



Primeiros passos com o WildFly Swarm

Uma nova e diferenciada opção
para criar microsserviços

Edição 150 :: R\$ 14,90

 DEV MEDIA

DevOps e Integração Contínua

Como adequar seu processo
de CI a essa nova cultura

INTRODUÇÃO AO JAVA 9

Conheça os novos recursos

Apache Cassandra e Java EE
Implementando uma aplicação
passo a passo com
WildFly e PrimeFaces

Dominando o Couchbase
Aprenda a desenvolver
soluções escaláveis com
um banco de dados NoSQL



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!

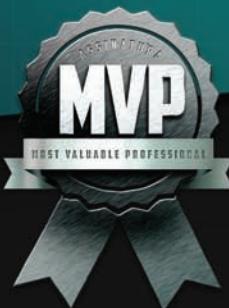
CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

- + 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**

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

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

Sumário

Artigo no estilo Curso

06 – Por dentro do banco de dados NoSQL Couchbase – Parte 2

[Fernando Henrique Fernandes de Camargo]

Artigo no estilo Curso

18 – Como usar o Apache Cassandra em aplicações Java EE – Parte 2

[Marlon Patrick]

Conteúdo sobre Novidades

Destaque - Vanguarda

32 – Introdução ao Java 9: Conheça os novos recursos

[José Guilherme Macedo Vieira]

Conteúdo sobre Novidades

48 – Simplificando o desenvolvimento de microserviços com o WildFly Swarm

[Joel Backschat]

Artigo no tipo Mentoring

Destaque - Reflexão

60 – DevOps: Como adequar seu processo de CI a essa nova cultura

[Pedro E. Cunha Brigatto]



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

REVISTAS DIGITAIS



Imagine poder ter acesso a todas as edições das revistas Java Magazine, .NET Magazine, SQL Magazine, Mobile Magazine, Engenharia de Software, ClubeDelphi e Easy Java.

São mais de 4.000 artigos publicados!

Uma verdadeira biblioteca online perfeita para seus estudos!



Para mais informações :

<http://www.devmmedia.com.br/mvp>



Por dentro do banco de dados NoSQL

Couchbase – Parte 2

Veja neste artigo como desenvolver aplicações escaláveis com Couchbase

ESTE ARTIGO FAZ PARTE DE UM CURSO

Muito tem se falado de NoSQL nos últimos anos. De forma simples, podemos descrevê-lo como o movimento que norteou a criação de bancos de dados bem especializados em certos problemas e alguns de uso mais geral. O que todos eles têm em comum é a capacidade de serem utilizados em grandes aplicações e conseguirem escalar o suficiente para que as mesmas continuem com boa performance mesmo com grande fluxo de dados e usuários simultâneos.

Dentre as opções disponíveis, destacam-se aquelas cuja estrutura de dados baseia-se em documentos. Isso porque eles podem servir tanto para aplicações de uso geral, como também para aplicações mais específicas, que exigem flexibilidade nos dados armazenados. O MongoDB e Couchbase são os conhecidos dessa categoria. O primeiro é, atualmente, o mais utilizado, enquanto o segundo vem ganhando destaque e apresentando ótimos resultados em benchmarks que provam sua superioridade em performance.

O Couchbase, banco de dados apresentado na primeira parte deste artigo, se mostra muito útil a seus usuários. Não só pela sua grande performance e ótima escalabilidade, mas também pela sua capacidade de sincronização com dispositivos móveis, uma nova funcionalidade que ainda não existe em seu concorrente. Com esse recurso, viabiliza-se a facilidade da sincronização de dados entre dispositivos móveis, que poderão operar sem acesso à internet, e o servidor, que será o centro de dados.

Fique por dentro

Este artigo apresenta a parte prática de um banco de dados NoSQL com estrutura de dados baseada em documentos: o Couchbase. Aqui será exposta a continuação do artigo que analisou a teoria do mesmo. Dessa vez, no entanto, será vista a configuração e implementação de um cliente Java que utiliza o Couchbase. Assim, esse conteúdo é útil para desenvolvedores que procuram um banco de dados com grande potencial de escalabilidade e/ou uma estrutura de dados bem flexível.

Com suas vantagens analisadas, foi vista a teoria sobre seu funcionamento, desde o gerenciamento do cluster e a conexão entre seus elementos, até o funcionamento interno de cada nó, o que faz com que toda essa performance proposta seja garantida. Dentre os conceitos relacionados a seu funcionamento, vale ressaltar a ausência de hierarquia entre os nós do cluster. Com o intuito de evitar um único ponto de falha, não se utiliza a tradicional arquitetura mestre-escravo. Ao invés disso, todos os nós se conhecem e a aplicação cliente poderá se conectar com qualquer um deles, sendo informada da existência de todos os outros automaticamente.

Esses nós, individualmente, garantem tempos de respostas na ordem de milissegundos através de um cache em memória, que armazena os últimos documentos acessados. Além disso, as operações de inserção e alteração são realizadas primeiro na memória, o que possibilita ainda mais agilidade.

Para implementar o escalonamento horizontal, os dados são distribuídos entre esses nós com o recurso de *auto-sharding*. E para se prevenir de desastres, uma réplica de cada documento é guardada em outro nó. Assim, caso um dos nós venha a ficar fora do ar, um

Bucket Name	Nodes	Item Count	Ops/sec	Disk Fetches/sec	RAM/Quota Usage	Data/Disk Usage
Exemplo	1	32	0	0	31.5MB / 512MB	52.4MB / 52.6MB

Access Control: Authentication Replicas: 1 replica copy Compaction: Not active

Cache Metadata: Value Eviction Disk I/O priority: Low

Buttons: Documents, Views, Compact, Edit

Figura 1. Tela de configuração dos Data Buckets

outro assume seu lugar e um rebalanceamento dos dados acontece a fim de redistribuir os dados e criar novas réplicas.

Com o funcionamento do Couchbase detalhadamente abordado, foram apresentados os campos reservados de seus documentos, o comportamento de cada um deles, bem como a capacidade de armazenar outros valores que não sejam documentos.

Por fim, uma técnica foi apresentada para que seja feito o versionamento de esquemas dos documentos de forma que sejam atualizados, quando cada documento for acessado naturalmente pela aplicação. Utilizando essa técnica, veremos a seguir o desenvolvimento de uma aplicação Java que se comunicará com um banco de dados Couchbase implementando as operações básicas e o mapeamento entre objetos e documentos.

Desenvolvimento com o Java SDK

Antes de iniciar a implementação com o cliente Java do Couchbase precisamos instalá-lo e configurá-lo. Para isso, está disponível na seção **Links** a URL com as instruções de download e instalação da versão gratuita, além de como executar o serviço do Couchbase. Esses detalhes variam de acordo com o sistema operacional do usuário.

Concluída essa etapa, acesse a URL `http://localhost:8091`. Nesse momento uma tela será apresentada para que seja feita a configuração inicial do banco de dados. Nessa tela pode-se registrar o administrador como desejar e, por ser apenas um banco usado para desenvolvimento, não é necessário mais de 1GB de memória em sua configuração. Além disso, também será questionada a quantidade de memória para o *bucket* padrão, que é o primeiro *bucket* criado no Couchbase. Como um outro *bucket* será criado para o exemplo, o padrão não tem importância e pode até mesmo ser deletado após essa configuração inicial. Então, recomenda-se colocar a quantidade mínima de memória para ele. Para esclarecer, *bucket* é, para o Couchbase, o equivalente a um *schema* do MySQL. Normalmente, será criado um para cada aplicação.

Após essas definições, o painel de controle do Couchbase será exibido, como mostra a **Figura 1**. Nesse painel, na aba *Data Buckets*, crie um novo bucket com o nome *Exemplo* e tipo *Couchbase*, visto que será utilizado para o armazenamento de documentos (o tipo *membbase* serve para armazenar apenas chave-valor). Feito isso, as opções desse bucket ficarão visíveis ao clicar na seta exibida ao lado de seu nome. Clique, então, nessa seta, depois em *Edit* e digite “exemplo” no campo de *password*.



Por dentro do banco de dados NoSQL Couchbase – Parte 2

Esta será a senha usada no exemplo de acesso ao banco de dados.

Em um projeto Java, seja ele Maven ou Gradle, deve-se adicionar o seguinte artefato referente ao SDK do Couchbase: `com.couchbase.client:java-client:2.1.4`. Caso seja criado um projeto Java comum Java, deve-se baixar o arquivo JAR desse cliente, cujo endereço também encontra-se na seção [Links](#). Outra biblioteca utilizada aqui será a Joda Time, que é muito conhecida como substituta da API nativa de gerenciamento de data e tempo do Java. Neste exemplo, ela será utilizada para salvar informações importantes no documento, como o momento em que foi criado e o momento de sua última modificação. Para o Maven e o Gradle, basta adicionar o seguinte artefato: `joda-time:joda-time:2.8.1`.

Com o projeto criado e as dependências satisfeitas, o próximo passo é estabelecer uma conexão com o banco de dados, o que pode ser feito em uma classe contendo um método `main()` como o código da [Listagem 1](#).

Nesse código demonstramos a simples tarefa de abrir e fechar uma conexão com o banco de dados. Note que começamos criando uma conexão com um cluster passando uma lista de IPs conhecidos. Nesse caso, por ser um único servidor local, sem qualquer cluster, podemos chamar o método `create()` sem qualquer argumento.

Listagem 1. Conexão com o Couchbase.

```
Public class Exemplo {  
    static Cluster cluster;  
    static Bucket bucket;  
  
    public static void main(String[] args){  
        cluster = CouchbaseCluster.create(new String[]{"127.0.0.1"});  
  
        bucket = cluster.openBucket("Exemplo", "exemplo");  
  
        // Uso do bucket  
  
        cluster.disconnect();  
    }  
}
```

Porém, se tivéssemos um cluster com vários servidores conectados, deveríamos passar uma lista com os IPs de alguns deles na chamada desse método.

Com uma conexão aberta para o cluster, deve-se conectar ao bucket criado para nossa aplicação. Para isso, chama-se o método `openBucket()` do cluster passando como argumento o nome e a senha do bucket. Por fim, após seu uso, fechamos a conexão com o cluster, o que possibilita a liberação dos recursos utilizados e desencadeia no fechamento da conexão do bucket.

A partir do momento em que uma conexão é aberta, o objeto que será utilizado por todo o sistema para acessar o banco será o **bucket**. Deste modo, esse pode ser configurado como singleton no sistema, possibilitando que seja injetado pelo Spring ou outro framework de injeção de dependências. Por outro lado, o cluster é usado apenas para abrir conexões com buckets e desconectar de todos eles de uma só vez através do `disconnect()`, caso o sistema utilize mais de um. Portanto, o objeto `cluster` terá esse método invocado no shutdown do sistema para que os recursos sejam liberados.

A seguir, os principais métodos da classe `Bucket` são explicados:

- **AsyncBucket async()**: retorna uma versão assíncrona do Bucket. Com essa versão, todos os métodos comuns do Bucket estarão disponíveis, porém, eles retornam um **Observable** do RxJava, biblioteca que vem crescendo em adoção e cujo uso é recomendado. Como está fora do escopo deste artigo explicá-la, serão mostrados aqui apenas os métodos síncronos;
- **JsonDocument get(String id)**: retorna o documento cujo identificador seja igual ao passado como parâmetro. Caso tal documento não exista, `null` será retornado;
- **JsonDocument getAndLock(String id, int lockTime)**: funciona da mesma maneira que o método anterior, contudo, esse método é utilizado para a concorrência pessimista, explicada anteriormente. Assim, caso o documento seja encontrado, ele será retornado e bloqueado. Caso esse método seja invocado novamente enquanto o documento estiver bloqueado, uma



exceção será lançada. Apesar de permitir ajustar a quantidade de segundos que o documento ficará bloqueado, caso esse não seja desbloqueado manualmente, o valor máximo que o documento permanecerá bloqueado é de 30 segundos, mesmo que seja passado um valor maior;

- <D extends Document<?>> Boolean unlock(D document): desbloqueia manualmente um documento;
- JsonDocument getAndTouch(String id, int expiry): funciona de maneira similar ao método `get(String id)`, com a diferença que o tempo de expiração do documento é reajustado com o valor passado como parâmetro. Esse método pode ser utilizado para acessar a sessão de um usuário logado em um sistema web, por exemplo;
- Boolean touch(String id, int expiry): também utilizado para atualizar o tempo de expiração de um documento, mas sem retorná-lo;
- <D extends Document<?>> D insert(D document): insere um documento no banco de dados caso ainda não exista. Caso contrário, um erro será lançado. Em caso de sucesso, o mesmo documento será retornado com o valor de CAS atualizado. Assim, novas atualizações podem ser feitas no mesmo com concorrência otimista;
- <D extends Document<?>> D upsert(D document): funciona da mesma maneira que o método anterior, com a diferença que, caso o documento já exista, ele será substituído;
- <D extends Document<?>> D replace(D document): é uma variante dos dois métodos anteriores. Diferentemente desses, esse terá sucesso apenas se o documento já existir. Caso contrário, um erro indicará que o mesmo não existe;
- <D extends Document<?>> D remove(D document): remove o documento do servidor. O documento retornado terá todas as propriedades vazias, restando apenas o identificador e o CAS;
- JsonDocument remove(String id): funcionamento idêntico ao método anterior. Pode ser utilizado quando apenas o identificador do documento estiver disponível;
- ViewResult query(ViewQuery query): faz uma busca por documentos através da query passada como parâmetro. O funcionamento desse tipo de busca será explorado mais adiante;
- JsonLongDocument counter(String id, long delta, long initial): cria ou atualiza um contador. Contudo, esse método armazena um número `Long` como valor da chave, ao invés de um documento JSON. Caso o identificador não exista, o contador receberá o valor inicial. Caso contrário, terá seu valor somado de delta. Essa operação é atômica, o que a torna segura para concorrência.

Vale notar que todas as operações que modificam um documento, vistas anteriormente, utilizam de concorrência otimista caso o valor de CAS esteja presente. Se houver diferença entre o valor de CAS passado pelo cliente e o valor do servidor, assume-se que tal documento foi alterado em paralelo e a exceção `CASMismatchException` é lançada para indicar o ocorrido.

Com a visão básica dos principais métodos de `Bucket`, o próximo passo é explorar a interface `Document` e sua principal implementação, `JsonDocument`, a qual usaremos na maior parte do tempo.

Começando pela interface, essa possui apenas quatro métodos, sendo todos eles do tipo getter para as propriedades básicas: `id()`, `content()`, `cas()` e `expiry()`. Seus nomes são sugestivos sobre o que cada um retorna, porém, vale analisar melhor o método `content()`.

Para entender o que o método getter para o conteúdo de um documento retorna, devemos observar que a interface `Document<T>` faz uso de generics para especificar o tipo de conteúdo. Essa interface tem como principal implementação a classe `JsonDocument`, que especifica `JsonObject` como tipo de conteúdo, o qual nada mais é do que um wrapper para `Map` que valida suas chaves e valores para assegurar a compatibilidade com um documento JSON. Além desta, outras implementações estão disponíveis, como a já citada `JsonLongDocument`, que armazena um `Long`, e outras equivalentes para cada tipo primitivo, como `JsonDoubleDocument`, `JsonBooleanDocument` e `JsonStringDocument`. Para armazenar conteúdo binário, existe ainda a `BinaryDocument`.

Sobre o tipo de documento mais comum, `JsonDocument`, é importante observar seus *factory methods*, os quais serão utilizados para criar instâncias do mesmo. Esses métodos possuem o nome `create()` e algumas combinações de argumentos, sendo eles: `id`, `content`, `cas` e `expiry`. Além desses, também existem métodos que permitem a cópia de documentos definidos com o nome `from()` e os mesmos argumentos anteriores, com a adição de `JsonDocument`. A função desses métodos é permitir a criação de documentos clones, naturalmente com identificadores diferentes, o que pode ser útil para permitir a cópia de registros pelo usuário para que esse faça as modificações que desejar.

Com essa simples API já é possível realizar todas as operações básicas com documentos. No entanto, ainda falta o método de conversão de entidades para documentos JSON.



Por dentro do banco de dados NoSQL Couchbase – Parte 2

Como mencionado na discussão sobre modelagem de documentos, será utilizado um esquema de mapeamento com controle de versão. Vejamos a explicação e o código de exemplo desse método no tópico a seguir.

Mapeadores de documentos

O mapeamento de objetos se dará através de classes simples, expostas adiante. Antes de mostrar esse código, no entanto, serão apresentadas as entidades de exemplo. Vale notar que todas elas implementarão uma classe abstrata, chamada **Entity**, com o intuito de tornar disponível em todas as entidades as seguintes propriedades: **String id**, **DateTime createdAt**, **DateTime lastModified** e **Long revision**. Com elas, armazenamos o identificador do documento, o momento em que o mesmo foi inserido, o momento em que foi modificado pela última vez e o valor do CAS.

Para exemplificar, serão criadas apenas duas entidades: **Task** e **Category**. A entidade **Task** representará uma tarefa a ser realizada e **Category** a encaixará em determinada categoria de tarefas. Tal aplicação será uma simples *TODO List*. Para torná-la o mais simples possível, ambas as entidades terão somente um campo, chamado **name**, para armazenar o nome da tarefa e da categoria, e a tarefa possuirá ainda uma referência para sua categoria, o que configura uma relação de um para muitos.

Com as entidades prontas, deve-se criar os mapeadores de documentos para as mesmas. A **Listagem 2** mostra a interface implementada por todos os mapeadores. Essa utiliza generics para especificar qual tipo de entidade tal mapeador tratará, especificando seus dois métodos de conversão. Além disso, quatro campos que estarão presentes em todos os documentos são especificados através de constantes. E para facilitar o trabalho das implementações dessa interface, a classe **AbstractDocumentMapper**, mostrada na **Listagem 3**, implementa os métodos do mapeador de forma a tratar os metadados do documento, como o identificador, momento de criação e última modificação e o valor do CAS.

O primeiro ponto a se notar no código de **AbstractDocumentMapper** é o uso da classe **ISODateFormat**, da biblioteca Joda Time. O objetivo com isso é que todos os timestamps sejam salvos no formato padrão definido pela ISSO 8601. Tal formato tem a vantagem de ter um ordenamento natural das datas, de forma que uma ordenação simples de Strings colocaria os timestamps em ordem.

Listagem 2. Interface que especifica os métodos de um mapeador, bem como especifica nomes de campos para os metadados.

```
public interface DocumentMapper<E extends Entity> {  
  
    String TYPE = "type";  
    String TYPE_VERSION = "typeVersion";  
    String CREATED_AT = "createdAt";  
    String LAST_MODIFIED = "lastModified";  
  
    E getEntity(JsonDocument document);  
    JsonDocument getDocument(E entity);  
}
```

Listagem 3. Classe abstrata criada com o propósito de ser superclasse de todos os mapeadores.

```
public abstract class AbstractDocumentMapper<E extends Entity> implements DocumentMapper<E> {  
  
    protected DateTimeFormatter dateTimeFormatter;  
  
    protected AbstractDocumentMapper() {  
        this.dateTimeFormatter = ISODateTimeFormat.dateTime();  
    }  
  
    @Override  
    public JsonDocument getDocument(E entity) {  
        JsonObject properties = JsonObject.create();  
  
        String id = entity.getId();  
  
        Long rev = entity.getRevision();  
  
        properties.put(CREATED_AT, entity.getCreatedAt().  
            toString(dateTimeFormatter));  
  
        properties.put(LAST_MODIFIED, entity.getLastModified().  
            toString(dateTimeFormatter));  
  
        return JsonDocument.create(id, properties, rev != null ? rev : 0);  
    }  
  
    protected void fillMetadata(E entity,JsonDocument document){  
        entity.setId(document.id());  
        entity.setRevision(document.cas());  
        JsonObject properties = document.content();  
        entity.setCreatedAt(dateTimeFormatter.parseDateTime((String)  
            properties.get(CREATED_AT)));  
        entity.setLastModified(dateTimeFormatter.parseDateTime((String)  
            properties.get(LAST_MODIFIED)));  
    }  
}
```



Também vale notar que foi utilizada a versão do método `create()` da classe `JsonDocument` que espera um valor de CAS. Quando tal valor não está disponível, normalmente utilizariamos outra versão desse método. Porém, para facilitar, simplesmente passamos 0 quando o CAS for nulo, visto que esse é o valor padrão do Couchbase, equivalente a nulo. O restante da implementação dessa classe é clara e não necessita de maiores explicações.

No exemplo abordado, serão duas classes que herdam de `AbstractDocumentMapper`: `TaskDocumentMapperV1` e `CategoryDocumentMapperV1`. O sufixo V1 indica que essas classes são responsáveis por mapear a primeira versão de seus respectivos documentos. A lógica de versionamento será mostrada mais adiante. As **Listagens 4** e **5** mostram essas duas classes mapeadoras.

Como pode-se verificar, o código dos mapeadores é bem simples. Eles utilizam de métodos criados na superclasse para preencher e obter os metadados dos documentos e manipulam diretamente o conteúdo desses, que é armazenado em uma instância de `JsonObject`. E assim como é feito em um JSON comum, as propriedades são armazenadas nos pares chave-valor, tendo seus nomes como chaves. Dessa forma, utilizamos o método `put()` para preencher cada propriedade.

Deve-se notar o ajuste das propriedades `TYPE` e `TYPE_VERSION`, as quais são usadas para armazenar o tipo de documento e a versão do esquema. Na **Listagem 5**, o tipo é ajustado com o valor `category`, definido como constante na classe `CategoryDocumentMapper`, que ainda será mostrada. E por se tratar da primeira versão, ajustamos o campo de versão com o número inteiro 1. Também pode-se notar na **Listagem 4** que o identificador da categoria de determinada tarefa é armazenado em um campo do documento, caso a categoria exista. Isso cria o mapeamento de um para muitos, de maneira parecida com o método adotado em bancos de dados relacionais.

Para finalizar a lógica dos mapeadores, as **Listagens 6** e **7** mostram as implementações das classes `TaskDocumentMapper` e `CategoryDocumentMapper`. Ambas foram utilizadas anteriormente nas **Listagens 4** e **5**, onde suas constantes `DOC_TYPE` foram usadas para armazenar o tipo correto de seus documentos.

A lógica de versionamento implementada por essas classes é simples. Elas possuem um mapa cuja chave especifica uma versão do documento e o valor é um mapeador para determinada versão. Ao mapear um documento para uma entidade, a versão do documento é verificada para se obter o mapeador correto. O caminho contrário, em que a entidade será convertida em documento, é sempre feito com a versão atual. Quando uma nova versão de um documento é criada, a constante que especifica a versão atual será atualizada, e um mapeador, para tal versão, será criado para a mesma e adicionado ao mapa. Para manter a compatibilidade, versões anteriores do mapeador serão atualizadas. Isso porque esses deverão ler um documento em versão antiga e mapeá-lo para a versão mais nova da entidade, que poderá ter seus campos modificados.

Listagem 4. Classe mapeadora dos documentos de tarefa de versão 1.

```
public class TaskDocumentMapperV1 extends AbstractDocumentMapper<Task> {  
  
    public static final String NAME = "name";  
    public static final String CATEGORY_ID = "categoryId";  
  
    @Override  
    public Task getEntity(JsonDocument document) {  
        JsonObject properties = document.content();  
  
        String name = (String) properties.get(NAME);  
  
        Task task = new Task();  
        task.setName(name);  
  
        String categoryId = (String) properties.get(CATEGORY_ID);  
        if(categoryId != null){  
            task.setCategory(new Category(categoryId));  
        }  
  
        fillMetadata(task, document);  
  
        return task;  
    }  
  
    @Override  
    public JsonDocument getDocument(Task task) {  
        JsonDocument document = super.getDocument(task);  
  
        JsonObject properties = document.content();  
  
        properties.put(TYPE, TaskDocumentMapper.DOC_TYPE);  
        properties.put(TYPE_VERSION, 1L);  
  
        properties.put(NAME, task.getName());  
  
        Category category = task.getCategory();  
        if(category != null){  
            properties.put(CATEGORY_ID, category.getId());  
        }  
  
        return document;  
    }  
}
```

Listagem 5. Classe mapeadora dos documentos de categoria de versão 1.

```
public class CategoryDocumentMapperV1 extends AbstractDocumentMapper<Category> {  
  
    public static final String NAME = "name";  
  
    @Override  
    public Category getEntity(JsonDocument document) {  
        JsonObject properties = document.content();  
        String name = (String) properties.get(NAME);  
  
        Category category = new Category();  
        category.setName(name);  
  
        fillMetadata(category, document);  
  
        return category;  
    }  
  
    @Override  
    public JsonDocument getDocument(Category category) {  
        JsonDocument document = super.getDocument(category);  
  
        JsonObject properties = document.content();  
  
        properties.put(TYPE, CategoryDocumentMapper.DOC_TYPE);  
        properties.put(TYPE_VERSION, 1L);  
  
        properties.put(NAME, category.getName());  
  
        return document;  
    }  
}
```

Por dentro do banco de dados NoSQL Couchbase – Parte 2

Listagem 6. Implementação da lógica de versionamento dos mapeadores dos documentos de tarefa.

```
public class TaskDocumentMapper implements DocumentMapper<Task> {  
  
    public static final String DOC_TYPE = "task";  
  
    public static final Integer CURRENT_VERSION = 1;  
  
    private Map<Integer, DocumentMapper<Task>> mappers;  
  
    public TaskDocumentMapper(TaskDocumentMapperV1  
        taskDocumentMapperV1) {  
        this.mappers = new HashMap<>();  
        this.mappers.put(CURRENT_VERSION, taskDocumentMapperV1);  
    }  
  
    @Override  
    public Task getEntity(JsonDocument document) {  
        return mappers.get(((Number)document.content().get(TYPE_VERSION))  
            .intValue()).getEntity(document);  
    }  
  
    @Override  
    public JsonDocument getDocument(Task task) {  
        return mappers.get(CURRENT_VERSION).getDocument(task);  
    }  
}
```

Listagem 7. Implementação da lógica de versionamento dos mapeadores dos documentos de categoria.

```
public class CategoryDocumentMapper implements  
    DocumentMapper<Category> {  
  
    public static final String DOC_TYPE = "category";  
  
    public static final Integer CURRENT_VERSION = 1;  
  
    private Map<Integer, DocumentMapper<Category>> mappers;  
  
    public CategoryDocumentMapper(CategoryDocumentMapperV1  
        categoryDocumentMapperV1) {  
        this.mappers = new HashMap<>();  
        this.mappers.put(CURRENT_VERSION, categoryDocumentMapperV1);  
    }  
  
    @Override  
    public Category getEntity(JsonDocument document) {  
        return mappers.get(((Number)document.content().get(TYPE_VERSION))  
            .intValue()).getEntity(document);  
    }  
  
    @Override  
    public JsonDocument getDocument(Category category) {  
        return mappers.get(CURRENT_VERSION).getDocument(category);  
    }  
}
```

Também vale observar que esses mapeadores, responsáveis pelo versionamento de mapeadores, não implementam a classe **AbstractDocumentMapper**, mas sim a interface **DocumentMapper**. O objetivo é deixar transparente ao resto do sistema o mecanismo de mapeamento de documentos. Normalmente, essas instâncias serão criadas através de injeção de dependências e os utilizadores das mesmas não precisarão se preocupar com detalhes do versionamento.

Antes de partirmos para a demonstração do uso destas classes, é preciso entender que elas devem ser combinadas para se obter objetos completos. Ao observarmos a **Listagem 4**, por exemplo,

notamos que o objeto criado a partir de um documento de tarefa possui uma instância vazia de **Category**, com apenas o identificador preenchido. Para preencher esse objeto com uma categoria completa, deve-se pesquisar por seu documento, utilizar o mapeador de categoria e então ajustá-la com um objeto completo. A responsabilidade de fazer essa combinação geralmente fica por conta de um DAO, que irá interagir com documentos e usar os mapeadores para transformá-los em objetos.

Finalizando essa seção sobre mapeamento de documentos, a **Listagem 8** mostra o método de inserção e atualização de entidades. Esse método é utilizado na **Listagem 9**, a qual demonstra



Listagem 8. Código de inserção/atualização de entidades. Normalmente estaria em um DAO.

```
private static <T extends Entity> void upsert(DocumentMapper<T> documentMapper, T entity) {
    if(entity.getCreatedAt() == null){
        entity.setCreatedAt(new DateTime());
    }
    entity.setLastModified(new DateTime());
    JsonDocument document = documentMapper.getDocument(entity);
    JsonDocument updatedDocument = bucket.upsert(document);
    entity.setRevision(updatedDocument.cas());
}
```

Listagem 9. Código que demonstra o uso básico do Couchbase.

```
public static void main(String[] args){
    // Abre conexão com o banco de dados
    cluster = CouchbaseCluster.create(new String[]{"127.0.0.1"});
    bucket = cluster.openBucket("Exemplo", "exemplo");

    // Instancia os mapeadores
    CategoryDocumentMapper categoryDocumentMapper = new CategoryDocumentMapper(new CategoryDocumentMapperV1());
    TaskDocumentMapper taskDocumentMapper = new TaskDocumentMapper(new TaskDocumentMapperV1());

    // Cria as entidades com identificadores UUID aleatórios
    Category category = new Category();
    category.setId(UUID.randomUUID().toString());
    category.setName("Categoria 1");

    Task task = new Task();
    task.setId(UUID.randomUUID().toString());
    task.setName("Tarefa 1");
    task.setCategory(category);

    // Insere as entidades recém criadas
    upsert(categoryDocumentMapper, category);
    upsert(taskDocumentMapper, task);

    // Realiza alterações nas entidades
    category.setName("Categoria 1 modificada");
    task.setName("Tarefa 1 modificada");

    // Persiste as alterações no banco
    upsert(categoryDocumentMapper, category);
    upsert(taskDocumentMapper, task);

    // Recupera o objeto completo de tarefa com sua respectiva categoria
    JsonDocument taskDocument = bucket.get(task.getId());
    task = taskDocumentMapper.getEntity(taskDocument);
    JsonDocument categoryDocument = bucket.get(task.getCategory().getId());
    category = categoryDocumentMapper.getEntity(categoryDocument);
    task.setCategory(category);

    // Remove os registros do banco
    bucket.remove(category.getId());
    bucket.remove(task.getId());

    // Finaliza a conexão
    cluster.disconnect();
}
```

todo o processo de inserção, modificação, atualização, obtenção completa e, por fim, remoção das entidades de categoria e tarefa.

O código da **Listagem 8** é bem simples. Ele verifica se a entidade já possui uma data de criação e, caso não possua, cria uma com a data atual. Por outro lado, a data de modificação é sempre atualizada. Depois disso, o mapeador é utilizado para converter o objeto em documento e utiliza-se o método **upsert()**, que insere ou atualiza um documento. Assim, o documento retornado por esse método tem sua propriedade de CAS extraída e ajustada na entidade. Isso permite que futuras atualizações da entidade sejam possíveis evitando um **CASMismatchException**.

Da mesma forma, o código da **Listagem 9** não possui complicações. Deve-se notar a combinação dos mapeadores para se obter um objeto completo. Primeiramente, o documento de tarefa é obtido e convertido em objeto. Então, busca-se sua categoria utilizando o identificador do objeto incompleto de categoria. Por fim, o documento de categoria é convertido em sua entidade e passado para o objeto de tarefa. Isso é algo que normalmente estaria dentro de um DAO, mas para fins de facilitar a visualização, foi colocado aqui. O restante do código é autoexplicativo. Basicamente, altera-se as entidades, as armazena no banco utilizando o método apresentado na **Listagem 8** e depois as remove através do método **remove()**, que recebe o identificador do documento como parâmetro.

Com o código demonstrado é possível fazer todas as operações básicas com documentos, desde que se tenha seu identificador. Mas normalmente essa pré-condição não é satisfeita. Ao invés disso, um sistema normalmente lista entidades de acordo com determinadas condições para que o usuário escolha o que fazer com alguma dessas entidades. Em um banco de dados relacional, isso é feito com queries SQL. A seguir, veremos como buscar registros no Couchbase.

Utilizando Views

A forma de busca por dados utilizada no Couchbase é algo novo e completamente diferente das tradicionais consultas SQL. Assim, ao invés de utilizar uma linguagem de busca de dados como em bancos relacionais, o Couchbase utiliza da lógica de MapReduce para indexar os dados em Views, as quais são uma forma de organização de um index de busca criado no banco e permitem extração, filtro, agregação e busca de informação.

O processo de criação de uma View se dá pela configuração de uma ou duas funções. A primeira é a função de map, que filtra entradas e pode extrair informação. Seu resultado final é uma lista ordenada de chave-valor, chamada de index, que é armazenada em disco e atualizada de forma incremental à medida que os documentos são atualizados. Opcionalmente, pode-se prover uma segunda função, chamada reduce, que tem o objetivo de somar, agrregar ou realizar outros cálculos sobre as informações.

Quando nos referimos a funções, estamos mencionando rotinas programadas em JavaScript, as quais serão executadas em cada um dos documentos. Essas rotinas de map e reduce das Views são armazenadas como strings em documentos especiais JSON,

Por dentro do banco de dados NoSQL Couchbase – Parte 2

conhecidos como design documents, e então esses são associados ao bucket. Vale ressaltar que cada design document pode armazenar uma ou mais Views.

Depois que a View é criada, pode-se utilizá-la para buscar informações. Isso é feito através de seu index, o qual ordena de forma crescente as chaves emitidas para cada documento. Assim, é possível fazer buscas por chaves iguais a determinado valor, que iniciem com algum valor, entre outros. Essas chaves podem ser emitidas com qualquer campo de um documento. Então, se for desejada uma busca por contatos cujo número de telefone se inicie com o DDD 62, uma View será criada filtrando apenas documentos de contatos e usando o número de telefone como chave. Por fim, uma busca nessa View com chaves que iniciem com 62 será executada e os contatos serão obtidos.

Para entender melhor o funcionamento desse tipo de busca, podemos começar criando uma View para buscar todas as tarefas do exemplo anterior. Para isso, entre no console web do Couchbase novamente, abra a aba de Views e selecione o bucket *Exemplo*, criado anteriormente. Nessa tela, nota-se que existem duas abas: *Development Views* e *Production Views*. Essa diferença existe para que, ao criar Views no banco de dados, esse não tenha quedas de performance enquanto o processo de index é feito. Portanto, uma *Development View* trabalhará apenas com um subconjunto dos dados, a não ser que seja requisitado o contrário.

Como estamos no momento de criação da View, devemos utilizar a aba *Development Views* e clicar em *Create Development View*. Em seguida, preencha tanto o campo *Design Document Name* quanto o campo *View Name* com *all_tasks* e clique em *Save*. Note que o *Design Document Name* é prefixado com *_design/dev_*, sem opção de mudança, visto que é um padrão fixado pelo Couchbase. Com os campos preenchidos como especificado, o design document será criado com a linguagem JavaScript e uma View chamada *all_tasks* será criada junto a ele. Clicando em *Edit* na View recém-criada, uma tela apresentará o código da View em duas partes: Map e Reduce.

Em Map está presente um código simples, sendo essa uma função JavaScript que recebe como parâmetro o documento e os

metadados do mesmo. Vemos também uma chamada para o método *emit()*, que basicamente adiciona ao index uma chave e valor. Nesse código de exemplo, apenas o identificador do documento é emitido com um valor nulo. Isso porque o valor aqui é utilizado apenas pela função Reduce, que se encontra vazia, visto que é opcional.

Caso essa View seja mantida dessa forma, ela simplesmente buscará por todos os documentos e os ordenará pelo identificador. Além de não ser o que desejamos, também não faz sentido essa ordenação, visto que geramos os identificadores com UUID aleatório no código da [Listagem 9](#). Uma ordenação mais eficaz, por exemplo, seria pela data de criação.

Deste modo, como desejamos listar todas as tarefas e ordená-las por data de criação, utilizaremos o código mostrado na [Listagem 10](#) como função Map.

Listagem 10. Função Map para listar todas tarefas ordenadas por data de criação.

```
function (doc, meta) {
  if(meta.type == "json" && doc.type == "task"){
    emit(doc.createdAt);
  }
}
```

O funcionamento dessa função é simples. Primeiro, certifica-se que o documento atual é do tipo JSON, lembrando que ele poderia ser binário, armazenar Strings, ou outros tipos mencionados anteriormente. Então, verificamos se o tipo do documento, armazenado no campo *type*, é igual a **task**, lembrando que esse campo foi criado em nossos mapeadores. Por fim, emitimos o campo *createdAt*, que tem como valor a data de criação em formato ISO. Como não será utilizada uma função de Reduce, o valor passado para a função *emit()* é nulo. Em JavaScript, um argumento nulo pode simplesmente ser ignorado.

Para testar essa View pode-se executar parte do código mostrado na [Listagem 9](#), retirando o trecho que remove os documentos



inseridos. Assim, ao clicar em *Show Results*, será apresentada uma tabela com a Key igual à data de criação e o Value igual a **null**. Cada linha apresenta ainda um link para visualizarmos o documento. Com nossa primeira View criada e testada, pode-se voltar à aba *Development Views*, localizar o design document *all_tasks* e clicar em *Publish*. Feito isso, esse passará para a aba de produção com o nome *_design/all_tasks*. Observe que o prefixo *_design/dev_* foi alterado automaticamente.

Para demonstrar a utilização dessa View pelo sistema, foi criada a **Listagem 11**. Nela, implementamos uma **ViewQuery** que ordenará as chaves de forma decrescente, pulará os primeiros 0 registros e limitará em 10 os registros retornados, demonstrando assim o uso de paginação com os métodos **skip()** e **limit()**. Traduzindo essa query, obteremos as dez últimas tarefas criadas no sistema, sendo elas ordenadas em ordem decrescente de criação, ou seja, as mais novas são apresentadas primeiro.

Ainda observando esse código, percebemos que a classe **ViewResult** retornada pelo método **query()** do bucket é um iterator que contém as linhas obtidas como resultado. Cada uma dessas linhas tem o identificador do documento, a chave e valor emitidos, além do documento representado. E assim como feito antes, o documento é convertido em objeto utilizando um mapeador.

Dessa forma, para obter a categoria de cada tarefa deve-se fazer uma busca pelo identificador de cada uma. Para que isso se torne desnecessário, será criada outra View que fará algo parecido com o conhecido JOIN do SQL.

O processo de criação dessa View será o mesmo, mas a chamaremos de ‘tasks_with_categories’. O código da função Map dela é apresentado na **Listagem 12**.

Listagem 11. Código que demonstra o uso da View *all_tasks* para listar as últimas dez tarefas.

```
ViewQuery viewQuery = ViewQuery.from("all_tasks", "all_tasks")
    .descending()
    .skip(5)
    .limit(10);
ViewResult viewResult = bucket.query(viewQuery);
for(ViewRow row : viewResult){
    JsonDocument document = row.document();
    task = taskDocumentMapper.getEntity(document);
}
```

Listagem 12. Função Map para listar todas tarefas ordenadas por data de criação, com suas categorias.

```
function (doc, meta) {
    if(meta.type == "json"){
        if(doc.type == "category"){
            emit([meta.id]);
        }
        if(doc.type == "task"){
            emit([doc.categoryId, doc.createdAt]);
        }
    }
}
```

A parte a se notar dessa técnica é a seguinte: chaves também podem ser emitidas como arrays, como pode ser visto nas invocações de **emit()**. Quando isso é feito, as chaves continuarão sendo ordenadas naturalmente, mas utilizando todos os elementos do array de forma que o elemento seguinte é considerado apenas em caso de empate na ordenação baseada no elemento anterior. Voltando à View criada, essa emitirá todos os documentos de categoria e tarefa. Os documentos de categoria terão como chave um array de elemento único com o identificador de cada documento, enquanto os documentos de tarefa terão um array com o identificador da categoria e a data de criação da tarefa como chave.

Quando essas chaves são ordenadas, as categorias e suas tarefas ficam juntas, visto que suas chaves começam com o identificador da categoria. Com essas chaves emitidas, o registro da categoria aparece antes das tarefas, já que sua chave possui apenas um elemento, e logo após a categoria, todas as tarefas relacionadas são dispostas, ordenadas por sua data de criação.

