



Edição 149 :: R\$ 14,90

 DEV MEDIA

DevOps e Integração Contínua nas nuvens

Turbine o DevOps automatizando a análise de projetos com SonarQube

Jigsaw: Desenvolvimento modularizado no Java 9

Conheça a principal novidade da próxima versão do Java

CASSANDRA E COUCHBASE

Persistência de dados com soluções NoSQL

Spring e AngularJS
Alta produtividade com Java e JavaScript

Boas práticas de desenvolvimento
Aprenda a programar com qualidade



MVP

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

APlique esse investimento
na sua carreira...

E mostre ao mercado
quanto você vale!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

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

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

 **DEVMEDIA**



Edição 149 • 2016 • ISSN 1676-8361

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

Conteúdo sobre Novidades

06 – Projeto Jigsaw: Desenvolvimento modularizado no Java 9

[Cristiano Mariano de Oliveira]

Conteúdo sobre Boas Práticas

16 – Aplicando boas práticas em todo o processo de desenvolvimento

[André Pedralho]

Artigo no estilo Curso

24 – Por dentro do banco de dados NoSQL Couchbase – Parte 1

[Fernando Henrique Fernandes de Camargo]

Artigo no estilo Curso

31 – Como usar o Apache Cassandra em aplicações Java EE – Parte 1

[Marlon Patrick]

Conteúdo sobre Boas Práticas, Conteúdo sobre Novidades

44 – Alta produtividade em Java com Spring e AngularJS

[Rodrigo Engelberg]

Conteúdo sobre Novidades, Artigo no estilo Solução Completa

54 – DevOps e a Integração Contínua nas nuvens

[Pedro E. Cunha Brigatto]



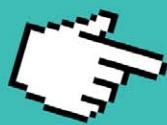
Dê seu feedback sobre esta edição!

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>

 **DEVMEDIA**

Projeto Jigsaw: Desenvolvimento modularizado no Java 9

Entenda como desenvolver e distribuir aplicações Java menores, mais rápidas e seguras

No último mês de outubro ocorreu, em São Francisco, Califórnia, o JavaOne 2015, conferência anual do Java realizada pela Oracle. Esta edição contou com mais de 400 sessões, incluindo apresentações técnicas, hands-on, tutoriais, keynotes e sessões de discussão. Em um dos keynotes mais aguardados, Mark Reinhold, arquiteto-chefe da plataforma Java, utilizou todo o seu tempo para falar sobre o Projeto Jigsaw. Ainda no evento, foi realizada uma série de workshops para mostrar o estado atual do desenvolvimento das funcionalidades trazidas pelo recurso e, assim, aumentar o entendimento e o feedback da comunidade sobre elas. Essa maior exposição do projeto Jigsaw deixa clara sua relevância para a tecnologia Java e a importância que a Oracle está dando ao assunto.

Mas, afinal, o que é esse tal Projeto Jigsaw? É um projeto para a adição de funcionalidades de modularização à plataforma Java. Os dois principais objetivos são: em primeiro lugar, a definição e a implementação de um modelo de módulos que possa ser utilizado no desenvolvimento de novas aplicações; e, por último, mas não menos importante, a modularização do JRE e do JDK em si, utilizando esse mesmo modelo.

Caso você já tenha estudado ou utilizado OSGi, deve estar familiarizado com o desenvolvimento modular e deve estar, também, se perguntando o motivo de tanto barulho em torno do Projeto Jigsaw. A resposta para esse questionamento, no entanto, é bem simples: além da facilidade de se ter a funcionalidade nativamente, o motivo pelo qual se decidiu implementar a solução

Fique por dentro

Este artigo é útil nas situações onde o desenvolvedor queira entender e aplicar as funcionalidades de modularização que estarão disponíveis na versão 9 do Java, através do Projeto Jigsaw. Essas novas funcionalidades têm como objetivo definir um modelo de módulos que poderá ser utilizado pelos desenvolvedores e também na modularização do JRE e do JDK em si. Inicialmente, veremos a importância do projeto para a comunidade Java, suas motivações e seu histórico. Também passaremos pela compatibilidade com o código existente e algumas das deficiências do projeto. Por fim, para que os conhecimentos adquiridos sejam melhor fixados, teremos no artigo uma parte prática, onde será construída uma aplicação dividida em módulos e, então, será gerada uma imagem customizada de JRE contendo apenas os módulos necessários para a execução desta aplicação.

dentro da plataforma é o fato de só assim ser possível modularizar a própria plataforma.

Para entendermos melhor o Projeto Jigsaw e suas implicações para os desenvolvedores Java, primeiramente falaremos sobre as motivações por trás dos principais objetivos: o modelo de módulos e a modularização do JRE e do JDK. Em seguida, passaremos por um breve histórico do projeto, desde suas origens no OSGi, até sua estrutura atual e os adiamentos que ocorreram em sua entrega. Falaremos também sobre compatibilidade, o que pode afetar principalmente os desenvolvedores que utilizam recursos internos do JDK, que deveriam ser utilizados somente pelos desenvolvedores da plataforma. Para fechar a parte teórica do artigo, analisaremos algumas deficiências do projeto, como a ausência de um mecanismo para o controle de versões dos módulos.

Logo após, teremos uma parte prática, com exemplos que visam facilitar o entendimento e mostrar o poder dessa nova tecnologia. Assim, será construída uma aplicação de linha de comando que executa um simples cálculo de adição. A partir disso, a aplicação será dividida em módulos, para que seja possível ver o relacionamento e as restrições entre eles, e ao final, será gerada uma imagem de JRE contendo somente os módulos da plataforma utilizados pela nossa aplicação.

Modelo de Módulos

Atualmente, os modificadores de acesso Java desempenham muito bem seu papel no encapsulamento das classes. Porém, isso ocorre somente quando as classes envolvidas estão no mesmo pacote. Se uma classe precisa ser acessada por outra classe localizada em um pacote diferente, ela precisa necessariamente ser pública. Além de reduzir a segurança, essa limitação contribui em muito para o JAR Hell (vide **BOX 1**). A ideia no Projeto Jigsaw é que os módulos sirvam como uma camada adicional de abstração, acima do nível de **package**, permitindo que se declare quais são os pacotes que serão exportados para módulos que dependam do módulo que está sendo desenvolvido. A estrutura de um módulo pode ser vista na **Figura 1**. Com isso, os pacotes que não estiverem declarados como exportados ficarão visíveis somente por classes que estejam dentro do mesmo módulo. Quando analisamos esse objetivo mais detalhadamente, fica claro que ganhamos além de segurança, desempenho, já que o escopo para a busca de classes no tempo de execução será reduzido somente às classes que são exportadas.



Figura 1. Estrutura de um módulo

Modularização do JRE e do JDK

Por conta da adição de novas funcionalidades, o JRE tem aumentado bastante de tamanho a cada nova versão do Java. Como o JRE é uma aplicação monolítica, a máquina que vai executar qualquer aplicação Java deve ter sua instalação completa, independentemente do tamanho da aplicação que tenha sido desenvolvida. Essa característica geralmente não é uma limitação para

BOX 1. JAR Hell

JAR Hell é como são conhecidos os problemas decorrentes do conflito entre versões de uma ou mais classes quando uma aplicação Java é executada. Quando estão presentes no classpath mais de uma versão da mesma classe, a JVM utiliza a classe que for encontrada primeiro. No entanto, não necessariamente essa é a versão da classe que deve ser utilizada pelo trecho de código que está sendo executado. Isso pode fazer com que a aplicação gere algum erro ou, o que é pior, tenha um comportamento diferente do esperado e traga grandes prejuízos. Embora pareça algo que não ocorra com frequência, é muito fácil exemplificarmos esse cenário com um caso concreto. Supondo que sua aplicação utilize Hibernate (que tem o **dom4j** como dependência) e que você também queira utilizar o **dom4j** em uma outra funcionalidade, caso você utilize uma versão diferente daquela que é referenciada pelo Hibernate, você pode acabar com os JARs das duas versões no classpath da sua aplicação. Nesse caso, qual versão da classe será utilizada pelo código que você desenvolveu? É difícil saber. Sua aplicação acaba de entrar no "Inferno dos JARs"! Assim, ela poderá ter diferentes comportamentos em diferentes ambientes e talvez você comece a ver os sintomas da síndrome do "Na minha máquina funciona".

computadores, mas pode inviabilizar a instalação de aplicações em alguns dispositivos menores, que têm fortes restrições de armazenamento e memória.

Pensando nisso, na versão 8 do Java foi feito um grande avanço na direção da solução desse problema: foram incluídos os Compact Profiles. Esses profiles nada mais são do que três subconjuntos possíveis do JRE completo, contendo apenas alguns pacotes. Essa possibilidade, imediatamente, passou a beneficiar os desenvolvedores que precisavam distribuir suas aplicações para dispositivos menores. Porém, as versões disponíveis não atendem com exatidão as necessidades das aplicações.

Com a modularização da plataforma é possível criar imagens que contenham apenas os módulos que sejam essenciais à execução das aplicações. Esse recurso facilita em muito a distribuição para dispositivos menores, como os que estão sendo utilizados para suportar aplicações para a Internet das Coisas (IoT), e é visto por muitos desenvolvedores como o maior ganho trazido pelo projeto.

Histórico

Embora tenha se falado muito em modularização recentemente, esse não é um assunto novo. Para se ter uma ideia de como essa questão vem sendo discutida há tempos pela comunidade, a OSGi Alliance foi fundada em 1999. Antes conhecida como *Open Services Gateway initiative*, a OSGi Alliance é um consórcio que tem como objetivo criar especificações e implementações de referência para modularização de sistemas feitos em Java.

Alguns anos mais tarde, em 2005, foi criada a JSR 277 (*Java Module System*), que tratava basicamente das mesmas áreas que o OSGi. Já em 2007, ainda na época da Sun, foi criado o projeto Jigsaw, que além de cobrir os objetivos da JSR 277, acrescentou o objetivo de modularização da plataforma. Embora o projeto seja antigo, ele demorou a deslanchar. Como foi iniciado nos anos finais da Sun, que já não tinha as mesmas condições financeiras dos tempos áureos, não foi designado um time muito forte para o projeto. Ademais, a maioria dos desenvolvedores não estava alocada

Projeto Jigsaw: Desenvolvimento modularizado no Java 9

somente no Projeto Jigsaw, e por isso, dedicava somente parte de seu tempo a ele. Em 2009, durante o período de aquisição da Sun pela Oracle, o projeto chegou inclusive a ficar congelado. Porém, em 2011, a comunidade começou a perceber que um sistema de módulos era peça fundamental para a continuidade da adoção da plataforma.

A tecnologia OSGi é a mais utilizada para o desenvolvimento de aplicações modularizadas atualmente e é o maior concorrente da tecnologia de módulos implementada no Projeto Jigsaw, sendo empregada na construção da IDE Eclipse e também por muitos desenvolvedores de aplicações. Esta é uma tecnologia madura e, com certeza, será utilizada por muito tempo, mesmo após a introdução de módulos como uma funcionalidade nativa no Java. O maior problema do OSGi é que, por ser uma tecnologia implementada em cima da plataforma, não serviria para modularizar a própria plataforma. Por isso a necessidade de se construir um modelo de módulos próprio.

Nota

Para que seja possível unir as forças das duas tecnologias em uma mesma aplicação, foi criado o Projeto Penrose, que tem como objetivo permitir a interoperabilidade entre a tecnologia nativa de módulos e o OSGi.

A estrutura atual

Em 2011 o projeto voltou com força total. Começou aí uma fase exploratória de quatro anos, finalizada em 2014. Essa fase moldou o projeto em sua estrutura atual, a qual é formada por cinco JEPs e uma JSR (vide **BOX 2**): a JSR 376 (*Java Platform Module System*), que define o sistema de módulos da plataforma. Os objetivos das JEPs do projeto são:

- **JEP 200 (The Modular JDK)**: discutir como seria feita a modularização do JDK, ou seja, como seriam divididas em módulos as classes do JDK;
- **JEP 201 (Modular Source Code)**: implementar no código as decisões tomadas na JEP 200;
- **JEP 220 (Modular Run-Time Images)**: definir a estrutura das imagens de JRE;
- **JEP 260 (Encapsulate Most Internal APIs)**: definir quais as classes internas do JRE que poderiam ser acessadas pelos desenvolvedores e quais deixariam de estar disponíveis; e
- **JEP 261 (Module System)**: implementar no código a JSR 376.

Adiamentos na entrega

A entrega das funcionalidades de modularização estava inicialmente prevista para o Java 7. No entanto, como o projeto acabava de ser retomado e tem um grau de complexidade altíssimo, não foi possível atender à expectativa inicial. A versão 7 foi, então, lançada sem essas funcionalidades, que foram postergadas para a versão 8.

À época do lançamento do Java 8 o projeto já tinha a maioria de suas definições, mas ainda não tinha um protótipo implementado. Por isso, mais uma vez, ficou decidido que a versão 8 seria lança-

da e as funcionalidades seriam postergadas para o Java 9. Esses atrasos causaram muito barulho na comunidade e algumas vozes chegaram, até mesmo, a sugerir o abandono do projeto e a adoção do OSGi como padrão para a modularização. Contudo, a Oracle defende que foram seguidas as premissas do desenvolvimento ágil: se uma funcionalidade não está pronta, entregue a versão sem ela e adie essa funcionalidade para a próxima versão.

A data de corte para o desenvolvimento de todas as funcionalidades do Java 9 era 10 de dezembro de 2015. Nessa data deveriam ser iniciadas as fases de ajuste para que o lançamento fosse realizado em setembro de 2016. Porém, como apenas recentemente foi lançada uma versão do JDK onde é possível testar a modularização e a comunidade está dando bastante feedback, Mark Reinhold decidiu solicitar um adiamento de seis meses para a entrega do Java 9. Assim, a nova previsão é que todas as funcionalidades estejam desenvolvidas em maio de 2016, quando iniciarão os ajustes para que o lançamento seja feito em março de 2017.

BOX 2. JSR

JSR (Java Specification Request) é o nome dado ao documento que é criado para a discussão e implementação de qualquer funcionalidade na plataforma Java. Respeitando o processo de desenvolvimento do Java, as JSRs são submetidas ao JCP (Java Community Process), um mecanismo criado para que a comunidade possa participar na definição das especificações. Nesse processo, as JSRs passam por revisões públicas e, após a aprovação pelo comitê executivo, têm sua versão final lançada, em conjunto com o código-fonte de uma implementação de referência.

A existência das JSRs é uma das características que explicam o sucesso da plataforma Java. Embora a Oracle coordene as atividades, representantes dos grandes fornecedores de tecnologia, como IBM e Red Hat, também fazem parte do JCP. Deste modo, eles podem sugerir melhorias que afetem seus produtos relacionados à plataforma.

Como o processo envolvido na publicação de uma JSR envolve muita formalidade, foi criado um outro documento, chamado JEP (JDK Enhancement Proposal), para que os desenvolvedores da plataforma consigam fazer alterações no JDK com mais agilidade. Este documento não passa pelo JCP, no entanto, quando alguma alteração envolve também mudanças em interfaces públicas da plataforma, uma JSR deve ser aberta em paralelo, para que o assunto seja discutido.

Compatibilidade

Um dos fatores mais importantes para os desenvolvedores da plataforma Java é manter a compatibilidade entre as versões. Assim, para garantir que aplicações que não foram construídas em módulos sejam compiladas e executadas no Java 9, foi criado um mecanismo chamado “Módulo Sem Nome”. Nessa entidade fictícia são colocadas todas as classes do classpath que não estão segregadas em módulos. Na prática, é como se o desenvolvedor tivesse criado um módulo que exportasse todos os seus pacotes e dependesse de todos os outros módulos. Mesmo sendo possível compilar e executar as aplicações dessa forma, os desenvolvedores são encorajados a portar suas aplicações para que utilizem a funcionalidade de módulos em sua totalidade e, assim, tenham acesso a todos os seus benefícios.

Mas nem tudo são flores: os arquitetos decidiram aliar à introdução dos módulos uma reestruturação das APIs do JDK que ficariam visíveis aos desenvolvedores.

Embora isso torne as aplicações mais seguras, a decisão de esconder uma classe em especial, `sun.misc.Unsafe`, tem gerado muita discussão. Apesar de ela ser uma classe interna, que não deveria ser utilizada pelos desenvolvedores, muitas aplicações dependem dela para ter um nível maior de interação com o sistema operacional e assim conseguir implementar algumas melhorias de desempenho e correções de bugs. Como forma de amenizar a insatisfação, foi criado um flag que, quando passado para a JVM, permitiria a utilização dessa classe. Porém, muitos desenvolvedores alegam que o flag seria necessário em tantos casos que começaria a ser habilitado para a maioria das aplicações, perdendo assim o seu objetivo, que é restringir o uso da classe.

Outra questão que pode trazer problemas para os desenvolvedores é a reestruturação dos diretórios do JRE. Embora não seja uma prática recomendada, vários desenvolvedores confiavam no fato de que a estrutura seria sempre a mesma e por isso temos situações como aplicações que apontam diretamente para arquivos como `rt.jar` e `tools.jar`, que não estão presentes na nova estrutura.

Deficiências

Uma questão que tem gerado bastante discussão na comunidade é a ausência de um mecanismo nativo para o controle de versões das dependências. Ainda que a especificação da JSR 367 deixe claro que o controle de versões está fora do escopo do projeto e deva ser deixado para as ferramentas de build como o Maven, algumas pessoas da comunidade dizem que esse é um problema que deve ser adicionado ao escopo. Sem um controle de versões adequado, o JAR Hell que enfrentamos atualmente poderia se tornar um Module Hell, ou seja, só mudaríamos o problema de lugar.

Uma solução que já foi implementada no JDK para tentar evitar o Module Hell é validar se existe mais de um módulo com versões diferentes no `modulepath`, como veremos na parte prática do artigo. Essa abordagem, contudo, parece ser muito radical e pode trazer problemas para aplicações que tenham dependências transitivas com versões diferentes. Uma solução sugerida é colocar um número de versão no nome do módulo, tornando-o único. No entanto, mesmo que essa seja uma solução possível, não teríamos uma padronização nas nomenclaturas, o que pode trazer mais confusão. Diante do exposto, esse é um assunto que ainda será muito debatido até termos a versão final do Java 9.

Obtendo o JDK com o Projeto Jigsaw

Até setembro, toda a discussão sobre módulos era baseada sómente em exemplos teóricos. A partir de então, foram liberadas as primeiras versões do JDK 9 com o Projeto Jigsaw. Contudo, é importante ressaltar que o código-fonte do projeto ainda não foi integrado à linha principal do desenvolvimento da plataforma. Deste modo, para conseguir executar os exemplos do artigo, é necessário baixar uma versão específica do JDK, como pode ser visto na **Figura 2**.

Após o download e a descompactação do JDK, é necessário ajustar as variáveis de ambiente `JAVA_HOME` e `PATH` para que apontem para a nova versão.

Em uma máquina Linux, o procedimento a ser feito é demonstrado na **Listagem 1**. As variáveis são as mesmas em máquinas que estejam executando o Windows.

Ainda que recentemente tenham sido lançadas versões do Eclipse e NetBeans com um suporte básico ao Java 9, todos os passos do exemplo serão executados via linha de comando. Contudo, você pode também configurar sua IDE de preferência e tentar executá-los.

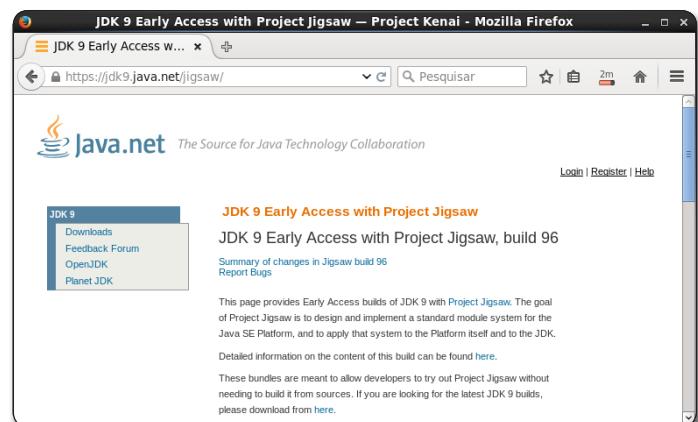


Figura 2. Página de download da JDK 9 com o Projeto Jigsaw

Listagem 1. Exemplo de instalação do JDK 9 em um servidor Linux.

```
$ sudo mkdir -p /usr/java  
$ cd /usr/java  
$ sudo cp ~/Downloads/jigsaw-jdk-9-ea+96_linux-x86_bin.tar.gz.  
$ sudo tar zxf jigsaw-jdk-9-ea+96_linux-x86_bin.tar.gz  
$ export JAVA_HOME=/usr/java/jdk-9  
$ export PATH=$JAVA_HOME/bin:$PATH  
$ java -version  
java version "9-ea"  
Java(TM) SE Runtime Environment (build 9-ea+96-jigsaw-nightly-h4110-  
20151218)  
Java HotSpot(TM) Client VM (build 9-ea+96-jigsaw-nightly-h4110-20151218,  
mixed mode)
```

Nota

Listando o diretório do JDK é possível perceber que não existe mais um diretório chamado `jre`. A partir do Java 9, o JDK e o JRE têm a mesma estrutura. O JDK é apenas uma imagem de JRE contendo, além dos módulos necessários em tempo de execução, os módulos utilizados em tempo de desenvolvimento.

Criando uma aplicação modular

Para ilustrar as funcionalidades do Projeto Jigsaw, criaremos uma aplicação exemplo composta por dois módulos: um módulo para a interface com o usuário (para tornar o exemplo mais simples, a aplicação não terá interface gráfica); e um módulo para o serviço, que fará um simples cálculo de adição de números inteiros. Ao final, criaremos um terceiro módulo para que seja possível visualizar com mais clareza as restrições de acesso entre classes de módulos diferentes.

Projeto Jigsaw: Desenvolvimento modularizado no Java 9

A estrutura de diretórios, apresentada na **Figura 3**, é descrita a seguir:

- **app**: nesse diretório será gravada uma imagem customizada de JRE, contendo apenas os módulos necessários à execução da aplicação de exemplo;
- **mlib**: diretório onde serão gravados os arquivos JAR gerados a partir dos módulos;
- **mods**: será utilizado para a gravação dos arquivos class de cada módulo; e
- **src**: local onde serão armazenados os arquivos de código-fonte.

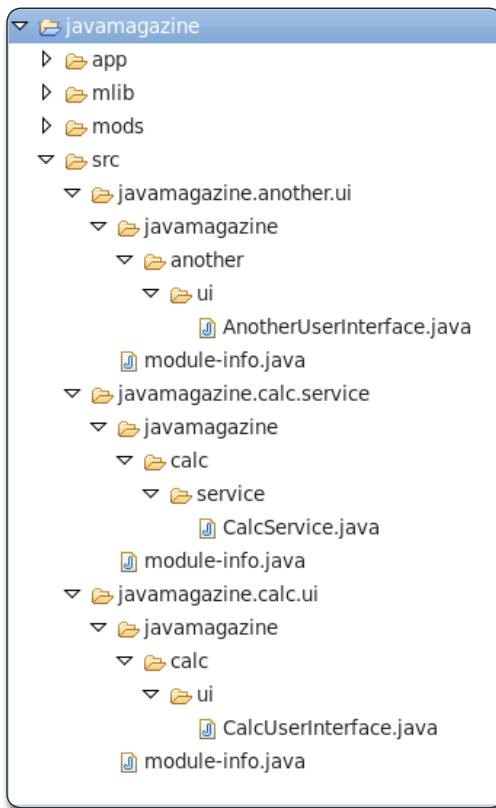


Figura 3. Estrutura de diretórios da aplicação de exemplo

Criando o primeiro módulo

Primeiramente, faremos a declaração do módulo **javamagazine.calc.ui**, como demonstrado na **Listagem 2**. Neste momento, saiba que não há nenhuma restrição para a nomenclatura, mas, por convenção, o nome do módulo geralmente é composto pela parte comum entre os nomes dos pacotes que estão presentes nele.

A declaração do módulo é feita em um arquivo chamado *module-info.java*, localizado na raiz do mesmo. Quando compilado, esse arquivo também é transformado em um arquivo .class, exatamente como uma classe.

A **Listagem 3** mostra a implementação da classe **CalcUserInterface**. Essa é a classe principal da nossa aplicação de exemplo e será utilizada como o ponto de entrada, através de seu método **main()**.

Nota

Após muita discussão, percebeu-se que fazer essa declaração em um arquivo Java comum tornaria mais fácil a manipulação do arquivo pelas IDEs e também a compilação e a execução, já que não seriam necessários muitos tratamentos especiais para ele.

Listagem 2. Declaração do módulo javamagazine.calc.ui em seu arquivo module-info.java.

```
module javamagazine.calc.ui { }
```

Listagem 3. Implementação da classe CalcUserInterface.

```
package javamagazine.calc.ui;
public class CalcUserInterface {
    public static void main(String[] args) {
        System.out.println(8 + 9);
    }
}
```

Para que fique evidente o ganho que teremos adiante ao modularizar a aplicação, neste momento implementaremos todo o escopo do exemplo em apenas um módulo. Sendo assim, o cálculo de adição será feito na própria classe (**CalcUserInterface**).

Terminada a codificação do primeiro módulo, precisamos fazer a compilação do mesmo, conforme a **Listagem 4**. Nesse código, criaremos o diretório onde serão gravados os arquivos .class e, então, executaremos o compilador (*javac*) para a geração destes. Note que o compilador é chamado com o parâmetro *-d*, indicando o diretório de saída da compilação, e os nomes dos arquivos Java, separados por espaço. Caso os comandos sejam executados sem erros, será possível verificar, através do *find*, as classes geradas no diretório *mods/javamagazine.calc.ui*.

Listagem 4. Compilação das classes do primeiro módulo.

```
$ mkdir mods/javamagazine.calc.ui
$ javac -d mods/javamagazine.calc.ui src/javamagazine.calc.ui/module-info.java
src/javamagazine.calc.ui/javamagazine/calc/ui/CalcUserInterface.java
$ find mods/javamagazine.calc.ui/
mods/javamagazine.calc.ui/
mods/javamagazine.calc.ui/javamagazine
mods/javamagazine.calc.ui/javamagazine/calc
mods/javamagazine.calc.ui/javamagazine/calc/ui
mods/javamagazine.calc.ui/javamagazine/calc/ui/CalcUserInterface.class
mods/javamagazine.calc.ui/module-info.class
```

Com isso já é possível executar o nosso primeiro módulo. Observe, no código a seguir, que são passados como parâmetro para o comando *java* a localização dos módulos (através do parâmetro *-modulepath*) e o módulo que deve ser executado (através do parâmetro *-m*):

```
$ java -modulepath mods -m javamagazine.calc.ui
/javamagazine.calc.ui.CalcUserInterface
```

O valor para o parâmetro `-m` deve ser composto pelo nome do módulo, uma barra e o nome da classe principal do módulo, local onde foi implementado o método `main()`. Caso a execução seja bem-sucedida, será possível ver na tela a saída do programa.

Nota

No parâmetro `-m` deve ser utilizada uma barra normal (não invertida). Esse padrão deve ser mantido mesmo que você esteja utilizando o Windows, onde normalmente são especificadas barras invertidas para indicar o caminho de um arquivo.

Criando o segundo módulo

Na **Listagem 5** é feita a declaração do segundo módulo do exemplo: `javamagazine.calc.service`. Como diferencial, nessa listagem é possível perceber a presença da cláusula `exports`. Deve haver uma declaração desse tipo para cada pacote do módulo que deva ficar acessível pelos módulos que dependam desse. No exemplo, utilizamos também a cláusula opcional `to`, para indicar que apenas o módulo `javamagazine.calc.ui` poderá utilizar as classes que foram exportadas.

O código da classe `CalcService`, que implementa o serviço de cálculo de adição, aparece na **Listagem 6**. Para mantermos a simplicidade, o serviço é representado apenas por um método estático, sem interfaces. Posteriormente, adicionaremos a chamada a esse serviço a partir do nosso primeiro módulo.

Listagem 5. Declaração do módulo `javamagazine.calc.service` em seu arquivo `module-info.java`.

```
module javamagazine.calc.service {  
    exports javamagazine.calc.service to javamagazine.calc.ui;  
}
```

Listagem 6. Implementação da classe `CalcService`.

```
package javamagazine.calc.service;  
public class CalcService {  
    public static Integer add(Integer a, Integer b) {  
        return a + b;  
    }  
}
```

Finalizada a implementação do segundo módulo, deve ser realizado para ele o mesmo procedimento efetuado para o primeiro, como pode ser visto na **Listagem 7**. Sendo assim, será criado um diretório para a saída da compilação de suas classes e, então, será feita a compilação. No momento da chamada ao compilador, é necessário informar o parâmetro `-modulepath`, para que o módulo `javamagazine.calc.ui`, referenciado na cláusula `to`, seja encontrado. Mais uma vez, através do comando `find`, podemos ver a lista de arquivos gerados.

Unindo os dois módulos

É uma boa prática separar o código da camada de apresentação do código de eventuais serviços com regras de negócio que sejam referenciados pela aplicação.

Em outras palavras, os desenvolvedores devem sempre ter como objetivo implementar classes que tenham alta coesão e baixo acoplamento. A alta coesão é obtida a partir do momento que cada classe tem uma única responsabilidade na aplicação e o baixo acoplamento é possível através do uso de interfaces, o que possibilita a troca da implementação concreta a ser utilizada em tempo de execução. No exemplo do artigo, criamos duas classes (uma em cada módulo) com responsabilidades diferentes, aumentando assim a coesão. No entanto, para não aumentar a complexidade do cenário proposto, não utilizaremos interfaces para o serviço, ou seja, as classes ainda terão um alto nível de acoplamento, o que não é recomendado em sistemas reais.

Para que seja possível visualizar a relação entre os módulos, teremos que alterar a classe `CalcUserInterface`, implementada no primeiro módulo. O primeiro passo é substituir a linha que faz o cálculo na própria classe pela chamada ao serviço implementado na classe `CalcService`. E para fazer uso do serviço, é necessário também adicionar a cláusula `import`, como pode ser visto na **Listagem 8**. Note que agora passamos valores diferentes (5 e 7) na chamada ao método de adição. Desta forma, obtendo 12 como resultado teremos a certeza de que a saída está sendo realmente gerada pelo serviço.

Listagem 7. Compilação das classes do segundo módulo.

```
$ mkdir mods/javamagazine.calc.service  
$ javac -modulepath mods -d mods/javamagazine.calc.service src/javamagazine.calc.service/module-info.java src/javamagazine.calc.service/javamagazine.calc.service/CalcService.java  
$ find mods/javamagazine.calc.service/  
mods/javamagazine.calc.service/  
mods/javamagazine.calc.service/javamagazine  
mods/javamagazine.calc.service/javamagazine/calc  
mods/javamagazine.calc.service/javamagazine/calc/service  
mods/javamagazine.calc.service/javamagazine/calc/service/CalcService.class  
mods/javamagazine.calc.service/module-info.class
```

Listagem 8. Classe `CalcUserInterface` alterada para chamar o serviço implementado no segundo módulo.

```
package javamagazine.calc.ui;  
import javamagazine.calc.service.CalcService;  
public class CalcUserInterface {  
    public static void main(String[] args) {  
        System.out.println(CalcService.add(5, 7));  
    }  
}
```

Feitas as mudanças, executaremos novamente a compilação das classes do módulo `java.magazine.ui`, através dos comandos ilustrados na **Listagem 9**. Desta vez, no entanto, o resultado será um erro de compilação. Isso acontece porque o compilador não encontrou o pacote `javamagazine.calc.service` e, por consequência, a classe `CalcService`.

Para que o compilador consiga encontrar a classe, é necessário alterar o arquivo `module-info.java` do módulo `javamagazine.calc.ui`, especificando que ele requer o módulo `javamagazine.calc.service`. Essa informação é adicionada através da cláusula `requires`. O arquivo alterado pode ser visualizado na **Listagem 10**.

Projeto Jigsaw: Desenvolvimento modularizado no Java 9

Listagem 9. Erro na compilação do primeiro módulo.

```
$ javac -modulepath mods -d mods/javamagazine.calc.ui  
src/javamagazine.calc.ui/module-info.java src/javamagazine.calc.ui/javamagazine/  
calc/ui  
/CalcUserInterface.java  
src/javamagazine.calc.ui/javamagazine/calc/ui/CalcUserInterface.java:2:  
error: package javamagazine.calc.service does not exist  
import javamagazine.calc.service.CalcService;  
^  
src/javamagazine.calc.ui/javamagazine/calc/ui/CalcUserInterface.java:5:  
error: cannot find symbol  
System.out.println(CalcService.add(5, 7));  
^  
symbol: variable CalcService  
location: class CalcUserInterface  
2 errors
```

Listagem 10. Inclusão da dependência javamagazine.calc.service no módulo javamagazine.calc.ui.

```
module javamagazine.calc.ui {  
    requires javamagazine.calc.service;  
}
```

Feito isso, a compilação será executada com sucesso. A aplicação pode, então, ser executada para visualizarmos a saída com seu novo comportamento, que indica que a versão que está sendo executada é realmente aquela que faz referência ao serviço disponibilizado pelo outro módulo (vide [Listagem 11](#)).

Pronto! Acabamos de criar o nosso primeiro módulo com dependências e conseguimos executá-lo sem erros. Nesse exemplo, as declarações **exports** e **requires** aparecem em suas formas mais simples, mas elas podem conter parâmetros adicionais, para utilização em cenários mais complexos, como veremos a seguir:

Criando o terceiro módulo

Para ilustrar exemplos mais complexos das cláusulas do arquivo *module-info.java*, vamos criar um terceiro módulo, chamado **javamagazine.another.ui** (veja a [Listagem 12](#)). Como este módulo também fará referência ao serviço disponibilizado pelo segundo, o código-fonte será basicamente o mesmo do primeiro, **javamagazine.calc.ui**. Na [Listagem 13](#) é apresentada a implementação da classe de interface com o usuário, chamada de **AnotherUser-Interface**.

Como a cláusula **exports** do módulo **javamagazine.calc.service** indica através da cláusula **to** que apenas o módulo **javamagazine.calc.ui** terá acesso às classes exportadas, ao tentar executar a compilação desse novo módulo será obtido um erro, que informa que o pacote **javamagazine.calc.service** não foi encontrado ([Listagem 14](#)).

Embora esse pacote seja exportado, ele é exportado apenas para o módulo **javamagazine.calc.ui**, e como estamos compilando o módulo **javamagazine.another.ui**, é como se o pacote não existisse para o compilador.

Ainda que de pouco uso para aplicações mais simples, essa possibilidade é bastante útil para os desenvolvedores da plataforma e para os desenvolvedores de bibliotecas. Nesse tipo de aplicação normalmente há classes utilitárias internas que devem ser referenciadas somente por classes que estejam no mesmo projeto.

Listagem 11. Compilação das classes do primeiro módulo com referência ao segundo módulo.

```
$ javac -modulepath mods -d mods/javamagazine.calc.ui src/javamagazine.calc.  
ui/module-info.java src/javamagazine.calc.ui/javamagazine/calc/ui/  
CalcUserInterface.java  
$ java -modulepath mods -m javamagazine.calc.ui/javamagazine.calc.  
ui.CalcUserInterface  
12
```

Listagem 12. Declaração do módulo **javamagazine.another.ui** em seu arquivo *module-info.java*.

```
module javamagazine.another.ui {  
    requires javamagazine.calc.service;  
}
```

Listagem 13. Implementação da classe **AnotherUserInterface**.

```
package javamagazine.another.ui;  
import javamagazine.calc.service.CalcService;  
public class AnotherUserInterface {  
    public static void main(String[] args) {  
        System.out.println(CalcService.add(5, 7));  
    }  
}
```

Listagem 14. Erro na compilação do terceiro módulo.

```
$ mkdir mods/javamagazine.another.ui  
$ javac -modulepath mods -d mods/javamagazine.another.ui src/javamagazi-  
ne.another.ui/module-info.java src/javamagazine.another.ui/javamagazine/  
another/ui/AnotherUserInterface.java  
src/javamagazine.another.ui/javamagazine/another/ui/AnotherUserInterface.  
java:2: error: package javamagazine.calc.service does not exist  
import javamagazine.calc.service.CalcService;  
^  
src/javamagazine.another.ui/javamagazine/another/ui/AnotherUserInterface.  
java:5: error: cannot find symbol  
System.out.println(CalcService.add(5, 7));  
^  
symbol: variable CalcService  
location: class AnotherUserInterface  
2 errors
```

É possível ainda utilizar outras declarações no arquivo *module-info.java*. Por exemplo, podemos citar a cláusula **uses**, na qual são indicados os serviços que o módulo utiliza. Os módulos que provêm esses serviços, por sua vez, devem explicitar isso através das cláusulas **provides** (qual o serviço provido) e **with** (qual a classe que implementa o serviço). Essa opção permite que as aplicações tenham um nível menor de acoplamento, já que não há no código uma referência explícita à classe que implementa o serviço.

