



Edição 146 :: R\$ 14,90

 DEVMEDIA



Assinaturas digitais em Java
Como implementar os formatos no padrão nacional

Persistência em MongoDB com a API Morphia
Simplificando o desenvolvimento
de soluções NoSQL

DevOps – Uma abordagem prática
Integrando NetBeans, Maven,
JUnit, GitHub e Jenkins

Aplicações web escaláveis com Crux
Saiba como utilizá-lo e explore
recursos do HTML5 e JavaScript

OPEN DATA

Aprenda a criar
soluções de
serviço público



Dominando os testes automatizados
Como escrever testes e montar a
suite ideal para sua aplicação

Conhecendo o Apache Tomcat e o TomEE
De container web a um servidor
de aplicações completo

MVP

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

APlique esse investimento
na sua carreira...

E mostre ao mercado
quanto você vale!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

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

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

 **DEVMEDIA**

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Diogo Souza (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araújo

Distribuição

FC Comercial e Distribuidora S.A

Rua Teodoro da Silva, 907, Grajaú - RJ

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

Atendimento ao leitor

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

Publicidade

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

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

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

Fale com o Editor!

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

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



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

FÓRUM DEVMEDIA

O lugar perfeito para você ficar por dentro de tudo o que acontece nas tecnologias do mercado atual



No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!

ACESSE AGORA
www.devmedia.com.br/forum

Sumário

Conteúdo sobre Novidades

06 – Conhecendo o Apache Tomcat e o TomEE

[Geucimar Brilhador]

Conteúdo sobre Boas Práticas

14 – DevOps: Uma abordagem prática

[Juliano Rodrigo Lamb, Everton Coimbra de Araújo e Evando Carlos Pessini]

25 – Assinaturas digitais em Java: conheça e implemente os formatos no padrão nacional

[Emerson Sachio Saito e Fabiano Castro Pereira]

Conteúdo sobre Novidades, Artigo no estilo Solução Completa

38 – Open Data: desenvolvendo uma aplicação para monitoramento de ônibus

[Eduardo Felipe Zambom Santana e Luiz Henrique Zambom Santana]

Artigo no estilo Solução Completa

47 – Desenvolva aplicações web escaláveis com Crux Framework

[Samuel Almeida Cardoso]

Artigo no estilo Solução Completa

56 – Morphia: Veja como simplificar o desenvolvimento de aplicações NoSQL

[Lincoln Fernandes Coelho]

Conteúdo sobre Boas Práticas, Conteúdo sobre Engenharia de Software

65 – Dominando os tipos de testes automatizados

[Daniel Medeiros de Assis]



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.devmaster.com.br/mvp>



Conhecendo o Apache Tomcat e o TomEE

De container web a um servidor de aplicações corporativas completo. Veja esse artigo e conheça essas duas poderosas ferramentas, o Apache Tomcat e o TomEE

OApache Tomcat foi lançado oficialmente pela Apache Software Foundation (ASF) em 1999, como uma implementação de referência para a Servlet API. A sua primeira versão, curiosamente, já foi a 3.0, nunca existindo as versões 1.x ou 2.x do servidor. Antes disso, em 1995, a Servlet API foi concebida por James Gosling, e sua primeira implementação foi desenvolvida para o servidor Jeeves, criado pela JavaSoft, empresa da Sun Microsystems. De acordo com James Duncan, desenvolvedor Java que coordenava o projeto, “*O Jeeves é um dos poucos servidores que pode ser estendido de qualquer maneira, como uma espécie de canivete suíço*”.

Ainda segundo Duncan, que também trabalhou na especificação de Servlets e na implementação do Tomcat, em um processo de reestruturação promovido pela Sun, ele e sua equipe ficaram responsáveis por formalizar a Servlet API. Nesta ocasião, ao se preparar para codificar a implementação de referência, ele precisava nomear o pacote em que seriam gravados os arquivos e com base na capa de um livro sobre Servlets, que em breve seria lançado pela O'Reilly, surgiu o nome Tomcat. O nome não foi usado logo no início e o servidor foi lançado como JavaServer Web Development Kit, ou apenas JSWDK. Na **Listagem 1** temos a especificação da Servlet API que foi disponibilizada com o JSWDK 1.0.

Segundo Martin Bond, autor do artigo Tomcat Kick Start, a Sun distribuiu o Tomcat pela primeira vez no Java Web Services Developer Pack (Java WSDP) e na versão 1.3 do JSWDK. Enfim, em 1999, quando a Sun doou o código para a Apache, o nome Tomcat foi estabelecido definitivamente e ele se tornou a implementação de referência da Servlet API e do JSP. Se o projeto Jeeves fosse considerado como a primeira versão do Tomcat e

Fique por dentro

Este artigo trata de servidores de aplicações Java, mais especificamente, do servidor Apache Tomcat e sua extensão Apache TomEE. No texto o autor aborda detalhes desde a origem do servidor nos estabelecimentos da Sun Microsystems, em 1999, discorrendo sobre seu papel como contêiner web que serve de implementação de referência para as especificações Servlet e JavaServer Pages (JSP), apresentando os componentes da especificação JSR-316, referente à plataforma Java EE 6, até chegar ao Apache TomEE, versão estendida do Apache Tomcat que se propõe a ser uma implementação completa dos perfis definidos na plataforma corporativa do Java. Este assunto irá ampliar a visão do leitor em relação aos servidores Java e será útil no momento de definir qual servidor utilizar para rodar sua aplicação corporativa.

o JSWDK como a segunda versão, ficaria fácil entender por que o Tomcat só passou a existir a partir da versão 3.0 e nunca houveram as versões anteriores. No entanto, mesmo o código da JSWDK foi totalmente reescrito pelo próprio Duncan, em 1998.

A decisão de usar o 3.0 como versão inicial do Tomcat se deve à junção do servidor Apache JServ, construído com base na Servlet API 2.0, com a implementação do Servlet 2.2, realizada por Duncan. Após a conclusão da codificação e entrega do código fonte à Apache, ele comentou: “*A partir de então [referindo-se à doação do código], a dinâmica do código livre se consolidou. O Tomcat foi reescrito algumas vezes por diferentes equipes. Ampliado. Encolhido. Mas ele cumpriu seu objetivo: facilitar o uso de Servlets em todo o mundo e servir de referência para o funcionamento delas*”.

Nota

Antes da doação do servidor para a Apache, o JSWDK era um produto comercial da Sun cujo período de avaliação era de apenas 120 dias e custava, após este período, 295 dólares [ORACLE, 1997:2].

Versão Tomcat	Lançamento	Servlet	JSP	EL	WebSocket	Versão Java
8.x - 8.0.24	2014	3.1	2.3	3.0	1.1	7 ou superior
7.x - 7.0.63	2011	3.0	2.2	2.2	1.1	6 (WebSocket 1.1 requer 7 ou superior)
6.x - 6.0.44	2007	2.5	2.1	2.1	N/A	5 ou superior
5.5.x - 5.5.36 (arquivado)	2003	2.4	2.0	N/A	N/A	1.4 ou superior
4.1.x - 4.1.40 (arquivado)	2002	2.3	1.2	N/A	N/A	1.3 ou superior
3.3.x - 3.3.2 (arquivado)	1999	2.2	1.1	N/A	N/A	1.1 ou superior

Tabela 1. Versões do Apache Tomcat

Listagem 1. Servlet.java, v 1.4 20/04/1999.

```
package javax.servlet;

import java.io.IOException;

public interface Servlet {
    public void init(ServletConfig config) throws ServletException;
    public ServletConfig getServletConfig();
    public void service(ServletRequest req, ServletResponse res)
        throws ServletException, IOException;
    public String getServletInfo();
    public void destroy();
}
```

Depois da Sun propor a criação do Java Community Process (JCP), em 1998, apresentar a especificação oficial da Servlet API 2.2, em 1999, e entregar o código fonte para a Apache, a utilização do Tomcat cresceu rapidamente. De acordo com Jason Hunter, membro da Apache Software Foundation, a Sun queria difundir as tecnologias Servlet e JSP e a Apache era o caminho para isso. Além disso, a Sun não tinha interesse em abrir o código fonte, mas entregá-lo para a Apache, que dominava o mercado de servidores, foi um passo importante para torná-lo o servidor Java mais utilizado do planeta.

Em 2007, segundo Srini Penchikala, arquiteto e conferencista, o Tomcat era utilizado por aproximadamente 64% dos desenvolvedores da comunidade Java. E em 2013, de acordo com Vladimir Sor, criador da ferramenta Plumbr para medição de performance em aplicações Java, em 2013 o Tomcat ocupava 43% do mercado de servidores de aplicações.

Servlet e o Apache Tomcat

O Servlet é o coração do servidor Apache Tomcat. É uma tecnologia da plataforma Java para estender e evoluir o servidor. Seu modelo de desenvolvimento baseado em componentes abstrai do programador preocupações de baixo nível, como sockets, streams, serialização e afins. Através do Tomcat, o Servlet viabiliza um ambiente de desenvolvimento que roda código Java puro no lado servidor, tornando disponível toda a família de APIs Java, como a JDBC, Java EL, WebSockets e recursos do protocolo HTTP. Somando-se a isso o JSP, torna-se possível aos programadores criar páginas web com conteúdo estático e dinâmico.

Sob os cuidados da ASF, o Tomcat tornou-se uma implementação

de código aberto, com licença Apache versão 2 e referência das tecnologias Servlet e JSP.

Na **Tabela 1** é apresentada a evolução do servidor em conjunto com a evolução do Servlet.

Como pode ser verificado, a Servlet API 3.1 é a versão corrente. Definida na JSR-340, foi implementada pelo Apache Tomcat 8.x. A versão 4.0 da API, por sua vez, foi proposta na JCP através da JSR-369, por Shing Wai Chan em julho de 2014, e seu desenvolvimento está sendo coordenado por ele e Ed Burns. Apesar de não haver, ainda, uma data definida para sua conclusão, espera-se que ela esteja pronta até o primeiro semestre de 2017, momento em que deverá ser lançado o Apache Tomcat 9.x. A grande novidade que ela reserva é a compatibilidade com o protocolo HTTP 2.0.

A aceitação e utilização de Servlets pelo mercado não deixa dúvida de que ela é uma das mais populares e consolidadas tecnologias para se construir aplicações web com a plataforma Java. Diante disso, empresas como IBM, Oracle, Red Hat, BEA, entre outras, desenvolveram seus próprios servidores baseados na API, assunto que abordaremos no tópico sobre servidores de aplicações Java.



Conhecendo o Apache Tomcat e o TomEE

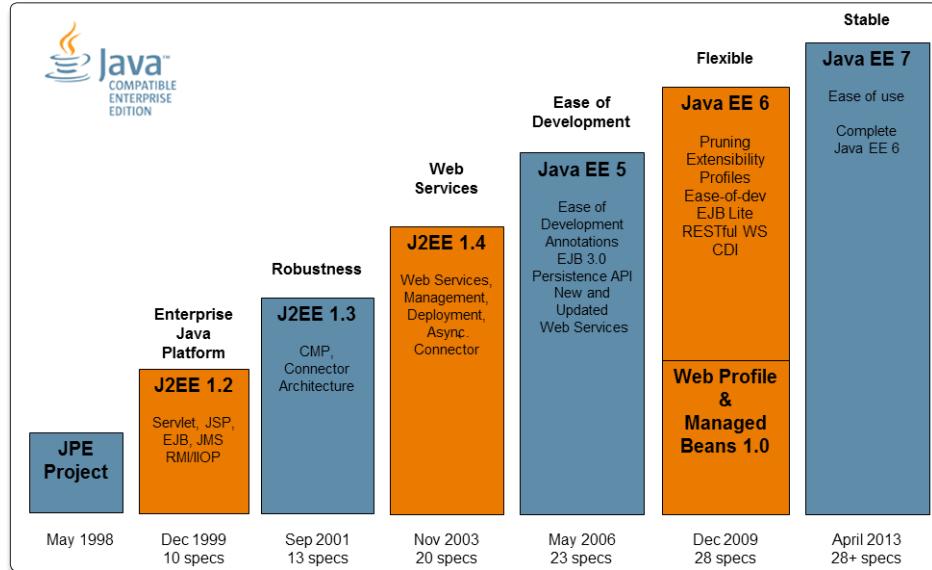


Figura 1. Especificações Java EE

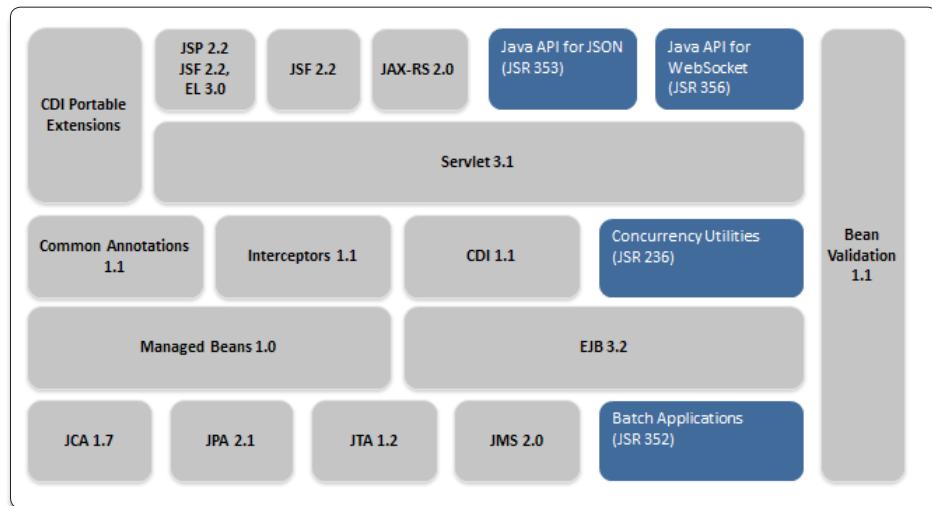


Figura 2. Especificação Java EE 7



Especificações e implementações Java EE

No final de 1997, com o intuito de tornar mais fácil aos desenvolvedores a criação de aplicações corporativas, a Sun publicou o primeiro rascunho dos Enterprise JavaBeans (EJBs). No ano seguinte, observando a importância deste fato, um grupo de empresas, composto pela Sun, HP, IBM, Novell, Oracle, Informix e WebLogic, anunciou a versão 1.0 da especificação. A partir disso, livres de questões de baixo nível como gerenciamento de transações, threads ou balanceamento de carga, os desenvolvedores poderiam se concentrar nas regras de negócio e deixar os detalhes

de funcionamento da aplicação para o servidor. Ainda sobre o assunto, em junho de 1998, a WebLogic divulgou a seguinte nota: “Hoje, a WebLogic anuncia que seu Servidor de Aplicações Tengah é totalmente compatível com a Especificação Enterprise JavaBeans 1.0. Os Enterprise JavaBeans tornam muito fácil desenvolver, distribuir e gerenciar aplicações comerciais críticas em Java. A versão 3.1 do Tengah é o primeiro produto a suportar o Enterprise JavaBeans 1.0”.