Com essa ordenação, a responsabilidade do código Java que utilizará essa View será de diferenciar os documentos de categoria dos de tarefa. Assim, ao iterar pelas linhas obtidas como resultado, sempre que uma categoria for encontrada ela será guardada em uma variável temporária e todas as tarefas subsequentes serão admitidas como da categoria corrente e terão seu valor de categoria ajustado com ela. O código com essa lógica implementada pode ser visualizado na **Listagem 13**.

Outro uso para essa View seria a busca por todas as tarefas de uma categoria já conhecida. Isso poderia ser feito com os métodos **startKey()** e **endKey()** de **ViewQuery**, como mostrado na **Listagem 14**. Esses métodos são usados para especificar um intervalo de chaves dos documentos que serão encontrados. Assim, uma chave que, em ordenação natural, seja superior à **startKey()** e inferior à **endKey()** terá seu documento listado.

No entanto, para que a técnica de listagem das tarefas de determinada categoria funcione, devemos saber que um elemento vazio de um array toma precedência sobre qualquer valor, quando tratamos de ordenação natural. Então, utiliza-se como chave inicial um array com um único elemento, a chave da categoria procurada, e a chave final será um array cujo primeiro elemento é a chave da categoria e o segundo é a **String** especial “\uefff”. Essa **String** representa um único caractere UTF-8, que é grande o suficiente para garantir que nenhum valor do segundo elemento das chaves seja maior em uma ordenação alfabética. Traduzindo tudo isso, buscamos por chaves que começem com o identificador da categoria e aceitamos qualquer valor como segundo elemento da chave, desde vazio até um valor qualquer.

Essas técnicas podem ser combinadas com outras para se buscar registros através de Views. Para isso, no entanto, deve ser feito um bom planejamento com o intuito de reutilizar essas Views na medida do possível, pois uma criação excessiva delas pode gerar um impacto bastante negativo na performance do banco de dados.

A outra função de uma View é fazer resumo de dados. Para isso, recomenda-se o uso da função **Reduce**, que não será apresentada

Por dentro do banco de dados NoSQL Couchbase – Parte 2

nesta artigo, mas que é simples de ser implementada. Na seção **Links** estão disponíveis guias para um estudo mais aprofundado sobre Views e isso inclui sua função Reduce.

Listagem 13.

Código utilizado para se obter todas as tarefas com categorias.

```
ViewQuery viewQuery = ViewQuery.from("tasks_with_categories", "tasks_with_categories");
ViewResult viewResult = bucket.query(viewQuery);
Category currentCategory = null;
List<Task> tasks = new ArrayList<>();
for(ViewRow row : viewResult){
    JsonDocument document = row.document();
    String docType = (String) document.content().get(DocumentMapper.TYPE);
    if(CategoryDocumentMapper.DOC_TYPE.equals(docType)){
        currentCategory = categoryDocumentMapper.getEntity(document);
    }
    else if(TaskDocumentMapper.DOC_TYPE.equals(docType)){
        Task currentTask = taskDocumentMapper.getEntity(document);
        currentTask.setCategory(currentCategory);
        tasks.add(currentTask);
    }
}
```

Listagem 14.

Parâmetros de busca para se obter tarefas de determinada categoria.

```
viewQuery.startKey(JsonArray.from(categoryId));
viewQuery.endKey(JsonArray.from(categoryId, "\ueeff"));
```

Uma boa notícia para aqueles que ficaram receosos com essa forma de pesquisar dados, em versões mais novas do Couchbase é encontrada uma nova ferramenta: a N1QL. Conhecida como o “SQL para JSON”, possui uma linguagem praticamente idêntica ao SQL. Ela não foi abordada neste artigo porque é relativamente nova e vem evoluindo nas últimas versões do Couchbase.

Para suprir as necessidades de grandes aplicações como o Facebook e as soluções do Google, diversas opções de bancos de dados foram surgindo, resultando na criação do acrônimo NoSQL para defini-las. Dentre elas, bancos como o Apache Cassandra e BigTable foram criados e por fim abertos ao público.

Nessa linha de novos bancos, o primeiro a se destacar com estrutura de dados baseada em documentos foi o MongoDB, o qual permanece sendo o mais utilizado nos dias de hoje. Porém, com a fusão do CouchDB com o Membase surgiu um forte concorrente, que foi apresentado neste artigo: o Couchbase.

Além de demonstrar melhor performance que a concorrência em testes de benchmark, o Couchbase facilita a criação de novos tipos de aplicação: aquelas que requerem funcionamento offline. Um grande exemplo disso são aplicações mobile que têm o conteúdo fornecido por um servidor, mas que deverão trabalhar frequentemente sem acesso à internet. Para essas existe a combinação do Couchbase Server com o Couchbase Lite interligados pelo Sync Gateway. Pelo lado do servidor, deve-se simplesmente configurar o Sync Gateway para criar um meio de acesso ao Couchbase Server. Já pelo lado da aplicação mobile, utiliza-se o Couchbase Lite para armazenar, recuperar e sincronizar os dados com o servidor. Sua programação é muito semelhante à

encontrada neste artigo e os dados são sincronizados sem grande esforço do desenvolvedor.

Mesmo quando aplicações mobile não estão em discussão, o Couchbase prova ser um ótimo banco de dados para diversas situações. Um exemplo muito interessante para ele seriam as lojas virtuais, afinal, em épocas de grandes promoções, como a famosa Black Friday, podemos notar que grande parte dessas lojas acaba não aguentando o fluxo de usuários, chegando a cair ou apresentar problemas. Com o uso de Couchbase torna-se possível um escalonamento horizontal temporário na camada de banco de dados. Assim, a loja virtual poderia operar com um único nó em datas comuns e fazer a adição de nós quando um grande fluxo de acesso for previsto.

Enfim, não só o Couchbase, mas as soluções NoSQL em geral ampliam o horizonte de possibilidades. E algumas delas foram apresentadas neste artigo, o qual também teve como objetivo ensinar o leitor a utilizar esse banco de dados de grande potencial.

Autor



Fernando Henrique Fernandes de Camargo

fernando.camargo.ti@gmail.com

É desenvolvedor Java EE, Android e Grails. Atualmente é mestreando em Engenharia de Computação na UFG. Desenvolve em Java desde 2009 e para Android desde 2012. Possui a certificação OCJP6, artigos publicados na Easy Java Magazine e na Java Magazine, além de palestras e minicursos apresentados em eventos.



Links:

Artigo sobre NoSQL.

www.couchbase.com/nosql-resources/what-is-no-sql

Comparação entre Couchbase e CouchDB.

www.couchbase.com/couchbase-vs-couchdb

Guia de desenvolvimento do Couchbase.

docs.couchbase.com/developer/dev-guide-3.0

Artigo sobre concorrência otimista e pessimista.

blog.couchbase.com/optimistic-or-pessimistic-locking-which-one-should-you-pick

Instruções de instalação do Couchbase.

docs.couchbase.com/admin/admin/install-intro.html

Guia de desenvolvimento do Java SDK.

docs.couchbase.com/developer/java-2.1/java-intro.html

JAR com SDK do Couchbase.

mvnrepository.com/artifact/com.couchbase.client/java-client/2.1.4

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Programador Java: Por onde começar?

Descubra nesse vídeo como entrar na carreira Java com o pé direito!



DEVMEDIA

Como usar o Apache Cassandra em aplicações Java EE - Parte 2

Aprenda neste artigo como implementar e executar comandos no Cassandra seguindo as melhores práticas através de uma aplicação Java EE

ESTE ARTIGO FAZ PARTE DE UM CURSO

Com a explosão da Internet que vivenciamos atualmente, muitos novos desafios estão surgindo para a indústria de software, desde a preocupação com a escalabilidade das aplicações, que agora possuem milhões de usuários, até mesmo a melhoria contínua da usabilidade desses sistemas, dado que os usuários se tornam cada vez mais exigentes e desejam mais facilidades.

Um desses desafios trata-se do armazenamento e processamento da imensa massa de dados gerada pelos diversos serviços disponíveis. Por conta dela, novas tecnologias de banco de dados emergiram nos últimos anos para atender uma série de requisitos que o modelo mais tradicional (Relacional) não conseguiu suprir. A esse novo movimento de tecnologias de bancos de dados deu-se o nome de NoSQL.

Diante da relevância do tema, apresentamos na primeira parte desse artigo um dos bancos NoSQL mais renomados desse ecossistema, o Apache Cassandra. Para

Fique por dentro

Veremos neste artigo uma visão prática da utilização do Apache Cassandra seguindo as melhores recomendações do mercado. Assim, para quem conhece o Cassandra num nível apenas teórico, poderá aqui “colocar a mão na massa” e dar os primeiros passos neste que é um dos bancos de dados NoSQL mais empregados.

Ademais, tudo é feito dentro do contexto de uma aplicação Java EE, demonstrando como essas duas tecnologias podem ser integradas. A aplicação de exemplo utilizará o driver da DataStax para se comunicar com o Cassandra, WildFly 9, PrimeFaces 5.3, Cassandra 2.2, além de outras tecnologias.

isso, foram abordados em detalhes vários aspectos da arquitetura do Cassandra, fornecendo ao leitor um embasamento teórico fundamental para o aprendizado prático desse banco de dados. Além disso, demos início à implementação de uma aplicação Java EE, onde detalhamos a preparação do ambiente de desenvolvimento, a configuração do projeto, a apresentação do driver Cassandra, uma introdução ao DevCenter e o desenvolvimento inicial da integração entre Cassandra e Java EE.

Nesta segunda etapa do artigo, iremos aprofundar a parte prática e evoluir a aplicação de modo que ela se torne funcional já com quase todas as funcionalidades propostas. Ao final o leitor será

capaz de executar comandos no Cassandra via Java, tanto usando CQL diretamente quanto usando a API object-mapping. O leitor também irá adquirir um entendimento básico sobre modelagem de dados no Cassandra, que como dito na primeira parte do artigo, é bem diferente da modelagem relacional.

Para tanto, evoluiremos nossa aplicação adicionando funcionalidades para executar comandos CQL de forma a seguir as recomendações do driver Cassandra, bem como demonstraremos a maneira adequada de utilizar tais funcionalidades. Além disso, serão abordados os novos recursos do PrimeFaces para criar páginas responsivas, a API de validação client-side dessa biblioteca, algumas features do CDI, como o uso de qualifiers e eventos, alguns recursos do DeltaSpike, entre outros. Todas essas opções serão utilizadas de forma a facilitar a implementação da aplicação de exemplo, bem como demonstrar a integração de todas essas tecnologias.

Evoluindo o WebShelf

Na primeira parte do artigo iniciamos o desenvolvimento do WebShelf: uma aplicação que possibilita aos seus usuários manter uma prateleira de livros online ao estilo do Amazon Shelfari. Até o momento, as principais configurações do projeto já foram demonstradas e explicadas, o que nos possibilita agora focar mais nas funcionalidades da aplicação.

Elaborando a página responsiva de Cadastro de Usuário com PrimeFaces

Como na primeira parte foi criado o template padrão (*default.xhtml*) das páginas JSF para a aplicação WebShelf, podemos agora criar a tela de Cadastro de Usuário. A **Listagem 1** traz essa implementação.

De modo simples, essa página define os dois parâmetros que o template precisa através da tag **ui:param**. Em seguida, adiciona o seu conteúdo no template através da tag **ui:define**.

Para deixar a tela mais responsiva envolvemos todo o conteúdo da página numa **div** que tem a classe **ui-fluid** do PrimeFaces. Através dessa classe diversos componentes deste framework são renderizados de forma responsiva.

Ainda pensando na responsividade, utilizamos o componente **p:panelGrid** (não confundir com o componente padrão do JSF **h:panelGrid**) em conjunto com o Grid CSS, ambos do PrimeFaces. O Grid CSS é um layout leve que produz uma interface responsiva para celulares, tablets e desktops e está presente no PrimeFaces desde sua versão 5.1, possibilitando ao desenvolvedor dividir a tela em 12 colunas de mesmo tamanho.

O **p:panelGrid**, ao ser configurado com o layout grid, irá fazer uso do Grid CSS para definir o tamanho de cada uma das suas colunas. Isso é configurado através do atributo **columnClasses**, que define uma classe CSS indicando quantas colunas do Grid CSS elas devem ocupar (**ui-grid-col-1**, **ui-grid-col-4**, **ui-grid-col-7**) totalizando as 12 colunas que preenchem a tela. No nosso exemplo, a primeira coluna do **p:panelGrid** irá ocupar 1 coluna do Grid CSS (**ui-grid-col-1**), a segunda coluna irá ocupar 4 colunas do Grid CSS (**ui-grid-col-4**) e a terceira 7 colunas do Grid CSS. Caso você precise fazer um arranjo diferente, é interessante saber que existem 12 classes **ui-grid-col-***, indo de **ui-grid-col-1** até **ui-grid-col-12**.

Listagem 1. Tela de cadastro de Usuário (*public/insertUser.xhtml*).

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:p="http://primefaces.org/ui"
 template="/WEB-INF/templates/default.xhtml">

<ui:param name="mainContentTitle" value="Cadastro de Usuário"/>
<ui:param name="renderedMenuBar" value="false"/>

<ui:define name="mainContent">
<h:form prependId="false">
<div class="ui-fluid">
<p:panelGrid columns="3" layout="grid"
 columnClasses="ui-grid-col-1,ui-grid-col-4,ui-grid-col-7"
 styleClass="ui-panelgrid-blank">

<p:outputLabel for="userName" value="Nome"/>
<p:inputText id="userName" value="#{userController.user.name}"/>
<p:message for="userName"/>

<p:outputLabel for="userLogin" value="Login"/>
<p:inputText id="userLogin" autocomplete="false"
 value="#{userController.user.login}"/>
<p:message for="userLogin"/>

<p:outputLabel for="userPassword" value="Senha"/>
<p:password id="userPassword" autocomplete="false"
 match="userPasswordConfirmation"
 value="#{userController.password}"/>
<p:message for="userPassword"/>

<p:outputLabel for="userPasswordConfirmation"
 value="Confirmar Senha"/>
<p:password id="userPasswordConfirmation" autocomplete="false"
 value="#{userController.password}"/>
<p:message for="userPasswordConfirmation"/>

<p:commandButton value="Salvar" action="#{userController.insert}"
 ajax="false" validateClient="true"/>

</p:panelGrid>
</div>
</h:form>
</ui:define>
</ui:composition>
```



Como usar o Apache Cassandra em aplicações Java EE - Parte 2

Para encerrar, temos um **p:commandButton** para invocar o método que irá gravar o usuário no banco de dados. Esse botão irá usar a validação no lado do cliente oferecida pelo PrimeFaces (**validateClient="true"**). Dessa forma, erros simples de validação como tamanho máximo do campo e campo obrigatório podem ser alertados antes de qualquer requisição ser enviada para o servidor, evitando *roundtrips* desnecessários e melhorando a performance da aplicação. Para que essa feature seja usada, é necessário ativá-la através do parâmetro **primefaces.CLIENT_SIDE_VALIDATION** no *web.xml*, como pode ser visto na **Listagem 5** da parte 1, publicada na edição anterior.

Desenvolvendo o controller de Usuário

Implementada a página web do Cadastro de Usuário, na **Listagem 2** criaremos o *controller* que irá intermediar a camada de negócio com a UI.

Como podemos notar, **UserController** é um bean CDI com escopo View. Esse escopo é obtido através da classe **javax.faces.view.ViewScoped**, presente a partir do JSF 2.2, e não deve ser confundida com a antiga classe **javax.faces.bean.ViewScope**, que tem vários problemas, como pode ser visto em *The benefits and pitfalls of ViewScoped* (veja a seção **Links**).

A anotação **@Named** permite que o bean possa ser acessado por um nome específico, por exemplo, em telas JSF através de Expression Language. Como não foi especificado nenhum nome no atributo **value**, então o nome do bean passa a ser o nome da classe com a primeira letra minúscula: **userController**.

Já o atributo **messages** usa a classe **JsfMessage** do DeltaSpike, uma biblioteca que fornece diversas extensões para se trabalhar com Java EE e que nasceu da junção de outros players (JBoss Seam e Apache CODI). A classe **JsfMessage** fornece uma maneira elegante e simples de adicionar mensagens JSF no contexto atual. Para isso, basta criar uma interface como na **Listagem 3**.

Preparando CassandraCluster para suportar object-mapping

Antes de implementar a lógica de negócios, será necessário adicionar alguns métodos na classe **CassandraCluster** para que possamos, de fato, executar comandos no Cassandra, visto que anteriormente havíamos definido apenas alguns atributos. O primeiro deles é o método **mapper()**, apresentado a seguir. Sua função será criar uma instância da classe **Mapper** que irá possibilitar o uso da API *object-mapping* do driver DataStax.

```
public <E> Mapper<E> mapper(Class<E> entityClazz){  
    return mappingManager.mapper(entityClazz);  
}
```

Este método simplesmente delega sua execução para o método **MappingManager.mapper()** e como parâmetro recebe qualquer classe anotada com **@Table**, como é o caso de **User**.

O seu retorno é uma instância da classe **Mapper** parametrizada com o mesmo tipo da classe do parâmetro **entityClazz**, e dessa forma, irá prover operações como busca, inserção e deleção de

Listagem 2. Código do controller de usuário (webshelf-web).

```
package br.com.devmedia.webshelf.controller;  
  
import java.io.Serializable;  
  
import javax.faces.view.ViewScoped;  
import javax.inject.Inject;  
import javax.inject.Named;  
  
import org.apache.deltaspike.jsf.api.message.JsfMessage;  
import org.hibernate.validator.constraints.NotBlank;  
  
import br.com.devmedia.webshelf.model.User;  
import br.com.devmedia.webshelf.service.UserBean;  
import br.com.devmedia.webshelf.util.Messages;  
  
@Named  
@ViewScoped  
public class UserController implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
  
    @Inject  
    private UserBean userBean;  
  
    @Inject  
    private JsfMessage<Messages> messages;  
  
    private User user = new User();  
  
    @NotBlank(message="Senha: não pode está em branco.")  
    private String password;  
  
    public User getUser() {  
        return user;  
    }  
  
    public String getPassword() {  
        return password;  
    }  
  
    public void setPassword(String password) {  
        this.password = password;  
    }  
  
    public String insert() {  
        user.setClearPassword(password);  
        this.userBean.insertUser(this.user);  
        messages.addInfo().insertUserSuccess();  
        return "login.xhtml?faces-redirect=true";  
    }  
}
```

Listagem 3. Classe de mensagens (webshelf-business).

```
package br.com.devmedia.webshelf.util;  
  
import org.apache.deltaspike.core.api.message.MessageBundle;  
import org.apache.deltaspike.core.api.message.MessageTemplate;  
  
@MessageBundle  
public interface Messages {  
  
    @MessageTemplate("Agora é só informar os dados de login e iniciar a sua  
    prateleira de livros online!")  
    String insertUserSuccess();  
  
    @MessageTemplate("Login/Senha inválido.")  
    String invalidCredentials();  
}
```

registros na tabela correspondente da entidade sem que seja necessário escrever o CQL diretamente.

Implementando o cache de PreparedStatements em CassandraCluster

Como dito na seção “Regras de utilização do driver DataStax”, se você perceber que irá executar um statement de forma repetida, recomenda-se que essas instruções sejam feitas através de **PreparedStatement**. Além disso, as instâncias dessa classe precisam ser mantidas num cache para evitar que o mesmo CQL seja preparado mais de uma vez, o que pode gerar problemas de performance.

Para implementar tais recomendações utilizamos no nosso exemplo um **Map de PreparedStatements**, onde o **Map** será instantiado uma única vez dentro da classe **CassandraCluster** e com isso se encarregará de fazer o cache desses objetos (vide **BOX 1**). A ideia é tornar a própria **String** que representa o comando CQL (passada como parâmetro) a chave do **Map**. Dessa forma, para cada instrução CQL passada nesse método haverá apenas um **PreparedStatement**. Essa estratégia pode ser vista através do método **CassandraCluster.prepare()**, apresentado na **Listagem 4**.

Listagem 4. Cache de PreparedStatement em CassandraCluster.

```
private BoundStatement prepare(String cql){  
    if(!preparedStatementCache.containsKey(cql)){  
        preparedStatementCache.put(cql, session.prepare(cql));  
    }  
  
    return preparedStatementCache.get(cql).bind();  
}
```

O ganho com *PreparedStatements* é verificado quando uma instrução é executada repetidas vezes, pois com esse tipo de statement o parse acontece uma única vez em cada nó que for executá-lo. Nas execuções subsequentes, apenas o ID do statement e os valores dos parâmetros são enviados pela rede. Considerando isso em um ambiente distribuído com vários nós, uma melhora de performance significativa pode ser obtida.

Por fim, saiba que o método **prepare()** sempre retornará uma nova instância de **BoundStatement**, a qual será utilizada pelos clientes para fazer o bind dos parâmetros contidos na instrução CQL.

Adicionando método para execução de comandos no CassandraCluster

Para que a classe **CassandraCluster** esteja com todas as principais funcionalidades disponíveis, agora falta incluir o método **execute()**, apresentado na **Listagem 5**. Como o próprio nome já diz, esse método será o responsável por executar instruções CQL no Cassandra.

Apesar desse código ser pequeno é interessante prestar bem atenção para entender o que acontece. Basicamente, ele recebe um **Statement** e delega a chamada para o método **Session.execute()** que, por sua vez, irá executar o comando no Cassandra.

Ao finalizar a execução, o método retorna um **ResultSet** que pode ser utilizado para obter os dados de uma consulta (no caso do comando ser uma consulta).

Listagem 5. Método para execução de CQL em CassandraCluster.

```
@Lock(LockType.READ)  
public ResultSet execute(Statement stmt){  
    return session.execute(stmt);  
}
```

BOX 1. Caches

Como já demonstrado na implementação de **CassandraCluster**, a utilização de caches ao se trabalhar com Cassandra é de fundamental importância. Portanto, vale a pena estudar mais a fundo as estratégias de cache a fim de identificar a que melhor se adequa à sua realidade. Apesar de ser possível utilizar maps para esse intuito, como fizemos aqui, esta não é a única maneira e, principalmente, não é a melhor abordagem para grandes aplicações. Por exemplo, você pode precisar de um cache que expira itens (mais antigos, menos utilizados, etc.) ou caso contrário irá acumular uma grande quantidade de objetos e poderá ter problemas de estouro de memória.

A parte interessante nesse método é a presença da anotação **@Lock**. Essa anotação faz parte da especificação EJB e deve ser usada em conjunto com EJBs singleton, como é o caso de **CassandraCluster**. Um EJB singleton, por padrão, não permite que mais de uma thread invoque um método de negócio. Nesse caso, entenda-se método de negócio como qualquer método de classe pública, como é o caso de **execute()**.

Assim, um cliente terá que esperar o outro terminar para que então seu pedido seja atendido. Por exemplo, suponha que a thread **A** chamou **execute()** passando um statement que vai demorar cinco minutos para finalizar sua execução. Quando a chamada de **A** estava com 1 minuto de execução, a thread **B** também invocou **execute()** com outro statement, que nesse caso irá levar apenas 1 segundo para executar. No entanto, como **A** iniciou a execução primeiro, a thread **B** só será atendida quando **A** terminar, ou seja, o comando de **B** que deveria levar apenas 1 segundo irá levar 4 minutos (tempo restante para completar a execução de **A**) e 1 segundo.



Como usar o Apache Cassandra em aplicações Java EE - Parte 2

Para que isso não aconteça, utilizamos `@Lock(LockType.READ)` para sinalizar ao container EJB que esse método, especificamente, não deve funcionar como da forma descrita anteriormente. Assim, não haverá lock no método `execute()` e qualquer cliente que invocá-lo será atendido de imediato.

Nesse momento os leitores mais experientes podem se perguntar: “Mais o que acontece quando diversas threads chamam esse método simultaneamente? Não há perigo de uma chamada atrapalhar a outra?”. A resposta é que não há problema algum. Como dito na primeira parte deste artigo, a classe `Session` é thread-safe e, portanto, pode ser utilizada num ambiente multithreading sem qualquer perigo.

Outro método que poderia sofrer esse mesmo problema seria o `prepare()`. Isso porque ele também pode ter execuções que vão levar um tempo considerável e, por isso, ficar aguardando sua finalização geraria um grande gargalo. No nosso cenário isso não irá acontecer porque esse método é privado e, sendo assim, não sofre o lock do container, pois não é considerado um método de negócio. No entanto, se sua aplicação precisar expor esse método, teria também que remover o lock do container EJB. Caso você necessite usar o controle de lock constantemente, isso pode indicar que é preciso repensar a implementação, como explicado no **BOX 2**.

BOX 2. Exposição de Session

Assim como os métodos `execute()` e `prepare()`, caso outros métodos da classe `Session` precisem ser expostos em `CassandraCluster`, uma outra estratégia seria criar uma classe encapsuladora; `CassandraSession`, por exemplo. Essa nova classe teria uma instância de `Session` encapsulada e só seriam expostos os métodos que fossem convenientes, evitando-se expor métodos de configuração da classe `Session` para toda a aplicação. Na classe `CassandraCluster`, então, seria criado um método `getCassandraSession()` que retornaria esse novo objeto. Dessa forma, a classe `Session` se manteria privada a um único ponto da aplicação, bem como a preocupação com locks seria eliminada, já que os métodos de longa duração seriam chamados fora do contexto EJB: `cassandraCluster.get``CassandraSession().execute()`.

Criando o CQL para inserir Usuário

Agora que o método `prepare()` está implementado, pode-se criar algumas instruções, como a de inserção de um usuário, demonstrada na **Listagem 6**. O intuito desse método é disponibilizar para o cliente uma instância de `BoundStatement` com o comando necessário para inserir um usuário no Cassandra. De posse do statement o desenvolvedor poderá setar os valores dos parâmetros (identificados pelo caractere `?`) e então fazer a execução do comando.

Note que a sintaxe do CQL nesse exemplo é bem semelhante à do SQL. A principal diferença está no uso da instrução `IF NOT EXISTS`, que será melhor explicada na seção “Usando Lightweight Transactions (LWT) para garantir a unicidade”, logo mais à frente.

Enfim, o bean de Usuário

Feitas as melhorias em `CassandraCluster`, agora é possível criar o bean da entidade usuário, `UserBean`. Trata-se de um EJB Stateless que usa CDI para injetar algumas dependências, como

verificado na **Listagem 7**. O sufixo “Bean” refere-se à convenção de nomes de EJB.

Listagem 6. CQL para inserir usuário (CassandraCluster).

```
public BoundStatement boundInsertUser(){  
    return prepare("INSERT INTO webshelf.user(login,name,password)  
    VALUES (?,?,?) IF NOT EXISTS;");  
}
```

Listagem 7. Código do bean de usuário (webshelf-business).

```
package br.com.devmedia.webshelf.service;  
  
import java.util.Set;  
  
import javax.ejb.Stateless;  
import javax.ejb.TransactionAttribute;  
import javax.ejb.TransactionAttributeType;  
import javax.inject.Inject;  
import javax.validation.ConstraintViolation;  
import javax.validation.ConstraintViolationException;  
import javax.validation.Validator;  
  
import com.datastax.driver.core.BoundStatement;  
import com.datastax.driver.core.ResultSet;  
import com.datastax.driver.mapping.Mapper;  
  
import br.com.devmedia.webshelf.data.CassandraCluster;  
import br.com.devmedia.webshelf.exception.BusinessRuleException;  
import br.com.devmedia.webshelf.model.User;  
  
@Stateless  
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)  
public class UserBean {  
  
    @Inject  
    private CassandraCluster cassandra;  
  
    @Inject  
    private Validator validator;  
  
    public User findUserByLogin(String login) {  
        Mapper<User> mapper = cassandra.mapper(User.class);  
        return mapper.get(login);  
    }  
  
    public void deleteUser(User user) {  
        Mapper<User> mapper = cassandra.mapper(User.class);  
        mapper.delete(user);  
    }  
  
    public void insertUser(User user) {  
        executeBeanValidation(user);  
  
        BoundStatement insertUser = cassandra.boundInsertUser();  
        insertUser.bind(user.getLogin(), user.getName(), user.getPassword());  
  
        ResultSet result = cassandra.execute(insertUser);  
  
        if (!result.wasApplied()) {  
            throw new BusinessRuleException("Login já existente.");  
        }  
    }  
  
    private void executeBeanValidation(User user) {  
        Set<ConstraintViolation<User>> constraintViolations = validator.validate(user);  
  
        if (!constraintViolations.isEmpty()) {  
            throw new ConstraintViolationException(constraintViolations);  
        }  
    }  
}
```

Como pode ser visto no método `executeBeanValidation()`, nesta classe injetamos um Validator para invocar as validações da API Bean Validation. Além disso, é injetado um `CassandraCluster` para realizar a comunicação com o Cassandra.

Ainda analisando este código, o método `findUserByLogin()` obtém um `Mapper<User>` e então faz a consulta pelo login, que nesse caso é também a primary key da tabela. Essa consulta é realizada através do método `Mapper.get()`, que aceita uma lista de parâmetros correspondente à primary key da tabela na ordem declarada na sua criação. O retorno é um objeto com os campos devidamente preenchidos graças ao mapeamento feito na classe `User`. Operação semelhante acontece no método `deleteUser()`, que também faz uso de `Mapper` para remover o registro.

Usando Lightweight Transactions (LWT) para garantir a unicidade

Por fim, tem-se o método `insertUser()`, o qual cadastra o usuário no Cassandra. Apesar da classe `Mapper` possuir um método `save()` que poderia ser usado aqui, foi necessário criar um statement manualmente através do método `CassandraCluster.boundInsertUser()` para que fosse possível usar *lightweight transactions*, já que o `Mapper` não oferece (ainda) essa possibilidade.

Como já informado, no WebShelf o login do usuário é único. Para garantir essa regra em bancos de dados relacionais, normalmente usamos uma transação, na qual é executada uma consulta para verificar a existência do login. Caso ele não exista, é executado o comando de insert e finalmente é feito o commit da operação.

No Cassandra, por sua vez, não é possível proceder dessa forma devido à ausência de transações ACID. Desse modo, se você tentar fazer isso, poderá cair numa race condition, como exemplificado na **Figura 1**. Para contornar esse problema foi criado o conceito de LWT, que no caso da inserção do usuário se caracteriza pelo uso da seguinte condição ao fim do insert: IF NOT EXISTS. Ou seja, ao invés de fazer uma consulta para verificar se o login existe ou não, no ato do insert já será feito essa checagem sem que outras requisições interfiram na operação.

A execução dessa instrução retorna um `ResultSet` a partir do qual é possível checar se o insert foi aplicado ou não através do método `wasApplied()`. Caso este retorne `false`, significa que o login já existe.

Contudo, como dito na primeira parte deste tutorial, essa feature deve ser usada com moderação, pois impacta fortemente na performance. Nesse exemplo, optou-se por fazer uso de LWT porque a regra de unicidade de usuário é considerada crítica para a aplicação.

Implementação do Login

Como em quase todas as aplicações web, também iremos desenvolver um mecanismo de login. O funcionamento deste será simples: o usuário terá que informar o seu login e sua senha e, caso tudo esteja correto, deverá ser redirecionado para a página

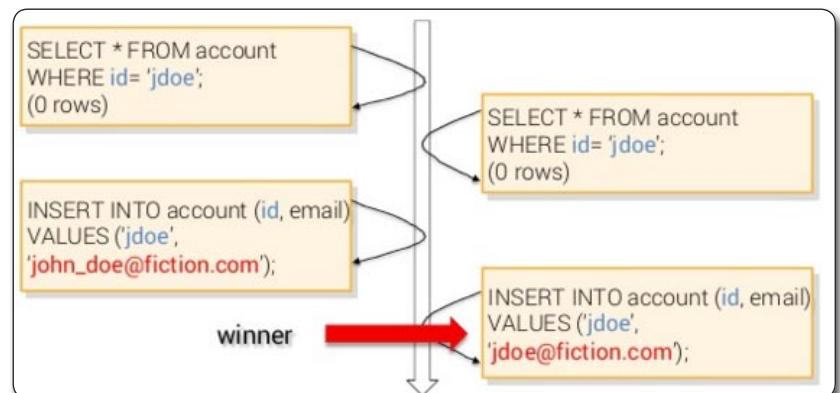


Figura 1. Exemplo de race condition – Fonte: FinishJUG

inicial da aplicação, caso contrário, uma mensagem de erro deverá ser mostrada. Para quem ainda não tem cadastro, será disponibilizado um botão que levará o usuário para a tela de Cadastro de Usuário.

Elaborando página responsiva de login com PrimeFaces

A tela de login tem quatro componentes principais, conforme pode ser visto na **Listagem 8**: um input de login, outro da senha, um botão para efetuar o login e outro para se cadastrar.

Observe que o parâmetro `renderedMenuBar`, logo no início do código, foi setado para `false`, já que não deve ser exibido nenhum menu enquanto o usuário não se logar.

Com o intuito de deixar a tela responsiva, utilizamos os mesmos recursos apresentados na **Listagem 1**. Assim, todos os componentes visuais da página foram englobados por uma `div` que tem a classe `ui-fluid` e os inputs foram organizados dentro de componentes `p:panelGrid` configurados para usar o Grid CSS (`layout="grid"`). Além disso, os `p:panelGrid` definiram o tamanho de suas colunas em função das 12 colunas que o Grid CSS usa para dividir a tela responsivamente (`ui-grid-col-*`).



Como usar o Apache Cassandra em aplicações Java EE - Parte 2

Uma coisa a observar é que nessa página temos dois **p:panelGrid**. Isso foi necessário porque, para gerar uma melhor visualização da tela, as configurações dos tamanhos das colunas são diferentes para a área dos campos texto (*Login* e *Senha*) e a área dos botões (*Entrar* e *Cadastrar-se*). Enquanto a primeira área usa os tamanhos 1, 4 e 7 (**columnClasses="ui-grid-col-1,ui-grid-col-4,ui-grid-col-7"**) para suas três colunas, a segunda área usa os tamanhos 1, 2 e 2 (**columnClasses="ui-grid-col-1,ui-grid-col-2,ui-grid-col-2"**).

Ainda com relação aos botões, o primeiro (*Entrar*) será utilizado para fazer o login e o segundo (*Cadastrar-se*) servirá para redirecionar o usuário para a tela de cadastro de usuário, caso o mesmo ainda não tenha se registrado no site.

As Figuras 2 e 3 demonstram o comportamento responsivo da tela de login.

Desenvolvendo controller de login

Após criarmos o XHTML do login, podemos desenvolver a classe que efetuará as principais ações dessa tela: **LoginController**. Basicamente, essa classe terá a responsabilidade de executar as funcionalidades de login e logout da aplicação e pode ser visualizada na **Listagem 9**.

Listagem 8. Código da tela de login (public/login.xhtml).

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:p="http://primefaces.org/ui"
    template="/WEB-INF/templates/default.xhtml">

    <ui:param name="mainContentTitle" value="WebShelf"/>
    <ui:param name="renderedMenuBar" value="false"/>

    <ui:define name="mainContent">
        <h:form prependId="false">
            <div class="ui-fluid">
                <p:panelGrid columns="3" layout="grid"
                    columnClasses="ui-grid-col-1,ui-grid-col-4,ui-grid-col-7"
                    styleClass="ui-panelgrid-blank">
                    <p:outputLabel for="userLogin" value="Login"/>
                    <p:inputText id="userLogin" autocomplete="false"
                        value="#{loginController.login}"/>
                    <p:message for="userLogin"/>

                    <p:outputLabel for="userPassword" value="Senha"/>
                    <p:password id="userPassword" autocomplete="false"
                        value="#{loginController.password}"/>
                    <p:message for="userPassword"/>
                </p:panelGrid>
                <p:panelGrid columns="3" layout="grid"
                    columnClasses="ui-grid-col-1,ui-grid-col-2,ui-grid-col-2"
                    styleClass="ui-panelgrid-blank">
                    <h:outputText />
                    <p:commandButton value="Entrar"
                        action="#{loginController.doLogin()}" ajax="false"
                        validateClient="true"/>
                    <p:button value="Cadastrar-se" outcome="insertUser"/>
                </p:panelGrid>
            </div>
        </h:form>
    </ui:define>
</ui:composition>
```



Figura 2. Tela de login visualizada em um computador



Figura 3. Tela de login visualizada em um celular

Visto que não precisará guardar nenhum estado entre requests, esta classe é um bean CDI com escopo de requisição (**@RequestScoped**). E como ela será acessada nos XHTMLs (*login.xhtml* e *default.xhtml*) através de Expression Language, também foi anotada com **@Named**.

Dentre os atributos da classe, os campos de login e senha (utilizados na página de login) são devidamente validados com o auxílio da API Bean Validation através da anotação **@NotNull**. Dessa forma, ambos são obrigatórios. Além desses dois, existem mais três campos na classe, todos injetados via CDI:

- **HttpServletRequest**: utilizado para obter acesso a **HttpSession**;
- **JsfMessage**: serve para adicionar mensagens ao contexto JSF; e
- **UserBean**: utilizado para consultar o usuário.

A principal funcionalidade provida por **LoginController** é o login do usuário, que é realizado através do método **doLogin()**. Este faz a pesquisa do usuário pelo seu login e caso o mesmo exista e sua senha seja igual à senha informada, então é considerado um login válido e assim o objeto usuário é armazenado na sessão. Caso não exista o usuário ou sua senha seja diferente do inputado, uma mensagem de erro é retornada.

Por fim, o método **logout()** simplesmente remove o usuário da sessão e invalida-a, bem como redireciona o usuário para a tela de login.

Listagem 9. Controller do login (webshelf-web)

```
package br.com.devmedia.webshelf.controller;

// imports omitidos...

@Named
@RequestScoped
public class LoginController implements Serializable {

    private static final long serialVersionUID = 1L;

    @Inject
    private HttpServletRequest request;

    @Inject
    private UserBean userBean;

    @Inject
    private JsfMessage<Messages> messages;

    @NotBlank(message="Senha: não pode está em branco.")
    private String password;

    @NotBlank(message="Login: não pode está em branco.")
    private String login;

    public String getPassword() {
        return this.password;
    }

    public String getLogin() {
        return this.login;
    }

    public void setPassword(String senha) {
        this.password = senha;
    }

    public void setLogin(String usuario) {
        this.login = usuario;
    }

    public String doLogin() {
        User user = userBean.findUserByLogin(login);

        String encryptedPassword = User.encryptPassword(password);

        if(user==null || !user.getPassword().equals(encryptedPassword)){
            messages.addWarn().invalidCredentials();
            return null;
        }

        if(request.getSession(Boolean.FALSE) != null){
            request.getSession(Boolean.FALSE).invalidate();
        }

        request.getSession().setAttribute("loggedinUser", user);
        return "/private/home.xhtml?faces-redirect=true";
    }

    public String logout() throws ServletException {
        request.getSession().removeAttribute("loggedinUser");
        request.getSession().invalidate();
        return "/public/login.xhtml?faces-redirect=true";
    }
}
```

É oportuno salientar que a maneira utilizada aqui para fazer o controle de login do sistema é muito simples e, portanto, deve ser evitada em produção, conforme comentado no **BOX 3**.

BOX 3. Cassandra vs JAAS

Diferentemente de bancos de dados relacionais, não existe uma forma out-of-box para implementar um security-domain no WildFly com o Cassandra. Por isso, nessa aplicação de exemplo utilizamos um mecanismo próprio e bem simples de controle de acesso, ao invés de usar o JAAS (Java Authentication and Authorization Service). No entanto, num ambiente de produção recomenda-se investir tempo para criar um módulo de login que possa utilizar o Cassandra em conjunto com o JAAS ou a utilização de algum framework de segurança como o Apache Shiro.

Disponibilizando o usuário logado como bean CDI

Obter o usuário logado é uma das tarefas mais comuns a qualquer aplicação web. Para isso, iremos criar a classe **ResourceProducer**, que fará uso das facilidades do CDI a fim de tornar essa tarefa mais simples dentro do projeto WebShelf. Essa implementação é mostrada na **Listagem 10**.

Na classe **ResourceProducer** temos apenas um método: **getLoggedInUser()**. Este produz beans CDI (**@Produces**) com escopo de sessão (**@SessionScoped**) e que são acessíveis via Expression Language (**@Named**) através do nome **loggedinUser**. A outra anotação (**@LoggedInUser**) é um qualifier CDI para denotar que o bean aqui produzido equivale ao usuário logado. Sua implementação e explicação serão apresentadas com a **Listagem 11**.

