->
22     <dependency>
23       <groupId>javax.servlet</groupId>
24       <artifactId>servlet-api</artifactId>
25         <version>2.5</version>
26         <scope>provided</scope>
27     </dependency>
28     <dependency>
29       <groupId>javax.servlet.jsp</groupId>
30       <artifactId>jsp-api</artifactId>
31       <version>2.1</version>
32       <scope>provided</scope>
33     </dependency>
34     <dependency>
35       <groupId>javax.servlet</groupId>
36       <artifactId>jstl</artifactId>
37       <version>1.2</version>
38     </dependency>
39
40     <!-- Test -->
41     <dependency>
42       <groupId>junit</groupId>
43       <artifactId>junit</artifactId>
44       <version>3.8.1</version>
45       <scope>test</scope>
46     </dependency>
47   </dependencies>
48
```

um comando, o *default* será utilizado. Por segurança, esse profile tem as configurações do plugin *tomcat7-maven-plugin* para fazer o deploy no ambiente local, evitando deste modo uma instalação não desejada em outro ambiente. Já o profile de *dev* tem os valores para realizar o deploy no servidor de dev, e o mesmo é feito para os outros ambientes também.

Na **Listagem 8**, entre as tags **<build>**, estão definidos os plugins que o Maven utilizará e que não vêm configurados por padrão. Na linha 132, por exemplo, temos o plugin *tomcat7-maven-plugin*, que é responsável por fazer o deploy do pacote no Tomcat. Já o plugin *maven-release-plugin*, na linha 150, serve para gerar as tags e branches do projeto na hora de fechar a release. Na linha 168, por sua vez, o plugin *maven-war-plugin* gera o pacote criando o arquivo com a extensão *war*.

Ainda nessa listagem, na linha 176, a tag **<filtering>**, do plugin *maven-war-plugin*, serve para substituir variáveis que o Maven manipula e que estão nos arquivos do projeto. Essa tag tem a mesma funcionalidade da tarefa *replace* que existe no Ant, e que em nosso exemplo vai substituir as ocorrências de *\${projeto}*

.version} no arquivo *index.jsp* pelo número da versão que está sendo gerada.

Ajustando o projeto com o Gradle

O terceiro e último projeto utiliza o Gradle como ferramenta de build. Assim como as demais, também possui um arquivo de configuração, chamado *build.gradle*, e seu conteúdo é escrito como se fosse um script (o mesmo formato do Groovy). A **Listagem 9** mostra qual seria o conteúdo do mesmo para o projeto. Neste momento, pelo tamanho do arquivo, já podemos identificar uma das grandes vantagens do Gradle: a sua capacidade de configuração utilizando poucas linhas.

Assim como o Maven, o Gradle também gerencia as dependências do projeto, bastando apenas listá-las no arquivo de configuração. Além disso, o Gradle também oferece diversos plugins para facilitar a execução das tarefas. Os plugins utilizados nesse projeto (*java*, *war*, *cargo*, *release* e *svntools*) foram declarados entre as linhas 17 e 21. Já entre as linhas 30 e 37 estão definidas todas as dependências.

Listagem 7. Configurando as configurações dos ambientes no pom.xml – Parte 2.

```
49   <profiles>
50     <profile>
51       <id>default</id>
52       <activation>
53         <activeByDefault>true</activeByDefault>
54       </activation>
55       <build>
56         <plugins>
57           <plugin>
58             <groupId>org.apache.tomcat.maven</groupId>
59             <artifactId>tomcat7-maven-plugin</artifactId>
60             <configuration>
61               <url>http://localhost:8090/manager/text</url>
62               <path>/compra</path>
63               <update>true</update>
64               <username>tomcat</username>
65               <password>tomcat</password>
66             </configuration>
67           </plugin>
68         </plugins>
69       </build>
70     </profile>
71
72     <profile>
73       <id>dev</id>
74       <build>
75         <plugins>
76           <plugin>
77             <groupId>org.apache.tomcat.maven</groupId>
78             <artifactId>tomcat7-maven-plugin</artifactId>
79             <configuration>
80               <url>http://dev.compra.com.br:8080/manager/text</url>
81               <path>/compra</path>
82               <update>true</update>
83               <username>tomcat</username>
84               <password>tomcat</password>
85             </configuration>
86           </plugin>
87         </plugins>
88       </build>
89     </profile>
90
91   <profile>
92     <id>qa</id>
93     <build>
94       <plugins>
95         <plugin>
96           <groupId>org.apache.tomcat.maven</groupId>
97           <artifactId>tomcat7-maven-plugin</artifactId>
98           <configuration>
99             <url>http://qa.compra.com.br:8080/manager/text</url>
100            <path>/compra</path>
101            <update>true</update>
102            <username>${tomcat.username}</username>
103            <password>${tomcat.password}</password>
104          </configuration>
105        </plugin>
106      </plugins>
107    </build>
108  </profile>
109
110 <profile>
111   <id>prod</id>
112   <build>
113     <plugins>
114       <plugin>
115         <groupId>org.apache.tomcat.maven</groupId>
116         <artifactId>tomcat7-maven-plugin</artifactId>
117         <configuration>
118           <url>http://compra.com.br:8080/manager/text</url>
119           <path>/compra</path>
120           <update>true</update>
121           <username>${tomcat.username}</username>
122           <password>${tomcat.password}</password>
123         </configuration>
124       </plugin>
125     </plugins>
126   </build>
127 </profile>
128 </profiles>
129
```

Listagem 8. Configurando os plugins no pom.xml – Parte 3.