Um exemplo típico é o módulo **java.sql**, que utiliza o serviço **java.sql.Driver**. Para que sejam instanciadas corretamente, as implementações dos drivers para cada SGBD devem declarar com que classe provêm esse serviço. Assim, o módulo para MySQL, por exemplo, poderia ter em seu arquivo *module-info.java* a seguinte declaração: **provides** **java.sql.Driver** **with** **com.mysql.jdbc.Driver**.

Trabalhando com JARs de módulo

Um arquivo JAR de módulo contém, além de classes, o descritor *module-info.class*. Esses arquivos JAR podem ser criados de maneira

muito semelhante à criação de um JAR comum, utilizando-se o utilitário *jar* com o parâmetro *--create*, como pode ser visto na **Listagem 15**. O utilitário pode receber ainda parâmetros adicionais, como *--module-version*, caso queira gerar um módulo versionado, e *--main-class*, caso queira indicar a classe principal do módulo que terá seu método *main()* invocado na execução.

Para rodar um módulo contido em um JAR, o programa *java* deve ser chamado informando-se no parâmetro *-modulepath* a localização do arquivo a ser executado e dos demais módulos necessários à execução e, através do parâmetro *-m*, o nome do módulo que se deseja executar.

Como parte das restrições implementadas para se evitar o JAR Hell, ou Module Hell, podemos ver na **Listagem 16** o erro que é apresentado quando se tenta executar uma aplicação e duas versões diferentes (1.0 e 2.0) do mesmo módulo estão presentes. Com o classpath em sua forma atual, seria utilizada a primeira versão encontrada, o que, como já foi dito, pode provocar um erro ou um comportamento diferente daquele esperado pelo desenvolvedor.

Listagem 15. Criação e execução de um JAR de módulo.

```
$ jar --create --file=mlib/javamagazine.calc.service@1.0.jar --module-version=1.0 -C mods/javamagazine.calc.service .
$ jar --create --file=mlib/javamagazine.calc.ui.jar --main-class=javamagazine.calc.ui.CalcUserInterface -C mods/javamagazine.calc.ui .
$ find mlib
mlib
mlib/javamagazine.calc.service@1.0.jar
mlib/javamagazine.calc.ui.jar
$ java -modulepath mlib -m javamagazine.calc.ui
12
```

Listagem 16. Erro na Geração do JAR de módulo na presença de duas versões do mesmo módulo.

```
$ jar --create --file=mlib/javamagazine.calc.service@2.0.jar --module-version=2.0 -C mods/javamagazine.calc.service .
$ find mlib
mlib
mlib/javamagazine.calc.service@1.0.jar
mlib/javamagazine.calc.ui.jar
mlib/javamagazine.calc.service@2.0.jar
$ java -modulepath mlib -m javamagazine.calc.ui
Error occurred during initialization of VM
java.lang.module.ResolutionException: Two versions of module javamagazine.calc.service found in mlib
    at java.lang.module.Resolver.findWithBeforeFinder(java.base@9-ea/Resolver.java:806)
    at java.lang.module.Resolver.resolve(java.base@9-ea/Resolver.java:283)
    at java.lang.module.Configuration.resolve(java.base@9-ea/Configuration.java:246)
    at jdk.internal.module.ModuleBootstrap.boot(java.base@9-ea/ModuleBootstrap.java:188)
    at java.lang.System.initPhase2(java.base@9-ea/System.java:1911)
```

Embora nesse caso o erro apresentado seja de tempo de execução, se tivéssemos tentado compilar novamente o primeiro módulo, um erro muito parecido seria obtido, pois o comportamento do JRE e do JDK entre as fases, principalmente a de compilação e a de execução, deve ser o mais parecido possível. Isso deixa clara a

preocupação que o Projeto Jigsaw tem com aquilo que é chamado de “Fidelidade entre todas as fases”. Seguindo esse princípio, mais erros passam a ser identificados durante a compilação, reduzindo assim os erros em tempo de execução.

Gerando uma imagem customizada de JRE

Podemos criar uma imagem de JRE customizada, contendo apenas os módulos básicos e aqueles necessários à execução do módulo que foi desenvolvido. Dessa forma, o JRE fica muito menor, tendo sua distribuição facilitada, principalmente para dispositivos pequenos. Para criar essa imagem, faremos uso do utilitário *jlink*, como pode ser visto na **Listagem 17**.

Com o intuito de demonstrar o poder da ferramenta, informaremos alguns parâmetros adicionais que fazem com que arquivos desnecessários não sejam incluídos na imagem e que aqueles que sejam, fiquem comprimidos.

Em nossa primeira tentativa de criar uma imagem de JRE, no entanto, ocorre um erro. Esse erro é exibido por conta do arquivo que criamos para demonstrar como funciona a prevenção ao *Module Hell* (*javamagazine.calc.service@2.0.jar*). Sendo assim, após removermos o arquivo, conseguimos executar o *jlink* com sucesso e também o JRE gerado por ele.

Listagem 17. Geração de imagem customizada de JRE.

```
$ jlink --modulepath $JAVA_HOME/jmods:mlib --addmods javamagazine.calc.ui --output app/java --exclude-files *.diz --strip-java-debug on --compress-resources on --compress-resources-level 2
Error: Two versions of module javamagazine.calc.service found in mlib
Usage: jlink <options> --modulepath <modulepath> --addmods <mods> --output <path>
use --help for a list of possible options
$ rm -rf mlib/javamagazine.calc.service@2.0.jar
$ jlink --modulepath $JAVA_HOME/jmods:mlib --addmods javamagazine.calc.ui --output app/java --exclude-files *.diz --strip-java-debug on --compress-resources on --compress-resources-level 2
$ app/java/bin/java -m javamagazine.calc.ui
12
$ du -ch app | grep total
35M total
```

Note que a imagem gerada no exemplo ficou com cerca de 35MB. Obviamente ainda não se trata de uma distribuição pequena, levando-se em consideração o tamanho e a funcionalidade do exemplo. Porém, quando comparado aos 201MB de um JRE completo, que pode ser visto na **Listagem 18**, o tamanho pode ser considerado um avanço enorme e já abre um grande leque de possibilidades para os desenvolvedores.

A geração de imagens menores é possível graças à aplicação da estrutura de módulos ao próprio JRE. Nessa reestruturação, foi criado um módulo chamado *java.base* (vide **Listagem 19**) que não tem nenhuma dependência e do qual todo módulo criado depende implicitamente. Ele atua mais ou menos como a classe *Object*, quando falamos de classes, e exporta os pacotes que são considerados fundamentais para a plataforma, como *java.io* e

Projeto Jigsaw: Desenvolvimento modularizado no Java 9

`java.lang`, além do pacote que contém o sistema de módulos: `java.lang.module`.

Listagem 18. Demonstração do tamanho de um JRE 9 completo.

```
$ cd /usr/java  
$ sudo cp ~/Downloads/jre-9-ea+96_linux-x86_bin.tar.gz.  
$ sudo tar zxf jre-9-ea+96_linux-x86_bin.tar.gz  
$ du -ch jre-9 | grep total  
201M  total
```

Listagem 19. Trecho do arquivo module-info.java do módulo java.base.

```
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;  
    ...  
}
```

A qualidade do trabalho realizado por Mark Reinhold e os demais arquitetos da plataforma é incontestável. Com o conteúdo exposto até aqui, é evidente que o Projeto Jigsaw trará uma série de ganhos aos desenvolvedores Java e usuários de suas aplicações. Os problemas que são tratados pelo projeto têm presença constante no cotidiano e, embora existam soluções para alguns deles, nada melhor do que uma solução nativa e definitiva.

Contudo, como o projeto envolve uma grande refatoração no JDK, é importante que os desenvolvedores estudem as novas funcionalidades e tentem compilar e executar suas aplicações no JDK 9 com o Projeto Jigsaw, reportando qualquer erro que encontrem e não esteja relacionado a alguma incompatibilidade esperada.

Visto que a primeira versão do JDK foi lançada recentemente, provavelmente esteja repleta de cenários não cobertos. Neste momento, portanto, tudo o que os desenvolvedores da plataforma mais querem e precisam é de feedback sobre esses cenários. Um excelente lugar para começar é a lista de discussão, onde é possível assistir ou participar dos debates. As conversas têm sempre um ótimo nível.

Autor



Cristiano Mariano de Oliveira

cristianomariano@gmail.com

é Pós-Graduando em Engenharia de Software e Graduado em Ciência da Computação pela Universidade Paulista. Trabalha com Java há 10 anos. Possui as certificações SCJP, SCWCD e SCBCD.



Links:

Site oficial do Projeto Jigsaw.

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

Download do JDK 9 com o Projeto Jigsaw.

<https://jdk9.java.net/jigsaw/>

Lista de discussão sobre o Projeto Jigsaw.

<http://mail.openjdk.java.net/mailman/listinfo/jigsaw-dev>

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

Aplicando boas práticas em todo o processo de desenvolvimento

Aprenda neste artigo como tornar o processo de criação de sistemas Java mais eficiente

A rotina básica de desenvolvimento de software é simples: baixar o código de um repositório, codificar, validar o que foi criado e, se tudo estiver como esperado, integrar os novos artefatos ao repositório e testar as novas funcionalidades. Esta rotina, no entanto, pode ficar mais complexa ao ter novos passos adicionados, ser compartilhada através da programação em pares, não fazer uso de controle versão, entre outras possibilidades. Além disso, ela também pode ser simplificada, principalmente em cenários onde não há trabalho em equipe. Independentemente disso, isto é, em qualquer um desses casos, o que realmente irá torná-la impraticável ou ineficiente é a utilização de más práticas de desenvolvimento. Para a finalidade deste artigo, expor boas práticas nesse contexto, a rotina descrita inicialmente será considerada como o padrão adotado no dia a dia de uma empresa de TI.

A partir disso, será possível identificar, em cada uma das fases, problemas ou dificuldades causadas pelo uso de práticas indevidas, que podem levar ao atraso de entregas ou até mesmo do projeto como um todo. Descrever brevemente cada uma dessas fases e listar alguns dos problemas que ocorrem durante cada uma será o objetivo da primeira parte deste artigo. Posteriormente, serão apresentadas práticas para tornar o processo de implementação mais eficiente e eficaz, focando na melhoria da qualidade do código fonte, na metodologia e práticas de testes utilizadas.

Fase de aquisição ou atualização do código

A primeira fase da rotina básica de desenvolvimento mencionada é a de aquisição do código. Esta fase ocorre não apenas no início do desenvolvimento, mas a cada vez

Fique por dentro

Este artigo será útil para desenvolvedores, líderes técnicos e gestores de equipes de criação de software interessados em tornar o processo de implementação mais eficiente e eficaz e melhorar a qualidade do código fonte de suas entregas. Para isso, o artigo sugere boas práticas de desenvolvimento e de padrões de projeto que, se bem aplicados, favorecem a elaboração de um código limpo, conciso e de fácil manutenção e alteração.

O foco deste será na linguagem de programação Java, mas muitos dos itens abordados podem ser levados em consideração na construção de qualquer software, em ambientes diversos. Contudo, não serão abordadas boas práticas de design de interface de usuário ou de geração de assets para criação de telas visto que este é um assunto que, por si só, merece um artigo à parte.

que é necessário atualizar o código local. Então, se a atualização toma um minuto do programador e este atualiza o código 15 vezes ao dia, somam-se 15 minutos diários. Este tempo aumenta para uma hora e 15 minutos em uma semana e resulta em aproximadamente cinco horas em um mês de trabalho.

Se este mesmo cálculo, do tempo despendido em tarefas que podem ser evitadas, for aplicado em toda a rotina de desenvolvimento, é possível verificar quanto tempo um time desperdiça em tarefas que não agregam valor ao produto a ser entregue.

A fase de aquisição de código é prejudicada pelo uso de más práticas de várias formas. Uma das atitudes a ser evitada é a utilização de nomenclatura inadequada no repositório ou nos branches. Elas não causam grandes impactos negativos no cronograma do projeto, mas dificultam a identificação do código a ser evoluído. Outra atitude a ser evitada é a adição de arquivos desnecessários no sistema de controle de versão.



Estes arquivos podem deixar o repositório grande e confuso e tornar o processo de atualização do código demorado.

A aplicação de boas práticas nesta fase resulta na diminuição do tempo gasto por cada um dos membros do time para integrar e atualizar o código com o armazenado no repositório. Atitudes simples como não adicionar arquivos gerados durante o processo de compilação e utilizar um sistema de controle de versão são suficientes para tornar a fase de aquisição de código muito mais produtiva.

Fase de codificação

A fase de codificação é aquela que oferece as melhores oportunidades para aplicação de boas práticas. Na rotina usada de exemplo neste artigo, esta etapa inclui não somente a codificação em si, mas também a modelagem e a arquitetura do aplicativo a ser construído.

A adoção de práticas ruins durante a codificação geralmente prejudica o trabalho em equipe, levando à criação de produtos de baixa qualidade ou que não atendem às necessidades dos usuários ou clientes, como aplicativos que consomem muita bateria, fazem uso do processador em excesso e/ou que apresentam muitos problemas durante a execução.

O alto consumo de processador pode ser relacionado à duplicidade de informações armazenadas, chamadas inoportunas e inserções, seleções ou atualizações desnecessárias a um banco de dados, por exemplo. O alto consumo de bateria, por sua vez, pode ser consequência do uso excessivo do processador, como também do acesso a informações remotas através de rede de dados e Wi-Fi, ou mesmo do uso abusivo do GPS e de outros sensores do aparelho.

Usualmente, o processo de codificação é executado de maneira individual ou em duplas, com o objetivo de evoluir um código compartilhado por uma equipe. A falta de critérios pré-estabelecidos de formatação do código, o desleixo, a falta de atenção ou a ausência do sentido de coletividade do código e do empenho por parte de membros do time são alguns dos fatores que prejudicam o trabalho.

Portanto, saiba que aplicar boas práticas durante a codificação resulta em um código bem estruturado, limpo, conciso e de fácil compreensão, e esta melhoria da qualidade do código viabiliza o aumento da produtividade do time em todas as fases de qualquer processo de software.

Fase de validação do código desenvolvido

A terceira fase da rotina de desenvolvimento em análise é a validação do código implementado. Diferentemente dos conhecidos testes de software, esta fase não é necessariamente executada por testadores, mas pelo próprio programador com o intuito de avaliar se sua implementação está gerando os resultados esperados, antes de integrar o novo código com o gerado pelo restante da equipe.

Nesta fase, a utilização de práticas ruins normalmente não traz consequências imediatas e os problemas geralmente “aparecem”

após a integração do código ou em releases subsequentes. Na verdade, as consequências são imprevisíveis: o código integrado pode cumprir com seu propósito, não causar bugs de regressão e nem novos bugs; como também pode causar problemas de compilação do código do repositório utilizado por toda a equipe.

A aplicação de boas práticas de desenvolvimento nesta etapa diminui a quantidade de problemas detectados principalmente em releases subsequentes, pois a probabilidade de adição de código que cause os chamados bugs de regressão ou novos bugs é reduzida. Dentre as boas práticas, pode-se citar a programação em pares e a consulta aos casos de uso relacionados às funcionalidades implementadas.

Fase de integração do novo código

A fase de integração de código é a quarta da rotina de desenvolvimento usada como exemplo neste artigo. Deste modo, esta etapa é diretamente impactada pela aplicação de decisões ruins nas fases anteriores. Pensando nisso, muitas das boas práticas sugeridas para a codificação, por exemplo, visam agilizar e não tornar a fase de integração do código um transtorno para os participantes do projeto.

Da mesma forma, a não utilização de boas práticas – especificamente na fase de integração – gera impactos negativos nas outras etapas da rotina. A integração de código indevido pode causar transtornos na fase de aquisição do código por outros membros da equipe, como descrito anteriormente, pode levar à evolução de código fonte sem padrão e dificultar as fases de codificação e testes, pode complicar o processo de integração do código de outros participantes do projeto, assim como pode levar ao surgimento de bugs de regressão e à perda do histórico da evolução do código, que têm outras consequências negativas para o processo de construção do software.

O uso de boas práticas neste momento é essencial para a qualidade do produto, assim como decisões equivocadas podem inutilizar boas ações aplicadas anteriormente. Dentre as tarefas a considerar, estão a revisão de código e o uso de sistemas de controle de versão.

Fase de testes

A fase de testes é a última da rotina definida como exemplo. Entretanto, é importante deixar claro que este artigo não defende a ideia de que os testes de software devem ser executados após a codificação, muito pelo contrário. Executar os testes durante a implementação das funcionalidades é uma boa prática que deve ser considerada e que será detalhada posteriormente.

Nesta fase, os casos de teste preparados são executados com o intuito de verificar se os requisitos esperados estão implementados da maneira correta, se os fluxos navegacionais estão como previstos e se o produto funciona a contento. Em outras palavras, é nesta fase onde a qualidade do produto é validada.

Assim como nas outras fases da rotina, a aplicação de más práticas durante os testes leva à diminuição da qualidade do produto criado.

Por exemplo, a não automatização torna os ciclos de testes mais demorados e com cobertura menor, já que os testadores, geralmente, concentram seus esforços apenas nos casos de testes mais críticos e importantes, para que o cronograma do projeto não seja impactado.

O uso de boas práticas nesta fase é importante para agilizar o processo de desenvolvimento, melhorar a compreensão e a confiabilidade do código, entre outros. Como sugestões a considerar, estão a já citada automatização, o desenvolvimento orientado a testes e a realização das atividades de testes durante a codificação.

Utilizando boas práticas de desenvolvimento

O trabalho de desenvolvimento de software deve ser visto como uma atividade coletiva e colaborativa. Com base nisso, as boas práticas sugeridas neste artigo visam, dentre outras coisas, facilitar ou viabilizar a colaboração entre os participantes de um projeto de software. Por isso, o primeiro e mais importante dos passos para aumentar a eficiência e a eficácia de um time é que os membros deste tenham consciência de que estão trabalhando em conjunto, com um objetivo em comum, e que compartilham e evoluem um recurso único, o código fonte.

O segundo passo é que o time tenha as condições básicas para aplicar as boas práticas de desenvolvimento. As boas práticas tornam as equipes mais produtivas, mas pode ser preciso uma mudança de mentalidade gerencial ou organizacional para atingir melhores resultados. Uma mudança da metodologia de desenvolvimento, por exemplo, pode ser necessária, e o apoio gerencial é fundamental para que isto ocorra.

O terceiro passo é que todos os envolvidos na criação do aplicativo estejam sempre em busca de melhorar o cenário atual, independentemente de que situação seja esta. Para que seja possível identificar pontos de melhoria e adquirir o conhecimento necessário para corrigir os defeitos, é fundamental ter acesso a treinamentos, revistas e sites especializados. Além disso, é interessante que o próprio time promova eventos com o intuito de compartilhar o conhecimento entre seus membros.

Metodologias de desenvolvimento

Existem diferentes metodologias que podem ser aplicadas em processos de desenvolvimento. Dentre elas é possível destacar as metodologias tradicionais e as ágeis. As metodologias tradicionais requerem que os custos, as atividades e as entregas sejam definidas antes do início do projeto e que estes mudem minimamente no decorrer da execução do mesmo. Já as metodologias ágeis permitem e encorajam essas mudanças, com o objetivo de estar sempre guiando o desenvolvimento de um produto que melhor atenda às necessidades do cliente e dos usuários.

Alguns dos princípios das metodologias ágeis as fazem mais interessantes, quando comparadas às metodologias tradicionais, para times que buscam melhorar a qualidade dos códigos, dos produtos e aumentar a produtividade. As metodologias ágeis incentivam a programação em pares, a realização de testes durante a codificação, a participação de todo o time na estimativa do

tempo de duração do projeto, o trabalho em equipe, dentre outras atividades que levam a um ganho na qualidade do código e do produto, além de aumentar a eficiência e a eficácia do time.

A programação em pares possibilita a diminuição do tempo necessário para solucionar problemas complexos. Além disso, o código produzido pela dupla tende a ter uma qualidade próxima da ideal, pois já está sendo revisado durante a codificação. Há, também, uma redução nas exceções e erros decorrentes deste código, já que existem duas pessoas atentas aos detalhes do que está sendo produzido.

Ao envolver todo o time na estimativa do tempo de duração do projeto, as metodologias ágeis tornam as estimativas mais precisas. Como consequência, o número de horas extras, atrasos e prazos curtos, que são fatores que causam piora na qualidade do código e do produto, são minimizados.

Dentre os eventos previstos por essas metodologias durante o desenrolar do projeto estão as reuniões de retrospectiva. Nestas ocasiões o próprio time lista os problemas e propõe melhorias a serem aplicadas imediatamente. Isto proporciona uma melhoria contínua em todos os aspectos do projeto, incluindo o código fonte. Há também reuniões com todos os membros do projeto para apresentar as entregas parciais para o cliente ou seu representante. Este tipo de atividade faz com que os membros do time se sintam responsáveis pelo aplicativo e, consequentemente, pelo código fonte, pelas atividades e pelo cumprimento dos prazos.

Existem outras vantagens em se aplicar metodologias ágeis no processo de desenvolvimento, assim como há vantagens em se aplicar abordagens tradicionais. Por isso, um estudo aprofundado do tema é sugerido antes de se definir aquela que melhor se adequa à organização, ao time ou ao produto a ser construído.

Ao implantar uma metodologia é importante que não se façam alterações significativas em seus princípios e recomendações, pois elas foram criadas a partir de estudos e observações comportamentais e foram evoluindo através da experiência de uso em situações diversas.

Ademais, como já mencionado, a mudança da forma de trabalhar durante a codificação depende de apoio gerencial, pois esta pode ir de encontro aos costumes e à cultura organizacional. Ainda assim, vale ressaltar que utilizar uma metodologia é essencial para melhorar a produtividade de um time e propiciar a organização necessária para se construir um software com sucesso. Esta é, também, a principal razão da aplicação de uma metodologia ser uma excelente prática de desenvolvimento.

Boas práticas de programação

Independentemente de a organização ter definido, ou não, uma metodologia de desenvolvimento, outras boas práticas podem ser aplicadas para aumentar a produtividade da equipe e a qualidade do produto por ela criado. São ações que dependem apenas dos programadores para serem utilizadas e têm influência direta na evolução do aplicativo, pois são relacionadas ao próprio código fonte. O objetivo de se utilizar estas práticas é melhorar a legibilidade do código ou a arquitetura do mesmo.

Como já mencionado, a melhoria da compreensão do código fonte tem impacto direto nas fases de aquisição, codificação, testes e integração com o repositório. Ao adquirir um código de difícil leitura, o profissional levará um tempo muito maior para conseguir evoluí-lo ou corrigir algum problema. Além disso, a codificação e a validação do que foi programado poderão ser feitos de maneira incorreta, gerando defeitos no código ou complicando ainda mais o seu entendimento. A integração com artefatos de baixa qualidade pode levar a uma concepção equivocada e provocar remoções ou alterações desnecessárias, o que, por sua vez, pode causar novos defeitos.

Por isso, uma prática importante e que deve ser mantida como prioridade não somente na fase de codificação, mas em toda rotina de desenvolvimento, é a manutenção de um código fonte legível. Esta atitude é benéfica para o projeto e para o time, pois facilita a compreensão do resultado do trabalho de cada um.

O código fonte ideal é aquele bem estruturado, limpo, conciso, de fácil entendimento e, obviamente, que não apresenta problemas de compilação e provê os resultados esperados. Este conceito, no entanto, é relativo, pois cada membro do projeto tem sua concepção sobre como deve ser um código legível. É principalmente por este motivo que se deve incentivar o sentimento de colaboração e coletividade entre os membros do time. Lembre-se que o trabalho em equipe é essencial durante todo o projeto, inclusive na definição das práticas a serem adotadas, visto que o código será compartilhado entre todos. Além disso, é importante que todos sigam o padrão estabelecido, caso contrário a situação pode ficar ainda mais complexa.

De maneira geral, um código legível é direto, simples e fácil de compreender, não possui duplicidades, é eficiente e faz apenas o que é proposto. Para auxiliar na busca por este resultado, a seguir serão descritas boas práticas que visam a criação de um código fonte de qualidade.

Para manter o código legível, uma das primeiras sugestões é adotar apenas um idioma em tudo o que for produzido. O código deve ser escrito em um idioma que seja comprehensível por todos os envolvidos no projeto. Se este for compartilhado apenas por brasileiros, por exemplo, os comentários, os nomes das classes, das variáveis, das constantes e afins podem ser escritos em Português. Entretanto, é preciso lembrar que ao implementar um grande sistema, por exemplo, podem ser utilizadas bibliotecas e SDKs em Inglês. Nesses casos, o idioma Inglês é o recomendado. A **Listagem 1** mostra um exemplo onde há mistura dos idiomas Português e Inglês.

A indentação é um dos aspectos visuais mais importantes para facilitar ou dificultar a leitura e o entendimento do código. Primeiramente é importante decidir se serão empregados tabulações ou espaços, e depois, definir o tamanho da indentação. Normalmente, utiliza-se quatro espaços.

Ainda em relação à indentação, é importante que todo o código aninhado esteja indentado, assim como as quebras de linhas longas.

Em relação ao comprimento das linhas, é importante definir um limite. Linhas muito longas tendem a ser confusas ou, até mesmo,

não caberem na tela. Os limites normalmente utilizados são 60, 80 ou 100 caracteres. A quebra da linha pode ser feita manualmente em um ponto que não prejudique muito o entendimento do conteúdo.

A **Listagem 2** mostra um exemplo de um código onde o comprimento da linha ultrapassa o tamanho ideal (linha 1) e um código onde a prática da quebra manual da linha é aplicada para facilitar a compreensão (linhas 6, 7 e 8). Nesta mesma listagem a indentação correta é utilizada com o mesmo objetivo de melhorar a legibilidade, como no alinhamento dos elementos do if das linhas 6, 7 e 8.

Listagem 1. Código com mistura de idiomas.

```
01 class Duck {  
02     boolean isAndando = false;  
03  
04     void anda() {  
05         isAndando = true;  
06     }  
07 }
```

Listagem 2. O comprimento da linha do código e a indentação interferem na legibilidade.

```
01 if (SuperMarket.getInstance().isOpen() && !DrugStore.getInstance().isOpen() &&  
     this.isSick && this.hasMoney)  
02 {  
03     goToSuperMarket();  
04 }  
05  
06     if (SuperMarket.getInstance().isOpen() &&  
07     !DrugStore.getInstance().isOpen() &&  
08     this.isSick && this.hasMoney)  
09 {  
10     goToSuperMarket();  
11 }
```

Outra boa prática é criar nomes consistentes para as classes, variáveis e constantes. Apenas pelo nome destes elementos deve ser possível entender a razão de existirem ou o que fazem. Não há um limite sugerido para o tamanho dos nomes.

A utilização de poucos comentários também é primordial para a manutenção de um código limpo e conciso. Pode parecer contraditório, mas comentários demais poluem o código. Além disso, se for necessário adicionar comentários para explicar o objetivo de uma classe, método, variável ou qualquer outro módulo, é porque o nome deste elemento ou o código aninhado nele pode ser melhorado.

A **Listagem 3** mostra um exemplo de código onde nomes bem definidos de método e variáveis tornam desnecessário o uso de comentários, tanto para explicar a razão de existir do método quanto para compreender o seu funcionamento.

Uma atitude que colabora na manutenção de um código fonte conciso é adicionar conteúdo somente quando necessário. Adicionar código para uso futuro, mesmo quando este é bem escrito, dificulta o entendimento do conteúdo por outro profissional, que irá procurar justificativas para a existência daquele código. Este conteúdo desnecessário pode, ainda, complicar a integração com o repositório.

Aplicando boas práticas em todo o processo de desenvolvimento

Listagem 3.

Nome e conteúdo do método devem tornar comentários desnecessários.

```
01 // Method that makes the duck walk if it is stopped
02 void makeDuckWalkIfStopped()
03 {
04     // If the duck is stopped, make it walk
05     if (duck.isStopped()) {
06         duck.walk();
07     }
08 }
```

Outra forma de manter o código fonte conciso é não repetir conteúdo. Código repetido resulta em linhas desnecessárias e a correção de um defeito nestes casos deve ser feita a cada ocorrência deste código. Nestes casos, até mesmo o autor pode ter dificuldades em recordar cada correção a ser feita.

A legibilidade do código também pode ser melhorada ao se utilizar a Orientação a Objetos com sabedoria. Classes bem escritas são concisas e representam um único objeto com suas características e comportamentos. Ademais, o relacionamento entre os objetos também deve ser respeitado, evitando-se situações como a da **Listagem 4**, onde um objeto **Pato** instancia um objeto **Galinha**. O correto seria que o objeto **Celeiro** instanciasse o objeto **Galinha**.

Listagem 4.

Relação incorreta entre objetos.

```
01 class Celeiro {
02     Celeiro()
03     {
04         // outras inicializações
05         new Pato();
06     }
07 }
08
09 class Pato {
10     Pato()
11     {
12         // outras inicializações
13         new Galinha();
14     }
15 }
16
17 class Galinha {
18     Galinha()
19     {
20         // inicializar o objeto
21     }
22 }
```

Além de práticas que garantem a legibilidade do conteúdo criado, é importante evitar processamentos desnecessários para se criar produtos de qualidade. Apesar do aumento do poder de processamento e da capacidade de bateria dos computadores pessoais, dispositivos móveis e servidores, é primordial que o programador continue se preocupando com estes quesitos. A **Listagem 5** mostra um exemplo de código onde há processamento desnecessário em um loop. Note que a declaração da **String** **input** e a recuperação do índice de um de seus caracteres pode ser feita apenas uma vez: fora do loop, por exemplo.

Listagem 5.

Recuperação do índice do caractere na string pode ser feita apenas uma vez.

```
01 char[] alphabet = {'a', 'b', 'c', 'd', 'e', 'f', ... 'w', 'x', 'y', 'z'};
02 for(int index = 0; index < alphabet.length; ++index)
03 {
04     string input = "bus";
05     int indexOfU = input.indexOf('u');
06     if (input.charAt(indexOfU) == alphabet[index])
07     {
08         return true;
09     }
10 }
```

Nota

Existem boas práticas que se aplicam a plataformas específicas. Na plataforma Android, por exemplo, é recomendado executar o mínimo possível de processamentos em Activities, pois estas são montadas e desfeitas a cada vez que são mostradas e escondidas, respectivamente. Deste modo, os processamentos que não precisam ser executados a cada carregamento da Activity devem ser movidos para outras classes.

Utilização de um padrão de arquitetura

Outra forma de manter o código fonte organizado e proporcionar condições favoráveis ao desenvolvimento de software de qualidade é através do emprego do padrão de arquitetura correto. O padrão recomendado varia de acordo com o projeto ou plataforma alvo. Entretanto, o padrão conhecido em Inglês como Model-View-Control, ou MVC, tem sido o mais difundido e recomendado para o desenvolvimento de sistemas Java.

Este padrão sugere que o código do sistema seja dividido em três camadas que interagem entre si com o objetivo de melhorar a organização, possibilitar o reuso do código de forma inteligente, facilitar a automação de testes unitários, entre outras vantagens.

A primeira camada é a Model, ou modelo. Esta define os dados a serem exibidos na interface do usuário, por exemplo, as informações armazenadas em um banco de dados ou em um arquivo. A segunda camada é a View, ou visão. Esta mostra os dados da Model para o usuário e repassa comandos vindos do usuário ou eventos do sistema para a Control (ou controlador), que, por sua vez, representa a terceira camada, sendo responsável por atualizar os dados da Model e o que deve ser mostrado na View.

Como dito, o MVC é o padrão de arquitetura mais utilizado no planejamento de sistemas Java. Entretanto, outros também podem ser utilizados, como o Model-View-Presenter, ou MVP. Neste padrão, a camada Presenter, ou apresentador, é responsável por formatar os dados da Model para serem apresentados na View.

Apesar das vantagens, existem situações em que não é aconselhável ou viável a separação do código nas três camadas do MVC ou do MVP. Sistemas que não possuem uma interface gráfica, por exemplo, não necessitam da View, assim como softwares estritamente visuais, como simples calendários, não precisam da Model. A adoção de um padrão de arquitetura é aconselhável quando há a possibilidade de separação de papéis entre as interfaces, classes e pacotes do software.

Nota

Para o desenvolvimento de aplicativos Android, o padrão MVP é o mais indicado devido ao uso correto e esperado das Activities. Essas são as classes básicas da plataforma e são responsáveis por tratar os eventos de janela como pause e resume, além de controlar e mostrar os dados para o usuário, papel esperado pela camada View do MVP. Já a camada Presenter é representada por classes que formatam a interface na qual as informações serão mostradas, como a ListView.

Utilização de padrões de projeto

Padrões de projeto são modelos para solucionar problemas comumente encontrados em sistemas de software e melhorar o design do código através da simplificação de suas estruturas básicas e do bom uso da orientação a objetos.

Os padrões de projeto aumentam a produtividade de um time por proverem modelos de programação comprovadamente funcionais e já testados em outras soluções, modelos estes que contribuem para a modularização e reutilização de estruturas de código. Esta modularização propicia um maior desacoplamento das estruturas do código e favorece possíveis mudanças no decorrer da implementação. Já a reutilização das estruturas de código proporciona o aumento da velocidade do processo de desenvolvimento.

O uso correto da orientação a objetos, uma das premissas dos padrões de projeto, permite uma melhor divisão de responsabilidades entre as classes, métodos e estruturas em geral do sistema. Além disso, os modelos aperfeiçoam a legibilidade do código, principalmente para profissionais que já conhecem e utilizam os padrões.

Entretanto, é necessário estar ciente que a aplicação de padrões de projeto pode ser onerosa tanto para o time e para o projeto, quanto para o software em criação. Isso porque mesmo times experientes e conheedores da motivação de se aplicar cada um dos padrões podem ter dificuldades em identificar a melhor opção a ser utilizada nas diversas situações encontradas. O processo que envolve desde conhecer o objetivo de uso de cada padrão de projeto até implementá-los da forma correta é demorado e requer repetição e estudo. Em outras palavras, a curva de aprendizado dos padrões projeto é longa.

Outro fator importante a ser analisado antes de se decidir em aplicar ou não um padrão de projeto é verificar se este modelo trará mais impactos positivos do que negativos. Isto porque os padrões de projeto podem afetar negativamente o desempenho do aplicativo. Lembre-se que a modularização do código proporcionada pelos padrões requer a adição de camadas e chamadas que poderiam não existir se os padrões não fossem utilizados.

Uma boa prática relacionada ao uso de padrões de projeto na codificação de sistemas Java é seguir os padrões já estabelecidos na plataforma alvo. O **BOX 1** mostra alguns exemplos de padrões já difundidos no SDK Android.

Outras boas práticas de desenvolvimento

As boas práticas citadas previamente sugerem, basicamente, melhorias na forma de se escrever o código fonte e no uso de me-

todologias de desenvolvimento. Entretanto, existem outras boas práticas a serem utilizadas que podem melhorar o rendimento do time na rotina básica de criação de software.

BOX 1. Padrões de projeto utilizados na plataforma Android

No código fonte da plataforma Android ou nos SDKs disponíveis para os desenvolvedores é possível encontrar diversos exemplos de uso de padrões de projeto. O padrão Builder, por exemplo, foca em tornar a inicialização de objetos complexos mais intuitiva. Basicamente, deixa-se de utilizar construtores com muitos parâmetros em favor de métodos set. Isto evita que haja a troca involuntária da posição de dois ou mais parâmetros, facilita a identificação de cada parâmetro e aumenta a legibilidade do código por tornar desnecessária a leitura da documentação do construtor para entender a inicialização do objeto em questão.