Listagem 10. Código da classe ResourceProducer (webshelf-web)

```
package br.com.devmedia.webshelf.util;

import javax.enterprise.context.SessionScoped;
import javax.enterprise.inject.Produces;
import javax.inject.Inject;
import javax.inject.Named;
import javax.servlet.http.HttpSession;

import br.com.devmedia.webshelf.model.User;

public class ResourceProducer {

    @Inject
    private HttpSession session;

    @Produces
    @LoggedInUser
    @SessionScoped
    @Named("loggedinUser")
    protected User getLoggedinUser() {
        User loggedinUser = (User)session.getAttribute("loggedinUser");

        if(loggedinUser == null) {
            loggedinUser = new User();
        }

        return loggedinUser;
    }
}
```

Como podemos observar, a implementação do método `getLoggedInUser()` é bem simples. Ele retorna o usuário logado que está na sessão caso exista um. Vale lembrar que o usuário logado é colocado na sessão através do método `LoginController.doLogin()`. Se não houver nenhum usuário logado, então ele devolve um objeto `User` “vazio”. Isso porque não é permitido retornar nulo em métodos produtores no CDI.

Listagem 11. Qualifier CDI para Usuário logado

```
package br.com.devmedia.webshelf.util;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Qualifier
@Target({ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface LoggedInUser {

}
```

A anotação `@LoggedInUser` é um qualifier CDI (`@Qualifier`) usado para produzir/injetar objetos do tipo `User` que representam um usuário logado. Um qualifier nada mais é do que um meio de se fornecer várias implementações para o mesmo tipo de bean.

Por padrão, toda classe é produzida pelo CDI através de seu construtor sem argumentos, no entanto, é possível instruí-lo a criar o bean de outras maneiras. No caso de `User`, criamos o método `ResourceProducer.getLoggedInUser()`, que irá produzir beans que representam um usuário logado, e para denotar isso, este método foi anotado com `@LoggedInUser`. Dessa forma, quando uma classe for injetar um objeto `User`, o CDI terá agora duas opções para gerar o objeto: a opção default (construtor sem argumento) e o método `ResourceProducer.getLoggedInUser()`.



A questão que fica é: como informar ao CDI que desejamos obter o objeto gerado pela classe `ResourceProducer`, ao invés do objeto gerado pelo construtor default? É nesse contexto que entra o qualifier. É através dele que o CDI saberá qual dos produtores usar. Veja o exemplo na [Listagem 12](#).

Nesse caso, como o atributo `loggedInUser` está anotado com `@LoggedInUser`, ele será produzido pelo método `ResourceProducer.getLoggedInUser()`, visto que somente este método produz usuários com tal qualificador. Já o atributo `dummyUser`, por não especificar nenhum qualifier, será gerado pelo produtor padrão, que no caso é o construtor default da própria classe (`new User()`).

Listagem 12. Exemplo de uso de um Qualifier CDI

```
@Inject
@LoggedInUser
private User loggedInUser;

@Inject
private User dummyUser;
```

Protegendo páginas privadas

Agora que já temos como obter e saber se existe um usuário logado na aplicação, vamos criar um mecanismo de segurança para bloquear o acesso a páginas privadas de usuários não autenticados. Para isso, iremos utilizar um evento do CDI através da classe `SecurityObserver`, apresentada na [Listagem 13](#).

Nesta classe, o método `checkAfterRestoreView()` será notificado pelo CDI todas as vezes que ocorrer o evento After Restore View do JSF. Isso acontece porque este método possui um parâmetro anotado com `@Observes` que é utilizado para indicar ao CDI que o método precisa ser avisado quando o evento observado ocorrer.

O evento é definido com a próxima anotação; nesse caso, `@AfterPhase(JsfPhaseId.RESTORE_VIEW)`. Essa anotação é fornecida pelo módulo JSF da biblioteca DeltaSpike e possibilita à aplicação monitorar o ciclo de vida JSF como eventos do CDI, dispensando assim o uso de PhaseListeners. Para conseguir capturar os eventos, essa anotação deve ser utilizada em objetos do tipo `PhaseEvent`.

Além de notificar o evento, o CDI ainda injetará os três parâmetros do método: `PhaseEvent`, `HttpServletRequest` e também o usuário logado (`@LoggedInUser`). Note que o `PhaseEvent` nem é utilizado, mas é necessário para que se possa usar `@AfterPhase`.

Baseado nessas informações recebidas o método `checkAfterRestoreView()` não permitirá que usuários logados acessem a tela de login e os redirecionará para a página home da aplicação (`redirectLoggedInUserToHome()`). Da mesma forma, também não permitirá que usuários não logados acessem páginas privadas, redirecionando-os para a tela de login (`redirectAnonymousToLogin()`).

Cadastro de Livro

O cadastro de livro permitirá ao usuário do WebShelf cadastrar novos livros que, por ventura, ele não tenha conseguido encontrar

na pesquisa. Essa funcionalidade seguirá o mesmo molde do cadastro de usuário, que é composto de uma tela JSF, um controller e um bean de negócio.

Listagem 13. Classe de segurança para páginas privadas (webshelf-web).

```
package br.com.devmedia.webshelf.security;

import javax.enterprise.event.Observes;
import javax.faces.context.FacesContext;
import javax.faces.event.PhaseEvent;
import javax.servlet.http.HttpServletRequest;

import org.apache.deltaspike.jsf.api.listener.phase.AfterPhase;
import org.apache.deltaspike.jsf.api.listener.phase.JsfPhaseld;

import br.com.devmedia.webshelf.model.User;
import br.com.devmedia.webshelf.util.LoggedInUser;

public class SecurityObserver {
    private static final String URL_PATTERN_PRIVATE_PAGES = "/private/";
    private static final String LOGIN_PAGE = "/public/login.xhtml";
    private static final String HOME_PAGE = "/private/home.xhtml";

    protected void checkAfterRestoreView(@Observes @AfterPhase
        (JsfPhaseld.RESTORE_VIEW) PhaseEvent event,
        HttpServletRequest request, @LoggedInUser User user) {
        this.redirectLoggedInUserToHome(user, request);
        this.redirectAnonymousToLogin(user, request);
    }

    private void redirectLoggedInUserToHome(User user, HttpServletRequest request) {
        if (request.getRequestURI().contains(LOGIN_PAGE)) {
            if (user.getLogin() != null) {
                handleNavigation(HOME_PAGE);
            }
        }
    }

    private void redirectAnonymousToLogin(User user, HttpServletRequest request) {
        if (request.getRequestURI().contains(URL_PATTERN_PRIVATE_PAGES)) {
            if (user.getLogin() == null) {
                handleNavigation(LOGIN_PAGE);
            }
        }
    }

    private void handleNavigation(String page) {
        FacesContext context = FacesContext.getCurrentInstance();
        context.getApplication().getNavigationHandler().handleNavigation
            (context, null, page + "?faces-redirect=true");
    }
}
```

Nota

Atualmente a API de object-mapping é bastante limitada, e por conta disso seu uso será restrito a cenários mais simples, normalmente CRUDs sem qualquer feature mais complexa do Cassandra.

Listagem 14. Código da classe Book (webshelf-business).

```
package br.com.devmedia.webshelf.model;

import java.nio.ByteBuffer;
import javax.validation.constraints.NotNull;
import org.hibernate.validator.constraints.NotBlank;

public class Book {

    @NotNull(message = "ISBN: não pode estar em branco.")
    private Long isbn;

    @NotBlank(message = "Título: não pode estar em branco;")
    private String title;

    @NotBlank(message = "Autor: não pode estar em branco.")
    private String author;

    private String country;

    private String publisher;

    @NotNull(message = "Imagem: não pode estar vazio.")
    private byte[] image;

    //getters and setters

    public ByteBuffer getImageBuffer() {
        return ByteBuffer.wrap(image);
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((isbn == null) ? 0 : isbn.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Book other = (Book) obj;
        if (isbn == null) {
            if (other.isbn != null)
                return false;
        } else if (!isbn.equals(other.isbn))
            return false;
        return true;
    }
}
```

Criando a classe modelo de Livro

A primeira etapa a seguir é criar a classe de domínio **Book**, que será composta dos seguintes campos: ISBN(único), título, autor, país, editora e uma imagem. A implementação pode ser vista na **Listagem 14**. Diferentemente de **User**, esta classe não utilizará a API de *object-mapping*, pois trata-se de um cenário mais complexo, onde será necessário criar mais de uma tabela na base de dados para representar a mesma entidade. Assim, para não termos que gerar várias classes de livro (uma para cada tabela), preferiu-se não usar essa feature do driver Cassandra.

Por se tratar de um POJO, **Book** é uma classe simples que contém basicamente seus campos e os respectivos métodos de acesso. A mesma também faz uso de Bean Validation para assegurar as validações básicas sob os seus atributos.

Um importante aspecto nessa classe é a implementação da dupla **equals()** e **hashCode()**. Assim, a classe poderá ser utilizada em conjunto com a API Collections de forma mais segura e consistente. Como pode ser visto, um **Book** é considerado igual a outro se os seus ISBNs forem iguais. Isso é importante porque, por conta da desnormalização do Cassandra, essa unicidade do ISBN não poderá ser garantida via banco de dados, como veremos logo mais.

Outra parte a se prestar atenção é o método **getImageBuffer()**. O intuito desse método é converter o campo **byte[]** que representa uma imagem em um objeto do tipo **ByteBuffer**. A razão disso é que o driver do Cassandra utiliza esta última classe para mapear atributos Java com suas colunas do tipo blob, justamente o tipo da coluna imagem nas tabelas de livro.

Implementando a página responsiva de Cadastro de Livro com PrimeFaces

A **Listagem 15** apresenta o código da tela de Cadastro de Livro. De forma simples, essa página contém os inputs necessários para preencher todos os campos da classe **Book**, e assim como as demais, também foi implementada de maneira responsiva, utilizando os mesmos recursos do PrimeFaces. Portanto, não terá seu código detalhado.

Modelando o cadastro de livros no Cassandra

Antes de pensarmos na modelagem do cadastro de livros, é importante contextualizarmos outra funcionalidade que irá impactar diretamente na modelagem: a pesquisa de livros. Essa pesquisa será um dos principais recursos do WebShelf e é através dele que será possível consultar livros por ISBN, Título e Autor.

A utilização desses três campos do cadastro de livros para realizar buscas demonstra claramente três padrões de consulta. Como explicado na seção sobre modelagem de dados no Cassandra, na primeira parte do artigo, uma das melhores práticas é modelar suas tabelas com base nas consultas que precisarão ser realizadas.



Listagem 15. Tela de Cadastro de Livro (`private/insertBook.xhtml`).

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:p="http://primefaces.org/ui"
  template="/WEB-INF/templates/default.xhtml">

<ui:param name="mainContentTitle" value="Cadastro de Livro"/>
<ui:param name="renderedMenuBar" value="true"/>

<ui:define name="mainContent">
  <h:form prependId="false" enctype="multipart/form-data">
    <div class="ui-fluid">
      <p:panelGrid columns="3" layout="grid"
        columnClasses="ui-grid-col-1,ui-grid-col-4,ui-grid-col-7"
        styleClass="ui-panelgrid-blank">

        <p:outputLabel for="bookTitle" value="Título:"/>
        <p:inputText id="bookTitle" value="#{bookController.book.title}" />
        <p:message for="bookTitle" />

        <p:outputLabel for="bookAuthor" value="Autor:"/>
        <p:inputText id="bookAuthor" value="#{bookController.book.author}" />
        <p:message for="bookAuthor" />

        <p:outputLabel for="bookPublisher" value="Editora:"/>
        <p:inputText id="bookPublisher"
          value="#{bookController.book.publisher}" />
        <p:message for="bookPublisher" />

        <p:outputLabel for="bookCountry" value="País:"/>
        <p:inputText id="bookCountry"
          value="#{bookController.book.country}" />
        <p:message for="bookCountry" />

        <p:outputLabel for="bookIsbn" value="ISBN:"/>
        <p:inputText id="bookIsbn" value="#{bookController.book.isbn}" />
        <p:message for="bookIsbn" />
      </p:panelGrid>
    <br />
    <p:panelGrid columns="1" layout="grid"
      columnClasses="ui-grid-col-1"
      styleClass="ui-panelgrid-blank">

      <p:fileUpload id="bookImage" label="Imagem" auto="true"
        allowTypes="/(.|\\/)(jpg|jpeg|png)$/" sizeLimit="20480"
        value="#{bookController.image}"
        invalidFileMessage="Tipo de arquivo inválido."
        invalidSizeMessage="Tamanho de arquivo não permitido."
        fileUploadListener="#{bookController.uploadImage}" process="@this"
        update="@this bookImageName">
        </p:fileUpload>

      <h:outputText id="bookImageName"
        value="#{bookController.image == null ? ': bookController.image.
        fileName'}" />

      <h:outputText />

      <p:commandButton value="Salvar" action="#{bookController.insert}"
        ajax="false" validateClient="true"/>
    </p:panelGrid>
  </div>
</h:form>
</ui:define>
</ui:composition>
```

Essa abordagem normalmente resulta na criação de uma tabela para cada padrão de consulta. Isso significa que para atender os requisitos da pesquisa de livros vamos criar três tabelas: *book_by_isbn*, *book_by_title* e *book_by_author*, todas elas com os mesmos dados. A diferença será a constituição de sua *primary key*, mais especificamente, a *partition key*, como pode ser notado na [Listagem 16](#).

Essa modelagem é o principal motivo para não usarmos *object-mapping* nas funcionalidades relacionadas ao cadastro de livros, já que a definição da tabela através de **@Table** é estática e teríamos que criar três classes para mapear cada uma das tabelas.

Listagem 16. Criação das tabelas de Livro.

```
CREATE TABLE IF NOT EXISTS webshelf.book_by_isbn (
    isbn bigint,
    title text,
    author text,
    country text,
    publisher text,
    image blob,
    PRIMARY KEY (isbn)
);

CREATE TABLE IF NOT EXISTS webshelf.book_by_title (
    isbn bigint,
    title text,
    author text,
    country text,
    publisher text,
    image blob,
    PRIMARY KEY (title, isbn)
);

CREATE TABLE IF NOT EXISTS webshelf.book_by_author (
    isbn bigint,
    title text,
    author text,
    country text,
    publisher text,
    image blob,
    PRIMARY KEY (author, isbn)
);
```

E porque precisamos de três tabelas ao invés de uma, se todas têm as mesmas informações?

A questão é que no Cassandra qualquer query precisa filtrar a tabela no mínimo pelas colunas que compõem sua *partition key*, ou seja, se você quiser filtrar a consulta de livros apenas pelo Título, terá que ter uma tabela onde o campo *title*, sozinho, seja a *partition key*. Dito isso, podemos observar que a única diferença entre cada tabela apresentada na [Listagem 15](#) é justamente esse elemento.

Assim, a tabela *book_by_isbn*, por exemplo, que será usada para buscar livros de acordo com o seu ISBN, tem sua *primary key* composta apenas por esse campo, e consequentemente, sua *partition key* também é formada somente por esse campo.

Já a tabela *book_by_title*, que será usada para buscar livros de acordo com o título, tem sua *primary key* composta pelas colunas *title* e *isbn*. O campo *title* vem na primeira posição para assegurar que o mesmo será a *partition key* da tabela. Deste modo, poderão ser

executadas consultas utilizando apenas esse o campo como filtro. Todavia, como a coluna *title* não garante a unicidade dessa tabela, foi incluído o campo *isbn* na *primary key*.

A tabela *book_by_author*, por sua vez, tem intensão semelhante a *book_by_title*, sendo a única diferença que a mesma será usada para pesquisar livros de acordo com o autor.

Nota

Vale lembrar que a *partition key* é o primeiro campo ou o primeiro conjunto de campos da *primary key*. Os exemplos a seguir deixam mais clara essa definição:

- PRIMARY KEY (*user_login*, *status*, *book_isbn*): *user_login* é a *partition key*;
- PRIMARY KEY (*(user_login, status)*, *book_isbn*): *user_login* e *status*, colocados entre parênteses, formam a *partition key*.

A lógica por trás da *partition key*, como explicado na primeira parte do artigo, é que através dessa chave o Cassandra determina em qual nó ficará armazenado o dado. Daí a necessidade de sempre ter esse filtro nas consultas, pois sem essa restrição, como o Cassandra poderia saber em qual nó do cluster está a informação?

Também pelo mesmo motivo, as condições suportadas para as colunas que compõem a *partition key* são apenas igual (=) e **in**, ou seja, o Cassandra precisa saber o valor exato da *partition key* para determinar em qual máquina buscar a informação. Se fosse possível fazer algo como uma operação *like*, o Cassandra teria de varrer todas as máquinas do cluster para achar o dado procurado, o que resultaria numa grande perda de performance. Um bom exemplo desse mecanismo é descrito na primeira parte do artigo na seção “Distribuição – A chave para a escalabilidade horizontal”.

Vale ressaltar que mesmo sendo permitido o uso do **in**, essa clausula é vista como uma má prática por vários desenvolvedores, pois pode obrigar o Cassandra a obter dados de vários nós diferentes numa única operação. Por exemplo, digamos que no **in** você informe três valores, e suponha que o particionador do Cassandra distribuiu os dados dessas três chaves em máquinas diferentes. Nesse cenário, uma única consulta fará com que o Cassandra colete dados de três nós distintos para então retornar a resposta para o cliente. Esse tipo de operação envolvendo várias máquinas acarreta uma perda de performance considerável e por isso deve ser evitada.

Desenvolvendo a lógica de negócio de Livro

Como já modelamos as tabelas do cadastro de livros, agora é possível pensar na lógica de negócios que será inserida na classe **BookBean**, implementada na [Listagem 17](#).

Assim como **UserBean**, esta classe também é um EJB Stateless sem suporte a transações que injeta, através de CDI, os objetos **Validator** e **CassandraCluster**. No entanto, o método **insertBook()** traz novos conceitos. Como os livros precisam ser cadastrados em três tabelas, a operação de insert é composta de três statements, cada um inserindo os dados numa tabela distinta.

Como usar o Apache Cassandra em aplicações Java EE - Parte 2

Listagem 17. Código da classe BookBean (webshelf-business)

```
package br.com.devmedia.webshelf.service;

//imports omitidos...

@Stateless
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class BookBean {

    @Inject
    private CassandraCluster cassandra;

    @Inject
    private Validator validator;

    public void insertBook(Book book) {
        executeBeanValidation(book);

        BatchStatement batch = new BatchStatement();

        batch.add(cassandra.boundInsertBookByISBN().bind(book.getIsbn(),
            book.getTitle(), book.getAuthor(), book.getCountry(), book.getPublisher(),
            book.getImageBuffer()));
        batch.add(cassandra.boundInsertBookByTitle().bind(book.getIsbn(),
            book.getTitle(), book.getAuthor(), book.getCountry(), book.getPublisher(),
            book.getImageBuffer()));
        batch.add(cassandra.boundInsertBookByAuthor().bind(book.getIsbn(),
            book.getTitle(), book.getAuthor(), book.getCountry(), book.getPublisher(),
            book.getImageBuffer()));

        cassandra.execute(batch);
    }

    private void executeBeanValidation(Book book) {
        Set<ConstraintViolation<Book>> constraintViolations = validator.validate(book);

        if (!constraintViolations.isEmpty()) {
            throw new ConstraintViolationException(constraintViolations);
        }
    }
}
```

Essa operação de gravar o livro nas três tabelas modeladas (*book_by_isbn*, *book_by_title* e *book_by_author*), de uma maneira geral, poderia ser feita de duas formas: executar cada insert numa operação independente ou executar os três inserts numa única operação (batch).

A primeira abordagem, no entanto, poderia resultar numa base de dados inconsistente, visto que o Cassandra não tem o conceito de transações ACID. Dessa maneira, uma operação poderia completar com sucesso e outra não. Por exemplo, suponha que o primeiro comando seja o insert na tabela *book_by_isbn* e que o mesmo completa com sucesso. Em seguida, durante os inserts nas tabelas *book_by_title* e *book_by_author*, imagine que aconteça algum erro e tais comandos não sejam gravados na base. Nessa situação, o livro que deveria estar presente nas três tabelas só existirá em uma, pois mesmo que os últimos dois comandos tenham sofrido algum erro, a primeira execução já foi completada com sucesso, gravada na base e já se encontra disponível para os demais clientes consultarem.

Usando BatchStatements para garantir a atomicidade

Para contornar o problema descrito utilizamos a segunda abordagem – o uso de batches – como recomenda a quarta regra de utilização do driver DataStax. Por isso declaramos **BatchStatement** no método **insertBook()**.

Nesse método, o batch é composto pelos três inserts do cadastro de livro, como pode ser notado através das chamadas ao método **add()** da classe **BatchStatement**. Este método recebe como parâmetro um Statement que é enfileirado para posterior execução na ordem em que foi adicionado. No código da **Listagem 17** os statements são criados pelos métodos **CassandraCluster.boundInsert*()**, que serão melhor explanados ao analisarmos a **Listagem 18**.

Observe que o método **add()** não executa o comando ainda. Isso só acontece na chamada ao método **CassandraCluster.execute()**, onde o **BatchStatement** é passado como parâmetro (**cassandra.execute(batch)**). Nesse momento, a execução de cada um dos statements adicionados previamente acontece numa operação atômica, e com isso, os três inserts funcionarão ou nenhum será de fato persistido.

É importante ressaltar que o principal intuito para usar batches no Cassandra é o que acabamos de citar: atomicidade. Portanto, não espere melhorias de performance por conta disso. Na verdade, essa atomicidade geralmente implica numa piora da performance, já que o Cassandra terá que se preocupar com esse quesito (atomicidade da operação), o que envolverá várias checagens extras para garantir a consistência e que não ocorrem numa operação isolada.

Preparando instruções CQL para inserção de livros

Para fechar o cadastro de livros falta apenas criar os métodos na classe **CassandraCluster** que irão gerar os PreparedStatements com os comandos necessários para a inclusão. Esses métodos podem ser visualizados na **Listagem 18**.

Note que cada um dos métodos **boundInsertBook*()** prepara uma instrução CQL para executar um insert em uma das tabelas de livro. Como essas tabelas são estruturalmente iguais, mudando apenas a primary key e a partition key, os três comandos são semelhantes, diferenciando-se unicamente pelo nome da tabela alvo.

Listagem 18. Comandos CQL para inserção de livros (CassandraCluster)

```
public BoundStatement boundInsertBookByAuthor(){
    return prepare("insert into webshelf.book_by_author (isbn,title,author,country,
        publisher,image) values(?,?,?,?,?,?)");
}

public BoundStatement boundInsertBookByTitle(){
    return prepare("insert into webshelf.book_by_title (isbn,title,author,country,
        publisher,image) values(?,?,?,?,?,?)");
}

public BoundStatement boundInsertBookByISBN(){
    return prepare("insert into webshelf.book_by_isbn (isbn,title,author,country,
        publisher,image) values(?,?,?,?,?,?)");
}
```

Note ainda que as implementações desses métodos também são bem parecidas, consistindo apenas em declarar o método **prepare()** para que seja criado um **PreparedStatement** para cada um dos inserts e que estes sejam armazenados num cache para utilização em operações subsequentes. Em seguida, o método **prepare()** retorna um **BoundStatement** que, por sua vez, é retornado para os clientes poderem setar os parâmetros dos CQLs (caractere ?) e executar o comando.

De acordo com o que apresentamos neste artigo, ficou evidente que o Apache Cassandra é uma opção NoSQL bastante viável para utilização com Java EE e as diversas tecnologias que cercam essa plataforma. Isso pôde ser constatado quando abordamos o uso do Cassandra numa aplicação com tecnologias como EJB, CDI, JSF, PrimeFaces e DeltaSpike.

Neste segundo artigo o foco foi completamente voltado para a parte prática, e mesmo para quem não conhecia muito a respeito do Cassandra, podemos dizer que o leitor agora tem uma boa visão das possibilidades, vantagens, melhores práticas e restrições que essa tecnologia impõe.

Embora vários dos principais tópicos já tenham sido discutidos, obviamente isso é apenas o início e ainda será preciso aprender muitas outras técnicas, features e conceitos para se tornar um profissional com domínio mais concreto sobre a tecnologia. Para tanto, recomendamos que busque conhecer novas técnicas de modelagem baseadas em cenários mais complexos. Isso fará com que você "abra a cabeça" para pensar de uma forma "mais Cassandra e menos relacional".

Duas ótimas fontes são as diversas apresentações de cases reais disponíveis no SlideShare e no YouTube.

Além disso, sugerimos que o leitor se empenhe em ler a documentação sobre CQL e Driver DataStax (veja na seção **Links**) para que consiga atingir um nível profissional no Cassandra.

Para encerrar, vale ressaltar que se você não leu a primeira parte terá bastante dificuldade em compreender aspectos mais avançados e abstratos da arquitetura do Cassandra.

Por isso, aconselhamos que dê um passo atrás e leia o artigo introdutório, onde foi apresentado um embasamento teórico essencial para evoluir no aprendizado dessa tecnologia.

Links:

Apache Cassandra Product Guide.

<http://docs.datastax.com/en/cassandra/2.2/index.html>

DataStax - CQL Product Guide.

<http://docs.datastax.com/en/cql/3.3/index.html>

DataStax - Java Driver Product Guide.

<http://docs.datastax.com/en/developer/java-driver/2.1/index.html>

DataStax Academy.

<https://academy.datastax.com>

PrimeFaces Web Site.

<http://www.primefaces.org>

Documentação DeltaSpike.

<https://deltaspike.apache.org/documentation>

Autor



Marlon Patrick

marlon.patrick@mpwtecnologia.com - marlonpatrick.info
É bacharel em Ciéncia da Computação e atua como Consultor Java no Grupo Máquina de Vendas com aplicações de missão crítica, tem oito anos de experiência com tecnologia Java e é certificado SCJP 5.



Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Introdução ao Java 9: Conheça os novos recursos

Aprenda neste artigo as principais novidades do JDK, que trará para o Java a modularização, jShell, HTTP 2.0, JMH, entre outras melhorias

A plataforma Java é uma das opções mais populares quando pensamos em desenvolvimento de aplicações, e isso se torna ainda mais evidente quando o escopo são as soluções voltadas para a web. Atualmente, estima-se que cerca de nove milhões de desenvolvedores adotam o Java. Completando 20 anos em breve, a plataforma é classificada por muitos como antiga, nesse universo em que novas soluções surgem e desaparecem num piscar de olhos.

Todo esse tempo, obviamente, possibilitou mais robustez e confiabilidade ao Java, no entanto, por ser projetada com alguns conceitos hoje tidos como obsoletos, a exemplo da arquitetura não-modular, possui uma estrutura monolítica, ou seja, não dividida em módulos. Essa ausência, presente em algumas das linguagens mais modernas, torna mais difícil o reuso e a manutenção do código, restringindo a sua utilização, principalmente, em dispositivos de baixa capacidade de processamento. Por causa disso, há muito tempo a comunidade Java solicita uma grande reforma na estrutura da plataforma, com o objetivo de torná-la modular.

Vale lembrar, ainda, que ao longo de sua história a plataforma Java cresceu de um pequeno sistema criado para dispositivos embarcados para uma rica coleção de bibliotecas, que atende às mais diversas necessidades e que precisam rodar em ambientes com sistemas operacionais e recursos de hardware distintos. Hoje sabemos que possuir um canivete suíço com tantos recursos é essencial ao desenvolvedor, contudo, essa abundância também traz alguns problemas:

Fique por dentro

Este artigo apresenta as principais funcionalidades previstas para a mais nova versão da linguagem Java: a tão esperada modularização, a API de suporte ao protocolo HTTP 2.0, bem como outras menores, como o jShell, a biblioteca JMH, entre outras. Essas novidades do Java 9 possibilitarão uma grande melhoria de desempenho às aplicações, principalmente às que executam em dispositivos com baixo poder de processamento. Sendo assim, é de suma importância que desenvolvedores e arquitetos tenham domínio sobre o potencial dos novos recursos, de forma a tirar o melhor proveito dos mesmos em suas próximas soluções.

- **Tamanho:** O JDK sempre foi disponibilizado como um grande e indivisível artefato de software, com várias bibliotecas e recursos para o desenvolvedor. A inserção de novidades na plataforma ao longo dos anos culminou no aumento constante deste, tornando-o cada vez mais pesado;
- **Complexidade:** O JDK é profundamente interconectado, ou seja, as bibliotecas são muito dependentes umas das outras, compondo assim uma estrutura monolítica. Além disso, com o tempo essa estrutura resultou em conexões inesperadas entre APIs e suas respectivas implementações, levando a um tempo de inicialização e consumo de memória maiores, degradando o desempenho de aplicações que necessitam da plataforma para funcionar. Para se ter uma ideia, um programa que escreve um simples “Alô, mundo!” no console, ao carregar e inicializar mais de 300 classes, leva, em média, 100ms em uma máquina desktop.

Sabendo que quanto maior a aplicação maior será o tempo necessário para inicializá-la, é fundamental evoluir esse cenário para aprimorar o tempo de inicialização e o consumo de memória. Com esse objetivo, os desenvolvedores do Java optaram por dividir o JDK, especificando um conjunto de módulos separados e independentes.

Praticamente concluído, o processo de reestruturação do Java em módulos consiste na identificação de interconexões entre bibliotecas e na eliminação dessas dependências quando possível. Isso reduz o número de classes carregadas em tempo de execução e, consequentemente, melhora tanto o tempo de inicialização quanto o consumo de memória, pois ao reduzir o acoplamento a relação de dependência entre certas classes diminuirá. Deste modo, com o JDK modular, apenas os módulos necessários para inicializar a aplicação serão carregados. Outra consequência é que a utilização de módulos poderá ser empregada não apenas pelo JDK, mas também por bibliotecas e aplicações, de tal forma a melhorar cada vez todo o universo Java.

Em conjunto com a comunidade, a modularização do JDK está sendo desenvolvida pela Oracle – sob a liderança de Mark Reinhold – em um projeto de codinome Jigsaw (quebra-cabeça). Parte do JDK 9, está previsto para ser entregue no primeiro semestre de 2017 e, uma vez que é a maior, principal e mais aguardada mudança em relação à versão 8, é o assunto do primeiro tópico apresentado neste artigo.

O segundo tópico abordado será o jShell, ferramenta de linha de comando que possui suporte a REPL (*Read Eval Print Loop*). Assim, a partir de agora será possível escrever e testar expressões em Java na linha de comando, sem a necessidade de escrever classes e métodos. Essa ferramenta já está disponível para o público através das versões de acesso antecipado (*Early Access*) do JDK 9 com o intuito de obter retorno da comunidade sobre o seu funcionamento e sugestões para futuras melhorias.

O terceiro assunto que analisaremos neste artigo será o JMH, ou *Java Microbenchmarking Harness*, o qual fornece uma estrutura para construir, rodar e analisar testes de desempenho em vários níveis de granularidade. A importância dessa funcionalidade vem da necessidade de se obter avaliações de desempenho precisas, uma vez que o tempo de aquecimento (*warmup time*) para inicializar os recursos para rodar o teste de desempenho, bem como as otimizações automáticas realizadas pelo compilador Java no código a ser testado, causam grande impacto no resultado dessas avaliações. Essa constatação é ainda mais evidente quando se trata de operações que duram apenas micro ou nanossegundos.

O quarto tópico diz respeito à alteração do coletor de lixo (*garbage collector*) padrão do Java, que passa a ser o G1, o qual funciona melhor em JVMs (*Java Virtual Machine*) cuja memória de alocação (*heap space*) é superior a 4GB, característica muito comum nos dias atuais. Dessa forma, é esperado que o gerenciamento de memória da JVM seja mais eficiente,

melhorando o desempenho do funcionamento da máquina virtual como um todo.

O quinto tópico destaca a implementação de uma API com suporte ao protocolo HTTP 2.0 e websockets. Essa API é muito importante porque a especificação da versão 2.0 do HTTP foi disponibilizada já há algum tempo, baseada na implementação do protocolo SPDY do Google, o qual recebeu críticas muito positivas, pois de fato possibilita a aceleração da navegação na web. Assim, o objetivo é fazer com que o Java possa se antecipar à adoção massiva do novo HTTP e forneça uma opção para a utilização do mesmo nas aplicações desenvolvidas com essa plataforma.

Por fim, será analisada a melhoria da API de processos do Java, pois até o momento o JDK possui uma grande limitação para controlar e gerenciar processos do sistema operacional. Hoje, para obter um simples número de identificação de um processo no sistema operacional, é preciso utilizar código nativo do SO (em geral em C), o que dificulta a implementação e leva a um alto acoplamento.

Projeto Jigsaw – Modularização do JDK

O projeto Jigsaw é um esforço da Oracle e da comunidade Java para a implementação da modularização da plataforma, tendo em mente os seguintes objetivos, de acordo com a página oficial do projeto:

- Tornar a plataforma Java SE e o JDK mais facilmente escaláveis para dispositivos pequenos e/ou de baixo poder de processamento;
- Melhorar a segurança e manutenibilidade da plataforma Java SE e do JDK;
- Permitir um melhor desempenho das aplicações Java;
- Tornar mais fácil aos desenvolvedores a construção e manutenção de bibliotecas e grandes aplicações nas plataformas Java.

Para alcançar esses objetivos, foi proposto projetar e implementar um sistema de módulos padrão para a plataforma Java SE e JDK.



Introdução ao Java 9: Conheça os novos recursos

A partir disso, os requisitos para a criação do sistema de módulos foram especificados e, em seguida, foi feito um rascunho (draft) com essa especificação.

De acordo com a JSR 376 (documento de especificação dos requisitos), o sistema de módulos possui duas características fundamentais:

1. Configuração confiável (Reliable configuration): Substituição do mecanismo de classpath padrão por uma forma dos componentes de software declararem dependências explícitas, a fim de evitar conflitos; e

2. Forte encapsulamento (Strong encapsulation): Permissão para um componente de um módulo declarar quais dos seus tipos de dados declarados como públicos são acessíveis a componentes de outros módulos e quais não são.

Nota

Nas versões anteriores do Java, o tipo público especifica que qualquer classe no classpath, independentemente do pacote em que está, pode acessar aquele tipo diretamente. Como os módulos são uma nova entidade que abrange pacotes/componentes, é preciso informar quais tipos que foram declarados como públicos são acessíveis a outros componentes que fazem parte de outros módulos.

Essas características auxiliam no desenvolvimento de aplicações, bibliotecas e da própria plataforma Java, pois garante maior escalabilidade com aplicações/plataformas mais enxutas, e apenas com as dependências necessárias; maior integridade, ao evitar a utilização de dependências cujas versões não são as mais adequadas; assim como grandes melhorias em termos de desempenho, com a resolução de dependências e coleta de lixo mais eficientes, bem como um tamanho menor dos arquivos de execução.

Módulos

Os módulos são um novo tipo de componente do Java, possuindo um nome e um código descritivo, isto é, um arquivo que define os detalhes e propriedades dos mesmos, além de outras informações adicionais. Essas informações adicionais descrevem, basicamente, configurações de serviços ou recursos que o módulo pode utilizar.

No código descritivo, por sua vez, o módulo deve informar se faz uso de tipos (classes, interfaces ou pacotes) de outros módulos, de forma que a aplicação possa compilar e executar sem erros. Para isso, deve-se utilizar a cláusula **requires** em conjunto com o nome do recurso. Além disso, pode ser necessário informar ao compilador quais tipos desse módulo podem ser acessados por outros, ou seja, declarar quais recursos ele exporta, o que é feito através da cláusula **exports**.

A partir disso, o sistema de módulos localiza os módulos necessários e, ao contrário do sistema de classpath, garante que o código de um módulo acesse apenas os tipos dos módulos dos quais ele depende. Como complemento, o sistema de controle de acesso da linguagem Java e da máquina virtual também previnem

que códigos acessem tipos oriundos de pacotes que não são exportados pelos módulos que os definem.

Com o intuito de evitar a forte dependência entre módulos, um módulo pode declarar que utiliza (**uses**) uma interface (tipo genérico) em vez de uma classe (tipo específico) de um determinado serviço cuja implementação é fornecida (**provided**) em tempo de execução por outro módulo. Essa estratégia permite que os desenvolvedores possam, entre outras coisas, estender a API do Java, codificando classes que implementam uma interface que é utilizada pelo módulo em desenvolvimento, interface essa declarada através da sintaxe **uses nomedainterface**. Isso ocorre porque a dependência do módulo que declara a interface com a classe criada pelo desenvolvedor é resolvida em tempo de compilação e de execução, sem necessidade de quaisquer intervenções. Juntamente a essa capacidade de resolução de componentes genéricos, o sistema de módulos mantém a hierarquia atual de classloaders, facilitando a execução ou migração de aplicações legadas para versão 9 do Java.

Tendo em vista que os módulos são descritos através de um código, a maneira mais simples de declará-los é especificar apenas os seus respectivos nomes em um arquivo, conforme o código a seguir:

```
module br.com.devmedia {}
```

Como já mencionado, no entanto, existem mais opções de configuração, como informar que um módulo depende de outro, utilizando a cláusula **requires**. Desse modo, supondo que o módulo criado anteriormente (**br.com.devmedia**) dependa de outro (**br.com.devmediaOutro**), a estrutura utilizada para descrever essa relação será semelhante à exposta na **Listagem 1**.

É possível, ainda, declarar os pacotes do módulo cujos tipos públicos podem ser acessados por outros, através da cláusula **exports**. Na **Listagem 2**, **br.com.devmedia** define que os pacotes **br.com.devmedia.pacote1** e **br.com.devmedia.pacote2** podem ser utilizados por outros módulos. Portanto, se na declaração de um módulo não existe a cláusula **exports**, o módulo não irá, de forma alguma, exportar pacotes para outros.

Listagem 1. Declaração do módulo **br.com.devmedia** com dependência ao módulo **br.com.devmediaOutro**.

```
module br.com.devmedia {  
    requires br.com.devmediaOutro;  
}
```

Listagem 2. Configuração do módulo **br.com.devmedia** com dependência e exports declarados.

```
module br.com.devmedia {  
    requires br.com.devmediaOutro;  
    exports br.com.devmedia.pacote1;  
    exports br.com.devmedia.pacote2;  
}
```

E a declaração do módulo, como é feita? Por convenção, deve ser criado um arquivo descritor, de nome *module-info.java*, no diretório raiz do módulo, por exemplo:

```
module-info.java  
br/com/devmedia/pacote1/FabricaPacoteUm.java  
br/com/devmedia/pacote2/ClassePacoteDois.java  
... (outros pacotes/classes/interfaces)
```

Feito isso, o descritor poderá ser compilado, assim como feito com as classes/interfaces, e resultará em um artefato de nome *module-info.class*, colocado no mesmo diretório dos demais arquivos compilados.

Por fim, saiba que, do mesmo modo que os nomes dos pacotes, os nomes dos módulos não podem estar em conflito. Assim, a forma recomendada de se nomear módulos, a fim de evitar problemas, é utilizar o padrão de nome de domínio ao contrário, como já demonstrado nos exemplos anteriores.

Artefatos de módulos

Como sabemos, as ferramentas de linha de comando da plataforma Java são capazes de criar, manipular e consumir arquivos JAR. Diante disso, de forma a facilitar a adoção e migração para a nova versão do JDK, os desenvolvedores criaram também o arquivo JAR modular. Como se pode imaginar, um arquivo JAR modular é um arquivo JAR comum que possui a definição do módulo compilada (o arquivo *module-info.class*) no seu diretório raiz (vide **BOX 1**). Por exemplo, um JAR para o módulo especificado anteriormente teria a estrutura apresentada na **Listagem 3**.

BOX 1. Arquivo JAR modular

Pode ser utilizado como um módulo (a partir do Java 9) ou como um arquivo JAR comum (em todas as versões). Assim, para fazer uso de um arquivo JAR modular como se fosse um JAR comum, basta informar o mesmo no classpath da aplicação. Dessa forma o arquivo *module-info.class* será ignorado. Essa estratégia permite que aplicações legadas, que executam em versões anteriores do Java, possam se beneficiar de bibliotecas desenvolvidas de forma modular, mantendo com isso a retrocompatibilidade.

Listagem 3. Estrutura do JAR para o módulo exemplo.