```
130 <build>
131   <plugins>
132     <plugin>
133       <groupId>org.apache.tomcat.maven</groupId>
134       <artifactId>tomcat7-maven-plugin</artifactId>
135       <version>2.2.</version>
136     </plugin>
137     <plugin>
138       <groupId>org.apache.maven.plugins</groupId>
139       <artifactId>maven-compiler-plugin</artifactId>
140       <version>2.3.2</version>
141     <configuration>
142       <source>1.6</source>
143       <target>1.6</target>
144       <compilerArgument>-Xlint:all</compilerArgument>
145       <showWarnings>true</showWarnings>
146       <showDeprecation>true</showDeprecation>
147       <encoding>UTF-8</encoding>
148     </configuration>
149   </plugin>
150   <plugin>
151     <groupId>org.apache.maven.plugins</groupId>
152     <artifactId>maven-release-plugin</artifactId>
153     <version>2.4.2</version>
154   <configuration>
155     <providerImplementations>
156       <svn>javasvn</svn>
157     </providerImplementations>
158   </configuration>
159   <dependencies>
160     <dependency>
161       <groupId>com.google.code.maven-scm-provider-svnjava</groupId>
162       <artifactId>maven-scm-provider-svnjava</artifactId>
163       <version>2.0.6</version>
164       <scope>compile</scope>
165     </dependency>
166   </dependencies>
167 </plugin>
168 <plugin>
169   <groupId>org.apache.maven.plugins</groupId>
170   <artifactId>maven-war-plugin</artifactId>
171   <version>2.6.</version>
172   <configuration>
173     <webResources>
174       <resource>
175         <directory>src/main/webapp</directory>
176         <filtering>true</filtering>
177       </resource>
178     </webResources>
179   </configuration>
180 </plugin>
181 </plugins>
182 </build>
183 </project>
```

Listagem 9. Configurando o build.gradle.

```
01 repositories{
02   mavenCentral()
03 }
04 buildscript {
05   repositories {
06     jcenter()
07     maven {
08       url 'http://maven.tmatesoft.com/content/repositories/snapshots/'
09     }
10   }
11 dependencies {
12   classpath 'com.bmuschko:gradle-cargo-plugin:2.1.1',
13   'net.researchgate:gradle-release:2.1.2',
14   'at.bxm.gradleplugins:gradle-svntools-plugin:1.1.1'
15 }
16 }
17 apply plugin:'java'
18 apply plugin:'war'
19 apply plugin:'com.bmuschko.cargo'
20 apply plugin:'net.researchgate.release'
21 apply plugin:'at.bxm.svntools'
22
23 archivesBaseName = 'compra'
24 compileJava.options.encoding = 'UTF-8'
25
26 configurations {
27   provided
28   provided.extendsFrom(compile)
29 }
30 dependencies {
31   providedCompile 'javax.servlet:servlet-api:2.5'
32   providedCompile 'javax.servlet.jsp:jsp-api:2.1'
33   testCompile 'junit:junit:4.10'
34   runtime 'javax.servlet:jstl:1.2'
35   cargo 'org.codehaus.cargo:cargo-core-uberjar:1.4.5'
36   'org.codehaus.cargo:cargo-ant:1.4.5'
37 }
38 cargo {
39   containerId = 'tomcat7x'
40   port = 8090
41   deployable {
```

```
42   context = 'compra'
43 }
44 remote {
45   hostname = project.getProperty(profile + 'tomcat.hostname')
46   username = project.getProperty('tomcat.username')
47   password = project.getProperty('tomcat.password')
48 }
49 }
50
51 task branchRelease(type: at.bxm.gradleplugins.svntools.SvnBranch) {
52   branchName = version
53   commitMessage = version
54   username = project.getProperty('svn.username')
55   password = project.getProperty('svn.password')
56 }
57 branchRelease.onlyIf { !project.hasProperty('skipCreateBranch') }
58
59 release {
60   versionPropertyFile = 'gradle.properties'
61   failOnUnversionedFiles = false
62   failOnUpdateNeeded = false
63   revertOnFail = true
64   scmAdapters = [
65     net.researchgate.release.SvnAdapter
66   ]
67   svn {
68     username = project.getProperty('svn.username')
69     password = project.getProperty('svn.password')
70   }
71 }
72 afterReleaseBuild.dependsOn branchRelease
73
74 war {
75   eachFile { copyDetails ->
76     if (copyDetails.path == 'index.jsp') {
77       filter { line ->
78         line.replace('${project.version}', version)
79       }
80     }
81   }
82 }
```

Automatizando a geração de pacotes com o Jenkins

Entre as linhas 45 e 47, as propriedades do plugin `cargo`, responsável por fazer o deploy do pacote no Tomcat, são definidas dependendo do valor do parâmetro `profile`, passado no comando de execução do Gradle. As propriedades `hostname`, `username` e `password` recebem o valor que está descrito no arquivo de propriedades do Gradle, o `gradle.properties` (vide [Listagem 10](#)), sendo que o valor da propriedade `hostname` do plugin é obtido em tempo de execução a partir da concatenação do valor do parâmetro `profile` e do texto `".tomcat.hostname"`. Por exemplo, se o parâmetro `profile` receber o valor `"dev"`, será a propriedade `"dev.tomcat.hostname"` do arquivo `gradle.properties` que será lida e atribuída à propriedade `hostname`.

Na tarefa `branchRelease`, na linha 51, e na configuração do plugin `release` (encarregado de criar uma tag e atualizar os números das versões no código), o usuário e senha para acessar o SVN são lidos das propriedades `svn.username` e `svn.password`, que podem ser definidas no arquivo de propriedades do Gradle, ou no momento da execução do comando pelo Jenkins. Note que apenas no plugin de `release` do Gradle que precisamos passar explicitamente as credenciais do SVN. No Ant e no Maven as tarefas de `release` obterão esses valores no momento de sua execução.

Ainda na [Listagem 9](#), a linha 72 chama a task `branchRelease` antes de iniciar a tarefa `afterReleaseBuild` (que já vem definida dentro

do plugin `release` e é executada automaticamente após o pacote ser criado). Dessa forma, só depois que a release for construída com sucesso que a branch poderá ser criada. Já a linha 57 especifica que a task `branchRelease` só deverá ser executada se o parâmetro `skipCreateBranch` não for informado no momento de sua execução. Assim, podemos aproveitar a mesma tarefa para criar a release a partir do trunk (que requer a criação de uma tag e de uma branch) ou de alguma branch (que cria apenas outra tag).