O Tengah era um produto comercial cujo preço inicial era de 9.995 dólares. Além de implementar EJBs, o servidor possuía uma interface gráfica para gerenciá-los, escalabilidade através de um cache de dados, pool de conexões, execução multithreaded e implementava também Java Naming and Directory Interface (JNDI), Java Database Connectivity (JDBC), Remote Method Invocation (RMI), HTTP Servlets e gerenciamento de eventos Java.

Ditando a evolução, em maio de 1998 a Sun anunciou a Java Professional Edition (JPE). No início, os Servlets e JSPs eram as únicas tecnologias da plataforma, mas com o passar do tempo, novos recursos foram sendo adicionados. Para se ter uma ideia, a Servlet 1.1 foi divulgada em janeiro de 1999 e o JSP 1.0 em junho deste mesmo ano. Isso mostra que na JPE de 1998, o que existia era apenas a versão 1.0 de Servlet e alguns fragmentos do que viria a ser as JSPs. Em 1999, a Sun iniciou a implementação do servidor de referência para a plataforma Java EE, que ficou conhecido como GlassFish.

Apesar da JPE anunciada em 1998 não possuir muitos recursos, não demorou muito para que ela ganhasse corpo. Neste mesmo ano, a Sun lançou a segunda versão da linguagem Java, abreviada como J2SE. Seguindo esta mesma linha, em dezembro do ano seguinte ela atualizou a versão corporativa da plataforma e a renomeou para J2EE 1.2, onde o número 2 indicava a versão da linguagem e 1.2 o kit de desenvolvimento. Nesta versão, além da atualização da linguagem, foram incluídas as tecnologias Servlet 2.2, JSP 1.1, EJB 1.1, RMI-IIOP 1.1, JavaMail 1.1,

Java Naming and Directory Interface (JNDI) 1.2, JDBC Standard Extension 2.0, Java Message Service (JMS) 1.0, Java Transaction API (JTA) 1.0 e JavaBeans Activation Framework (JAF). As versões J2EE 1.3 e 1.4 seguiram o mesmo padrão de nome, mantendo o número 2 da linguagem e alterando apenas o número do kit. Contudo, como a nomenclatura ficou confusa em função das duas numerações, no lançamento da versão 1.5 a Sun renomeou mais uma vez a plataforma, unificando as versões da linguagem e do kit, e esta passou a ser chamada apenas de Java EE 5. A **Figura 1** apresenta um histórico de sua evolução e a **Figura 2**, as APIs que compõem a plataforma atual, a Java EE 7.

Até a versão 5 da plataforma Java EE, os servidores de aplicações que desejassem ser compatíveis precisavam implementar todas as APIs especificadas. Para simplificar essa questão, a plataforma Java EE 6, divulgada em dezembro de 2009, flexibilizou o processo de homologação criando Perfis. Os Perfis definem subconjuntos de tecnologias da plataforma Java EE, procurando atender as demandas arquiteturais dos sistemas corporativos e tornando os servidores de aplicações mais leves. Para saber mais sobre Perfis, veja o artigo de Vítor Souza, **Java EE 6 Web Profile**, publicado na edição 100 da Revista Java Magazine.

Após a aquisição da Sun Microsystems em abril de 2009, a Oracle passou a ser a empresa responsável por homologar os servidores compatíveis com a plataforma Java EE. A lista completa dos servidores certificados pode ser vista no site da Oracle, página *Java EE Compatibility*. Dentre eles, podemos ver o Apache TomEE 1.0, servidor de aplicações analisado neste artigo e que é certificado com o Perfil Web do Java EE 6.

0 que é um servidor de aplicações?

Do ponto de vista técnico, um servidor de aplicações, diferentemente de um servidor web, cujo objetivo principal é expor conteúdo HTML para os navegadores, é um servidor que oferece serviços que cuidam de aspectos como segurança, transações,

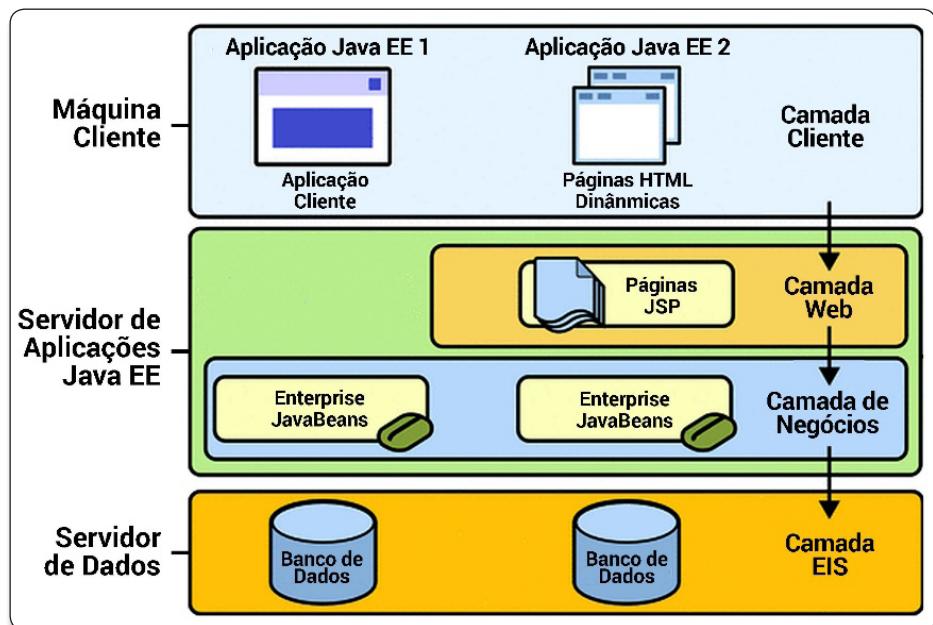


Figura 3. Arquitetura em Camadas

clusterização, controle de falhas, balanceamento de carga, etc.

De uma forma mais didática, poderíamos defini-lo como um gerenciador de aplicações que tem como objetivo principal fornecer dados para sistemas locais ou remotos, cuidando das questões técnicas. Ao utilizar um servidor de aplicações, os desenvolvedores se preocupam apenas com a lógica de negócio e o servidor assegura, de forma transparente, que as transações ocorram regularmente.

A forma mais comum de utilização do servidor de aplicações é através de uma solução desenvolvida em 3-camadas (vide **Figura 3**), na qual existe um cliente de interfaces gráficas com o usuário (GUI) que roda em uma máquina separada, um servidor que processa as requisições na camada web e as regras de negócio em outra camada e uma terceira que cuida de aspectos como banco de dados, transações e fornece os serviços de segurança, acesso a dados e persistência.

Em algumas arquiteturas, o servidor de aplicações também é utilizado como servidor web. Nesses casos, as camadas de apresentação e de negócios se misturam, fornecendo recursos para a construção de páginas dinâmicas, serviços de clusterização, controle de falhas, balanceamento

de carga, entre outros, em apenas uma máquina.

Servidores de aplicações Java

Como a plataforma Java se mostrou uma tecnologia robusta e promissora no desenvolvimento de aplicações do tipo cliente-servidor, várias implementações da especificação do Java EE foram desenvolvidas. Dentre elas, podemos citar:

- **Jetty** – Servidor Java web de código aberto, baseado no protocolo HTTP e na API Servlet. Foi desenvolvido pela Eclipse Foundation como um projeto independente. Tornou-se muito popular e usado em produtos da Apache, como ActiveMQ,



Geronimo, Maven, Spark, no Google App Engine, Eclipse e na API de Streaming do Twitter. O servidor está atualizado com a última versão da Servlet API e suporta JSP e recursos como AJP, JASPI, JMX, JNDI, OSGi, SPDY e WebSocket;

- **JBoss Application Server** ou apenas JBoss AS – Servidor de aplicações de código aberto, disponível sob a licença GNU Lesser General Public License, é utilizado para construir, distribuir e hospedar aplicações e serviços Java de grande volume transacional. Em 2014, a JBoss renomeou o servidor para WildFly;

- **GlassFish** – Servidor de aplicações de código aberto que foi originalmente criado pela Sun Microsystems e agora pertence à Oracle Corporation;

- **Apache Geronimo** – Servidor de aplicações de código aberto desenvolvido pela ASF e distribuído sob a licença Apache. É compatível com a especificação Java Enterprise Edition 6 e suporta tecnologias como JMS, EJBs, Connectors, Servlets, JSP, JSF, Unified Expression Language e JavaMail;

- **WebLogic** – Servidor de aplicações comercial desenvolvido inicialmente por uma *start-up* chamada WebLogic, criada em 1995 e comprada pela BEA Systems em 1998. Pertence atualmente à Oracle Corporation, que comprou a BEA Systems em 2008. Com codinome Tengah, o WebLogic foi o primeiro servidor de aplicações J2EE;

- **Resin** – Servidor web e de aplicações criado pela Caucho Technology. Está disponível sob a licença GPL e também em uma versão comercial. Como diferencial, suporta o Java EE e também o *mod_php/PHP*, através de um mecanismo conhecido como Quercus. É um servidor mais antigo do que o Apache Tomcat;

- **WebSphere Application Server (WAS)** ou IBM WebSphere – Servidor de aplicações desenvolvido pela IBM em 1998. Roda nas plataformas Windows, Linux, Unix e AS/400 e até mesmo em mainframes. O WebSphere também é um produto que segue as especificações para execução e hospedagem de aplicações Java EE.

Existem ainda os servidores OC4J da Oracle, Virgo da Eclipse, JOnAS, Blazix e outros mais que não foram citados neste artigo pela baixa representatividade no mercado.

De acordo com o gráfico apresentado na **Figura 4** é possível observar que o Tomcat é, desconsiderando o fato de que alguns servidores são completos e ele implementa apenas algumas especificações da plataforma Java EE, um dos servidores de aplicações Java mais utilizados do mercado.

Apache Tomcat/TomEE

No desenvolvimento de aplicações Java de pequeno ou médio porte, os recursos disponíveis no Apache Tomcat quase sempre são suficientes. No entanto, à medida que os projetos crescem, arquitetos e programadores se veem mergulhados em um emaranhado de bibliotecas e suas dependências. A necessidade de criar aplicações web complexas que dependem de um grande número de APIs – como JavaServer Faces, Java Persistence API, Injeção de Dependências, entre outras – e, consequentemente, de bibliotecas que as implementam, torna o dia a dia dos desenvolvedores mais complexo. Ainda assim, aplicações corporativas de médio ou grande porte nem sempre precisam da pilha completa de recursos disponível na plataforma Java EE. Tudo que se precisa é de um contêiner web compatível com os recursos de clusterização e replicação que foram incluídos no Tomcat 5.0, em 2004.

A complexidade das plataformas J2EE 1.2, 1.3 e 1.4 gerou uma forte discussão sobre sua aplicabilidade que durou anos. Rod

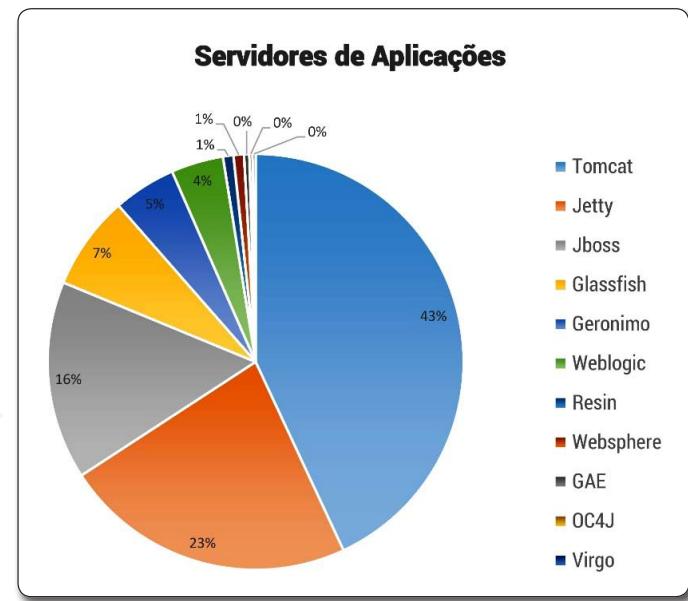


Figura 4. Distribuição do mercado de servidores de aplicações Java [SOR, 2013]

Johnson, criador do Framework Spring, um dos maiores críticos destas plataformas, concentrou suas críticas na dificuldade de criação e manutenção dos EJBs. Pensando nisso, na definição da plataforma Java EE 6, alguns recursos foram incluídos para simplificar o processo de desenvolvimento e distribuição das aplicações. Com a criação dos Perfis, assunto discutido em detalhes na edição 100 da Java Magazine, a pilha de componentes para se criar uma aplicação compatível com a plataforma Java EE foi cortada ao meio. Deste modo, as implementações de algumas interfaces, desnecessárias em várias circunstâncias, foram removidas e os EJBs que até então só podiam ser distribuídos em pacotes EARs, agora também podem ser empacotados em arquivos WARs sem a necessidade de criar arquivos especiais de configuração. Assim, a partir da versão 6 da plataforma Java EE, os recursos necessários à aplicação podem ser melhor utilizados, deixando-a mais leve ao carregar apenas o que realmente agrupa valor ao seu funcionamento.

TomEE, um servidor de aplicações corporativas leve

Em 1999, David Blevins, membro da comunidade Java e atual integrante da JSR-366, voltada para a plataforma Java EE 8, desenvolveu uma implementação de código aberto da especificação EJB integrável ao Apache Geronimo que ficou conhecida como OpenEJB. Anos mais tarde, esta implementação tornou-se um projeto da ASF e foi reescrita para integrar-se ao Tomcat. Da integração entre OpenEJB e Tomcat surgiu a versão corporativa chamada de TomEE.

Apesar do Tomcat ser um servidor Java web com grande expressividade no mercado, ele não tem suporte a transações, pool de conexões, JNDI globais, segurança integrada, anotações como `@Resource`, `@PersistenceUnit`, `@Inject`, `@EJB`, `@PersistenceContext`, `@DataSourceDefinition`, `@WebService`, entre outras. Diante disso, o Apache TomEE foi concebido para suprir esta lacuna, mas mantendo o mesmo princípio de seu “irmão”, de ser um servidor simples e leve. Confirmando o que acabou de ser dito, a filosofia da equipe de desenvolvimento é: **Ser Simples**, evitar complexidade e deixar os usuários trabalharem. **Ser Tomcat**, trabalhar com os mesmos recursos e ferramentas que ele provê. **Ser Certificado**, ou seja, tornar-se uma alternativa para qualquer servidor que implemente o Perfil Web. O objetivo é que mesmo mantendo a leveza e robustez do servidor Apache Tomcat, ele ofereça aos desenvolvedores os recursos necessários para se ter uma aplicação corporativa robusta em um servidor de aplicações leve.