```
META-INF/  
META-INF/MANIFEST.MF  
module-info.class  
br/com/devmedia/pacote1/FabricaPacoteUm.class  
br/com/devmedia/pacote2/ClassePacoteDois.class  
...(outras classes e pacotes)
```

Módulos da plataforma Java SE 9

A especificação da plataforma Java SE 9 a divide em um conjunto de módulos. Entretanto, uma implementação da mesma pode conter todos ou apenas alguns módulos. Essa opção é importante porque alguns projetos não precisam fazer uso de todos os módulos, tornando assim mais fácil escalar as aplicações, como também o

ambiente de execução do Java, para dispositivos cada vez menores, uma vez que os executáveis também serão menores.

O único módulo que deve estar presente em todas as implementações é o módulo base, o qual possui o nome **java.base**. Este define e exporta todos os pacotes que fazem parte do núcleo da plataforma, incluindo o próprio sistema de módulos (vide **Listagem 4**).

Listagem 4. Declaração do módulo **java.base** do JDK9.

```
module java.base {  
    exports java.io;  
    exports java.lang;  
    exports java.lang.annotation;  
    exports java.lang.invoke;  
    exports java.lang.module;  
    exports java.lang.ref;  
    exports java.lang.reflect;  
    exports java.math;  
    exports java.net;  
    ...  
}
```

Qualquer outro módulo do Java sempre irá depender do módulo base, o qual não depende de ninguém. Como consequência, todos os outros módulos compartilharão o prefixo “java.” em seu nome, como o **java.sql** (para conectividade com o banco de dados), **java.xml** (para processamento de XML) e **java.logging** (para registro de operações/eventos). Entretanto, existem exceções, como os módulos que fazem parte apenas do JDK. Por convenção, esses utilizam o prefixo “jdk.”.

Resolução e dependência

Para melhor compreender como os módulos são encontrados ou resolvidos em tempo de compilação ou de execução, é preciso entender como eles estão relacionados entre si. Supondo que uma aplicação faça uso do módulo **br.com.devmedia**, mencionado anteriormente, como também do módulo **java.sql**, da plataforma



Introdução ao Java 9: Conheça os novos recursos

Java 9, o módulo que conteria o núcleo da aplicação seria descrito conforme a **Listagem 5**.

Listagem 5. Declaração do módulo br.com.devmedia-app.

```
module br.com.devmedia-app {  
    requires br.com.devmedia;  
    requires java.sql;  
}
```

Note que o módulo núcleo da aplicação, no caso **br.com.devmedia-app**, seria o ponto de partida para que o sistema encontre as dependências necessárias para a mesma, tendo como base os nomes dos módulos indicados pela cláusula **requires** do descriptor do módulo inicial, como demonstrado no exemplo. Saiba, ainda, que cada módulo do qual o módulo inicial depende pode depender de outros. Nestes casos, o sistema de módulos procurará recursivamente por todas as dependências de todos os outros, até que não reste nenhuma a ser acrescentada. Logo, se essa relação de dependência entre módulos fosse desenhada em um gráfico, o desenho teria a forma de um grafo, no qual cada relação entre duas dependências seria expressa por uma aresta e cada dependência seria um vértice.

Com o intuito de construir o grafo com as dependências do módulo **br.com.devmedia-app**, o compilador precisa ler as descrições dos módulos que ele depende (**java.sql** e **br.com.devmedia**). Logo, para melhor compreender a construção desse grafo, é preciso dar uma olhada na declaração dos módulos que se depende.

Em virtude de a declaração do módulo **br.com.devmedia** já ter sido demonstrada em uma listagem anterior, na **Listagem 6** é possível ver a declaração da outra dependência, o **java.sql**.

Listagem 6. Declaração do módulo java.sql.

```
module java.sql {  
    requires java.logging;  
    requires java.xml;  
    exports java.sql;  
    exports javax.sql;  
    exports javax.transaction.xa;  
}
```

Nota

Não foi criada uma nova listagem com a descrição do módulo **br.com.devmedia** porque já foi demonstrado na **Listagem 2**. Além disso, para ser mais breve, também foram omitidas as descrições dos módulos **java.logging** e **java.xml**, dos quais o módulo **java.sql** depende.

Baseado nas declarações dos módulos apresentados, o grafo resultante para a resolução de dependências do módulo **br.com.devmedia-app** contém os vértices e arestas descritos na **Figura 1**.

Nessa figura, as linhas de cor azul representam as dependências explícitas, ou seja, que estão descritas no *module-info.java*, nas cláusulas **requires**, enquanto as linhas de cor cinza representam as dependências implícitas ou indiretas de cada módulo em relação ao módulo base.

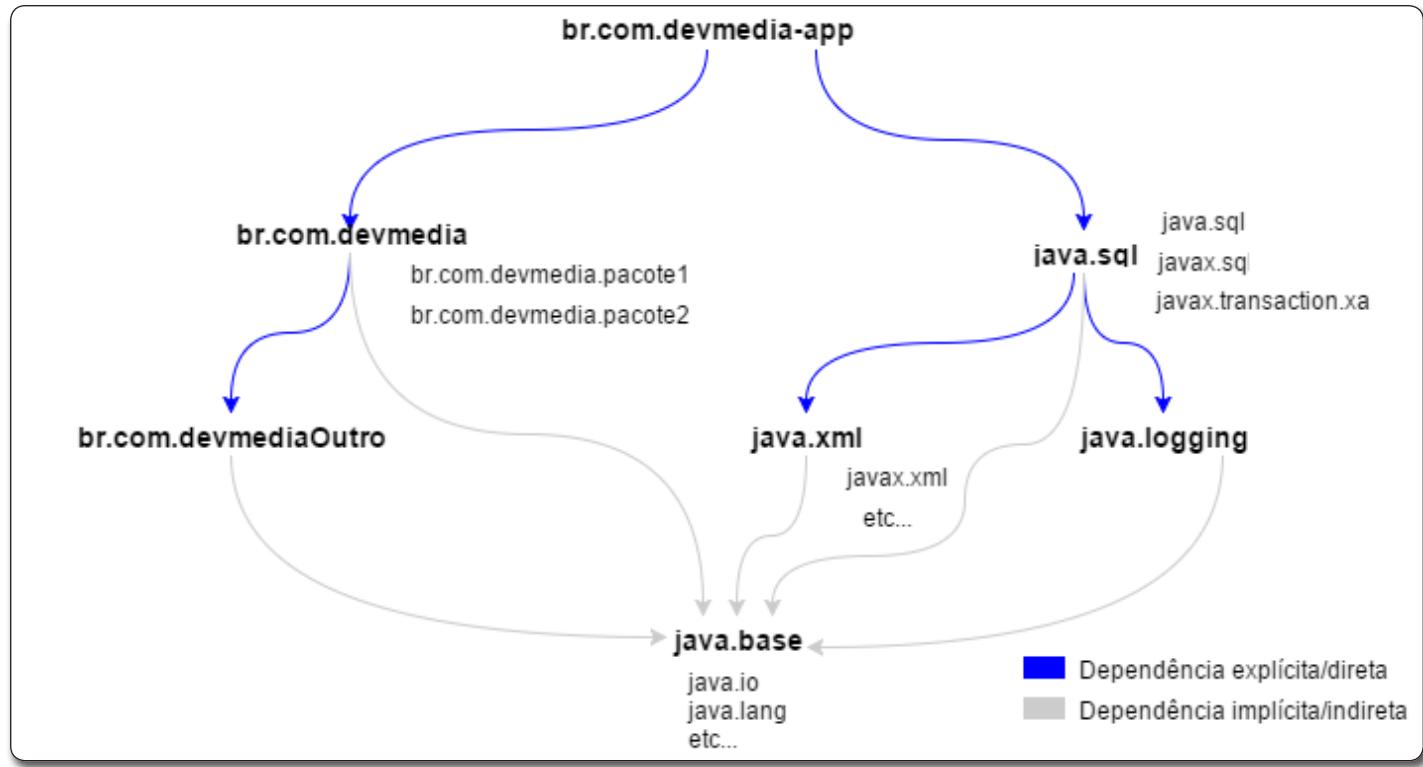


Figura 1. Grafo de módulos da aplicação

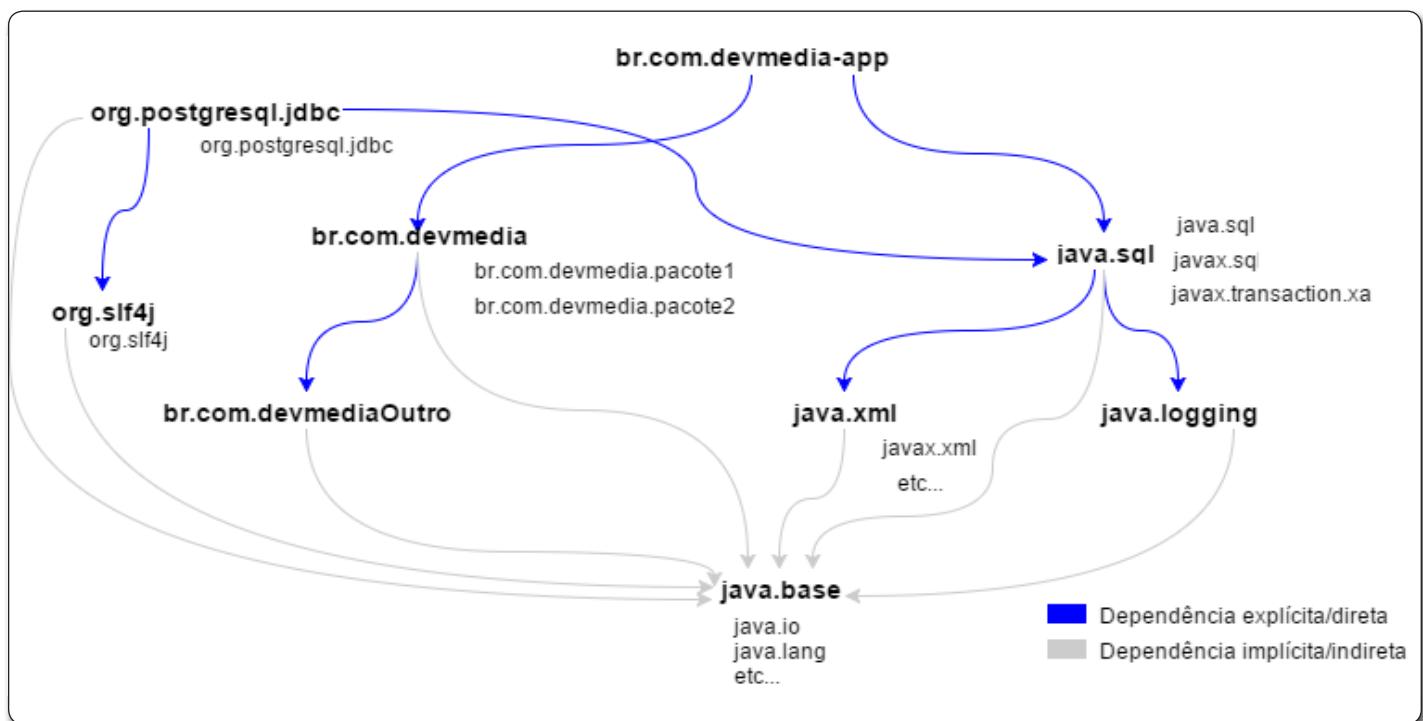


Figura 2. Grafo de Módulos com o módulo `org.postgresql.jdbc` adicionado

Serviços

Como sabemos, o baixo acoplamento de componentes de software alcançado através de interfaces de serviços e provedores de serviços é uma poderosa ferramenta para a construção de grandes sistemas. Por esse motivo, o Java suporta há um bom tempo essa técnica através da classe `java.util.ServiceLoader`, a qual é utilizada pelo JDK e também por bibliotecas e aplicações.

Essa classe localiza provedores de serviços em tempo de execução ao procurar por arquivos de configuração dos serviços na pasta *META-INF/services*. Entretanto, se um serviço é fornecido por um módulo, esses arquivos não estarão no classpath da aplicação. Sendo assim, foi preciso implementar uma nova estratégia para localizar os provedores de serviços e carregá-los corretamente, como demonstraremos a seguir.

Vejamos um exemplo: suponha que o módulo **br.com.devmedia-app** usa um banco de dados PostgreSQL e que o driver JDBC (recurso) para o mesmo seja fornecido em um módulo declarado de acordo com a **Listagem 7**.

Conforme pode ser observado nessa listagem, a declaração de exportação de `org.postgresql.jdbc` refere-se ao pacote do módulo de mesmo nome, que possui o driver JDBC do banco de dados PostgreSQL. Esse driver é uma classe Java que implementa a interface de serviço `java.sql.Driver`, contida no módulo `java.sql` (veja a nota a seguir). Além disso, para que o módulo `org.postgresql.jdbc` possa fazer uso dos módulos dos quais depende (`java.sql` e `org.slf4j`), é preciso que os mesmos sejam adicionados ao grafo de módulos em tempo de execução, conforme a **Figura 2**, tornando possível que a classe de carregamento de

serviços `ServiceLoader` instancie a classe do driver, procurando, através de reflection, por classes no pacote `org.postgresql.jdbc` que implementem a interface `java.sql.Driver`.

Listagem 7. Declaração do módulo `org.postgresql.jdbc`

```
module org.postgresql.jdbc {
    requires java.sql;
    requires org.slf4j;
    exports org.postgresql.jdbc;
}
```



Introdução ao Java 9: Conheça os novos recursos

Para realizar essas adições ao grafo, o compilador deve ser capaz de localizar os provedores de serviços nos módulos a ele acessíveis. Uma forma de fazer isso seria realizar uma busca por dependências na pasta *META-INF/services*, da mesma forma que a classe **ServiceLoader** faz em versões anteriores do Java, só que agora com módulos. Todavia, uma alternativa melhor, por viabilizar uma melhoria de performance, é possibilitar que os criadores de módulos de acesso a serviços externos (como o acesso a banco de dados através da API JDBC) forneçam uma implementação específica do serviço, através da cláusula **provides**, como pode ser observado na **Listagem 8**.

O ganho de performance se deve ao fato de que em vez de realizar uma busca por uma implementação da interface **java.sql.Driver** na pasta *META-INF/services* – a qual pode ter centenas de serviços – um módulo que define exatamente a classe que implementa essa interface evita a necessidade de realizar qualquer busca extensiva, ganhando assim tempo na compilação.

Nota

Quando um módulo depende diretamente de outro, então o código do primeiro será capaz de referenciar tipos do segundo. Para esse tipo de situação, diz-se que o primeiro módulo lê o segundo ou, da forma inversa, que o segundo módulo é legível ao primeiro. No gráfico da **Figura 1**, o módulo `br.com.devmedia-app` lê os módulos `br.com.devmedia` e `java.sql`, mas não o `java.xml`, por exemplo. Essa característica de legibilidade é a base da configuração confiável, pois permite que o sistema de módulos garanta que:

- Cada dependência é satisfeita por um único módulo;
- Dois módulos não podem referenciar um ao outro de forma a criar uma dependência cíclica, mesmo que ambos declarem os respectivos `requires` e `exports`. O compilador identificará isso como um erro;
- Cada módulo só pode fazer referência uma única vez a um pacote específico (com mesmo nome e conteúdo), mesmo que indiretamente. Exemplo: se um módulo M1 faz referência a dois módulos (M2 e M3) que dependem de um pacote Z, o compilador perceberá que Z é referenciado duas vezes. Nesse contexto, para evitar desperdício de memória o compilador só carregará esse pacote uma vez, o que faz com que as aplicações também se tornem mais eficientes;
- Se dois (ou mais) módulos definem pacotes com o mesmo nome, mas conteúdos diferentes, ambos podem ser carregados na memória sem que haja comprometimento no funcionamento da aplicação. A princípio, essa garantia pode parecer conflitante com a anterior, mas não o é por envolver dois módulos distintos;
- A configuração confiável não é somente mais confiável, mas também mais rápida, pois diferentemente do que é feito com classpathes na versão anterior do Java, quando um código em um módulo referencia um tipo contido em um pacote, há uma garantia de que aquele pacote está definido em algum módulo do qual ele depende (explicitamente ou implicitamente). Dessa forma, quando for necessário buscar pela definição de algum tipo específico, não há a necessidade de procurar por ele em vários módulos ou por todo o classpath, como era feito em versões anteriores, tornando as execuções das operações mais rápidas.

Nota

Na **Listagem 7** pode parecer ambíguo o fato de um módulo chamado `org.postgresql.jdbc` ter um pacote exportado de mesmo nome. O fato é que as implementações de drivers jdbc para bancos de dados geralmente são anteriores ao sistema de módulos do Java e devido a isso estão encapsuladas dentro de pacotes. Logo, de forma a facilitar a identificação desse pacote “legado”, definiu-se a convenção de criar um módulo com o mesmo nome do pacote que possui a implementação do driver.

Ao declarar **provides java.sql.Driver with org.postgresql.Driver**, o criador do módulo está informando que uma instância da classe `org.postgresql.Driver` deve ser criada para que seja possível utilizar o serviço `java.sql.Driver`. Por sua vez, é igualmente importante que o módulo que faz uso de um determinado serviço declare esse uso em sua própria descrição, através da cláusula **uses** (vide **Listagem 9**).

Listagem 8. Declaração do módulo `org.postgresql.jdbc`.

```
module org.postgresql.jdbc {  
    requires java.sql;  
    requires org.slf4j;  
    exports org.postgresql.jdbc;  
    provides java.sql.Driver with org.postgresql.Driver;  
}
```

Listagem 9. Declaração do módulo `java.sql`.

```
module java.sql {  
    requires public java.logging;  
    requires public java.xml;  
    exports java.sql;  
    exports javax.sql;  
    exports javax.transaction.xa;  
    uses java.sql.Driver;  
}
```

Analisando as descrições dos módulos anteriores é muito fácil ver e entender que um deles provê um serviço que pode ser utilizado pelo outro. Essas declarações explícitas permitem que o sistema de módulos garanta, em tempo de compilação, que a interface do serviço a ser utilizado (no caso, o `java.sql.Driver`) está acessível tanto para o provedor quanto para o usuário do mesmo. Mais ainda, garante que o provedor do serviço (`org.postgresql.Driver`) implementa, de fato, os serviços por ele declarados. Dessa forma, antes da aplicação ser executada é possível verificar uma série de erros que poderiam ocorrer devido à ausência de alguma dependência, o que em versões anteriores do Java só era possível em tempo de execução. Essa característica agrega mais confiança de que o software funcionará corretamente, e também viabiliza mais agilidade ao desenvolvimento de aplicações, pois diminui a necessidade de executar a aplicação com a finalidade de identificar erros.

Class loaders (carregadores de classes)

Outra mudança importante no Java é que agora cada módulo estará associado a um class loader (**BOX 2**) em tempo de execução, entretanto, cada carregador de classe pode estar associado a vários módulos. Sendo assim, um class loader pode instanciar tipos de um ou mais módulos, desde que esses módulos não interfiram entre si (exportem o mesmo pacote) e que esses tipos só sejam instanciados por um único class loader.

Essa definição é essencial para possibilitar a retrocompatibilidade com versões anteriores da linguagem, uma vez que mantém a hierarquia de class loaders da plataforma. Desse modo, os class loaders **bootstrap** e **extension** continuam a existir na nova versão

e serão utilizados para carregar as classes pertencentes a módulos da própria plataforma. Já o class loader **application/system** também foi mantido, mas com a função de carregar tipos oriundos de módulos de dependências da aplicação. Essa flexibilidade facilita a modularização de aplicações legadas, uma vez que os class loaders das versões anteriores podem ser estendidos com o intuito de carregar classes contidas em módulos com poucas alterações.

BOX 2. Class loaders

Os class loaders fazem parte do ambiente de execução do Java (JRE) e são responsáveis por carregar classes dinamicamente na máquina virtual (JVM). Logo, devem localizar as bibliotecas, ler o conteúdo delas e carregar suas respectivas classes.

Quando a JVM é iniciada, três class loaders são utilizados:

- **Bootstrap ClassLoader** – Responsável por carregar as bibliotecas da plataforma Java, esse class loader foi implementado em código nativo (não é uma classe Java) e carrega classes Java importantes, como a `java.lang.Object`, a qual é o pai de todos os objetos Java, bem como outros códigos na memória. Até a versão 8 do Java, as classes que esse class loader instancia ficavam empacotadas no `rt.jar`, presente no diretório `jre/lib` do JRE. No entanto, como na versão 9 não teremos mais arquivos JAR e sim módulos (ainda que se possa utilizar um módulo como se fosse um JAR para retrocompatibilidade), essas classes foram movidas para arquivos JMOD. A partir disso, o arquivo JMOD que contém a classe Object, por exemplo, é o `java.base.jmod`, localizado no diretório `/jmods/java.base.jmod/classes/java/lang/` do novo JDK;
- **Extension ClassLoader** – Nas versões anteriores do Java, carregava as classes oriundas de bibliotecas presentes no diretório extension do JRE. Entretanto, com o advento da modularização, as classes do diretório extension foram migradas para módulos. Logo, esse class loader continua a carregar as mesmas classes, só que agora a partir de módulos. A classe que implementa esse class loader é a `ExtClassLoader`, a qual é uma inner class na classe `sun.misc.Launcher`;
- **Application/System ClassLoaders** – Responsável por carregar classes que estão no classpath da aplicação (variável de ambiente `CLASSPATH`). É implementada pela classe `AppClassLoader`, a qual também é uma inner-class na classe `sun.misc.Launcher`. Saiba, ainda, que todas as classes criadas pelos desenvolvedores (por exemplo, as classes main) são carregadas por esse class loader.

jShell = Java + REPL

Como já mencionado, o jShell é uma ferramenta REPL de linha de comando para a linguagem Java, possibilitando assim a execução de declarações e expressões Java de forma interativa. Essa funcionalidade é importante porque permite a iniciantes na linguagem executar, de forma rápida, expressões quaisquer, acelerando o aprendizado ao fornecer uma resposta imediata na tela, sem a necessidade de realizar compilação e execução de código em uma ferramenta de desenvolvimento.

Além de ser uma ótima opção para iniciantes, o jShell viabiliza aos desenvolvedores experimentar algoritmos, criar protótipos ou até mesmo tentar utilizar uma nova API. Essa ferramenta pode ser testada por qualquer pessoa que baixar as versões early access do Java 9 e o JAR do projeto Kulla, projeto cujo objetivo é implementar o jShell.

Passos para a instalação e configuração

O primeiro passo para utilizar o jShell é baixar a versão early access do JDK 9. Feito isso, basta descompactar o arquivo para o

destino de sua preferência e configurar a variável de ambiente para apontar para esta versão do JDK. Logo após, pressione as teclas `Windows+R` para visualizar uma janela de execução que solicitará ao usuário o programa que ele deseja abrir. Neste momento, digite `cmd` e clique em `OK`.

Nota

Nesse artigo, a pasta na qual o arquivo foi extraído foi a `F:\java9\jdk-9`. Deste modo, sempre que houver esse caminho no texto, troque pelo caminho da pasta para a qual você extraiu o JDK em sua máquina.

Com o terminal de linha de comando aberto, digite `set %JAVA_HOME%=F:\java9\jdk-9` para criar uma variável de ambiente chamada `JAVA_HOME` cujo conteúdo é o caminho de instalação do JDK. Em seguida, digite `echo %JAVA_HOME%` para confirmar que a variável foi criada corretamente. Assim, só falta adicionar esta variável à variável de ambiente `Path` – utilizada pelo sistema operacional – com a execução do seguinte comando: `set Path=%JAVA_HOME%\bin;%Path%`.

Com o JDK 9 baixado e instalado, o segundo passo é instalar o jShell. Para isso, basta baixar o arquivo JAR (veja o endereço indicado na seção **Links**) e renomeá-lo para `kulla.jar`.

Utilizando o jShell

Esta seção tem o intuito de demonstrar, brevemente, as principais funcionalidades do jShell, de modo que o leitor tenha uma melhor compreensão do funcionamento da ferramenta. Portanto, detalhes de implementação serão omitidos, focando apenas nas formas básicas de uso.

O primeiro passo é executar o terminal de linha de comando, tal qual informado anteriormente: teclas `Windows + R` e digitar `cmd`.

Com o terminal aberto, mude o diretório do mesmo para o diretório no qual está o arquivo `kulla.jar`, digitando `cd /d f:\java9\kulla\`. Uma vez alterado o diretório, o jShell pode ser executado com o comando `java -jar kulla.jar`.



Introdução ao Java 9: Conheça os novos recursos

Ao utilizar o jShell no terminal é possível inserir trechos de código como declarações, variáveis, métodos, definições de classes, importações e expressões que a ferramenta automaticamente tomará as providências necessárias para executá-los e mostrar o resultado.

Deste modo, pode-se, por exemplo, digitar `System.out.println("Olá!");` sem a necessidade de criar uma classe e/ou um método para imprimir um texto na tela, conforme expõe a **Figura 3**.

```
f:\java9\kulla>java -jar kulla.jar
| Welcome to JShell -- Version 9-ea
| Type /help for help

-> System.out.println("Olá!");
Olá!
->
```

Figura 3. Imprimindo um texto com o jShell

Por padrão, o jShell mostra informações descritivas sobre os comandos que foram executados pelo usuário como, por exemplo, na definição de uma variável (vide **Figura 4**). Essa funcionalidade é importante porque permite ao usuário compreender exatamente o que o jShell fez, principalmente quando se está aprendendo a utilizar a linguagem.

```
->
int x = 45;
| Added variable x of type int with initial value 45
```

Figura 4. Inicializando uma variável no jShell

Nota

A utilização do ponto-e-vírgula na definição de variáveis é opcional no jShell.

É possível também executar expressões e cálculos em geral. Nesses casos, toda vez que o usuário fornecer uma expressão como `2+2` uma variável temporária será criada com o resultado da operação, de tal forma que o usuário possa fazer referência à mesma posteriormente. Isso pode ser verificado na **Figura 5**, na qual uma variável temporária, chamada `$3`, é criada com o resultado da expressão `2+2` e em seguida o valor dessa variável é somado ao valor da variável `x` (`valor = 45`), resultando no número 49.

E quando é necessário informar códigos com mais de uma linha, como na definição de métodos e classes, o usuário não precisa se preocupar com detalhes de implementação, pois o jShell consegue detectar que os dados fornecidos pelo usuário estão incompletos. Esse comportamento pode ser observado na **Figura 6**, na qual foi definido um método que multiplica um número inteiro por 2.

```
-> 2+2
| Expression value is: 4
| assigned to temporary variable $3 of type int

-> x+$3
| Expression value is: 49
| assigned to temporary variable $4 of type int
```

Figura 5. Operações matemáticas e expressões no jShell

```
-> int multiplicaPorDois(int x){
->     return x*2;
-> }
| Added method multiplicaPorDois(int)

-> multiplicaPorDois(2)
| Expression value is: 4
| assigned to temporary variable $6 of type int
```

Figura 6. Criando e executando um método Java no jShell

JMH – Java Microbenchmarking Harness

Outra interessante funcionalidade do novo JDK é o JMH (*Java Microbenchmarking Harness*). Esse recurso nada mais é do que uma ferramenta/biblioteca para construir, rodar e analisar nano, micro, mili e macro benchmarks (vide **BOX 3**) que foi desenvolvida pelo russo Aleksey Shipilev (vide **BOX 4**) e posteriormente integrada ao Java 9 como solução oficial para testes de desempenho na plataforma Java.

Essa ferramenta foi muito requisitada porque o Java não possuía uma opção para realizar esses tipos de medição, dependendo de soluções de terceiros e dificultando, dessa forma, a realização de testes de desempenho confiáveis.

Visto que vários fatores podem distorcer os resultados, como otimizações realizadas pelo próprio compilador ou pelo hardware, a realização desse tipo de teste é uma tarefa difícil. Saiba que quando se trata de testes de desempenho em baixíssima escala, como é o caso de microbenchmarks, cada operação que



está sendo medida leva tão pouco tempo que qualquer tarefa que for executada antes ou depois da mesma e estiver dentro do escopo de medição levará mais tempo que a operação em análise, distorcendo assim o resultado.

BOX 3. Micro/nano/mili/macro benchmarking

É a medição de performance de pequenos trechos de código que levam micro/nano/milisegundos para serem executados. Esse tipo de medição é ideal para avaliar a performance de um componente pequeno ou a chamada a um método/função de um programa, por exemplo.

BOX 4. Aleksey Shipilev

Engenheiro de performance em ambientes Java formado em Ciência da Computação pela Universidade de ITMO, em São Petersburgo, que trabalha atualmente na Oracle. Desenvolveu o JMH como um projeto de código aberto com outros colegas, e devido à qualidade deste, a Oracle optou por incorporá-lo como um framework para testes de desempenho no JDK 9, a fim de auxiliar no desenvolvimento de suites de testes de desempenho mais precisas, bem como facilitar a execução desses testes em pequenos trechos de código. Além desse, Shipilev contribui ativamente com vários projetos de código aberto.

Essa observação é importante porque é possível que o desenvolvedor tenha escrito um trecho de código que o compilador entenda, por exemplo, que não realiza nenhuma tarefa realmente necessária, sendo, portanto, removido pelo mesmo em tempo de compilação, de forma a otimizar a execução da aplicação. Com a otimização do código, no entanto, a aplicação executará mais lentamente do que sem quaisquer intervenções do compilador. Isso porque as melhorias são realizadas apenas quando a aplicação é inicializada, fazendo com que o tempo total para a mesma terminar todas as tarefas de uma determinada execução seja maior, já que contabiliza o tempo levado pelo compilador para otimizar o código, mais o tempo que o código otimizado precisa para finalizar a execução.

Deste modo, os resultados dos testes de desempenho desse trecho de código estarão distorcidos. Com isso em mente, saiba que uma das premissas para bons testes de desempenho é fazer com que a contagem do tempo de execução do teste só inicie após as otimizações no código da aplicação, viabilizando testes que possibilitem obter, com precisão, o tempo levado pela mesma para realizar as tarefas. Esse é o propósito do JMH.

Enfim, com o JMH será possível escrever testes de desempenho mais confiáveis, uma vez que o mesmo é fortemente integrado à plataforma Java, o que facilita o monitoramento do gerenciamento de memória da JVM, das otimizações do compilador, entre outros fatores que podem distorcer os resultados dos testes.

Novo garbage collector padrão (G1)

Um dos equívocos que grande parte dos desenvolvedores Java cometem é pensar que a JVM possui apenas um coletor de lixo (*garbage collector*), quando

na verdade dispõe de quatro: o serial, o paralelo, o cms e o G1. Relacionado a isso, recentemente a Oracle percebeu que boa parte dos usuários tem uma melhor experiência com as aplicações quando elas rodam sobre um ambiente que faz uso do coletor de lixo G1, em vez do paralelo, que era utilizado como padrão. Assim, a empresa fez uma proposta de melhoria à comunidade, através da JEP 248, para tornar o G1 o coletor padrão a partir do Java 9.

Nota

Com o objetivo de garantir a precisão do tempo medido, o JMH faz uma série de operações que permitem que a contagem do tempo só comece após as otimizações do compilador.

Nota

Não faz parte do escopo deste artigo entrar em detalhes sobre o funcionamento do JMH e como escrever testes de desempenho com essa ferramenta.

O coletor de lixo paralelo (*Parallel GC*) tem a vantagem de utilizar vários processos Java para fazer a limpeza da memória e compactá-la, liberando o espaço mais rapidamente para a alocação de mais recursos. Por outro lado, a desvantagem é que para conseguir realizar a limpeza a JVM precisa pausar os processos do software que está rodando, escanear toda a memória do mesmo, com o intuito de identificar quais objetos não podem ser acessados, e descartar esses objetos, para então voltar a executar os processos. Essa paralisação é crítica para grandes aplicativos que precisam ser altamente responsivos, pois quanto mais memória o aplicativo usa, maior o tempo de paralisação e resposta. Um dos motivos para isso acontecer é que assim como os coletores de lixo serial e CMS, o paralelo estrutura a memória utilizável pela JVM em três seções: *young generation*, *old generation* e *permanent generation*, cada uma com um tamanho fixo (veja a [Figura 7](#)).

Por sua vez, o coletor de lixo G1 é mais moderno, tendo sido incluído na atualização 4 do JDK 7, e foi desenvolvido visando

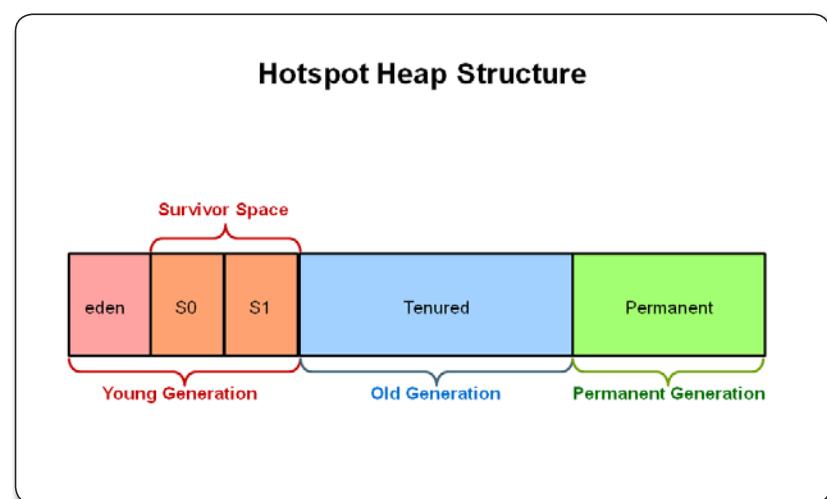


Figura 7. Estrutura de memória da JVM com coletor de lixo diferente do G1

a necessidade de suportar melhor uma quantidade de memória disponível para a JVM superior a 4GB, o que passou a ser bastante comum em máquinas mais modernas e, principalmente, em servidores. Para isso, o G1 possui outra estratégia de gerenciamento: dividir a área de memória em blocos de mesmo tamanho, cada um como um pedaço contíguo da memória virtual.

Assim, cada bloco pode assumir o estado de *eden*, *survivor* ou *old* para designar que pertencem às áreas de mesmo nome, tal qual ocorre nos outros coletores, com a diferença de que não existe um número fixo para a quantidade de blocos para cada área, conforme demonstra a Figura 8. Portanto, o tamanho dessas áreas pode variar de acordo com a necessidade do coletor de lixo, o que garante uma maior flexibilidade no uso da memória.

No momento de executar a coleta de lixo na memória, o G1 percorre todas as três regiões (*eden*, *survivor* e *old generation*) e faz a transição de objetos alocados de uma região para a outra, conforme o tempo de vida de cada um. Em seguida, inicia a desalocação dos objetos pela região mais vazia até a mais cheia, potencializando a liberação de memória. Esse método assume que as regiões mais vazias têm mais probabilidade de possuir objetos que podem ser descartados.

Nota

É importante mencionar que o G1 não executa a coleta de lixo a todo momento. Em vez disso, os desenvolvedores especificam um tempo de pausa aceitável para a aplicação e, baseado nisso, o coletor de lixo adota um modelo de predição para alcançar esse tempo desejado. Além disso, considerando dados de outras coletas, estima quantas regiões podem ser coletadas durante o período especificado.

G1 Heap Allocation

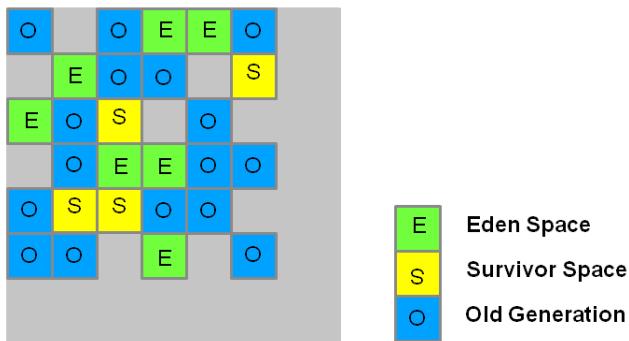


Figura 8. Estrutura de memória da JVM com o coletor de lixo G1

HTTP 2.0

A versão 2.0 do protocolo HTTP foi definida como padrão em fevereiro de 2015 pelo IESG (*Internet Engineering Steering Group*) e até o fim desse mesmo ano os navegadores web mais utilizados adicionaram o suporte à mesma. De forma simples, podemos descrever o HTTP 2.0 como uma evolução da versão 1.1, tendo sido baseada no protocolo aberto SPDY, cujo desenvolvimento foi iniciado pelo Google.

Como o HTTP 1.1 data de 1999 e não recebeu melhorias desde então, essa versão possui uma série de limitações/problemas, principalmente devido à evolução da web, a qual passou de uma coleção de hyperlinks e textos para a necessidade de viabilizar a transmissão de conteúdos mais pesados, como áudios, vídeos, interfaces mais complexas, animações, entre outras coisas. Essa evolução evidenciou as limitações do protocolo e serviu de motivação para a criação de uma versão mais nova, a 2.0.

Para compreender melhor esse cenário, considere a seguinte analogia: imagine que todas as ruas do mundo tivessem sido construídas na época das carroças e fossem estreitas, esburacadas e com baixo limite de velocidade. Provavelmente levaria um bom tempo para chegar de um ponto a outro, principalmente por causa da velocidade dos cavalos. Agora, imagine que nada mudou nas estradas, mas ao invés de cavalos são utilizados carros. Nesse cenário, a velocidade do cavalo deixa de ser um problema, mas congestionamentos começam a surgir porque a quantidade de carros aumentou (a população cresceu) e eles ocupam mais espaço.

O que ocorre com o tráfego de dados na web não é muito diferente dessa analogia. O protocolo HTTP foi criado há 25 anos e, como já citado, a versão mais atualizada foi padronizada apenas em 1999. Todo esse tempo é uma eternidade quando lidamos com tecnologia da informação, pois em nossa área as evoluções são muito frequentes. Portanto, da mesma forma que as estradas estreitas e esburacadas de outrora, a web, quando da padronização do protocolo HTTP, era bem diferente da atual: as páginas web eram pequenas, as conexões de internet, mais lentas, e os recursos



de hardware, limitados e caros. De certa forma, as máquinas e a ausência de banda larga eram o gargalo para “viagens” mais rápidas.

Quando um navegador carrega uma página web utilizando o HTTP 1.1, ele só pode solicitar um recurso (uma imagem, um arquivo JavaScript, etc.) por vez a cada conexão com o servidor, conforme a **Figura 9**. Deste modo, note que o navegador passa um bom tempo apenas aguardando a resposta de cada solicitação.

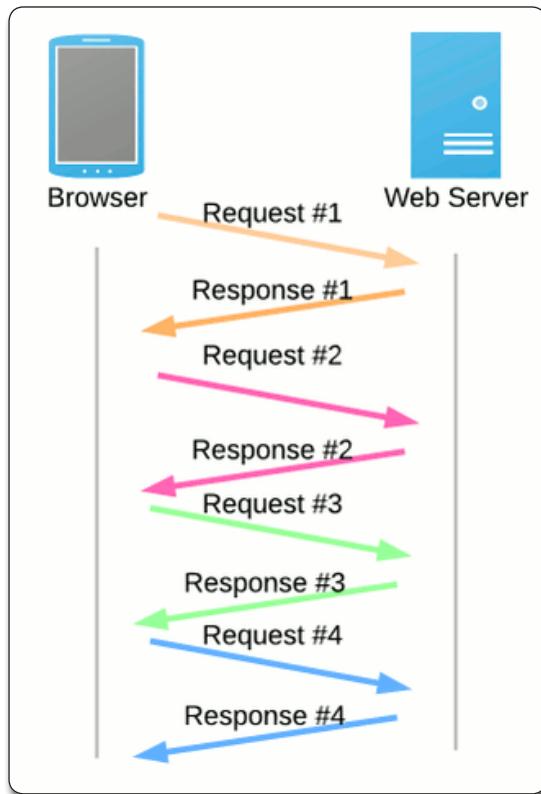


Figura 9. Modelo de comunicação do HTTP 1.1

Já que o HTTP 1.1 não permite que sejam feitas múltiplas solicitações em paralelo numa mesma conexão, pois analogamente ao caso das estradas, só há uma faixa para ir e voltar, e os navegadores tentam contornar essa limitação criando mais uma conexão com o servidor (criando uma nova faixa na estrada), como pode ser observado na **Figura 10**.

Acontece que utilizar duas conexões melhora um pouco a velocidade, mas não resolve o problema, pois como pode ser observado na imagem, o navegador ainda passa um bom tempo esperando a resposta da solicitação e só conseguirá baixar dois recursos por vez. Embora seja possível criar até seis conexões com o servidor de destino, essa ainda não é uma boa opção, porque cada conexão será usada de maneira ineficiente, desperdiçando tempo.

Neste cenário, se levarmos em conta que uma página comum em um site moderno possui, por exemplo, 100 recursos, o tempo para baixar cada recurso culmina em uma página que será carregada lentamente.

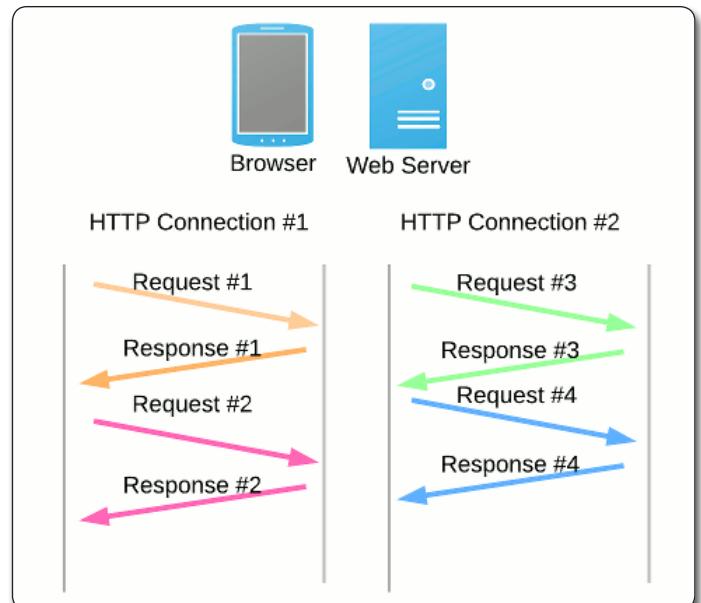


Figura 10. Navegadores tentam paralelizar criando mais conexões

Essa ineficiência para carregar recursos é o motivo da adoção por muitos desenvolvedores web da compactação de arquivos JavaScript e CSS, pois uma vez o recurso possuindo um tamanho menor, o site carregará mais rápido. Entretanto, medidas como essas são apenas formas de contornar o problema e não de resolvê-lo. Embora seja recomendado continuar com a otimização das páginas web para baixar cada vez menos recursos e em tamanhos menores, o problema não será resolvido até que haja uma alteração fundamental na forma de comunicação do protocolo HTTP, maximizando a utilização das conexões e minimizando o tempo de espera/resposta. Eis que surge o HTTP 2.0.

A solução

O HTTP/2 busca fazer melhor uso das conexões com os servidores web ao modificar como as solicitações e respostas trafegam pela rede, o que era uma grande limitação da versão anterior do protocolo. Com a nova versão, o navegador realiza uma única conexão com o servidor web e então pode, de acordo com a necessidade, efetuar várias solicitações e receber várias respostas ao mesmo tempo, como sinaliza a **Figura 11**.

De acordo com essa figura, o navegador utiliza uma única conexão e enquanto está enviando mais uma solicitação ao servidor (#6), também está recebendo várias respostas para solicitações feitas anteriormente como, por exemplo, o cabeçalho para o arquivo #3, o corpo do arquivo solicitado pela requisição #1, o corpo do arquivo solicitado pela requisição #3, cujo cabeçalho já foi recebido, e o cabeçalho do arquivo solicitado pela requisição #2.

Note que com apenas uma conexão o navegador consegue enviar várias requisições e receber várias respostas, em ordem diversa, otimizando assim o tráfego de dados. Note, também, que as respostas para uma única solicitação podem ser fragmentadas em vários pedaços menores, como aconteceu com os dados

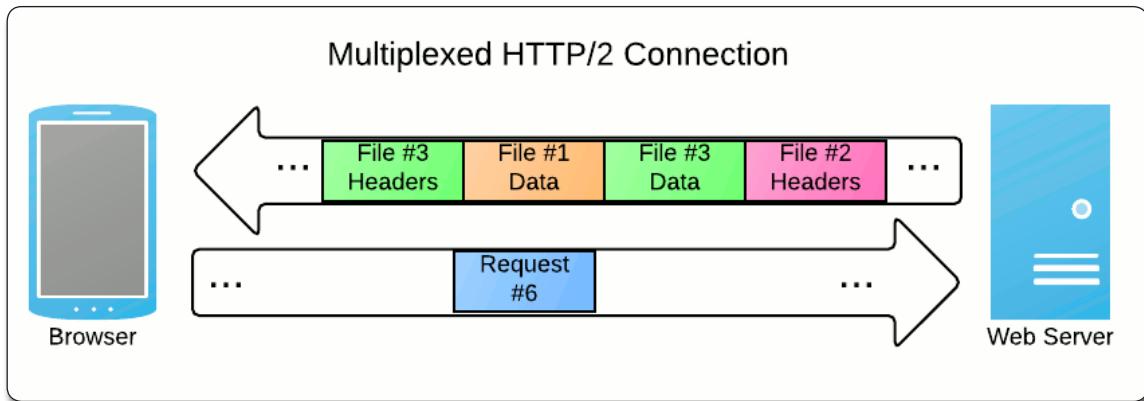


Figura 11. Modelo de comunicação do HTTP 2.0

solicitados pela requisição #3. Isso é importante porque permite que o servidor responda mais rápido, ainda que com partes menores. Além disso, evita que o servidor só processe a requisição seguinte após terminar a atual – o que causa gargalo no tráfego das informações – visto que agora tem a capacidade de enviar várias respostas de uma única vez. Logo, ao dividir o tráfego em pequenos blocos, tanto o cliente (que faz a solicitação) quanto o servidor (que devolve o que o cliente pediu) passam a ser capazes de lidar com múltiplas solicitações e respostas, aumentando, portanto, a velocidade do diálogo entre as partes.

Analogamente, essa situação seria como ir a um supermercado e pedir a alguém para pegar uma lista completa de itens de uma única vez, por exemplo: “Pegue o leite, os ovos e a manteiga”. Logo, a pessoa retornará com todas as coisas da lista solicitada. No HTTP 1.1, seria necessário pedir um item por vez, por exemplo: “Pegue o leite” → “Aqui está o leite”, “Agora pegue os ovos” → “Aqui estão os ovos” e assim sucessivamente.

É fácil observar que é muito mais eficiente pedir todos os itens de uma só vez, economizando tempo e, assim, agilizando todo o processo. Em outras palavras, toda a comunicação é muito mais

eficiente fazendo uso de uma única conexão com multiplexação, como ocorre no HTTP 2.0. Essa abordagem, apesar de um pouco mais complexa de entender, é fundamental para uma internet cada vez mais rápida.

Em primeiro lugar, as conexões não precisarão esperar que um recurso termine de ser transferido para atender a outras solicitações. Por exemplo, suponha que existe uma página web com várias imagens (algo como o Google Imagens) e o navegador precisa fazer uma requisição para receber cada uma. Com a multiplexação, em vez de esperar que o servidor termine de enviar todas as partes de uma imagem para que possa requisitar a próxima, ele pode solicitar um conjunto de imagens ou todas elas de uma vez e começar a recebê-las em partes. Nesta proposta, caso seja menor e possua menos partes a serem baixadas, a segunda imagem, por exemplo, pode ser renderizada antes mesmo da primeira. Essa característica previne um problema bastante conhecido na versão anterior, chamada de *head-of-line blocking* (bloqueio do primeiro da fila, em tradução livre), que ocorre quando um recurso de tamanho grande (por exemplo, uma imagem de fundo de 1MB) bloqueia todos os outros recursos de serem baixados até que ele seja baixado completamente.

Outra vantagem do HTTP 2 é o conceito de *Server Push*. Este representa a capacidade de um servidor de enviar um recurso ou conteúdo para um cliente/navegador web sem que o mesmo tenha solicitado. Por exemplo, quando um navegador visita um site, o servidor envia o logotipo ao mesmo sem que seja necessário haver uma solicitação. Essa atuação proativa do servidor permite que os recursos sejam carregados muito mais rápido do que anteriormente.

Vale ressaltar, ainda, que o HTTP 2 funciona melhor utilizando a proteção dos dados trafegados via SSL. A conjunção entre o protocolo HTTP e a criptografia SSL é chamada de protocolo HTTPS. Saiba que, embora seja possível fazer a transferência de dados sem essa proteção na versão 2, o protocolo SPDY exigia a utilização do HTTPS. Deste modo, por questões de compatibilidade com o SPDY, boa parte dos servidores web requer a segurança via SSL, o que acaba sendo positivo, pois força a proteção dos dados dos usuários na comunicação entre computadores.



O que isso significa para o Java?

O Java, desde a versão 1.0, provê a utilização do protocolo HTTP. Entretanto, grande parte do código que garante esse suporte é de um período anterior ao surgimento desse protocolo e por conta disso foi baseado em bibliotecas de comunicação independentes de redes (classe **URL**, por exemplo). Nesse ponto é interessante lembrar que quando o protocolo foi inserido na API do Java não era esperado que a web pudesse se tornar tão dominante e acessível como é hoje. Além disso, nesse mesmo período, como o HTTPS ainda não tinha sido criado, por motivos óbvios não foi incluído na API, o que ocorreu posteriormente.

Lembre-se, ainda, que o protocolo HTTP e a web possuem décadas de existência e somando-se o fato de não terem acompanhado a demanda (uso de streaming, interfaces responsivas e mais pesadas, entre outros), apresentam hoje uma baixa eficiência para trafegar dados. Como se não pudesse piorar, atualmente a preocupação com a segurança dos dados é tão grande que mesmo os sites mais simples estão adotando o HTTPS. No entanto, esse protocolo é ainda mais lento que o HTTP, pois necessita criptografar e descriptografar tudo o que transporta. Como resultado de tudo isso, a baixa capacidade de lidar com um mundo cada vez mais ágil e conectado e volumes de dados cada vez maiores através da rede culmina em críticas por parte dos usuários, que têm a sua experiência de navegação e utilização comprometidas.

Para dificultar ainda mais, é válido ressaltar que o recurso fornecido pelo JDK ao HTTP também possui grandes limitações, não permitindo, por exemplo, fazer uso do mesmo na íntegra, ou seja, os desenvolvedores, além de terem que lidar com as deficiências do protocolo, não conseguem explorar todos os seus recursos. Essa deficiência do JDK faz com que muitos desenvolvedores adotem bibliotecas de terceiros, como a Apache HttpComponents, o que torna a criação de uma API que forneça suporte completo ao HTTP/2 algo primordial para o Java, visto que esse será um dos recursos mais importantes durante a próxima década.

Em vista disso, a nova API do HTTP – contida no Java 9 – servirá de base para codificar com a versão 2 do protocolo. Para isso, a API Java que era compatível com a versão 1 do HTTP teve seu código reescrito, valendo-se da necessidade de ser independente da versão do HTTP (retrocompatível) e oferecer formas de utilizar as funcionalidades do HTTP/2. Ademais, para utilizar esse protocolo em aplicações Java será bastante simples, bastando implantá-las em servidores web que funcionem com a nova versão desse protocolo. Essa facilidade é um incentivo ao desuso de soluções incomuns (*hacks* ou gambiarras) que buscavam contornar as limitações do HTTP 1.1 nas aplicações legadas.

Melhorias na API de processos

Atualmente o Java fornece um recurso limitado para o controle de processos (threads ou tarefas) do sistema operacional, pois provê apenas uma API básica para configurar o ambiente e inicializar tarefas, dificultando a manipulação das mesmas. Para conseguir o número de identificação de um processo (PID - *Process ID*), por exemplo, é preciso usar código nativo do SO. Assim, caso a aplicação seja

executada em diferentes sistemas operacionais, é necessário ter uma versão implementada com o código específico para cada um.

A **Listagem 10** expõe o código para recuperar o PID de um processo no Linux utilizando a versão atual da API. Note que há, na segunda linha, um código que faz uso de comandos nativos (por exemplo, */bin/sh*) para realizar uma tarefa simples. Com isso, a implementação se torna mais difícil, o mesmo vale para a manutenção, e o código ainda estará mais sujeito a erros. Enfim, é fundamental oferecer uma solução que evite o uso de código nativo para controlar processos.

Listagem 10. Impressão do número de identificação de um processo utilizando a versão atual da API.