Para substituir os valores das propriedades que o Gradle utiliza em tempo de execução nos arquivos do projeto, a linha 75 procura, dentro do pacote que está sendo criado, pelo arquivo `index.jsp` e o altera trocando as ocorrências de `${projjet.version}` pelo valor atual da versão gerada (linha 78).

Na [Listagem 10](#) é mostrado o arquivo de propriedades que o Gradle utilizaria em nosso exemplo. Note que algumas propriedades desse arquivo ficaram em branco, pois deverão ser passadas como parâmetro no comando do Gradle. Se outra propriedade definida no arquivo for passada como parâmetro na execução da tarefa, o valor que estiver no arquivo é desconsiderado.

Configurando o Jenkins

Com os três projetos devidamente configurados, basta colocar no Jenkins as tarefas das ferramentas de build que ele deve chamar

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

na execução de seus jobs. Os jobs criados ficarão listados na página principal da ferramenta, mas se preferir, posteriormente você pode agrupá-los em abas para uma melhor organização das tarefas. Antes, no entanto, é preciso instalar alguns plugins utilitários e cadastrar os usuários e senhas que a ferramenta utilizará. A **Figura 2** mostra o menu principal exibido na home do Jenkins.

Listagem 10. Conteúdo do arquivo gradle.properties.

```
version=1.0.0-SNAPSHOT
svn.username=
svn.password=
profile=default
tomcat.username=tomcat
tomcat.password=tomcat
default.tomcat.hostname=localhost
dev.tomcat.hostname=dev.compra.com.br
qa.tomcat.hostname=qa.compra.com.br
prod.tomcat.hostname=compra.com.br
```



Figura 2. Menu principal do Jenkins

Sendo assim, na tela exibida após clicar na opção *Gerenciar Jenkins*, instale os seguintes plugins: *Mask Passwords Plugin*, *Credentials Binding Plugin* e *Gradle Plugin*. Eles servem para mascarar as senhas digitadas pelo usuário no console de build do Jenkins e para permitir que jobs tenham acesso às credenciais que foram armazenadas no próprio ambiente de integração contínua. Ainda nesta tela, na opção *Configurar o sistema*, dependendo da ferramenta de build configurada no projeto, ajuste a versão para o Ant, Maven ou Gradle. Verifique ainda a versão do Subversion que o seu projeto está utilizando e a selecione também nesta tela.

Figura 3. Configuração geral do job compra-trunk

Figura 4. Configuração específica do Job compra-trunk para o Ant

Voltando ao menu da **Figura 2**, na opção *Credentials*, cadastre o usuário e senha do repositório SVN para que no momento da criação dos jobs você possa referenciá-los.

Agora, estamos aptos a criar os jobs no Jenkins. É através do job que informamos à ferramenta o que ela deve fazer. Na tela de criação de jobs, apresentada após selecionar a opção *Novo Job*, são exibidas três opções de projeto: *projeto software free-style*, *projeto Maven* e *projeto de múltiplas configurações*. Ao criar jobs para os projetos com Ant e Gradle, utilizaremos o tipo de projeto *software free-style*. Já para os jobs do projeto com o Maven, devemos optar pelo tipo *projeto Maven*.

Para atender ao processo de integração contínua que definimos anteriormente, devemos criar quatro jobs: **compra-trunk**, **compra-release**, **compra-deploy-release** e **compra-release-from-branch**, cada um com objetivos e configurações específicos. Como são três projetos com ferramentas de build diferentes, ao final serão criados 12 jobs. Diante disso, para simplificar, mostraremos as telas de criação dos quatro jobs com os campos que são comuns aos três projetos, e em seguida, os campos que variam de acordo com a ferramenta de build.

Criando o job de build contínuo e deploy a partir do trunk

Como informado anteriormente, desejamos que todo commit no trunk faça o Jenkins executar os testes, gerar e fazer o deploy do pacote no ambiente de dev. Para viabilizar essa tarefa, vamos criar um job no Jenkins e chamá-lo de **compra-trunk**, como mostra a **Figura 3**. Neste job, não é preciso definir parâmetros de construção (pois estes serão informados na seção de *Build*) e ele deve ser executado automaticamente a cada cinco minutos.

Como podemos verificar, a configuração na seção *Trigger de build* está com o campo *Consultar periodicamente o SCM* marcado e com o valor “H/5 * * * *” no campo *Agenda*, confirmando o que mencionamos no final do parágrafo anterior. Na seção *Gerenciamento de código fonte* é informada a URL do SVN onde o código está armazenado e selecionada as credencias que deverão ser utilizadas quando o job executar para obter o código do projeto no servidor.

A única seção na criação do job que varia de acordo com a ferramenta adotada (Ant, Maven ou Gradle) é a de *Build*, e nela encontramos o botão *Adicionar passo no build*, que possui as seguintes opções: *Chamar alvos Maven de alto nível*, *Executar no comando do Windows*, *Executar shell*, *Invocar Ant* e *Invoke Gradle script* (o nome está em inglês porque nem todos os plugins possuem uma versão para o português). Estas opções servem

para adicionar comandos que o Jenkins deve executar no momento da construção do pacote. Em nossos exemplos, utilizaremos apenas os dois últimos tipos de passos, já que para o Maven, o tipo de job *projeto Maven* adiciona automaticamente a seção de *Build* (ou *Construir*, em versões do Jenkins em português).

As **Figuras 4**, **5** e **6** mostram as configurações da seção *Build* da tela de criação do job para cada uma das ferramentas de build que estamos utilizando.