A primeira versão oficial do TomEE foi lançada em abril de 2012 e logo conseguiu a certificação do Perfil Web Java EE 6. Agora, está perto de ser um servidor totalmente compatível com o Java EE 7. Blevins disse: *“Eu penso que um dos fatores de nosso sucesso é o princípio de manter tudo o que o Tomcat já tem. O TomEE é a versão Java EE do Tomcat, o mais popular servidor de aplicações Java, e é suportado por todas as ferramentas que suportam o Tomcat, mesmo que elas não saibam disso. Enquanto o TomEE não é distribuído com o Tomcat, algumas IDEs já adicionaram suporte para aproveitarem os recursos*

extras que o TomEE traz. JRebel foi a primeira em 2012, depois IntelliJ, Jelastic e mais recentemente NetBeans, todos apoiadores orgulhosos. Atualmente o que nossos usuários mais querem é o suporte do TomEE para Java EE 7, e vários dos projetos Apache que são necessários para o Java EE 7 estão quase prontos”.

No desenvolvimento do Apache TomEE, os desenvolvedores optaram por seguir um caminho diferente de outros servidores disponíveis no mercado. A maioria dos servidores Java EE, que utilizaram o Tomcat em seu núcleo, transformaram-no em um container EJB alterando sua arquitetura. O TomEE, ao contrário, incorpora componentes a ele, tornando-o em um servidor compatível com o Java EE sem mexer em sua estrutura.

Na **Tabela 2** é possível observar quais APIs e implementações foram acrescentadas.

APIs	Implementações
Servlet, JSP e JSTL	Nativas do Tomcat.
JPA	Apache OpenJPA
JSF	Apache MyFaces
EJB	Apache OpenEJB
JMS	Apache ActiveMQ
CDI	Apache OpenWebBeans
Bean Validation	BVal
JAX-RS e JAX-WS	Apache CXF
JTA	Apache Geronimo Transaction
Javamail	Apache Geronimo JavaMail
Connector	Apache Geronimo Connector

Tabela 2. APIs incorporadas ao Tomcat



Componente	JSR	Tomcat	Perfil Web*	JAX-RS*	Plus
Servlets 3.x	315	✓	✓	✓	✓
JavaServer Pages (JSP)	245	✓	✓	✓	✓
Java Standard Tag Library (JSTL)	52	✓	✓	✓	✓
JavaServer Faces (JSF)	314		✓	✓	✓
Java Transaction API (JTA)	907		✓	✓	✓
Java Persistence API (JPA)	317		✓	✓	✓
Java Contexts and Dependency Injection (CDI)	299		✓	✓	✓
Java Authentication and Authorization Service (JAAS)	196		✓	✓	✓
Java Authorization Contract for Containers (JACC)	115		✓	✓	✓
JavaMail API	919		✓	✓	✓
Bean Validation	303		✓	✓	✓
Enterprise JavaBeans (EJB)	318		✓	✓	✓
Java API for RESTful Web Services (JAX-RS)	311			✓	✓
Java API for XML Web Services (JAX-WS)	224				✓
Java EE Connector Architecture	323				✓
Java Messaging Service (JMS)	919				✓

Tabela 3. Distribuições do Apache TomEE com base na especificação Java EE 6

Do ponto de vista do desenvolvimento, não importa quais implementações estão acopladas ao servidor, elas são transparentes. Ou seja, o que importa é poder utilizar os recursos da plataforma Java EE em um servidor compatível, independentemente de quais bibliotecas os implementem.

Distribuições do Apache TomEE

Na versão atual, o TomEE possui três distribuições: **Perfil Web** (*Web Profile*), **JAX-RS** e **Plus**. A primeira distribuição, compatível com o Perfil Web da especificação Java EE 6, é a menor delas, com apenas 27MB. Já a segunda, JAX-RS, também é construída com base no Perfil Web, mas conta também com a adição do suporte a JAX-RS, através de uma versão simplificada do Apache CXF. Por fim, a Plus fornece um pacote completo com todos os componentes disponíveis para TomEE, incluindo JMS, JAX-WS e JCA. Porém, esta última ainda não possui a certificação Java EE. Na Tabela 3 é possível ter uma visão clara das distribuições e os componentes que elas adicionam ao Tomcat. As colunas marcadas com asterisco referem-se àquelas certificadas como compatíveis ao Perfil Web da especificação Java EE 6.

O desenvolvedor ainda tem a opção de baixar a versão TomEE PluME, que, para a especificação JavaServer Faces, utiliza a implementação Mojarra, e para a Java Persistence API, utiliza o Eclipse Link. As letras “ME” da palavra PluME referem-se, respectivamente, às iniciais das implementações Mojarra e Eclipse Link.

Há, também, a possibilidade de baixar o servidor OpenEJB 4.7.2 e rodá-lo de forma independente, ou sua versão chamada de “Drop-in WARs”, que pode ser integrada diretamente ao Tomcat. Este pacote é útil para aplicações previamente desenvolvidas que utilizam, em produção, o servidor Tomcat. Assim, basta adicionar o módulo OpenEJB a ele para torná-lo em um servidor de aplicações corporativas.

Preparando-se para o Java EE 7 e o futuro

O Apache TomEE está na versão 1.7.2, que roda sobre o Tomcat 7 e é voltado para a plataforma Java EE 6. Já a equipe do projeto



está concentrada no desenvolvimento do Apache TomEE 2.0.x, que rodará sobre o Tomcat 8.x e implementará os recursos da plataforma Java EE 7. Ao que tudo indica, esta nova versão será lançada como Apache TomEE 7.0, em referência à atual plataforma corporativa do Java.

O poder de fogo do Apache TomEE

No decorrer do artigo, vimos a origem do Apache Tomcat, seus recursos em relação às especificações e servidores Java EE e as implementações que foram adicionadas a ele para torná-lo um servidor completo de aplicações corporativas, o Apache TomEE. Obviamente, como o artigo é essencialmente teórico, ainda não foi possível testar seu poder de fogo. Portanto, em uma edição futura, vamos mergulhar em exemplos práticos para experimentar os recursos Java EE incluídos nele.

Neste momento, aqueles que desejarem experimentá-lo, podem baixar os exemplos no site do TomEE (veja a seção **Links**), que discorrem sobre os assuntos: Session Beans, EJB Lite e EJB Legados, referenciamento de EJBs, JMS e MDBs, DataSources, Java EE Connectors, Meta-Annotations, REST, EntityManagers, Transactions, Proxy Beans, CDI, web services, segurança, técnicas de teste e muito mais.

Ainda que o mercado possua inúmeros servidores de aplicações, o Apache Tomcat é um servidor web maduro e robusto capaz de responder por boa parte das aplicações Java web que rodam atualmente na Internet. Agora, com sua distribuição TomEE, compatível com o Perfil Web, a Apache aumenta ainda mais o seu poder de fogo, elevando seu servidor web à categoria de servidor de aplicações corporativas com recursos para competir com os grandes servidores pré-estabelecidos no mercado.

Além disso, a distribuição Apache TomEE simplificará o dia a dia de milhares de programadores que utilizam Servlets e JSPs, mas precisam de bibliotecas adicionais para complementar a arquitetura de suas aplicações.

Links:

Endereço do TomEE com diferentes exemplos.

<http://tomee.apache.org/examples-trunk/index.html>

Java EE and GlassFish Server Roadmap Update.

https://blogs.oracle.com/theaquarium/entry/java_ee_and_glassfish_server

Servlet History.

<https://jamesgdriscoll.wordpress.com/2010/02/09/servlet-history/>

Introducing the new Servlet API 2.1.

<http://www.javaworld.com/article/2076838/java-web-development/introducing-the-new-servlet-api-2-1.html>

WebLogic Offers Industry's First Implementation of Enterprise JavaBean 1.0 Specification.

<http://www.prnewswire.com/news-releases/weblogic-offers-industrys-first-implementation-of-enterprise-javabean-10-specification-78042537.html>

Most popular application servers.

<http://www.javacodegeeks.com/2013/03/most-popular-application-servers.html>

BOND, Martin. Tomcat Kick Start. 1 ed. Indianapolis, EUA: Sams, p. 12. Novembro, 2002.

BRITAIN Jason, DARWIN Ian F.; Tomcat: The Definitive Guide. 2 ed. Sebastopol, EUA: O'Reilly, p. 36-37. Outubro, 2007.

WUTKA, Mark. Special Edition. Using Java Server Pages and Servlets. Indianapolis, EUA: Quê, p. 668. Outubro, 2000.

HAEFEL, Richard Monson-. Enterprise JavaBeans. 3 ed. Sebastopol, EUA: O'Reilly, p. 3. Setembro, 2001.

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Autor



Geucimar Brilhador

geucimar@gmail.com



Possui as certificações OCP-JP e OCP-JWCD. É Professor do Bacharelado em Sistemas de Informação da Universidade Positivo, em Curitiba-PR, Especialista em Engenharia de Software pela Universidade Federal do Paraná (UFPR), Especialista em Java e Desenvolvimento para Dispositivos Móveis pela Universidade Tecnológica Federal do Paraná (UTFPR), formado em Sistemas de Informação pela UFPR, atua como Professor, Pesquisador, Analista e Desenvolvedor.

DevOps: Uma abordagem prática

Veja nesse artigo como utilizar o NetBeans, Maven, GitHub, Jenkins e JUnit para melhorar o desenvolvimento em equipe

No atual cenário de ascensão de novas soluções e tecnologias, pode-se citar a cultura DevOps entre os grandes destaques. Este novo conceito, que objetiva criar uma cultura de colaboração, permite aumentar o fluxo de trabalho contínuo, valendo-se principalmente da automação de tarefas. Com este fluxo de trabalho, além da padronização das atividades de desenvolvimento, espera-se que o produto de software ganhe em estabilidade e robustez. Ademais, a adoção deste quadro de trabalho que a cultura DevOps acaba criando contribui com a redução do tempo consumido no desenvolvimento, uma vez que um software pode ser pensado como um processo de fabricação em uma linha de montagem.

Neste momento vale ressaltar que a cultura DevOps não é composta por uma solução pronta e empacotada, mas sim pela adoção de diferentes opções do mercado. Portanto, não é correto afirmar que exista a solução perfeita que resolva todos os problemas a que o DevOps se propõe. É necessária, então, a opção por um conjunto de ferramentas, dentre as diversas oferecidas pelo mercado, de acordo com as condições encontradas na empresa, como o escopo do projeto, o sistema operacional utilizado, a habilidade da equipe desenvolvedora, dentre outras variáveis. Como opções mais comuns, podemos citar o Maven, JUnit, Jenkins e Git. Em conjunto, tais tecnologias proporcionam redução do tempo de desenvolvimento e automação de tarefas.

A aplicação dos princípios da cultura DevOps, por meio da incorporação de certas tecnologias no processo de desenvolvimento, impõe mudanças no fluxo de trabalho ao qual as equipes estão habituadas, uma vez que ajustes deverão ser feitos para adequar-se ao modo pelo qual as ferramentas interagem. Os benefícios são observados em qualquer circunstância, mas ganham

Fique por dentro

Este artigo apresenta uma introdução ao processo de desenvolvimento guiado pelos princípios da cultura DevOps. Esta demonstração é feita através da implementação de um exemplo básico de um projeto Java utilizando diversas tecnologias, a saber: IDE NetBeans, versionamento de código com o Git e armazenamento deste em um servidor remoto GitHub, testes unitários com JUnit e integração contínua com o Jenkins. Apesar de utilizar um grande número de ferramentas, o artigo tem foco em descrever a integração entre elas, possibilitando que um iniciante em programação possa utilizar todas elas sem que para isso precise ter conhecimentos avançados. A arquitetura obtida com essa integração faz com que o Jenkins verifique mudanças junto ao repositório e, assim que detectá-las, construa o projeto e execute os testes, notificando a equipe em caso de falhas ou sucesso.

em importância quando o processo de desenvolvimento de software é realizado integrando equipes responsáveis pelo desenvolvimento (programadores, por exemplo) e infraestrutura (responsáveis por colocar a aplicação em execução), principalmente por oferecer mecanismos para resolver problemas observados nesta integração. Estes problemas dizem respeito, em sua maioria, a erros que possam ocorrer durante o *deploy*, processo no qual uma equipe atribui à outra a responsabilidade: a equipe de desenvolvimento atribui o problema a uma infraestrutura inadequada para execução e a equipe de infraestrutura culpa a equipe de desenvolvimento por não implementar (ou comunicar) corretamente os requisitos da aplicação, por exemplo.

É possível observar, no entanto, a adoção de um quadro de trabalho diferente do tradicional, em novas empresas de desenvolvimento de software, pois uma única equipe é responsável por todo o processo, desde as fases de construção até a implantação da aplicação, devendo esta mesma equipe resolver qualquer problema que ocorra inclusive na disponibilização junto ao cliente. A proposta do DevOps não atende exatamente a este

cenário, e sim a um cenário com equipes distintas, cada uma com suas responsabilidades (desenvolvimento e infraestrutura, por exemplo).

De maneira geral, com base nas vantagens observadas com a utilização das tecnologias DevOps, pode-se dizer que muitos processos são simplificados (como o gerenciamento de código-fonte), que a comunicação entre os membros da equipe é melhorada (como no caso de falha ou sucesso ao executar um teste, em que toda a equipe é notificada imediatamente) e que a automação das atividades traz grandes benefícios. Esta automação cria um ambiente formal, o qual possibilita que novas aplicações sejam criadas adotando as mesmas ferramentas já instaladas, configuradas e padronizadas. Deste modo, softwares são produzidos de modo repetitivo, em um ambiente controlado.

Considerando o cenário de desenvolvimento de software e os benefícios da cultura DevOps, este artigo apresenta uma proposta de integração de ferramentas sem se preocupar com o tamanho da aplicação a ser construída, pois a proposta atende a soluções de diferentes tamanhos e mantém o foco na arquitetura.

Criação de um projeto Maven

O Maven é uma ferramenta pertinente à cultura DevOps, pois seu propósito está na automação do processo de compilação de projetos, podendo ser utilizado com diferentes linguagens (sendo mais comum em projetos Java). Além disso, o Maven é capaz de efetuar o gerenciamento de todas as dependências (ou bibliotecas) externas ao projeto, de maneira automatizada, bastando que o desenvolvedor especifique qual dependência deseja utilizar e sua versão. E como mais uma facilidade, para seu uso no NetBeans não é necessária instalação, pois esta é ocorre junto com a instalação da IDE.

A criação de um projeto Maven gera uma estrutura de arquivos e pacotes pré-definida conforme o tipo de aplicação e suas configurações são especificadas no arquivo *pom.xml*. Este arquivo contém todas as informações do projeto e, de maneira simplificada, sua principal função está na especificação das dependências ao funcionamento e compilação do projeto.

Antes de iniciar qualquer codificação, é preciso definir que se trabalhará com esta tecnologia no projeto, ou seja, para utilização do Maven, é necessária a criação de um “Projeto Maven” no NetBeans (ou outra IDE). Em projetos já existentes, nos quais a equipe opte pela sua adoção, é possível adicioná-lo manualmente, mas para isso deve-se adequar a estrutura do projeto ao formato requisitado pelo Maven, o que pode implicar em muitas alterações.