```
public static void main(String[] args) throws Exception{  
  
    Process proc = Runtime.getRuntime().exec(new String[]{"./bin/sh","-c",  
    "echo $PPID"});  
  
    if(proc.waitFor() == 0)  
    {  
        InputStream in = proc.getInputStream();  
        int available = in.available();  
        byte[] outputBytes = new byte[available];  
        in.read(outputBytes);  
        String pid = new String(outputBytes);  
        System.out.println("ID do processo:" + pid);  
    }  
}
```

Com base nisso, o Java 9, através da JEP 102, aprimora a API de processos oferecendo uma solução simples, mas com várias funcionalidades, que permite um maior controle sob os processos do sistema operacional. Dentre as funcionalidades, destaca-se a capacidade de obter o PID ou outros detalhes do processo da JVM sobre a qual a aplicação está rodando, bem como informações de processos criados através da API, como pode ser observado nas **Listagens 11 e 12**.

Na primeira listagem temos a implementação de uma classe que tem a finalidade de obter as informações do processo da JVM que está executando a aplicação e imprimi-las no terminal do sistema operacional. Para isso, a variável **processoAtual**, da classe **ProcessHandle**, obtém o controle do processo que está em execução através do comando **ProcessHandle.current()**. Em seguida, há a execução do **System.out.println()**, que imprime um cabeçalho, e uma chamada ao método **imprimeDetalhesProcesso()**, que recebe como parâmetro **processoAtual**. Esse método obterá as informações do processo através do comando **processoAtual.info()** e irá armazená-las na variável **processoAtualInfo**, da classe **ProcessHandle.Info**. Feito isso, com as informações do processo em mãos, basta imprimi-las no terminal através do **println()**, que pode ser observado no restante do código.

Já na **Listagem 12** temos a implementação de uma classe que cria um processo que abre o terminal do Windows e lista todas as informações desse processo. Essa classe inicia sua execução através da chamada ao método **exec()** – **Runtime.getRuntime().exec("cmd /k dir")** – o qual retorna um objeto **Process** que

Introdução ao Java 9: Conheça os novos recursos

representa o processo recém-criado para a execução do comando `cmd /k dir`, passado como parâmetro. Em seguida, o método `imprimeDetalhesProcesso()`, cujo código pode ser visto na [Listagem 11](#), é invocado, recebendo como parâmetro um controlador do processo (`ProcessHandle`) obtido através de `processo.toHandle()`.

Listagem 11. Impressão na tela das informações do processo atual com a nova API.

```
public class ExemploProcessoAtual{
    public static void main(String[] args){
        //Pega o controle (handle) do processo em andamento
        ProcessHandle processoAtual = ProcessHandle.current();
        System.out.println("**** Informacao do processo atual ****");
        imprimeDetalhesProcesso(processoAtual);
    }

    public static void imprimeDetalhesProcesso(ProcessHandle processoAtual){
        //Retorna uma instância do objeto que referencia as informações do processo
        ProcessHandle.Info processoAtualInfo = processoAtual.info();
        if (processoAtualInfo.command().orElse("").equals("")){
            return;
        }
        //Imprime o id do processo na tela
        System.out.println("PID: " + processoAtual.getPid());
        //Imprime o comando de início do processo na tela. Se não tiver um comando,
        //retornar vazio - orElse("")
        System.out.println("Comando: " + processoAtualInfo.command().orElse(""));
        //Imprime os parâmetros do comando de início do processo na tela
        String[] parametros = processoAtualInfo.parametros().orElse(new String[]{});
        if (parametros.length != 0){
            System.out.print("parametros:");
            for(String parametro : parametros){
                System.out.print(parametro + " ");
            }
            System.out.println();
        }
        //Imprime o horário de início do processo na tela
        System.out.println("Início: " + processoAtualInfo.startInstant().orElse(Instant.now()).toString());
        //Imprime o tempo que o processo está rodando
        System.out.println("Rodando:" + processoAtualInfo.totalCpuDuration()
            .orElse(Duration.ofMillis(0)).toMillis() + "ms");
        //Imprime o nome do usuário ao qual o processo está vinculado
        System.out.println("Usuario: " + processoAtualInfo.user().orElse(""));
    }
}
```

Listagem 12. Código que cria um processo e imprime suas informações.

```
import java.io.IOException;
import java.io.*;
import java.util.*;

public class Exemplo2{
    public static void main(String[] args) throws IOException{
        //Cria um processo
        Process processo = Runtime.getRuntime().exec("cmd /k dir");
        System.out.println("**** Detalhes do processo criado ****");
        //Imprime detalhes do processo criado
        imprimeDetalhesProcesso(processo.toHandle());
    }

    public static void imprimeDetalhesProcesso(ProcessHandle processoAtual){
        //conteúdo igual ao mesmo método na listagem 11
    }
}
```

A [Listagem 13](#), por sua vez, apresenta um código que demonstra a capacidade de listar processos que estão rodando no sistema operacional, bem como suas propriedades (PID, nome, estado, etc.). Contudo, como a lista de processos é muito grande, para demonstrar mais um recurso da nova API codificaremos alguns filtros para limitar o resultado. Assim, filtraremos os processos que possuam um comando de inicialização associado, de forma a não obter subprocessos – uma vez que eles não possuem comandos para inicializá-los –, e limitaremos a seleção a três processos.

Nota

Quando se fala em subprocessos não ter comandos para inicializá-los, não quer dizer que eles não realizam tarefas, mas sim que os processos que os gerenciam que são responsáveis por inicializá-los, manipulá-los e encerrá-los.

Como precisamos acessar cada processo para obter suas informações, utilizamos o método `ProcessHandle.allProcesses()`. Esse método retorna uma stream e a partir dela realizamos as filtragens com o comando `.filter(processHandle -> processHandle.info().command().isPresent())`, que usa o controlador para acessar as informações dos processos que têm um comando associado (`isPresent()`). Em seguida, pode-se identificar que a lista será restrita a três processos com a chamada a `.limit(3)`. Por último, tem-se a etapa da execução do código que envolve imprimir as informações de cada processo da lista final. Para isso, é chamado o método `forEach()`, que percorre a lista de processos e imprime no terminal de linha de comando as informações dos mesmos valendo-se do método `imprimeDetalhesProcesso()`.

Listagem 13. Imprime as informações de três processos que estão rodando no sistema operacional

```
public class ExemploListagemTodosProcessosSO{
    public static void main(String[] args){
        //lista os processos que estão rodando no sistema operacional
        ProcessHandle.allProcesses()
            .filter(processHandle -> processHandle.info().command().isPresent())
            .limit(3)
            .forEach((processo) ->{
                imprimeDetalhesProcesso(processo);
            });
    }

    public static void imprimeDetalhesProcesso(ProcessHandle processoAtual){

        //conteúdo igual ao mesmo método na listagem 11
    }
}
```

As demonstrações realizadas nas listagens anteriores, principalmente na [Listagem 10](#), reforçam a necessidade de implementação de uma solução para tal problema. Felizmente, a nova versão da API de processos resolve essa antiga pendência, permitindo controlar centenas de processos com uma única thread, diminuindo a complexidade da codificação e maximizando o desempenho das soluções que fazem uso desse recurso.

Portanto, com a nova API os desenvolvedores terão muito mais controle sobre os processos em um sistema operacional, além do óbvio ganho em produtividade e segurança. Apesar disso, é recomendado muito cuidado ao utilizar todo esse controle, pois pode interferir na estabilidade do ambiente que roda a aplicação e por isso é imprescindível ter um conhecimento aprimorado desses recursos.

Como verificado, o Java 9 trará melhorias importantes e há muito esperadas pela comunidade, empresas e desenvolvedores que fazem uso da plataforma. Essas novidades, em sua maioria, têm como meta obter ganhos em termos de desempenho e, principalmente, escalabilidade, a fim de atender também dispositivos com pouco poder de processamento. Sendo assim, podemos afirmar que a novidade de maior destaque é mesmo o sistema de módulos.

Essa nova estrutura do Java possui muitas facetas e por isso alguns detalhes não foram descritos neste documento, devido ao espaço. No entanto, grande dos desenvolvedores precisa conhecer apenas alguns conceitos, como os apresentados, pois são os que realmente serão utilizados no dia a dia, a exemplo da declaração de módulos e da criação de arquivos JAR modulares.

Em relação ao JShell, espera-se que seja uma ferramenta notável para diminuir, principalmente, a curva de aprendizado da linguagem. Logo, a expectativa é que os aprendizes façam bastante uso da mesma, como também os usuários mais avançados, a fim de testar alguns comandos e expressões antes de efetivamente inseri-los nas aplicações. Ademais, essa funcionalidade aproxima a linguagem Java das linguagens de script, as quais já possuem soluções como essa por padrão.

O JMH, por sua vez, é um recurso que deverá ser explorado por desenvolvedores mais avançados, que possuem conhecimento em testes de desempenho, visto que é difícil obter resultados precisos com esse tipo de teste. Entretanto, é uma API de extrema importância, uma vez que o Java não possuía uma forma de implementar tais testes sem recorrer a bibliotecas de terceiros, o que dificultava bastante a obtenção de resultados satisfatórios.

Já a definição do G1 como o coletor de lixo padrão é uma alteração bem-vinda e bastante esperada pela comunidade, após longos debates acerca do assunto. Essa alteração permitirá menos pausas nas aplicações e garantirá uma melhor experiência ao usuário. Nesse momento algumas pessoas podem argumentar que a opção de escolher o G1 como coletor de lixo já existia nas versões anteriores, porém isso não deixa essa mudança menos importante, pois muitos desenvolvedores e administradores não sabiam da existência dessa opção.

Com relação ao HTTP 2.0, é um protocolo que promete ser uma revolução na Internet, ao oferecer maior capacidade de transferência de conteúdo através da multiplexação e compressão de dados. Essa é uma evolução indispensável e que vem para suprir as deficiências da versão atual do HTTP. Apesar de ainda pouco adotado, por ser bastante recente, a tendência é que seu uso cresça exponencialmente ao longo dos anos. Portanto, é fundamental que o Java se antecipe e disponibilize uma API que possibilite o uso do novo protocolo nas aplicações, o que se materializará com o advento da versão 9 da plataforma.

Por último, temos as melhorias na API de processos, que embora pareçam de menor relevância, serão de grande ajuda para assegurar a manutenibilidade do código de aplicações, frameworks e servidores de aplicação (WildFly, GlassFish, WebLogic, Jetty, etc.) que necessitam fazer uso de paralelismo ou acessar processos do sistema operacional. Essas mudanças, assim como a modularização, exigiram bastante trabalho por parte da comunidade, pois foi necessário modificar a API original de cada plataforma para a qual a JVM está disponível.

Como verificado, com o Java 9 grandes avanços serão possíveis, o que torna imprescindível aos desenvolvedores dominar as novidades a fim de tirar o melhor proveito de todos os recursos. Com esse intuito, neste artigo começamos a dar os primeiros passos.

Autor



José Guilherme Macedo Vieira

jguilhermemv@gmail.com

Arquiteto de Software e desenvolvedor Java EE de uma das empresas públicas de tecnologia da informação mais respeitadas do país, a DATAPREV. Possui mais de 10 anos de experiência na plataforma Java. Projetou soluções de alta escalabilidade, fazendo uso, em especial, da plataforma Java EE. Tem ministrado diversos treinamentos in-company, bem como participado de projetos inovadores na área de Big Data. Pesquisador e entusiasta do Apache Cassandra, com o qual trabalhou em projetos em áreas como análise de riscos e detecção de fraudes. Colaborador da revista Java Magazine.



Links:

Modularidade do Java 9.

<http://paulbakker.io/java/java-9-modularity/>

Especificação do sistema de módulos.

<http://openjdk.java.net/projects/jigsaw/spec/sotms/>

Especificação para tornar o G1 o coletor de lixo padrão.

<http://openjdk.java.net/jeps/248>

Especificação do JMH.

<http://openjdk.java.net/jeps/230>

Especificação das melhorias da API de processos.

<http://openjdk.java.net/jeps/102>

Especificação da API do HTTP/2.

<http://openjdk.java.net/jeps/110>

Projeto Kulla

https://adopt-openjdk.ci.cloudbees.com/view/OpenJDK/job/langtools-1.9-linux-x86_64-kulla-dev/lastSuccessfulBuild/artifact/

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Simplificando o desenvolvimento de microsserviços com o WildFly Swarm

Conheça neste artigo uma nova abordagem da Red Hat para criar microsserviços utilizando Java

A expansão da internet no Brasil e no mundo traz consigo novas necessidades para quem trabalha com toda a espécie de ferramenta ou equipamento que faz esse grande universo de conteúdo virtual girar. Esse ávido anseio por um mundo mais conectado gera novos desafios para vários campos da computação. Desafios esses que vão desde a necessidade de hardwares cada vez mais potentes até a preocupação cada vez maior com questões relacionadas à segurança e performance. Isso acaba por surtir um impacto direto no desenvolvimento de novas ferramentas para esse mundo mais conectado.

Diante disso, atualmente, uma aplicação serve como uma ferramenta estratégica da empresa, funcionando de maneira similar a um hub de informações e fluxos dentro dela, ou mesmo como um produto final entregue aos seus clientes. Nessa nova realidade, arquitetos e desenvolvedores presenciam uma verdadeira revolução que vem acontecendo em nossa área, impulsionada pelas questões supracitadas que se tornam a cada dia mais críticas.

Escalabilidade e alta disponibilidade, por exemplo, nunca foram itens tão essenciais, exigindo que as equipes de desenvolvimento estejam em constante evolução. Para arquitetos de software e gerentes, no entanto, ainda pesa outro elemento no momento de planejar a solução, o custo. Apesar de ser um fator que deve ser considerado

Fique por dentro

Este artigo apresenta o Swarm, nova abordagem oferecida pela Red Hat para implementar microsserviços em Java. Assim, desenvolvedores que desejam aprender sobre o assunto podem utilizar esse conteúdo como ponto de partida para entender como migrar suas aplicações ou criá-las do zero adotando essa ferramenta. Além disso, vamos analisar o atual estado do projeto, funcionalidades que já são suportadas e a expectativa para as próximas versões.

em qualquer empresa em nível global, no Brasil ainda temos a questão da alta do dólar, que atua como um agravante.

Quando falamos em aplicações que possuem um grande fluxo de usuários, como grandes portais, existe a tendência de disponibilizar seus serviços através de servidores localizados em nuvem, como a Amazon ou o Microsoft Azure. Contudo, apesar de grandes empresas fornecedoras de serviços em nuvem já disponibilizarem servidores no Brasil ou pelo menos na América Latina, a contratação desses serviços acaba sendo afetada diretamente pela variação cambial, por ser taxada em dólar. Ainda assim, mesmo sofrendo com aspectos sobre os quais não temos controle, podemos melhorar o custo otimizando código, além de um setup mais refinado da estrutura que funciona como base.

Para tanto, lembre-se que grandes aplicações exigem também uma equipe alinhada. Assim, ferramentas que facilitem a

comunicação e a gestão das pessoas ganham importância por organizarem e disseminarem informação de maneira rápida e ágil. O aplicativo Slack é uma solução que aposta fortemente na comunicação, não somente entre pessoas, mas também entre equipes, departamentos, filiais e ferramentas, oferecendo integração a uma vasta gama de opções. Nesse contexto, aplicações como o Octopus, Deploybot e Jenkins podem, através dela, manter os interessados informados de um modo mais eficiente e sem a necessidade de criar hooks dentro das mesmas para estabelecer a relação com o Slack, possibilitando uma melhor comunicação sem grande esforço. Desse modo, todos os envolvidos nas diferentes fases do projeto conseguem ser notificados de uma maneira mais rápida e simples.

Nesse cenário, o desenvolvimento também é atingido por tais mudanças, entrando em cena novas ferramentas que oferecem a oportunidade de fazer mais com menos e de forma mais rápida. Várias delas que há poucos anos nem existiam e hoje são referências de mercado, como é o caso do MongoDB; ou você, há seis anos, pensaria em usar uma ferramenta para armazenar seus dados que não adota pelo menos as três formas normais?

No que tange a arquitetura de software, um modelo que vem sendo adotado por grandes portais, inclusive no Brasil, é o modelo de microserviços. Impulsionada por especialistas como Martin Fowler, a ideia ganha apoiadores a cada dia. Em um modelo monolítico, onde a aplicação é pensada e construída como um bloco único, qualquer necessidade de aumentar a performance precisa ser pensada e realizada de um modo geral, afetando o todo. Por outro lado, quando a aplicação é dividida em microserviços, só existe a necessidade de escalarmos aqueles que necessitam da performance extra, o que reflete diretamente no custo de alavancar somente uma pequena parte da aplicação.

A ideia de dividir para melhor gerenciar recursos não é algo novo no universo Java. Este já apresenta diferentes soluções preparadas para essa nova maneira de se construir serviços, a exemplo do Spring Boot, uma das mais conhecidas. Tendo em seu core implementações dos containers Tomcat, Jetty e Undertow, oferece uma configuração rápida e simples, bastando poucas linhas para você ter um container web funcional com sua aplicação já embarcada e sem a necessidade de um deploy para tal.

Observando esse novo cenário, a Red Hat também vem buscando inovar. Na versão 9 do WildFly, por exemplo, houve muitas mudanças, como a implementação do Undertow no lugar do Tomcat, buscando assim tornar seu servidor web um container mais flexível, sem deixar de fornecer tudo o que um container full Java EE pode oferecer. Contudo, para abraçar de vez a realidade das aplicações voltadas para microserviços, foi necessário mais uma vez reinventar o container. A modularização apresentada na versão 7 do JBoss AS foi reestruturada e a maioria dos elementos que eram carregados por padrão na inicialização do container foram removidos. Agora, qualquer elemento que seja necessário para a aplicação precisa ser adicionado manualmente.

O fruto de toda essa reformulação gerou um novo produto, o WildFly Swarm, que promete não ser um novo container, como

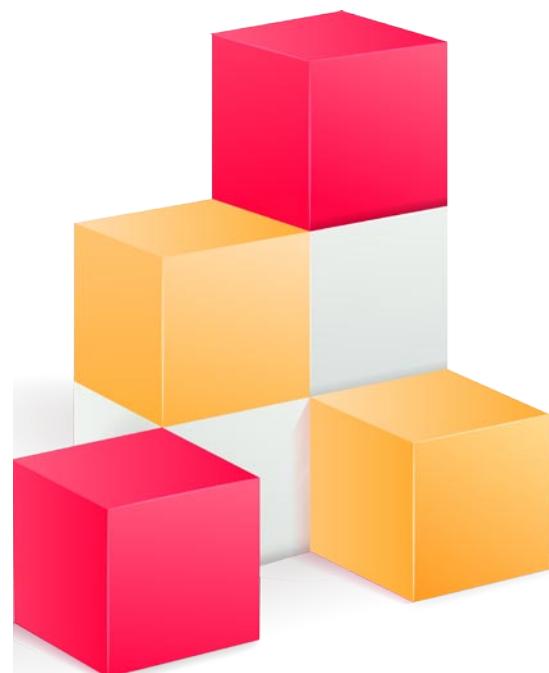
consta na própria página da ferramenta, e sim uma nova maneira de empacotar e rodar aplicações Java EE, fazendo isso de maneira flexível, por carregar somente o que a aplicação precisa; madura, por respeitar todos os padrões do escopo enterprise do Java; e direta, por carregar seu projeto e o container em um único JAR.

Por que Java e microserviços são uma boa ideia?

Caso você já tenha trabalhado com outras linguagens ou mesmo acompanha outras tecnologias, sabe que muitas vezes o Java é tido por outras comunidades como uma linguagem na qual não é fácil produzir algo de modo rápido. E a maneira como a linguagem trata seus novos adeptos também não é das mais sutis, exigindo muitas horas de estudo e prática até que você realmente consiga dar frutos de qualidade. Outro aspecto é que, por muitas vezes, um projeto relativamente pequeno pode chegar em poucos ciclos de desenvolvimento à casa de dezenas de dependências, demandando mais do container.

Note que ao falar dessa maneira até podemos ter a impressão de que falta inovação nesse universo. No entanto, observando os últimos anos podemos constatar que o Java tem buscado adotar vários aspectos de outras linguagens, como as expressões lambda, para atualizar seu repertório nativo e das bibliotecas que lhe dão suporte. Dessa maneira, ela consegue melhorar seu relacionamento com os desenvolvedores, pois o desejo da comunidade impulsiona a linguagem, justamente por ser um grupo que vocaliza muito bem seus anseios. Com isso, novas metodologias como microserviços logo ganham implementações – a exemplo do Spring Boot e do WildFly Swarm – que permitem transformar containers robustos e usá-los como pequenos serviços.

A construção de aplicações utilizando microserviços, faz você pensar de maneira diferente em como dividir a aplicação, e a preparar para crescer ao longo do tempo.



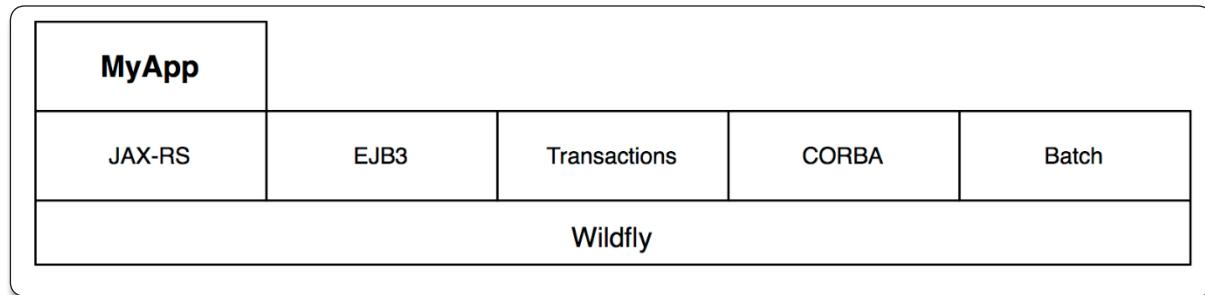


Figura 1. Aplicação carregada em um modelo padrão

Desse modo ganhamos mais possibilidades, podendo focar em pequenas partes sem a necessidade de escalar o todo. Além disso, a lógica de negócio pode ser desenvolvida por equipes e tecnologias diferentes, pois a aplicação deixa de ser um bloco monolítico para se tornar um conjunto formado de partes menores, que unidas formam o todo.

Diante disso, a aplicação se torna orgânica, mutável, pois a não obrigatoriedade de seguir uma linguagem ou framework torna mais flexível a migração para algo mais performático sem grandes impactos. Por termos pequenas aplicações agrupadas e se por qualquer que seja o motivo exista a necessidade de migrar toda a parte de persistência de um Hibernate para um EclipseLink, podemos realizar tal tarefa de modo gradativo, atualizando um serviço por vez, como também temos a possibilidade de manter as duas implementações, pois uma pode tratar determinado serviço de maneira mais eficiente que outra.

Saiba, ainda, que no gerenciamento de dependências também existem ganhos. Em um projeto monolítico, muitas vezes temos bibliotecas que atendem apenas a uma pequena parte da aplicação. Quando dividimos o projeto em vários serviços, podemos carregar menos bibliotecas em cada um, o que torna o gerenciamento das classes pela JVM uma tarefa mais amena, pois a máquina virtual

precisará manter uma quantidade menor de classes carregadas em cada uma das suas instâncias.

Assim, com o intuito de abranger essa nova proposta para a construção de aplicações, acrescentando tudo que já foi consolidado no WildFly, foi criado o Swarm, solução que se beneficia de características como modularização para possibilitar extrair mais performance do container e utilizar menos recursos de software e hardware.

Enfim, o projeto Swarm vem para reforçar esse princípio, mas com a ideia de ir além da modularização do container, se oferecendo como uma ferramenta que busca auxiliar o desenvolvedor a construir aplicações com partes menores que funcionam de maneira independente, mas mantendo a comunicação entre si.

O antes e o depois

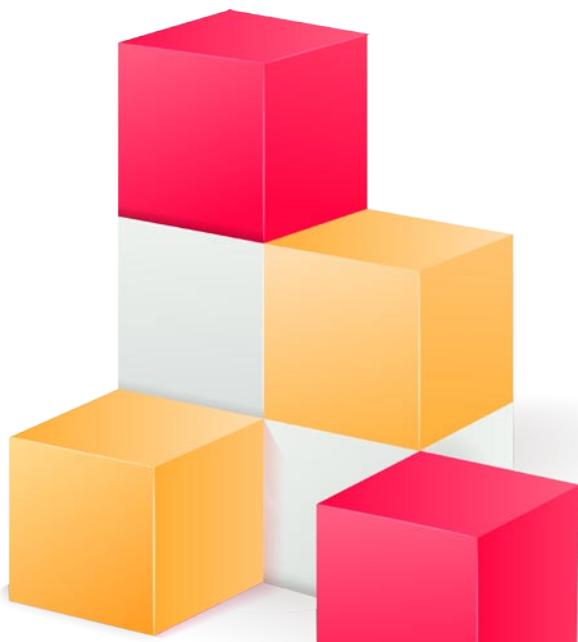
Ao utilizar um servidor de aplicações como o WildFly para desenvolver soluções que adotam uma arquitetura voltada para microsserviços, realizamos o setup inicial do container e, então, iniciamos o desenvolvimento da solução em si. O problema de adotar servidores de aplicações padrão para arquiteturas desse tipo é que provavelmente você acabará carregando mais funcionalidades do que sua solução necessita, o que pode levar ao consumo excessivo de recursos. A Figura 1 demonstra essa afirmação, mostrando que o serviço MyApp precisa somente da API JAX-RS, mas o WildFly carrega várias outras APIs.

Para lidar com esse problema o Swarm faz uso de um recurso bem conhecido no Java, o Uber JAR (ou Fat JAR). Diferente do modelo convencional, esse tipo de JAR contém não somente o projeto final compilado, mas pode também conter todas as dependências embutidas (a maneira como ele é empacotado lembra muito a dos arquivos WAR). A diferença do arquivo gerado é que ele contém também uma instância do Swarm. Assim, o que temos ao final é um arquivo auto executável que contém a aplicação e o próprio container com os módulos selecionados pelo desenvolvedor. A Figura 2 mostra a estrutura do serviço MyApp nesse modelo.

O resultado dessa mudança é uma menor utilização de recursos, gerando um único artefato de tamanho reduzido.

Um pouco de história

Para compreender o Swarm, não podemos ignorar a história que veio antes dele. E caso nunca tenha utilizado o JBoss AS, a



contextualização é válida para que você não se perca naquilo que pode tornar uma verdadeira sopa de letrinhas.

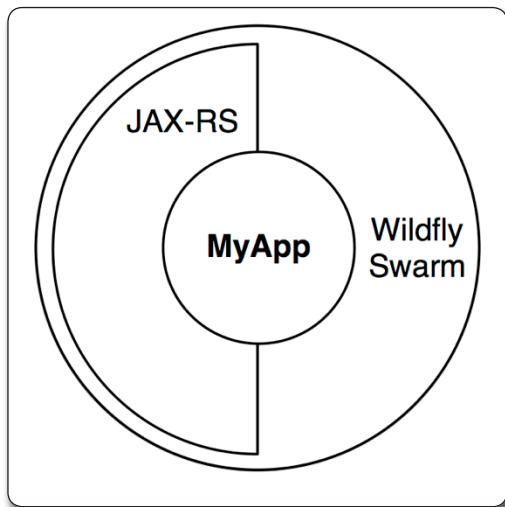


Figura 2. Modelo de um serviço criado com o Swarm

Em meados de 1999, nasceu a primeira versão do JBoss, chamado de EJBoss (*Enterprise Java Beans in Open Source System*). Um ano após esse lançamento, saiu a versão 1.0 e o nome do projeto passou a se chamar JBoss AS, ou *JBoss Application Server*. Nessa época o projeto era liderado por Marc Fleury, um dos responsáveis por alavancar a ferramenta. Com passar do tempo, o container ganhou força e seguiu evoluindo com o Java ao longo dos anos, impulsionado por uma forte comunidade; a mesma comunidade que viu o JBoss ganhar novas funcionalidades e muitas vezes, por consequência, se tornar mais pesado. Então, já em 2011, o container foi totalmente remodelado. Nascia, assim, a versão 7, que trouxe uma estrutura modularizada com a adoção do OSGI.

A partir de então, passamos a obter um maior controle das dependências, além do ciclo de vida dos módulos. A configuração do servidor também se tornou mais simples e de muitos arquivos antes necessários, passamos a utilizar apenas dois: o *standalone.xml*, para quando o ambiente não for clusterizado; e o *domain.xml*, para quando o ambiente for clusterizado. Desse momento em diante todas as configurações do container, desde o pool de conexões, threads, entre outras, passaram a ser referidas como subsistemas, o que fez com que os desenvolvedores tivessem que repreender a lidar com os recursos conhecidos de uma maneira diferente.

Nota

O OSGI pode ser visto como um conjunto de especificações responsável por definir um ambiente de computação padronizado e orientado a componentes.

Outra novidade muito bem-vinda foi a implementação do CLI, que se tornou uma poderosa ferramenta para configurarmos o ambiente, além de facilitar a automatização de várias operações,

como parada e inicialização do container, deploy de aplicações, configurações de portas e pools, etc.

No entanto, em 2013 o projeto foi renomeado para WildFly, e um dos motivos para tal mudança foi o fato de termos duas versões distintas do JBoss: a versão AS, para a comunidade; e o JBoss EAP, para as aplicações enterprise, oferecendo, também, como grande diferencial, suporte direto da Red Hat e seus parceiros.

Por fim, em 2015 foi criado o Swarm, que não vem como um substituto para o JBoss AS ou para o WildFly, mas sim como uma opção encontrada pela Red Hat para desconstruir e reconstruir o container de um modo que atenda as aplicações focadas em microserviços; algo que só foi possível graças à reformulação do JBoss AS em 2011. Agora, com o Swarm, só precisamos ser específicos em quais são as nossas necessidades e tudo que não for explicitamente declarado não estará presente na hora que o projeto for iniciado.

Nota

Caso queira saber mais sobre a história do JBoss AS, em edições anteriores da Java Magazine existem vários artigos que explicam detalhadamente as diferenças entre todas as versões.

Qual o impacto ao migrar uma aplicação para o Swarm?

Possibilitar que o usuário migre seu projeto que funciona no JBoss AS ou no WildFly sem grandes esforços é uma das grandes preocupações demonstradas pela equipe do projeto. Contudo, levar seu projeto para o Swarm significa dividi-lo em partes mentores e para isso existe a possibilidade de alguns elementos precisarem ser alterados. Se isso ocorrer, provavelmente será porque sua aplicação deixará de ser uma solução monolítica para ser dividida em pequenos serviços independentes.

Neste momento, a camada de persistência ainda é um ponto que gera muita discussão, por não haver um consenso de que cada microserviço deve ter ou não sua própria base de dados. Note que tal decisão pode impactar diretamente na quantidade de esforço necessário na hora de migrar sua aplicação, pois dividir algo que foi pensado para trabalhar como um bloco único, se mal planejado, pode gerar códigos repetidos em serviços diferentes.

Outro possível problema é que ao criar microserviços teremos mais pools de conexão ativos com o banco de dados e isso pode vir a ser um gargalo. Assim, é importante analisar como cada parte da aplicação lida com a informação. Ao realizar essa tarefa conseguimos identificar que nem tudo que está ligado diretamente ao banco precisa realmente estar, o que nos permite considerar a adoção de ferramentas que vão auxiliar a dispersar as solicitações realizadas diretamente às bases de dados. Neste momento uma boa opção é avaliar o que Martin Fowler chamou de persistência poliglota, o qual possibilita que cada serviço seja atendido pelo tipo de banco de dados que mais acolhe suas necessidades. dessa forma, um serviço de busca, por exemplo, pode se beneficiar de soluções como o ElasticSearch, dados que não possuem uma estrutura relacional podem estar em um MongoDB, assim como outros que possuem pouca ou nenhuma taxa de alteração podem

ser acessados via Redis. No entanto, tenha em mente também que ao agregar tecnologias a mais, além do ganho em flexibilidade e, provavelmente, desempenho, teremos um custo e complexidade maiores, afinal, todo o planejamento relacionado à disponibilidade e escalabilidade deverá ser aplicado às novas tecnologias, assim como teremos que contar com uma equipe mais qualificada e/ou maior.

Nosso primeiro projeto com Swarm

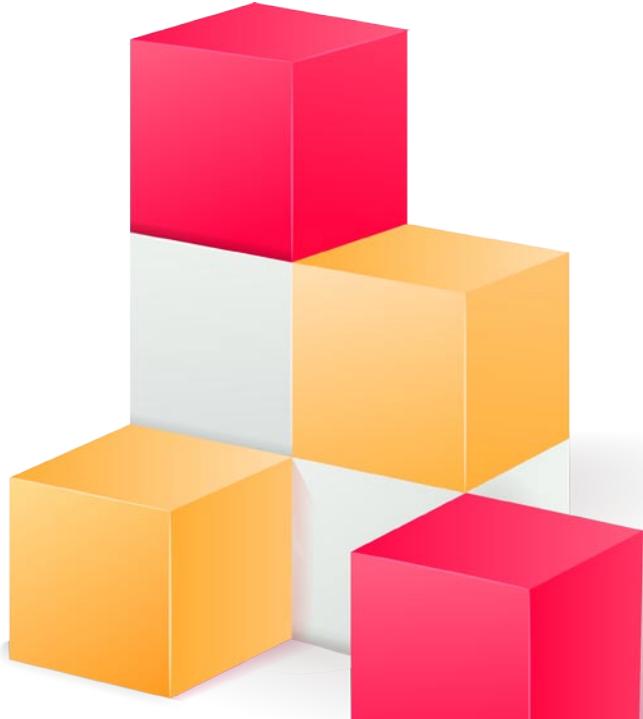
Para a nossa primeira experiência com o Swarm, vamos criar um pequeno projeto com um servlet simples que responderá a uma requisição e mostrará o famoso “Hello World”. Deste modo, crie um novo projeto utilizando o archetype Quickstart do Maven, ou seja, abra o prompt de comando no Windows ou o terminal no Linux/Mac e digite o comando:

```
mvn archetype:generate -DgroupId=br.com.devmedia -DartifactId=SwarmStart  
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Como podemos notar, o projeto criado é extremamente simples e traz somente uma única dependência, o JUnit.