Para o Ant, como indicado na **Figura 4**, basta clicar no botão *Adicionar passo no build* e adicionar um passo do tipo *Invocar Ant*, selecionando em seguida a versão do Ant que você configurou na tela *Gerenciar Jenkins*. A definição do ambiente em que o deploy será feito acontece com a atribuição de algum valor à propriedade *profile*, que neste job recebe o valor fixo do ambiente **dev**, já que queremos que esse job faça deploys apenas no servidor de dev. No campo *Alvos*, por sua vez, é informado o nome da tarefa do Ant que o Jenkins chamará quando este job for executado.

Para o projeto Maven, como mostra a **Figura 5**, basta colocar no campo *Metas e Opções*, da seção *Construir*, o comando do Maven “*clean install tomcat7:deploy*” com o argumento “*-Pdev*”, que faz a propriedade *profile* receber o valor **dev**. Primeiramente, esse comando limpa a pasta onde o Maven cria o arquivo do pacote (WAR). Logo após, cria o arquivo WAR com a aplicação usando as



Figura 5. Configuração específica do Job compra-trunk para o Maven

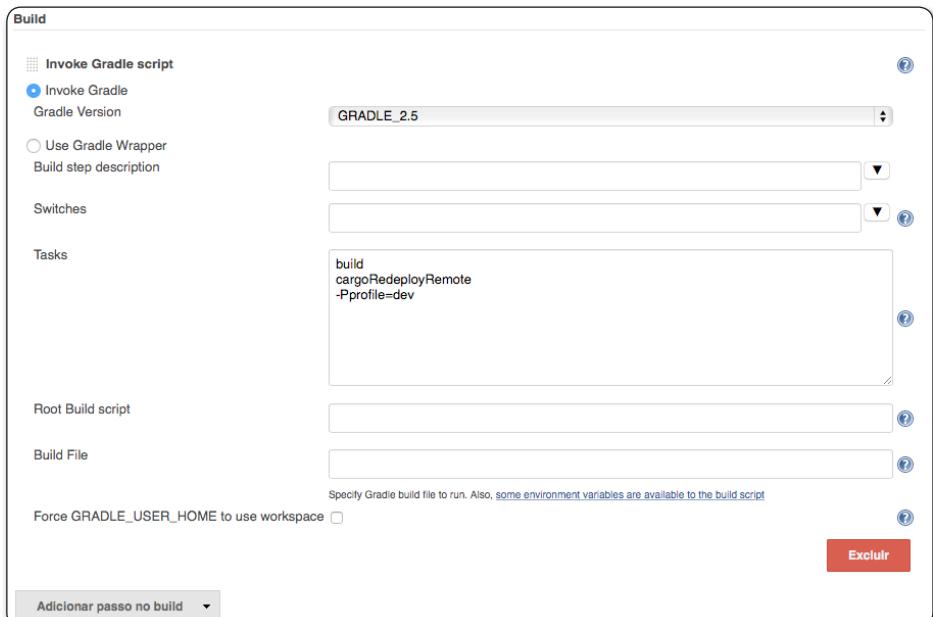


Figura 6. Configuração específica do Job compra-trunk para o Gradle

Nome do Maven project: compra-release

Descrição:

[HTML escapado] Visualizar

Descartar builds antigos

Este build é parametrizado

Parâmetro string

Nome: release.version
Valor padrão:
Descrição:
[HTML escapado] Visualizar

Excluir

Parâmetro string

Nome: dev.version
Valor padrão:
Descrição:
[HTML escapado] Visualizar

Excluir

Adicionar parâmetro

Desabilitar builds (Nenhum novo build será executado até que este projeto seja habilitado novamente.)

Execute os builds se necessário

Opções avançadas do projeto

Avançado...

Gerenciamento de código fonte

Nenhuma CVS CVS Projectset Subversion

Módulos

Repository URL: https://compra.com.br/repos/compra/trunk
Credentials: rafaelsvn***** (compra) Add

Local module directory: .
Repository depth: infinity
Ignore externals:

Add module... Add additional credentials...

Additional Credentials

Check-out Strategy: Always check out a fresh copy
Delete everything first, then perform 'svn checkout'. While this takes time to execute, it ensures that the workspace is in the pristine state.

Navegar no repositório: (Auto)

Avançado...

Trigger de builds

Construir um SNAPSHOT sempre que executar uma construção
 Construir após a construção de outros projetos
 Construir periodicamente
 Consultar periodicamente o SCM

Ambiente de build

Mask passwords (and enable global passwords)

Passwd Parameters, or any other type of build parameters selected for masking in Hudson's/Jenkins' main configuration screen (Manage Hudson > Configure System), will be automatically masked.

Use secret text(s) or file(s)

Bindings

Username and password (separated)

Username Variable: svn.username
Password Variable: svn.password
Credentials: Specific credentials Parameter expression rafaelsvn***** (compra) Add

Excluir

Adicionar

Figura 7. Configuração geral do Job compra-release na tela de criação de job no Jenkins

configurações do profile *dev* e, finalmente, faz o deploy do pacote gerado no Tomcat.

Por fim, como expõe a **Figura 6**, para criar o job utilizando o projeto Gradle, é necessário adicionar um passo do tipo *Invoke Gradle Script* e selecionar a versão do Gradle que você configurou na tela *Gerenciar Jenkins* no campo *Gradle Version*. A definição do ambiente em que o deploy será feito acontece com a passagem do parâmetro *profile*, que neste job recebe o valor fixo do ambiente *dev*. Ainda nesta imagem, no campo *Tasks* são colocadas as tasks *build* e *cargoRedeployRemote*, que fazem a tarefa de gerar o pacote e realizar o deploy no Tomcat, respectivamente. A task *build* é uma tarefa que já vem por padrão no Gradle. Já a task *cargoRedeployRemote* é implementada dentro do plugin *cargo*. Por isso elas não foram declaradas explicitamente no arquivo de configuração da ferramenta de build.