Com o intuito de facilitar esta etapa, o NetBeans simplifica a criação de projetos Maven por meio de um assistente (acessado pelo menu *Arquivo > Novo projeto*). O assistente (também conhecido como *wizard*) é composto por diferentes janelas, nas quais podemos informar a configuração. Na primeira janela, selecione a primeira opção *Aplicação Java*, que trata do desenvolvimento de uma aplicação Java básica. Na segunda, é solicitado o nome do projeto (identificado neste exemplo como “SimpleJavaProject”) e alguns parâmetros de configuração, a saber: *id do grupo*, *id do*

artefato e a *versão*. O parâmetro *id do grupo* corresponde ao identificador da organização (br.com.google, por exemplo). O segundo parâmetro, *id do artefato*, usualmente corresponde ao nome do projeto, e aliado ao *id do grupo* representa o identificador único deste (considerando que não existirão dois projetos de mesmo nome, em uma mesma organização ou domínio). Por fim, o parâmetro *versão*, como esperado, indica a versão do artefato.

Ao concluir o assistente de criação do projeto, será criada uma estrutura em pacotes, dependências e arquivos, sugerindo uma organização para a aplicação, conforme demonstra a **Figura 1**. Observe a existência de um pacote chamado *Dependências Java*, no qual pode ser verificado o JDK em uso, e um pacote chamado *Dependências*, para as dependências externas que podem surgir em função da adição de funcionalidades à aplicação como, por exemplo, o uso de determinado tipo de banco de dados.

No desenvolvimento de uma aplicação é comum o uso de bibliotecas externas. Com o Maven, como já explicitado, a descrição destas é feita no *pom.xml*, especificando-se o nome e versão para que, antes da construção do projeto, esta ferramenta efetue o download de todos os arquivos indicados. Deste modo, para transferir a solução em desenvolvimento para outro computador, não é preciso empacotar todas as bibliotecas necessárias, pois estas serão obtidas automaticamente pelo Maven na primeira construção do projeto junto ao novo local.

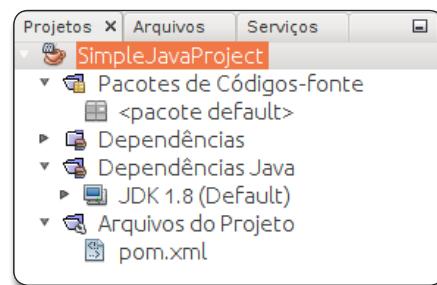


Figura 1. Estrutura de arquivos e pacotes

Com o NetBeans tem-se outra vantagem na utilização do Maven, pois não é necessário editar manualmente o arquivo *pom.xml*, e sim fazê-lo por meio de uma janela gráfica. Essa janela pode ser exibida a partir de um clique com o botão contrário do mouse no pacote *Dependências*, apresentado na **Figura 1**.

Concluída a etapa de criação do projeto Maven, podemos seguir com a codificação da aplicação para manipulação de informações relativas à uma pessoa, conforme o diagrama de classes apresentado na **Figura 2**, que exibe a classe **Person**. Como o número de atributos e métodos é irrelevante para demonstração da integração, serão consideradas somente informações como nome, sobrenome, gênero, peso e altura. Além dos tradicionais métodos de atribuição/recuperação, também conhecidos como setters/getters, será proposta a implementação de um método que permita o cálculo do índice de massa corpórea, *calculateIMC()*.

A implementação desta classe permitirá a instanciação de objetos do tipo **Person** e os valores dos atributos deste objeto, quando

alimentados com informações válidas, possibilitarão a utilização do método `calculateIMC()`, além de permitir a demonstração da execução de testes com o JUnit nas próximas seções. Entende-se por informações válidas, valores que não geram a chamada de exceções no código e interrompam a execução do programa.

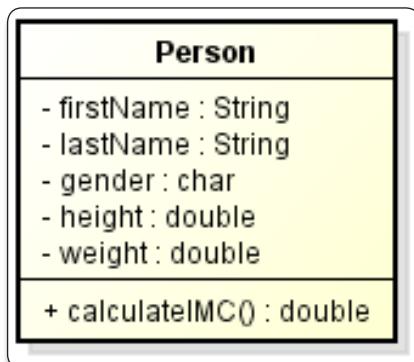


Figura 2. Representação gráfica da classe Person

Versionamento de código com o Git

O versionamento do código-fonte de um projeto por meio do Git possibilita que o desenvolvedor (ou a equipe) identifique os trechos cujo código passou por mudanças, bem como os responsáveis por elas. Este controle auxilia no gerenciamento e possibilita que todos trabalhem em uma versão atualizada e comum do projeto.

Nota

É possível utilizar o versionamento de código sem o uso de um projeto Maven e a recíproca também é verdadeira. O uso destas tecnologias combinadas, no entanto, proporciona um ganho ainda maior na automação de tarefas e controle do projeto.

Para iniciar o controle de versões que é instalado automaticamente com o NetBeans, basta clicar com o botão contrário do mouse sobre o nome do projeto e acessar a opção *Controle de Versão > Inicializar Repositório Git*. Será necessário informar ainda o caminho em que o repositório deve ser iniciado, o qual é recomendado coincidir com o diretório do projeto. O versionamento pode ser feito a qualquer momento, levando-se em conta que só passará a controlar as alterações de código depois que iniciado.

Uma vez iniciado o repositório, já é possível observar seu funcionamento com a utilização da cor verde no nome da classe, identificando as classes não adicionadas ao repositório, tanto na hierarquia de pacotes do projeto, como na barra de títulos da janela do editor de código (vide Figura 3). Note nesta imagem que a classe **Person** está destacada na cor verde, indicando que trata-se de um arquivo ainda não presente no repositório. Se o arquivo contiver alterações em relação ao arquivo presente no repositório, o mesmo será apresentado em cor azul, sob o status *modificado*. Ademais, a execução do Git pode ser identificada por meio de um pequeno ícone em tom azul (na forma de uma base de dados) próximo ao nome do projeto e seus pacotes.



Figura 3. Inicialização do repositório Git junto aos arquivos do projeto

Para efetuar o controle de alterações, o servidor Git faz uso de três áreas distintas, conhecidas no IDE NetBeans como: *Árvore de trabalho*, *Índice* e *Head* (Figura 4). A primeira destas três áreas corresponde à *Árvore de trabalho* e compreende o local onde os arquivos do projeto estão localizados fisicamente. A partir do momento em que o controle de alterações for executado, estes arquivos passam a ser monitorados e o Git a ser o responsável por gerenciar essas alterações. No entanto, é o desenvolvedor quem dirá que as alterações foram finalizadas e que se tem uma nova versão estável daquele arquivo. Para fazer isso, o desenvolvedor deve adicionar o referido arquivo à área de *Índice* (conhecida também como *Stage*). Todo esse processo pode ser aplicado a um conjunto de arquivos, sendo possível realizar modificações em mais de um arquivo e adicionar todos ao Índice de uma só vez, para então se ter uma nova versão estável do projeto. Com um conjunto de alterações estáveis, pode-se efetuar a operação de *commit*, indicando que os arquivos passaram para a área de *Head* do repositório.

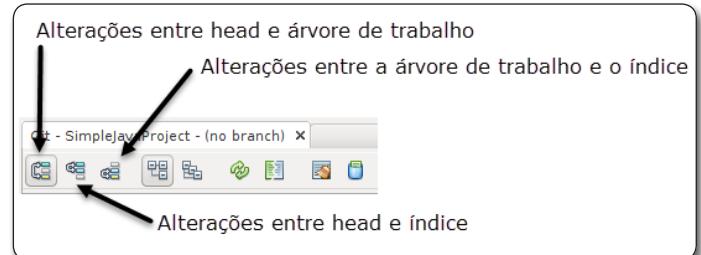


Figura 4. Painel para visualização das áreas do Git

O painel apresentado na Figura 5 ilustra a relação de arquivos com seu nome, status (adicionado/modificado) e caminho, sendo possível tomar decisões (como adicionar ao índice e realizar *commit*) individualmente, por meio de um menu de contexto. Neste momento, o IDE identificou que os arquivos *Person.java* e *pom.xml* foram “adicionados” ao projeto e ainda não estão versionados. Essas alterações são identificadas pelo “controle de alterações”, que exibirá neste painel informações que indicam se o arquivo foi editado e contém alterações em relação ao seu último *commit*.

Preferencialmente, um *commit* deve ser feito quando houver um conjunto de alterações e estas já tiverem sido verificadas como estáveis, com o devido tratamento dos erros de programação

que possam causar a interrupção do programa, caracterizando assim uma nova versão. É comum que cada *commit* efetuado corresponda a uma nova funcionalidade implementada no projeto.

Para que o controle de alterações registre corretamente todas as operações realizadas, depois de feita a edição em um arquivo de código-fonte, deve-se primeiramente adicioná-lo ao *Índice* e, posteriormente, efetuar o *commit*, adicionando o conjunto de alterações ao *Head* do repositório. Note que é possível fazer o *commit* diretamente, sem adicionar o arquivo ao índice, mas isso não é recomendado. A passagem pelas três áreas permite um melhor gerenciamento das versões, possibilitando recuperar determinada versão ou mesmo resolver arquivos com conflito.

Adicionar um arquivo ao *Índice* é uma operação simples e feita por meio de um menu obtido com o botão contrário do mouse sobre o arquivo desejado. Já a adição de um arquivo ao *Head* do repositório, por meio de uma operação de *commit*, requer mais informações, como autor e descrição das modificações efetuadas, utilizando uma janela específica para tal. Na interface apresentada para o *commit* (vide Figura 6), pode-se identificar o autor das alterações, bem como uma descrição destas, o que é fortemente recomendado. Essa informação possibilita aos demais membros da equipe (remotos ou não) o conhecimento sobre as alterações realizadas.

Ao clicar no botão *Fazer commit*, a classe **Person** é adicionada ao *Head* do repositório, e o Git continuará monitorando este arquivo em busca de alterações em relação à versão já existente no repositório. Identificado o processo de monitoramento do Git, será dada continuidade ao desenvolvimento do projeto através da adição de uma nova classe, chamada **App**, com o método principal (**main()**) dentro dela, responsável apenas pela exibição de uma mensagem em console.

Todas essas operações – sejam de adição, edição ou exclusão de arquivos – podem ser visualizadas no NetBeans, que exibe um histórico completo desde a inicialização do Git. Este relatório pode ser observado pelo menu *Equipe > Mostrar histórico*.

Git - SimpleJavaProject - (no branch) x		
Nome do arquivo	Status	Caminho
Person.java	-/Adicionado	...a/br/com/company/modelo/Person.java
pom.xml	-/Adicionado	pom.xml

Figura 5. Painel ‘mostrar alterações’ do Git com os arquivos pom.xml e Person.java adicionados

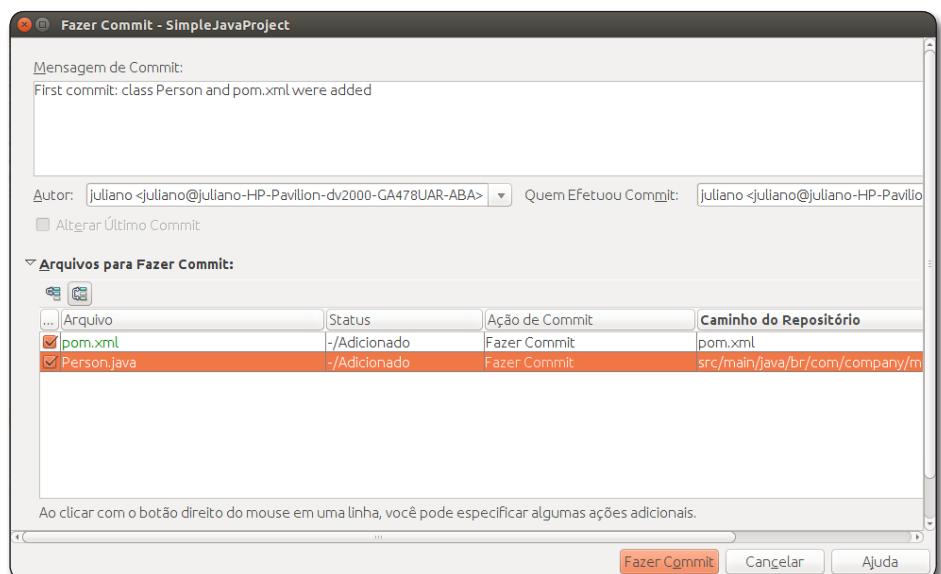


Figura 6. Interface pela qual é feito o commit do código-fonte

```

1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package br.com.company.app;
7
8  import br.com.company.modelo.Person;
9
10 /**
11  * @author juliano
12  */
13
14 public class App {
15     public static void main(String[] args) {
16         Person p = new Person("Juliano Rodrigo", "Lamb", 'm', 1.90, 80.0);
17         System.out.println("Main method");
18     }
19 }

```

Figura 7. Identificação das alterações no código-fonte da classe App

O IDE destaca ainda alterações em tempo de edição, em relação ao arquivo existente no repositório, sendo possível identificar a adição de código (realçado com marcador em verde em frente a linha) e remoção de código (realçado com marcador em vermelho em frente a linha). Na Figura 7,

por exemplo, é possível verificar que a classe **App** já está no repositório e que a instrução de **import** da linha 8 e a instanciação de um objeto **Person** na linha 16 foram adicionadas. Verifica-se ainda, que ao final do arquivo, entre as linhas 18 e 19, foi feita a remoção de código.

É importante ressaltar que o gerenciamento de versões que o Git realiza é refletido localmente, isto é, em um único computador. Para que seja possível o desenvolvimento de software de modo colaborativo, com o compartilhamento de código versionado entre membros geograficamente dispersos, é necessária a utilização de um serviço adequado a esse fim, como o GitHub.

Versionamento de código no NetBeans com Git e GitHub

Este tópico detalha os procedimentos necessários para o desenvolvimento de software colaborativo, utilizando o serviço de hospedagem GitHub. Existem outros serviços que possibilitam a cooperação remota, como o BitBucket, no entanto, visto a facilidade de operação e difusão perante a comunidade de desenvolvimento, o GitHub será o que analisaremos.

Nota

Existem muitos recursos adicionais para o gerenciamento de código utilizando o Git, seja integrado a uma IDE (assim como no NetBeans), seja via linha de comando. Pode-se adotá-lo ainda no versionamento de outros projetos de software, como em projetos de desenvolvimento Web e inclusive em outras linguagens.

Criação de um repositório

Uma vez iniciado o versionamento do código-fonte de um projeto com o Git, deve-se levar em conta que o NetBeans realiza todo o processo localmente. Considerando que as equipes de desenvolvimento encontram-se dispersas geograficamente, torna-se necessário disponibilizar o material através da nuvem. Para isso, a utilização do GitHub com o NetBeans não requer a instalação de nenhuma ferramenta especial, sendo feita a integração diretamente do ambiente de desenvolvimento. É necessária, entretanto, a criação de uma conta e escolhida uma opção de uso, entre gratuita (não é permitida a criação de repositórios privados) ou paga (pode-se criar repositórios privados).

A visibilidade de um repositório, sendo ela pública ou privada, faz com que este esteja acessível a qualquer pessoa na internet de duas maneiras: o **acesso** (operação identificada como *extrair*, no NetBeans) de arquivos do repositório e o **envio** (operação identificada como *expandir*, no NetBeans) de arquivos ao repositório.