Neste momento, abra a pasta do projeto e, em sua raiz, acesse o arquivo *pom.xml* para adicionar as propriedades conforme a **Listagem 1**. Vale destacar que o Swarm também oferece integração com o Gradle, mas até a versão na qual essa matéria se baseou ainda não existia documentação sobre isso.

Além das duas propriedades iniciais, que indicam que nosso código está em UTF-8 e que deve ser compilado da mesma forma, na linha 4 definimos a versão mínima do Maven. Já nas linhas 5 e 6 definimos a versão 8 do Java para o código fonte e também para os arquivos compilados.



Certifique-se que o número das versões esteja igual ou superior ao da **Listagem 1**, pois são as versões mínimas requeridas para se trabalhar com o Swarm. Portanto, se sua aplicação possui alguma dependência ou recurso restrito do Java 7, é preciso primeiro atualizar a aplicação e suas dependências. Por último, na linha 7 definimos a versão do Swarm que será adotada na declaração das dependências. Assim, garantimos que todos os recursos dessa solução que utilizaremos no projeto sejam compatíveis.

Conforme já mencionado, sabemos que o Swarm é modularizado. Dessa forma, é através das dependências que informamos no POM que controlamos os módulos que serão utilizados em nosso projeto.

Listagem 1. Variáveis de configuração para o projeto.

```
01. <properties>  
02.   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
03.   <project.reporting.outputEncoding>UTF-8</project.reporting  
        .outputEncoding>  
04.   <maven.min.version>3.2.1</maven.min.version>  
05.   <maven.compiler.target>1.8</maven.compiler.target>  
06.   <maven.compiler.source>1.8</maven.compiler.source>  
07.   <version.wildfly-swarm>1.0.0.Alpha6</version.wildfly-swarm>  
08. </properties>
```

A **Listagem 2** traz a declaração das duas dependências do Swarm que faremos uso em nosso primeiro exemplo.

A primeira é o Undertow, novo servidor web do WildFly que vem para substituir o antigo JBoss Web Server, um fork do Tomcat. Essa dependência agrupa muitas funcionalidades ao Swarm, dentre elas a possibilidade de utilizarmos JavaScript em nosso back-end. Já conhecido no WildFly por ser flexível e performático, o Undertow trabalha com APIs bloqueantes e não bloqueantes baseadas em NIO e oferece suporte a Servlets 3.1.

Listagem 2. Declaração do Undertow e do Weld.

```
01. <dependency>  
02.   <groupId>org.wildfly.swarm</groupId>  
03.   <artifactId>undertow</artifactId>  
04.   <version>${version.wildfly-swarm}</version>  
05. </dependency>  
06. <dependency>  
07.   <groupId>org.wildfly.swarm</groupId>  
08.   <artifactId>weld</artifactId>  
09.   <version>${version.wildfly-swarm}</version>  
10. </dependency>
```

A segunda dependência é o Weld, implementação do CDI da Red Hat que possibilita a criação de contextos e injeção de dependências para a plataforma Java EE. Por ser um padrão JCP, está presente em vários servidores além do próprio Swarm, como o WebSphere, GlassFish, Oracle WebLogic, dentre outros.

Ademais, pode ser encontrado em implementações para ambientes servlet-only como o Tomcat e o Jetty.

Agora necessitamos declarar os plug-ins que serão utilizados na fase de build do nosso projeto. A **Listagem 3** mostra cada um deles.

Listagem 3. Plugins utilizados no projeto para configuração e geração do WAR, controle do Swarm.

```
01. <build>
02.   <finalName>${project.artifactId}</finalName>
03.   <plugins>
04.     <plugin>
05.       <groupId>org.apache.maven.plugins</groupId>
06.       <artifactId>maven-war-plugin</artifactId>
07.       <version>2.6</version>
08.       <configuration>
09.         <failOnMissingWebXml>false</failOnMissingWebXml>
10.       </configuration>
11.     </plugin>
12.     <plugin>
13.       <groupId>org.wildfly.swarm</groupId>
14.       <artifactId>wildfly-swarm-plugin</artifactId>
15.       <version>${version.wildfly-swarm}</version>
16.       <executions>
17.         <execution>
18.           <goals>
19.             <goal>package</goal>
20.           </goals>
21.         </execution>
22.       </executions>
23.       <configuration>
24.         <bundleDependencies>true</bundleDependencies>
25.       </configuration>
26.     </plugin>
27.     <plugin>
28.       <groupId>org.apache.maven.plugins</groupId>
29.       <artifactId>maven-compiler-plugin</artifactId>
30.       <version>3.3</version>
31.       <configuration>
32.         <source>${maven.compiler.source}</source>
33.         <target>${maven.compiler.target}</target>
34.       </configuration>
35.     </plugin>
36.   </plugins>
37. </build>
```

Como podemos notar, foram declarados três plug-ins. Caso já tenha algum contato com o Maven, dois deles provavelmente você já deve ter utilizado. O primeiro é o **maven-war-plugin**. Utilizamos essa opção para gerarmos o WAR e também informar ao Maven que nosso projeto não possuirá um *web.xml*. Assim, por mais que nosso projeto seja do tipo web, quando o compilarmos através do Maven ele será executado com sucesso. O segundo plugin mais conhecido é o **maven-compiler-plugin**, e no momento que o projeto for compilado ele verificará se o Maven

instalado na máquina tem a versão 3.3, conforme a linha 30. Já nas linhas 32 e 33 garantimos que a versão do Java seja a oito, para atender a necessidade do Swarm, que necessita dessa versão ou superior.

A novidade na **Listagem 3** fica por conta do plugin do Swarm. Através dele podemos configurar vários aspectos de como queremos que nossa aplicação seja tratada. Nesse caso, informamos que queremos que as dependências da aplicação estejam dentro do JAR que será gerado, como declarado na linha 24.

Com isso, terminamos a configuração do *pom.xml* e podemos partir para a implementação da única classe que vamos utilizar no exemplo. Basicamente, nossa classe estenderá **HttpServlet** para sobrecrever o método **doGet()** e devolver para a tela o famoso “Hello World” no estilo Swarm. Na **Listagem 4** podemos verificar o código da classe **HelloSwarm**.

Listagem 4. Implementação da classe HelloSwarm.

```
01. package br.com.devmedia;
02.
03. import java.io.IOException;
04. import javax.servlet.ServletException;
05. import javax.servlet.annotation.WebServlet;
06. import javax.servlet.http.HttpServlet;
07. import javax.servlet.http.HttpServletRequest;
08. import javax.servlet.http.HttpServletResponse;
09.
10. @WebServlet(name = "HelloSwarmServlet", urlPatterns = "/HelloSwarm")
11. public class HelloSwarm extends HttpServlet{
12.
13.     private static final long serialVersionUID = 1L;
14.
15.     @Override
16.     protected void doGet(HttpServletRequest req, HttpServletResponse rep)
17.             throws ServletException, IOException {
18.         rep.getWriter().write("Hello Swarm");
19.     }
20. }
```

Com a classe pronta é possível notar que não existe nenhum tipo de **import** relacionado ao Swarm, ou seja, toda a implementação do código é baseada em um servlet simples. Além disso, como não temos o arquivo *web.xml*, utilizamos a anotação **@WebServlet** para indicar o nome do servlet, através da propriedade **name**, e o caminho que o mesmo deve mapear, através da propriedade **urlPatterns**, conforme é mostrado na linha 10.

Na linha 16 sobrecrevemos o método **doGet()** para que, quando o path especificado for acessado, possamos devolver nossa mensagem para a tela. Já na linha 17, através do objeto **rep**, que é uma instância de **HttpServletResponse**, conseguimos pegar o objeto **PrintWriter** e, através dele, enviar a mensagem de volta para a tela.

Feito isso, para ver o projeto funcionado, vamos utilizar o plugin do Maven para compila-lo, gerar o WAR e colocá-lo em execução.

Assim, acesse o prompt de comando no Windows ou o terminal no Mac ou Linux, navegue até a pasta raiz do projeto e digite o comando:

```
mvn wildfly-swarm:run
```

Em sua primeira execução o Swarm pode levar um bom tempo para finalizar. Isso porque o Maven precisará realizar o download de várias dependências. Portanto, até tudo estar pronto, pode demorar um certo tempo. Ao final do processo você verá no console mensagens informando que o Undertow está no ar e que sua aplicação já está disponível. Então, basta acessá-la através do navegador ao informar a URL <http://localhost:8080>HelloSwarm>.

Com a página carregada, a mensagem que definimos em nossa classe é exibida. Nesse momento é possível afirmar que o fato de utilizarmos o Swarm não afetou em nada o código necessário para criar um servlet. Contudo, vejamos a pasta *target* para identificar o artefato gerado. Dentre vários arquivos e pastas presentes nesse local, existe um JAR com o nome do projeto mais o sufixo “-swarm.jar”. Trata-se do arquivo do projeto que acabamos de criar pronto para ser executado, ou seja, já compilado com o Swarm e as dependências que definimos.

Como importante curiosidade, o artefato gerado tem tamanho bem reduzido: em torno de 436KB. Além disso, lembre-se que não é preciso fazer o deploy desse arquivo no WildFly, por exemplo, que em sua recém-lançada versão 10, ocupa 132 MB com a instalação padrão. Ainda assim, Ken Finnigan, um dos principais envolvidos

no Swarm, já informou, durante o Java One 2015, que a equipe trabalha para reduzir ainda mais o arquivo gerado.

Obviamente, o tamanho do arquivo também é afetado pelo número de dependências, ou mesmo pelo modo como configuramos o Swarm. Repare na linha 2 da **Listagem 5**. Ela mostra a parte da configuração de parâmetros do Swarm apresentada na **Listagem 3** referente ao plugin do Swarm, mas aqui alteramos o valor de **bundleDependencies** para **false**. Assim, as dependências não serão incluídas no arquivo gerado e serão buscadas diretamente no repositório Maven durante a execução do projeto. Por causa desse ajuste que conseguimos um tamanho tão reduzido. Além disso, adicionamos as linhas 3, 4 e 5 para mudarmos alguns parâmetros do projeto.

Na linha 3, temos o parâmetro **httpPort**, que especifica em qual porta o Swarm irá rodar. No caso, trocamos o valor para 9080. dessa forma, ao executar novamente o projeto, para acessar a aplicação devemos utilizar o endereço <http://localhost:9080>HelloSwarm>.

Ainda existem itens relacionados a configurações do Swarm que serão disponibilizadas ao longo das releases, lançadas quase que semanalmente. No entanto, já temos opções como, por exemplo, definir uma porta específica para debug (veja as linhas 4 e 5 da **Listagem 5**), o que nos permite utilizar ferramentas como as do Eclipse para conectar nossa aplicação e examiná-la.

Listagem 5. Alteração da porta padrão do Swarm, ativação do modo debug e definição de porta para isso.

```
01. <configuration>
02.   <bundleDependencies>false</bundleDependencies>
03.   <httpPort>9080</httpPort>
04.   <debug>true</debug>
05.   <debugPort>9081</debugPort>
06. </configuration>
```

Trabalhando com o mínimo

Ao final do primeiro exemplo podemos notar que a quantidade de configurações para o projeto poder executar é baixa, e desejamos que essa característica persista, pois um dos nossos objetivos é que os serviços sejam pequenos, seja pelo seu tamanho ou pela quantidade de configurações necessárias. Note que se um serviço possui um setup com muitos parâmetros, existe a possibilidade de ele acumular muitas atribuições e, por isso, talvez seja uma boa ideia quebrá-lo em serviços menores. Lembre-se que cada um deve ser planejado para ser ágil e evitar que um mesmo trabalho seja realizado em locais diferentes.



No planejamento, é importante definir também quais partes do Swarm vamos utilizar, inclusive os subsistemas herdados do WildFly. Como vimos, cada uma delas deve ser informada no POM como uma dependência simples. A documentação oficial as coloca como uma coleção bem definida de recursos a serem adicionados em sua aplicação, e cada uma dessas pequenas partes recebe o nome de Fraction.

Assim, quando adicionamos dependências como o Weld e o Undertow, estamos adicionando pequenas partes ao nosso Swarm, que passa a carregar subsistemas que, por padrão, não estariam presentes.

Nota

O Swarm ainda não possui todos os módulos do WildFly no formato de fractions, contudo, Ken Finnigan afirmou que a comunidade tem sido muito receptiva e vem contribuindo consideravelmente para o amadurecimento da ferramenta, o que deve acelerar o ritmo de lançamento das próximas releases e implementação de novas fractions.

A inclusão de fractions em um projeto deve ser feita através do POM, do mesmo modo que você incluiria uma dependência Maven. Como exemplo, a **Listagem 6** mostra o que precisamos adicionar ao POM para poder utilizar a fraction de log.

Listagem 6. Declaração da dependência do fraction de logging.

```
<dependency>
<groupId>org.wildfly.swarm</groupId>
<artifactId>wildfly-swarm-logging</artifactId>
<version>${version.wildfly-swarm}</version>
</dependency>
```

Pronto! Temos uma API de log em nossa aplicação. Agora, para acionar o log, é preciso informar o nível que desejamos. Supondo que seja em modo debug, basta adicionar o seguinte parâmetro no comando `mvn wildfly-swarm:run`:

`-Dswarm.logging=DEBUG`

Dessa forma, o comando para executar nossa aplicação seria o seguinte:

```
mvn wildfly-swarm:run -Dswarm.logging=DEBUG
```

Isso é suficiente para iniciar o serviço em modo debug.

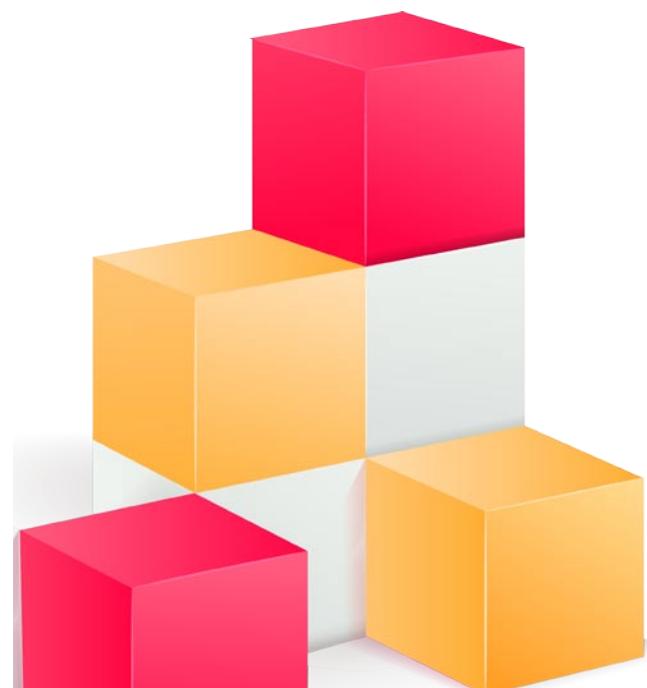
Apesar dessa opção ser eficiente para setar opções que podem mudar através de script a cada novo start, uma maneira de configuração do Swarm que vem sendo bem recebida pelos desenvolvedores são as customizações através das classes main de cada serviço. O único problema dessa abordagem, por enquanto, é que só pode ser utilizado quando você empacota seu projeto como um JAR, ou seja, fica indisponível caso seu projeto esteja definido para ser empacotado como um WAR.

Criando classes main customizadas

Em projetos Java SE é comum criarmos uma classe para ser a principal. Como diferencial, essa classe deve implementar o método `main()`, a ser chamado quando o JAR for executado. No Swarm, curiosamente, também podemos utilizar esse recurso. Para isso, no entanto, precisamos, primeiro, nos certificar que o tipo de empacotamento no POM esteja definido como JAR. Em seguida, devemos criar uma classe simples, `App`, que servirá como nossa classe principal. Por fim, basta implementar o método `main()`. A **Listagem 7** mostra o código dessa classe.

Listagem 7. Código da classe App.

```
01. package br.com.devmedia;
02.
03. import org.jboss.shrinkwrap.api.ShrinkWrap;
04. import org.wildfly.swarm.container.Container;
05. import org.wildfly.swarm.undertow.WARArchive;
06.
07. public class App
08. {
09.     public static void main( String[] args ){
10.         try {
11.             Container container = new Container();
12.             container.start();
13.             System.out.println( "Swarm no ar" );
14.             WARArchive deployment = ShrinkWrap.create(WARArchive.class);
15.             deployment.addClass(HelloSwarm.class);
16.             container.deploy(deployment);
17.         } catch (Exception e) {
18.             e.printStackTrace();
19.         }
20.     }
21. }
```



Tipo	Descrição
JARArchive	Uma versão melhorada do JAR que fornece métodos fáceis para adicionar dependências.
WARArchive	Uma versão melhorada do WAR que fornece métodos fáceis para adicionar conteúdo estático como páginas HTML, CSS e JavaScript, assim como arquivos de configuração, como o web.xml.
JAXRSArchive	Web archive que permite criar uma aplicação REST a partir de uma classe sem a necessidade de declararmos a anotação @Application-Path. Basta adicionar essa classe ao JAXRSArchive através do método addResource() e depois definir o nome do serviço com o método setApplicationName("nome-serviço"). Assim, a URL REST estará disponível em http://localhost:8080 /nome-serviço.
Secured	Tipo de arquivo que injeta o arquivo keycloak.json e configura restrições de segurança. O Keycloak é uma solução de SSO para aplicações web que disponibilizam serviços através de chamadas RESTful.
RibbonArchive	Arquivo que pode registrar serviços baseados em Ribbon-based, solução que age como um load balancer de chamadas desenvolvida pela Netflix.

Tabela 1. Tipos de arquivos oferecidos na versão atual do Swarm

Quando procedemos dessa forma, para iniciar o projeto devemos realizar todo o trabalho que anteriormente era efetuado automaticamente pelo Swarm. Assim, agora precisamos subir o container, preparar os arquivos que estarão no projeto e, por último, realizar o deploy.

Na linha 11 temos o código que cria um container, uma instância do Undertow, e como você pode imaginar, toda a inicialização do mesmo é feita na linha 12. Com isso, temos o Swarm já funcionando, mas ainda sem o nosso projeto.

Na linha 14 criamos um **WARArchive**, que age como um arquivo WAR carregado em memória. Aqui vale uma ressalva: note que ainda estamos utilizando o empacotamento do nosso projeto como um JAR. Usamos o recurso de alocação em memória como um WAR apenas para realizar o deploy de nosso serviço.

Nesse código, fizemos uso também da classe **ShrinkWrap**, que facilita criar, importar, exportar e manipular arquivos em memória. Na linha 14 utilizamos seu método estático **create()**, que aceita como parâmetro uma classe genérica para definir o tipo de arquivo a ser criado. Como queremos alocar um WAR em memória para realizar o deploy em nosso container, declaramos a classe **WARArchive**. A partir daí podemos adicionar os nossos recursos, como classes, páginas HTML e arquivos JavaScript, que, diferentemente do que ocorre em um projeto web padrão no Swarm, precisarão ser explicitamente inseridos no WAR virtual que estamos manipulando.

A **Tabela 1**, retirada da própria documentação do Swarm, mostra os tipos de arquivos virtuais que o mesmo disponibiliza. Saiba que esses tipos de arquivos estão diretamente relacionados ao tipo de aplicação que você vai querer realizar o deploy. Como estamos fazendo o deploy de um projeto web padrão, utilizamos o **WARArchive**.

Voltando à explicação do código da **Listagem 7**, o próximo passo é adicionar nossa classe ao deploy que iremos realizar, como demonstra a linha 15. Logo após, na linha 16, realizamos de fato o deploy.

Agora, precisamos inserir no POM o apontamento para a classe main customizada (**App**). Sendo assim, depois do elemento **bundleDependencies**, adicione a seguinte linha:

```
<mainClass>br.com.devmedia.App</mainClass>
```

Caso tenha criado um projeto com um pacote ou classe com nomes diferentes, não esqueça de atualizar o seu POM. Em seguida, inicie o Swarm novamente e observe que obtivemos o mesmo resultado de quando não utilizamos a classe customizada.

Vale ressaltar que o uso de uma classe principal customizada nem sempre é opcional. Em alguns fractions, como o do JSF, existe a necessidade de criar a classe main customizada porque precisaremos incluir arquivos HTML, CSS, JavaScript, dentre outros ao nosso deploy.

Adicionando outros fractions ao nosso projeto

Para entender como podemos acrescentar outros tipos de arquivo ao nosso projeto e como gerenciá-los através da classe main, vamos adicionar o fraction do JSF. De preferência, use o mesmo projeto dos exemplos anteriores, pois queremos, ao final, ter a funcionalidade que criamos anteriormente e a que vamos criar agora.

Dito isso, dentro da pasta *main* de seu projeto, crie uma pasta chamada *resources*. Em seguida, dentro de *resources*, crie a pasta *WEB-INF* e dois arquivos, com os nomes de *index.html* e *index.xhtml*. Nas **Listagens 8** e **9** temos o código desses arquivos.

Note, na **Listagem 8**, que foi declarado um simples redirect para o verdadeiro index, na linha 4, forçando o redirecionamento. Na **Listagem 9**, observe que usamos alguns componentes JSF. Na linha 7, por exemplo, declaramos um **ui:composition** para importar um arquivo de template que criaremos na sequência. Na linha 9, declaramos um elemento **outputText** para receber o retorno do método **welcomeFromJSF()**, de uma classe que ainda vamos criar.

Agora, dentro da pasta *WEB-INF*, crie os arquivos *web.xml* e *template.xhtml*. Como você já deve ter notado, montamos manualmente a estrutura de pastas de um projeto web simples.

A **Listagem 10** apresenta o nosso *web.xml*. Ao analisá-lo, veremos que nas linhas 5 e 6 definimos que o projeto se encontra em desenvolvimento. Esse parâmetro não tem impacto direto em nossa aplicação e serve mais como uma alerta para o JSF, que pode, através dele, saber como atuar da melhor maneira conforme o ambiente. Por exemplo, se o stage estiver como **Development**, ele vai proporcionar mais detalhes no caso de erros e alertas. Da linha 8 à linha 12, registramos o Faces Servlet e informamos que o mesmo deve ser carregado no mesmo momento que a

aplicação. Por último, entre as linhas 13 e 16, informamos que o Faces Servlet deve monitorar todas as URLs que terminem com a extensão .xhtml.

Listagem 8. Conteúdo do arquivo index.html.

```
01. <!DOCTYPE html>
02. <html>
03. <head>
04.   <meta http-equiv="Refresh" content="0; URL=index.xhtml">
05. </head>
06. </html>
```

Listagem 9. Conteúdo do arquivo index.xhtml.

```
01. <?xml version='1.0' encoding='UTF-8'?>
02. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
03. <html xmlns="http://www.w3.org/1999/xhtml"
04.   xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
05.   xmlns:h="http://xmlns.jcp.org/jsf/html">
06.   <body>
07.     <ui:composition template="/WEB-INF/template.xhtml">
08.       <ui:define name="content">
09.         <h:outputText value="#{welcome.welcomeFromJSF()}">
10.       </ui:define>
11.     </ui:composition>
12.   </body>
13. </html>
```

Listagem 10. Conteúdo do arquivo web.xml.

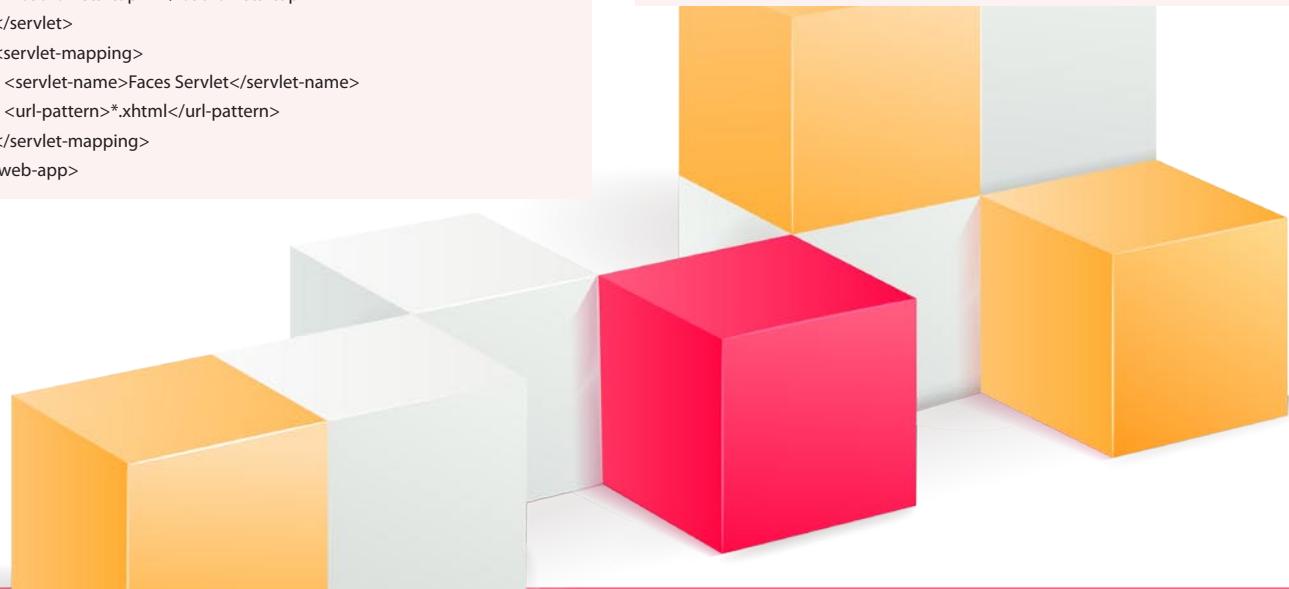
```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03.   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
   http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
04. <context-param>
05.   <param-name>javax.faces.PROJECT_STAGE</param-name>
06.   <param-value>Development</param-value>
07. </context-param>
08. <servlet>
09.   <servlet-name>Faces Servlet</servlet-name>
10.  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
11.  <load-on-startup>1</load-on-startup>
12. </servlet>
13. <servlet-mapping>
14.   <servlet-name>Faces Servlet</servlet-name>
15.   <url-pattern>*.xhtml</url-pattern>
16. </servlet-mapping>
17. </web-app>
```

Já a **Listagem 11** apresenta o código de *template.xhtml*. Como podemos notar, é um XHTML simples com algumas tags do JSF apenas para criar alguns elementos que serão utilizados na tela. Na linha 15, por exemplo, temos a tag **<ui:insert>**, que possibilita inserir código HTML gerado no back-end, e na linha 17, criamos outra **<div>**, onde inserimos uma imagem com o logo do Swarm.

Logo após, dentro do pacote **br.com.devmedia**, crie a classe **Welcome** (veja a **Listagem 12**), a qual será utilizada para gerar a mensagem que vamos inserir em nosso HTML através do método **welcomeFromJSF()**.

Listagem 11. Conteúdo do arquivo template.xhtml.

```
01. <?xml version='1.0' encoding='UTF-8'?>
02. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
03. <html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
   xmlns:h="http://xmlns.jcp.org/jsf/html">
04.   <h:head>
05.     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
06.     <title>Exemplo de Swarm + JSF</title>
07.   </h:head>
08.   <h:body>
09.     <div id="top">
10.       <ui:insert name="top">
11.         <h4>Seu primeiro Facelet via Swarm.</h4>
12.       </ui:insert>
13.     </div>
14.     <div id="content">
15.       <ui:insert name="content">Content</ui:insert>
16.     </div>
17.     <div id="bottom">
18.       <ui:insert name="bottom">Powered by
         </ui:insert>
19.     </div>
20.   </h:body>
21. </html>
```



Listagem 12. Código da classe Welcome.

```
01. package br.com.devmedia;
02. import javax.enterprise.inject.Model;
03.
04. @Model
05. public class Welcome {
06.     public String welcomeFromJSF() {
07.         return "Bem vindo ao JSF!! Ou você prefere hello world?";
08.     }
09. }
```

Note que essa classe servirá como um Model do JSF, o que é especificado através da anotação **@Model**, na linha 4. Com isso, os métodos dela ficarão expostos de uma maneira que conseguiremos acioná-los do nosso HTML através da tag **insert** do JSF.

Assim finalizamos a parte web de nosso projeto e podemos adicionar o fraction do JSF ao POM, o que é feito conforme a **Listagem 13**.

Para concluir, altere nossa classe **App** para adicionar os arquivos recém-criados ao arquivo WAR, conforme mostra a **Listagem 14**.

Na linha 16 adicionamos nossa nova classe ao deploy. Da linha 17 até a linha 20, utilizamos um método diferente para inserir os arquivos web, **addAsWebResource()**, que nos auxilia informando que o artefato incluído deve ser tratado como um elemento estático. Já nas linhas 22 e 24, adicionamos os arquivos *web.xml* e *template.xhtml* através do método **addAsWebInfResource()**. Assim, eles serão tratados como arquivos de pré-configuração do projeto. Na linha 25, todas as dependências são inseridas efetivamente no deploy; na linha 26 é a vez de **HelloSwarm**, fazendo com que a classe que implementamos anteriormente também esteja presente; e na linha 27, realizamos o deploy.

Neste momento, para visualizar as atualizações, inicie novamente o Swarm executando:

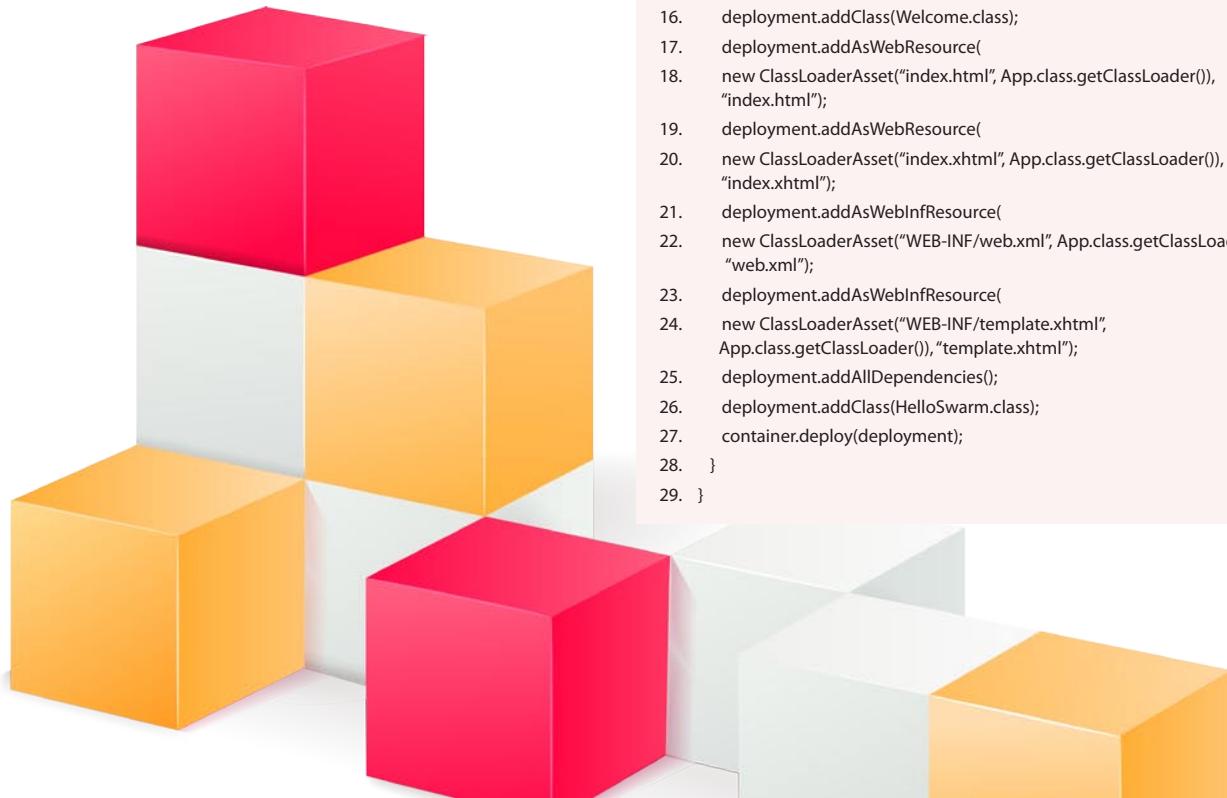
```
mvn wildfly-swarm:run
```

Listagem 13. Dependência do fraction do JSF a ser adicionado ao pom.xml.

```
01. <dependency>
02.   <groupId>org.wildfly.swarm</groupId>
03.   <artifactId>jsf</artifactId>
04.   <version>${version.wildfly-swarm}</version>
05. </dependency>
```

Listagem 14. Código da classe App atualizado.

```
01. package br.com.devmedia;
02.
03. import org.jboss.shrinkwrap.api.ShrinkWrap;
04. import org.jboss.shrinkwrap.api.asset.ClassLoaderAsset;
05. import org.wildfly.swarm.container.Container;
06. import org.wildfly.swarm.undertow.WARArchive;
07.
08. public class App
09. {
10.     public static void main( String[] args ) throws Exception
11.     {
12.         Container container = new Container();
13.         container.start();
14.         System.out.println("Swarm no ar");
15.         WARArchive deployment = ShrinkWrap.create(WARArchive.class);
16.         deployment.addClass(Welcome.class);
17.         deployment.addAsWebResource(
18.             new ClassLoaderAsset("index.html", App.class.getClassLoader()),
19.             "index.html");
20.         deployment.addAsWebResource(
21.             new ClassLoaderAsset("index.xhtml", App.class.getClassLoader()),
22.             "index.xhtml");
23.         deployment.addAsWebInfResource(
24.             new ClassLoaderAsset("WEB-INF/web.xml", App.class.getClassLoader()),
25.             "web.xml");
26.         deployment.addAsWebInfResource(
27.             new ClassLoaderAsset("WEB-INF/template.xhtml",
28.                 App.class.getClassLoader()), "template.xhtml");
29.         deployment.addAllDependencies();
30.         deployment.addClass(HelloSwarm.class);
31.         container.deploy(deployment);
32.     }
33. }
```



Ao acessar a URL <http://localhost:8080>HelloSwarm>, você notará que nossa página recém-criada será exibida. Isso porque nosso projeto web, por padrão, procura pelo arquivo *index.html* e como nosso index redireciona o fluxo para o arquivo *index.xhtml*, todo o conteúdo deste é mostrado, inclusive nosso cumprimento customizado pela classe **Welcome**. A Figura 3 mostra nossa pequena tela no browser.