BroadcastReceivers são exemplos de uso do padrão Observer. Este padrão considera que um objeto é responsável por notificar seus dependentes quando há mudanças de estados. No caso dos BroadcastReceivers, as mudanças de estados são os eventos do sistema ou da aplicação recebidos. É possível, também, encontrar exemplos de uso dos padrões Adapter, Memento, Chain of Responsibility, ViewHolder, Proxy, Composite, Facade, entre outros. Um estudo aprofundado destes padrões e seus objetivos é indicado para programadores interessados em melhorar a qualidade do código fonte de seus aplicativos.

Como descrito anteriormente, mesmo em um processo simples é possível enumerar um alto número de problemas causados pelo uso de práticas inadequadas. Porém, as atividades listadas a seguir, se bem aplicadas pelo time de desenvolvimento, evitam o surgimento desses problemas ou os corrigem.

A utilização de uma ferramenta para controle de versão é uma atividade essencial para qualquer time de desenvolvimento ou projeto. Além da segurança de haver uma ou mais cópias do código fonte, o controle de versão possibilita a geração de releases, o acompanhamento do histórico da evolução do código e facilita o compartilhamento do código entre várias pessoas e a localização de bugs. O uso do Git para controle de versão é uma excelente prática a ser considerada.

Ao decidir adotar uma ferramenta para controle de versão, é preciso definir quais os arquivos, ou tipos de arquivos devem ser adicionados ao repositório. Arquivos binários, por exemplo, não têm necessidade de estarem sob versionamento, pois são gerados durante a compilação.

Outros arquivos que não devem estar sob controle de versão são aqueles utilizados para armazenar informações que variam de um programador para outro, como os caminhos de diretórios onde estão os arquivos do aplicativo, acessados durante a compilação. Quando atualizados com os dados do ambiente de outro programador, existe uma grande chance de ocorrerem erros de compilação, mesmo o código fonte estando perfeito.

Outra vantagem do uso de ferramentas de controle de versão é a possibilidade de se criar branches do código fonte. Os branches podem ser utilizados para separar códigos estáveis, já aprovados e testados daqueles recém-implementados, por exemplo. Esta é, inclusive, uma boa prática de desenvolvimento, pois facilita a identificação e correção de bugs, já que existe uma chance maior de os bugs terem sido adicionados em códigos não estáveis.

Entretanto, é importante evitar a criação e uso de branches em excesso. Muitos branches causam confusão aos membros da equipe que não os criaram e tornam o processo de atualizar o código mais lento.

Uma forma simples e ainda não descrita para otimizar a codificação é a utilização de ferramentas que visam automatizar tarefas para manter o código fonte limpo e organizado; prática esta associada ao uso de recursos presentes em algumas IDEs, como plug-ins para formatação e ordenação do código. Estes recursos podem ser configurados com as regras acordadas entre os membros do time e, ao serem executados, o código é organizado de acordo com o combinado.

Outra boa prática ainda não descrita e que pode tornar a fase de validação melhor aproveitada é a execução dos casos de testes relacionados ao que foi implementado pelo próprio desenvolvedor. Desta forma, reduz-se consideravelmente a probabilidade de que novos bugs sejam acrescentados através do código recém-implementado.

Com o intuito de diminuir as chances de adição ou reaparecimento de bugs, adotar um processo de revisão na fase de integração do código é bastante aconselhado. Neste processo a implementação é verificada e validada por outros membros do time, que devem sugerir mudanças em relação à legibilidade, ao uso da Orientação a Objetos, aos possíveis defeitos ou qualquer situação que vá de encontro ao que foi combinado para alcançar a meta de qualidade.

Outra excelente prática de desenvolvimento é a automatização dos processos de compilação, execução de testes e geração de releases. Nestes casos, a utilização de ferramentas como o Jenkins possibilita que essas atividades sejam realizadas de forma mais rápida e segura quando comparadas com a execução de forma manual.

Enfim, muitas são as práticas que podem ser adotadas na implementação de sistemas Java. Por isso, é importante que o processo de definição destas, leve em consideração as que mais se adequam aos objetivos do projeto, do produto e do time envolvido.

Boas práticas de testes de software

O processo de preparação e execução de testes de software é complexo e, assim como a codificação, sempre passível de melhorias. Não somente por isso, existem boas práticas a serem seguidas por todos os participantes do projeto para que as atividades de testes sejam realizadas de forma a criar sistemas com sucesso.

Uma opção a ser avaliada é considerar a participação dos profissionais de testes em todas as fases do projeto. Na primeira fase, quando o projeto é analisado e estimado, o testador colabora na identificação de riscos, exceções, casos de uso e cenários esquecidos. Além disso, por participar da avaliação do produto a ser criado juntamente com o restante do time desde o início, o testador não necessita de um período de adaptação ao projeto no momento de executar os ciclos de testes.

Neste período de adaptação, normalmente, é preciso que outro participante pare suas atividades constantemente para explicar ao testador os requisitos, fluxos e funcionalidades planejados para aquele ciclo.

Outra prática recomendada é a realização de testes de software durante a codificação. Esta sugestão é um importante complemento às metodologias ágeis, pois estas sugerem que o projeto seja dividido em partes menores, chamadas sprints. Usualmente, estes têm duração de duas ou três semanas, período no qual ocorrem a codificação, os testes e a correção dos defeitos de partes do produto em criação. Isto evita o acúmulo de problemas, pois estes são corrigidos em poucos dias ou horas após serem adicionados ao código.

Como comparação, na metodologia tradicional a fase de testes e correção de problemas ocorre somente após a implementação de todos os requisitos. Isto faz com que um problema adicionado no código no início do projeto persista por toda a fase de codificação, podendo causar outros defeitos. Além disso, os programadores, provavelmente, estarão fora do contexto deste problema quando ele for encontrado, levando a um aumento do tempo para a correção do mesmo.

Outra forma de melhorar a eficiência na identificação de defeitos é através da automatização dos testes. Ao tornar o processo de execução de casos de testes automático, espera-se reduzir a interação humana em atividades repetitivas e que não requerem validações subjetivas. Assim, o testador poderá se concentrar na execução manual de casos de testes mais complexos e na expansão da cobertura dos testes ou em testes exploratórios.

A automatização também facilita e complementa a validação do código desenvolvido. Isto porque logo após implementar uma nova funcionalidade ou corrigir um problema, o próprio desenvolvedor tem condições de executar uma bateria de casos de testes com o intuito de verificar se há o surgimento de um novo defeito após a adição do novo código. Esta forma de trabalho facilita a correção de defeitos, já que o programador está trabalhando no contexto onde o problema ocorre.

Como descrito anteriormente, ferramentas como o Jenkins podem ser utilizadas para executar os testes a cada vez que há integração de código no repositório. Esta prática é útil para complementar a validação do código feita pelo desenvolvedor durante a codificação.

Outra boa prática é desenvolver o produto direcionando o código para a automatização de testes, o chamado Test Driven Development, ou TDD. Neste processo, o programador primeiramente desenvolve um caso de teste e em um segundo momento cria o mínimo de código necessário para cumprir este teste. A vantagem de uso do TDD é que a automatização dos testes ocorre naturalmente e há uma maior garantia de que o produto atenda aos requisitos e tenha um padrão de qualidade elevado.

Além disso, o código produzido em um processo TDD tende a ser mais conciso, modularizado, com menos dependências e,

por consequência, mais fácil de interpretar. Isto porque apenas o essencial para cumprir o caso de teste é adicionado ao repositório, eliminando complexidades desnecessárias.

O uso de boas práticas é essencial para um ganho de produtividade de qualquer time de desenvolvimento. Estas práticas devem ser estudadas e avaliações constantes devem ser feitas com o objetivo de identificar as mudanças ou adaptações necessárias na organização ou no time para que elas sejam aplicadas sem causar transtornos e gerem os resultados esperados. Por isso, recomenda-se fortemente a continuidade nos estudos nesta área, o que pode ser feito através da leitura de livros, participação em treinamentos sobre padrões de projeto e desenvolvimento ágil, por exemplo, entre outras opções.

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Autor



André Pedralho

andre.pedralho@gmail.com

Mestre em Ciência da Computação na área de Recuperação de Informação, desenvolvedor de software, Scrum Product Owner, Gerente de Projetos e um eterno aprendiz. Mais de 50 aplicativos desenvolvidos e publicados, experiência em desenvolvimento Open Source, Tecnologias Web e de máquinas de renderização WebKit e Mozilla.



Links:

Livro recomendado: Clean Code: A Handbook of Agile Software Craftsmanship
<https://books.google.com.br/books?id=dwSFQAAQAAJ>.

Livro recomendado: Head First Design Patterns
<https://books.google.com.br/books?id=NXlrAQAAQBAJ>.

Livro recomendado: Essential Scrum: A Practical Guide to the Most Popular Agile Process
<https://books.google.com.br/books?id=3vGEcOfCkdwC>.

Por dentro do banco de dados NoSQL

Couchbase – Parte 1

Conheça este banco de dados e obtenha escalabilidade em seus sistemas

ESTE ARTIGO FAZ PARTE DE UM CURSO

Há cerca de 40 anos os bancos relacionais vêm sendo largamente utilizados. Com uma arquitetura centralizada e um esquema de tabelas para armazenamento de dados, eles sempre foram o principal meio de armazenamento e recuperação de dados da grande maioria dos sistemas. Porém, eles foram criados em uma época em que os sistemas comumente eram construídos em apenas duas camadas: um cliente que apresentava a interface sendo utilizada pelo usuário no computador do mesmo e um banco de dados acessível através da rede fechada da empresa. Isso significa que não houve uma preocupação muito grande com os acessos simultâneos, visto que isso não ocorria naquela época.

Acontece que com a chegada da Internet e o grande número de usuários que veio com ela, esse panorama está mudando. Os sistemas, por exemplo, passaram a ser implementados em três camadas: o cliente, que utiliza navegadores web e dispositivos móveis; o servidor; e a camada de armazenamento e recuperação dos dados. O problema é que tais clientes não são mais apenas aqueles que se encontram dentro da mesma empresa. Agora eles podem estar espalhados por todo o mundo, navegando na aplicação através da Internet.

Por causa disso, grandes redes sociais como o Facebook, sistemas de busca como o Google, entre outros, vêm recebendo uma quantidade inimaginável de acesso

Fique por dentro

Este artigo apresenta o Couchbase, um banco de dados NoSQL com grande potencial de escalabilidade, pronto para aplicações acessadas globalmente e capaz de prover sincronização de dados para aplicações mobile. Com base nisso, esse conteúdo é útil para desenvolvedores que procuram alternativas ao clássico banco de dados relacional, principalmente àqueles que precisam implementar aplicações com expectativa de grande quantidade de acessos simultâneos, bem como àqueles que pretendem criar soluções mobile que funcionem mesmo sem acesso à internet e possibilitem a sincronização dos dados criados offline.

por seus usuários e também por outros sistemas. Com o grande crescimento do acesso à internet em todo o mundo, esses sites agora precisam suportar diversos usuários durante todo o dia e em todos os dias do ano, o que os obriga a buscar pela mais alta taxa de disponibilidade.

Outro efeito colateral dessa grande quantidade de usuários é o grande volume de dados gerados pelos mesmos. Esses dados vêm sendo coletados por grandes empresas devido a seu grande valor de mercado e começam a ser utilizados como fonte de informação ao suporte de decisões estratégicas e mineração de dados. Esse novo uso faz com que ainda mais dados sejam coletados e mais espaço seja requerido para armazená-los.

Para suprir essas necessidades, tanto os sistemas quanto seus respectivos bancos de dados devem suportar a grande quantidade de acessos simultâneos, acomodar uma enorme quantidade de dados e estarem sempre disponíveis. O método comumente utilizado para que os sistemas Web suportem vários acessos simultâneos e permaneçam sempre disponíveis é o escalonamento horizontal. Ou seja, múltiplos computadores ou máquinas virtuais rodam o



mesmo aplicativo e a carga de usuários é distribuída entre eles. Deste modo, em caso de atualizações, não é necessário deixar todo o sistema fora do ar, mas apenas uma máquina por vez.

No entanto, com bancos de dados relacionais, o método mais comum é o escalonamento vertical. Isso significa que ao invés de designar múltiplas máquinas para essa função, uma única máquina é utilizada e, quando preciso, investe-se no melhoramento da mesma. Em geral, investe-se em HD maior para suportar mais dados, um processador e uma conexão mais rápida para suportar o alto número de acessos simultâneos, bem como no crescimento da memória RAM. Isso acontece devido à natureza centralizada dos bancos relacionais, o que torna necessário que eles estejam em uma única máquina.

O principal problema do escalonamento vertical é a relação custo-benefício. A curva desse, nesse caso, não é linear. Assim, apesar de perceber-se bom desempenho no início, chega-se a um ponto em que maiores investimentos em hardware geram pouca melhora de performance do sistema. Além disso, as melhorias são fixas e no caso de sistemas que recebem grandes quantidades de acesso em determinados períodos de tempo, observa-se um gasto desnecessário de recursos em momentos com pouco acesso.

Para amenizar esses problemas existem algumas técnicas que possibilitam estender/prolongar o uso de bancos de dados relacionais, como sharding, desnormalização e cache distribuído. Porém, essas técnicas apenas tentam compensar as limitações destes bancos com escalonamento horizontal.

Outro problema encontrado está relacionado à natureza esquemática dos bancos de dados relacionais. Todos eles requerem que um esquema de tabelas seja previamente especificado, e todos os dados serão inseridos nessas tabelas. Caso sejam necessários novos campos, uma atualização do esquema precisará ser feita, tornando o banco de dados indisponível durante a mesma. Para tabelas com grande quantidade de dados, isso pode significar horas de indisponibilidade, o que pode afetar gravemente um sistema.

Esse esquema de tabelas também gera problemas na implementação de programas, que normalmente trabalham com a orientação a objetos. Isso porque tais objetos devem ser convertidos em múltiplas tabelas relacionadas por foreign keys em nível de banco de dados e, para recuperá-los por completo com todas as suas relações com outros objetos, diversas tabelas devem ser combinadas. Mesmo com os frameworks conhecidos que são utilizados hoje em dia, como o Hibernate, isso continua sendo um desafio, visto que as consultas complexas criadas pelos mesmos podem causar problemas de performance.

Ademais, com a grande exigência atual de se armazenar todos os tipos de dados, torna-se indispensável um modelo mais flexível de persistência, que não implique na criação de um esquema para cada nova informação a ser adicionada.

Pensando em todos esses problemas, grandes empresas como o Google, Facebook e outras criaram suas próprias soluções, as quais foram abertas à comunidade e passaram a ser conhecidas como NoSQL, da sigla Not Only SQL (Não Somente SQL).

O principal objetivo desses bancos de dados é obter o tão importante escalonamento horizontal, assim como viabilizar modelos mais flexíveis de armazenamento, alta disponibilidade e suportar a alta demanda de usuários e dados.

Para tornar possível o escalonamento horizontal, todos os bancos NoSQL têm suporte a *auto-sharding*, o que faz com que os dados automaticamente sejam distribuídos entre os servidores sem a necessidade de participação da aplicação. Assim, novos servidores podem ser adicionados e removidos da camada de dados sem comprometer sua disponibilidade. Além disso, as operações de inserção e consulta naturalmente se espalham pelos nós do banco de dados. Outra funcionalidade também suportada pelas soluções NoSQL é a replicação de dados, o que assegura que múltiplas cópias sejam distribuídas pelo cluster, garantindo maior disponibilidade e recuperação de desastres. Com tudo isso é possível afirmar que o sistema nunca ficará offline.

Outro destaque é que mesmo com essa distribuição dos dados pelos servidores, esses bancos mantêm sua capacidade de pesquisa. Para reduzir a latência e aumentar a taxa de transferência de dados, eles possuem cache integrada, recurso que mantém em memória os dados mais acessados. Por sua vez, se um banco relacional é utilizado para algo semelhante, muito provavelmente os desenvolvedores precisarão criar essa camada de cache.

A partir de modelos de dados flexíveis, cada tipo de banco de dados NoSQL concentra-se em resolver problemas específicos. Atualmente, existem quatro categorias de soluções NoSQL, que podem ser identificadas através do formato dos dados, a saber: *key-value* (similar a uma hash table, armazena uma chave e seu respectivo valor), *column family* (as chaves apontam para múltiplas colunas, que são arranjadas em uma família de colunas), documentos (similar a *key-value*, com a diferença que a chave aponta para um documento, que possui uma porção de outras chaves e valores) e grafos (possibilita o armazenamento de grafos, que podem ser muito úteis em redes sociais, por exemplo).

A categoria abordada aqui será a de documentos, visto que o tema central do artigo, Couchbase, é um banco de dados que armazena primariamente documentos, apesar de também suportar o formato *key-value*.

Quando se busca por esse tipo de banco de dados, o mais conhecido é o MongoDB. Isso porque ele carrega certas similaridades que tornam fácil seu uso para aqueles acostumados com SQL. Como os outros bancos desse tipo, o MongoDB tem documentos como unidade de armazenamento, podendo aninhar objetos para criar documentos complexos. Esses documentos são agrupados em coleções e são pesquisáveis de forma similar às consultas feitas em bancos de dados relacionais.

Um competidor direto do tão conhecido MongoDB é o Apache CouchDB, que também tem documentos como unidade básica de armazenamento. Esse foi criado pela CouchOne Inc., que mais tarde se fundiu à Membase Inc. para criarem o Couchbase, opção que combina o modelo de dados orientado a documentos e a capacidade de indexação e pesquisa do CouchDB com a alta performance, fácil escalabilidade e alta disponibilidade do Membase.

Assim é criada a grande rivalidade entre MongoDB e Couchbase. Apesar de menos conhecido que seu rival, o Couchbase vem apresentando um grande crescimento em seu uso. A exemplo disso, temos o Viber (famoso concorrente do WhatsApp), que utilizava a combinação de MongoDB e Redis, e migrou para Couchbase puro, devido à melhor escalabilidade desse.

Outra capacidade recentemente adicionada e que vem atraindo mais usuários é a versão mobile do Couchbase. Essa versão pode ser embarcada em dispositivos móveis Android e iOS, bem como em aplicações HTML5, e permite a sincronização de dados com um banco de dados central. Com isso, é possível trabalhar offline e manter os dados atualizados, ao estabelecer uma conexão com a internet. Mais sobre isso será falado mais adiante.

Muito pode-se discutir sobre a rivalidade entre esses bancos de dados. Porém, esse não é o foco deste artigo, que visa apresentar o Couchbase e seu Java SDK. Por isso, começaremos explicando a arquitetura e funcionamento desse banco de dados.

Arquitetura do Couchbase

Para alcançar o tão almejado escalonamento horizontal o Couchbase funciona como um cluster, que nada mais é do que um grupo de nós interconectados. Neste grupo cada nó pode estar em máquinas físicas ou virtuais distintas, interconectados através da internet, e diferentemente da arquitetura comum, em que um dos nós é considerado mestre e os outros, escravos, nesse caso todos os nós são considerados iguais, não havendo hierarquia.

Do ponto de vista do desenvolvedor, no entanto, isso pode parecer estranho. Afinal, qual IP deve-se configurar como servidor do banco de dados para determinada aplicação? O que acontece é que o SDK provido para o Couchbase já trata esse tipo de problema. Ele permite a configuração de uma lista com alguns IPs de nós conhecidos. Então, o SDK tentará se conectar a um deles e, quando a conexão com qualquer um for bem-sucedida, tal nó informará ao SDK todos os nós conectados no cluster. Assim, caso um nó tenha sido adicionado ou removido, não será necessário atualizar a lista de nós manualmente na aplicação cliente. Essa lista de nós se mantém atualizada em tempo de execução à medida que novos nós são inseridos ou removidos.

Com isso, evita-se a hierarquização do cluster. A grande vantagem dessa abordagem é que não haverá um único ponto de falha. Se, por outro lado, houvesse um nó mestre e os outros fossem seus escravos, ou o cluster inteiro ficaria fora do ar, caso o mestre caísse, ou seria necessária alguma lógica para decidir um novo mestre. O objetivo dessa falta de hierarquização é tornar o banco de dados altamente disponível, possibilitando que ele continue funcionando mesmo que alguns nós venham a cair temporariamente.

Mas então, onde estão os dados? Cada nó armazenará uma porção dos documentos, o que é chamado de *auto-sharding*, e uma réplica de cada documento também é armazenada em outro nó. Assim, se por eventualidade aquele nó que possuía determinados documentos venha a ficar fora do ar, o cluster passa por um auto balanceamento, fazendo com que as réplicas assumam o lugar dos originais e novas réplicas sejam criadas. Do mesmo modo, quando

novos nós são adicionados ou um nó previamente existente retorna, o processo de balanceamento também é executado.

No entanto, tal arquitetura gera alguns problemas, como onde encontrar determinado documento e como fazer algum tipo de pesquisa. Felizmente, esses são resolvidos de forma transparente ao desenvolvedor pelo próprio SDK. Ao estabelecer a conexão com o cluster, o cliente não só sabe sobre todos os nós disponíveis, mas também recebe informações sobre onde encontrar cada documento pela chave. Esses dados de localização de documentos são sincronizados constantemente, para que qualquer busca por chave seja feita diretamente no nó adequado.

A pesquisa por documentos através de determinados critérios, também conhecida como query, é feita da seguinte maneira: o SDK se comunica com cada um dos nós para realizar uma consulta, os nós retornam seus resultados e ele os agrupa, gerando o resultado final. Essa comunicação com os nós e a coleta dos resultados é toda feita de forma transparente pelo SDK ficando, portanto, implícito ao código do cliente. Mais detalhes sobre esse tipo de pesquisa serão apresentados mais adiante, visto que esse é o tópico mais complexo ao lidar com o Couchbase.

Entendido como os nós se organizam e se comunicam com o SDK, vale conhecer o funcionamento de um nó. Esse, para garantir tempos de resposta de milissegundos, opera com um cache em memória. Esse cache é configurado individualmente em cada nó, onde define-se a quantidade de memória reservada que será utilizada para manter os documentos mais utilizados. Ademais, as operações de leitura e escrita de dados são realizadas diretamente em memória, não sendo necessária uma operação de I/O com o disco rígido, o que garante uma velocidade de resposta muito alta.

Após essas operações em memória os dados modificados são escritos no disco. Para otimizar essa escrita, apenas operações de append são realizadas em arquivo. No entanto, essas operações podem gerar grande quantidade de lixo devido à redundância criada por várias versões de um mesmo documento permanecer em disco, mas apenas a mais atual ser válida. Esse lixo é removido por outro processo que roda de tempos em tempos. Além desse, mais um processo que é executado é o de sincronização dos documentos, sendo este o responsável por manter as réplicas dos documentos atualizadas.

Com essa arquitetura o Couchbase consegue garantir máxima performance e ótima escalabilidade para um grande número de acessos simultâneos com a simples adição de nós. Vale ressaltar, também, que o cache nativo desse banco torna desnecessária a camada de cache comumente inserida entre a aplicação e o banco de dados, quando lidamos com soluções relacionais.

Ainda há muitos detalhes sobre a arquitetura desse banco de dados. Alguns desses, os pertinentes ao desenvolvimento, serão abordados em seus respectivos tópicos. Outros, no entanto, não são tão importantes do ponto de vista do programador e, por isso, não serão explorados aqui. Seguiremos, portanto, com os tópicos relacionados ao desenvolvimento de sistemas que utilizem o Couchbase.

Modelagem de documentos

A primeira grande diferença entre desenvolver utilizando bancos relacionais e desenvolver utilizando uma solução NoSQL com a estrutura de dados de documentos é a forma como modela-se os dados a serem persistidos e recuperados. Como dito anteriormente, os bancos relacionais não se adaptam tão bem ao paradigma de programação mais comum atualmente, o orientado a objetos. Com esses bancos, objetos são convertidos em registros de tabelas para que possam ser armazenados e isso faz com que qualquer relação entre objetos exija a criação de novas tabelas, mesmo que determinado objeto referenciado jamais seja recuperado separadamente.

O que acontece com esse tipo de conversão é um excesso de tabelas criadas. Para exemplificar esse cenário pode-se pensar na entidade usuário, a qual possui um endereço. Em um banco de dados relacional, mesmo que esse endereço seja único para aquele usuário e jamais seja recuperado separadamente, ele comumente será armazenado em outra tabela e um join será necessário para obtê-lo. Pode-se até colocar seus atributos na tabela do usuário, no entanto isso torna o esquema confuso, já que registros de usuário passam a armazenar campos de endereço.

Do outro lado, temos bancos de dados cuja estrutura de dados são documentos. No caso do Couchbase, tais documentos são estruturados no bem conhecido formato JSON, que funciona como um mapa de chave-valor. Os valores possíveis de tal mapa podem ser tanto de um tipo básico, como **String**, **boolean**, número e ainda um vetor, quanto outro mapa, que poderá ter seu próprio conjunto de chave-valor. Assim, é possível aninhar esses mapas e construir estruturas mais complexas.

Esse tipo de estrutura se assemelha bastante com objetos, por permitir que dados mais complexos sejam armazenados, o que facilita a conversão. Inclusive, em JavaScript, por exemplo, os objetos podem ser convertidos diretamente em documentos JSON e vice-versa sem o uso de qualquer biblioteca externa. Outras linguagens possuem bibliotecas que fazem tal conversão automaticamente através de reflection. E no caso do Java temos as famosas bibliotecas Jackson e Gson. Portanto, modelar documentos é algo muito mais próximo da modelagem de objetos.

Voltando ao caso citado anteriormente, em vez de criar um documento para o usuário e outro para o endereço, podemos simplesmente criar um único documento contendo os dados do usuário e uma chave contendo o mapa com os atributos do endereço. A **Figura 1** mostra como esse exemplo seria modelado com um esquema relacional e a **Figura 2** mostra sua versão em documento JSON.

Como pode-se notar, a possibilidade de aninhar objetos em documentos traz uma grande vantagem. Porém, deve-se ter certo planejamento ao modelar tais documentos, pois tal vantagem pode se tornar um problema, dependendo dos objetivos do sistema. Se, por exemplo, fosse necessário um relatório que distribuisse em um mapa a localização dos usuários sem discriminá-los, o aninhamento do endereço no documento do usuário tornaria necessário que tal relatório buscasse os dados completos de cada

usuário, mesmo que a maioria dos dados fosse desnecessária. Nesse caso, talvez fosse mais interessante criar outro documento com o endereço e relacioná-los, assim como no esquema relacional. A **Figura 3** mostra essa nova organização.

Nome	Email	Idade
Fernando Camargo	fernando.camargo.ti@gmail.com	24
Luíza Silva	luiza.silva@gmail.com	25
Logradouro	Número	Bairro
Rua 31-A	145	Setor Aeroporto
Rua 17-A	1139	Setor Aeroporto

Figura 1. Esquema relacional da relação entre usuário e endereço

```
{  
  "nome": "Fernando Camargo",  
  "email": "fernando.camargo.ti@gmail.com",  
  "idade": 24,  
  "endereco": {  
    "logradouro": "Rua Cd. Bonfim",  
    "numero": 178,  
    "bairro": "Setor Central"  
  }  
}
```

Figura 2. Esquema JSON para relação entre usuário e endereço

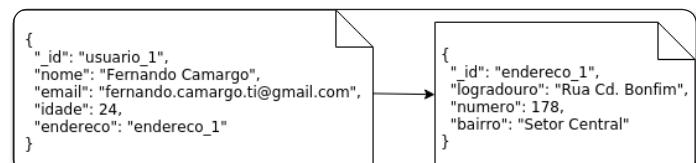


Figura 3. Esquema JSON com separação do documento de usuário e endereço

Assim como em um banco de dados relacional, é possível criar ligações entre documentos de maneira semelhante às *foreign keys*. Porém, aqui essas ligações não são verificadas pelo próprio banco. A responsabilidade de criar e manter tais ligações é da aplicação, por causa da natureza dos bancos de dados NoSQL, que não forciam nenhum esquema rígido para os dados.

Comumente, temos objetos se relacionando de uma das seguintes maneiras: um para um, um para muitos e muitos para muitos.

Por dentro do banco de dados NoSQL Couchbase – Parte 1

Em bancos relacionais, a primeira pode ser feita atribuindo-se a *foreign key* a um dos lados da relação, geralmente a mais importante. Já no caso de um para muitos, limita-se a colocar a chave do lado de muitos. Enquanto isso, no Couchbase, devido à sua capacidade de armazenar estruturas mais complexas em documentos JSON, temos uma opção a mais para a relação de um para muitos. Ao invés de limitarmo-nos a colocar a chave apenas do lado de muitos, é possível criar um *array* de chaves para criar a referência do lado unitário da relação, ou seja, um documento terá um campo que armazenará uma lista de chaves, as quais referenciarão outros documentos. Isso é demonstrado na **Figura 4**.

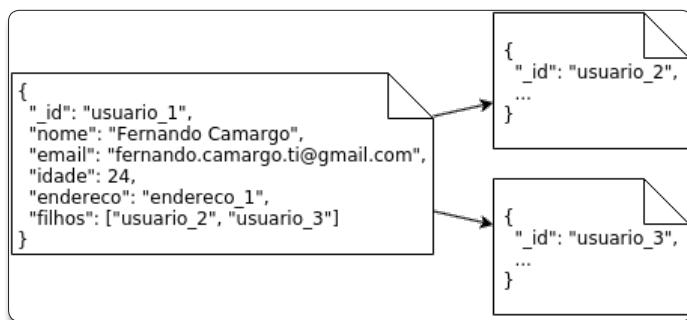


Figura 4. Demonstração NoSQL da relação um para muitos com referências do lado unitário

Com a possibilidade de colocar a chave de referência tanto do lado unitário quanto do lado múltiplo, qual deles deve-se escolher? Para tal decisão tem-se duas considerações a serem observadas. A primeira é sobre a concorrência na modificação de documentos. Caso a entidade unitária contenha uma lista de referências para a outra entidade, seu documento deverá ser modificado cada vez que um novo membro for criado para manter a lista atualizada, o que pode gerar conflitos em casos de modificação concorrente. E a segunda consideração é sobre o modo de pesquisa a ser utilizado. Para o uso de views e index do Couchbase, que serão apresentados mais adiante, não seria possível obter todos os documentos e seus relacionados em uma única chamada se a chave estiver do lado unitário. Para obtê-los, uma chamada extra seria necessária para recuperar os documentos relacionados através da lista de chaves.

Continuando essa análise sobre a modelagem de documentos, ainda deve-se estudar algumas particularidades. A primeira delas é sobre os campos especiais do Couchbase, utilizados para o armazenamento de metadados. A seguir, esse assunto será explorado.

ID

O primeiro campo especial do Couchbase é um já visto nos exemplos anteriores: **id**. Esse campo armazena a chave do documento, sendo utilizado para identificá-lo no cluster e para recuperá-lo diretamente. A chave tem algumas propriedades interessantes de se notar, como:

- Não é permitido espaço;
- Tem **String** como tipo de dados;

- Separadores e identificadores são permitidos, como o travessão. Por exemplo: "usuario_5";
- Deve ser única. Na tentativa de inserir outro documento com a mesma chave, o anterior será substituído ou um erro será gerado;
- O tamanho máximo é de 250 bytes e todas permanecem na memória RAM e não são removidas nem mesmo em caso de falta de memória. Portanto, deve-se ter isso em mente ao planejar o tamanho das chaves utilizadas.

Para preencher os valores dessas chaves, é comum o uso de **UUID** (*Universally Unique Identifier*). Esses são identificadores gerados com a garantia de serem universalmente únicos, como o próprio nome indica. Em Java, usa-se a classe **java.util.UUID** para essa tarefa.

Também é comum, mas não necessário, prefixar a chave com um texto que indique de qual entidade se trata. Porém, veremos adiante sobre o uso de uma propriedade para se especificar a qual entidade determinado documento se refere.

Devido à capacidade do Couchbase de armazenar chave-valor, além de documentos, também é possível o uso de um número sequencial como chave dos documentos. Para tal, usa-se o operador **increment**, o qual recebe uma chave, um valor inicial e o valor de incremento. A operação desse método é atômica, o que significa que pode ser utilizada por vários servidores em paralelo. A primeira chamada desse método para determinada chave cria um valor numérico com o valor inicial especificado e chamadas posteriores para essa chave fazem com que seu valor seja acrescido com o valor do incremento. Ao final da operação o valor atual atribuído à chave é retornado à aplicação, a qual poderá utilizá-lo como desejar. Um uso comum desse recurso é como número sequencial na criação de chaves para novos documentos.

Para exemplificar o operador **increment**, consideremos a criação de uma chave para documentos de usuário. Com esse fim, criaremos o contador chamado "**usuario_counter**". Seu valor inicial será 1 e ele será incrementado em 1 a cada novo documento. Para não haver choque de chaves com outros documentos com chave sequencial, os documentos de usuário terão suas chaves prefixadas com "**usuario_**" seguidas do número sequencial, resultando em chaves como: "**usuario_1**", "**usuario_2**" e assim sucessivamente. A partir disso, cada tipo de documento poderia ter um contador que seria utilizado como sequencial para suas chaves.

CAS

Outro campo muito importante é o **CAS**, utilizado em operações do tipo **CAS** (*Compare And Swap*). Para explicar esse tipo de operação, vamos abordar primeiramente a diferença entre bloqueio otimista e pessimista.

Começamos a ver essa diferença com o exemplo de concorrência de dois usuários por determinado documento. O caso mais simples ocorre quando ambos os usuários abrem o mesmo documento, o editam e tentam salvar. O resultado disso depende da estratégia de bloqueio adotada.

Naturalmente, o segundo usuário a salvar não deveria simplesmente sobrepor as alterações do primeiro, principalmente porque ele detinha uma cópia desatualizada do documento no momento em que tentou persistir suas alterações.

Para resolver esse problema, existem duas formas de bloqueio no Couchbase: o otimista, para quando assumimos ser raro o caso citado anteriormente, e o pessimista, em que assumisse que isso vai acontecer constantemente e que tais conflitos não possam ocorrer.

Para utilizar-se de um bloqueio pessimista, usa-se a API de get-and-lock do Couchbase para obter um documento e definir um tempo limite para manutenção do bloqueio. Assim, até que o documento seja liberado diretamente pelo sistema ou tenha seu tempo limite estourado, não será permitido que esse documento seja obtido por outro usuário, o que garante que não haverá concorrência para realizar modificações do documento.

No entanto, o bloqueio pessimista tem um custo elevado, visto que prejudica a taxa de acesso e modificação no sistema, além de solicitar que o banco de dados gerencie tais bloqueios. Por isso, a não ser que esse seja necessário, recomenda-se o uso de um bloqueio otimista através dos métodos da API que fazem uso de CAS.

O bloqueio otimista, por sua vez, funciona da seguinte maneira: um campo especial do documento armazena a versão atual do mesmo. A partir disso, caso esse documento seja modificado, esse campo é atualizado e uma futura tentativa de atualizar o documento com uma versão antiga resultará em erro. Portanto, no caso apresentado anteriormente, quando o segundo usuário tentar salvar suas alterações, o sistema retornará um erro avisando que o documento foi atualizado por outro usuário. Dependendo da implementação da aplicação, pode-se auxiliar o usuário a misturar as modificações ou fazer com que o mesmo atualize o documento e refaça as alterações.

Para utilizar esse tipo de bloqueio tem-se o campo CAS no documento. Seu valor é gerado automaticamente pelo Couchbase e atualizado quando o documento é alterado. No momento em que uma operação de alteração é realizada, esse valor será conferido com o valor atual presente no banco de dados. Caso o valor seja diferente, isso indicará que o sistema está utilizando uma versão anterior do documento e a operação será negada.

TL

Para permitir a criação de documentos com a capacidade de expirar, existe um campo responsável por armazenar o TTL (*Time To Live*) destes documentos. Para que esse recurso funcione, como parte das operações de manutenção, o Couchbase periodicamente remove todos os itens cujo tempo de expiração tenha sido alcançado.