O GitHub proporciona um espaço no qual a equipe pode centralizar todo o código-fonte na nuvem, criando assim um ponto único de acesso e com isso facilitan-

do o compartilhamento de informações e a comunicação entre desenvolvedores.

Para enviar um projeto ao GitHub é necessário acessar a URL do serviço, disponível na seção **Links**, e efetuar o cadastro de um usuário, através da opção *sign up*. Para isso, deve ser informado um login e uma senha, assim como a opção do plano.

Com o cadastro concluído, autentique-se informando os dados de login e senha para acesso à página inicial (*dashboard*) do GitHub. Essa página mostrará os repositórios do usuário (proprietário) e um registro de atividades em cada um dos repositórios em que o usuário colabora. Uma barra de menu, ao topo, acompanha toda a página e, por meio dela, é possível buscar por projetos abertos no GitHub, verificar as informações de sua conta e criar um repositório, conforme destaque na **Figura 8**.

Ao clicar na opção para criação de um repositório, a **Figura 9** é exibida. Neste momento deve ser informado o nome do repositório, sua visibilidade, dentre outras opções, como é o caso da descrição.

Para concluir o processo, clique em *Create Repository* ao final da página, sendo então encaminhado para a página inicial do repositório que acabou de criar. É junto a esta página principal que será possível observar os *commits* enviados e todo o código-fonte do projeto.

Enviar arquivos a um repositório

Assim como em um repositório local, as operações básicas ao se trabalhar com um repositório remoto são: enviar e acessar um arquivo. No NetBeans, estas operações são traduzidas como *Expandir* e *Extrair*, respectivamente, possibilitando o envio/recuperação das informações (código-fonte).

Estas funcionalidades são acessíveis nesta IDE por meio do menu *Git > Remoto*. Ao fazer a expansão (envio para a nuvem), serão solicitadas algumas informações, tais como URL do repositório, usuário e senha, como apresenta a **Figura 10**, correspondente à primeira interface do assistente do NetBeans para o envio remoto.

Após a interface da **Figura 10**, outras caixas de diálogo serão apresentadas,



Figura 8. Barra de menu do GitHub, com destaque para a opção de criação de um novo repositório

Figura 9. Interface para alimentação das informações necessárias à criação de um repositório

informando ao usuário o status deste commit em relação ao conteúdo do repositório, ou seja, o código-fonte pode estar sendo adicionado pela primeira vez, ou então, atualizado. Ao final do assistente os arquivos serão enviados e ficarão disponíveis na nuvem, sendo acessíveis conforme a visibilidade configurada no momento da criação do repositório. Na **Figura 11** é possível visualizar a página principal do repositório *LittleSimpleJavaProject*, criado após a conclusão dos passos da **Figura 10**.

Vale lembrar que o envio remoto é feito de forma manual. Assim, a cada nova atualização significativa do código-fonte, deve ser feito um *commit* local e, na sequência, encaminhado a um servidor remoto, por meio da operação de *expandir*, conforme demonstrado anteriormente, na **Figura 10**. Após realizado o upload, o GitHub fará a listagem das versões enviadas, permitindo o acompanhamento da evolução destas, apresentando o autor e a data de envio (vide **Figura 12**).

Como pode-se observar, cada “linha” corresponde a um *commit*, ou seja, a uma versão do projeto. Observe ainda que cada *commit* possui um identificador único (à direita), mas como este campo é criado automaticamente, é natural distinguir uma versão da outra através de sua descrição textual, a qual é feita junto ao NetBeans. Ressalta-se, deste modo, a importância do texto da mensagem informada ao efetuar o *commit*, já que esta é apresentada na página inicial do projeto, possibilitando identificar as alterações realizadas ao longo das diferentes versões.

Além do controle das versões, o GitHub oferece uma interface (vide **Figura 13**) em que é possível navegar pelos arquivos enviados e visualizar, em cada um deles, as adições destacadas em verde e em vermelho as linhas em que alguma instrução foi apagada.

Configuração do arquivo pom.xml

Como apresentado anteriormente, o arquivo *pom.xml* pode ser configurado manualmente, ou então, através da interface gráfica do NetBeans. Para utilizar o modo gráfico, deve-se localizar o pacote

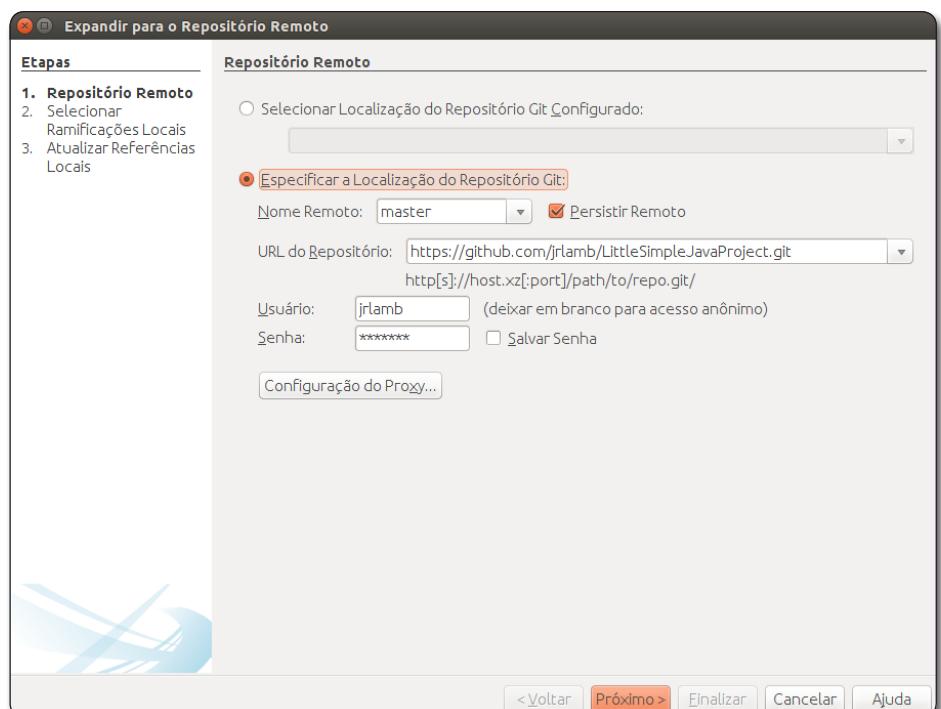


Figura 10. Expansão do projeto SimpleJavaProject para um repositório GitHub

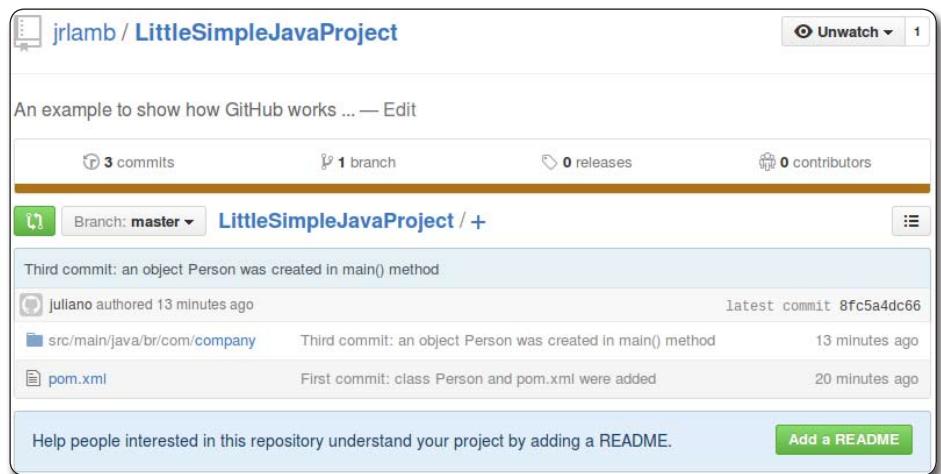


Figura 11. Visualização dos arquivos expandidos por meio do navegador

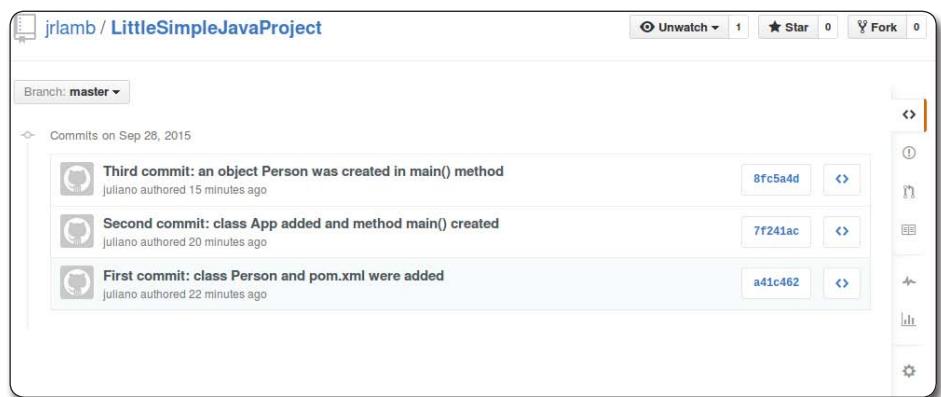


Figura 12. Histórico de versões de um projeto, enviadas ao GitHub

DevOps: Uma abordagem prática

The screenshot shows a GitHub commit history titled "Third commit: an object Person was created in main() method". It displays a single file change in "src/main/java(br/com/company/app/App.java)". The code adds a new import statement and creates a Person object in the main() method. The commit message is "Juliano authored 16 minutes ago".

Figura 13. Visualização do arquivo App.java e as alterações em relação ao último commit

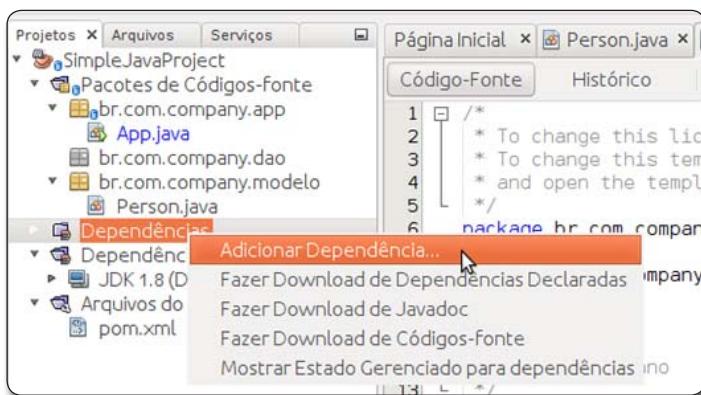


Figura 14. Menu para "Adicionar Dependência..."

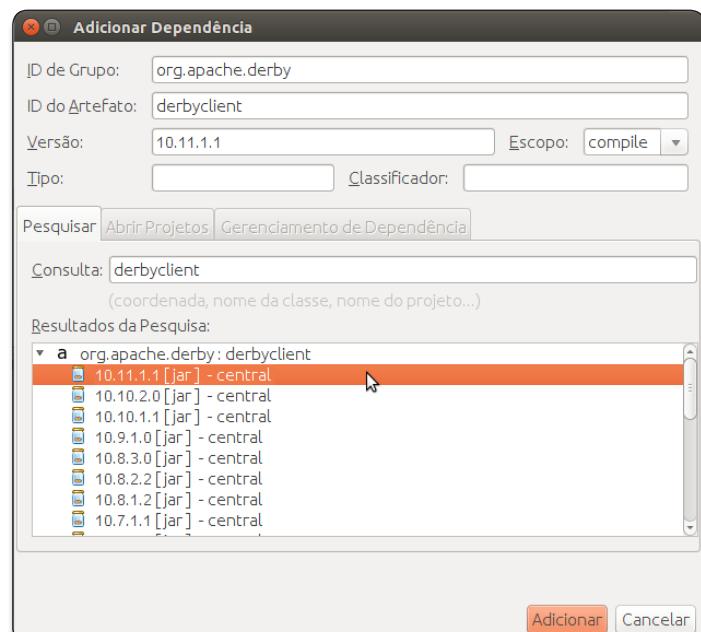


Figura 15. Interface para busca de bibliotecas

Dependências, junto à hierarquia de pacotes, clicar com o botão contrário do mouse, e então selecionar a opção *Adicionar Dependência...*, conforme demonstra a Figura 14.

Ao selecionar esta opção é apresentada a interface de busca exibida na Figura 15. Por meio desta é possível buscar por bibliotecas oferecidas por diferentes fornecedores. Como exemplo, pode-se citar as bibliotecas para conexão com bancos de dados relacionais. Ao adotar uma delas o desenvolvedor não precisará se preocupar em como sua aplicação se comunicará com o SGBD, e sim com as operações que a aplicação faz no banco. Um exemplo de biblioteca para conexão com bancos de dados relacionais é a *derby-client*.

Após selecionar a biblioteca e confirmar a operação por meio do botão *Adicionar* ao fim da tela, o arquivo *pom.xml* é automaticamente atualizado com a dependência. A Figura 16 mostra o código gerado automaticamente pela seleção de *derby-client*, por meio de um bloco com destaque em verde junto ao número da linha (este destaque é automático em função de que o controle de alterações está ativado). Vale ressaltar ainda que ao clicar em *Adicionar* já é iniciado o download dos arquivos da biblioteca solicitada, sendo possível acompanhar o progresso da atividade no rodapé do ambiente.

Concluído o download, as bibliotecas já estão disponíveis para uso e, com isso, o programador pode escrever o código correspondente à conexão com um banco Derby. O fato de se adicionar esta dependência, no entanto, não cria o banco, tampouco já conecta algum banco existente com a aplicação. Todas essas ações devem ser feitas pelo programador.

A grande vantagem deste processo está no fato que sempre a biblioteca *derby-client* estará disponível a todos os desenvolvedores do projeto, ou seja, o programador não precisará realizar o download manual dessa e, possivelmente, de muitas outras que podem compor um projeto. Além disso, no momento em que for adicionado um novo colaborador na equipe e este *clonar* o projeto, não será necessário preocupar-se com as bibliotecas, pois a especificação

presente no *pom.xml* garante que todas as dependências serão transferidas automaticamente para o computador.

Criação de testes com o JUnit

Os testes unitários são elementos fundamentais a todo processo de desenvolvimento que busca um software de qualidade, uma vez que permitem detectar a presença de erros precocemente, ou seja, antes que a aplicação seja disponibilizada aos usuários. Considerando que já estamos utilizando o controle de versão do Git integrado ao NetBeans, a escrita e execução dos testes pode ser feita diretamente no mesmo ambiente de desenvolvimento, já que este é integrado ao framework JUnit, o que facilita o processo e dispensa o uso de outra ferramenta para manipulação dos testes.

Uma classe de testes JUnit é estruturada com métodos voltados a identificar o início do teste, o seu término e, principalmente, validar o resultado produzido por alguma função específica do programa, comparando-o com um valor conhecido. Para escrever uma classe de testes, deve-se acessar o menu *Ferramentas > Criar/atualizar testes* e informar os parâmetros solicitados. O assistente criará, então, uma classe contendo a assinatura dos métodos e restará ao programador implementá-los, informando, por exemplo, os valores a serem comparados.