Criando o job de fechamento da release a partir do trunk

Precisamos também de um job que feche uma release do projeto a partir do código que está no trunk, criando uma tag com o nome da release e também uma branch para eventuais correções que precisem ser feitas. Este job se chamará “compra-release”, como mostra a **Figura 7**. Ele recebe como parâmetros o número da versão da nova release (*release.version*) e a versão que continuará sendo desenvolvida no trunk (*dev.version*).

Ainda na **Figura 7**, na seção *Ambiente de build*, as configurações de *Mask Password* e *Use secret text(s) or files(s)* também são ativadas, sendo que a primeira faz com que as senhas que forem usadas no job sejam exibidas com asteriscos no console de execução da ferramenta, e a última atribui os valores das credenciais cadastradas no Jenkins às variáveis *svn.username* e *svn.password*.

As **Figuras 8, 9 e 10** apresentam as configurações da seção *Build* para os projetos com Ant, Maven e Gradle, respectivamente.

No campo *Propriedades*, as propriedades *svn.username* e *svn.password* da tarefa *release*, configurada no Ant, recebem os valores das credenciais que foram selecionadas na seção *Bindings* da **Figura 7** e que foram atribuídas às variáveis *svn.username* e *svn.password*. A única tarefa do Ant que será chamada nesse job é a *release*, informada no campo *Alvos*, pois todas as ações que esta tarefa executará já estão definidas no arquivo *build.xml* do próprio projeto com Ant.

Voltando nossa atenção ao projeto com Maven, neste job precisamos informar o seguinte comando do plugin *maven-release-plugin* no campo *Metas e opções* para que o Jenkins o chame no momento da execução do job:

```
-batch-mode clean -e release:branch release:prepare release:perform
-Dusername=${svn.username} -Dpassword=${svn.password} -Dtag=${release.version}
-DreleaseVersion=${release.version} -DdevelopmentVersion=${dev.version}
-DupdateBranchVersions=true -DbranchName=${release.version}
```

Estas instruções dizem ao Maven para executar o comando em modo batch (sem esperar por uma entrada do usuário), limpando primeiro a pasta onde o pacote será gerado e depois criando uma

Automatizando a geração de pacotes com o Jenkins

Build

Invoker Ant

Versão do Ant: **ANT_APACHE**

Alvos: **release**

Arquivo de Construção:

Propriedades:

```
svn.username=${svn.username}  
svn.password=${svn.password}
```

Opções Java:

Excluir

Adicionar passo no build ▾

This screenshot shows the configuration for an Ant build step in Jenkins. It includes fields for the Ant version (ANT_APACHE), target (release), build file (empty), properties (containing SVN credentials), and Java options (empty). A red 'Excluir' button is visible at the bottom right.

Figura 8. Configuração específica do Job compra-release para o Ant

branch e uma tag com o código do pacote. A branch e a tag são criadas com o nome recebido no parâmetro **release.version**, e o valor da versão do código no trunk vem no parâmetro **dev.version**. As propriedades **svn.username** e **svn.password** são passadas ao Maven através dos parâmetros **username** e **password**.

Analisando agora a configuração para o projeto Gradle, na Figura 10, o campo **Tasks** recebe o nome da tarefa **release** com seus respectivos parâmetros. Os primeiros parâmetros são o usuário e senha do SVN. Em seguida, temos **release.version** e **newVersion**, que recebem os valores das versões que deverão ficar no código da branch da release e no código do trunk. No entanto, se esses parâmetros forem com o valor em branco, o parâmetro **release.userAutomaticVersion** fará com que o próprio plugin **release** defina os valores das versões automaticamente. Dessa forma, quando algum usuário executar esse job no Jenkins, a tarefa **release** criará uma tag e uma branch com o código da versão que está sendo fechada, além de atualizar o número da versão no código principal da aplicação.

Construir

POM Raiz: **pom.xml**

Metas e opções:

```
-batch-mode clean -e release:branch release:prepare release:perform -Dusername=${svn.username} -Dpassword=${svn.password}
```

Avançado...

This screenshot shows the configuration for a Maven build step in Jenkins. It includes fields for the POM root (pom.xml) and goals (-batch-mode clean -e release:branch release:prepare release:perform -Dusername=\${svn.username} -Dpassword=\${svn.password}). An 'Avançado...' button is visible at the bottom right.

Figura 9. Configuração específica do Job compra-release para o Maven

Build

Invoke Gradle script

Invoke Gradle
Gradle Version: **GRADLE_2.5**

Use Gradle Wrapper
Build step description:

Switches:

Tasks:

```
release  
-Psvn.username=${svn.username}  
-Psvn.password=${svn.password}  
-Prelease.version=${release.version}  
-PnewVersion=${dev.version}  
-Prelease.useAutomaticVersion=true
```

Root Build script:

Build File:

Specify Gradle build file to run. Also, some environment variables are available to the build script

Force GRADLE_USER_HOME to use workspace

Excluir

Adicionar passo no build ▾

This screenshot shows the configuration for a Gradle build step in Jenkins. It includes fields for the Gradle version (GRADLE_2.5), task (release), and parameters (-Psvn.username=\${svn.username}, -Psvn.password=\${svn.password}, -Prelease.version=\${release.version}, -PnewVersion=\${dev.version}, -Prelease.useAutomaticVersion=true). A 'Specify Gradle build file to run...' field and a 'Force GRADLE_USER_HOME to use workspace' checkbox are also present. A red 'Excluir' button is visible at the bottom right.