Por padrão, todos os documentos possuem TTL 0, o que significa que jamais expirarão. Para ajustar tal valor, os SDKs possuem um parâmetro nos métodos de inserção e atualização do documento. O valor realmente armazenado no banco de dados é do tipo Unix Time, o que significa que se armazena a quantidade de segundos

passados desde 1º de janeiro de 1970, às 00:00:00, em vez do valor relativo ao momento em que o documento foi salvo. Porém, para facilitar, os SDKs provêm métodos auxiliares para ajustar o TTL relativo ao momento atual.

A aplicação mais comum desse recurso do Couchbase é para o armazenamento de sessões dos usuários. Seu uso se justifica porque sistemas com um grande número de acessos comumente utilizam escalonamento horizontal com balanceamento de carga. Esse escalonamento, no entanto, faz com que múltiplas requisições de determinado cliente sejam atendidas por diferentes servidores, e isso torna inviável que as sessões sejam armazenadas em memória, visto que elas ficariam inacessíveis em outros servidores. Para resolver esse problema é comum armazenar os dados da sessão em uma camada compartilhada, solução esta que é comumente viabilizada com o uso do Redis.

Contudo, com o Couchbase essa camada extra torna-se desnecessária. Como visto anteriormente, ele apresenta uma boa performance por fazer cache em memória dos documentos recentemente acessados. E devido à sua capacidade de armazenar até mesmo conteúdo binário como anexo aos documentos, pode-se fazer serialização dos dados da sessão, facilitando a implementação do gerenciamento destas. Para manter a sessão viva, atualiza-se o campo de TTL a cada vez que uma nova requisição com determinada sessão seja recebida.

Considerações finais sobre a modelagem de documentos

A análise sobre a modelagem de documentos será concluída com um dos assuntos mais importantes: os tipos de documentos e o versionamento dos mesmos.

Do ponto de vista do Couchbase, não há distinção entre documentos. Diferentemente de um banco de dados relacional, no qual diferentes registros residem em tabelas distintas, para o Couchbase todo documento é simplesmente o valor para uma chave, o que permite a criação de modelos de dados bem flexíveis. Entretanto, na maioria dos sistemas temos entidades bem definidas e que são raramente alteradas. Assim, a não ser em casos específicos de necessidade, os documentos terão que seguir determinados esquemas vinculados às suas entidades. Porém, esse esquema não é fixado em banco de dados. Ao invés disso, se trata de um sistema informal, conhecido e controlado pela aplicação cliente. Por isso, usa-se um campo do documento para especificar o seu tipo.

O campo utilizado para isso, portanto, não é reservado pelo banco de dados. Segue-se, então, a convenção de se definir um campo com o nome *type* e nele armazenar uma **String** única que especifica o tipo do documento. Para os exemplos anteriores, pode-se preencher tal campo com ‘usuario’ e ‘endereco’, por exemplo, e esses valores poderiam ser armazenados em constantes definidas na aplicação cliente.

Entretanto, apenas a especificação do tipo do documento não é suficiente. Em um banco relacional determina-se um esquema para cada tabela e, se tal esquema deve ser modificado, a tabela e todos os seus registros são atualizados. No caso de um banco sem esquema pré-definido, como o Couchbase, deve-se estabelecer

uma forma de atualizar documentos quando há uma modificação considerável em sua estrutura. Como não é desejado que todos os documentos de determinado tipo sejam atualizados de uma só vez devido à indisponibilidade que isso geraria, utiliza-se uma técnica que visa atualizar documentos apenas quando estes forem acessados.

Essa técnica recomendada necessita de dois campos extras em cada documento. O primeiro, já citado, será responsável por armazenar o tipo do documento, sendo chamado *type*; e o segundo, por armazenar a versão do esquema, possuindo o nome *version*. Nesse modelo, caso seja necessária a adição de um novo campo no documento, não é preciso criar uma nova versão do documento. Contudo, caso sejam necessárias mudanças mais significativas, como a remoção de um campo para adicionar outro ou uma mudança em referências para outros documentos, é indicada a atualização da versão do documento.

Para realizar esse versionamento serão criados mapeadores na aplicação que serão responsáveis por fazer o mapeamento entre o objeto e o documento de determinada versão. Deste modo, a versão atual do documento será conhecida pelo sistema e sempre que ele for armazenar um objeto, usará o mapeador da última versão. Porém, na leitura de um documento será utilizado o mapeador da versão do documento, sendo esse responsável por criar o objeto mesmo que seja uma versão antiga. Quando esse objeto for atualizado no banco, naturalmente seu respectivo documento será transformado para a versão mais atual.

Para exemplificar, considere uma primeira versão de um documento do tipo usuário. Tal versão possui um único campo para nome, o qual armazena o nome completo do usuário. Por algum motivo, deseja-se separar o nome em dois campos: primeiro e último nome. Portanto, uma nova versão desse documento será criada e o objeto no sistema passará a ter dois campos para nome, em vez de um. O mapeador da primeira versão deverá, então, ler o campo único, dividi-lo em dois através de split com o primeiro espaço, por exemplo, e retornar o objeto de usuário com os dois campos novos. Quando o sistema for salvar novamente esse usuário, ele será salvo na nova versão, contendo os dois campos de nome.

Ainda nesse exemplo, supondo que simplesmente deseja-se adicionar um campo de telefone comercial, não será necessária uma nova versão. Em vez disso, o mapeador da versão atual será simplesmente atualizado para ler e gravar esse novo campo. Quando um documento que não possui tal campo for lido, o respectivo objeto simplesmente terá valor nulo para esse campo.

Esse esquema de versionamento de documentos pode gerar um pouco mais de trabalho para o desenvolvedor, no entanto, evita que o banco de dados passe por um longo período de atualização de esquema, o que o tornaria indisponível. Ademais, atualizações que requerem novas versões deverão ser raras na maioria dos casos, o que faz com que essa técnica de versionamento não crie grande peso para os desenvolvedores.

A técnica de versionamento apresentada aqui é muito útil para acabar com os problemas de indisponibilidade de bancos de dados, de forma que cada documento terá seu esquema atualizado apenas quando acessado. Vale notar que essa técnica pode ser aplicada a outros bancos de dados baseados em documentos, como o MongoDB, principal rival do Couchbase.

A segunda parte deste artigo mostrará na prática a utilização do Couchbase, desde sua configuração até o desenvolvimento de uma aplicação cliente com seu Java SDK. A técnica de versionamento aqui apresentada será implementada e terá seu uso demonstrado. Também serão mostrados os meios de acesso a documentos, inserção, remoção e pesquisa. Por fim, o leitor estará apto a utilizar esse banco de dados NoSQL em suas aplicações.

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!



Como usar o Apache Cassandra em aplicações Java EE - Parte 1

Veja nesse artigo como tirar proveito dessas tecnologias trabalhando em conjunto

ESTE ARTIGO FAZ PARTE DE UM CURSO

Ao longo das últimas décadas muitas coisas aconteceram no mundo da tecnologia: mudanças em linguagens de programação, novas arquiteturas, diferentes metodologias de desenvolvimento, entre outros. No entanto, uma coisa permanecia intacta: bancos de dados relacionais eram a escolha padrão para armazenar dados. Com o crescimento acelerado da Internet e a necessidade cada vez mais comum de manipular altos volumes de dados, isso mudou um pouco. Uma nova tecnologia emergiu e vem se consolidando nos últimos anos: são os chamados bancos de dados NoSQL.

Um dos principais problemas dos bancos de dados relacionais para lidar com grandes massas de dados é o fato de que sua arquitetura cria dificuldades para que esses bancos rodem em cluster. Dessa forma, quando surge a necessidade de escalar as alternativas normalmente são:

- **Escalabilidade Vertical:** consiste em aumentar os recursos do servidor (memória, CPU, disco e etc.). Além de ter um limite máximo real, normalmente tem custos proibitivos;

- **Sharding:** essa técnica divide os dados da aplicação em mais de um servidor, distribuindo melhor a carga. O problema é que traz uma enorme complexidade para a aplicação, perde as melhores vantagens dos bancos relacionais, como integridade referencial, e continua tendo um ponto único de falha;

Fique por dentro

Este artigo será útil para profissionais que trabalham com aplicações Java EE e desejam incorporar ao seu pool de tecnologias um banco de dados NoSQL para lidar com as crescentes demandas por performance: o Apache Cassandra.

O artigo irá apresentar uma visão geral do Apache Cassandra incluindo conceitos chave, modelo de dados, constraints, ferramentas, boas práticas, entre outros. Além disso, irá demonstrar através da implementação de uma aplicação simples como combinar esse banco de dados com as tecnologias do Java EE, e para isso, será utilizado o driver da DataStax, WildFly 9, PrimeFaces 5.3, Cassandra 2.2, além de outras tecnologias.

- **Master-Slave:** um servidor (Master) recebe todas as escritas e replica para as demais instâncias (Slaves), as quais podem atender apenas requisições de leitura. Apesar de poder distribuir melhor a carga entre vários servidores, continua tendo um ponto único de falha e não consegue ter escalabilidade nas operações de escrita por ter apenas um servidor atendendo esse tipo de requisição. Ademais, pode acarretar em custos que inviabilizam sua adoção.

Por esse motivo os bancos de dados NoSQL vêm se popularizando cada vez mais. Executar em cluster com naturalidade, ter alta disponibilidade, facilidade de rodar na nuvem são aspectos comuns nesses novos bancos, pois nasceram justamente para resolver esse tipo de problema. Nesse contexto, o Apache Cassandra se destaca por possuir um modelo arquitetural que proporciona todas essas funcionalidades de uma maneira que minimiza a complexidade existente nesse tipo de ambiente.

Assim, nas próximas seções este artigo irá apresentar o Apache Cassandra de maneira mais detalhada, com o intuito de proporcionar ao leitor um embasamento teórico para o melhor entendimento da tecnologia, bem como irá demonstrar por meio de um exemplo prático algumas das funcionalidades explicadas. Além disso, também será exposta uma abordagem para usá-lo em conjunto com a plataforma Java EE. E para tornar o exemplo mais próximo do nosso dia a dia, outras tecnologias serão usadas, como o PrimeFaces e seus novos recursos de responsividade, CDI e DeltaSpike.

Conhecendo o Apache Cassandra

O Cassandra é um banco de dados NoSQL orientado a colunas desenvolvido em Java. Criado pelo Facebook e depois doado para a Fundação Apache, hoje é reconhecido na indústria de software como um banco de dados massivamente escalável, de alta disponibilidade, distribuído, dentre outras características essenciais para suportar volumes de dados colossais, com crescimento exponencial e carga excessiva de requisições.

Antes de analisar outros detalhes, a seguir serão apresentados alguns conceitos importantes para facilitar o entendimento do restante do artigo:

- **Cluster:** consiste num grupo de máquinas (nós) onde os dados são distribuídos e armazenados. Pode ser composto de um único nó (*single-node cluster*) ou vários nós em diversos data centers. A **Figura 1** apresenta um exemplo;
- **Data center:** uma subdivisão dos nós do cluster, os quais estão ligados para propósitos de segregação de replicação e carga. Por exemplo, é possível configurar o Cassandra para replicar dados apenas entre nós do mesmo data center, o que normalmente envolve menos latência do que replicar através de múltiplos data centers. Não se trata necessariamente de um data center físico;
- **Nó:** uma máquina que faz parte do cluster e que consequentemente armazena dados da base;
- **Keyspace:** tem o conceito similar a um database no PostgreSQL, onde tabelas são agrupadas para uma finalidade específica. Normalmente para separar dados de aplicações diferentes;
- **Família de colunas (Column-Family):** nas versões mais atuais do Cassandra esse termo foi substituído por Tabela. Trata-se de um conjunto de pares chave/valor (nome da coluna/valor da coluna) onde são armazenadas as informações da base. Pode-se dizer que com o CQL3 (*Cassandra Query Language*) esse termo ficou obsoleto.

CQL – O SQL do Cassandra

O CQL (*Cassandra Query Language*), como o próprio nome já diz, é a linguagem de consulta para o Cassandra. Atualmente na versão 3.3, é a interface primária para estabelecer comunicação com essa base de dados. Além disso, por possuir muitos aspectos similares ao SQL, não se restringindo apenas ao nome, o CQL3 facilita bastante o aprendizado de profissionais que estão habituados a trabalhar com bancos de dados relacionais. Antes do CQL a interface padrão do Cassandra era a Thrift API (vide **BOX 1**).

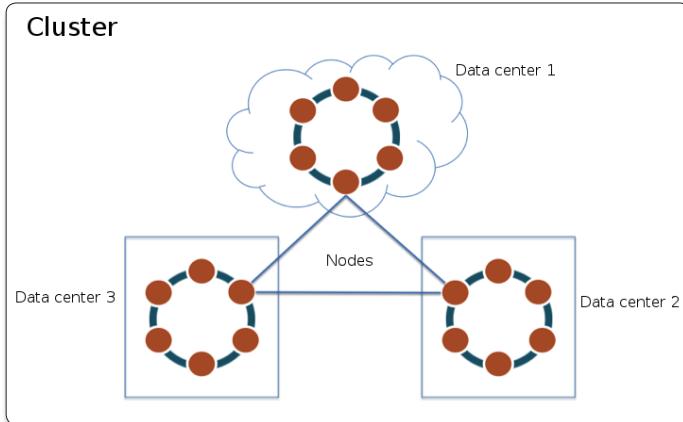


Figura 1. Representação de um cluster multi-data center – Adaptado de DataStax

BOX 1. Thrift API

Nos primórdios do Cassandra a única opção disponível para consulta era a Thrift API, uma interface baseada no protocolo RPC e que era bastante burocrática e difícil de entender à primeira vista. Em seguida houve algumas melhorias com o advento do CQL, mas ainda assim muitas características da Thrift API estavam presentes. Somente com o CQL3 o Cassandra pôde ter uma forma de comunicação mais intuitiva, simples e produtiva.

O CQL3 também impactou a forma de modelar dados para o Cassandra. Assim, caso você esteja interessado em se aprofundar no assunto é importante entender que existe uma fase “pré-CQL3” e outra “pós-CQL3”. Isso irá facilitar os estudos e evitará confusão ao aprender dicas e boas práticas diferentes para cada uma dessas fases. Neste artigo, iremos focar no CQL3.

Acessando o CQL

A maneira mais comum de acessar o CQL é através da ferramenta *cqlsh*, como pode ser observado na **Figura 2**. Trata-se de um cliente de linha de comando que vem junto com a instalação do Cassandra (*CASSANDRA_HOME/bin/cqlsh*).

```
cqlsh> select * from webshelf.user where login='marlon';
  login | name          | password
  +-----+-----+
  marlon | Marlon Patrick | c8f759a539858b08e9e46251blaef09f09
(1 rows)
cqlsh>
```

Figura 2. Executando comandos CQL através do *cqlsh*

Caso queira optar por uma ferramenta gráfica, o DataStax DevCenter é uma ótima opção (vide **Figura 3**). Esta ferramenta é baseada no Eclipse e traz algumas views especializadas para o Cassandra:

- **View Connections:** Espaço onde você pode gerenciar todas as conexões que criou para algum cluster do Cassandra.
- **View Schema:** Aqui são listados todos os objetos de uma determinada conexão, o que permite uma visualização hierárquica da estrutura do banco (*keyspace > tabela > coluna*);

- **View CQL Scripts:** Através dessa view é possível gerenciar scripts CQL: criar, editar, deletar;
- **View Results:** Exibe o resultado da última consulta executada; e
- **Editor CQL:** Editor que possibilita escrever e executar comandos CQL, faz destaque de palavras reservadas, tem *code completion* e ainda possibilita escolher a conexão onde o comando será executado para cada arquivo aberto.

Escolher entre uma ferramenta e outra normalmente é uma questão de preferência. O DevCenter é mais indicado para quem está iniciando devido a diversas facilidades que uma IDE pode proporcionar, como: wizards para criação de conexões, keyspaces e tabelas, abas para se trabalhar com múltiplos servidores, gerenciamento de scripts, destaque de palavras reservadas, entre outros. O cqlsh, por sua vez, é mais utilizado por quem tem familiaridade com a linha de comando e não está muito a fim de abrir uma IDE pesada na sua máquina. Como facilidade, o cqlsh tem o recurso de *tab completion*, bastante útil a quem está acostumado com a “telinha preta”.

Modelagem - Desnormalizar é preciso

Quando se fala de modelagem de dados em bancos NoSQL, normalmente temos que deixar de lado quase tudo que é considerado boa prática no mundo relacional. No Cassandra não é diferente, e mais, várias das boas práticas da modelagem relacional são consideradas anti-padrões.

Nesse banco a normalização dos dados é considerada um destruidor de performance. Portanto, quase sempre é melhor desnormalizar para escalar a base. Por isso, não se preocupe tanto com a repetição dos dados ou com a quantidade maior de escritas que isso provoca. O Cassandra sabe lidar muito bem com essa situação.

Outra diferença é que nos bancos relacionais o foco da modelagem são as tabelas (entidades), onde a partir das mesmas diversos relacionamentos são obtidos através de joins e chaves estrangeiras. Já as boas práticas do Cassandra instruem a guiar sua modelagem baseado nas consultas. Assim, um padrão recomendado é ter uma tabela por consulta. Por exemplo, se sua aplicação precisa consultar Usuários por nome e por login, serão criadas duas tabelas: *usuario_por_nome* e *usuario_por_login*, ambas com os mesmos dados (desnormalização).

O problema da normalização e do foco em entidades é que essas técnicas acabam distribuindo os dados de forma inadequada, e para um banco como o Cassandra, isso pode significar ter que consultar vários nós do cluster para encontrar a informação, o que pode acarretar um grande problema de desempenho. As recomendações apresentadas visam minimizar ao máximo o acesso a múltiplos nós.

Partition Key

Todas as tabelas do Cassandra precisam definir uma chave denominada Partition Key. Esta tem como principal utilidade determinar em qual nó do cluster um dado será armazenado e

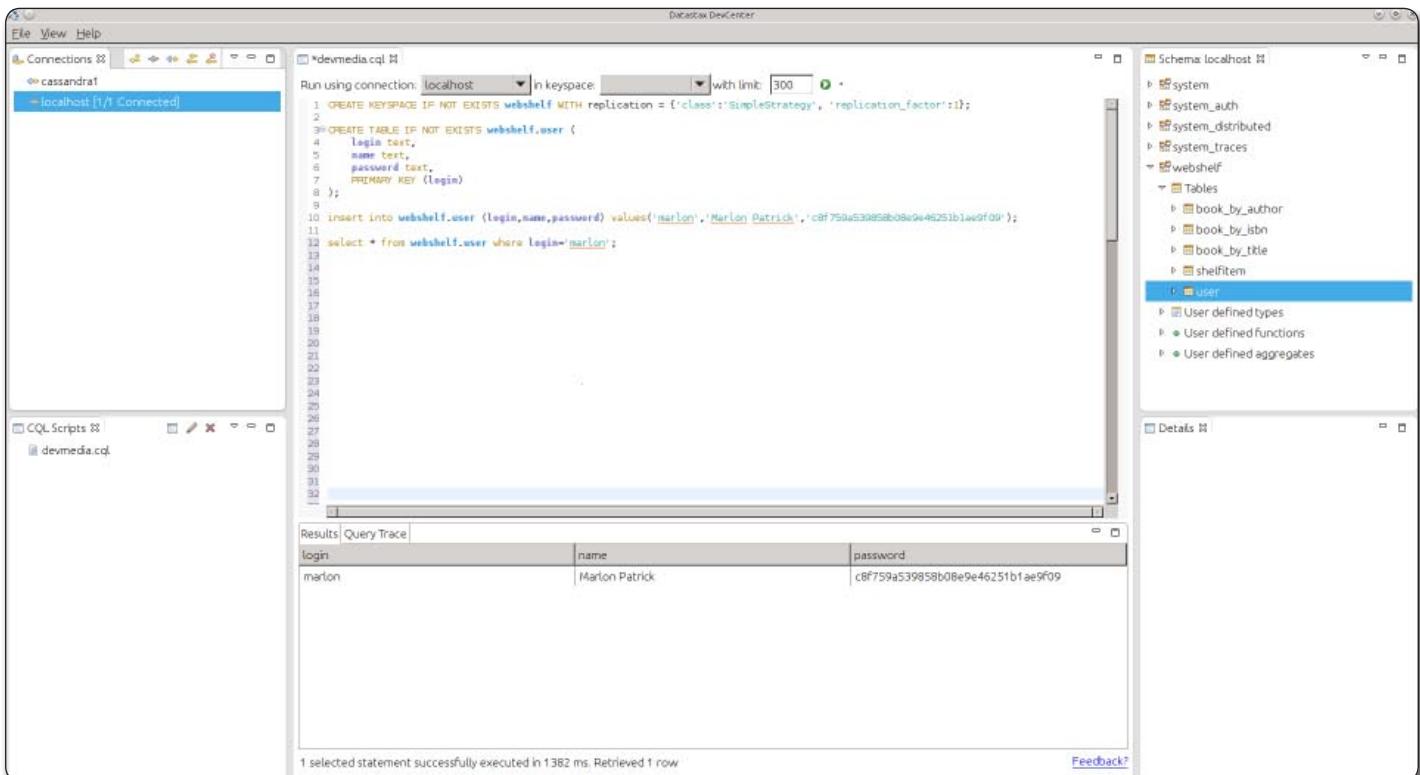


Figura 3. Executando comandos CQL através do DevCenter

trata-se de um conceito fundamental a todos que lidam com essa base de dados.

No que se refere à modelagem, a partition key tem relação direta com os filtros de uma consulta. Isso porque no Cassandra qualquer query precisa filtrar a tabela no mínimo pelas colunas que compõem sua partition key. A lógica dessa restrição é que sem a partition key o Cassandra não tem como saber em qual nó a informação está armazenada.

Neste ponto vale ressaltar que não se deve confundir partition key com primary key. Por exemplo, digamos que uma tabela definiu sua chave primária da seguinte forma:

PRIMARY KEY (*user_login*, *status*, *book_isbn*)

Nesse cenário, a primary key é composta pelas colunas *user_login*, *status* e *book_isbn*, enquanto a partition key se resume à primeira coluna, que no caso é *user_login*. As demais são conhecidas como clustering columns.

Clustering Column

Outro importante aspecto do Cassandra são as Clustering Columns. Essas colunas fazem parte da primary key, mas não da partition key. No exemplo apresentado anteriormente, as colunas *status* e *book_isbn* seriam as clustering columns.

A função dessas colunas é determinar a ordenação pela qual os dados serão organizados no disco para uma determinada partition key. É como uma ordenação padrão. Assim, no exemplo supracitado, uma vez que a tabela foi filtrada pela coluna *user_login* (partition key), os dados apresentados estarão ordenados pelas colunas *status* e *book_isbn*, mesmo que não se use um ORDER BY. Essa ordenação padrão ainda pode ser definida na criação da tabela como ASC ou DESC para cada uma das clustering columns. Como essa ordenação já é garantida no momento de armazenar a informação no disco, existe um enorme ganho de desempenho, pois o banco de dados não precisa fazer isso em memória para cada consulta.

Vale informar que o Cassandra só aceita ordenação (ORDER BY) baseado nas clustering columns. Deste modo, a ordenação dos dados para uma consulta pode modificar a forma como modelamos as tabelas. Isto porque o nosso objetivo deve ser executar o SELECT sem precisar especificar uma cláusula ORDER BY e mesmo assim sempre obter os dados na ordem desejada.

Outra maneira que as clustering columns podem afetar a modelagem é que essa ordenação padrão também irá trazer ótima performance nos filtros por intervalos, por exemplo, um período de data. Assim, se você perceber que sua consulta irá precisar desse tipo de filtragem, é fundamental escolher bem as clustering columns.

Distribuição - A chave para a escalabilidade horizontal

Um dos pontos fortes do Cassandra é a sua arquitetura distribuída com suporte a diversas configurações e tamanhos de cluster, desde um único nó a até centenas de nós, como acontece em algumas empresas como eBay, Netflix e Apple.

Essa distribuição é feita de forma automática, não necessitando que desenvolvedores e arquitetos se preocupem em implementar algum tipo de sharding via aplicação. Um componente conhecido como partitioner é o responsável por essa tarefa.

Um partitioner basicamente é uma função que gera um token (hash) a partir da partition key e então distribui os dados entre os nós do cluster baseado nesse token de maneira uniforme e transparente para o desenvolvedor. Em outras palavras, cada nó é responsável por um range compreendido pelo token.

Exemplo de distribuição de dados

Para exemplificar o funcionamento do mecanismo de distribuição de dados, vamos supor que exista uma tabela chamada *pessoas_por_nome*, que tem como partition key o campo *Nome*. Dessa forma, essa coluna será usada pelo particionador para gerar os hashes sempre que uma nova linha for inserida na tabela. Para ilustrar melhor essa situação, apresentamos na **Tabela 1** alguns valores para a coluna *Nome* e os seus respectivos hashes, que foram gerados por um partitioner hipotético.

Nota

O Cassandra oferece algumas opções de particionadores, sendo o Murmur3Partitioner a opção padrão e mais recomendada. Esse particionador gera tokens de 64 bits que englobam um range de -263 a 263-1. Para mais detalhes, acesse Apache Cassandra Product Guide (veja o endereço na seção [Links](#)).

Partition Key	Hash
José	-2245462676723223822
Maria	7723358927203680754
João	-6723372854036780875
Isabel	1168604627387940318

Tabela 1. Partition keys e seus respectivos hashes

Agora, imagine que o cluster dessa aplicação é composto por quatro máquinas, como exemplificado na **Figura 4**, e que cada um dos nós é responsável por armazenar os dados de um determinado range do hash gerado pelo partitioner. Por exemplo, nesse cluster o nó C armazenará as linhas que tenham um hash de 0 a 46116860118427387903.

Considerando o conjunto de dados da **Tabela 1** sendo inseridos num cluster com a configuração da **Figura 4**, teríamos uma distribuição dos dados conforme a **Tabela 2**. Assim, a linha que tem a coluna *Nome* igual a João seria armazenada pelo nó A, pois o partitioner gerou um hash para essa partition key o qual fica dentro do intervalo da máquina A.

Replicação - A mágica por trás da alta disponibilidade

No Cassandra a replicação de dados é algo tão natural como transações são para bancos de dados relacionais. Ela pode acontecer num mesmo data center, em múltiplos data centers ou em múltiplas zonas de cloud. Isso vai depender de sua estratégia de replicação.

Nó	Ínicio range	Fim range	Partition Key	Hash
A	-9223372036854775808	-4611686018427387903	João	-6723372854036780875
B	-4611686018427387904	-1	José	-2245462676723223822
C	0	4611686018427387903	Isabel	1168604627387940318
D	4611686018427387904	9223372036854775807	Maria	7723358927203680754

Tabela 2. Distribuição dos dados no cluster

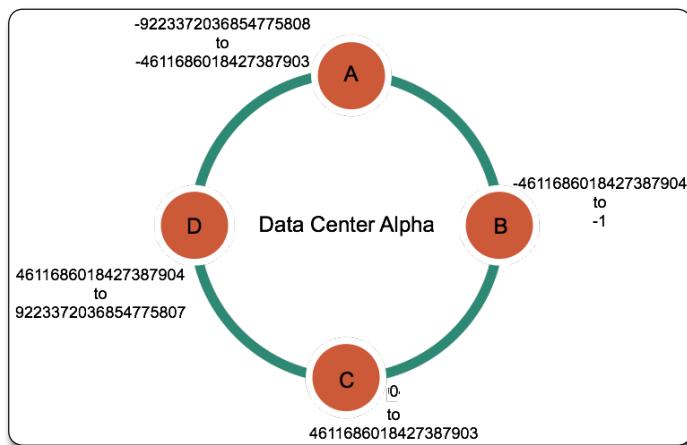


Figura 4. Nós do cluster e seus respectivos ranges - Adaptado de: DataStax

A configuração mais importante quando se fala de replicação é o replication factor. Ele é configurado na criação de cada keyspace (similar a um database no PostgreSQL) e sua função é definir o total de cópias de uma linha no cluster, fazendo com que cada cópia resida em um nó diferente. Se o replication factor é 1, então só haverá uma cópia por linha no cluster. Se o replication factor for 3, então quer dizer que para cada linha haverá três cópias no cluster.

Uma vez que o número de cópias é definido através do replication factor, ainda é possível parametrizar o processo de replicação para indicar como essas cópias serão distribuídas em diferentes data centers. Por exemplo, para um replication factor 4, pode-se armazenar duas cópias no data center 1, uma cópia no data center 2 e uma cópia no data center 3.

Dentre os principais benefícios que a replicação proporciona destaca-se a tolerância a falhas e o aumento da confiabilidade da aplicação.

Consistência - Nem forte, nem fraca: Tunável

Um tema quase certo em todas as discussões sobre NoSQL é o Teorema CAP. No próprio site da DevMedia existem diversos artigos que abordam o assunto. Por isso, apresentaremos aqui apenas um pequeno resumo, mostrado no **BOX 2**.

No que se refere ao Teorema CAP, o Cassandra é um sistema AP, proporcionando alta disponibilidade e tolerância à partição. Isso significa que os dados eventualmente ficarão consistentes em todas as réplicas, mas também podem, em determinadas janelas de tempo, ficar inconsistentes. Para minimizar essa situação, o Cassandra estende o conceito de consistência eventual, oferecendo o que chama de consistência tunável.

BOX 2. Teorema CAP

Resumidamente, esse teorema explica que num sistema distribuído é impossível ter, ao mesmo tempo, consistência, disponibilidade e tolerância à partição. A seguir, apresentamos uma breve explanação sobre cada uma dessas propriedades:

Consistência: implica que todos os clientes sempre visualizarão os mesmos dados;

Disponibilidade: se o cliente pode se conectar a um nó, então este nó deve ser capaz de ler e escrever dados;

Tolerância à Partição: significa que o cluster continuará trabalhando mesmo que ele seja dividido por quebras na comunicação da rede. Por exemplo, um cluster composto de dois data centers deverá funcionar mesmo que esses data centers não consigam se comunicar.

Tunable Consistency

Consistência tunável nada mais é do que a capacidade de configurar o grau de consistência desejado, seja num nível mais global ou em nível de operação (select, insert, delete, update). Dessa forma, numa operação mais crítica para a aplicação o desenvolvedor pode setar a consistência para um nível forte, e numa operação menos importante, pode usar consistência fraca. Isso faz com que o Cassandra haja tanto como um sistema AP (*Availability e Partition Tolerant*) quanto como um sistema CP (*Consistency e Partition Tolerant*).

Alguns exemplos de nível de consistência disponíveis são:

- **ONE:** basta um único nó responder à requisição que a resposta é retornada para o cliente;
- **QUORUM:** para o cliente obter a resposta será necessário que um quórum de nós retorne, onde, o quórum é igual a FATOR REPLICACAO/2 + 1;
- **ALL:** antes de retornar ao cliente, todas as réplicas deverão responder. Vale ressaltar que “todas as réplicas” equivale ao fator de replicação, o que não quer dizer todas as máquinas do cluster.

Além destes existem vários outros níveis de consistência que podem resultar numa diversidade de maneiras de controlar esse aspecto de uma determinada operação, o que traz grande flexibilidade para os usuários do Cassandra. Para conhecer todas as opções disponíveis acesse Apache Cassandra Product Guide (veja a seção **Links**).

No que se refere à operação de leitura, o dado mais recente de todos os nós consultados será retornado ao cliente. Em seguida, caso haja algum nó desatualizado, ele será sincronizado em background (*read repair*). A **Figura 5** demonstra um exemplo de leitura com consistência QUORUM. Note que o nó 10 é o coordenador da operação, o nó que se comunica diretamente com o cliente. Os nós 1 e 6 possuem a mesma informação, sendo que o

nó 6 foi escolhido para responder à requisição. Já o nó 3 está com a informação desatualizada e por isso receberá um read repair. O **BOX 3** explica como o Cassandra verifica qual dado é mais atual que outro.

Nas operações de escrita, a consistência determina quantos nós devem gravar a informação até que um retorno de sucesso seja dado ao cliente, no entanto, as réplicas ainda serão determinadas de acordo com o replication factor (RF). Por exemplo, digamos que o RF seja 3 e a operação tenha consistência ONE. O que irá acontecer é que assim que o primeiro nó responder com sucesso o cliente já será notificado de que a operação foi bem-sucedida, mas o Cassandra continuará a operação nas outras duas máquinas para completar a quantidade de cópias definidas no RF.

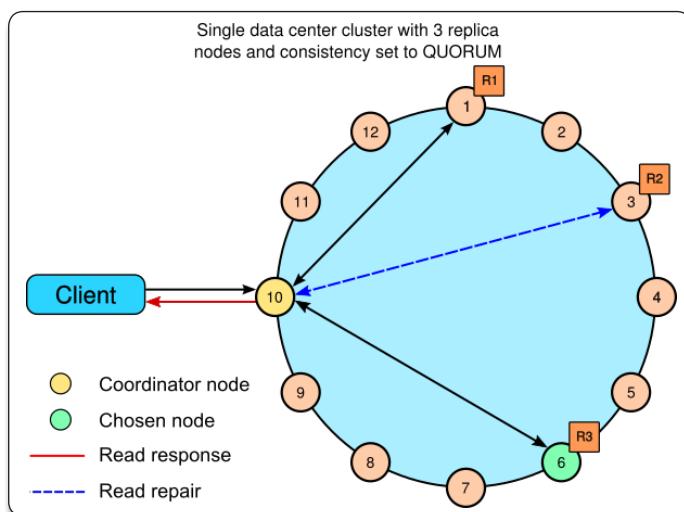


Figura 5. Funcionamento de uma operação de leitura com consistência QUORUM -

Fonte: DataStax

BOX 3. Last Write Wins

Para cada coluna de uma tabela no Cassandra, além do seu valor também é armazenado um timestamp correspondente à última atualização do campo. Assim, caso haja uma atualização concorrente numa coluna, a mais recente é a que prevalecerá. Essa é uma técnica de resolução de conflito conhecida como Last Write Wins. É através desse timestamp também que o Cassandra checa se a informação de um nó é mais recente do que a que está em outro para retorná-la nas consultas, bem como para disparar um read repair aos nós desatualizados.

Gerenciamento de transações - Sai ACID entra lightweight

O Cassandra não suporta transações ACID, mas especificamente, não suporta o "C" (Consistência) já que não contempla joins, integridade referencial ou mecanismos como commit e rollback. A ausência dessas features, no entanto, é compensada pela alta disponibilidade e escrita extremamente rápida que esse banco oferece.

Com relação às demais características, embora sejam um pouco diferentes do que acontece em bancos relacionais, elas existem: as escritas são duráveis (armazenadas num dispositivo não volátil como um disco rígido ou SSD), o isolamento é suportado em nível

de linha (uma operação numa linha só é visível a outros usuários quando a mesma é completada) e a atomicidade também é oferecida em nível de linha (a inserção ou atualização de colunas de uma mesma linha são tratadas como uma única operação de escrita).

Lightweight Transactions

Em bancos de dados relacionais uma arquitetura master-slave é uma estratégia comum para se implementar um cluster. Por outro lado, o Cassandra usa uma arquitetura do tipo peer-to-peer, na qual qualquer nó do cluster tem o mesmo papel, ou seja, todos podem receber escritas e leituras.

Por conta dessa abordagem, as operações de escrita no Cassandra podem ter problemas com requisições concorrentes que atualizam a mesma informação. Apesar da consistência tunável poder resolver muitos desses problemas, ainda há casos críticos em que se precisa ter uma maior "regulação" entre essas requisições concorrentes.

A **Figura 6** relata uma operação concorrente para criar um usuário. Nesse exemplo o usuário só deve ser criado se não houver nenhum outro com o mesmo login (id). No entanto, o que acontece é que as duas operações criam o usuário com o mesmo login e a segunda requisição acaba sobreescrivendo a informação da primeira. Esse cenário é conhecido como race condition.

É com intuito de resolver esse problema que o Cassandra desenvolveu o que chama de lightweight transaction. Uma lightweight transaction permite garantir que uma sequência de operações será executada sem interferência de outras. Algo semelhante ao nível de isolamento serializável oferecido por bancos de dados relacionais. Veja alguns exemplos na **Figura 7**.