Um teste na classe **Person**, por exemplo, para ser executado, precisa da instânciação de um objeto **Person**, pelo qual o programador terá acesso aos valores dos atributos para calcular o IMC. Conhecidos os atributos e o valor que deverá ser resultante da rotina, o programador pode escrever o código do método de teste comparando a variável de retorno com o valor conhecido.

O código do teste para a função de cálculo do IMC pode ser observado na **Listagem 1**, onde são apresentados dois métodos: um comparando o valor de retorno com 0 e outro comparando com o valor esperado (~22.16). A lógica dos testes é algo que normalmente é definido pelo programador. Para este caso, por exemplo, foram estipuladas condições (*asserts*) para garantir que a função esteja produzindo valor diferente de zero, diferente de nulo

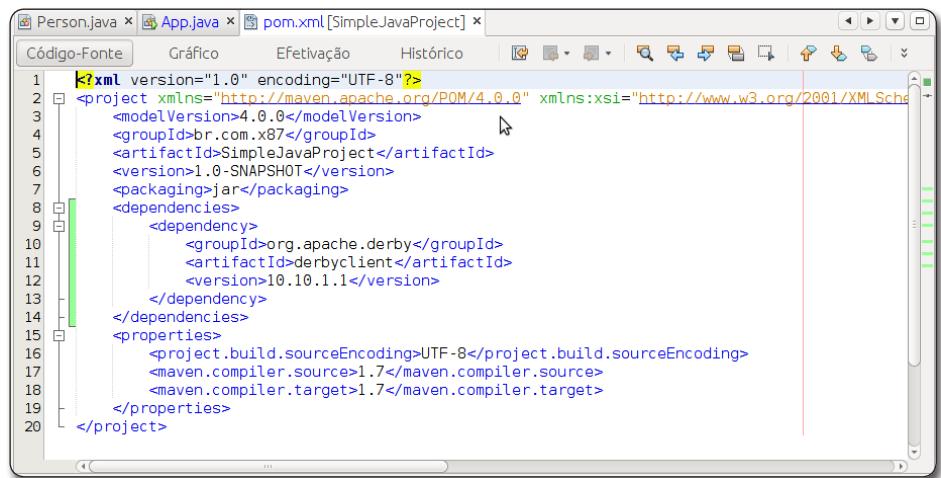


Figura 16. Arquivo *pom.xml* configurado com derby-client

e igual ao valor esperado. Ao atender a essas três restrições, pode-se afirmar que o método **calculateIMC()** está calculando o que deveria.

Na classe de teste, ambos os métodos tiveram suas assinaturas produzidas automaticamente pelo assistente (executado na criação da classe de teste), restando, portanto, ser implementada a lógica. Neste código, o primeiro método foi escrito certificando que o valor é diferente de 0.0 (instrução da linha 6). Já o segundo, certifica que o valor não é nulo (linha 14) e que deve ser é igual a ~22.16, conforme esperado (linha 15).

Ao executar essa classe de teste no NetBeans (pelo atalho *Ctrl+F6* ou então pelo menu *Executar > Testar Arquivo*), ambos deverão passar, o que garante que o programa está implementado com a lógica correta.

Integração com o servidor de integração contínua Jenkins

O Jenkins possibilita a execução de uma série de atividades relacionadas ao desenvolvimento de software, atuando como um servidor de integração contínua. Dentre essas atividades, será analisada a integração com o GitHub e a construção de um build, precedida pela execução de testes unitários.

Sua instalação é feita por meio de um arquivo WAR colocado no diretório *webapps* de um servidor Tomcat, possibilitando assim que seja acessado por meio da URL <http://localhost:880/jenkins> em qualquer

Listagem 1. Testes para o método de cálculo do IMC.

```
01. @Test
02. public void testCalculeIMCError() {
03.     System.out.println("calculeIMCError");
04.     double expResult = 0.0;
05.     double result = App.calculateIMC(person);
06.     assertEquals(expResult, result);
07. }
08.
09. @Test
10. public void testCalculeIMCSuccess() {
11.     System.out.println("calculeIMCSucess");
12.     double expResult = 22.1606648199446;
13.     double result = App.calculateIMC(person);
14.     assertEquals(expResult, result);
15.     assertEquals(expResult, result, 0.0);
16. }
```

browser. Logo que a instalação for concluída e o ambiente estiver disponível, é necessário configurar o Jenkins, efetuando a adição de um plugin para comunicação com o repositório do GitHub. Essas configurações são feitas via interface administrativa do próprio ambiente de integração contínua (Figura 17), de maneira intuitiva.

Antes de criar um job, deve-se configurar o Jenkins informando o local onde se encontram as ferramentas que ele utilizará para construir o projeto, o que é mostrado na Figura 18. Além destas, a localização da instalação do Maven também deve ser efetuada, em interface similar à apresentada.

Um dos diferenciais do Jenkins é que ele viabiliza o monitoramento de diversos projetos. Para tanto, para cada um devemos criar um job, que é responsável por conhecer a localização do projeto e seu

Gerenciar o Jenkins

O Jenkins sem segurança permite que qualquer um na rede inicie processos.
⚠ Considerar habilitar a autenticação para evitar acessos indesejados.

- Configurar o sistema**
Configurar opções globais e caminhos
- Configurar segurança global**
Faça a segurança no Jenkins; defina quem pode usar/acessar o sistema.
- Recarregar configuração do disco**
Descartar todos os dados carregados na memória e recarregar tudo do sistema de arquivos. Isso é útil quando seus arquivos de configuração foram modificados diretamente no disco.
- Gerenciar plugins**
Adiciona, remove, desabilita e habilita plugins que podem incrementar as funcionalidades do Jenkins. (**Atualizações disponíveis**)

Figura 17. Painel administrativo do Jenkins

A interface de usuário do Jenkins para configuração de JDK e Git. No topo, uma barra com o nome do projeto "Jenkins". Abaixo, seções para "JDK Instalações" e "Git Instalações".

JDK
Nome: Java
JAVA_HOME: /usr/lib/jvm/java-7-openjdk-i386
 Instalar automaticamente
Adicionar JDK
Lista de JDK instalações nesse sistema

Git
Name: Git
Path to Git executable: /usr/bin/git
 Instalar automaticamente
Add Git
description

Figura 18. Configuração do Jenkins

A interface de usuário do Jenkins com o menu principal. À esquerda, uma barra lateral com ícones para "Novo job", "Usuários", "Histórico de compilações", "Gerenciar Jenkins" e "Credentials". O centro exibe uma saudação "Bem-vindo ao Jenkins!" e uma instrução para clicar em "Novo job" para iniciar. Abaixo, seções para "Fila de builds" (nenhum build na fila) e "Estado do executor de builds" (1 Parado, 2 Parado).

Figura 19. Menu principal do Jenkins

código-fonte, além de armazenar outras informações. A partir disso será possível executar o build e os testes automaticamente.

A criação de um *job* no Jenkins é feita por meio de seu menu lateral, com a opção *Novo job* (vide Figura 19). Ao acessá-la, dentre as alternativas disponíveis, escolha *Construir projeto maven*.

Definido o tipo de projeto (Maven), o processo de criação é continuado por meio de várias telas, sendo a primeira delas apresentada na Figura 20. Neste momento deve-se informar um nome para o projeto, sua descrição e, se for utilizado o código-fonte junto a um servidor remoto, sua identificação e URL deste repositório remoto (com credenciais de acesso).

Na próxima janela são exibidas diversas opções de configuração adicionais, sendo importante observar a opção relativa à localização do arquivo *pom.xml*. O Jenkins utilizará esse arquivo para resolver as dependências do projeto. Caso isto não seja feito, assume-se a localização padrão e caso não seja possível localizá-lo, o usuário será prontamente avisado por mensagens de alerta.

Finalizada a criação do *job*, o usuário é encaminhado à página inicial do projeto, onde é possível acompanhar todas as informações a ele relacionadas, conforme ilustra a Figura 21. O menu lateral permite, dentre outras opções, construir o projeto a qualquer momento (por meio do menu *Construir agora*) e, ao centro da



tela, observar um registro de todas as atividades (por meio do menu *Mudanças recentes*). Observa-se ainda, abaixo do menu lateral à esquerda, um campo destinado ao histórico de builds efetuadas. Como o projeto ainda não foi construído, não há nada a ser mostrado neste local.

A cada construção do projeto é possível acompanhar todo o andamento do processo, como explicita a Figura 22. O identificador “#1”, neste caso, indica que se trata da primeira construção. À medida que novos builds são executados, este contador é incrementado.

Essa construção pode ser disparada manualmente, pelo menu *Construir agora*, ou então de maneira automática, configurando nas opções de um job a construção em determinados intervalos de tempo. Ademais, é possível especificar o envio automático de e-mails aos membros do projeto, com o resultado do build.

Para visualizar o resultado de cada build, basta clicar em seu identificador, ou então, pelo arquivo de log. Na construção de um build, é verificada primeiramente a existência de erros no código (como erros de sintaxe) e em seguida são executados os testes unitários, validando se a aplicação produz os resultados que são esperados. A interface com o resultado da primeira construção do projeto pode ser observada na Figura 23.