Figura 3. Saída da tela montada com JSF, utilizando o Swarm

Com a classe principal customizada é notável que precisamos de um pouco mais de trabalho para rodar o projeto. Entretanto, o uso dessa classe possibilita um setup mais minucioso do projeto, viabilizando executar o deploy de classes ou arquivos específicos conforme a situação.

Como outro diferencial, apesar de recém-lançado, o Swarm já oferece suporte à ferramenta de monitoramento Hawkular – também da Red Hat – para você ter a capacidade de monitorar seus serviços e tomar ações de acordo com a necessidade.

Por fim, saiba que o Swarm vem conquistando adeptos rapidamente. Seu uso em produção, no entanto, ainda não é aconselhável, conforme relatado pelo próprio time da Red Hat. Na documentação ainda faltam itens como a integração com o Gradle, que já foi demonstrada no Java One 2015, mas ainda não consta no site oficial. Além disso, ferramentas de monitoramento mais populares, como New Relic e Ruxit, estão em fase de adequação para poder oferecer suporte ao Swarm, assim como ainda não está claro como irá funcionar a conexão com servidores HTTP como o Apache, para prover o balanceamento de carga entre instâncias de um mesmo serviço. De qualquer modo seu estudo é estimulado juntamente com a participação em fóruns, no Google

Groups e chats no IRC. Os responsáveis pelo projeto acompanham de perto a comunidade e realizam várias alterações ao longo de cada release lançada. Portanto, participe, report bugs e traga suas ideias e descobertas, auxiliando assim o projeto a crescer.

Autor



Joel Backschat

joel@cafecomjava.com.br

Bacharel em Sistemas da Informação pela Universidade da Região de Joinville, possui certificação SCJP e Adobe FLEX. Já trabalhou em empresas como TOTVS e Supero. Desde 2014 é arquiteto de software da Fcamara Formação e Consultoria nas áreas de inovação e logística portuária. Entusiasta de novas tecnologias, mantém o site cafecomjava.com.br para compartilhar suas experiências que vão do hardware ao software.



Links:

Página do Swarm.

<http://wildfly-swarm.io/>

Documentação oficial do WildFly Swarm.

<https://wildfly-swarm.gitbooks.io/>

Página do projeto Undertow.

<http://undertow.io/>

Palestra de Ken Finnigan sobre Swarm.

<https://developers.redhat.com/video/youtube/i1aiUaa8RZ8/>

Grupo no Google sobre o Swarm.

<https://groups.google.com/forum/#!forum/wildfly-swarm>

Canal no IRC sobre o Swarm.

<http://webchat.freenode.net/?channels=wildfly-swarm>

Matéria sobre persistência poliglota.

<http://martinfowler.com/bliki/PolyglotPersistence.html>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



DevOps: Como adequar seu processo de CI a essa nova cultura

Veja neste artigo como aplicar os conceitos de DevOps em um contexto de integração contínua

ESTE ARTIGO É DO TIPO MENTORING

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

Em artigo publicado na edição 149 da Java Magazine, aprendemos a construir um ambiente de integração contínua baseado em Maven e Jenkins, suportado por medições estáticas de qualidade a partir de uma plataforma chamada Sonar Qube. O objetivo principal de todo o artigo foi, a partir de um tutorial prático, introduzir algumas das ferramentas mais populares do mercado contemporâneo para o desenho de processos dinâmicos de desenvolvimento de software, com foco na prática DevOps.

O resultado desse trabalho, que servirá de ponto de partida para este artigo, foi resumido na **Figura 1**. Nela, vemos a máquina de desenvolvimento, configurada com uma IDE (neste caso, o Eclipse), Git e Maven. Uma vez que parte do código do software tenha sido escrito, ele passa por uma validação local, realizada a partir da execução de testes unitários. Então, esse mesmo código é submetido a um controle local e, a seguir, remoto, de versão.

Quando esse repositório remoto, hospedado em uma conta GitHub, recebe esse conteúdo, dispara uma requisição de execução do primeiro de uma cadeia de jobs em um servidor de integração contínua (Jenkins), hospedado em um *gear* OpenShift. A partir desse primeiro job, todos os demais são executados sequencialmente, e o resultado final é um build completo, associado a

Cenário

DevOps é um dos termos mais populares do momento no mercado da Tecnologia da Informação. Muito se fala a respeito, e a expectativa quanto aos benefícios de sua adoção é alta. Com base nisso, este artigo visa contribuir com a discussão e disseminação de alguns dos elementos básicos que o constituem, a partir de um misto de impressões, constatações de âmbito cultural e, também, tutoriais que, em uma abordagem essencialmente prática, introduzirão algumas das tecnologias e plataformas mais populares empregadas nesse contexto.

uma análise estática da qualidade do projeto. Essa análise é feita por uma instância do Sonar Qube, também hospedada em um *gear* OpenShift.

No artigo de hoje, retomaremos muitos dos pontos que acabamos de citar, estendendo-os e refinando-os para oferecer, ao final, um processo alinhado aos principais fundamentos e práticas do que hoje se rotula como DevOps no mercado de TI. Obviamente, por ser esse um tema ainda em franca evolução, a visão levantada ao longo deste material será, naturalmente, fonte para uma série de discussões complementares. Saiba que esse não é um material definitivo sobre DevOps, mas uma introdução que visa trazer ao leitor informações que o ajudem a adotá-lo em seu cotidiano.

Entendendo o nosso ponto de partida

O projeto-guia de todo este artigo, bem como daquele publicado na edição 149, consiste em uma aplicação Java, *stand-alone*, para o cálculo de resistências elétricas a partir de um padrão de faixas e cores. Esse projeto apresenta, além do código principal, testes unitários que validam todos os requisitos levantados. Todo esse

conteúdo está devidamente salvo em um repositório público no GitHub, podendo ser visto e baixado pelo leitor a partir da referência que se encontra na seção **Links**.

A primeira grande decisão que tivemos de tomar, antes mesmo de iniciar a escrita do software em si, relacionou-se à plataforma de gerenciamento do ciclo de vida desse projeto. Esse é um ponto bastante importante, e qualquer escolha que se tome nesse instante refletirá, posteriormente, em todo o andamento do trabalho.

Por ciclo de vida de projeto, devemos entender atividades que vão desde o gerenciamento de dependências do produto até operações básicas como compilação, teste, empacotamento, implantação e, inclusive, execução.

Nesse contexto, o Maven ainda é uma ferramenta muito popular. É fato que, há alguns anos, vem disputando espaço com outras plataformas bem interessantes, como o Gradle, mas seu índice de utilização ainda é muito expressivo, sobretudo, no mundo corporativo. Ao final do artigo, na seção **Links**, recomendamos uma leitura complementar sobre esse tema.

Foi essa grande popularidade, aliada à maior familiaridade de toda a comunidade com o seu modelo de configuração, que nos levou a adotar o Maven em nosso projeto.

Outro aspecto que já encontraremos pré-configurado antes do início deste texto é um processo de build em um servidor Jenkins, rodando sobre a plataforma OpenShift. O cenário de partida é algo como o ilustrado na **Figura 2**. Perceba que, atualmente, temos apenas um processo de build, composto por quatro jobs. O primeiro passo dessa cadeia envolve uma limpeza de toda a árvore de diretórios do projeto, eliminando dali quaisquer resquícios de material gerado em builds anteriores. Em seguida, por meio do job intitulado *devmediadevops_test_job*, executa-se a fase de testes do ciclo de vida padrão do Maven. Nesse momento, são colocados para rodar todos os testes unitários do projeto, que validarão toda a lógica principal do produto. Ao ordenarmos que

a fase *test* seja processada, o Maven executará, também, todas as fases configuradas como anteriores a essa, e o resultado dessa operação será a validação, a compilação e, por fim, o processamento de todos os testes unitários do projeto.

Caso todos os testes executem com sucesso, o Jenkins passará a executar o job seguinte, de nome *devmediadevops_package_measure_job*. O objetivo dessa tarefa é realizar o empacotamento de todos os módulos do projeto, gerar o pacote da aplicação em si e, em seguida, medir a qualidade de todo o material por meio de uma análise do Sonar Qube.

O empacotamento é realizado, novamente, a partir do Maven, por meio do comando *mvn package -DskipTests=true -f \$OPENSHIFT_DATA_DIR/workspace/sonar/devops/pom.xml*. Note que, nesse momento, os testes são ignorados, uma vez que já os executamos no job anterior. Portanto, na tarefa de nome *devmediadevops_package_measure_job*, estamos apenas compilando o projeto novamente, empacotando-o em seguida. A ação seguinte, configurada como um passo de *pós-build* nesse job, consiste na comunicação remota com o servidor em que o Sonar Qube está instalado (via SSH) para, então, disparar a execução da análise do projeto. O resultado é publicado em uma página gerada e alimentada pelo Sonar, e que tem a aparência e estrutura exibidas na **Figura 3**.

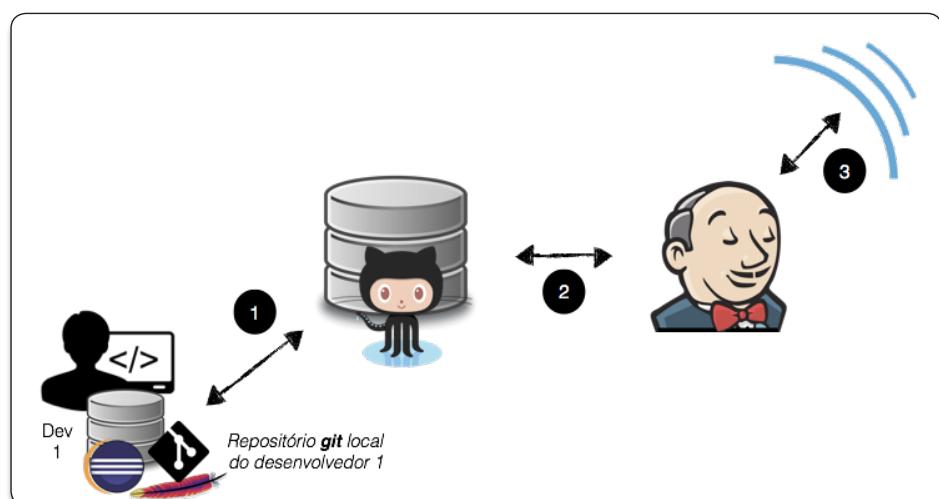


Figura 1. Estrutura de Integração Contínua e Análise Estática do projeto-guia

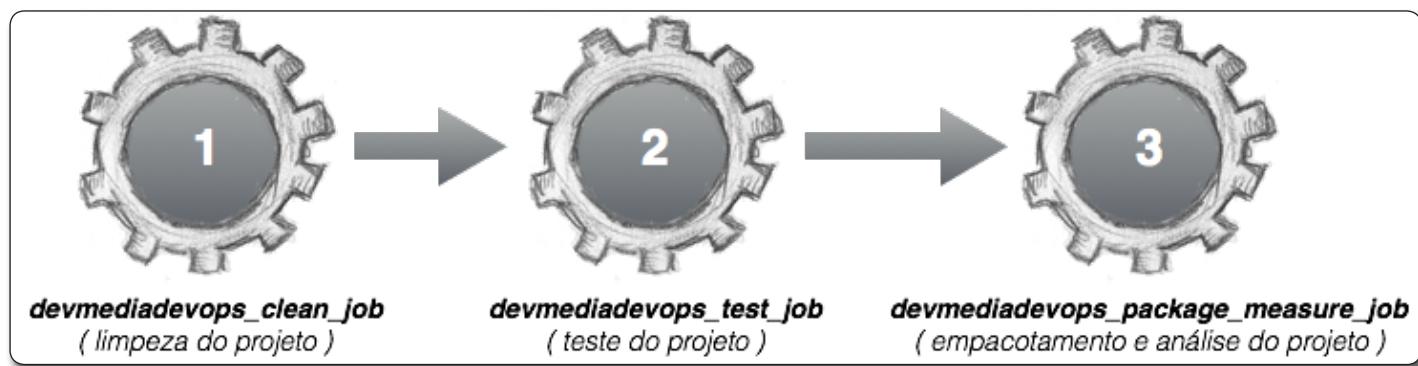


Figura 2. Configuração dos jobs no processo de integração contínua

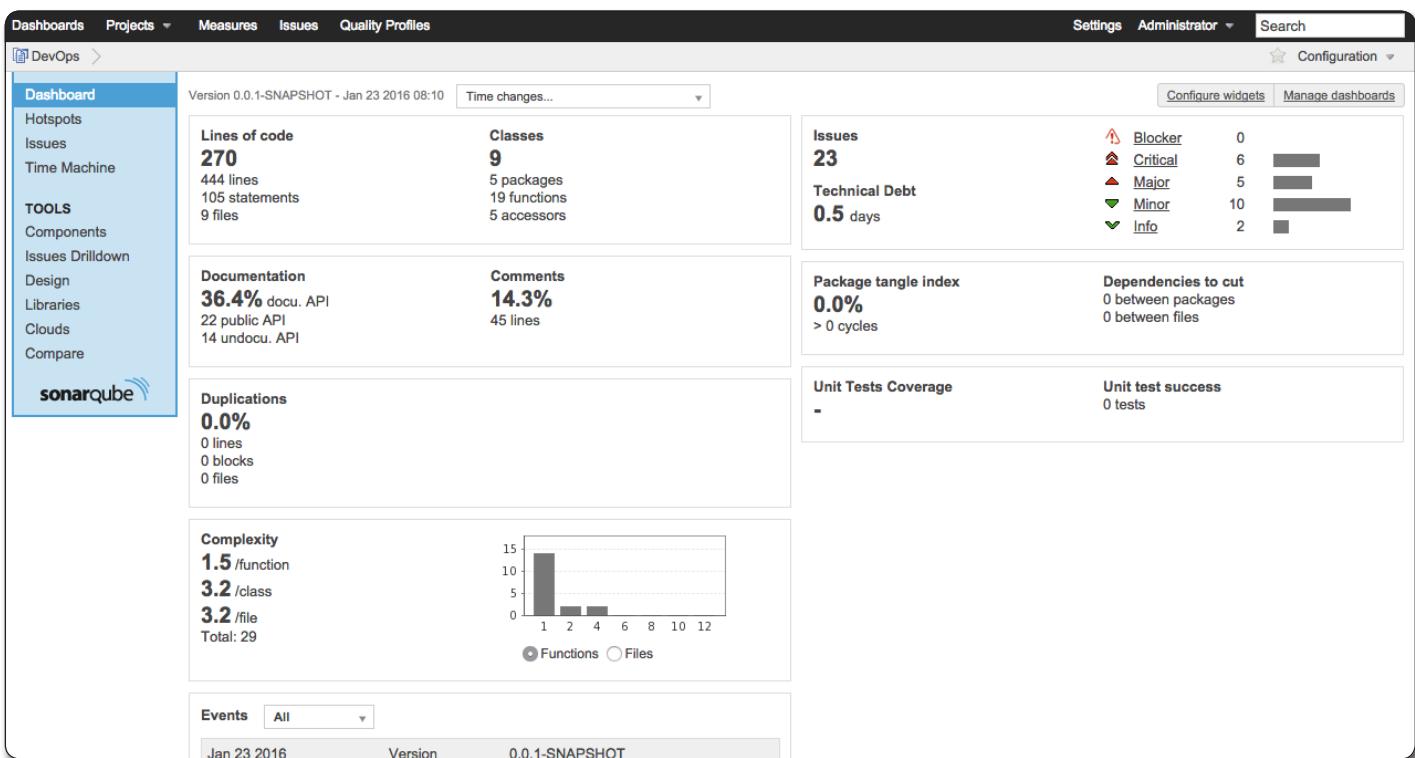


Figura 3. Dashboard do Sonar Qube após a execução de uma análise

Refinando o processo de integração contínua

Como acabamos de ver na seção anterior, o projeto original conta com apenas um processo de build. Um detalhe que não foi citado até aqui, mas que precisamos introduzir neste momento, é que essa cadeia de jobs é disparada automaticamente, sempre que um novo *commit* é realizado no repositório remoto.

Este processo automático é viabilizado a partir da utilização de um recurso chamado web hook, criado e administrado por meio da interface web do GitHub. Essa é a forma oferecida para contornar uma limitação imposta pela empresa, de não permitir conexões remotas via SSH com seus servidores. Os detalhes sobre o funcionamento desse mecanismo podem ser encontrados em uma referência informada na seção [Links](#).

Temos, portanto, um processo relativamente pesado em nossas mãos. Imagine que, a cada submissão de código nesse repositório, todo esse conjunto de atividades – algumas bem demoradas, como a análise estática feita pelo Sonar – é executado. Isso pode tornar o cotidiano de um projeto bastante lento, e prejudicar muito mais do que ajudar o andamento do trabalho. E por que?

O fato a ser observado é que a nossa realidade vem sendo fortemente transformada pelo advento das metodologias ágeis, há uns bons anos. Uma das lições que essa nova filosofia de trabalho nos ensina, dentre tantas, é que integração frequente de código é uma das chaves para o sucesso. Ao avaliarmos, continuamente, o impacto da junção do conteúdo desenvolvido por todo um time, tornamo-nos muito mais

capazes de enxergar a verdadeira realidade de um projeto. Integrar continuamente nos permite antecipar problemas, aumentando em muito o nosso poder de reação e controle frente a qualquer desvio identificado.

Em decorrência disso, uma das práticas hoje bem populares no cotidiano de times de desenvolvimento de software é a submissão, ao menos uma vez por dia, de todo o código-fonte produzido por cada desenvolvedor. Ainda nesta linha, da busca de transparéncia e controle pleno, há uma preferência notável pelo uso de branch único (normalmente denominado *trunk* ou *master*, em sistemas como Subversion e Git), evitando-se ao máximo o uso de branches paralelos (muito comum em tempos não tão remotos assim).

Branches podem ser muito úteis em desenvolvimento paralelo, de refatoração significativa, ou ainda em provas de conceito. Entretanto, em times ágeis, não há muito sentido que desenvolvedores se isolem em ramos paralelos para, somente lá na frente, integrar todo o material que produziram. O ideal é que possamos observar a saúde do software continuamente. Nesta seção, apresentamos uma sugestão que pode ajudar o leitor a entender como todo esse ferramental pode ajudá-lo a estabelecer um controle de qualidade sobre o seu projeto.

Voltando um pouco ao problema da lentidão e ineficiência introduzido no início dessa seção, imagine o quanto complicado seria rodar uma análise do Sonar Qube a cada submissão de código. Esse processo é normalmente lento e, com certeza, sobrecarregaria desnecessariamente o processo de build.

Tipo de projeto	Freestyle project
Git URL	https://github.com/pedrobrigatto/devmedia_devops_series.git
Branches to build	*/master
Triggers	Trigger builds remotely Authentication Token > devmedia_devops_token
Build	Invoke top-level Maven Targets: clean verify -f \$OPENSHIFT_DATA_DIR/workspace/sonar/devops/pom.xml

Tabela 1. Configuração do novo job para execução por commit

A mesma linha de raciocínio é válida para a implantação de versões em servidores de QA ou produção. Isso, embora tenha uma natureza aparente de Dev, tem uma ligação importante com Ops também, dado que a implantação e a instalação de produtos/serviços são tarefas historicamente associadas a operadores.

Uma abordagem um pouco mais inteligente – e mais aderente à cultura de DevOps – seria um planejamento de processos de build que atenda melhor a rotina de trabalho neste projeto. Nesse novo plano, teríamos dois processos distintos de build, ambos definidos no mesmo servidor Jenkins, de forma que o primeiro, rotineiro e usado durante o horário de expediente regular, envolveria um fluxo bem mais simples de verificação do projeto; o outro, mais completo, seria agendado para executar apenas uma vez ao dia, envolvendo não apenas a verificação do software, mas a análise de qualidade e a gestão dos artefatos gerados. Veremos como colocar esse plano em ação a partir de agora.

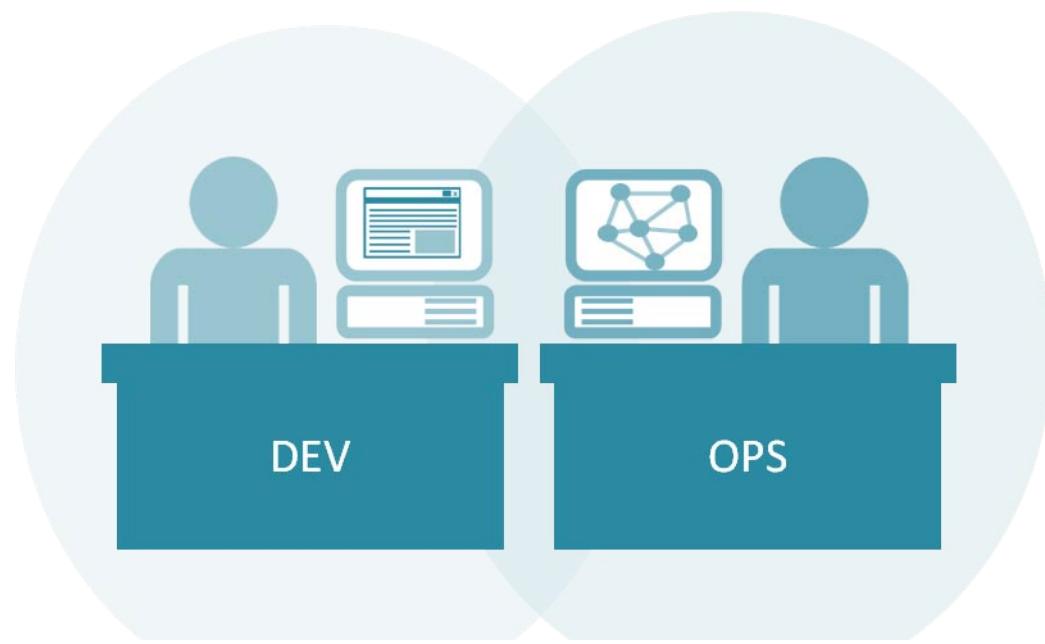
Redesenhandando o processo de integração contínua

Nesta seção, criaremos um novo job em nosso servidor Jenkins. Seu nome será *devmediadevops_daily_job*, e os detalhes de sua configuração estão contidos na **Tabela 1**. Esse é o job a que nos referimos na seção anterior, quando descrevemos um fluxo mais leve, rápido, que apenas verifica a consistência do projeto. Esse será o procedimento a ser executado a cada *commit* de desenvol-

edor e deve, portanto, ser iniciado a partir do repositório remoto hospedado no GitHub. Para isso, precisamos apenas nos certificar que o Web Hook configurado em nossa conta aponte para o job correto (https://devmediajenkins-pedrobrigatto.rhcloud.com/job/devmedia_daily_job/build?token=devmedia_devops_token), como podemos ver na **Figura 4**.

Isto já será suficiente para que o GitHub dispare a execução do job no Jenkins. O resultado prático disso é que, cada vez que um desenvolvedor submeter código ao repositório remoto, terá, dentro de poucos instantes, condições de saber se o que fez refletiu positiva ou negativamente no repositório em que todo o código, de todo o time, encontra-se reunido, integrado. Esse é um medidor fundamental da saúde de um projeto, não apenas para quem produz código, mas para todos os demais membros da equipe, que podem interferir e auxiliar na resolução de problemas tão logo eles sejam verificados.

O próximo passo é alterarmos, também, o primeiro job do outro fluxo de build (*devmediadevops_cleanup_job*). Recuperando as informações do início do artigo, vemos que ele define exatamente o token de autorização que utilizamos no job diário (*devmediadevops_daily_job*) e que, por sua vez, é empregado na configuração do web hook do GitHub, como já ilustrado na **Figura 4**. Precisamos alterar este trigger, abandonando o uso de um token e adotando o modelo de agendamento de builds.



DevOps: Como adequar seu processo de CI a essa nova cultura

A Figura 5 ilustra esse procedimento. Perceba que a opção *Trigger builds remotely* não está mais selecionada, tendo sido substituída pela opção *Build periodically*. O padrão de periodicidade é estabelecido a partir de cinco valores numéricos separados por um espaço em branco, na seguinte ordem:

- Minutos, compreendidos entre 0 e 59;
- Horas, compreendidas entre 0 e 23;
- Dias do mês, compreendidos entre 1 e 31;
- Mês, compreendido entre 1 e 12;
- Dia da semana, compreendido entre 0 e 7 (sendo 0 e 7 correspondentes a domingo).

Pela definição, concluímos que a estratégia adotada para nosso build ‘noturno’ é a de execução diária, com início programado para a meia-noite.

Os benefícios diretos de uma organização como a que acabamos de propor são, principalmente, maior agilidade na verificação do status do projeto por cada operação de commit (por meio de um processo de integração mais leve e objetivo, envolvendo apenas a compilação e verificação do código-fonte) e a garantia de, diariamente, termos um relatório diário da saúde do projeto, cujas informações são essenciais para a orientação de todo o trabalho a ser realizado. Através de relatórios como os gerados pelo

The screenshot shows the GitHub repository settings for 'pedrobrigatto / devmedia_devops_series'. The 'Webhooks & services' tab is selected. Under the 'Webhooks / Manage webhook' section, there is a form to configure a webhook. The 'Payload URL' field contains 'https://devmediajenkins-pedrobrigatto.rhcloud.com/job/devmedia_daily_jc'. The 'Content type' dropdown is set to 'application/json'. The 'Secret' field is empty. Below the form, there is a note about SSL verification and a 'Disable SSL verification' button. Three blue arrows point to the 'Payload URL' field, the 'Content type' dropdown, and the 'Just the push event.' radio button.

Figura 4. Configuração do Web Hook no repositório remoto do projeto

The screenshot shows the Jenkins job configuration for 'devmedia_daily_jc'. In the 'Build Triggers' section, the 'Build periodically' option is selected, while 'Trigger builds remotely' and 'Build after other projects are built' are unselected. The 'Schedule' field contains '0 0 * * *'. A warning message at the bottom says '⚠ Spread load evenly by using 'H 0 * * *' rather than '0 0 * * *''. The 'Poll SCM' checkbox is also unselected.

Figura 5. Configuração da periodicidade de execução do job

Sonar Qube, por exemplo, conseguimos extrair, facilmente, dados muito úteis, tais como débito técnico, porcentagem de cobertura de código via testes unitários, complexidade ciclomática, dentre outros.

Poderíamos, ainda, trabalhar com uma terceira ou quarta estratégia, caso a implantação do produto em servidores de QA e/ou Produção tivesse que seguir uma periodicidade particular, acordada entre todas as partes envolvidas. Tudo depende, portanto, dos acordos firmados, e um bom plano será sempre fundamental para que as expectativas traçadas sejam devidamente atendidas, de todas as partes.

Mas o que isso tem a ver com Ops?

À perspectiva do DevOps, em que o principal objetivo é eliminar a lacuna entre desenvolvimento e operações, um planejamento de builds com qualidade é essencial. Embora, aparentemente, possamos não enxergar um relacionamento explícito com Ops neste instante, um bom planejamento e acompanhamento constante da saúde do projeto são fundamentais para uma alta qualidade do que é entregue. Isso tem efeito direto em eventos subsequentes, como a implantação de sistemas, uma menor incidência de defeitos verificados a cada nova versão, menor esforço com atividades de suporte, dentre outros.

Além disso, o planejamento consistente de builds é importante para que o time de operações planeje adequadamente o provisionamento de recursos (principalmente em modelos de infraestrutura *in-house*, que normalmente requerem maior esforço por parte da equipe local, da própria empresa). Ainda que esteja tudo hospedado na nuvem, em plataformas como OpenStack, OpenShift, CloudBees ou Amazon, a previsibilidade do uso de recursos computacionais garante um desenho – e consequente contratação – mais preciso dos respectivos serviços.

Em todos os artigos mais recentes publicados pela Java Magazine acerca do tema DevOps, utilizamos um modelo de PaaS oferecido através da OpenShift, da Red Hat. A contratação de recursos por meio dessa plataforma depende de inúmeros aspectos, tais como

volume de dados, número de servidores e poder computacional dos mesmos, dentre outros. Portanto, conhecer nossa real demanda é fundamental para contratarmos corretamente.

Vejamos, agora, um aspecto completamente novo em relação ao que iniciamos analisando neste artigo. Uma vez que já estabelecemos dois processos de build separados para o dia-a-dia do time de projeto, precisamos garantir que todos os artefatos sejam devidamente versionados e controlados. Para isso, utilizamos uma solução extremamente popular de gerenciamento de artefatos, chamada Nexus.

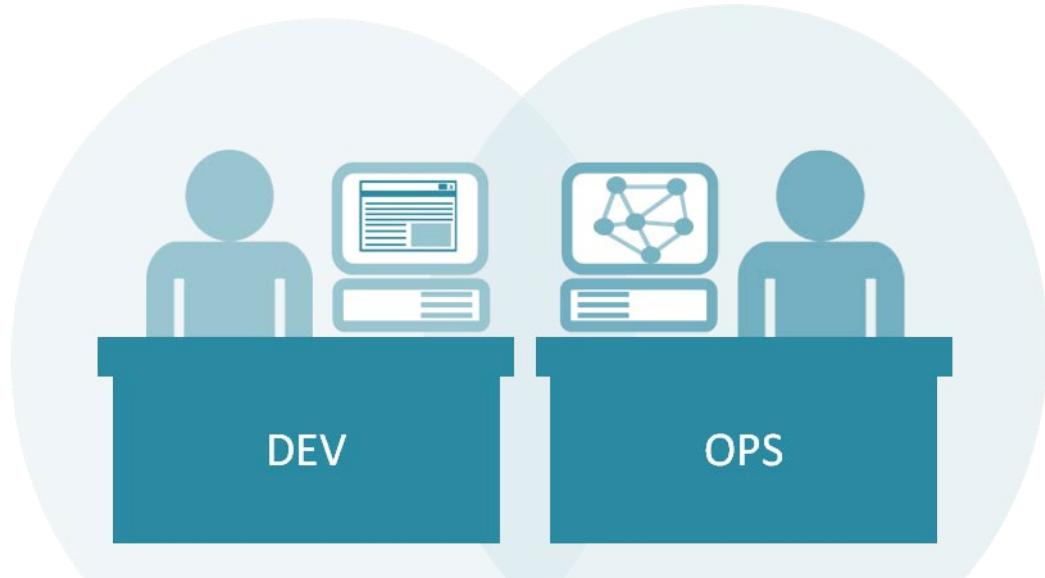
Entrega contínua: o deploy e o repositório de artefatos

O conceito de entrega contínua implica na submissão contínua de versões de um produto ou serviço de software à apreciação de equipes responsáveis pelo processo de verificação. O termo frequentemente usado para os locais em que esse material é publicado é o de ‘ambiente de qualidade’, pois são os profissionais de qualidade que normalmente fazem uso desses ‘entregáveis’, nessas condições.

Esta é, aliás, a grande diferença da Entrega Contínua para a Implantação Contínua (*Continuous Deployment*). Enquanto no primeiro caso estamos falando de um ambiente de qualidade, o segundo envolve exatamente o mesmo procedimento, mas em ambiente de produção (ou seja, acessado diretamente pelo cliente).

Como a OpenShift nos limita a uma quantidade máxima de três *gears* em sua oferta gratuita e gostaríamos que o leitor acompanhasse todos os passos do desenvolvimento deste artigo, optamos por investir nossa ‘última ficha’ na configuração de um servidor de gerenciamento de artefatos. O principal motivo é a importância que um ambiente como esse tem em qualquer empresa séria de desenvolvimento de software, garantindo um controle muito apurado sobre as diversas versões de um produto conforme ele evolui.

Outro motivo é que, como trabalharemos com a transferência de uma aplicação *stand-alone*, a implantação em si nada mais é que a transferência do arquivo em si. Para executá-la, a única exigência



é que a máquina hospedeira tenha o JRE 7 instalado. Um contexto totalmente diferente seria se, por exemplo, estivéssemos lidando com uma aplicação web, que implicaria no uso de recursos adicionais – como um container web – e justificaria, assim, uma demonstração a parte.

Preparando uma nova gear OpenShift e hospedando o Nexus

Para criar esse novo servidor, recorreremos à ferramenta *rhc*. Exatamente como fizemos com a gear *devmedianexus*, herdada do artigo da Edição 149 e já introduzida em seções anteriores, adotamos o cartridge *dyi-1.0* e usamos o esqueleto gerado por ele para instalar uma aplicação pré-configurada do Nexus, disponível em um repositório Git na web.

Começaremos pela análise da **Listagem 1**. O primeiro comando que executamos é o *rhc app create*, passando apenas o cartridge *dyi-1.0* como parâmetro. O resultado dessa operação é o provisio-namento de uma gear camada *devmedianexus*, cujas credenciais são informadas imediatamente após o comando ter sido concluído. Em seguida, acessamos o diretório do projeto, localmente, para iniciar o download e a preparação de uma versão pré-configurada do Nexus para, enfim, hospedá-la na gear recém-criada.

Listagem 1. Terminal – Preparação de uma aplicação DIY para configuração do Nexus na plataforma OpenShift.

```
Pedro-Brigattos-MacBook:DevMediaOpenshiftApps pedrobrigatto$ rhc app create devmedianexus dyi-0.1
RSA 1024 bit CA certificates are loaded due to old openssl compatibility
Application Options
-----
Domain: pedrobrigatto
Cartridges: dyi-0.1
Gear Size: default
Scaling: no

Creating application 'devmedianexus' ... done

Disclaimer: This is an experimental cartridge that provides a way to try unsupported languages, frameworks, and middleware on OpenShift.

Waiting for your DNS name to be available ... done

Cloning into 'devmedianexus'...
Warning: Permanently added the RSA host key for IP address '52.3.119.139' to the list of known hosts.

Your application 'devmedianexus' is now available.

URL: http://devmedianexus-pedrobrigatto.rhcloud.com/
SSH to: 568653ab2d5271af6400003d@devmedianexus-pedrobrigatto.rhcloud.com
Git remote: ssh://568653ab2d5271af6400003d@devmedianexus-pedrobrigatto.rhcloud.com/~/git/devmedianexus.git/
Cloned to: /Users/pedrobrigatto/Personal/Projects/DevMediaOpenshiftApps/devmedianexus

Run 'rhc show-app devmedianexus' for more details about your app.
Pedro-Brigattos-MacBook:DevMediaOpenshiftApps pedrobrigatto$
```

Essa versão do Nexus, que acabamos de mencionar, encontra-se disponível em um repositório do GitHub cuja URL é a *git://github.com/shekhargulati/nexus.git*. Os comandos executados para prepará-la e implantá-la foram agrupados na **Listagem 2**. O primeiro passo que demos foi, de dentro do diretório da gear criada (*devmedianexus*), adicionar ao Git uma referência para o branch do projeto que baixaremos, e cuja URL acabamos de citar. Em seguida, realizamos um merge do conteúdo desse branch com aquele encontrado no diretório *devmedianexus*, dando preferência ao material do branch caso algum conflito seja encontrado. Finalmente, submetemos todo o material para o nosso servidor. O processo de configuração e inicialização é normalmente bem rápido e, assim que concluído, já permite que acessemos a aplicação a partir do painel de administração de nossa conta OpenShift.

Acessando o Nexus a partir do Jenkins: a preparação do ambiente

Assim que a nossa versão do Nexus já estiver disponível, é necessário que configuremos o ambiente da gear *devmedianjenkins* para tornar possível o seu acesso a diretórios desse gerenciador de artefatos. Para isso, conectamo-nos a ela via SSH, usando o comando *ssh* apresentado na **Listagem 3**. Esse servidor Jenkins, é importante lembrar, faz parte do trabalho que herdamos de um artigo anterior, publicado na edição 149 da Java Magazine. Ao estabelecermos comunicação com a gear *devmedianjenkins*, navegamos até seu diretório *\$OPENSHIFT_DATA_DIR/.m2* para editarmos o arquivo *settings.xml* lá encontrado.

Ao abrindo-o para edição, inserimos o conteúdo exibido na **Listagem 4**. As modificações realizadas, na prática, estabelecem o seguinte:

- Um diretório a ser utilizado como repositório local de dependências, definido a partir do nó *localRepository*;
- A configuração dos servidores de *releases* e *snapshots* do Nexus, com suas respectivas credenciais de acesso (a partir de nós *<server>*);

O segundo servidor configurado na **Listagem 4**, que acabamos de mostrar, é usado pelo Jenkins para acessar os diretórios do Nexus, a fim de transferir os arquivos resultantes do processo de build. O usuário definido neste nó, identificado com as credenciais *deployment / deployment123*, é padrão do Nexus, mas o recomendado é que o leitor crie seus próprios usuários e os configure de acordo com as políticas de acesso que desejar, para garantir um controle maior sobre o servidor.

Por fim, salvamos todo o trabalho descrito até aqui. Isso é tudo o que precisamos fazer em termos de preparação de ambiente, do lado da gear *devmedianjenkins*. Na próxima seção, veremos o que precisamos fazer para que o projeto passe a ter seus artefatos implantados no Nexus.

Preparando o projeto para trabalhar com o Nexus

Um dos principais critérios que usamos para adotar o Maven como plataforma de gerenciamento do projeto é a sua grande flexibilidade.