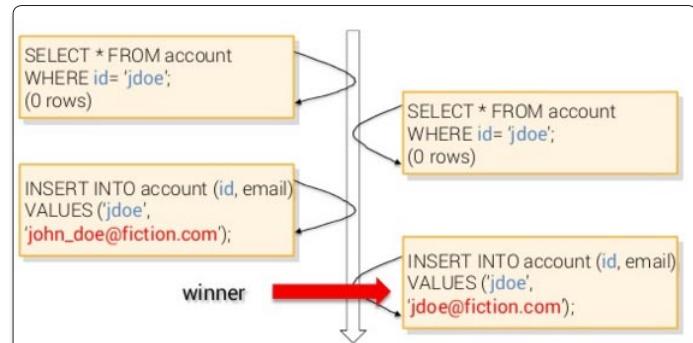


Figura 6. Exemplo de race condition – Fonte: FinishJUG

```
INSERT INTO webshelf.account (id, email)
VALUES('jdoe', 'john_doe@fiction.com')
IF NOT EXISTS;

UPDATE webshelf.account set email = 'jdoe@fiction.com'
WHERE id='jdoe'
IF email = 'john_doe@fiction.com';
```

Figura 7. Exemplo de uso de lightweight transaction

Tenha em mente que lightweight transactions podem impactar bastante na performance do Cassandra, visto que checar se um registro já existe em diversas réplicas distribuídas pela rede é uma operação custosa. Por isso, lightweight transactions devem ser utilizadas com cuidado e em poucos casos.

Cassandra na prática

Agora que você tem uma visão geral do que é o Cassandra e como funcionam vários aspectos importantes da arquitetura, vamos ao que interessa: a parte prática!

Ao longo deste tutorial vamos construir uma aplicação web (WebShelf) que imita o Amazon Shelfari, o qual permite aos usuários criar uma prateleira online de livros identificando o que já foi lido, o que se está lendo no momento e o que se pretende ler (vide **Figura 8**).

Para isso, vamos usar algumas tecnologias, onde destacam-se: Apache Cassandra 2.2.3, WildFly 9, PrimeFaces 5.3, Cassandra DataStax Driver 2.1.8, DataStax DevCenter 1.4.1 e Java EE 7.

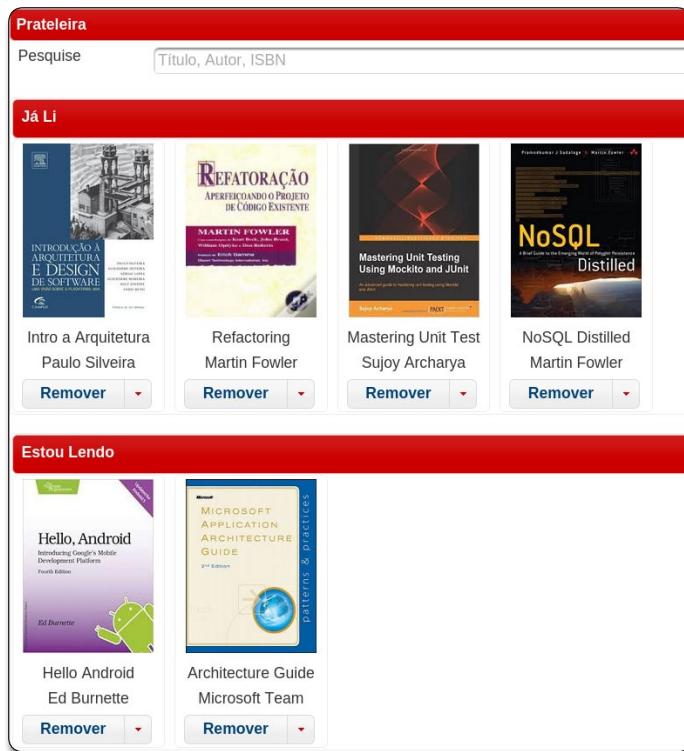


Figura 8. Prateleira de livros do WebShelf

Montando o ambiente

O tutorial foi desenvolvido utilizando o Eclipse Mars, mas você pode optar por qualquer IDE que achar melhor. Com a IDE definida, para colocar o Cassandra para rodar basta fazer o download, descompactar o arquivo, entrar na pasta `bin` e executar o comando `cassandra -f` para Linux ou `cassandra.bat -f` para Windows.

Para conectar no Cassandra a ferramenta escolhida foi o DevCenter. Baseado no Eclipse, para iniciá-lo basta fazer o download da versão mais adequada para o seu sistema operacional, descompactar

e executá-lo através do arquivo `DevCenter` dentro da pasta raiz. No decorrer do tutorial será mostrado como criar uma conexão.

Já o ambiente de execução da aplicação será o WildFly 9. Para quem não conhece esse servidor, ele é a evolução do JBoss AS com uma nova nomenclatura adotada pela Red Hat a partir da versão 8 e que implementa o Java EE 7. Como não faremos nenhuma configuração extra para este tutorial, basta realizar o download do mesmo e descompactá-lo. Para iniciar o servidor você pode usar a linha de comando. Assim, é só entrar na pasta `bin` e executar o script `standalone.sh` (Linux) ou `standalone.bat` (Windows). No entanto, durante o desenvolvimento recomenda-se configurar a IDE para poder gerenciá-lo de um ambiente centralizado. No Eclipse Mars o adapter do WildFly 9 já vem instalado por padrão. Portanto, basta acessar a view `Servers` e criar uma nova instância.

Configurando o projeto

Neste tópico iremos preparar a maior parte das configurações da aplicação de modo a nos concentrar posteriormente no código que mais interessa a este artigo. Portanto, vamos às configurações.

Criando o projeto Maven

O primeiro passo é criar o projeto no Eclipse, momento no qual informaremos que vamos utilizar o Maven como ferramenta de build e gerenciador de dependências. O projeto se chamará `webshelf` e terá dois módulos: `webshelf-business` e `webshelf-web`. A **Figura 9** apresenta essa estrutura.

O módulo `webshelf-business` terá, basicamente, a camada de negócio e a camada de acesso a dados. Já o módulo `webshelf-web` conterá o código relacionado à camada web da aplicação. Assim, ao rodar o build do projeto, o módulo `webshelf-web` irá gerar um arquivo WAR que possuirá, entre outras libs, o JAR referente ao módulo `webshelf-business`.

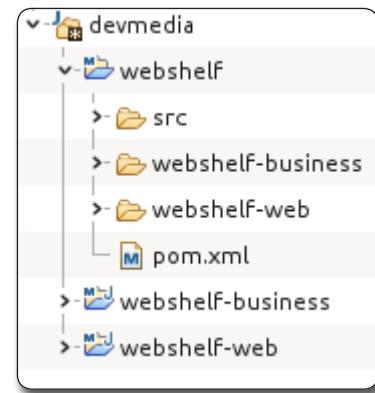


Figura 9. Estrutura do projeto

Configurando os poms

O pom do projeto principal, demonstrado na **Listagem 1**, terá as configurações que serão compartilhadas entre os dois módulos, por exemplo, dependências comuns ao módulo business e ao módulo web, como é o caso do CDI e Hibernate Validator. Além disso, para garantir que utilizaremos as mesmas APIs e versões

Como usar o Apache Cassandra em aplicações Java EE - Parte 1

contidas no WildFly, adicionamos uma dependência ao BOM (*Bill of Materials*) do WildFly 9. Assim você só precisa declarar as APIs do Java EE 7 que irá usar, sem se preocupar com versões ou escopo, pois estas configurações já estão definidas no BOM.

Listagem 1. Configuração do arquivo webshelf/pom.xml.

```
<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</groupId>
  <artifactId>webshelf</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>
  <modules>
    <module>webshelf-business</module>
    <module>webshelf-web</module>
  </modules>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <version.jboss.bom>9.0.1.Final</version.jboss.bom>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.wildfly.bom</groupId>
        <artifactId>jboss-javaee-7.0-wildfly</artifactId>
        <version>${version.jboss.bom}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>javax.enterprise</groupId>
      <artifactId>cdi-api</artifactId>
    </dependency>

    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
    </dependency>
  </dependencies>
</project>
```

Já as configurações e dependências dos módulos business e web, por sua vez, dizem respeito apenas a eles mesmos. Por exemplo, uma dependência declarada no módulo web não estará disponível no módulo business (vide **Listagens 2 e 3**). Isso é bastante útil para evitar que as camadas lógicas da aplicação se misturem, afinal, para que você precisa de classes do JSF na sua camada de negócio?

Portanto, os dois poms são simples, contendo basicamente as declarações de dependências que serão explicadas no decorrer do tutorial. Uma observação importante é que no *webshelf-web/pom.xml* existe uma declaração de dependência ao módulo business. O intuito disso é fazer com que o WAR do módulo web contenha na sua pasta *libs* o JAR do módulo business.

Listagem 2. Configuração do arquivo webshelf-business/pom.xml.

```
<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>
  <parent>
    <groupId>br.com.devmedia</groupId>
    <artifactId>webshelf</artifactId>
    <version>1.0.0</version>
  </parent>
  <artifactId>webshelf-business</artifactId>
  <packaging>ejb</packaging>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-ejb-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <ejbVersion>3.2</ejbVersion>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.jboss.spec.javax.ejb</groupId>
      <artifactId>jboss-ejb-api_3.2_spec</artifactId>
    </dependency>
    <!-- Driver do Cassandra -->
    <dependency>
      <groupId>com.datastax.cassandra</groupId>
      <artifactId>cassandra-driver-core</artifactId>
      <version>2.1.8</version>
    </dependency>
    <!-- API de object-mapping para o driver Cassandra -->
    <dependency>
      <groupId>com.datastax.cassandra</groupId>
      <artifactId>cassandra-driver-mapping</artifactId>
      <version>2.1.8</version>
    </dependency>
  </dependencies>
</project>
```

Ativando o CDI

Para que seja possível utilizar o CDI no projeto, será necessário criar o arquivo *beans.xml* nos dois módulos. No módulo business, o arquivo deve ficar localizado na pasta *src/main/resources/META-INF*, enquanto no módulo web deve ficar na pasta *src/main/webapp/WEB-INF*.

Listagem 3. Configuração do arquivo webshelf-web/pom.xml.

```
<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>
<parent>
  <groupId>br.com.devmedia</groupId>
  <artifactId>webshelf</artifactId>
  <version>1.0.0</version>
</parent>
<artifactId>webshelf-web</artifactId>
<packaging>war</packaging>

<build>
  <plugins>
    <plugin>
      <artifactId>maven-war-plugin</artifactId>
      <version>2.6</version>
      <configuration>
        <warName>webshelf</warName>
      </configuration>
    </plugin>
  </plugins>
</build>

<repositories>
  <repository>
    <id>prime-repo</id>
    <name>PrimeFaces Maven Repository</name>
    <url>http://repository.primefaces.org</url>
    <layout>default</layout>
  </repository>
</repositories>

<dependencies>
  <dependency>
    <groupId>br.com.devmedia</groupId>
    <artifactId>webshelf-business</artifactId>
    <version>1.0.0</version>
    <type>ejb</type>
  </dependency>
  <dependency>
    <groupId>org.jboss.spec.javax.servlet</groupId>
    <artifactId>jboss-servlet-api_3.1_spec</artifactId>
  </dependency>
  <dependency>
    <groupId>org.jboss.spec.jsf.faces</groupId>
    <artifactId>jboss-jsf-api_2.2_spec</artifactId>
  </dependency>
  <dependency>
    <groupId>org.apache.deltaspike.modules</groupId>
    <artifactId>deltaspike-jsf-module-api</artifactId>
    <version>1.5.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.deltaspike.modules</groupId>
    <artifactId>deltaspike-jsf-module-impl</artifactId>
    <version>1.5.0</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>5.3</version>
  </dependency>
  <dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>blitzer</artifactId>
    <version>1.0.10</version>
  </dependency>
</dependencies>
</project>
```

A **Listagem 4** apresenta o conteúdo desse arquivo, que deve ser o mesmo nos dois módulos.

Listagem 4. Configuração do arquivo beans.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd" bean-discovery-mode="all">
</beans>
```

Preparando o web.xml

Como se trata de uma aplicação web, não poderia faltar o *web.xml*. Esse arquivo deve ficar no módulo web em *src/main/webapp/WEB-INF* e é apresentado na **Listagem 5**. Nele temos algumas configurações que merecem atenção. A primeira delas é o mapeamento do servlet JSF (o Faces Servlet). O intuito dessa configuração é mapear o servlet para qualquer URL terminada com *.xhtml, que é a extensão usada nas páginas JSF do projeto.

Outra configuração é a definição do tema do PrimeFaces através do parâmetro **primefaces.THEME**. No nosso exemplo optamos por utilizar o **blitzer**. Caso você prefira outro, basta alterar esse parâmetro com o nome do tema escolhido e incluir a dependência no *webshelf-web/pom.xml*. Para saber os temas que o PrimeFaces disponibiliza, basta acessar *PrimeFaces Web Site* ([Links](#)).

Além do tema, outra configuração relacionada ao PrimeFaces é a ativação da API de validação client-side, disponível desde a versão 4 (**primefaces.CLIENT_SIDE_VALIDATION**). Através dessa feature é possível realizar diversas validações via JavaScript, evitando que seja feita uma requisição desnecessária ao servidor. O melhor é que para os casos padrões de validação, como **required**, **length** e **range**, você não precisará escrever nenhum código, o que é aplicável também às validações padrões da API Bean Validation. Para mais detalhes, veja *PrimeFaces Web Site* na seção [Links](#).

Acessando o Cassandra via Java

Para estabelecer a conexão com o Cassandra será utilizado o driver da DataStax, empresa que presta um serviço de suporte

Como usar o Apache Cassandra em aplicações Java EE - Parte 1

para o Cassandra e tem sido uma das principais mantenedoras desse banco. Além desta, existem diversas outras opções para se conectar com o Cassandra via Java, como o Hector ou Astyanax, mas essas APIs foram construídas inicialmente em cima da interface Thrift, a qual nas versões mais atuais do Cassandra está em desuso. Portanto, é preferível o uso de uma API que trabalhe em cima do CQL e seu protocolo nativo.

Listagem 5. Código do arquivo web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
  <context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>blitzer</param-value>
  </context-param>
  <context-param>
    <param-name>primefaces.CLIENT_SIDE_VALIDATION</param-name>
    <param-value>true</param-value>
  </context-param>
  <context-param>
    <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
    <param-value>true</param-value>
  </context-param>
  <context-param>
    <param-name>javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_
      AS_NULL</param-name>
    <param-value>true</param-value>
  </context-param>
</web-app>
```

O driver DataStax também oferece diversas features que o tornam uma forma extremamente recomendada de se comunicar com o Cassandra, a saber: chamadas assíncronas, balanceamento de carga configurável, failover transparente, entre outras.

Para começar, vamos criar a classe **CassandraCluster**, conforme a **Listagem 6**. Essa classe será responsável por gerenciar a comunicação com o Cassandra encapsulando os objetos do driver de modo a centralizar qualquer configuração relacionada ao banco de dados.

Neste código, a anotação **@Singleton** define a classe como um EJB Singleton. Já a anotação **@TransactionAttribute** foi colocada para sinalizar que esse EJB não suporta controle de transação.

Isso porque não existe uma maneira padronizada de se criar um datasource num servidor de aplicação Java EE que gerencie conexões do Cassandra e, portanto, não há como tirar proveito dessa feature da especificação EJB. Além disso, o Cassandra não suporta transações ACID, tendo seu próprio conceito e protocolo de gerenciamento de transações que não se encaixam no modelo oferecido pelo EJB.

Listagem 6. Classe para comunicação com o Cassandra (módulo webshelf-business).

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

import java.util.HashMap;
import java.util.Map;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Singleton;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;

import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.PreparedStatement;
import com.datastax.driver.core.Session;
import com.datastax.driver.mapping.MappingManager;

@Singleton
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class CassandraCluster {

    private Cluster cluster;
    private Session session;
    private MappingManager mappingManager;
    private Map<String, PreparedStatement> preparedStatementCache =
        new HashMap<>();

    @PostConstruct
    private void init() {
        cluster = Cluster.builder().addContactPoint("localhost").build();
        session = cluster.connect();
        mappingManager = new MappingManager(session);
    }

    @PreDestroy
    private void destroy() {
        session.close();
        cluster.close();
    }
}
```

Regras de utilização do driver DataStax

Aqui é necessário observar quatro regras básicas ao se trabalhar com o driver:

- Deve haver apenas uma instância da classe **Cluster** para cada cluster que sua aplicação precisar conectar. Essa instância precisará existir durante todo o ciclo de vida da aplicação;
- Deve haver apenas uma instância da classe **Session** por keyspace, ou, uma instância única para toda a aplicação, onde os comandos terão que especificar o keyspace. Essa instância precisa existir durante todo o ciclo de vida da aplicação;
- Caso seja necessário executar um mesmo statement repetidas vezes, é fortemente recomendado que se use **PreparedStatement**;

- Onde for adequado, faça uso de batches para reduzir round-trips na rede e também obter operações atômicas.

Para implementar os pontos 1 e 2 optamos por fazer de **CassandraCluster** um EJB Singleton. Dessa forma, vamos garantir que apenas uma instância dessa classe irá existir durante todo o ciclo de vida da aplicação. Consequentemente, acontecerá o mesmo com os campos **cluster** e **session**, que são inicializados e fechados através dos callbacks **@PostConstruct** e **@PreDestroy**, os quais serão executados apenas uma vez seguindo o ciclo de vida de um EJB Singleton.

O atributo **preparedStatementCache** está relacionado com o item 3. Como o WebShelf irá executar determinados statements de maneira repetida, a aplicação irá usar **PreparedStatement** em vários pontos. No entanto, para otimizar o uso de **PreparedStatement** é recomendado fazer um cache desse tipo de objeto para evitar que o Cassandra tenha que “re-preparar” statements repetitivos. Se você não fizer isso, verá alguns warnings como esse:

```
Re-preparing already prepared query "SELECT...". Please note that preparing the same
query more than once is generally an anti-pattern and will likely affect performance.
Consider preparing the statement only once.
```

O atributo **mappingManager**, por sua vez, é responsável por prover a funcionalidade de object-mapping, algo como um JPA bem mais simplificado. A função da classe **MappingManager** é criar Mappers para as entidades da aplicação e esses Mappers serão encarregados diretos por executar comandos como **insert** e **select** baseados em objetos, ao invés de comandos CQL.

Cada **Mapper** criado é guardado num cache interno da classe **MappingManager** e cada um tem também um cache interno de **PreparedStatement**s para as operações que já executou. Dessa forma, esses caches evitam re-preparar statements que serão executados várias vezes e, portanto, nos ajudam a implementar também o item 3 das regras de utilização do driver. Para que esse esquema de caches funcione, é recomendado que haja apenas uma instância do tipo **MappingManager** para cada instância de **Session**.

Nota

As classes **Cluster**, **Session**, **MappingManager**, **Mapper** e **PreparedStatement**, todas do driver DataStax, são thread-safe e assim podem ser compartilhadas por múltiplas threads.

Cadastro de Usuário

A primeira funcionalidade do projeto WebShelf será o Cadastro de Usuário, o qual permitirá que qualquer pessoa se inscreva no site para usufruir de seus serviços.

Esta funcionalidade consistirá de uma tela simples, feita em JSF 2.2 com apoio do PrimeFaces 5.3. A página usará alguns recursos dessa biblioteca para prover melhor responsividade de seus componentes. Vale ressaltar que alguns desses recursos de responsividade podem ser usados desde a versão 5.1, quando o

time do PrimeFaces aumentou os esforços para melhor gradativamente essa característica da biblioteca.

Essa tela irá invocar um método no controller, que por sua vez irá delegar para um método de negócio o qual será responsável por fazer algumas validações e então inserir o registro no Cassandra.

Mapeando a entidade de Usuário

No WebShelf um usuário terá login, nome e senha, sendo que o login deve ser único. Usaremos essa entidade para exemplificar a API de object-mapping (API OM) do driver Cassandra, como pode ser visualizado na **Listagem 7**. Essa API permite que o desenvolvedor tenha algumas features similares, embora bem menos poderosa, a um ORM como o Hibernate.

Listagem 7. Código da classe User (módulo webshelf-business).

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

import java.io.Serializable;
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.NotBlank;

import com.datastax.driver.mapping.annotations.PartitionKey;
import com.datastax.driver.mapping.annotations.Table;

@Table(keyspace = "webshelf", name = "user")
public class User implements Serializable {

    private static final long serialVersionUID = 1L;

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

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

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

    //constructors

    //getters and setters

    public void setClearPassword(String password) {
        this.password = encryptPassword(password);
    }

    public static String encryptPassword(String clearPassword){
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            md.update(clearPassword.getBytes());
            return new BigInteger(1, md.digest()).toString(16);
        } catch (NoSuchAlgorithmException exception) {
            throw new RuntimeException("Ocorreu um erro ao tentar criptografar a senha,"+exception);
        }
    }
}
```

É importante esclarecer que a maior parte do desenvolvimento com esse driver será utilizando comandos CQL diretamente, mas em alguns casos o object-mapping pode facilitar a implementação de cenários mais simples e por isso decidimos demonstrá-lo neste tutorial.

Para usar a API OM é preciso anotar a classe com `@Table` informando o nome do keyspace e da tabela. Além disso, a anotação `@PartitionKey` deve ser colocada no campo que representa a partition key da tabela (no caso de `User`, o campo `login`). Outras anotações úteis também estão disponíveis como, por exemplo, `@Column`, para informar o nome da coluna caso não seja igual ao nome do atributo na classe, e `@Transient`, para ignorar um campo no mapeamento. Como essa API usa métodos acessores para recuperar e gravar valores nos campos, também é necessário ter os respectivos `get`s/`set`s. Por fim, temos também a anotação `@NotNull`, a qual faz parte do Hibernate Validator (uma implementação da especificação Bean Validation) e como o nome diz, não permite que o campo fique em branco (nulo ou vazio).

Criando a tabela de usuário no Cassandra

Como visto anteriormente, podemos executar comandos CQL no Cassandra pelo cliente de linha de comando `cqlsh`, mas, para este tutorial iremos utilizar a ferramenta gráfica DevCenter.

A partir deste ambiente, na view `Connections` é possível criar uma nova conexão informando um nome qualquer para identificar a conexão e um ou mais hosts que fazem parte do cluster. Esse wizard pode ser visto na **Figura 10**. Como a configuração padrão do Cassandra não solicita senha, não é necessário fornecer nenhuma credencial.

O script disponibilizado na **Listagem 8** cria o keyspace da aplicação e também a tabela de usuário. Para executar esses comandos no DevCenter basta usar a tecla de atalho `Alt+F11` ou clicar no botão de execução no editor CQL. Também é possível criar esses dois objetos através de wizards que o próprio DevCenter oferece. Para isso, acesse o menu `File > New` e escolha a opção desejada.

O keyspace é como um database em bancos de dados como o PostgreSQL. É nele que serão definidas as tabelas e demais objetos do banco que porventura sejam necessários. No CQL de criação do keyspace existem dois pontos importantes que podem ser observados.

O primeiro é o `replication_factor`, que, como já informado em seções anteriores, define em quantos nós diferentes será armazenada a mesma informação. Como o nosso exemplo usará apenas um nó, o replication factor será de 1, pois não é uma boa prática ter esse valor maior do que o número de nós, além do que, pode gerar diversos erros.

O segundo ponto é a classe que representa a estratégia de replicação. No nosso exemplo será usada `SimpleStrategy`, que é indicada apenas para cenários onde existe um único data center. Caso haja mais de um DC deve-se usar `NetworkTopologyStrategy`. Para mais detalhes, consulte Apache Cassandra Product Guide na seção **Links**.

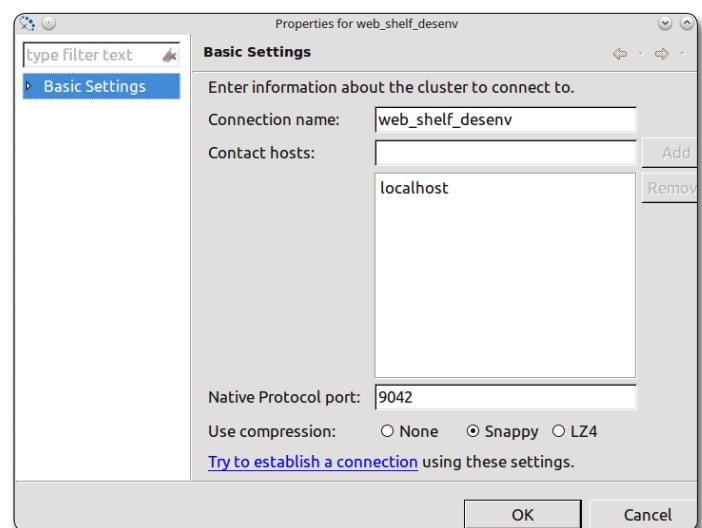


Figura 10. Criação de conexão no DevCenter

No que se refere à criação da tabela, não há muito o que explicar, visto a sintaxe ser bem semelhante ao SQL. O script simplesmente cria a tabela `user` dentro do keyspace `webshelf`, caso essa tabela já não exista, e define o campo `login` como sendo a primary key da tabela. E como o campo `login` é o primeiro campo da primary key, ele também será usado pelo Cassandra como partition key da tabela.

Listagem 8. Código para criação da tabela de usuário.

```
CREATE KEYSPACE IF NOT EXISTS webshelf WITH replication =  
{'class':'SimpleStrategy','replication_factor':1};  
  
CREATE TABLE IF NOT EXISTS webshelf.user (  
    login text,  
    name text,  
    password text,  
    PRIMARY KEY (login)  
);
```

Implementando o template padrão para as páginas JSF

Antes de criar a tela do cadastro de usuário, a primeira coisa a fazer é criar um template padrão para as telas do WebShelf, conforme a **Listagem 9**. O objetivo com isso é colocar o código que é comum a todas as páginas nesse template e fazer uso do mesmo nos demais XHTMLs. Assim, evita-se duplicação de código, aumenta-se o reuso, melhora-se a padronização do layout da aplicação, entre outros benefícios.

O que esse template faz é basicamente dividir a tela em duas partes: uma área para o menu e outra que ocupa a maior parte da tela para o conteúdo das páginas. Isso é feito através dos componentes `p:layout` e `p:layoutUnit`. Caso seja necessário, esses componentes podem dividir a tela em cinco partes: `header` (north), `footer` (south), `left` (west), `right` (east) e `center`.

A meta tag `viewport` serve para informar ao browser como controlar as dimensões e escalas da página. Em outras palavras, é através dessa tag que o browser pode gerar uma visualização responsiva baseado no tamanho da tela do dispositivo.

Listagem 9. Template padrão para telas do WebShelf (WEB-INF/templates/default.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">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:p="http://primefaces.org/ui">

<h:head>
  <title>WebShelf</title>
  <meta name="viewport" content="user-scalable=no, width=device-width,
    initial-scale=1.0, maximum-scale=1.0"/>
</h:head>

<h:body>
  <f:view encoding="UTF-8" contentType="text/html" locale="pt_BR">
    <p:layout fullPage="true">
      <p:layoutUnit id="layoutWest" position="west" header="Menu"
        collapsible="true" collapsed="true" rendered="#{renderedMenuBar}">
        <h:form preprendId="false">
          <p:menubar>
            <p:menuItem value="Home" outcome="home"/>
            <p:menuItem value="Logout" action="#{loginController.logout}"/>
          </p:menubar>
        </h:form>
      </p:layoutUnit>

      <p:layoutUnit id="layoutCenter" position="center"
        header="#{mainContentTitle}">
        <p:messages autoUpdate="true" globalOnly="true"/>
        <ui:insert name="mainContent"/>
      </p:layoutUnit>
    </p:layout>
  </f:view>
</h:body>
</html>
```

Note ainda que a área do menu irá iniciar sempre ocultada (**<p:layoutUnit collapsed="true">**) e poderá ser expandida pelo usuário (**<p:layoutUnit collapsible="true">**). O template também define o parâmetro **renderedMenuBar**, para identificar se o menu será ou não renderizado.

Já a parte central da tela será reservada para o conteúdo específico das demais páginas. Essa área é definida pela tag **<p:layoutUnit position="center">** e faz uso do parâmetro **mainContentTitle** como título do painel principal da aplicação. Além disso, possui um componente de renderização de mensagens do PrimeFaces para exibir mensagens globais do JSF (sem associação com nenhum componente da tela), que é útil, por exemplo, para alertar o usuário em caso de algum erro ou mensagem informativa. Por fim, a tag **ui:insert** determina o local onde o conteúdo de cada um dos XHTMLs será inserido no template.

O Apache Cassandra é um banco de dados NoSQL que vale a pena se conhecer para ter como alternativa aos tradicionais bancos de dados relacionais. Isso decorre do fato de ele prover uma série de facilidades e condições essenciais para uma arquitetura distribuída em nível de banco de dados.

Como desenvolvedor, a visão aqui apresentada dá uma base conceitual do assunto que o deixa apto a se aprofundar no “Planeta Cassandra”. No entanto, vale ressaltar que é importante conhecer

um pouco mais à fundo a parte prática, tema que será abordado com mais detalhes no próximo artigo.

Outro aspecto a ser observado é que o Cassandra tem suas aplicações, mas não é uma bala de prata que irá resolver todos os problemas relacionados a banco de dados. Assim, é interessante saber identificar em quais cenários essa tecnologia se encaixa melhor e para quais ela não é uma boa solução. Por exemplo, foi informado que ele não tem suporte a transações ACID. Deste modo, se sua aplicação precisa disso, provavelmente o Cassandra não será uma boa opção. Por outro lado, caso seu software esteja precisando escalar linearmente, distribuir dados pela rede e ter tolerância a falhas, o Cassandra será um forte candidato.

Para identificar esses cenários, a melhor opção é ler sobre cases de empresas que já estão usando esse banco e entender quais problemas ele resolveu. Existem diversos relatos desse tipo no site Planet Cassandra (veja a seção **Links**). Da mesma forma, você pode ler sobre empresas que deixaram o Cassandra e adotaram outra solução para aprender quando ele não é tão eficiente.

Autor



Marlon Patrick

marlon.patrick@mpwtecnologia.com - marlonpatrick.info
Bacharel em Ciência 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.



Links:

NoSQL Distilled.

<http://martinfowler.com/books/nosql.html>

Apache Cassandra Product Guide.

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

DataStax Dev Blog.

<http://www.datastax.com/dev/blog>

Planet Cassandra.

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

Cassandra introduction at FinishJUG.

<http://pt.slideshare.net/doanduyhai/cassandra-introduction-at-finishjug>

Cassandra Modeling Kata.

<https://github.com/allegro/cassandra-modeling-kata>

PrimeFaces Web Site.

<http://www.primefaces.org>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Alta produtividade em Java com Spring e AngularJS

Saiba como aumentar sua produtividade utilizando ferramentas e dicas a serem aplicadas em projetos Java que façam uso de Spring e AngularJS

Hoje em dia, para atender uma tarefa de criar um cadastro de clientes, por exemplo, o mesmo profissional poderá atuar em diversas frentes, como: realizar ajustes no banco de dados, criar serviços, controladores e testes, e customizar a interface visual. Como se não bastasse, o desenvolvedor precisa, ainda, garantir a qualidade do código e manter uma boa produtividade, para que não ocorram atrasos na entrega do software. Contudo, ser produtivo em um ambiente com diversos pontos a se preocupar não é uma tarefa fácil.

Em cenários como este, para enfrentar tantos desafios, um dos primeiros passos é conhecer bem a arquitetura do projeto, os frameworks e os padrões adotados pela equipe. No entanto, isso pode ser um grande problema, pois nem todos os desenvolvedores possuem a mesma facilidade para construir telas e estilizar layouts como têm para programar do lado servidor, por exemplo. E a mesma situação ocorre para desenvolvedores com perfil voltado para design de front-end. Deste modo, quando surge a necessidade de trabalhar onde não é o mais comum, a produtividade e a qualidade costumam cair consideravelmente. Não que seja por falta de capacidade, mas pelo fato do profissional ter focado seus estudos apenas no back-end ou no front-end.

Pensando nisso, com o intuito de aumentar a produtividade e a qualidade da codificação de seus sistemas, tanto no back-end quanto no front-end, serão abordadas no decorrer deste artigo dicas de ferramentas e boas práticas

Fique por dentro

Este artigo apresenta sugestões de frameworks, ferramentas e boas práticas para aumentar a produtividade em projetos Java, sendo útil para profissionais que já atuam com desenvolvimento web, iniciantes e equipes que utilizam metodologias ágeis. Para aqueles que estão buscando melhorar sua produção com qualidade, encontrarão neste artigo orientações que podem ser aplicadas desde a criação da estrutura do projeto, passando pela camada de persistência, criação de serviços REST, até a criação das telas utilizando AngularJS e HTML5.

de implementação utilizando Java e AngularJS. Esta solução para servidor e cliente vem se tornando um padrão para aplicações em diversos projetos devido à eficiência, escalabilidade e crescimento de mercado dessas tecnologias.

Tecnologias a seu favor

Hoje em dia há uma grande oferta de tecnologias, linguagens, ferramentas e frameworks no mercado para facilitar o trabalho de um profissional de tecnologia da informação. Uma grande parte dessa oferta está disponível na rede sob licença open source, podendo ser utilizada ou até melhorada por qualquer pessoa que esteja interessada. Porém, essa grande diversidade de opções pode gerar muitas dúvidas, o que leva a escolhas de tecnologias que não se adequam à solução proposta e consequentemente provocam atraso no desenvolvimento do projeto.

Para que não ocorram escolhas indevidas, o desenvolvedor deve buscar conhecer melhor os pontos positivos e negativos de cada tecnologia ou ferramenta que está interessado em utilizar. Lembre-se que não existe uma solução “bala de prata” a ser aplicada em todos os casos. A escolha ideal é aquela em que os pontos negativos encontrados pela equipe podem ser contornados de tal forma que não atrapalhe o andamento das atividades e seus pontos positivos sejam suficientes para justificar a decisão.

Arquitetura x Design x Solução

Por mais que o conceito de arquitetura de software seja difícil de ser compreendido, é de grande importância ter ciência do quanto esse conhecimento pode afetar o andamento de um projeto. Segundo Martin Fowler, “o termo arquitetura envolve a noção dos principais elementos do sistema, as peças que são difíceis de ser mudadas. Uma fundação na qual todo o resto deve ser construído”. Já Ralph Johnson, do GoF, define como sendo “o conjunto de decisões de design que gostaríamos de ter feito no começo do projeto”.

Partindo dessas afirmações, ao pensar na construção de uma aplicação, saiba que as primeiras decisões de tecnologias, ferramentas e padrões irão afetar todo o resto do projeto, pois irão impactar no dia a dia de trabalho dos desenvolvedores e futuramente serão difíceis de serem alteradas, caso sejam encontrados problemas ou pouca aderência da equipe à solução. As escolhas corretas nessa etapa facilitarão as correções de problemas ou alterações futuras, caso surjam.

Depois de realizadas as escolhas das principais tecnologias, as próximas decisões serão sobre o design da aplicação. Para o arquiteto de software Neal Ford, as decisões de design são feitas em cima do que foi decidido para a arquitetura como, por exemplo, o comportamento do sistema e a interatividade com o usuário em casos de sucesso ou falha. Dessa maneira a arquitetura e design da aplicação devem ser pensados de forma consoante para dar ao projeto vantagens não só no momento de manutenção, mas também possibilitar a criação de um ambiente de desenvolvimento produtivo.

Esse artigo irá sugerir e dar dicas sobre tecnologias e ferramentas a serem utilizadas em projetos web, sendo preferencialmente aplicadas durante as decisões arquiteturais. Porém, a adoção do que será apresentado dependerá do contexto e das reais necessidades do projeto. É preciso compreender que a cultura, preferências e disponibilidade de tempo de cada equipe de desenvolvimento devem ser consideradas para que sejam feitas boas escolhas. Além disso, é importante que todos estejam cientes e de acordo com a forma de trabalho que será conduzida, pois mantendo o mesmo padrão se torna mais fácil realizar a manutenção do código, menos oneroso resolver problemas de desfalcque na equipe, será alcançada uma curva de aprendizado menor para novos integrantes, entre outras possibilidades.

Por que utilizar Spring e AngularJS?

Quando se constrói um software é muito comum fazer uso de frameworks para facilitar o trabalho de desenvolvimento.

Imagine um profissional que irá implementar uma nova funcionalidade em uma aplicação ter que pensar em quais padrões deve utilizar, lidar com o gerenciamento de transações, o protocolo HTTP, a conversão de objetos e ainda controlar conexões com o banco de dados. Certamente o trabalho seria muito mais complexo e demorado. Deste modo, utilizar frameworks para encapsular a complexidade dessas tarefas é uma excelente opção.

Um bom primeiro passo para aumentar a produtividade é optar pelo framework Spring. Solução open source, criada com o objetivo de simplificar a programação em Java, possibilita a construção de aplicações que antes só eram possíveis fazendo uso de EJBs. O Spring é uma ótima opção para projetos web, pois viabiliza módulos que facilitam e muito o desenvolvimento, como: Spring Data, para a camada de persistência; Spring Security, para cuidar da segurança da aplicação; e Spring REST, para simplificar a criação de web services. Além disso, também possui funcionalidades para injecção de dependências, programação orientada a aspectos e controle de transações, fazendo dele um poderoso aliado do desenvolvedor server-side.

Para o desenvolvimento front-end, é possível contar com outro aliado cheio de recursos, o AngularJS, solução em JavaScript mantida pelo Google. Este é um framework MVC construído para possibilitar a criação de diversos recursos na camada de aplicação, bem como abstrair da implementação complexidades como, por exemplo, AJAX em chamadas REST, controle de acesso a páginas, teste de código JavaScript, entre outras. Na prática, o AngularJS viabiliza a criação de tags – chamadas de diretivas – que são introduzidas em forma de atributos ou novos elementos, enriquecendo as funcionalidades das páginas HTML. Ademais, o framework permite uma ligação direta e bidirecional entre as camadas **model** e **view**, o que faz com que alterações em valores dos objetos JavaScript sejam aplicadas automaticamente nos elementos HTML, criando uma melhor experiência com conteúdos dinâmicos para o usuário.



Além do ganho em produtividade no desenvolvimento, o AngularJS e o Spring também possibilitam outras vantagens bastante úteis ao profissional, como uma documentação completa e rica em detalhes. No site oficial do AngularJS, por exemplo, existe um guia para os iniciantes que vai desde o design da interface, criação de controladores, serviços, filtros, diretivas, módulos e uso de rotas, até como realizar testes unitários, tudo com exemplos para facilitar a compreensão de como explorar corretamente os recursos.

Configuração do ambiente de desenvolvimento Java

Para que o trabalho flua e seja produtivo, o primeiro passo é ter um ambiente de desenvolvimento bem configurado. A primeira escolha a ser feita é a IDE a ser utilizada. O profissional deve escolher aquela na qual se sinta mais confortável, tenha maior familiaridade, viabilize bons recursos e conheça as teclas de atalhos mais úteis, para que seja otimizado o trabalho realizado no dia a dia.

Esse artigo não irá sugerir a adoção de nenhuma IDE, pois a melhor escolha vai depender das preferências de cada profissional. O mais importante é que ela seja compatível com o desenvolvimento em Java e dê suporte ao uso de ferramentas como o Git, para controle de versão, e Maven, para o gerenciamento do projeto e bibliotecas.

Tomada essa decisão, configure o cliente Git para controle de versão seguindo as orientações descritas no site oficial deste. Para o gerenciamento do projeto, utilize o Maven ou o Gradle. Para o banco de dados, opte pelo que atenda melhor as necessidades do projeto, seja ele relacional ou NoSQL. Contudo, a recomendação é que durante o desenvolvimento seja utilizado um banco em memória, pois tem a vantagem de ter o acesso mais rápido e menos burocrático.

Nota

No momento da escolha da IDE, leia artigos relacionados, analise o custo benefício caso ela não seja open source, veja opiniões de usuários a respeito de desempenho e bugs. Também verifique no site do desenvolvedor da IDE se ela possui suporte a ferramentas e frameworks mais conhecidos do mercado, pois isso irá facilitar bastante o seu trabalho e dos demais membros do projeto.

Spring Boot

Um framework para criação de aplicações web que vem crescendo muito é o Spring Boot. Ele foi criado com o intuito de aumentar a produtividade do desenvolvedor ao fazer com que conceitos como REST, HTTP e containers web fiquem mais simples de ser utilizados. Além disso, esta solução é construída em cima do Spring Framework, herdando, portanto, os benefícios de esconder a complexidade e deixar o trabalho mais fácil.

A equipe que implementou o Spring Boot construiu um site, chamado Spring Initializr (veja a seção [Links](#)), para ajudar na etapa inicial de criação do projeto web. Neste é possível descrever algumas informações do projeto, como as bibliotecas que deseja utilizar, e no final é gerado um arquivo ZIP baseado no Spring Boot com todas as configurações prontas para o Maven. O Spring Initializr cria um projeto com uma estrutura que segue basicamente o layout convencional dos projetos criados pela ferramenta de gerenciamento de dependências.

Caso já tenha um projeto em andamento e deseja apenas acrescentar o Spring Boot, não há problemas. Para fazer isso, é preciso adicionar o artefato `spring-boot-starter-parent` como `parent` e a dependência do Spring Boot Web no `pom.xml` principal da aplicação, como pode ser visto na [Listagem 1](#).

Para que o Spring consiga inicializar corretamente a aplicação, é preciso definir um ponto único de entrada. Isso é feito criando uma classe que chama o método estático `run()` da classe `SpringApplication` passando como parâmetro a própria classe criada. O nome da classe pode ser qualquer um, desde que ela receba a anotação `@EnableAutoConfiguration`, como pode ser visto na [Listagem 2](#). Com isso, o Spring Boot é capaz de inicializar a aplicação, configurá-la e executá-la em um contêiner web como o Tomcat, que já é integrado ao framework. Concluída a inicialização, também ficarão disponíveis para acesso os end-points que serão configurados adiante.

Além disso, o projeto não fica dependente de configurações extras da IDE, pois, para rodar a aplicação, basta executar a classe `ApplicationMain` ou utilizar o comando do Maven `mvn spring-boot:run`.

Spring Data

A recomendação para a camada de persistência é o uso do Spring Data. O principal objetivo do framework é padronizar e facilitar o uso de diferentes tecnologias de armazenar informações, como os bancos de dados relacionais e os NoSQL, encapsulando a complexidade das operações de leitura, escrita e alteração de dados, tirando do desenvolvedor a necessidade de implementar classes concretas para acesso aos repositórios.



Listagem 1. Código do arquivo pom.xml.