Figura 20. Interface para criação de um novo job no Jenkins

Figura 21. Página inicial do job criado

Figura 22. Menu de opções de um job e build em andamento

Figura 23. Resultados do primeiro build

Resultado do Teste



Figura 24. Relatório visual

```
TESTS
-----
Running br.com.company.app.AppTest
Called a AppTest
calculeIMCError
Called a AppTest
calculeIMCSuccess
Called a AppTest
main
0 resultado do IMC: 22.1606648199446
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.088 sec
Results :
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

Figura 25. Log de construção (parcial)



Figura 26. Página inicial do Jenkins contendo resumo de um job

Ainda no resultado da construção do projeto *LittleSimpleJavaProject*, note a ausência de mensagens relativas a erros de código, pois nenhum alerta do compilador foi encontrado (nenhuma falha). Visualiza-se também que nenhum dos testes falhou, sendo possível observar o log completo clicando sobre o campo *Resultado de testes*, o qual apresenta um relatório gráfico conforme a **Figura 24**.

Além desse relatório, pode-se acompanhar todas as mensagens geradas pelo compilador em console durante a construção do projeto e também o registro da execução dos testes (vide **Figura 25**).

Concluído o primeiro build, a página inicial do Jenkins é atualizada e pode-se acompanhar visualmente se o projeto está sendo construído com sucesso (observando um ícone ilustrativo similar a um tempo ensolarado) ou falha, conforme a **Figura 26**. Deste modo, é possível controlar o status de vários *jobs* ao mesmo tempo.

Com o conteúdo apresentado, confirma-se a importância da adoção de ferramentas como NetBeans, Git, Maven e Jenkins como facilitadoras na incorporação do DevOps na construção de software. Estas soluções, quando combinadas, promovem a automação e colaboração em todas as etapas do processo de desenvolvimento, possibilitando uma melhora direta na qualidade do projeto.

Destaca-se, entretanto, que a utilização destas sem o comprometimento de todos pode não gerar melhorias. Portanto, também é de grande importância a criação de um ambiente de trabalho favorável e que demonstre que o uso de tais recursos pode simplificar a implementação e evitar retrabalhos.

Links:

Site do Apache Maven.
<http://maven.apache.org/>

Site do GitHub.
<https://github.com/>

Página do projeto Jenkins.
<http://jenkins-ci.org/>

Página da ferramenta JUnit.
<http://junit.org/>

Página da IDE NetBeans.
<https://netbeans.org/>

REFFATTI, Luan Malaguti. Produtividade no desenvolvimento de aplicações por meio da cultura DevOps. Trabalho de Diplomação (Tecnologia em Análise e Desenvolvimento de Sistemas). Universidade Tecnológica Federal do Paraná. Medianeira 2014.

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Autor



Juliano Rodrigo Lamb

lamb@utfpr.edu.br

é professor da Universidade Tecnológica Federal do Paraná (UTFPR), campus Medianeira, onde atua nas disciplinas relativas a Engenharia de Software e Desenvolvimento de Sistemas, observando boas práticas de programação.



Autor



Everton Coimbra de Araújo

everton@utfpr.edu.br

Desde 1987 atua na área de treinamento e desenvolvimento. É mestre em Ciência da Computação, doutor em Engenharia Agrícola e professor efetivo da UTFPR, campus Medianeira.



Autor



Evandro Carlos Pessini

pessini@utfpr.edu.br

É professor da Universidade Tecnológica Federal do Paraná – campus Medianeira – atuando em disciplinas na área de desenvolvimento de sistemas (principalmente em disciplinas relacionadas à linguagem de programação Java) e na área da teoria da computação.



Possui mestrado em Ciência da Computação pela UFSC e doutorado em Ciência da Computação pela UFRN. As áreas de estudo e pesquisa de interesse incluem a verificação formal de sistemas e a Web Semântica.

Assinaturas digitais em Java: conheça e implemente os formatos no padrão nacional

Saiba como distinguir e implementar os formatos para geração de assinaturas digitais conforme os padrões legais definidos pela ICP-Brasil

O uso de certificados digitais é algo que tem se tornado bastante comum em aplicações web, sendo impulsionado principalmente por questões de segurança como, por exemplo, aquelas que visam garantir a autenticidade da identidade de um sistema web (ou site internet). Além deste caso, também é cada vez mais importante garantir a autoria e integridade das informações, isto é, saber quem a gerou e se o conteúdo não foi corrompido ou fraudado. A partir da informatização do sistema jurídico nacional, garantir a validade de documentos e processos é algo fundamental à legitimidade dos processos, por exemplo.

Neste cenário, uma das técnicas de certificação digital mais adotadas por sistemas informatizados são as assinaturas digitais. Prover a um sistema a capacidade de assinar qualquer tipo de conteúdo digital, além de assegurar a integridade da informação, viabiliza a garantia da validade jurídica no âmbito nacional e até internacional, uma vez que no Brasil, e em outros países, já existem dispositivos legais em forma de leis e normas que embasam essa garantia.

Com o crescimento do comércio eletrônico e o avanço dos sistemas computacionais, a legislação tributária brasileira criou um projeto chamado Sistema Pùblico de Escrituração Digital – SPED. Este padroniza uma série de documentos e procedimentos contábeis que eram

Fique por dentro

Este artigo é útil por apresentar os formatos de assinaturas digitais possíveis de serem implementados com a tecnologia Java e permitir que o leitor conheça as características técnicas e padrões necessários para a implementação. Serão abordados também os requisitos técnicos formais definidos pela ICP-Brasil, que é a entidade gestora da infraestrutura de certificação digital brasileira, para que as assinaturas geradas tenham validade legal em âmbito nacional. Além disso, analisaremos alguns exemplos práticos utilizando as principais bibliotecas em Java que auxiliam neste tipo de desenvolvimento.

comumente produzidos em papel para serem produzidos na forma de documentos digitais. Dentre os documentos e procedimentos padronizados, o mais conhecido é a Nota Fiscal Eletrônica (ou NF-e), e uma das obrigatoriedades deste projeto é a geração de assinatura digital dentro dos padrões definidos pela legislação brasileira – mais um motivo que sinaliza a relevância do assunto tratado neste artigo.

Assim como o SPED, há outras iniciativas para uso de assinaturas digitais tanto nos setores públicos como privados. Um bom exemplo vem do poder Judiciário.

Ademais, fora as questões tecnológicas inerentes à implementação da certificação digital, há também aspectos legais e normativos, pois elas devem garantir os mesmos direitos de uma



assinatura analógica, que é aquela que fazemos com uma caneta no papel.

Para tratar os aspectos normativos, foi criada uma instituição chamada ICP-Brasil, que define as regras sob as quais essas assinaturas devem ser geradas para serem válidas. O papel dela é definir, através de normativas técnicas, formatos, padrões e propriedades como, por exemplo, qual algoritmo deve ser usado para gerar as chaves criptográficas e qual o tempo de validade de um certificado.

Com base em tudo o que foi informado até aqui, neste artigo vamos abordar aspectos jurídicos relacionados a assinaturas digitais, tecnologias Java que podem ser empregadas para a geração destas assinaturas e, principalmente, alguns exemplos.

ICP-Brasil

Para que um certificado digital, que é um componente básico para a geração de assinaturas digitais, tenha garantia de autenticidade e até mesmo respaldo legal, ele deve ser reconhecido por uma infraestrutura de chaves públicas (ICP).

Uma ICP pode ser definida como uma instituição formal composta por procedimentos técnicos e organizacionais para geração de chaves criptográficas que envolve desde soluções em software e hardware até o tipo de hierarquia que a infraestrutura adota. É essa organização que dá sustentação às técnicas de certificação digital e também aos procedimentos burocráticos para sua implantação e uso. Em nosso país, há somente uma reconhecida oficialmente, e o nome dela é ICP-Brasil.

O objetivo principal de uma ICP é garantir a legitimidade e autenticidade das informações de um certificado. Para isso, adota e/ou especifica normas e padrões a serem seguidos pelos sistemas que venham a utilizar a certificação digital.

A organização/estrutura da ICP-Brasil é composta pelos seguintes itens:



- **Comitê Gestor:** É um grupo de pessoas que coordena o funcionamento organizacional e burocrático da ICP-Brasil, estabelecendo as políticas, os critérios e as normas para o funcionamento de toda a infraestrutura;

- **Comissão Técnica:** Chamada de Comissão Técnica Executiva (COTEC), tem a função de prestar suporte e consultoria ao Comitê Gestor em assuntos tecnológicos, sempre que o comitê solicitar. Em outras palavras, sempre que houver alguma decisão por parte do comitê que seja necessário um parecer técnico, a comissão é convocada;

- **Infraestrutura da ICP:** A estrutura da ICP-Brasil é dividida em três partes principais, conforme detalhado a seguir:

- **Autoridade Certificadora Raiz:** É o primeiro nível do que é chamado de cadeia de certificação, que por sua vez representa uma hierarquia na qual uma infraestrutura de chaves públicas se organiza. É papel da autoridade Raiz executar e fazer cumprir as Políticas de Certificados e normas técnicas e operacionais aprovadas pelo comitê Gestor. Gerar, distribuir, cancelar e gerenciar os certificados das autoridades certificadoras do nível abaixo do seu são funções da AC-Raiz. Além disso, é responsável por publicar os certificados que foram revogados (a LCR) e fiscalizar as Autoridades Certificadoras (ACs), Autoridades de Registro (ARs) e demais entidades habilitadas na ICP-Brasil. Além disso, verifica se as ACs estão atuando em conformidade com as diretrizes e normas técnicas estabelecidas pelo Comitê Gestor da ICP-Brasil;

- **Autoridade Certificadora (AC):** É a entidade responsável por garantir a autenticidade dos certificados digitais. Para isso, cada AC, tanto a raiz como as de nível inferior, possuem um par de chaves (uma pública e outra privada) que é utilizado para assinar os certificados emitidos por ela. É através da validação desta assinatura que é verificada a autenticidade de um certificado. Como qualquer hierarquia de ICP, o certificado da Autoridade Raiz é auto assinado e os das demais abaixo da Raiz, assinados pela primeira. Assim, é comum a existência de vários níveis intermediários de autoridades certificadoras, formando uma pirâmide, conforme ilustra a **Figura 1**. São algumas das principais responsabilidades da AC: publicar e fazer cumprir todas as normas, políticas e padrões para o funcionamento de uma ICP, que no âmbito da ICP-Brasil são definidos pelo comitê gestor;

- **Autoridade de Registro (AR):** Além de ser o último nível na hierarquia de uma ICP, a AR representa a autoridade responsável pela obtenção, análise e aprovação dos dados dos usuários para gerar os certificados. Também é responsabilidade desta entidade a guarda dos documentos necessários para a verificação da legitimidade e identidade do usuário. Deve, obrigatoriamente, estar credenciada junto a uma AC, pois é uma AR que representará fisicamente uma AC perante uma pessoa que deseja obter um certificado. Isto é, uma AR é um local físico para interface de uma AC, onde alguém se apresenta para pedir a emissão de um certificado. A AR deve oferecer segurança, pessoal especializado e os equipamentos necessários.

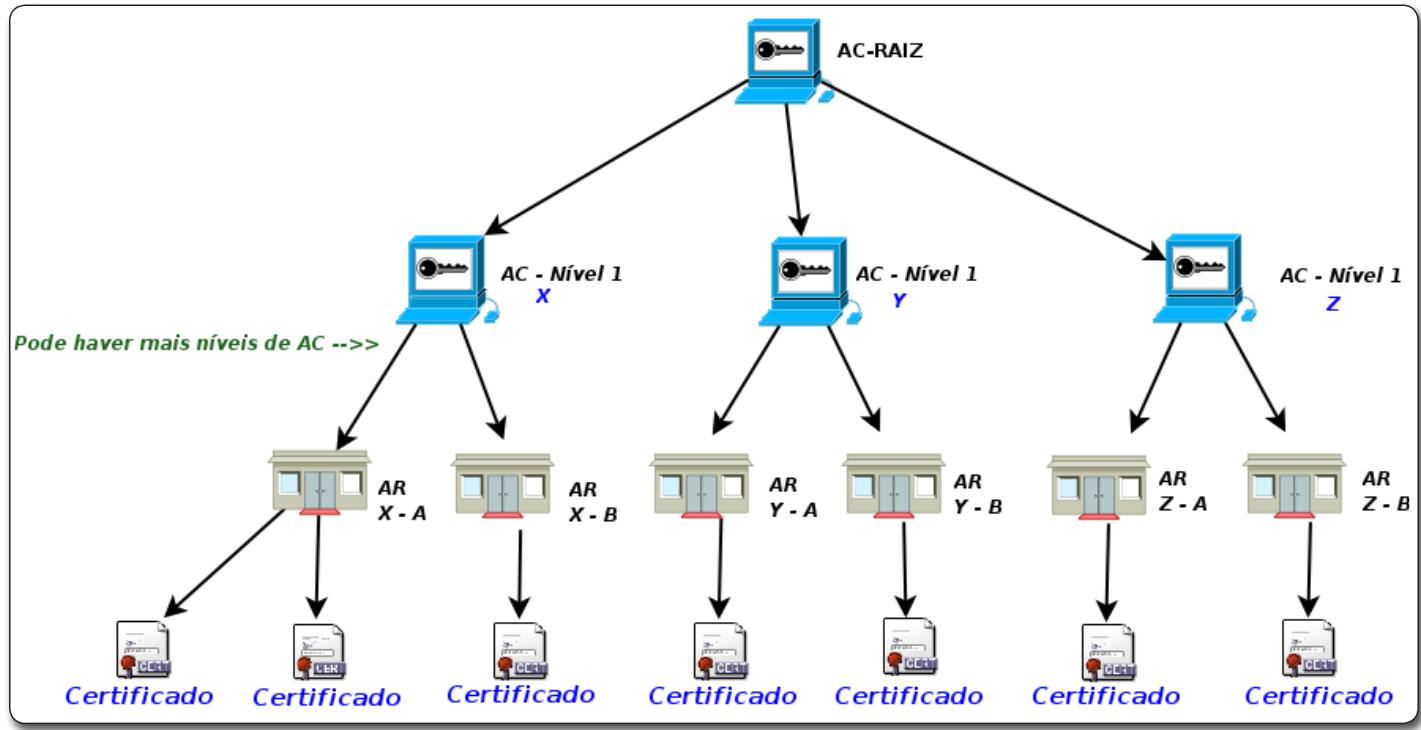


Figura 1. Hierarquia de uma infraestrutura de chaves públicas

Para efeitos legais, no âmbito nacional, a ICP-Brasil foi criada pela Medida Provisória 2.200-2, de 24.10.2001, sendo formada por um conjunto de Autoridades Certificadoras organizadas em conformidade com as diretrizes e normas técnicas estabelecidas por seu Comitê Gestor.

Conforme ilustra a Figura 1, uma das principais características da ICP-Brasil é sua estrutura hierárquica. No topo dessa estrutura encontra-se a Autoridade Certificadora Raiz e, abaixo dela, estão diversas entidades, em diversos níveis. A entidade responsável pelo gerenciamento e manutenção desta hierarquia é o Instituto Nacional de Tecnologia da Informação – ITI.

Assinaturas digitais e aspectos legais

Analisaremos agora o funcionamento dos procedimentos técnicos necessários para se trabalhar com assinaturas digitais e abordaremos, de forma resumida, como estes processos foram referenciados na legislação brasileira para que passassem a ter validade jurídica.

Como funciona o processo de geração e uso de assinaturas digitais

Quando estamos trabalhando com assinaturas digitais, temos que usar alguns conhecimentos e técnicas de criptologia, pois a assinatura digital é baseada num tipo de criptografia chamada de assimétrica. Nesse tipo de criptografia, cada pessoa (ou equipamento) possui um par de chaves, sendo uma privada (que deve ser mantida em segredo) e uma pública (que deve ser divulgada por meio do certificado digital). Este par de chaves possui a seguinte propriedade: tudo o que for cifrado com a chave privada só pode ser decifrado com a chave pública. Desta forma, se você cifrar

um documento inteiro com sua chave privada, qualquer pessoa que conheça a sua chave pública conseguirá decifrá-lo e assim saber que foi você quem cifrou, pois as técnicas de criptografia garantem que cada par de chaves criptográficas é único.

Este procedimento já é considerado uma assinatura digital, mas há dois problemas que devem ser considerados com esta abordagem (criptografar o documento inteiro):

- 1) Lentidão do processo, uma vez que a criptografia assimétrica exige bastante processamento para garantir a segurança – dependendo do tamanho do documento a ser assinado, pode ser bastante demorado criptografar todo o conteúdo;
- 2) O arquivo com a assinatura digital teria praticamente o mesmo tamanho do arquivo original e ainda tornaria o conteúdo original de acesso restrito ao processo de descriptografia, pois o mesmo estará cifrado dentro deste arquivo.

Para solucionar estes problemas, foi definido um passo intermediário: a obtenção de um hash (um resumo criptográfico) do documento, que é calculado muito rapidamente. Como o hash possui tamanho fixo e pequeno, apenas ele será cifrado com a chave privada, originando assim a assinatura digital do documento.

A Figura 2 ilustra este processo: o assinante (aquele que está realizando a assinatura) submete o documento a uma função de cálculo de hash para obter o resumo criptográfico e, em seguida, usa sua chave privada para cifrar o resumo, obtendo então um arquivo com a assinatura digital.

A Figura 3, por sua vez, ilustra o processo de verificação de uma assinatura digital. A pessoa (ou sistema) que deseja ve-

rificar a assinatura deve utilizar a chave pública do assinante (par correspondente à chave privada e que geralmente vem anexada à assinatura) para decifrar o arquivo de assinatura e obter o resumo criptográfico anteriormente cifrado pelo assinante. Depois, submete o documento à função de resumo criptográfico para comparar o hash gerado neste momento com o hash obtido no passo anterior. Se estes

valores forem iguais, a assinatura está válida, caso contrário, a mesma não pode ser considerada válida. Note que qualquer alteração no documento fará com que o resumo criptográfico correspondente seja alterado, invalidando a assinatura. É este mecanismo que permite confirmar que o documento não foi alterado por terceiros.

Em complemento aos processos de criação e validação de uma assinatura, existem

padrões tecnológicos que definem tipos de estruturas de dados, que são chamados de formatos, para gerar um arquivo de assinatura digital (hash criptografado), padrões estes que serão abordados posteriormente neste artigo.

Nota

Um hash é uma sequência de bits calculados por algoritmos de dispersão com tamanho pré-definido. Por exemplo, o algoritmo SHA-1 gera um hash com 160 bits a partir de cálculos realizados em cada byte de um documento digital. Como o hash geralmente tem um tamanho menor que o documento, é possível que exista mais de um documento com o mesmo valor de hash. Entretanto, a forma como os algoritmos são criados faz com que, computacionalmente, seja muito raro encontrar dois documentos com o mesmo hash.

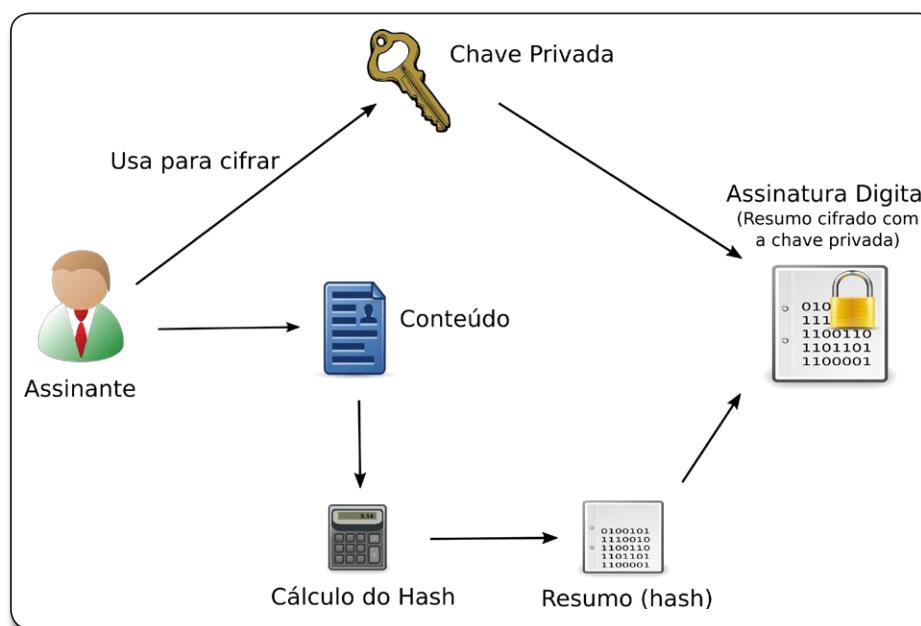


Figura 2. Processo de geração de uma assinatura digital

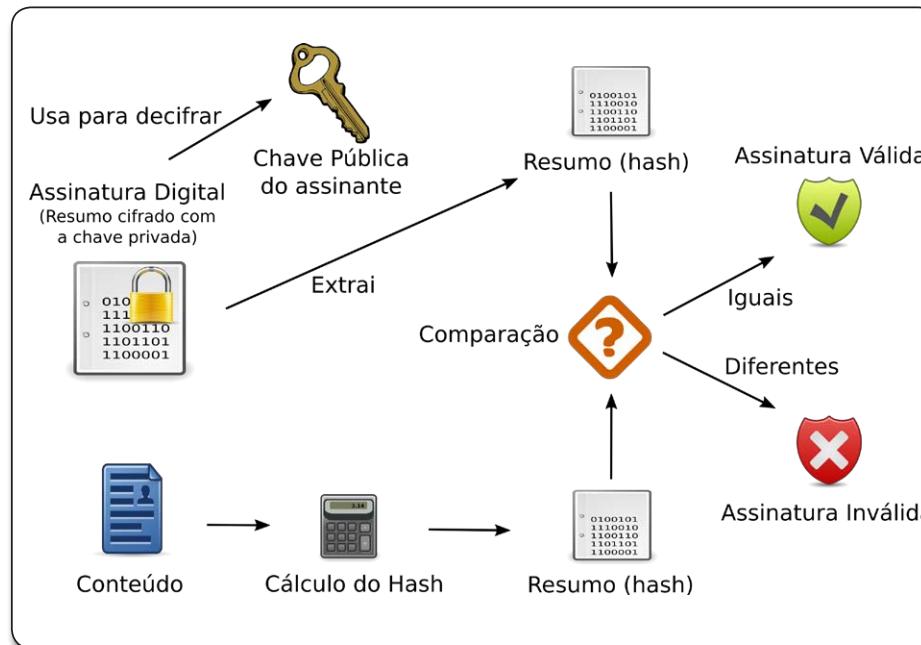


Figura 3. Processo de validação de uma assinatura digital

Legislação atual

Como sabemos, é preciso que a assinatura digital tenha um respaldo jurídico para que tenha validade legal. No nosso caso, além da legislação que define a própria ICP-Brasil, existem outros instrumentos jurídicos na legislação brasileira que embasam e apoiam a assinatura digital.

Um desses é a Lei N° 11.419, de 19 de dezembro de 2006, que trata da informatização do processo judicial. Desta lei, que trata de outros itens além da certificação digital, destacamos o seguinte trecho, que faz referência especificamente às assinaturas digitais:

"Art. 11. Os documentos produzidos eletronicamente e juntados aos processos eletrônicos



com garantia da origem e de seu signatário, na forma estabelecida nesta Lei, serão considerados originais para todos os efeitos legais.

§ 3º Os originais dos documentos digitalizados, mencionados no § 2º deste artigo, deverão ser preservados pelo seu detentor até o trânsito em julgado da sentença ou, quando admitida, até o final do prazo para interposição de ação rescisória.”.

Destacamos também o seguinte trecho da Medida Provisória nº 2.200-2 de 24/08/2001, que define a ICP-Brasil:

“Art. 10. Consideram-se documentos públicos ou particulares, para todos os fins legais, os documentos eletrônicos de que trata esta Medida Provisória.

§ 1º As declarações constantes dos documentos em forma eletrônica produzidos com a utilização de processo de certificação disponibilizado pela ICP-Brasil presumem-se verdadeiros em relação aos signatários, na forma do art. 131 da Lei no 3.071, de 10 de janeiro de 1916 - Código Civil.”.

Estes instrumentos legais proveem às assinaturas duas características formais:

1. Eficácia Probatória: A assinatura digital passa a ter validade jurídica para ser apresentada como prova que identifique a adulteração em um documento eletrônico falso;

2. Não Repúdio: Ou não recusa, é a garantia que o emissor de uma mensagem ou a pessoa que executou determinada transação de forma eletrônica (assinou um documento ou formulário eletrônico, por exemplo) não poderá, posteriormente, negar sua autoria.

Formatos de assinaturas digitais

Os formatos dos arquivos de assinaturas digitais são definidos por padrões internacionais e estão relacionados à tecnologia usada para gerar cada tipo de arquivo. Pelo aspecto legal, tais formatos devem ser reconhecidos juridicamente e para isso são necessárias normas para viabilizar a criação e reconhecimento dos documentos assinados digitalmente.

A partir destas especificações internacionais foram criadas, pela ICP-Brasil, as normas que estabelecem quais são os formatos aceitos e quais atributos cada um deles deve possuir ao ser gerada uma assinatura digital no Brasil.

Neste artigo, abordaremos os dois formatos internacionais que já foram normatizados pela ICP-Brasil (CAdES e XAdES), e um formato que ainda está em discussão para normatização (PAdES).

Formato CAdES

O formato CAdES (*CMS Advanced Electronic Signatures*) estabelece informações específicas a serem colocadas em uma assinatura digital que siga o formato original CMS (*Cryptographic Message Syntax*), o qual define a estrutura básica de uma mensagem assinada digitalmente que seja codificada com o formato binário ASN.1 (*Abstract Syntax Notation One*).

No formato CMS, a mensagem assinada digitalmente é composta pela mensagem propriamente dita (que pode estar no corpo da assinatura ou em um arquivo separado), a assinatura (resultado da cifragem do hash com a chave privada do assinan-

te) e a identificação do assinante. Opcionalmente, a mensagem assinada também pode ter atributos com informações extras.

Visando facilitar o registro de informações referentes à assinatura digital, o CAdES padroniza formatos a serem utilizados para estes registros, que são colocados em atributos de uma assinatura. Alguns exemplos de informações que podem ser registradas são: data e hora em que ocorreu a assinatura, local onde ocorreu a assinatura e referências que permitam validar a assinatura mesmo após a expiração do certificado do assinante.

Um atributo muito utilizado no padrão CAdES é o que informa a política de assinatura que foi levada em consideração no momento em que o documento foi assinado. A política de assinatura é um documento que define as regras a serem aplicadas para considerar uma assinatura válida, pois situações diferentes podem ter requisitos diferentes para o processo de validação de uma assinatura digital como, por exemplo, ter um carimbo de tempo (*Timestamp*) associado à assinatura.

Formato XAdES

O formato XAdES (*XML Advanced Electronic Signatures*) tem o mesmo objetivo do formato CAdES, porém, em vez de ser baseado no padrão CMS, este formato é baseado no padrão XMLDSIG (*XML-Signature Syntax and Processing*), que permite gerar assinaturas digitais codificando os dados usando a linguagem XML (*Extensible Markup Language*). Embora a assinatura seja codificada em XML, o documento original não precisa ser um documento XML. Assim, a assinatura pode ser gerada a partir de qualquer tipo de arquivo.

De forma análoga ao que ocorre no formato CAdES, o formato XAdES possui mecanismos para incorporar informações extras em assinaturas digitais. Estas informações recebem o nome de propriedades, em vez de atributos, e são codificadas como tags XML, em vez de estruturas ASN.1, que é a base do formato CAdES.

Formato PAdES

O formato PAdES (*PDF Advanced Electronic Signatures*) ainda está sendo normatizado na ICP-Brasil e em breve deverá ser publicado como um formato oficialmente aceito. Ele visa permitir que assinaturas digitais realizadas em documentos PDF (*Portable Document Format*) também possam receber informações extras que as tornem mais duradouras, possibilitando sua verificação mesmo que o certificado do assinante tenha o seu prazo de validade expirado. Outra informação extra bastante utilizada, por exigência das normas da ICP-Brasil, é a referência à política de assinatura que foi utilizada.

Para gerar a assinatura com as informações necessárias, o formato PAdES reaproveita tanto o formato CAdES quanto o formato XAdES. Deste modo, para o caso mais comum, onde o conteúdo do documento PDF por completo (desconsiderando informações extras em XML) será assinado, uma assinatura CAdES deve ser criada e embutida no documento em um local preestabelecido pelo padrão. Caso alguma informação em XML esteja presente no documento

PDF (o formato permite a inclusão de metadados em XML), deve ser utilizado o formato XAdES para criar a assinatura digital.

Geração e validação com Java

Para implementação de funcionalidades como a geração e validação de assinaturas utilizando a linguagem Java, existe a *Java Cryptography Architecture* (JCA), framework que faz parte da Java SE Security e define a arquitetura básica para garantir a segurança da informação, como autenticidade e privacidade, especificamente com o uso da tecnologia de certificação digital, e a *Java Cryptography Extension* (JCE), que é uma extensão da plataforma Java para tratar questões de criptografia e adota a mesma arquitetura definida pela JCA.

Ambas são tecnologias definidas pelo processo da comunidade Java (JCP – Java Community Process), da mesma forma que o JPA, JSF, entre outras e, portanto, precisam de um provedor (*provider*) que as implemente. Para isso, no próprio JDK existe o SunJCE, que é uma implementação para o JCE. Podemos citar também o Bouncy Castle, que é outro provedor para JCA/JCE bastante conhecido e com código aberto. Com o uso de um destes provedores é possível fazer a geração e validação de assinaturas digitais no formato CAdES. Para o formato XAdES (XML), a partir do Java SE 6 foi introduzida a Java XML Digital Signature API (JSR105), que fornece as funcionalidades para geração e validação relacionadas a esta opção.

Deste modo, a base para implementação das assinaturas nos principais formatos usando a linguagem Java está disponível no

JDK e também através de outros provedores, o que facilita bastante o trabalho do desenvolvedor. Entretanto, no caso do Brasil, existem algumas particularidades na geração das assinaturas relacionadas a alguns atributos específicos, como informações civis e legais (ex: CPF, CNPJ, PIS, Título de Eleitor, etc.), além de questões relacionadas à própria ICP-Brasil, como as cadeias de certificados e as listas de revogação. Essas diferenças em relação aos padrões internacionais não são abordadas nos principais provedores. A saída nestes casos é implementar essas diferenças no programa ou sistema que for desenvolvido ou utilizar alguma biblioteca ou framework nacional que as provenha.

Por se tratar de uma questão local, obviamente não encontraremos nenhuma biblioteca estrangeira que forneça esse tipo de solução. Sendo assim, obrigatoriamente é preciso implementar algo a mais além do que oferecem os provedores internacionais. Outra opção, como já citado, é buscar por soluções nacionais que implementem as características locais. Felizmente, hoje já existem algumas soluções pagas e outras em software livre, que atendem ao mercado nacional. Das soluções livres, a mais atualizada com relação às normativas é o componente de certificado digital do Framework Demoiselle, chamado Demoiselle-Certificate.

Exemplo com o provider padrão do Java

A **Listagem 1** mostra um exemplo de geração de assinatura usando o provider criado pela SUN, conforme a documentação no site da Oracle. Com este código é possível gerar um par de chaves (pública e privada) para criptografia (linhas 16 a 26), realizar a

Listagem 1. Geração de assinatura digital com JCA/JCE/SUN.