Listagem 2. Terminal – Configuração e ativação do Nexus na plataforma OpenShift.

```
Pedro-Brigattos-MacBook:DevMediaOpenshiftApps pedrobrigatto$ cd devmedianexus/
Pedro-Brigattos-MacBook:devmedianexus pedrobrigatto$ git remote add nexus git://
github.com/shekhargulati/nexus.git
Pedro-Brigattos-MacBook:devmedianexus pedrobrigatto$ git pull -s recursive -X theirs
nexus master
...
Pedro-Brigattos-MacBook:devmedianexus pedrobrigatto$ git push
warning: push.default is unset; its implicit value has changed in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the traditional behavior, use:

git config --global push.default matching

To squelch this message and adopt the new behavior now, use:

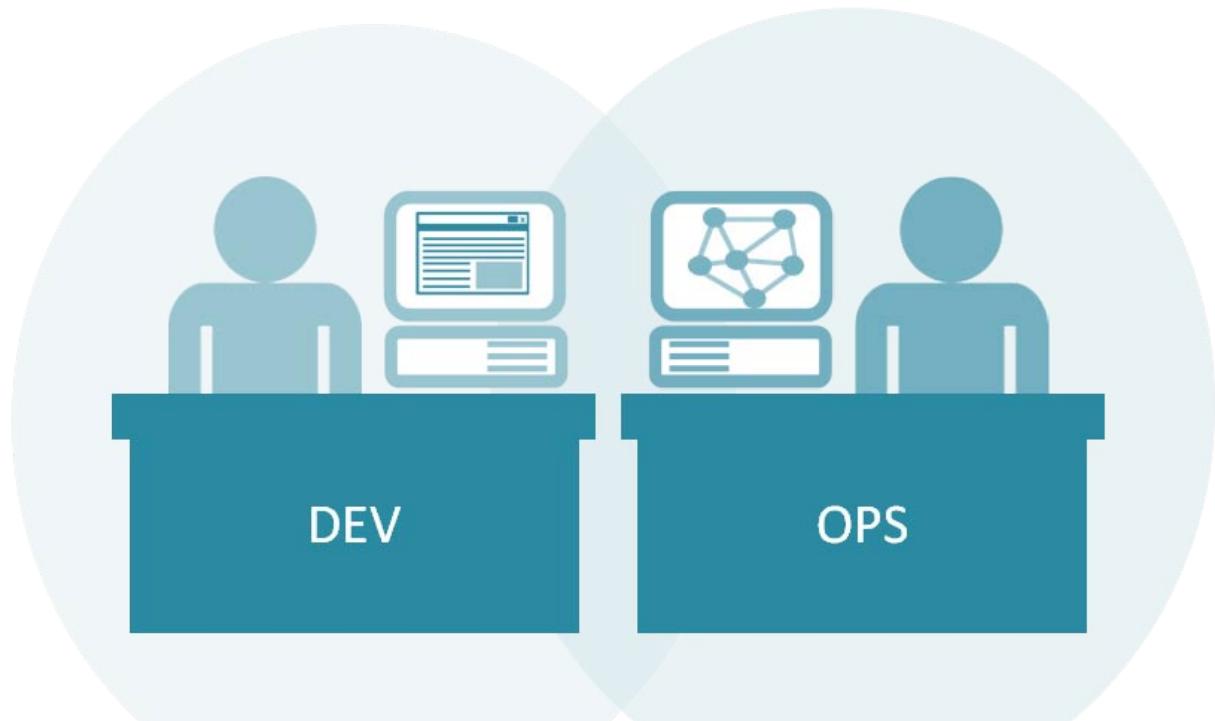
git config --global push.default simple

When push.default is set to 'matching', git will push local branches
to the remote branches that already exist with the same name.

Since Git 2.0, Git defaults to the more conservative 'simple'
behavior, which only pushes the current branch to the corresponding
remote branch that 'git pull' uses to update the current branch.

See 'git help config' and search for 'push.default' for further information.
(the 'simple' mode was introduced in Git 1.7.11. Use the similar mode
'current' instead of 'simple' if you sometimes use older versions of Git)

Counting objects: 771, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (472/472), done.
Writing objects: 100% (771/771), 111.50 MiB | 126.00 KiB/s, done.
Total 771 (delta 284), reused 764 (delta 280)
remote: Stopping DIY cartridge
remote: Building git ref 'master', commit d7b9251
remote: Preparing build for deployment
remote: Deployment id is ea9ba91b
remote: Activating deployment
remote: + ['`-d /var/lib/openshift/568653ab2d5271af6400003d/app-root/
data/tomcat`']
remote: + mkdir /var/lib/openshift/568653ab2d5271af6400003d/app-root/data//prefs
remote: + cp -rf /var/lib/openshift/568653ab2d5271af6400003d/app-root/runtime/
repo//diy/tomcat /var/lib/openshift/568653ab2d5271af6400003d/app-root/data/
remote: + cd /var/lib/openshift/568653ab2d5271af6400003d/app-root/data//tomcat
remote: + rm -rf logs
remote: + ln -s /var/lib/openshift/568653ab2d5271af6400003d/app-root/logs/ logs
remote: + sed -ig s/OPENSHIFT_APP_DNS/devmedianexus-pedrobrigatto.rhcloud.
com/conf/server.xml
remote: Starting DIY cartridge
remote: + export PLEXUS_NEXUS_WORK=/var/lib/openshift/568653ab2d5271af6400003d/app-root/data/
remote: + PLEXUS_NEXUS_WORK=/var/lib/openshift/568653ab2d5271af6400003d/app-root/data/
remote: + export 'CATALINA_OPTS=-Djava.util.prefs.userRoot=/var/lib/openshift/568653ab2d5271af6400003d/app-root/data/prefs'
remote: + CATALINA_OPTS=-Djava.util.prefs.userRoot=/var/lib/openshift/568653ab2d5271af6400003d/app-root/data/prefs'
remote: + cd /var/lib/openshift/568653ab2d5271af6400003d/app-root/data//tomcat
remote: + sed -ig s/OPENSHIFT_INTERNAL_IP/127.5.223.129/g conf/server.xml
remote: + bin/startup.sh
remote: -----
remote: Git Post-Receive Result: success
remote: Activation status: success
remote: Deployment completed with status: success
To ssh://568653ab2d5271af6400003d@devmedianexus-pedrobrigatto.rhcloud.
com/~/git/devmedianexus.git/
  52b117a..d7b9251 master -> master
Pedro-Brigattos-MacBook:devmedianexus pedrobrigatto$
```



DevOps: Como adequar seu processo de CI a essa nova cultura

Por meio de plug-ins, podemos expandir os recursos de nosso projeto e implantar nele inúmeras das técnicas e dos procedimentos que, juntos, caracterizam um contexto típico de DevOps.

No tocante a implantação de artefatos em servidores como o Nexus, o plug-in que adotamos neste artigo é o `org.sonatype.plugins:nexus-staging-maven-plugin`, cujos detalhes veremos adiante.

Listagem 3. Terminal – Dados de acesso ao gear em que o Jenkins está instalado.

```
Pedro-Brigattos-MacBook:DevMediaOpenshiftApps pedrobrigatto$ rhc create-app devmediajenkins jenkins-1 https://cartreflect-claytondev.rhcloud.com/reflect?github=majecek/openshift-community-git-ssh
```

...

```
Creating application 'devmediajenkins' ... done
```

Jenkins created successfully. Please make note of these credentials:

User: admin

Password: *****

Note: You can change your password at: <https://devmediajenkins-pedrobrigatto.rhcloud.com/me/configure>

```
Waiting for your DNS name to be available ... done
```

Cloning into 'devmediajenkins'...

Warning: Permanently added the RSA host key for IP address '54.84.13.138' to the list of known hosts.

Your application 'devmediajenkins' is now available.

URL: <http://devmediajenkins-pedrobrigatto.rhcloud.com/>

SSH to: 5**83***000**@devmediajenkins-pedrobrigatto.rhcloud.com

Git remote: ssh://**20**@devmediajenkins-pedrobrigatto.rhcloud.com/~/git/devmediajenkins.git/

Cloned to: /Users/pedrobrigatto/Personal/Projects/DevMediaOpenshiftApps/devmediajenkins

Run 'rhc show-app devmediajenkins' for more details about your app.

Listagem 4. settings.xml – Configuração do Maven no servidor do Jenkins.

```
<settings>
<localRepository>/var/lib/openshift/5683ab440c1e66ea82000098/app-root/data/.m2</localRepository>

<servers>
<server>
<id>releases</id>
<username>deployment</username>
<password>deployment123</password>
</server>

<server>
<id>snapshots</id>
<username>deployment</username>
<password>deployment123</password>
</server>
</servers>
</settings>
```

Para entender o que precisamos fazer, começaremos pelo estudo da **Listagem 5**. As características que abordaremos a partir de agora estão todas destacadas em negrito. A primeira delas é a alteração de uma propriedade do plug-in padrão utilizado pelo Maven para o processo de *deploy* (a saber, `org.apache.maven.plugins:maven-deploy-plugin`). Basicamente, precisamos solicitar ao Maven que ignore a execução desse plug-in. Fizemos isso para garantir que, para a fase de *deploy* de nosso projeto, seja sempre utilizado outro plug-in, o já citado `org.sonatype.plugins:nexus-staging-maven-plugin`. Essa garantia é estabelecida quando associamos, a esse plug-in que acabamos de citar, o identificador '*default-deploy*', para a fase e o goal *deploy*.

Por fim, ainda no arquivo `pom.xml`, podemos ver a declaração do repositório de snapshots do Nexus. Isso é importante para que, quando o job de deployment do Jenkins for executado, o caminho do repositório seja encontrado. O identificador desse servidor (*id*) é bastante importante, pois o seu valor deve corresponder a algum dos identificadores de servidores declarados no arquivo `settings.xml` do servidor Jenkins, cujo conteúdo já tivemos a oportunidade de avaliar pela **Listagem 4**. Quando todas as informações convergem, o job saberá tudo o que é necessário para que a implantação ocorra: credenciais de acesso, URL e caminho do repositório.

Assim que terminamos a edição do `pom.xml`, precisamos submetê-lo ao controle de versão do repositório GitHub. Para isso, executamos os passos listados a seguir:

- Certificamo-nos que estávamos dentro do diretório raiz do projeto (*devops*);
- Executamos o comando `git add .`;
- Com o comando `git commit -m "comentário de identificação"`, submetemos o conteúdo ao controle de versão local, descentralizado;
- A partir da execução da instrução `git push -u origin master`, enviamos as modificações realizadas localmente para o controle de versão remoto, sob o branch master (nossa único branch desse repositório no GitHub).

A próxima fase desse tutorial, agora que Nexus e Jenkins já estão configurados para se comunicar, é criar o quarto e último job da série para, finalmente, automatizar a transferência de módulos de nosso projeto para o nosso repositório controlado de artefatos.

O quarto job: transferindo artefatos para o Nexus

A criação desse job foi realizada a partir da função *Jenkins > New Item* no portal de administração do Jenkins. Trata-se de um item chamado *devmediadevops_deploy*, do tipo *Freestyle project*, e cujos únicos pontos de atenção encontram-se na seção *Build*.

Nela, clicamos no botão *Add build step* e selecionamos a alternativa *Invoke top-level Maven goals*, preenchendo-a com o conteúdo da **Listagem 6**. Mais uma vez, declaramos a referência completa ao descritor do projeto, por meio do parâmetro '*f*', garantindo que o projeto sempre será encontrado. A novidade aqui, entretanto, é o uso de outro parâmetro, denominado *-DaltDeploymentRepository*. Ele é usado para passar ao Maven uma referência explícita para o repositório no qual desejamos que os artefatos sejam implantados

Listagem 5. pom.xml – Configuração das dependências e do plug-in de deploy do Maven.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.devmedia.articles</groupId>
  <artifactId>devops</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>DevOps</name>

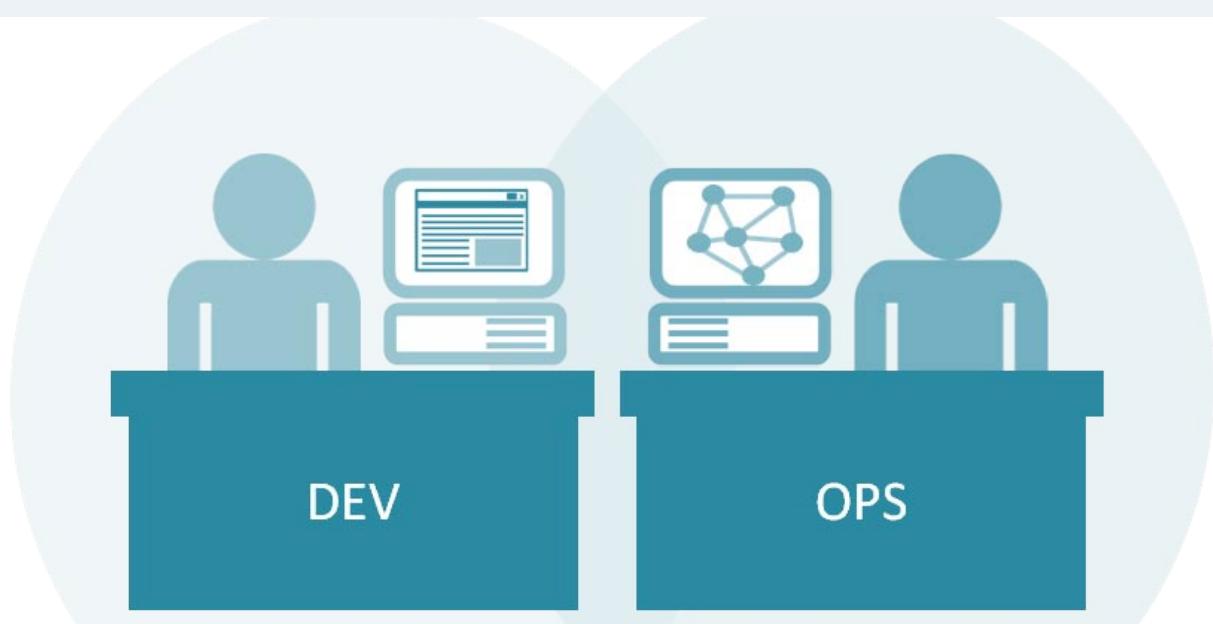
  <properties>
    <spring.version>4.2.2.RELEASE</spring.version>
    <junit.version>4.12</junit.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <sonar.language>java</sonar.language>
    <sonar.plugin.version>2.1</sonar.plugin.version>
  </properties>

  <modules>
    <module>model</module>
    <module>standalone-cli</module>
  </modules>

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

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>
  </dependencies>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>sonar-maven-plugin</artifactId>
          <version>${sonar.plugin.version}</version>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-deploy-plugin</artifactId>
          <configuration>
            <skip>true</skip>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.sonatype.plugins</groupId>
          <artifactId>nexus-staging-maven-plugin</artifactId>
          <executions>
            <execution>
              <id>default-deploy</id>
              <phase>deploy</phase>
              <goals>
                <goal>deploy</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
        <configuration>
          <serverId>releases</serverId>
          <nexusUrl>http://devmedianexus-pedrobrigatto.rhcloud.com/nexus/</nexusUrl>
        </configuration>
        <plugins>
          </plugins>
        </pluginManagement>
      </build>
      <distributionManagement>
        <repository>
          <id>snapshots</id>
          <name>Snapshots</name>
          <url>http://devmedianexus-pedrobrigatto.rhcloud.com/nexus/content/repositories/snapshots/</url>
        </repository>
      </distributionManagement>
    </project>
```



DevOps: Como adequar seu processo de CI a essa nova cultura

(neste caso, *snapshots*). Isso é tudo o que precisamos configurar no Jenkins para que o job consiga cumprir o seu papel. Para finalizar, clicamos em *Save* e registramos todo o trabalho.

Chegamos, enfim, ao final da configuração de toda a cadeia de jobs necessária para demonstrar, na prática, um fluxo simplificado de entrega contínua. As próximas seções, por sua vez, introduzirão alguns recursos, práticas e ferramentas adicionais, com o objetivo de complementar e enriquecer toda a bagagem que adquirimos, até aqui, sobre a cultura de DevOps.

Visualização e gerenciamento de jobs com o Build Pipeline

Em seções anteriores, aprendemos como configurar duas cadeias de jobs. A primeira delas, *devmediadevops_daily_job*, envolve apenas a verificação do código-fonte (compilação e testes) e será utilizada para executar builds automáticas a partir de commits de desenvolvedores. Já a segunda, iniciada a partir do job *devmediadevops_cleanup_job*, é executada diariamente, à meia-noite do horário do servidor em que o Jenkins está implantado, e envolve um conjunto maior de atividades, incluindo uma análise estática da qualidade do projeto e a geração de um relatório a ela relacionado.

Listagem 6. Job *devmediadevops_deploy* – Build step para implantação do artefato no repositório do Nexus.

```
deploy -f $OPENSHIFT_DATA_DIR/workspace/sonar/devops/pom.xml  
-U -DaltDeploymentRepository=snapshots::default::http://devmedianexus-pedro-  
brigatto.rhcloud.com/nexus/content/repositories/snapshots
```

Nesta seção, veremos uma forma gráfica, amigável, de trabalhar com essas cadeias de job. Trata-se de um plug-in do Jenkins chamado Build Pipeline, cuja configuração será discutida ao longo dos próximos parágrafos.

Para orientar o nosso estudo, observemos o conteúdo da **Figura 6**. Esta é a página que vemos quando, a partir da tela inicial da ferramenta de administração do Jenkins (cuja URL está informada na **Listagem 3**), navegamos até o item *Jenkins > Manage Jenkins > Manage Plugins > Available*. No campo de pesquisa informado no topo dessa página, devemos procurar pelo plug-in intitulado *Build Pipeline*. Quando fazemos isso, conforme a **Figura 6** também ilustra, temos como primeira opção o plug-in que desejamos. Basta, então, selecioná-lo e instalá-lo, reiniciando o servidor para que a configuração surta efeito.

Assim que o servidor for reiniciado e acessamos, mais uma vez, o painel de administração do Jenkins, já podemos começar a configurar a visualização da cadeia de jobs em um formato de pipeline. Vejamos a ilustração contida na **Figura 7**. Nela, observamos a existência de um botão com o sinal de "+", indicado pela seta mais ao topo. Ao clicarmos nele e selecionarmos a opção *Build Pipeline View* do formulário que se segue, somos apresentados a uma tela em que essa View será, enfim, configurada. Na seção *Label* da página em questão, observe que existe um campo com os dizeres *Select initial job*, seguido de uma caixa de seleção que lista todos os jobs configurados. Basta que informemos o job desejado (*devmediadevops_cleanup_job* para o caso da cadeia executada periodicamente, uma vez ao dia, ou *devmediadevops_daily_job*, para

The screenshot shows the Jenkins 'Manage Plugins' interface. The 'Available' tab is selected. A blue arrow points to the checkbox next to the 'Build Pipeline Plugin' entry. The plugin details are as follows:

Name	Version
Build Pipeline Plugin	1.4.9
Fail The Build Plugin	1.0
Runscope plugin	1.45
Build Graph View Plugin	1.1.1
Delivery Pipeline Plugin	0.9.8

At the bottom, there are buttons for 'Install without restart', 'Download now and install after restart', and 'Check now'. A status message says 'Update information obtained: 11 sec ago'.

Figura 6. Instalação do plug-in Build Pipeline do Jenkins

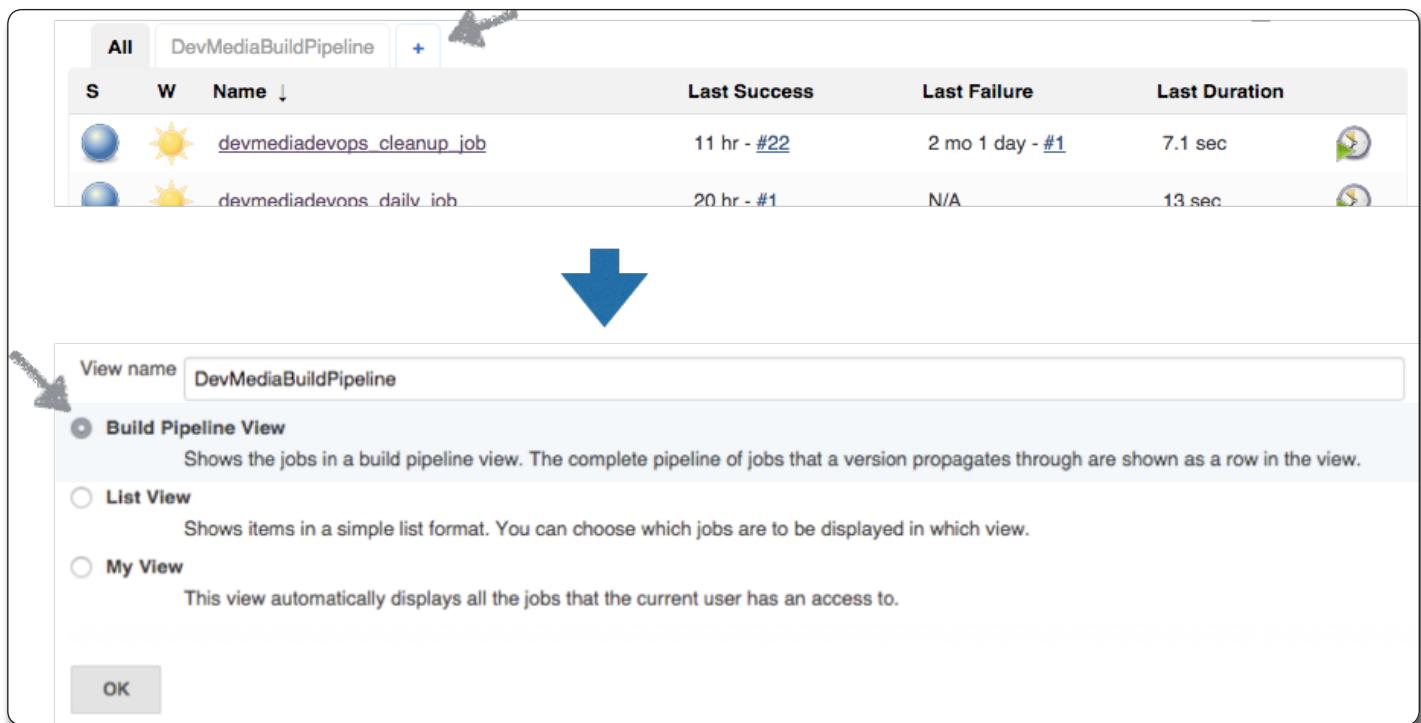


Figura 7. Criação de uma visualização de Pipeline no Jenkins

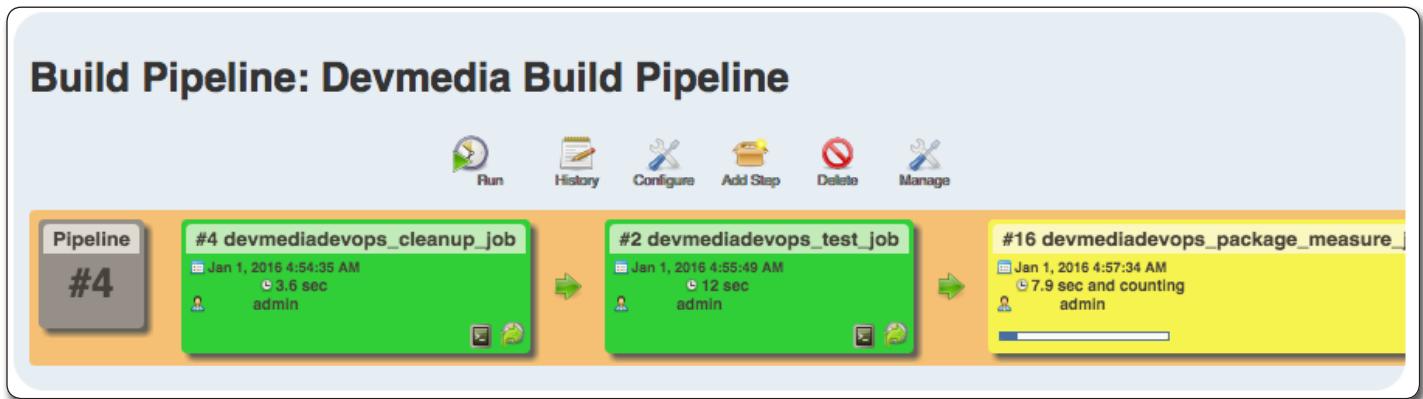


Figura 8. Visualização do pipeline no Jenkins

o job executado a cada submissão de código) e a pipeline será devidamente montada e exibida.

Assim que salvarmos essa configuração, o resultado será algo parecido com o que está ilustrado na Figura 8.

A partir dessa View, podemos testar, manualmente, a cadeia de jobs, acompanhando visualmente o andamento do processo. Da mesma forma, podemos repetir a execução de qualquer job, individualmente, se assim desejarmos. Essa é, portanto, apenas uma maneira amigável de administrar e/ou validar, visualmente, um fluxo de jobs.

Algumas palavras sobre provisionamento

Todo o conteúdo visto até esta seção tem um viés mais voltado para atividades ligadas ao desenvolvimento de software

do que àquelas tipicamente associadas ao time de operações. Entretanto, em DevOps, a alta qualidade no resultado final passa, necessariamente, por uma boa execução de todas as tarefas envolvidas no processo e, como procuramos, também, evidenciar, mesmo os movimentos de um único desenvolvedor podem gerar efeitos em todo o restante da cadeia, afetando inclusive a 'rotina Ops'.

Em DevOps, um dos temas categorizados como de operações – e que, até aqui, abordamos apenas implicitamente, por meio das plataformas que utilizamos – é o de provisionamento de recursos. Esse assunto, tal como visto até esse momento, foi sempre algo que usamos de forma transparente, ao consumirmos servidores – ou, melhor dizendo, *gears* – criados e gerenciados a partir de uma conta OpenShift.

No entanto, nem sempre teremos à nossa disposição os servidores da empresa – ou bancados por ela – para executar nossos testes locais. Nesses casos, podemos adotar as mesmas tecnologias que essas soluções de PaaS usam e, assim, montar nossos próprios ambientes para, senão replicar, simular o mais próximo da realidade os ambientes-alvo de nossas aplicações (mesmo sistema operacional, mesmos caminhos de diretório, mesmos scripts, etc.).

Uma das tecnologias muito populares no quesito de provisionamento de recursos, e que estudaremos de forma introdutória no texto que se segue, é o Vagrant. Essa ferramenta é desenvolvida e mantida por uma empresa chamada HashiCorp e apresenta uma forma muito simples para criar, configurar e gerenciar máquinas virtuais, atuando logo acima de ferramentas de virtualização como VMware e VirtualBox.

Essa facilidade de uso deve-se principalmente à forma como o Vagrant se apresenta para nós. Toda a definição de uma infraestrutura é escrita a partir de um arquivo chamado *Vagrantfile*, que se usa de uma linguagem pré-definida – e muito bem documentada em seu site oficial (disponível na seção **Links**) – para, passo a passo, criar todos os nós de um ambiente desejado, consideradas todas as características de poder de processamento, quantidade de espaço em disco, memória, configurações de rede, dentre outros. Veremos, logo mais, um pouco sobre a estrutura desse arquivo.

Assim que o Vagrant é instalado, podemos passar a utilizá-lo via linha de comando. Trata-se de uma ferramenta cujas instruções são bastante simples e com as quais rapidamente nos familiarizamos à medida do uso. Ao abrir um terminal e digitar, por exemplo,

vagrant -v, sua versão será impressa. O próximo passo para iniciar os trabalhos com essa ferramenta é escolher um diretório em que nosso projeto será criado. Feito isso, basta que, via linha de comando, executemos *vagrant init* que, automaticamente, gerará o já citado arquivo descritor do projeto (*Vagrantfile*).

A partir de agora, veremos um exemplo prático de como é construído um arquivo desses, tomando como base o conteúdo da **Listagem 7**.

Essa listagem apresenta a configuração de um cluster composto por três nós (máquinas virtuais) comunicando-se a partir de uma rede privada. A primeira linha dentro do bloco de configuração de nosso projeto consiste na definição da imagem a ser usada que, no vocabulário do Vagrant, é conhecida como *box*.

Nesse exemplo, usamos uma imagem do sistema operacional CentOS (*chef/centos-6.5*). Essa imagem será, em um primeiro momento, baixada para nossa máquina de trabalho e, posteriormente, utilizada no provisionamento. A imagem do CentOS, bem como uma série de outras (de outros sistemas operacionais, versões, etc.) encontram-se disponíveis a partir de um catálogo padrão mantido pela própria HashiCorp, conhecido como *Atlas*. O endereço para acessar esse repositório de imagens pode ser verificado na seção **Links**.

Embora estejamos utilizando uma imagem pública de uma máquina CentOS, nada impede, caso precisemos ou queiramos, definir as nossas próprias *boxes*, publicando-as em um repositório público ou privado, dependendo dos critérios adotados pela empresa ou pelo projeto.

O segundo ponto destacado em negrito na **Listagem 7** corresponde à definição de uma rede privada sobre o protocolo DHCP.

Listagem 7. Vagrantfile – Configuração de um cluster a partir do Vagrant.

```
# -*- mode: ruby -*-
# vi: set ft=ruby:

Vagrant.configure(2) do |config|


  config.vm.box = "chef/centos-6.5"

  if Vagrant.has_plugin?("vagrant-cachier")
    config.cache.scope = :box

    config.cache.enable :generic, {
      "wget" => { cache_dir: "/var/cache/wget" },
      "curl" => { cache_dir: "/var/cache/curl" },
    }
  end

  config.vm.network "private_network", type: "dhcp"

  config.vm.define "clusternode1" do |clusternode1|
    clusternode1.vm.network "forwarded_port", guest: 8082, host: 8082
    clusternode1.vm.provider "virtualbox" do |vb|
      vb.memory = "1024"
      vb.name = "clusternode1"
    end
  end

  config.vm.define "clusternode2" do |clusternode2|
    clusternode2.vm.network "forwarded_port", guest: 8082, host: 8083
    clusternode2.vm.provider "virtualbox" do |vb|
      vb.memory = "1024"
      vb.name = "clusternode2"
    end
  end

  config.vm.define "clusternode3" do |othernode1|
    clusternode3.vm.provider "virtualbox" do |vb|
      vb.memory = "1024"
      vb.name = "clusternode3"
    end
  end
```

Perceba o quanto é simples uma configuração que, normalmente, tomaria algum tempo caso tivesse que ser feita manualmente em ferramentas como o VirtualBox, por exemplo.

Em seguida, vemos o primeiro bloco correspondente a um nó do cluster em configuração. Note que usamos, aqui, uma propriedade denominada `config.vm.define` para definir uma nova VM e, como estamos criando mais de uma máquina nesse mesmo arquivo, rotulamos esse nó em particular como `clusternode1`. Os recursos e características que definimos para essa máquina são:

- Ferramenta de provisionamento: Virtual Box;
- Total de memória RAM disponível: 1 GB (dado que a unidade padrão é MB);
- Nome do nó: `clusternode1`;
- Redirecionamento da porta 8082 do guest (VM) para a porta 8082 do host.

Outro ponto muito importante e interessante da configuração é que, uma vez que a máquina tenha sido criada e esteja ativa, podemos informar ao Vagrant que o restante do procedimento de preparação será realizado a partir de um script desenvolvido por nós (e, portanto, altamente customizável). Isso é feito quando declaramos a instrução `clusternode1.vm.provision`, usando `:shell` como opção e, em seguida, informando o caminho (relativo à raiz do projeto) para o script que deve ser executado.

Da mesma forma, outros dois nós foram configurados, sendo que o último, somente com finalidade didática, de ilustração, utiliza um script diferente dos dois primeiros, sugerindo que atua como um nó de平衡amento de carga entre os dois nós iniciais do cluster (observe que, nesse caso, declaramos o uso de um script denominado `vm_bootstrap_with_loadBalancer.sh`).

Embora, a princípio, a sintaxe possa confundir um pouco, o importante a ser extraído desse material é que ocorre, a partir do conteúdo nele declarado, um procedimento relativamente complexo de preparação de ambiente, convertido em poucas linhas

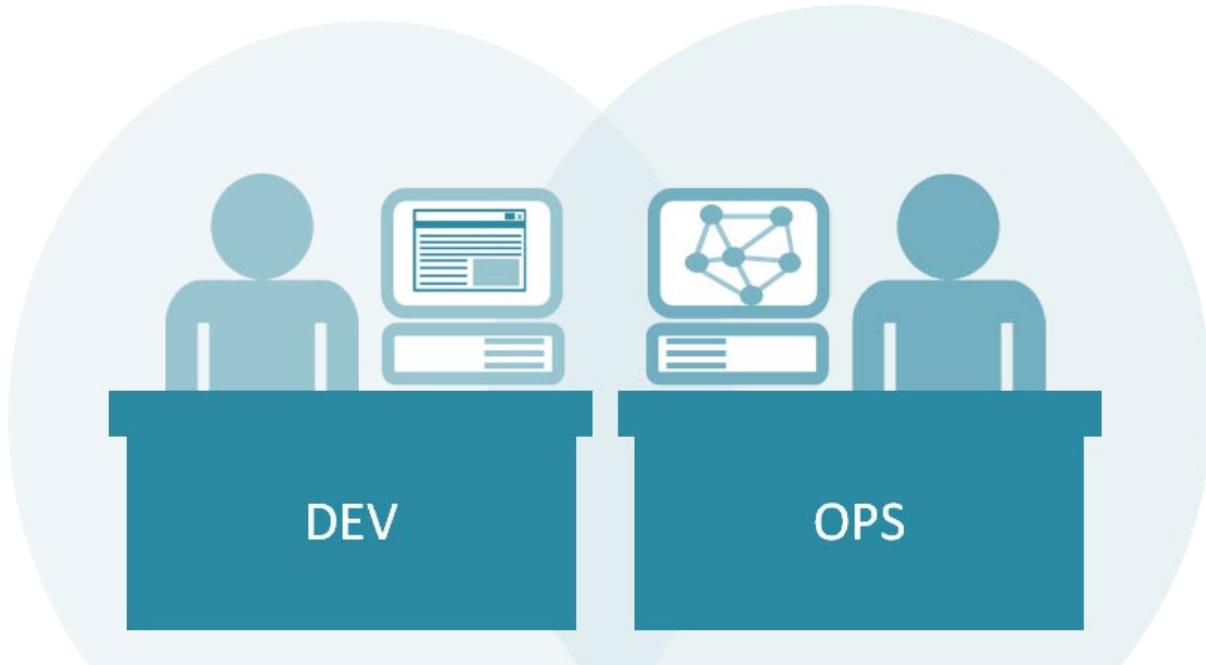
de um arquivo de declaração. Além disso, é válido notar que estamos falando de um arquivo de configuração que pode, portanto, ser versionado como qualquer outro. Uma vez submetido ao controle de versão, pode ser baixado, utilizado e até editado por qualquer membro do projeto, estabelecendo-se um ambiente de testes que todos os desenvolvedores do projeto considerarão em seu dia a dia de trabalho (seja localmente, caso suas máquinas suportem, ou mesmo em algum laboratório disponibilizado pela empresa).

Assim que o projeto estiver configurado e esse arquivo for salvo, podemos subir os nós deste cluster usando, via terminal, o comando `vagrant up <nome do nó>`. Caso não forneçamos nenhum nome de nó, todos eles serão iniciados. Por sua vez, quando informamos um nome específico, apenas aquele nó será criado e colocado em execução.

Assim que a(s) máquina(s) estiver(em) disponível(eis), podemos acessá-la(s) usando o comando `vagrant ssh <nome do nó>`, o que abre uma sessão de comunicação remota com a(s) máquina(s) em questão.

Por fim, quando tivermos encerrado nossas atividades e não mais desejarmos utilizar essa máquina, podemos tanto destruí-la, por meio do comando `vagrant destroy <nome do nó>` (liberando todos os recursos, inclusive espaço em disco, do hospedeiro), quanto somente desligá-la temporariamente, por meio do comando `vagrant halt <nome do nó>`.

Além do Vagrant, há outras soluções tão ou mais atraentes disponíveis no mercado (muitas delas gratuitamente). O OpenStack, por exemplo, é uma excelente plataforma cujo estudo recomendamos ao leitor que se interessa pelo tema. Há, ainda, aquelas que, por si só, são extremamente poderosas, mas que, ainda por cima, oferecem excelente integração com o Vagrant. Um exemplo fabuloso é o Docker, que tem uma filosofia diferente por trabalhar com containers ao invés de



máquinas virtuais, mas que se integra perfeitamente com o Vagrant, unindo o melhor dos dois mundos em uma solução combinada e poderosíssima de provisionamento de ambientes. Soluções mais recentes, como o *Buddy* (veja a seção **Links**), surgiram trazendo uma oferta de todo esse ferramental já devidamente integrado e pronto para ser utilizado, seguindo um modelo de oferta parecido com o da OpenShift. A referência para essa opção pode ser encontrada na seção **Links**.

DevOps é um conceito ainda muito recente e que, definitivamente, não pode ser implantado em empresa alguma sem passar por um intenso planejamento, muita discussão e uma fina sintonia entre todos os departamentos e pessoas envolvidas.

Trata-se de algo que envolve não apenas ferramentas, tecnologias, mas, principalmente, pessoas e uma mentalidade totalmente diferente do que se vinha aplicando no mundo corporativo até pouco tempo atrás. Ainda que empresas se adequem, existe um desafio ainda maior a ser vencido, que é incorporar o cliente nessa nova filosofia. O sucesso de uma prática DevOps só chegará nos casos em que todos, compreenderem o que está sendo feito, os objetivos a serem atingidos e as novas responsabilidades que surgem com esse novo formato.

A área de operações, historicamente, envolve uma série de processos que não serão automatizados tão cedo. Empresas tradicionais, com negócios tradicionais, com mentalidade mais conservadora, ainda compõem uma parcela significativa do mercado. Nesses casos, o encanto de uma abordagem promissora como DevOps esbarra, naturalmente, em preconceito, receio, incertezas e, principalmente, uma necessidade forte de provas, de fatos, de qualquer coisa que sustente, antecipadamente, todas as promessas que os ares do DevOps carregam consigo.

Desenvolvimento, por sua vez, é a área que menor pressão sofre, pois, ainda que não se consiga implantar rapidamente uma cultura DevOps em uma empresa, inúmeras das técnicas, práticas, tecnologias e plataformas apresentadas ao longo do texto podem ser, ainda que a conta-gotas, inseridas no cotidiano dos projetos e provar-se úteis e vantajosas. Esse modelo de inserção gradual, quase que em um caráter de prova de conceito, é muito mais bem aceito em um contexto de desenvolvimento do que naquele de operações.

Reducir essa lacuna entre Dev e Ops é, principalmente, trabalhar as pessoas e suas particularidades. É buscar, todos juntos, listar medidas que possam ajudar a tornar esse elo mais forte, menos turbulentos.

O ponto fundamental é desenvolver uma cultura corporativa em que as pessoas compreendam o valor real de uma mudança como a que DevOps propõe em suas linhas gerais para, só então, dar a guinada a partir de práticas e tecnologias disponíveis no mercado. O aparato técnico é imenso e muito bom, mas só funcionará em um modelo genuinamente DevOps se as mentes pensantes por trás de tudo isso absorverem, de fato, seus princípios.

Autor



Pedro E. Cunha Brigatto

pedrobrigatto.devmedia@gmail.com

Engenheiro da Computação graduado pela Universidade Federal de São Carlos, desenvolvedor certificado SAP Netweaver (Java Stack) e programador certificado SCJP. Especialista em Engenharia de Software graduado pela Unimep e pós-graduado em Administração pela Fundação BI-FGV, atua com desenvolvimento de software desde 2005. Atualmente atua como consultor técnico no desenvolvimento de soluções de alta disponibilidade na Avaya.



Links:

Código do projeto tema no GitHub.

https://github.com/pedrobrigatto/devmedia_devops_series

Texto refletindo sobre a opção entre Gradle e Maven.

<http://devops.com/2015/03/27/puzzle-gradle-maven/>

Trabalhando com hooks no Git.

<https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks>

Atlas – catálogo de imagens da HashiCorp.

<https://atlas.hashicorp.com/boxes/search>

Página oficial do Buddy.

<https://buddy.works/>

Você gostou deste artigo?

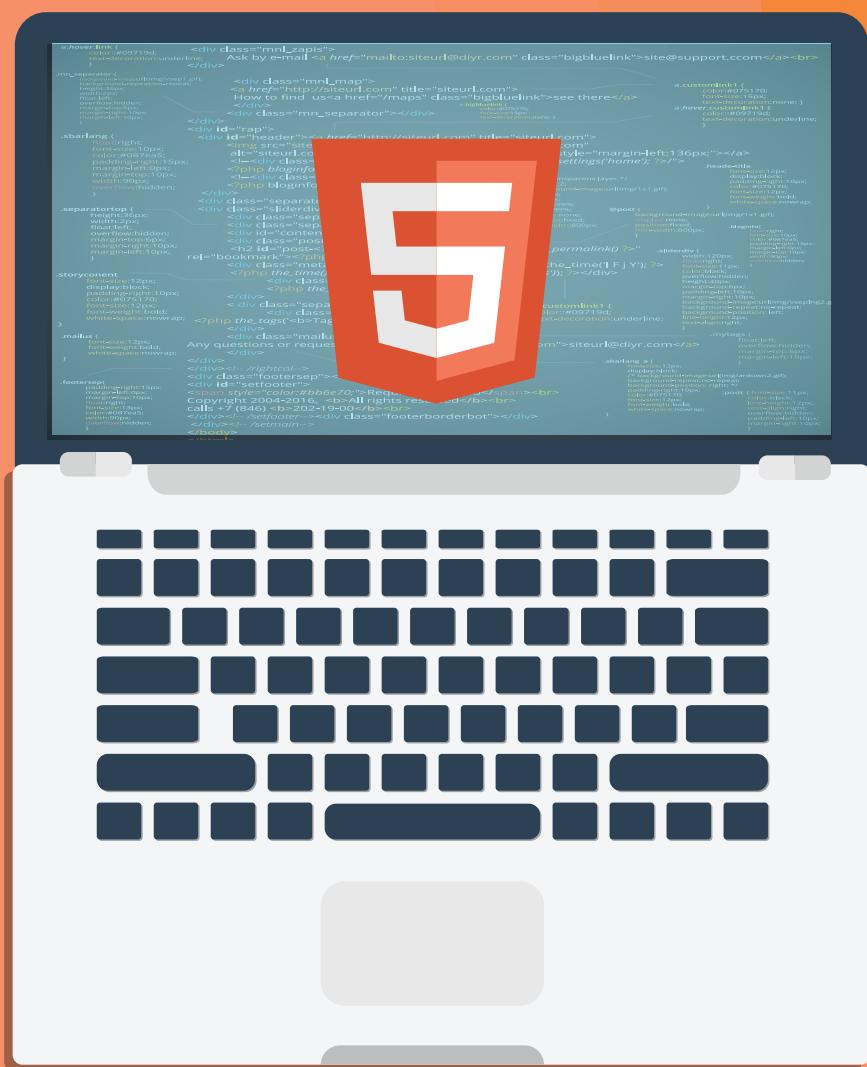
Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Guia HTML 5

Um verdadeiro manual de referência com tudo que você precisa sobre HTML!



DEVMEDIA

<http://www.devmedia.com.br/guias/guia-html/3>

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