```
<groupId>com.exemplo</groupId>
<artifactId>meuProjeto</artifactId>
<version>0.0.1-SNAPSHOT</version>
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.1.BUILD-SNAPSHOT</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Listagem 2. Código da classe ApplicationMain.

```
@EnableAutoConfiguration
public class ApplicationMain {
    public static void main(String[] args) {
        SpringApplication.run(ApplicationMain.class);
    }
}
```

Seja qual for o tipo de base de dados utilizado, as classes de repositório da camada de persistência (também conhecidas como DAOs) geralmente implementam métodos para realizar o CRUD (*Create-Read-Update-Delete*) de uma entidade de domínio, juntamente com métodos específicos de consulta, ordenação ou paginação. Com o Spring Data, a criação desses objetos de “repositório” fica mais fácil, pois são disponibilizadas interfaces genéricas para serem utilizadas.

Para criar os repositórios utilizando o framework, é preciso primeiramente importar a dependência no *pom.xml*. Feito isso, basta criar uma interface para cada entidade de domínio estendendo a interface **JpaRepository**, como visto na **Listagem 3**. Ao fazer isso, os métodos de CRUD como **save()**, **findAll()**, **delete()**, já existentes na interface **JpaRepository**, serão herdados, economizando escrita de código. Ademais, apesar da interface ficar disponível como um bean injetável, é interessante anotá-la com **@Repository**, para que o Spring possa identificá-la como sendo do tipo repositório.

A criação das classes concretas com as queries será realizada em tempo de execução pelo framework.

Outra funcionalidade muito útil do Spring Data é voltada para a codificação de pesquisas específicas. Normalmente em um projeto que não utiliza um framework para esse tipo de auxílio, o desenvolvedor precisaria criar uma interface e definir os métodos de consulta que posteriormente seriam implementados em uma classe concreta. Com o Spring Data, ganha-se produtividade também para essas situações, sendo necessário apenas adicionar na assinatura do método na interface a anotação **@Query**, passando a consulta no formato JPQL, como está exemplificado na **Listagem 3**.

Para encapsular as chamadas que irão acessar os dados da tabela **Usuario** através da interface **UsuarioRepository**, é interessante criar um service anotando-o com a anotação do Spring **@Service**, como visto na **Listagem 4**. Ele receberá a injeção da interface criada

para acesso ao repositório, podendo assim fazer chamadas aos métodos para gravar, buscar e excluir dados do banco. Nessa classe também é preciso criar o controle de transação, o que pode ser feito anotando os métodos com **@Transaction**. Nesse ponto, pode ser que seja necessário realizar configurações diferentes para cada situação. Para isso é recomendado dar uma lida na documentação do Spring, para entender o que atende melhor. Um exemplo de service pode ser visto na **Listagem 4**.

Listagem 3. Código da interface UsuarioRepository.

```
@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

    List<Usuario> findByNome(Usuario usuario);

    @Query("select u.endereco from Usuario u where u.matricula = ?u")
    Endereco findByMatricula(String matricula);

    Usuario findByEmail(String email);
}
```

Listagem 4. Código da classe UsuarioService.

```
@Service
public class UsuarioService {

    @Inject
    private UsuarioRepository usuarioRepository;

    @Transactional
    public Usuario incluir(Usuario usuario) {
        return usuarioRepository.save(usuario);
    }

    public Usuario detalhar(Long id){
        return usuarioRepository.findOne(id);
    }
}
```

Spring REST

O Spring Framework também oferece suporte à criação de web services RESTful. A partir deste, é possível disponibilizar de forma rápida e prática os controladores para acesso via REST, utilizando anotações das APIs do Spring MVC.

Para que um serviço possa ser acessado por meio do protocolo REST, é necessário anotar a classe com **@RestController**. Em seguida, os métodos que irão tratar as requisições devem conter a anotação **@RequestMapping** com os atributos **value** e **method** definindo, respectivamente, o caminho de acesso para aquele serviço e o tipo de método HTTP como, por exemplo, GET, POST, PUT ou DELETE. Ao chamar o serviço o **DispatcherHandler** do Spring direcionará a requisição para a função que tenha anotado o método HTTP e a URL correspondente ao que foi solicitado.

Para que um método seja requisitado e receba dados passados como parâmetros pela URL, podemos utilizar anotações, como a **@PathVariable**. Esta é utilizada na declaração dos parâmetros da função que irá atender a requisição. Sua meta é capturar o valor na URL e converter para um tipo de dado que possa ser

manipulado no Java, como visto no método `detalhar()` da **Listagem 5**. Já a anotação `@RequestBody` tem um objetivo parecido, mas com ela é possível obter todo o conteúdo existente no corpo da requisição, podendo assim converter um objeto JSON inteiro, por exemplo, para uma entidade conhecida no Java, desde que os nomes dos atributos sejam compatíveis, conforme o método `incluir()`, também da **Listagem 5**.

Ao concluir o processamento da requisição o método irá responder através do uso da classe `ResponseEntity`. Esta é responsável por associar um objeto de retorno ao corpo da resposta HTTP, possibilitando ainda a inclusão de um código de status. Um exemplo de uso desses recursos pode ser visto na **Listagem 5**.

Listagem 5. Código da classe `UsuarioRestController`.

```
@RestController
@RequestMapping("/usuarios")
public class UsuarioRestController {

    @Autowired
    private UsuarioService usuarioService;

    @RequestMapping(value = "/", method = RequestMethod.POST)
    public ResponseEntity<Usuario> incluir(@RequestBody Usuario usuario) {
        Usuario banco = usuarioService.incluir(usuario);
        return new ResponseEntity(banco, HttpStatus.CREATED);
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public ResponseEntity<Usuario> detalhar(@PathVariable("id") Long id) {
        return new ResponseEntity(usuarioService.detalhar(id), HttpStatus.OK);
    }
}
```

Lombok, Guava e Orika para economizar código

Para auxiliar um pouco mais o desenvolvedor, há alguns projetos interessantes a utilizar durante o processo de implementação. O Lombok, por exemplo, é um projeto criado com o objetivo de reduzir a quantidade de código necessário, deixando as classes Java menos verbosas. Por meio de anotações, ele gera automaticamente os métodos que são conhecidos como “clichês”. Por exemplo, classes que representam entidades de banco possuem métodos getters e setters para acesso aos dados. Neste caso, com as anotações `@Getter` e `@Setter` é possível informar ao Lombok para criar estes métodos de todos os atributos existentes na classe, como pode ser visto na **Listagem 6**. Também é possível, apenas através de anotações, especificar construtores, métodos `equals()` e `hashCode()`, entre outros.

Listagem 6. Uso do Lombok na classe `Usuario`.

```
@Getter
@Setter
public class Usuario {
    private Long id;
    private String nome;
    private Integer matricula;
}
```

Para que o Lombok se integre ao projeto é preciso instalar a biblioteca por meio do Maven ou manualmente, assim como instalar o plug-in para a IDE que esteja utilizando, seja ela o Eclipse, NetBeans ou IntelliJ, o que pode ser feito seguindo as orientações no site dos desenvolvedores (veja a seção [Links](#)).

Outro projeto interessante para facilitar o dia a dia de um desenvolvedor Java é o Google Guava. Este contém um conjunto de classes utilitárias para evitar códigos repetitivos, além de deixar os métodos mais limpos e legíveis. Caso deseje utilizar o Guava no projeto, basta ir ao site oficial e ler a documentação (veja a seção [Links](#)). Dentre os destaques, o Guava possui o recurso de pré-condições (Preconditions), através do qual é possível fazer verificações nos objetos de maneira mais elegante. A partir disso, o próprio Guava irá cuidar de lançar exceções, caso a regra de checagem não seja atendida. Por exemplo, quando é preciso verificar se um valor não é nulo antes de realizar uma operação e retornar uma mensagem de erro caso ocorra uma exceção. Para isso, basta utilizar o método `checkNotNull()` e passar como primeiro parâmetro o objeto que se deseja verificar seguido da mensagem de erro, para o caso deste não atender a condição, conforme o exemplo:

```
Preconditions.checkNotNull(usuario.getMatricula(), "A matrícula do usuário é inválida.");
```

Em uma aplicação com arquitetura em camadas é comum que objetos da camada de apresentação possuam características diferentes das dos objetos que os representam na camada de domínio. Por exemplo: para apresentar todos os dados de um **Usuario** pode ser necessário obter informações presentes em outras entidades, como **Endereço** e **Contato**, e concentrá-las



em um único objeto. Para abstrair essas diferenças existe um padrão de projeto conhecido como DTO (*Data Transfer Object*, também chamado de *Value Object*), que faz uso de um objeto para transferir os valores dos atributos de um lado para o outro. Dessa maneira, reduzimos o acoplamento e possibilitamos que a camada de domínio continua livre de configurações extras. Para isso, no entanto, é preciso que os valores dos atributos sejam populados de um objeto do tipo DTO para a(s) entidade(s) que o representa(m) e vice e versa, obedecendo à hierarquia de domínio e às devidas conversões de tipos.

Para facilitar esse trabalho de conversão existe um framework que se destaca no mercado, o Orika. Com ele é possível viabilizar o mapeamento de objetos complexos com nenhuma ou poucas configurações, ficando ele responsável por realizar todo o processo difícil de converter valores de um objeto para o outro.

O primeiro passo para utilizá-lo é adicionar sua dependência ao Maven. Em seguida, é interessante criar uma classe para encapsular as definições dos tipos a serem convertidos e os métodos para fazerem as chamadas ao mapper do Orika, responsável por realizar essas conversões. Para utilizar este framework é preciso criar uma nova instância de **DefaultMapperFactory** e chamar o método **classMap()** passando a classe que representa a entidade, origem, e a classe DTO, o destino, para a qual os dados serão convertidos, como visto na **Listagem 7**. Nesse ponto também é possível especificar configurações extras, para casos em que a conversão requeira mais detalhes, como um atributo que representa uma lista de outra entidade ou quando é necessário adicionar um converter para transformar atributos de uma tipagem para outra totalmente diferente. Logo após, deve-se obter um **MapperFacade** chamando o método **map(objetoOrigem, objetoDestino)** para executar a conversão, como visto no exemplo na **Listagem 7**.

Listagem 7. Trecho de código da classe UsuarioAssembler.

```
MapperFactory mapperFactory = new DefaultMapperFactory.Builder().build();
MapperFacade mapper = mapperFactory.getMapperFacade();

private void configure(MapperFactory mapperFactory){
    mapperFactory.classMap(Usuario.class, UsuarioDTO.class)
        .field("endereco.loogradour", "endereco")
        .byDefault()
        .register();
    mapperFactory.classMap(Contato.class, ContatoDTO.class)
        .byDefault()
        .register();
}

public UsuarioDTO paraDto(Usuario entidade){
    configure(mapperFactory);
    return mapper.map(entidade, new UsuarioDTO());
}
```

Nos casos em que o objeto de destino tem atributos diferentes da entidade de origem, ou quando os nomes dos atributos são diferentes, recomenda-se utilizar o método **field(atributoOrigem, atributoDestino)** passando como parâmetros os nomes dos atributos diferentes para que o Orika consiga fazer a conversão

(**Listagem 7**). E nos casos em que há um atributo que representa outra entidade que também precisa de conversão, deve-se fazer a mesma configuração realizada anteriormente: acrescentar uma nova chamada ao método **classMap()** passando a classe de origem e a de destino desse atributo, como a classe **Contato** e **ContatoDTO**, respectivamente.

Além destes, há muitos recursos interessantes e bastante úteis para facilitar o desenvolvimento de um projeto. Para conhecê-los, acesse a documentação do framework, disponível no site oficial do Orika (vide [Links](#)).

Configurando o ambiente front-end com Node.js, Grunt e Bower

Da mesma forma que é necessário configurar o ambiente Java para desenvolver o lado servidor, também é preciso configurar o ambiente para implementação da parte cliente. Como a quantidade de código JavaScript produzido durante o desenvolvimento com AngularJS é de um tamanho considerável, é recomendado que a IDE dê suporte à programação JavaScript, HTML e CSS. Caso a mesma não viabilize tais recursos, procure uma na internet que possa atendê-lo de forma aceitável.

Nota

Esse artigo não irá abordar nenhuma IDE para desenvolvimento front-end, pois isso varia de conhecimento, recurso e preferência de cada profissional. Caso você ainda não tenha optado por uma, é recomendado que procure em fóruns, outros artigos e leia bastante para tomar a melhor decisão, pois a produtividade é proporcional à qualidade e à adaptação do desenvolvedor à ferramenta.

O segundo passo a ser executado é realizar a instalação do Node.js. Este é uma plataforma que, diferentemente de outras como o Apache Tomcat, é capaz de interpretar código JavaScript através do mecanismo V8 do Google Chrome, facilitando a construção de aplicações rápidas e escaláveis. Para a tecnologia V8 o Google criou um interpretador eficiente em C++ que pode ser integrado a qualquer aplicação, não sendo necessariamente restrito para ser usado em um navegador. O Node.js usa essa mesma tecnologia, porém voltada para aplicações no servidor, e em sua versão mais nova, já traz integrado um gerenciador de pacotes chamado NPM, recurso de grande ajuda para instalar ferramentas utilitárias ao projeto. Os passos para realizar a instalação e configuração do Node.js irá depender do sistema operacional utilizado. Para mais orientações e realizar o download deste, acesse o site oficial do projeto (vide [Links](#)).

A próxima ferramenta a ser instalada é o Grunt, solução útil para automatizar tarefas relacionadas ao código compilado para o lado cliente, como compilar todos os JavaScripts e os estilos para dentro de um arquivo, e realizar a minificação para gerar o build para produção, semelhante ao Maven. A instalação é feita via gerenciador de pacotes do Node.js, o NPM.

Por último, tem-se a instalação da ferramenta Bower. Desenvolvida para facilitar o gerenciamento de bibliotecas JavaScript, com ela

é possível controlar as versões e realizar o download da biblioteca desejada, semelhante ao que o Maven faz com bibliotecas Java. A vantagem é que o projeto cliente, que irá para o controle de versão, não precisará conter as bibliotecas, economizando espaço. O Bower realizará o download das dependências quando um novo ambiente for configurado, armazenando o conteúdo dentro da pasta *bower_components*. Ademais, a instalação do Bower é feita da mesma forma que o Grunt, através do NPM.

Caso todas essas ferramentas sejam uma sopa de letrinhas e o desenvolvedor não tenha conhecimento ou experiência com nenhuma delas, é recomendável utilizar uma ferramenta muito interessante para a criação de projetos web modernos, o Yeoman (veja o **BOX 1**). A vantagem do Yeoman é que ele contém, por padrão, as duas ferramentas supracitadas (Grunt e Bower) embutidas.

BOX 1. Projeto web com Yeoman

O principal objetivo desse artigo não é fazer com que os desenvolvedores passem a utilizar as ferramentas aqui mencionadas. O objetivo é sugerir e dar dicas para o desenvolvimento ágil ser aplicado com uma boa produtividade, mantendo o código estável, confiável e com boa manutenibilidade.

Para desenvolvedores que não estão familiarizados com programação para o lado cliente, mas desejam criar um software moderno, o Yeoman pode atender bem a demanda. Este é uma ferramenta muito útil para dar o primeiro passo, pois já fornece um stack completo para a estrutura do projeto, bastando apenas decidir como utilizá-lo de acordo com os requisitos.

O fluxo de trabalho do Yeoman é composto por três ferramentas que colaboram com o aumento da produtividade, a saber:

- YO – Gerador da estrutura de arquivos pré-configuradas do projeto;
- Grunt – Gerenciador de tarefas automatizado; e
- Bower – Gerenciador de dependências do projeto.

A instalação do Yeoman é realizada via gerenciador de pacotes do Node.js, seguindo as orientações do site oficial do framework (veja a seção **Links**). Depois de instalado, deve-se rodar o comando `npm install generator-angular` para criar um projeto web padrão utilizando o generator do Angular, contendo HTML5 Boilerplate, jQuery, Bootstrap e AngularJS. Logo após, é necessário inicializar a aplicação com o YO usando o comando `yo angular`. Feito isso, observe na estrutura de pastas criada que o projeto já contém muita coisa do que precisa. A pasta app, por exemplo, contém os arquivos do projeto como Controllers do Angular, Filters e Diretivas. A pasta *bower_components* armazena as dependências do projeto. Já a pasta *node_modules* contém os módulos instalados pelo Node.js. E os arquivos *bower.json* e *Gruntfile* possuem, respectivamente, as configurações das bibliotecas e as tarefas automatizadas, de forma semelhante ao *pom.xml* do Maven. A partir desse momento, para iniciar a aplicação basta executar o comando `yo angular`.

Como diferencial, o Grunt também disponibiliza um servidor local, iniciado através da execução do comando `grunt server`. Além disso, o projeto vem, por padrão, com um plug-in do Grunt chamado Livereload. Este permite o desenvolvimento fluir e ser muito mais produtivo, pois com qualquer alteração no código ele já compila e atualiza a tela, sem a necessidade de acionar o F5 ou reiniciar o servidor.

Aproveitando recursos do AngularJS

O AngularJS é um framework JavaScript mantido pelo Google e criado com a principal intenção de facilitar o desenvolvimento web, encapsulando as chamadas REST e simplificando o reuso de funcionalidades através da criação de diretivas. Uma das prin-

cipais diretivas utilizadas para reaproveitamento de código é a **ng-view**. Esta permite a construção de aplicações single-page.

Nota

Single-page é uma técnica de desenvolvimento para web em que todas as páginas da aplicação se concentram em apenas uma, gerando, assim, uma experiência de uso mais fluida ao usuário, semelhante à utilização de um software desktop.

Utilizando a diretiva **ng-view** o código HTML repetido em todo o sistema, como de menus, títulos, configurações de perfil, *logout*, etc., ficará disponível para todas as páginas sem a necessidade de duplicação de elementos, bastando para isso que esses códigos sejam definidos em um template principal. Para tanto, basta declarar a **ng-view** nesse template e informar ao módulo de gerenciamento de rotas do AngularJS, o `ngRoute`, qual elemento HTML ele deve chamar para compor e renderizar as páginas da aplicação. Um exemplo disso pode ser visto na **Listagem 8**.

Listagem 8. Código da página index.html.

```
<body>
<!-- Código do menu e outros elementos em comum a todas as páginas -->
<div class="corpoAplicacao">
    <!-- Local onde será renderizada as páginas da aplicação através do ng-view
        <div ng-view></div>
    </div>
</body>
```

Para que seja possível renderizar as páginas é necessário também configurar suas rotas no módulo principal da aplicação. Essa configuração é importante para que o Angular saiba qual arquivo HTML está sendo requisitado para poder exibi-lo na tag **ng-view**, assim como qual controlador será carregado para aquela rota, como visto na **Listagem 9**.

Listagem 9. Controlador JavaScript principal.js.

```
principal.config(function ($routeProvider) {
    $routeProvider
        .when('/', {
            redirectTo: '/principal'
        })
        .when('/usuario', {
            templateUrl: 'partials/usuario/manterUsuario.html',
            controller: 'UsuarioCtrl'
        })
        .otherwise({redirectTo: '/404'});
});
```

Deste modo, alcança-se, ainda, aumento de desempenho, pois quando houver mudança de tela o JavaScript responderá e carregará somente os dados e o HTML que forem alterados, em vez de recarregar todo o HTML novamente. Isso torna a aplicação mais rápida e ágil, pois menos dados serão acessados no servidor web. Além disso, também é possível criar uma diretiva que simples-

mente recebe a rota como parâmetro e redireciona a navegação da aplicação, permitindo a economia de código nos controladores. Dessa maneira não há necessidade de criar um método no controlador da página para apenas realizar a trocar de tela, como o botão “voltar”.

Além destes, existem outros recursos interessantes para ganho de produtividade com o AngularJS. Um desses recursos é o mecanismo de validação de formulários, disponibilizado através de atributos da tag **input** existentes no HTML5, como **type**, **pattern** e **required**. O atributo **type**, por exemplo, especifica o tipo de controle para a entrada de dados. Ao declará-lo com a opção **text**, como feito no código `<input type="text">`, indica-se que o elemento poderá receber qualquer tipo de texto.

Ademais, o atributo **type** aceita outros parâmetros que mudam a forma de validação sem a necessidade de gastar tempo codificando. Por exemplo, **type="email"** informa que o campo só aceita como entrada um e-mail válido. Existe, também, **type="number"**, **type="url"** e **type="tel"**, para validar números, URLs e números de telefone, respectivamente. Já o atributo **pattern** é utilizado para criar máscaras de entrada de dados a partir de expressões regulares. Dessa maneira é possível criar um padrão de entrada sem a necessidade de usar uma API ou desenvolver um método complexo. Ainda dentro do formulário, use nos elementos de **input** considerados obrigatórios o atributo **required**. Com esse atributo definido, é informado que ele é de preenchimento obrigatório. O HTML5 só permitirá o submit do form quando todas as validações criadas forem atendidas.

Por fim, ao criar um formulário com o elemento **form**, utilize o atributo **ng-submit** passando a função do Controller que será responsável por realizar a ação quando o formulário for submetido. Veja um exemplo na **Listagem 10**.

Listagem 10. Formulário em HTML5 – manterUsuario.html.

```
<form name="form" ng-submit="cadastrar(usuario)">

<label>Nome: </label><br/>
<input type="text" ng-model="usuario.nome" required>

<label>Idade: </label><br/>
<input type="number" ng-model="usuario.idade" required>

<label>Telefone (formato: xx-xxxx-xxxx):</label><br/>
<input type="tel" pattern="^\\d{2}-\\d{4}-\\d{4}$" ng-model="usuario.telefone" required>

<label>Email: </label><br/>
<input type="email" ng-model="usuario.email" required>

<input type="submit" value="Salvar">
</form>
```

Separando responsabilidades e encapsulando chamadas

Uma maneira de deixar o código JavaScript do AngularJS mais claro e limpo é quebrar os controllers por responsabilidades. Não é interessante que o mesmo controlador fique responsável por realizar diversas tarefas como listar os dados, realizar validações,

cálculos, fazer chamadas REST, entre outras possibilidades, não é mesmo? Quanto mais código com propósitos diferentes houver em um mesmo lugar, mais confuso é para realizar a manutenção e menor é a produtividade quando é necessário efetuar alguma alteração no código.

Dessa maneira, procure criar um controller para cada tipo de ação da tela como, por exemplo, **ListarUsuarioCtrl**, **NovoUsuarioCtrl** e **AlterarUsuarioCtrl**. Assim o código fica limpo e sucinto em cada controlador, contendo apenas o que ele realmente deve fazer.

Para trafegar dados de um controller para outro, faça uso de recursos que o AngularJS disponibiliza, como os listeners. Por exemplo, quando o usuário da aplicação selecionar um dos itens listados pelo controlador **LitarUsuarioCtrl** para ser alterado, ele irá delegar essa ação para o controller **AlterarUsuarioCtrl** através de um evento que ambos conhecem, passando o objeto a ser alterado por parâmetro. Tanto o método de disparo, quanto o que aguarda ser chamado podem ser vistos na **Listagens 11 e 12**.

Listagem 11. Exemplo de listener - ListarUsuarioCtrl.

```
//Função chamada ao clicar no item da lista
$scope.alterarUsuario = function (usuario) {
  $scope.$broadcast('alterarUsuario', usuario);
};
```

Listagem 12. Exemplo de listener - AlterarUsuarioCtrl.

```
$scope.$on('alterarUsuario', function (evento, usuario) {
  //Atende a requisição e altera o Usuário recebido
});
```



Nota

Promise ou promessa, no JavaScript, representa o resultado de uma operação que foi realizada. Ela pode ser utilizada para definir o comportamento do que fazer após uma operação ser bem-sucedida ou gerar uma falha. Dessa maneira é possível controlar a execução do método de sucesso ou de falha, garantindo que esse código será executado somente após a conclusão de toda a operação.

Outra dica interessante é criar um serviço genérico para manter encapsuladas as chamadas mais comuns ao servidor da aplicação como, por exemplo, as requisições de salvar, alterar, listar e excluir. Dessa forma, ao criar uma nova tela e novos controladores, será necessário apenas injetar esse serviço que foi criado contendo as chamadas ao servidor. Para que isso seja possível, o service genérico irá montar a URL da requisição utilizando o parâmetro que receber de quem irá utilizá-lo, como visto na declaração da variável **resource** na **Listagem 13**. É uma boa prática que esse parâmetro seja o nome da entidade que consumirá o serviço e que esse nome seja o mesmo especificado no web service correspondente que atende as requisições no servidor.

Listagem 13. Exemplo de serviço genérico de encapsulamento de chamadas REST - **ApiRestResource**.

```
angular.module('principal').factory('ApiRestResource', ['$resource', function ($resource) {
  return function (entidade) {
    var resource = $resource('rest/' + entidade + '/:id', {id: '@id'});
    this.salvar = function (objeto) {
      return resource.save([], objeto,
        function onSave() { //Mostra mensagem de sucesso },
        function onError(error) { //Mostra mensagem de erro }
      ).$promise;
    };
    return this;
  };
}]);
```

O serviço genérico deve ainda tratar a resposta do servidor, sendo ela de sucesso ou de erro, retornando seu resultado para o controlador que o utiliza. A maneira ideal de fazer isso é retornar uma **\$promise**, passando a responsabilidade de manipular os dados obtidos nas requisições para quem o chamou.

Com essa abordagem o ganho de produtividade é notável, pois toda a complexidade da comunicação, obtenção de parâmetros e tratamento das respostas fica transparente aos controllers, como observado na **Listagem 14**. Assim, ao instanciar o serviço genérico **ApiRestResource** para criar a comunicação com o servidor, basta passar o nome da entidade como parâmetro. Com isso, não será necessário implementar no controlador **AlterarUsuarioCtrl** as definições do **ngResource**.

Listagem 14. Exemplo da utilização do serviço genérico - **AlterarUsuarioCtrl**.

```
angular.module('principal').controller('UsuarioCtrl', ['$scope', 'ApiRestResource', function ($scope, ApiRestResource) {
  var usuarioResource = new ApiRestResource('usuarios');

  $scope.salvar = function (usuario) {
    usuarioResource.salvar(usuario).then(function () {
      //Omitido código a ser executado após sucesso.
    });
  };
}]);
```

Como verificado, o método disponibilizado no exemplo é o **salvar**. Este recebe como parâmetro o objeto proveniente do controlador responsável por realizar a ação de salvar. No exemplo dessa listagem, o módulo **ngResource** do AngularJS é utilizado para efetuar as requisições REST ao servidor através do objeto **\$resource**, que é injetado para executar essa tarefa.

Saiba que também é possível definir variáveis para serem utilizadas na requisição. No exemplo, é criado o parâmetro **id** na declaração da URL. Assim, caso o objeto que é enviado ao servidor possua um atributo com o mesmo nome **id**, o **ngResource** automaticamente preenche seu valor na URL, no local que foi definido. Dessa forma, quando o objeto passado como parâmetro possui o atributo **id** preenchido, a requisição será feita para outro caminho, podendo outro método responder no servidor.

Portanto, o aumento da produtividade através da economia de código nos controladores é considerável. Note que os métodos de consulta e alteração ficam bem reduzidos e mais claros para compreensão. Outra vantagem é a manutenção, que se tornará mais rápida e prática caso seja necessário alterar algo na forma como são feitas as chamadas.

Antes de tudo isso, no entanto, é preciso definir um padrão de nomenclatura e utilização da URL com os verbos para que o serviço genérico que encapsula as requisições e o servidor que atende as demandas sejam criados de forma coesa. Um padrão a seguir é utilizar os métodos já definidos no **ngResource**, como **save**, **get** e **delete**, para inclusão, obtenção e deleção de dados,

respectivamente, pois possuem seus verbos já configurados. Veja um exemplo na **Tabela 1**.

Um importante complemento para assegurar a qualidade do código é ter uma boa cobertura de testes unitários. Relacionado a isso, uma grande vantagem do AngularJS é que ele separa o código JavaScript dos elementos HTML. Assim, diferentemente do que acontece em muitos casos com scripts jQuery, os controladores interagem diretamente com o `$scope`, deixando o código livre de HTML, o que torna os testes muito mais fáceis de serem implementados.

URL	ngResource	Verbo	Descrição
/usuarios	query({isArray: true})	GET	Consulta todos os usuários
/usuarios	save([], objeto)	POST	Salva um novo usuário
/usuarios/{id}	get([], objeto)	GET	Consulta o usuário pelo ID
/usuarios/{id}	save([], objeto)	POST	Altera o usuário pelo ID
/usuarios/{id}	delete([], objeto)	DELETE	Delete o usuário pelo ID

Tabela 1. Comparação do padrão de URLs com métodos do ngResource e seus verbos

Para isso, é recomendado utilizar o Jasmine, um framework para testes unitários de JavaScript que atende muito bem ao propósito. Com ele é possível criar testes para cada método dos controllers, e assim pode-se testar as chamadas REST, cálculos e retornos esperados. Junto ao Jasmine, sugere-se adotar o Karma, uma ferramenta de test-runner que pode ser configurada para rodar os testes em browsers diferentes. Ambos instaláveis via gerenciador de pacotes do Node.js.

Ademais, é possível configurar uma tarefa no Grunt para chamar o Karma no momento de gerar o build para produção. Deste modo, serão disparados testes unitários antes dessa etapa, o que é muito útil para minimizar as chances de erros inesperados no ambiente de produção.

O Spring fornece um bom suporte para a criação de serviços e clientes REST. De forma simples, elegante e ágil, abstrai uma grande complexidade, facilitando e aumentando a produção dos desenvolvedores. Consequentemente, reduz o tempo e custo de projetos que demandam a utilização de web services. Do outro lado, para consumo desses serviços, o AngularJS auxilia na criação de interfaces mais ricas e bonitas sem aumentar a complexidade do código. Unindo esses dois frameworks, juntamente com outras ferramentas e tecnologias citadas, o ganho em produtividade pode ser considerável.