Figura 10. Configuração específica do job compra-release para o Gradle

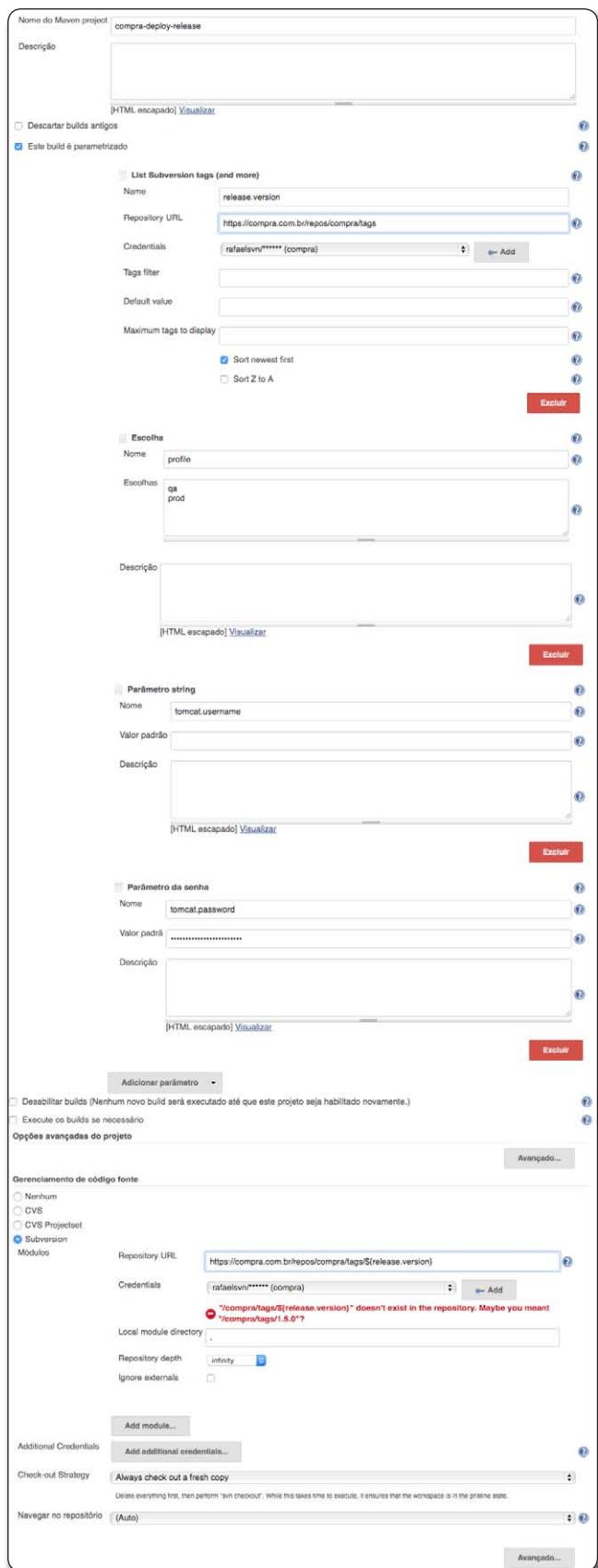


Figura 11. Configuração geral do job compra-deploy-release na tela de criação de job no Jenkins

Criando o job de deploy de releases

Depois que uma release é criada com o job “compra-release”, precisamos que ela seja instalada nos servidores da empresa. Para realizar essa tarefa, um novo job será criado com o nome “compra-deploy-release”, como mostra a **Figura 11**. Para executá-lo, o usuário precisará selecionar na tela de execução do job o número da release, o nome do ambiente em que deseja fazer o deploy e informar o usuário e a senha do Tomcat do ambiente selecionado. Dessa forma garantimos que apenas pessoas autorizadas realizem o deploy da aplicação em ambientes como QA e produção.

Na **Figura 11** o parâmetro `release.version` lista todas as tags criadas, permitindo assim o deploy apenas de versões fechadas do projeto. Não é necessário se preocupar com a mensagem de erro na seção *Gerenciamento de código fonte*, porque o Jenkins tenta validar a URL do SVN e não encontra (já que colocamos a variável `${release.version}` como parte do endereço), mas ela ficará com o valor correto no momento de execução, quando o parâmetro `release.version` receber algum valor.

Nas **Figuras 12, 13 e 14**, temos as configurações da seção *Build* para os projetos com Ant, Maven e Gradle, respectivamente.

No campo *Alvos* da **Figura 12**, apenas a tarefa *deploy*, que está descrita no arquivo de configuração do Ant, será chamada neste job. Ela criará um pacote a partir do código da tag selecionada na tela de execução do job no Jenkins e fará o deploy deste pacote no Tomcat correspondente ao profile também selecionado na tela. Observe que não é necessário passar novamente o parâmetro **profile** no campo *Alvos*, pois o próprio Jenkins, quando executar a tarefa de *deploy*, montará o comando passando todos os parâmetros que foram definidos na seção *Este build é parametrizado* da **Figura 11**.

Com relação à configuração para o Maven, no campo *Metas e Opções* da Figura 13, o comando limpa a pasta onde os arquivos do pacote são gerados, cria um arquivo war e faz o deploy dele no Tomcat do ambiente especificado no parâmetro **profile**, utilizando o usuário e senha também informados como parâmetros na tela de execução do job no Jenkins.

Voltando a falar sobre a configuração no Gradle, no campo *Tasks* da **Figura 14** as tarefas *build* (que é uma tarefa padrão do Gradle) e *cargoRedeployRemote* (do plugin cargo) são chamadas para construírem o pacote e fazerem a instalação dele no Tomcat. Neste caso, também não é necessário passar explicitamente o parâmetro *profile* no campo *Tasks*, pois o próprio Jenkins montará o comando do Gradle passando esse parâmetro.