Mesmo que o projeto já esteja em andamento e não seja possível acatar tudo o que foi abordado, é recomendado avaliar a adoção

das ferramentas de baixo impacto, como Lombok e Guava, para economia de código.

Além disso, os assuntos aqui abordados também podem ser aplicados utilizando o Yeoman. Este será responsável por criar toda a estrutura do lado cliente com o Grunt, Bower e AngularJS já configurados. E para o lado servidor, considere experimentar o Spring Initializr para especificar um projeto Java com Maven e os módulos Spring Data, Spring REST e Spring Security já prontos para uso. Por fim, faça testes, crie protótipos, faça sugestões à equipe de trabalho, mas não deixe de conhecer as novidades existentes no mercado para aumentar a produtividade e a qualidade no desenvolvimento de projetos de software.

Autor



Rodrigo Engelberg

rodrigoengelberg@gmail.com

Analista de Sistemas, especialista em Arquitetura de Software, graduado em Sistemas de Informação pelo Instituto Federal de Goiás. Trabalha com desenvolvimento Java desde 2010. Entusiasta JavaScript e novas tecnologias para aplicações web.

Site pessoal: <http://www.rodrigoengelberg.com.br>.



Links:

Página do projeto Spring Initializr.

<https://start.spring.io/>

Página do projeto Lombok.

<https://projectlombok.org>

Página do projeto Guava.

<https://github.com/google/guava>

Guia oficial do projeto Orika.

<http://orika-mapper.github.io/orika-docs/>

Página oficial do Node.js.

<http://nodejs.org>

Página do projeto Yeoman.

<http://yeoman.io/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



DevOps e a Integração Contínua nas nuvens

Aprenda nesse artigo como turbinar a prática de DevOps automatizando a análise de seus projetos com o SonarQube

Em artigos publicados nas edições 147 e 148 da Java Magazine, abordamos alguns dos conceitos chave em DevOps, tanto sob a perspectiva do Desenvolvimento quanto de Operações.

Nestas ocasiões, estudamos plataformas e tecnologias bastante populares, como Apache Maven, Jenkins, Eclipse e Redmine, resultando na configuração de um ambiente tal qual o ilustrado na **Figura 1**.

Neste texto, revisaremos muitos dos tópicos descritos nos artigos citados a fim de adicionarmos a esta cadeia um importante passo: a análise contínua de qualidade. Para isso, usaremos uma ferramenta muito popular, chamada SonarQube, que nos permitirá realizar medições apuradas e constantes de aspectos essenciais de um projeto de software, com destaque para um conceito denominado Débito Técnico.

Ao longo de todo o tutorial que se segue, usaremos maciçamente serviços oferecidos sob a chancela da ‘cloud computing’. Todas as ferramentas serão provisionadas por meio de soluções de PaaS e SaaS, tais como OpenShift e GitHub. A principal motivação para isto é ilustrar, principalmente, os inúmeros benefícios dos quais se pode usufruir ao se substituir o modelo de infraestrutura *in-house*, tais como: redução significativa de custos e uma considerável melhoria em indicadores como produtividade, eficiência, disponibilidade, escalabilidade e segurança.

Nosso estudo ao longo deste texto será iniciado pela configuração de um ambiente de integração contínua, estabelecendo um canal fluido de comunicação entre um repositório de código-fonte do projeto – hospedado em uma conta no GitHub – e um servidor Jenkins – criado e hospedado na plataforma OpenShift. O objetivo será garantir que, a cada submissão de código realizada por um desenvolvedor, seja disparado um processo de build no servidor de integração contínua, sem a necessidade de intervenção manual.

Fique por dentro

Este artigo ajudará a entender, na prática, como funciona o processo de análise estática de código em projetos gerenciados sob o modelo da integração contínua. Ao longo do texto, seremos apresentados aos passos necessários para, a partir de ferramentas e plataformas como Jenkins, Git, Maven e SonarQube, medir continuamente a qualidade de um projeto de software e avaliá-lo sob perspectivas fundamentais, tais como a do débito técnico.

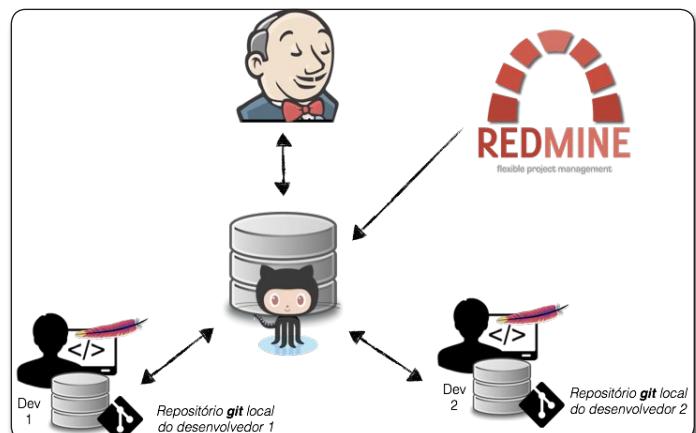


Figura 1. Estado inicial do projeto para este artigo

A Integração Contínua de Software

Um processo de integração contínua pode ser entendido como um conjunto de medidas que, quando adotadas, permitem que um software evolua de forma consistente e controlada. Tais medidas são:

- Gestão automatizada do ciclo de vida do software;
- Controle de versão do código-fonte;
- Automação de testes;
- Automação do processo de build.

O principal benefício desta prática é poder avaliar constantemente o impacto promovido por modificações na estrutura lógica de um projeto, seja por introdução de conteúdo novo ou adaptações em unidades de código-fonte pré-existentes. A partir deste modelo, uma equipe de desenvolvimento tende a aumentar significativamente o seu poder de reação a eventuais problemas que possam emergir ao longo do processo criativo, como conflitos de versão, falhas na execução de testes unitários, dentre outros. No trato popular, integração contínua é uma técnica que incorpora, na prática, a velha máxima de que ‘é preferível a prevenção ao remédio’.

Para seguir os passos desta primeira fase do tutorial, é necessário que o leitor crie a sua própria conta na OpenShift. O procedimento de cadastro é bem simples, e a página da plataforma pode ser encontrada na seção **Links**, ao final do artigo.

OpenShift e a linha de comando

A OpenShift é uma plataforma que oferece uma excelente ferramenta web de administração, com referências diretas para as aplicações hospedadas e informações essenciais sobre a natureza e o estado das mesmas. Entretanto, operar sistemas a partir de interfaces visuais pode ser um trabalho pouco eficiente, se comparado à flexibilidade e rapidez que um terminal oferece.

Neste sentido, a RedHat disponibiliza um poderoso utilitário chamado *rhc*, que utilizaremos em todas as fases do tutorial apresentado neste artigo. O processo de instalação está muito bem documentado na página oficial da OpenShift, e o link para este material está registrado na seção **Links**. Acompanharemos, nesta seção, o passo a passo da instalação e configuração do utilitário no computador utilizado para desenvolver toda a parte prática do artigo (um MacBook modelo 2009, rodando MacOS Yosemite, na versão 10.10.5).

O *rhc* é desenvolvido em Ruby. Por isso, o primeiro requisito para conseguirmos executá-lo é garantir que o Ruby esteja instalado. De acordo com a documentação oficial, a versão mínima a ser respeitada é a 1.8.7. No MacBook, tanto o Ruby quanto outras linguagens já são suportadas nativamente, não havendo a necessidade de realizarmos este procedimento. Para verificar a versão atualmente instalada no computador, usamos o primeiro comando destacado na **Listagem 1**. Observe que, neste caso, já tínhamos à disposição o Ruby configurado na versão 2.0.0p481, absolutamente aderente ao requisito que apresentamos a pouco.

O próximo passo é instalar o *rhc* propriamente dito. Para isto, diretamente em um terminal/prompt, executamos o comando *sudo gem install rhc*, também ilustrado a partir da **Listagem 1**. Ao rodar este comando, com privilégios de administrador, o *rhc* e suas dependências serão baixadas e instaladas, automaticamente e em um tempo relativamente curto. A seguir, basta que executemos outro comando, *rhc setup*, para nos conectarmos à nossa conta OpenShift e, então, passarmos a administrá-la via linha de comando. Assim que fizermos isso, o resultado será algo como o exibido na **Listagem 2**.

Listagem 1. Terminal – instalação da ferramenta *rhc* na máquina de desenvolvimento.

```
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ ruby -v
ruby 2.0.0p481 (2014-05-08 revision 45883) [universal.x86_64-darwin14]
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ sudo
gem install rhc
Password:
Fetching: net-ssh-2.9.2.gem (100%)
Successfully installed net-ssh-2.9.2
Fetching: net-scp-1.2.1.gem (100%)
Successfully installed net-scp-1.2.1
Fetching: net-ssh-gateway-1.2.0.gem (100%)
Successfully installed net-ssh-gateway-1.2.0
Fetching: net-ssh-multi-1.2.1.gem (100%)
Successfully installed net-ssh-multi-1.2.1
Fetching: archive-tar-minitar-0.5.2.gem (100%)
Successfully installed archive-tar-minitar-0.5.2
Fetching: highline-1.6.21.gem (100%)
Successfully installed highline-1.6.21
Fetching: commander-4.2.1.gem (100%)
Successfully installed commander-4.2.1
Fetching: httpclient-2.7.0.1.gem (100%)
Successfully installed httpclient-2.7.0.1
Fetching: open4-1.3.4.gem (100%)
Successfully installed open4-1.3.4
Fetching: rhc-1.38.4.gem (100%)
=====
=====
```

If this is your first time installing the RHC tools, please run 'rhc setup'

```
=====
Successfully installed rhc-1.38.4
Parsing documentation for net-ssh-2.9.2
Installing ri documentation for net-ssh-2.9.2
Parsing documentation for net-scp-1.2.1
Installing ri documentation for net-scp-1.2.1
Parsing documentation for net-ssh-gateway-1.2.0
Installing ri documentation for net-ssh-gateway-1.2.0
Parsing documentation for net-ssh-multi-1.2.1
Installing ri documentation for net-ssh-multi-1.2.1
Parsing documentation for archive-tar-minitar-0.5.2
Installing ri documentation for archive-tar-minitar-0.5.2
Parsing documentation for highline-1.6.21
Installing ri documentation for highline-1.6.21
Parsing documentation for commander-4.2.1
Installing ri documentation for commander-4.2.1
Parsing documentation for httpclient-2.7.0.1
Installing ri documentation for httpclient-2.7.0.1
Parsing documentation for open4-1.3.4
Installing ri documentation for open4-1.3.4
Parsing documentation for rhc-1.38.4
Installing ri documentation for rhc-1.38.4
10 gems installed
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$
```

Esta listagem nos traz algumas informações importantes, cujos detalhes serão analisados a partir de agora. O primeiro ponto se refere às credenciais de acesso à conta OpenShift. Em um primeiro momento, o utilitário nos questionará sobre esses dados para poder estabelecer uma conexão. Trata-se, portanto, de uma operação típica de login. Esses dados serão registrados e, a partir do primeiro acesso, estas serão as credenciais consideradas em todo novo comando executado via *rhc*, até que realizemos o logout a partir do comando *rhc logout*. Neste último caso, quando quisermos nos conectar novamente, teremos duas opções:

1. `rhc setup`, que nos conecta levando em conta o usuário utilizado na última sessão;
2. `rhc setup -l <username>`, que se conectará à conta OpenShift associada ao usuário `<username>` informado.

Por enquanto, isto é tudo o que precisamos saber sobre o rhc. Agora que já o temos instalado e configurado em nossas máquinas, passaremos para a próxima fase de nosso tutorial: a criação do servidor Jenkins.

Listagem 2. Terminal – setup do rhc.

```
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ rhc
setup
OpenShift Client Tools (RHC) Setup Wizard

This wizard will help you upload your SSH keys, set your application namespace,
and check that other programs like Git are properly installed.

If you have your own OpenShift server, you can specify it now. Just hit enter to use
the server for OpenShift Online: openshift.redhat.com.
Enter the server hostname: |openshift.redhat.com|

You can add more servers later using 'rhc server'.
RSA 1024 bit CA certificates are loaded due to old openssl compatibility

Login to openshift.redhat.com: pedrobrigatto@gmail.com
Password: *****

OpenShift can create and store a token on disk which allows to you to access the
server without using your password. The key is stored in your home
directory and should be kept secret. You can delete the key at any time by run-
ning 'rhc logout'.
Generate a token now? (yes|no) yes
Generating an authorization token for this client ... RSA 1024 bit CA certificates are
loaded due to old openssl compatibility
lasts about 1 month

Saving configuration to /Users/pedrobrigatto/.openshift/express.conf ... done

Your public SSH key must be uploaded to the OpenShift server to access
code. Upload now? (yes|no) yes

Since you do not have any keys associated with your OpenShift account, your new
key will be uploaded as the 'default' key.

Uploading key 'default' ... done

Checking for git ... found git version 2.3.8 (Apple Git-58)

Checking common problems .. done

Checking for a domain ... devopsdevmedia

Checking for applications ... found 1

pedrobrigatto http://pedrobrigatto-devopsdevmedia.rhcloud.com/

You are using 1 of 3 total gears
The following gear sizes are available to you: small

Your client tools are now configured.
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$
```

Criando um servidor Jenkins via rhc

A criação de um servidor Jenkins via rhc é um procedimento bastante simples e rápido. Há uma característica, entretanto, que devemos discutir antes de executarmos os comandos propriamente ditos. Trata-se de um desafio resultante do modelo de segurança estabelecido pela OpenShift, que bloqueia tanto a comunicação direta entre nós (ou *gears*, de acordo com o vocabulário da plataforma) quanto o acesso a alguns diretórios fundamentais em um processo de integração e entrega contínuas, como o `.m2` (padrão do Maven tanto para gerenciamento local de dependências quanto para a instalação de arquivos de configuração como `settings.xml`) e o `.ssh` (padrão para a instalação de chaves SSH e declaração de hosts conhecidos por um servidor).

Para contornar esta questão, criaremos uma aplicação utilizando dois ‘*cartridges*’ (o nome dado, no vocabulário OpenShift, para recursos, tais como frameworks, serviços web ou instâncias de bancos de dados):

1. ‘`jenkins-1`’, que é uma aplicação Jenkins pré-configurada;
2. ‘`GITSSH`’ (cuja referência para o código-fonte se encontra na seção **Links**), que consiste em algumas instruções em nível de script para contornar o aspecto do modelo de segurança da OpenShift, já citado.

No caso do segundo cartridge listado anteriormente, um executável denominado `git-ssh` é disponibilizado. Este executável redefine o comando `ssh` para utilizá-lo em conjunto com a opção `StrictHostKeyChecking` alterada para ‘`no`’ (seu valor padrão é ‘`yes`’). Além disso, o *cartridge* estabelece também um novo diretório para a instalação e definição de tudo o que tiver relação com a comunicação remota via SSH, que é o `$OPENSHIFT_DATA_DIR/git-ssh`; `OPENSHIFT_DATA_DIR`, no caso, é uma variável de ambiente automaticamente definida em qualquer gear criada em uma conta OpenShift, usada para guardar uma referência para o diretório de referência.

O resultado da criação da aplicação pode ser observado na **Listagem 3**. Perceba que iniciamos com o comando `rhc create-app`, com os seguintes parâmetros:

- `devmediajenkins`, representando o nome da instância Jenkins em criação;
- `jenkins-1`, representando o cartridge padrão de uma aplicação Jenkins, já descrito;
- A URL para o cartridge ‘`GITSSH`’, também já descrito antes.

Assim que o Jenkins é criado e implantado, uma mensagem é informada no terminal e, em seguida, as credenciais para acesso à interface web de administração são exibidas. Logo após, perceba que a URL para o servidor é automaticamente adicionada à lista de hosts conhecidos do computador que estamos usando para configurar o sistema. Então, as informações mais importantes de todas são finalmente mencionadas. É por meio dos endereços informados no final do processo que, efetivamente, nos conectaremos ao servidor Jenkins para realizar a maioria dos procedimentos de configuração nas próximas seções.

Listagem 3. Terminal – Criação do Jenkins pela linha de comando usando rhc.

```
Pedro-Brigattos-MacBook:DevMediaOpenshiftApps pedrobrigatto$ pwd  
/Users/pedrobrigatto/Personal/Projects/DevMediaOpenshiftApps  
Pedro-Brigattos-MacBook:DevMediaOpenshiftApps pedrobrigatto$ ls -la  
total 0  
drwxr-xr-x 2 pedrobrigatto staff 68 Dec 30 08:00 .  
drwxr-xr-x 19 pedrobrigatto staff 646 Dec 29 15:31 ..  
Pedro-Brigattos-MacBook:DevMediaOpenshiftApps pedrobrigatto$ rhc  
create-app devmediajenkins jenkins-1 "https://carreflect-claytondev  
.rhcloud.com/reflect?github=majecek/openshift-community-git-ssh"  
RSA 1024 bit CA certificates are loaded due to old openssl compatibility  
The cartridge 'https://carreflect-claytondev.rhcloud.com/  
reflect?github=majecek/openshift-community-git-ssh' will be downloaded  
and installed
```

Application Options

```
-----  
Domain: pedrobrigatto  
Cartridges: jenkins-1, https://carreflect-claytondev.rhcloud.com/  
reflect?github=majecek/openshift-community-git-ssh  
Gear Size: default  
Scaling: no
```

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: <usuário-servidor-jenkins>@devmediajenkins-pedrobrigatto
.rhcloud.com
Git remote: ssh://<usuário-servidor-git>@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.

Corrigindo o serviço de download do Jenkins

Agora que já temos a gear operando normalmente, é necessário configurar algumas características que serão úteis quando chegarmos à fase de criação dos jobs.

A primeira delas é, na realidade, um defeito associado ao serviço de download de componentes (*DownloadService*). Em instâncias recém-instaladas do Jenkins, a listagem das versões de componentes suportados pelo servidor não é exibida para o administrador, e a única maneira de se preencher esta informação seria por meio de um campo de texto, manualmente. Isto pode ser corrigido facilmente executando os passos listados a seguir:

1. Na página principal do Jenkins, vá em *Jenkins > Manage Jenkins*;

2. Selecione a opção *Script Console*;
3. Cole, no editor que aparece, o conteúdo da **Listagem 4**;
4. Execute o script.

A lógica deste script desabilita a verificação de assinaturas para, então, atualizar toda a lista de componentes. Logo após, como última etapa, a verificação de assinaturas é reabilitada. Este procedimento de 'refresh' dos componentes e suas respectivas versões é suficiente para que todas as versões de sistemas/ferramentas sejam listadas corretamente para nós, quando precisamos instalá-las.

Listagem 4. Script de correção para o serviço de download do Jenkins.

```
hudson.model.DownloadService.signatureCheck = false hudson.model  
.DownloadService.Downloadable.all().each { it.updateNow() } hudson.model  
.DownloadService.signatureCheck = true return
```

O próximo passo descreve um ajuste importante a ser feito para garantir que jobs configurados no Jenkins sejam executados. Este procedimento deve ser realizado a partir da tela de configuração do sistema, acessada via *Jenkins > Manage Jenkins > Configure System*.

Definindo o número de executores

O pré-requisito essencial para um job entrar em execução é a existência de um executor disponível. De acordo com a documentação, o número padrão de executores de uma instância recém-criada do Jenkins deveria estar configurada em 2, mas isto não é observado quando realizamos este procedimento por meio da OpenShift. O valor, neste caso, fica definido como 0, o que jamais permitiria que qualquer job entrasse em execução.

Para corrigir esta informação, precisamos realizar os passos listados a seguir:

1. Na página principal do Jenkins, vá em *Jenkins > Manage Jenkins*;
2. Selecione a opção *Configure System*;
3. Altere o valor do campo *# of executors* para 1;
4. Salve o trabalho.

Criando as chaves SSH para comunicação remota

Esta seção tratará da criação e instalação das chaves SSH, que permitirão a comunicação do Jenkins com outros sistemas, tais como SonarQube e Nexus, que exploraremos adiante. Para iniciarmos, lembremos das últimas informações exibidas na **Listagem 3**. Como antecipamos em uma seção anterior, tais dados serão fundamentais para que prossigamos com a configuração do ambiente de entrega contínua que propusemos neste artigo.

Para nos conectarremos remotamente ao servidor em que o Jenkins foi implantado, precisamos usar a informação da **Listagem 3** identificada como "SSH to:". Quando executarmos esta linha de comando em um terminal, exatamente como ela foi apresentada, abrimos uma comunicação direta com o servidor e, a partir de

DevOps e a Integração Contínua nas nuvens

então, podemos prosseguir com sua configuração via linha de comando.

Para acompanhar o processo detalhado nesta seção, faremos uso da **Listagem 5**. Veja que, assim que nos conectamos ao servidor, partimos para a criação do par de chaves pública e privada. Isto é feito por meio de uma ferramenta chamada *ssh-keygen*. As chaves foram criadas com o tipo RSA e foram instaladas no diretório *\$OPENSHIFT_DATA_DIR/git-ssh*. Lembre-se, aqui, da importância do que foi dito em seção anterior: as pastas localizadas no diretório ‘home’ do usuário criado no processo de criação do servidor Jenkins não permitem operações de escrita, sendo esta a principal razão de termos esta pasta alternativa configurada pelo *cartridge ‘GITSSH’* instalado no servidor.

O próximo passo é configurar estas chaves em todos os servidores que precisem se comunicar e modificar conteúdo entre si. É o que veremos na próxima seção.

Listagem 5. Terminal – criação do par de chaves para comunicação remota via SSH.

```
[devmediajenkins-pedrobrigatto.rhcloud.com git-ssh]">> pwd  
/var/lib/openshift/5683ab440c1e66ea82000098/app-root/data/git-ssh  
[devmediajenkins-pedrobrigatto.rhcloud.com git-ssh]">> ssh-keygen -t rsa -b 4096  
Generating public/private rsa key pair.  
Enter file in which to save the key (/var/lib/openshift/5683ab440c1e66ea82000098/.ssh/id_rsa): ./id_rsa  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
Your identification has been saved in ./id_rsa.  
Your public key has been saved in ./id_rsa.pub.  
The key fingerprint is:  
4e:9c:73:71:05:04:cfc0:ae:45:23:1b:42:94:af:52 5683ab440c1e66ea82000098@ex-std-node296.prod.rhcloud.com  
The key's randomart image is:  
++-[ RSA 4096]----+  
| oo...oo.. |  
| o o +o .. |  
| o * .+ . |  
| E o.o. o |  
| ..oS. . |  
| ...o o |  
| . . . . |  
+-----+  
[devmediajenkins-pedrobrigatto.rhcloud.com git-ssh]">> ls -la  
total 12  
drwxr-xr-x. 2 5683ab440c1e66ea82000098 5683ab440c1e66ea82000098  
36 Dec 30 05:20 .  
drwxr-x---. 12 5683ab440c1e66ea82000098 5683ab440c1e66ea82000098  
4096 Dec 30 05:13 ..  
-rw-----. 1 5683ab440c1e66ea82000098 5683ab440c1e66ea82000098 3243  
Dec 30 05:20 id_rsa  
-rw-----. 1 5683ab440c1e66ea82000098 5683ab440c1e66ea82000098 778  
Dec 30 05:20 id_rsa.pub  
[devmediajenkins-pedrobrigatto.rhcloud.com git-ssh]">>  

```

Associando a chave SSH gerada à conta OpenShift

A configuração das chaves SSH é muito simples e rápida. O que precisamos fazer para que tudo funcione conforme o esperado resume-se em:

- Copiar o conteúdo da chave pública gerada na seção anterior. Esta chave encontra-se em *\$OPENSHIFT_DATA_DIR/git-ssh/id_rsa.pub*;
- Adicionar esta chave em sua conta OpenShift;
- Adicionar esta chave em sua conta GitHub, somente se tiver intenção de realizar operações de escrita no repositório, a partir do Jenkins. Se este não for o caso – e em nosso tutorial não é, este passo pode ser ignorado.

Para copiar o conteúdo da chave pública (*id_rsa.pub*), executamos o último comando marcado em negrito, na **Listagem 5**. A seguir, navegamos até a página de URL <https://openshift.redhat.com/app/console/keys/new> – cujo conteúdo está exibido na **Figura 2** – e vinculamos esta chave SSH à nossa conta OpenShift.

Ao cumprirmos as etapas descritas até aqui, já temos um servidor de integração contínua operando normalmente na nuvem. Para encerrarmos o tópico sobre Integração Contínua, portanto, criaremos os dois primeiros jobs de nosso processo de Entrega Contínua.

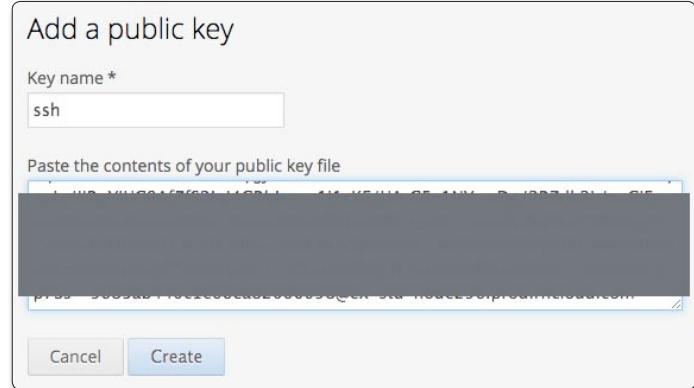


Figura 2. Formulário para associar uma chave SSH a uma conta OpenShift

O primeiro job: limpando e compilando o projeto

A configuração que adotaremos no Jenkins será composta por uma coleção de jobs e de acordo com o diagrama da **Figura 3**. Neste primeiro momento, concentraremos nossos esforços no desenvolvimento dos jobs 1 e 2, deixando os dois últimos para seções futuras.

O primeiro job, de nome *devmediadevops_clean_job*, é bastante simples e tem como único objetivo garantir que a árvore de diretórios do projeto seja devidamente limpa. Por limpeza, devemos entender a eliminação de todo e qualquer conteúdo gerado em builds anteriores.

Para criá-lo, selecionamos a opção *Jenkins > New Item* e, na tela seguinte, definimos seu nome como *devmediadevops_clean_job* e seu tipo como *Freestyle project*. Embora este projeto utilize o Apache Maven para gerenciar as fases de seu ciclo de vida, optamos pelo

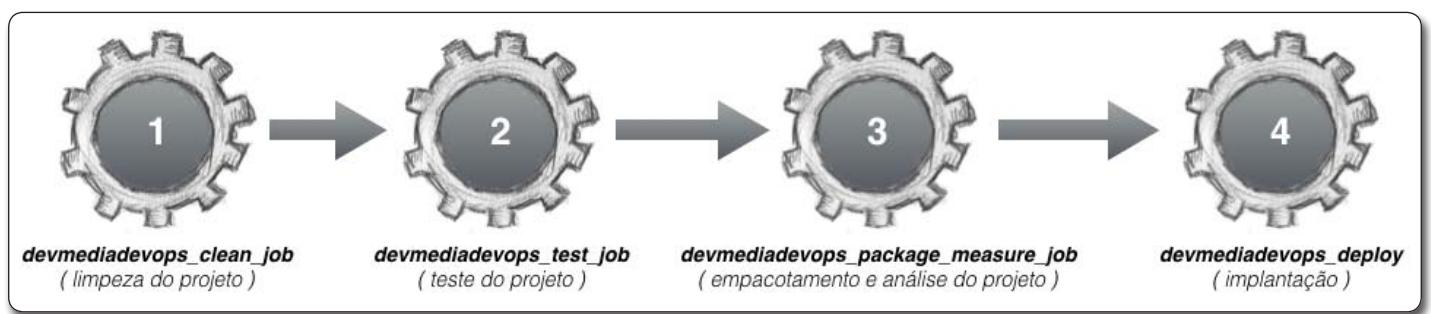


Figura 3. Cadeia de jobs desenvolvida no artigo

'modo livre' de criação de jobs – ao invés da opção '*Maven project*', porque este último modelo ainda apresentava alguns problemas que inviabilizavam sua utilização, no momento em que o artigo era escrito.

A etapa seguinte é a caracterização do job. Na seção '*Source code management*', preenchemos todos os campos de acordo com o conteúdo da **Tabela 1**. Resumidamente, é aqui que definimos qual é o repositório de código-fonte no qual estamos interessados, além do branch a ser considerado.

Característica	Detalhes
Tipo	Git
Repositories	URL: https://github.com/pedrobrigatto/devmedia_devops_series.git Credentials: none
Branches do build	*/master
Repository browser	(Auto)

Tabela 1. Configuração do repositório de código-fonte usado nos Jobs

O conjunto seguinte de propriedades a ser editado define o processo de build propriamente dito. Neste caso, os dados foram preenchidos seguindo-se as informações citadas na **Tabela 2**, das quais destacamos dois pontos importantes. O primeiro tem a ver com a configuração de um token que utilizaremos, adiante, para disparar sua execução remotamente a partir do conceito de *web hooks* do GitHub. Neste exemplo, definimos este token com o valor '*devmedia_devops_token*'.

O segundo ponto importante é a utilização de um parâmetro de execução do Maven, '*f*', que indica o caminho completo até o descriptor do projeto (*devops/pom.xml*). Esta é uma necessidade nossa, pois somente assim garantiremos que o Maven localizará, com certeza, o arquivo que precisa para orientar a sua execução. Veja, mais uma vez, que fizemos uso da variável *OPENSHIFT_DATA_DIR* e, a partir dela, completamos o caminho até o projeto, que sempre é baixado dentro do diretório *workspace* do Jenkins.

Por fim, perceba que definimos uma ação para quando o build for concluído (uma 'post-build action'), que vincula a execução do segundo job da cadeia à finalização bem-sucedida do primeiro. Em outras palavras, o job de verificação, que veremos na próxima

seção, será executado sempre que o primeiro, responsável pela limpeza da estrutura de diretórios do projeto, for finalizado com sucesso.

Característica	Detalhes
Build triggers	Opção 'Trigger builds remotely' selecionada. Authentication token: <i>devmedia_devops_token</i>
Build	Passo único Tipo: Invoke top-level Maven targets Goals: clean -f \$OPENSHIFT_DATA_DIR/workspace/sonar/devops/pom.xml
Post-build actions	Passo único Tipo: Build other projects Projeto: <i>devmediadevops_test_job</i> Condição selecionada: Trigger only if build is stable

Tabela 2. Propriedades relacionadas ao build do job *devmediadevops_clean_job*

O segundo job: a verificação do projeto

Assim que realizamos uma limpeza completa no diretório do projeto, é importante que verifiquemos se o código-fonte segue em conformidade com os requisitos acordados. Para isto, definimos o segundo job da cadeia, com o nome *devmediadevops_test_job*, com o objetivo único de executar todos os testes unitários existentes no projeto.

A lógica por trás deste job é permitir que o empacotamento e consequente disponibilização do produto para testes de qualidade e produção seja liberado somente quando todos os testes forem realizados com sucesso. Caso uma falha ocorra, a cadeia de jobs será interrompida e será necessária a análise manual de um operador – ou mesmo de um desenvolvedor – da situação.

Este job também foi criado como um *Freestyle project*, seguindo o mesmo procedimento já descrito na seção anterior, e seu nome foi definido como *devmediadevops_test_job*. As configurações da seção *Source code management* seguem exatamente as mesmas características já apresentadas na **Tabela 1**, e as únicas características que precisam ser destacadas neste job dizem respeito ao conteúdo da seção *Build*.

Observemos o conteúdo da **Tabela 3**. O *goal* estabelecido para este build, no caso, foi alterado de '*clean*' para '*test*'. Da mesma maneira que no primeiro job, entretanto, mantivemos o parâmetro '*f*', para garantir que o caminho até o descriptor do projeto seja sempre válido.

Novamente, perceba que o segundo job também tem uma ação de '*post-build*' a ele associado, estabelecendo que toda execução bem-sucedida da suíte de testes unitários do projeto dispare, imediatamente, o terceiro job da cadeia, responsável pelo empacotamento e análise estática de toda a lógica. Este job é o mais complexo de todo o tutorial e, para chegarmos nele, dedicaremos a próxima seção ao estudo de um componente fundamental deste ambiente de entrega contínua proposto para o artigo: o *SonarQube*.

Característica	Detalhes
Build	Passo único Tipo: Invoke top-level Maven targets Goals: test -f \$OPENSHIFT_DATA_DIR/workspace/sonar/devops/pom.xml
Post-build actions	Passo único Tipo: Build other projects: Projeto: devmediadevops_package_measure_job Condição selecionada: Trigger only if build is stable

Tabela 3. Configuração da seção Build do job devmediadevops_test_job

Analisando um projeto com o SonarQube

O SonarQube é uma plataforma destinada fundamentalmente à medição da qualidade de projetos de software. Para isso, utiliza como fontes de informação elementos como:

- Comentários;
- Código-fonte;
- Testes unitários.

Inúmeros são os aspectos avaliados a partir dos artefatos que acabamos de apresentar. Alguns, entretanto, merecem destaque e foram listados a seguir:

- Complexidade ciclomática;
- Duplicidade de código;
- Aderência a padrões pré-estabelecidos de codificação;
- Existência de causas potenciais de defeitos;
- Cobertura de testes.

Ao longo desta seção, aprenderemos como hospedar o SonarQube em um servidor utilizando uma conta OpenShift e, a partir daí empregá-lo em nosso ambiente de Entrega Contínua.

Criando uma instância do Sonar na OpenShift

A criação do SonarQube em uma conta OpenShift seguirá um caminho um pouco diferente do adotado para o Jenkins. Nesta ocasião, faremos uso de um cartridge denominado DIY (de "Do It Yourself" ou "Faça você mesmo"), que nos permitirá implantar na gear (provisionada pela *OpenShift*) uma aplicação externa. A grande diferença em relação ao procedimento com o Jenkins, é que não utilizaremos nenhum template já pré-definido de uma solução Sonar (como fizemos com o cartridge *jenkins-1*), trabalhando, neste caso, com um cartridge disponibilizado diretamente em um repositório Git, por um membro da comunidade.

Comecemos analisando o conteúdo da **Listagem 6**. O primeiro passo foi executar o comando *rhc app create*, para a criação da gear em nossa conta OpenShift, com os seguintes parâmetros:

- Nome: devmediasonar;
- Cartridges: *diy-0.1* e *postgresql-9.2*.

Na prática, este comando gera uma gear denominada *devmediasonar*, munida de uma aplicação em formato livre (*diy*) e um banco de dados PostgreSQL 9.2 também nomeado *devmediasonar*, a ser utilizado pelo SonarQube para controle e registro do estado das análises realizadas sobre o projeto.

Listagem 6. Terminal – criação da instância do SonarQube via rhc.

```
Pedro-Brigattos-MacBook:DevMediaOpenshiftApps pedrobrigatto$ rhc app create devmediasonar diy-0.1 postgresql-9.2
```

```
RSA 1024 bit CA certificates are loaded due to old openssl compatibility  
Application Options
```

```
-----  
Domain: pedrobrigatto  
Cartridges: diy-0.1, postgresql-9.2  
Gear Size: default  
Scaling: no
```

```
Creating application 'devmediasonar' ... done
```

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

```
PostgreSQL 9.2 database added. Please make note of these credentials:
```

```
Root User: admint2kflei  
Root Password: *****  
Database Name: devmediasonar
```

```
Connection URL: postgresql://$OPENSHIFT_POSTGRESQL_DB_HOST:$OPENSHIFT_POSTGRESQL_DB_PORT
```

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

```
Cloning into 'devmediasonar'...
```

```
Warning: Permanently added the RSA host key for IP address '54.86.73.64' to the list of known hosts.
```

```
Your application 'devmediasonar' is now available.
```

```
URL: http://devmediasonar-pedrobrigatto.rhcloud.com/  
SSH to: <usuário>@devmediasonar-pedrobrigatto.rhcloud.com  
Git remote: ssh://<usuário>@devmediasonar-pedrobrigatto.rhcloud.com/~/git/devmediasonar.git/  
Cloned to: /Users/pedrobrigatto/Personal/Projects/DevMediaOpenshiftApps/devmediasonar
```

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

A etapa seguinte consiste na configuração, de fato, de uma instância do SonarQube neste novo servidor, e em sua respectiva associação ao banco de dados já instalado. É a partir de agora que o leitor perceberá todo o poder deste template DIY oferecido pela OpenShift.