Criando o job de fechamento da release a partir de branches

Quando uma release é instalada em algum ambiente e os usuários começam a testá-la, bugs podem ser encontrados. Para corrigi-los, os desenvolvedores deverão trabalhar na branch correspondente à versão da release em que foi encontrado o problema. Depois que o problema for corrigido, o próximo passo é gerar uma nova versão da aplicação, mas a partir da branch corrigida e não do trunk. Assim, se, por exemplo, a release atual é a 1.0.0 e o trunk está com a versão 1.1.0-SNAPSHOT em desenvolvimento, a versão que seria gerada a partir da branch 1.0.0 seria, por exemplo,

Automatizando a geração de pacotes com o Jenkins

The screenshot shows the Jenkins job configuration interface for the 'Build' section. Under the 'Invocar Ant' heading, the 'Versão do Ant' dropdown is set to 'ANT_APACHE'. The 'Alvos' dropdown contains the value 'deploy'. The 'Arquivo de Construção' field is empty. The 'Propriedades' and 'Opções Java' fields are also empty. A red 'Excluir' button is located at the bottom right. At the bottom left, there is a 'Adicionar passo no build' dropdown menu.

Figura 12. Configuração específica do job compra-deploy-release para o Ant

a 1.0.1, onde apenas o último dígito seria incrementado a cada nova release.

Para orquestrar essas atividades, um novo job será criado com o nome “compra-release-from-branch”, como mostra a Figura 15.

Assim como nos tópicos anteriores, as configurações da seção *Build* para os projetos com Ant, Maven e Gradle são mostradas em imagens à parte; neste caso nas Figuras 16, 17 e 18.

Na primeira das três figuras, no campo *Alvos*, a tarefa *release* é definida para ser chamada pelo Jenkins e, deste modo, fechar uma nova release do sistema, contudo, usando o código da branch informada no parâmetro **branch.version**, que é informado na URL do SVN no campo *Repository URL* da Figura 15, seção *Gerenciamento de código fonte*.

Para concluir a configuração deste job no projeto que adota o Ant, no campo *Propriedades* (vide Figura 16), a propriedade **dev.version** é passada com o valor do parâmetro **branch.version**. Dessa forma, a tarefa *release* configurada no Ant não criará uma nova branch para esta nova versão,

The screenshot shows the Jenkins job configuration interface for the 'Build' section. Under the 'Construir' heading, the 'POM Raiz' dropdown is set to 'pom.xml'. The 'Metas e opções' dropdown contains the value 'clean install tomcat7:deploy -P\${profile} -Dtomcat.username=\${tomcat.username} -Dtomcat.password=\${tomcat.password}'. A grey 'Avançado...' button is located at the bottom right. At the bottom left, there is a 'Adicionar passo no build' dropdown menu.

Figura 13. Configuração específica do job compra-deploy-release para o Maven

The screenshot shows the Jenkins job configuration interface for the 'Build' section. Under the 'Invoke Gradle script' heading, the 'Gradle Version' dropdown is set to 'GRADLE_2.5'. The 'Tasks' dropdown contains the value 'build cargoRedeployRemote'. The 'Root Build script' and 'Build File' fields are empty. A note below states: 'Specify Gradle build file to run. Also, some environment variables are available to the build script'. The 'Force GRADLE_USER_HOME to use workspace' checkbox is unchecked. A red 'Excluir' button is located at the bottom right. At the bottom left, there is a 'Adicionar passo no build' dropdown menu.

Figura 14. Configuração específica do job compra-deploy-release para o Gradle

pois ela já está sendo gerada a partir de uma branch.

De volta ao projeto com Maven, neste job precisamos informar no campo *Metas e opções* o seguinte comando para executar o plugin *maven-release-plugin* do Maven:

```
--batch-mode clean -e release:prepare release:perform
-Dusername=${svn.username}
-Dpassword=${svn.password}
-DreleaseVersion=${release.version}
-DdevelopmentVersion=${branch.version}
-Dtag=${release.version}
```

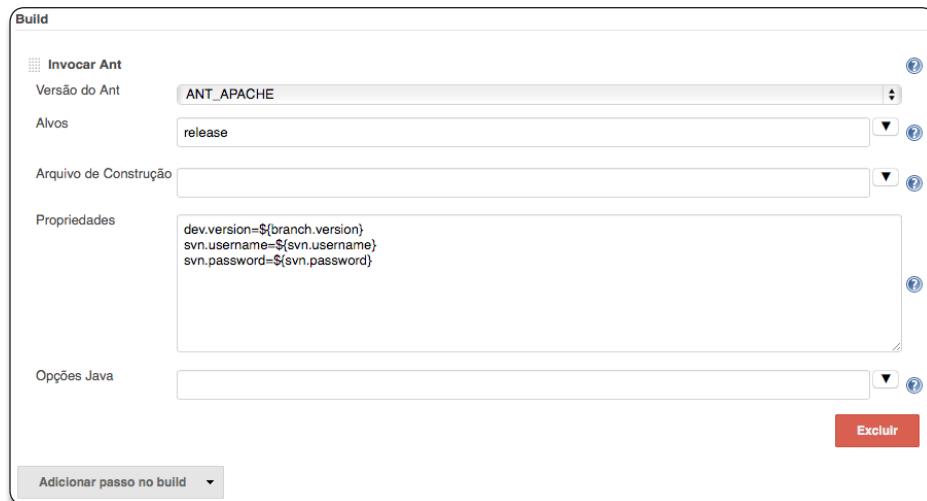


Figura 15. Configuração geral do job compra-release-from-branch no Jenkins



Figura 16. Configuração específica do job compra-release-from-branch para o Ant

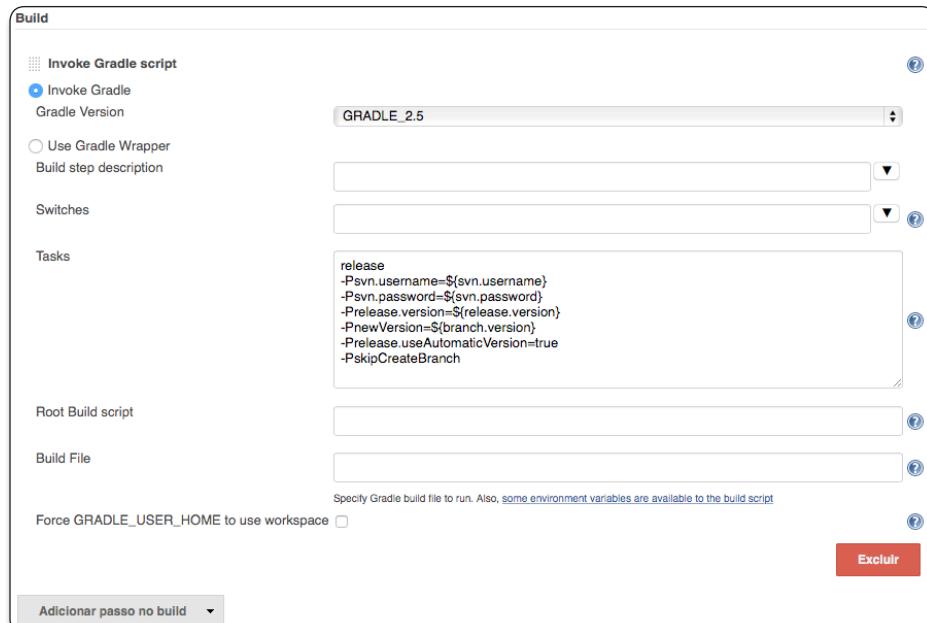


Figura 17. Configuração específica do Job compra-release-from-branch para o Maven

Estas instruções dizem para o Maven executar o comando em modo batch (sem esperar por uma entrada do usuário), limpando primeiro a pasta onde o pacote será gerado e depois criando uma tag com o código do pacote e com o nome recebido no parâmetro **tag**. Os valores recebidos nos parâmetros **releaseVersion** e **developmentVersion** serão colocados na propriedade **version** dos arquivos *pom.xml* que estão no código da tag e da branch, respectivamente.

O campo *Metas e Opções* da seção *Construir* (vide **Figura 17**) recebe o comando completo do Maven para executar o plugin *release* e criar uma nova tag no SVN com o nome recebido em **release.version**, e alterar o valor da versão do código da branch que foi selecionada para o valor recebido no parâmetro **branch.version**. Diferentemente do comando do job “compra-release”, no entanto, esse comando não cria uma nova branch.

Para encerrar a configuração do último job do projeto relacionado ao Gradle, no campo *Tasks* da **Figura 18**, a tarefa *release* também recebe como parâmetro (além dos já conhecidos do job *compra-release*) **skipCreateBranch**, que informa ao Gradle que ele não deve criar outra branch ao gerar o pacote desta nova release.

Agora, todos os jobs estão configurados e prontos para serem executados pelo Jenkins. Nossa primeiro job, o “compra-trunk” é executado automaticamente pelo Jenkins a cada cinco minutos, gerando o pacote e fazendo o deploy da aplicação no ambiente de dev. Deste modo não é necessário se preocupar com ele. Já os jobs “compra-release” e “compra-release-from-branch” serão executados pela equipe de dev, nos momentos de entrega de uma nova versão a partir do trunk ou a partir de uma branch, respectivamente. Por fim, o job “compra-deploy-release” pode ser executado pela equipe de QA ou de implantação depois que uma nova release for entregue pela equipe de dev e precisar ser instalada em algum ambiente.

Dependendo do seu projeto, automatizar a geração de pacotes pode ser um pouco mais trabalhoso do que o que foi mostrado, sendo necessário configurar mais plugins

Automatizando a geração de pacotes com o Jenkins

The screenshot shows the Jenkins job configuration for 'compra-release-from-branch'. It includes sections for parameters (release.version and branch.version), repository (Subversion URL: https://compra.com.br/repos/compra/branches/{branch.name}, credentials: rafaelsvn***** (compra)), and triggers (Poll SCM every 5 minutes). The 'Advanced...' button is visible at the bottom right of each section.

ou mudar toda a forma como sua equipe gerencia o fluxo de entrega de releases. No entanto, ainda assim esse esforço vale a pena, visto que as tarefas de criar e realizar o deploy de pacotes não precisam mais ser manuais, sendo muito mais proveitoso utilizar esse tempo em outras atividades como garantia de qualidade, monitoramento e melhoria da performance da aplicação, entre outras.

Como sabemos, não existe uma única forma de trabalhar, existe sim a que melhor se adéqua à sua equipe. Conhecendo como é simples trabalhar com o Jenkins e as ferramentas de build mais famosas do mercado, cabe a você e a sua equipe avaliar as opções existentes e decidir por aquela que seja mais fácil de aprender e utilizar, simplificando assim o seu dia a dia e deixando sua equipe e sua empresa ainda mais produtivas.

Autor



Rafael Campos Lima

<http://rafaelcamposl.blogspot.com.br>

Desenvolvedor Java Sênior, graduado em Ciências da Computação pela USP São Carlos, com certificação SCJP e SCWCD.



Links:

Site do projeto Jenkins CI.

<https://jenkins-ci.org/>

Site do projeto Apache Maven.

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

Site do projeto Apache Ant.

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

Site do Gradle.

<https://gradle.org/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Figura 18. Configuração específica do job compra-release-from-branch para o Gradle

**Conhecimento
faz diferença!**



Agilidade: Acompanhamento de projetos ágeis distribuído através do Daily Meeting

Projeto
Diagrama de sequência na prática

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

SOA
Processo e levantamento de requisitos de negócios – Parte 2

Qualidade de Software
Definição, características e importância

Automação de Testes

Cuidados a serem tomados na implantação

Processo e automação de testes de Software

Aulas desta edição:

- Atividades da Gerência de Projetos – Partes 10 a 14
- Estratégia de Teste Funcional baseada em Casos de Uso – Partes 5 a 9

ISSN 1983127-7

9781983127008 00028

+ de 290 vídeos
para assinantes

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

Faça um upgrade em sua carreira

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

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

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

 **DEVMEDIA**

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.

Supporte 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