Os comandos para download, configuração e instalação da versão do SonarQube que utilizamos foram concentrados na

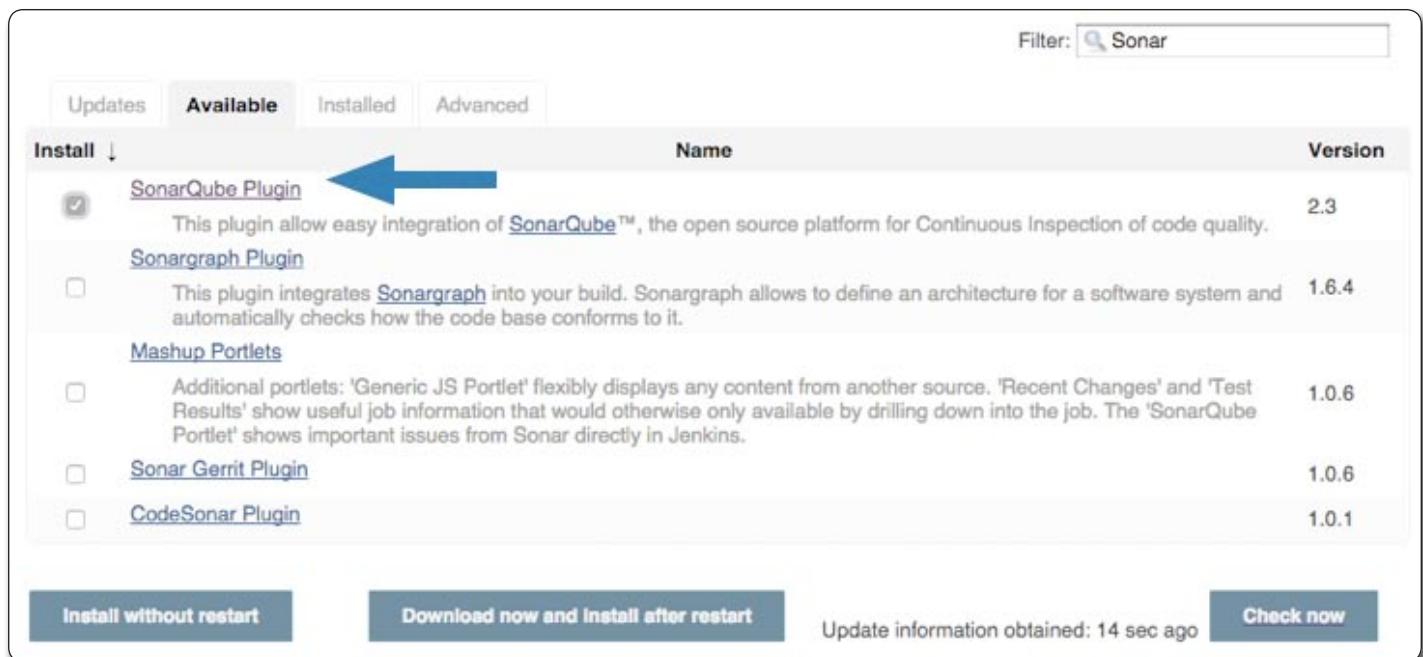


Figura 4. Instalação do plugin Sonar Qube no Jenkins

Listagem 7. Para começar, de dentro do diretório da nova gear criada (`devmediasonar`), removemos parte do conteúdo original da aplicação DIY gerada, por não ter utilidade alguma para nós. Fizemos isso por meio do comando `git rm`, excluindo recursivamente todo o conteúdo das pastas `.openshift`, `diy` e `misc`. Por fim, removemos também o arquivo `README.md`.

Listagem 7. Terminal – Submissão da aplicação Sonar e instanciação propriamente dita.

```
git rm -r diy .openshift misc README.md
git remote add upstream -m master https://github.com/worldline/openshift-sonarqube.git
git pull -s recursive -X theirs upstream master
git push
```

Em seguida, configuramos o Git com uma referência para o branch remoto de uma versão preparada do SonarQube que desejamos empregar em nosso projeto, por meio da URL <https://github.com/worldline/openshift-sonarqube.git>. O comando seguinte realiza uma fusão (*merge*) do conteúdo local com aquele disponível no branch adicionado pelo comando anterior, dando preferência absoluta para o material remoto, caso seja encontrado algum conflito.

Por fim, quando o procedimento de merge foi concluído, fizemos o upload de todo o conteúdo para a aplicação DIY no servidor. Dentro de poucos instantes, a aplicação Sonar já se encontrava disponível pelo painel de administração de nossa conta OpenShift.

Este cartridge *diy-1.0*, portanto, serviu-nos simplesmente para criar um ‘esqueleto’ de projeto para que, posteriormente, fosse preenchido com um projeto real definido de forma livre, com código produzido por nós ou, então, adaptado de outro projeto

disponível na web, que é o caso desta aplicação SonarQube que implantamos.

Embora já tenhamos esta aplicação devidamente instalada em nossa gear, ainda precisamos fazer com que ela seja acessada por jobs configurados no Jenkins, permitindo, assim, que o processo de integração contínua proposto inclua análises estáticas de qualidade do código-fonte do projeto. Isto é o que passaremos a ver ao longo da próxima seção.

O plug-in para integração entre Sonar Qube e Jenkins

A comunicação entre o Jenkins e o SonarQube começa a ser viabilizada por meio da instalação e configuração de um plug-in do Jenkins denominado SonarQube. Os passos necessários para instalarmos o plug-in são os seguintes:

- Na página inicial da aplicação web de administração do Jenkins, navegue até *Jenkins > Manage Jenkins > Manage Plug-ins*;
- Selecione a aba *Available*;
- No campo *Filter*, digite *SonarQube* e pressione *Enter*;
- Selecione e instale o plug-in *SonarQube*, clicando no botão *Install without restart*, conforme ilustrado na **Figura 4**;
- Reinicie o Jenkins para que a configuração seja aplicada.

Este plug-in nos permite definir uma conexão com uma instância do SonarQube e, por consequência, a execução de análises estáticas a partir de qualquer job que desejarmos. Entretanto, a configuração propriamente dita apresenta algumas peculiaridades que serão exploradas em mais detalhes ao longo das próximas seções.

Configurando a comunicação entre o Sonar e o Jenkins

Assim que instalamos o plug-in e reiniciamos o Jenkins, chegou a hora de configurá-lo de acordo com os dados da aplicação

SonarQube criada em uma seção anterior. Este procedimento é executado a partir dos passos listados a seguir:

- Acesse o painel de configuração do Jenkins, disponível em [Jenkins > Manage Jenkins > Configure System](#);
- Navegue por esta página até a seção *SonarQube*;
- Ao clicar no botão *Add SonarQube*, preencha os dados do formulário usando os dados listados na **Tabela 4**;
- Para entrar com os dados de comunicação com o banco de dados do SonarQube, clique no botão *Advanced* e preencha o restante do formulário de acordo com os dados listados na **Tabela 5**;
- Clique no botão *Save* para confirmar a operação.

Campo	Valor
Name	DevMediaSonar
Server URL	<a href="http://<URL-para-a-sua-aplicação-Sonar>/">http://<URL-para-a-sua-aplicação-Sonar>/
Sonar Account Login	Admin
Sonar Account Password	Admin
Disable	Não selecionar este campo

Tabela 4. Configuração do Sonar no Jenkins

Campo	Valor
Database URL	<code>jdbc:postgresql://\$OPENSHIFT_JENKINS_IP:15555/<nome_do_banco_de_dados></code>
Database login	<usuário gerado na criação da aplicação Sonar via rhc>
Database password	*****

Tabela 5. Configuração da comunicação com o banco de dados da gear devmediasonar

É importante que o leitor se atente ao fato de que os dados fornecidos nas **Tabelas 4** e **5** referem-se a dados da instância do SonarQube utilizada como guia para a elaboração deste artigo, de acordo com o conteúdo já apresentado em uma seção anterior, a partir da **Listagem 6**. Quando for seguir este tutorial para criar as suas próprias gears e aplicações, de acordo com os passos descritos neste tutorial, basta que substitua as informações em negrito nas duas tabelas por aquelas fornecidas pelo rhc ao executar todos os comandos informados.

Outro aspecto importante desta configuração é a URL de comunicação com o banco de dados. Como a política de comunicação entre gears estabelecida pela OpenShift é altamente restritiva, foi necessário adotar uma abordagem alternativa para que o SonarQube pudesse ser acionado a partir da execução de jobs no Jenkins.

Neste instante, basta entendermos que o endereço do banco de dados que passamos aqui trata-se, no fundo, de um endereço do Jenkins que atuará como uma espécie de ‘ponte’ para, via SSH, acessar o Sonar (um processo normalmente denominado de *tunnelamento SSH*). Perceba que a variável de ambiente empregada é a `OPENSHIFT_JENKINS_IP`, e a porta que empregaremos nesta configuração é a de número 15555. Esses dados são muito importantes e serão esclarecidos na próxima seção deste artigo, que consiste na criação do terceiro job de nosso processo de entrega contínua.

O terceiro job: empacotando e analisando o projeto com SonarQube

Como antecipado na seção anterior, este é o job mais complexo de todo o tutorial. Entenderemos, a partir de agora, a estratégia que empregamos para que a comunicação entre as gears `devmediajenkins` e `devmediasonar` pudesse ocorrer, bem como o layout padrão do resultado de uma análise estática realizada pelo SonarQube.

Este job também segue o modelo de um *Freestyle project*, e recebeu o nome de `devmediadevops_package_measure_job`. Da mesma forma que os demais, as características relacionadas à subseção *Source code management* foram definidas de acordo com o conteúdo da **Tabela 1**. Deste job, as principais seções em que precisamos focar nossos esforços foram *Build* e *Post-build Actions*.

Diferentemente do que vimos nos dois primeiros jobs, teremos dois passos de tipos distintos na seção *Build*. O primeiro deles será uma operação do tipo *Invoke Maven top-level targets*, responsável pelo empacotamento do projeto e que, assim como todos os outros passos desta natureza usados nos outros dois jobs já explorados neste artigo, receberá o argumento ‘-f’ com o caminho completo do descriptor (`pom.xml`) do projeto `devops`. Todo este conteúdo está devidamente documentado na **Listagem 8**, a título de referência.

O segundo passo, por sua vez, será do tipo *Execute shell* e é, efetivamente, aquele que gera as condições necessárias para que a chamada ao SonarQube ocorra. Este passo, cujos comandos são apresentados na **Listagem 9**, tem uma série de particularidades que serão destacadas ao longo dos próximos parágrafos.

O primeiro comando executado neste passo é uma conexão remota, via `ssh`, ao servidor em que o Sonar está instalado. O parâmetro `-i` fornecido nesta chamada serve para indicarmos o caminho completo até a chave SSH privada (`id_rsa`) criada em uma seção anterior para comunicação entre as duas gears. O parâmetro `-o` também é empregado, para informarmos a localização do arquivo de hosts mantido pela gear `devmediasonar`, na qual deverá ser salva uma referência à gear `devmediajenkins` antes que este job seja executado pela primeira vez. Voltaremos neste ponto um pouco mais à frente.

Listagem 8. devmediadevops_package_measure: Configuração do primeiro passo do job.

```
# Inicia o processo de build propriamente dito  
package -f $OPENSHIFT_DATA_DIR/workspace/sonar/devops/pom.xml
```

Listagem 9. devmediadevops_package_measure: Configuração do primeiro passo do job.

```
# Make sure Tunnel for Sonar is open  
# Find out IP and port of DB  
OPENSHIFT_POSTGRESQL_DB_HOST_N_PORT=$(ssh -i $OPENSHIFT_DATA_DIR/git-ssh/id_rsa -o "UserKnownHostsFile=$OPENSHIFT_DATA_DIR/git-ssh/known_hosts" -L $OPENSHIFT_JENKINS_IP:15555:$OPENSHIFT_POSTGRESQL_DB_HOST_N_PORT -N 56865ede7628e19c67000238@devmediasonar-pedrobrigatto.rhcloud.com '(echo "printenv OPENSHIFT_POSTGRESQL_DB_HOST"; printenv OPENSHIFT_POSTGRESQL_DB_PORT)')  
# Open tunnel to DB  
BUILD_ID=dontKillMe nohup ssh -i $OPENSHIFT_DATA_DIR/git-ssh/id_rsa -o "UserKnownHostsFile=$OPENSHIFT_DATA_DIR/git-ssh/known_hosts" -L $OPENSHIFT_JENKINS_IP:15555:$OPENSHIFT_POSTGRESQL_DB_HOST_N_PORT -N 56865ede7628e19c67000238@devmediasonar-pedrobrigatto.rhcloud.com &
```

Assim que a conexão entre as duas gears é estabelecida (a partir da execução do trecho de comando descrito acima, ‘ssh -i <arquivo de identidade> -o <KnownHosts> user@servidor’), são realizadas chamadas à função `printenv` (nativa em ambientes Unix/Linux) sobre duas variáveis de ambiente da gear `devmediasonar` (`OPENSHIFT_POSTGRESQL_DB_HOST` e `OPENSHIFT_POSTGRESQL_DB_PORT`, nesta sequência) para a construção da URL do banco de dados utilizado pelo SonarQube. Em outras palavras, o que executamos foi uma conexão remota com o servidor `devmediasonar` para, em seguida, obter as informações de acesso ao banco de dados PostgreSQL nele hospedado. Todo este procedimento de ‘descoberta’ desta URL é importante porque, uma vez que estamos lidando com um ambiente de cloud computing, não é razoável confiar em valores constantes, pré-definidos. Esta é, pois, uma estratégia bem mais flexível e que nos dá a garantia que precisamos de que trabalharemos sempre com as informações verdadeiras.

O texto final, resultante da concatenação da URL e do número da porta do banco de dados PostgreSQL, é salvo em uma variável de ambiente chamada `OPENSHIFT_POSTGRESQL_DB_HOST_N_PORT`, na gear `devmediajenkins`, e será usada para que o canal de comunicação entre as duas gears seja, de fato, aberto quando o job for executado.

Os primeiros trechos do segundo comando assemelham-se muito ao primeiro, pois também envolve a abertura de uma conexão via SSH entre as duas gears, `devmediasonar` e `devmediajenkins`. O restante do comando, entretanto, tem um propósito bastante diferente, definindo o encaminhamento de requisições com destino `$OPENSHIFT_JENKINS_IP:15555` (IP do servidor Jenkins e uma porta arbitrariamente escolhida para este `setup`) para o banco de dados hospedado na gear `devmediasonar`, representado pela já apresentada variável de ambiente `OPENSHIFT_POSTGRESQL_DB_HOST_N_PORT`. Tecnicamente falando, esta configuração estabelece um listener na gear `devmediajenkins`, monitorando a chegada de requisições à

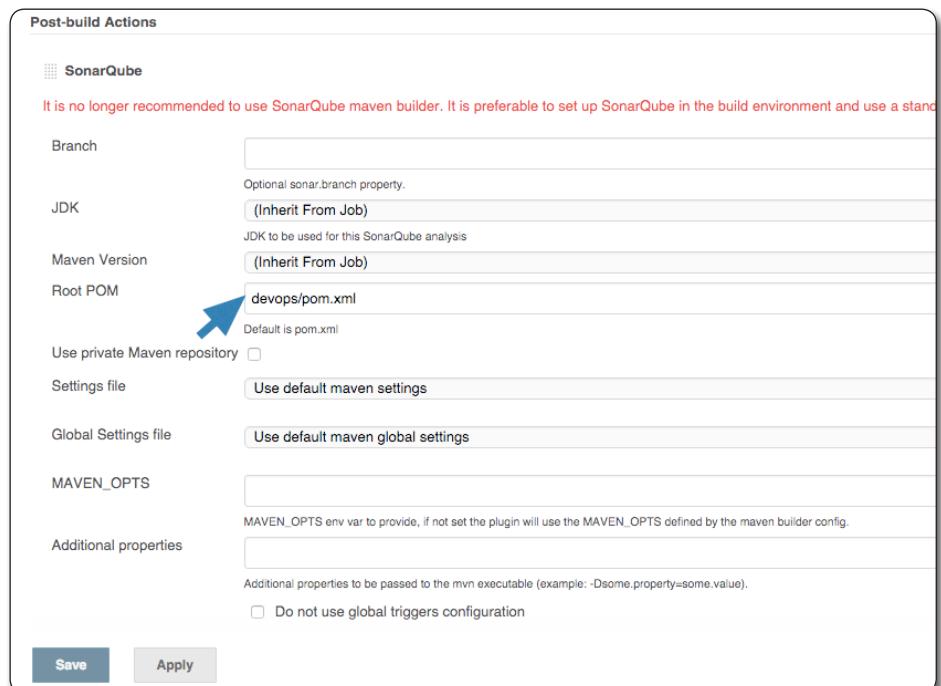


Figura 5. Configuração de uma ação de pós-build para análise estática do Sonar

porta de número 15555 e responsável por este redirecionamento citado há pouco. Tal encaminhamento de requisições foi definido a partir dos parâmetros listados a seguir:

- **L**, que define de fato o redirecionamento (origem > destino);
- **N**, uma instrução para que o comando remoto não seja executado (em outros termos, a necessidade aqui é apenas a configuração do comportamento, não a sua execução imediata).

Como um bom resumo de toda a história supracitada, podemos entender os dois passos deste job como a compilação e teste do projeto e, posteriormente, a configuração de um mecanismo de escuta no servidor Jenkins que define um mecanismo de encaminhamento de chamadas para o banco de dados hospedado na gear `devmediasonar` sempre que uma requisição chegue a uma porta definida arbitrariamente. Este encaminhamento/redirecionamento é o que, de fato, viabilizará a execução de uma análise estática do código-fonte por parte do SonarQube implantado em outra gear, diferente daquela em que o Jenkins está hospedado.

Depois que os dois passos de Build estão devidamente configurados, o próximo passo é criar uma ação de pós-build que é, efetivamente, a execução do Sonar. Nesse instante, fizemos uso do plug-in SonarQube instalado em um momento anterior. O que criamos – e que está ilustrado na **Figura 5** – foi uma *Post-build Action* do tipo SonarQube, clicando em seguida no botão chamado *Advanced*. De todos os campos que são exibidos neste momento, precisamos editar apenas aquele denominado *Root POM*, indicando o caminho relativo `devops/pom.xml`, conforme destacado pela seta azul. Este caminho é relativo sempre ao workspace do Jenkins, e o seu conteúdo pode ser visto inclusive através de sua aplicação web de administração.

Assim que cumprimos todos os requisitos e passos discutidos até aqui, estamos prontos para salvar todo o trabalho, clicando no botão *Save*. Entretanto, é importante que lembremos que, para tudo funcionar conforme o esperado, ainda precisamos realizar a primeira conexão SSH da gear `devmediajenkins` à `devmediasonar` para que ela seja

DevOps e a Integração Contínua nas nuvens

mapeada no arquivo de hosts conhecidos desta última. Portanto, a tarefa que ainda precisamos completar consiste em dois passos apenas, executados na ordem em que são listados:

1. Acesse a gear *devmediajenkins* via SSH de acordo com o comando exibido na **Listagem 3** (`ssh <usuário_jenkins> @<servidor_jenkins>`);
2. De dentro do servidor Jenkins, acesse a gear *devmediasonar*, via SSH, através do comando exibido na **Listagem 6** (`ssh<usuário_sonar@servidor_sonar>`);
3. De dentro dele, realize uma conexão, também via SSH, com o servidor Sonar.

Análises off-line no Eclipse: introdução ao SonarLint

Agora que entendemos como configurar a execução de análises estáticas e remotas através do SonarQube e em um contexto de Integração/Entrega contínua, veremos também como usufruir deste importante recurso em caráter *off-line*, em nível local, por meio do emprego de um plug-in do Eclipse conhecido como SonarLint.

Este plug-in está disponível através do *Eclipse Marketplace* (*Eclipse > Help > Eclipse Marketplace*). Ao acessá-lo, vemos uma janela como a ilustrada pela **Figura 6**. Nesta janela, por meio do campo identificado com o texto *Find*, faça uma pesquisa pelo nome “SonarLint” e, dentro de instantes, verá este plug-in como a primeira opção da lista, como indicado na figura. A instalação

em si é bastante trivial e não será detalhada neste artigo. Ao concluir-la e reiniciar o Eclipse, todas as configurações serão aplicadas e o plug-in estará pronto para uso em nossos projetos, a partir da IDE.

A execução de uma análise estática por meio do SonarLint pode ser realizada da seguinte maneira:

- Selecione o projeto desejado (em nosso caso, o projeto ‘*devops*’) na view *Project Explorer* da IDE;
- Clique com o botão direito do mouse sobre ele e selecione a opção *SonarLint > Analyze all files*.

O resultado deste procedimento será algo como o exibido na **Figura 7**. Basicamente, os mesmos parâmetros e critérios padrão de análise do Sonar foram utilizados nesta análise local, resultando nas mesmas observações que verificaremos quando rodarmos o Sonar remotamente, a partir de um build do Jenkins. A vantagem de se executar o SonarLint pela própria IDE é a velocidade com que descobrimos problemas ou más práticas, podendo corrigi-las antes até de submetermos o código-fonte ao controle de versão distribuído, no GitHub.

Já quando executamos o job Jenkins, o resultado pode ser observado diretamente por meio da aplicação Sonar cuja URL é informada na **Listagem 6**. O conteúdo será algo como o ilustrado na **Figura 8**.

O resultado de uma análise como esta é de uma riqueza enorme para a administração da evolução de um projeto de software. Por meio dos gráficos e tabelas apresentadas neste dashboard, somos capazes de, rapidamente, diagnosticar as debilidades mais urgentes de nossos projetos, que podem ir de uma simples – embora preocupante – má cobertura de código até um índice inadequado associado ao tema que discutiremos brevemente, a partir de agora, denominado Débito Técnico.

Débito Técnico: cuidado com ele!

Análises estáticas como as realizadas pelo SonarQube contribuem imensamente para comunicar o estado qualitativo de um projeto de software. Dashboards como o exibido na **Figura 8** são um manancial de informações que nos dão condições de lapidar o produto ou serviço em desenvolvimento para que se torne, continuamente, o mais profissional possível.

Tudo o que, direta ou indiretamente, influenciar negativamente nesta caminhada de aprimoramento acabará refletido em uma característica chamada ‘Débito Técnico’. O termo ‘débito’, aqui, tem o mesmo significado de dívida usado em outros cenários, e o valor a ele associado representa a quantidade estimada de dias de trabalho que seriam necessários para elevar um projeto ao seu estado ótimo. Esta métrica é o resultado de uma análise cruzada de uma série de aspectos, dos quais destacaremos e analisaremos sucintamente apenas dois dos mais comuns.

A cobertura de código por meio de testes automáticos, uma das perspectivas de análise apresentadas neste dashboard, é um critério muito decisivo na composição do débito técnico. Uma baixa taxa de cobertura reflete, frequentemente, em uma implementação bastante sujeita a falhas, representando um alto risco para o projeto.

O segundo ponto que destacamos, que também reflete no cálculo do débito técnico de um projeto, é o uso de más práticas de codificação e, também, a falta do uso de padrões de design tanto na definição da arquitetura quanto na implementação

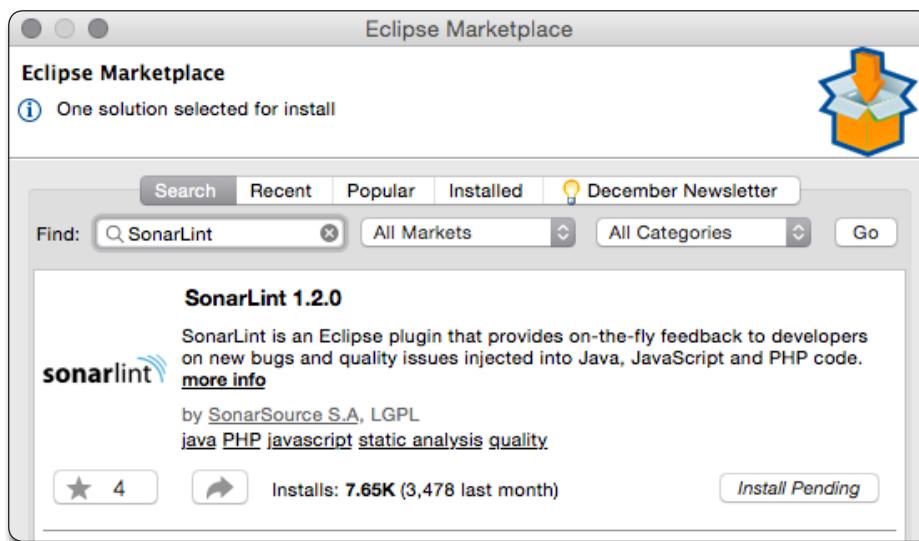


Figura 6. SonarLint disponível no Eclipse Marketplace para instalação

Date	Description	Resource
24 seconds ago	squid:S1161 : Add the "@Override" annotation above this method signature	CalculadoraR...
24 seconds ago	squid:S00105 : Replace all tab characters in this file by sequences of white-spaces.	CalculadoraR...
24 seconds ago	squid:S1604 : Make this anonymous inner class a lambda (sonar.java.source not set. Assuming 8 or greater.)	CalculadoraR...
24 seconds ago	squid:S1166 : Either log or rethrow this exception.	CalculadoraR...
24 seconds ago	squid:S2293 : Replace the type specification in this constructor call with the diamond operator ("<>"). (son...	CalculadoraR...
24 seconds ago	squid:S1166 : Either log or rethrow this exception.	CalculadoraR...
24 seconds ago	squid:S1166 : Either log or rethrow this exception.	CalculadoraR...
24 seconds ago	squid:S2293 : Replace the type specification in this constructor call with the diamond operator ("<>"). (son...	CalculadoraR...
24 seconds ago	squid:S2293 : Replace the type specification in this constructor call with the diamond operator ("<>"). (son...	CalculadoraR...
24 seconds ago	squid:S2293 : Replace the type specification in this constructor call with the diamond operator ("<>"). (son...	CalculadoraR...

Figura 7. Análise local do SonarLint sobre o projeto

de sistemas. Isto, invariavelmente, acaba originando uma série de problemas, tais como:

- Arquitetura engessada, prejudicando a evolução do sistema;
- Código duplicado, tornando o processo de manutenção muito mais complicado.

Embora a questão do Débito Técnico de um projeto seja fundamental, ela ainda tende a ser negligenciada em muitas empresas, especialmente por parte dos profissionais com maior influência sobre o futuro dos projetos. Não raramente, essas pessoas são menos conscientes das consequências que uma má gestão da qualidade pode trazer. Em geral, as primeiras atitudes em relação a este assunto acabam sendo tomadas tarde, quando a reparação dos danos já envolve um esforço muito maior, a um custo que, em alguns casos, chega a ser inviável.

Com ferramentas como o SonarQube, e dashboards tão intuitivos quanto os produzidos por ele, podemos não apenas gerenciar os riscos atuais de um projeto com muito mais controle, mas

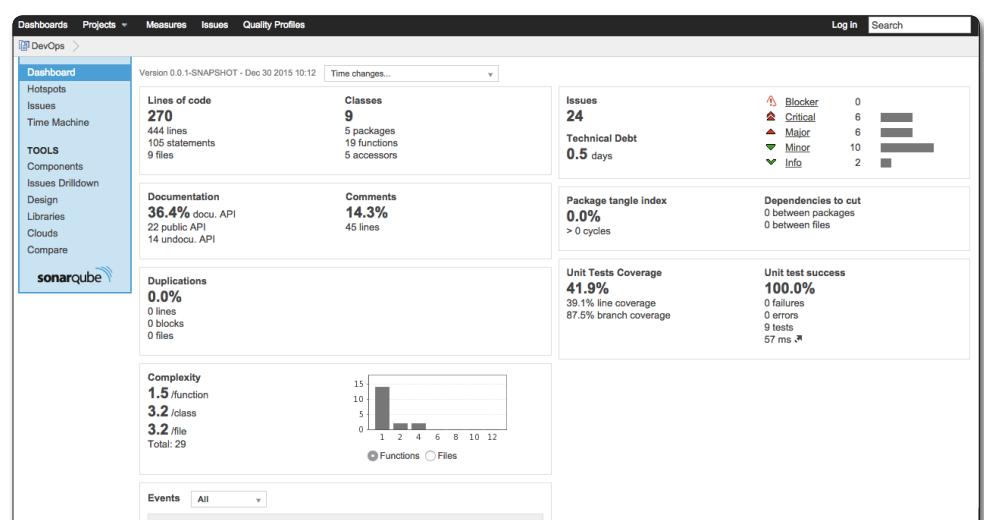


Figura 8. Resultado da análise feita pelo Sonar, a partir da gear devmediasonar

aproximar e educar *stakeholders* para a importância do amparo a atividades como design, *brainstorming*, *backlog grooming*, investigações tecnológicas. Quanto mais se investe na preparação de uma equipe, tanto no aspecto do negócio quanto da tecnologia, maiores as chances de se manter um projeto dentro de um bom padrão de qualidade, tornando-se menores os custos para mantê-lo e evoluí-lo.

Outra característica que requer cuidados especiais para manter um projeto dentro dos limites aceitáveis de qualidade é a comunicação. O desenvolvimento de um projeto torna-se mais suave e controlado à medida que a equipe responsável por ele se mantém alinhada. Neste sentido, processos ágeis como o Scrum têm, diariamente, ganhado mais e mais adeptos, mesmo em cenários em que as equipes estão dispersas geograficamente, pois sua filosofia de transparência, antecipação de problemas e elevado grau de colaboração ganham forte apoio de tecnologias e plataformas, licenciadas comercialmente ou não, como as suítes do Google (Hangout, Drive, Google+) e Atlassian (Confluence, HipChat, dentre outros) ou, ainda, produtos como Slack, Skype, Avaya Aura Collaboration Environment e Avaya Scopia Video Collaboration.

Em resumo, podemos dizer que o desafio de um débito técnico controlado passa principalmente pela observação contínua dos padrões de qualidade de um projeto de software (e o SonarQube nos facilita imensamente a vida, neste sentido) e, não menos importante, a combinação dos seguintes itens:

- Comunicação clara e objetiva, em todos os sentidos e entre todos;
- Uma boa suíte de ferramentas, com bom nível de integração entre elas;
- Um processo adequado às características da empresa, dos profissionais e clientes;
- Engajamento e absorção do mindset corporativo por parte de todos os membros.

A adoção da cultura DevOps, seja em um contexto corporativo ou pessoal, é um desafio que envolve inúmeras dimensões, tanto de natureza comportamental quanto tecnológica. Para ser bem-sucedido nesta jornada, é importante que nos equipemos com ferramentas capazes de nos auxiliar na identificação, categorização e no tratamento dos inúmeros desafios inerentes ao processo. Para sermos efetivamente profissionais aderentes ao conceito DevOps, precisamos ser transparentes, colaborativos, atentos e prevenidos, antecipando problemas sempre que possível e, quando isto não for possível, reagir o mais rápido possível a qualquer problema que ocorra.

Tais atributos não seriam realizáveis em um nível aceitável se não fosse o emprego de tecnologia. A antecipação de problemas, por exemplo, passa invariavelmente por um processo confiável e eficiente de verificação de software, que só se faz possível por meio de automação. A mesma linha de pensamento vale para tantas outras práticas, como a geração de builds ou, ainda, a implantação de versões do produto/serviço em ambientes de qualidade ou produção, dentro das linhas conhecidas como integração e entrega contínua.

Neste contexto que acabamos de introduzir, uma das mais poderosas atividades que existem é a análise qualitativa de projetos de software, por meio de sistemas como o SonarQube, visto ao longo do artigo. A partir dos dados obtidos por esta ferramenta, é possível extrairmos um panorama atual da saúde de um projeto de

software, sob inúmeras perspectivas: cobertura de código, aderência às boas práticas de desenvolvimento, grau de acoplamento entre módulos ou unidades lógicas de código (classes, principalmente), dentre tantas outras. Ao tornarmos essas análises realizadas pelo SonarQube contínuas (associadas, por exemplo, a cada submissão de código por um desenvolvedor), passamos todos a nos alimentar de informações chave para a (re)definição e/ou aprimoramento de condutas utilizadas em um projeto, tais como:

- Escrita de código-fonte, que passa a ser mais cuidadosa à medida que a equipe aprende com as falhas e os riscos reportados pelas análises;
- Design, refinado à medida que as avaliações de características como a complexidade ciclomática e as relações de coesão e acoplamento são apresentadas.

O resultado disso é, invariavelmente, o alcance de um padrão de trabalho cada vez menos sujeito a falhas e oscilações e que, consequentemente, contribui para o preenchimento dos espaços vazios que, historicamente, sempre marcaram a relação entre o universo de desenvolvimento e o de operações.

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

Correção para o problema relacionado ao serviço de download do Jenkins.

<https://issues.jenkins-ci.org/browse/JENKINS-26780>

Trabalhando com hooks no git.

<https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks>

Instalação do rhc.

<https://developers.openshift.com/en/managing-client-tools.html>

Tutorial para geração de um setup OpenShift com Jenkins, Sonar e um wrapper SSH.

<http://tinyurl.com/hl6v8r7>

Você gostou deste artigo?

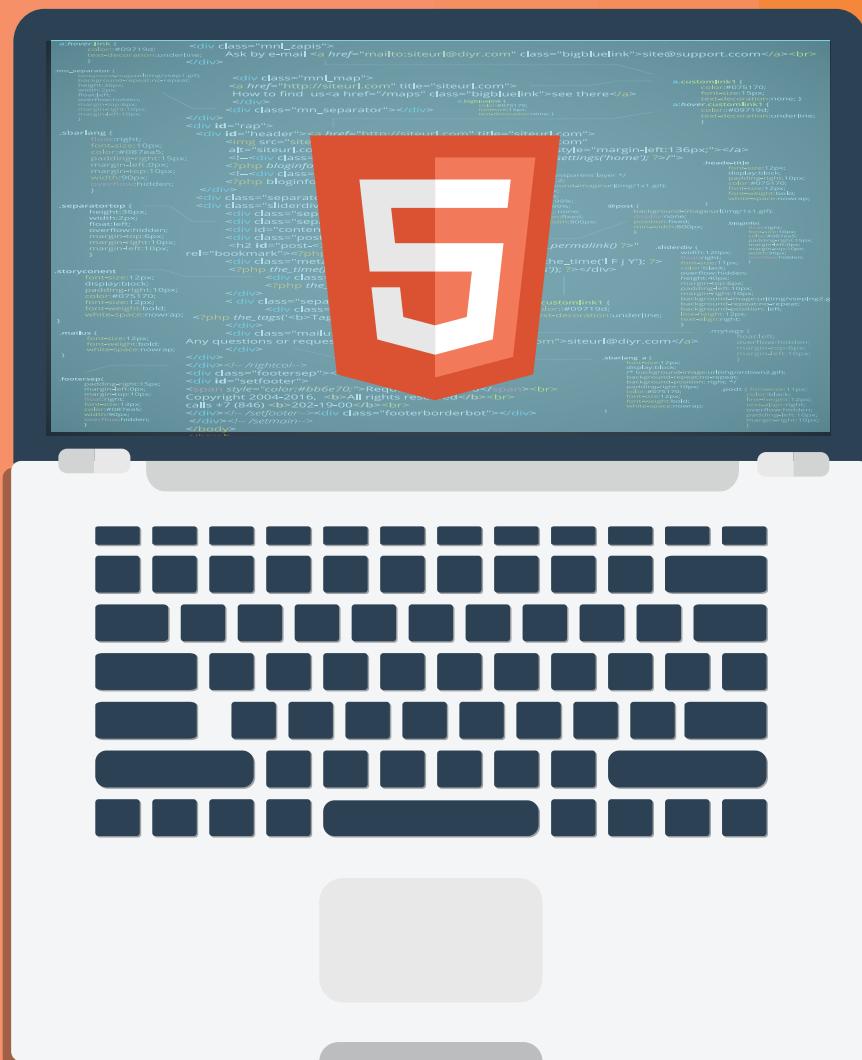
Dê seu voto em www.devmedia.com.br/javamagazine/feedback

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