```
01 import java.io.*;
02 import java.security.*;
03
04 class GenSig {
05
06     public static void main(String[] args) {
07
08         /* Gerando a assinatura digital */
09
10        //Valida se foi passado um arquivo (a ser assinado) como argumento
11        if (args.length != 1) {
12            System.out.println("Digite: GenSig NomeArquivoAssinar");
13        }
14        else try {
15
16            //Cria o gerador de par de chaves
17            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
18
19            //Inicializa o gerador de chaves
20            SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
21            keyGen.initialize(1024, random);
22
23            //Cria o par de chaves
24            KeyPair pair = keyGen.generateKeyPair();
25            PrivateKey priv = pair.getPrivate();
26            PublicKey pub = pair.getPublic();
27
28            //Obtém o objeto Assinatura (dsa) conforme o algoritmo e o provedor
29            Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
30
31            //Inicializa o objeto Assinatura (dsa)
32            dsa.initSign(priv);
33
34            //Passando os dados para serem assinados
35            FileInputStream fis = new FileInputStream(args[0]);
36            BufferedInputStream bufin = new BufferedInputStream(fis);
37            byte[] buffer = new byte[1024];
38            int len;
39            while ((len = bufin.read(buffer)) >= 0) {
40                dsa.update(buffer, 0, len);
41            };
42            bufin.close();
43
44            //Gera a assinatura propriamente dita
45            byte[] realSig = dsa.sign();
46
47            /* Salva a assinatura em um arquivo */
48            FileOutputStream sigfos = new FileOutputStream("Assinatura");
49            sigfos.write(realSig);
50            sigfos.close();
51
52            /* Grava a chave pública em um arquivo */
53            byte[] key = pub.getEncoded();
54            FileOutputStream keyfos = new FileOutputStream("ChavePublica");
55            keyfos.write(key);
56            keyfos.close();
57
58        } catch (Exception e) {
59            System.err.println("Erro: " + e.toString());
60        }
61    }
}
```

Listagem 2. Verificação de assinatura digital com JCA/JCE/SUN.

```
01 import java.io.*;
02 import java.security.*;
03 import java.security.spec.*;
04
05 class VerSig {
06     public static void main(String[] args) {
07         /* Verificando uma assinatura digital */
08         if (args.length != 3) {
09             System.out
10             .println("Digite: VerSig ChavePublica Assinatura NomeArquivoAssinado");
11         } else
12             try {
13                 /* Importar a chave pública */
14                 FileInputStream keyfis = new FileInputStream(args[0]);
15                 byte[] encKey = new byte[keyfis.available()];
16                 keyfis.read(encKey);
17                 keyfis.close();
18                 X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);
19                 KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
20                 PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
21
22                 /* Ler a assinatura */
23                 FileInputStream sigfis = new FileInputStream(args[1]);
24                 byte[] sigToVerify = new byte[sigfis.available()];
25                 sigfis.read(sigToVerify);
26                 sigfis.close();
27
28             /*
29             * Criar o objeto Signature do JCE e inicializar a verificação com a chave pública
30             */
31             Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
32             sig.initVerify(pubKey);
33
34             /* Atualiza e verifica os dados assinados */
35             FileInputStream datafis = new FileInputStream(args[2]);
36             BufferedInputStream bufin = new BufferedInputStream(datafis);
37
38             byte[] buffer = new byte[1024];
39             int len;
40             while (bufin.available() != 0) {
41                 len = bufin.read(buffer);
42                 sig.update(buffer, 0, len);
43             };
44             bufin.close();
45             boolean verifies = sig.verify(sigToVerify);
46
47             System.out.println("verificação da assinatura: " + verifies);
48
49         } catch (Exception e) {
50             System.err.println("Exceção " + e.toString());
51         };
52     }
53 }
```

assinatura (linhas 28 a 45) e salvá-la em um arquivo (linhas 48 a 50). Além disso, entre as linhas 53 e 56 é exemplificado como salvar em um arquivo a chave pública que foi gerada, uma vez que esta será utilizada para validar a assinatura (**Listagem 2**).

A execução desse código pode ser feita, por exemplo, da seguinte forma: *java GenSig Arquivo.txt*. Porém, esse processo não estaria completo do ponto de vista legal no Brasil, pois não temos a garantia de procedência da chave criptográfica, que só pode ser feita através da validação das chamadas cadeias de certificados, conforme exemplificado na **Figura 1**, que de acordo com a legislação brasileira, devem fazer parte da ICP-Brasil. Ademais, também não estarão incluídos os atributos obrigatórios definidos pela própria ICP-Brasil, chamados de políticas de assinaturas. Por isso que é essencial adquirir um certificado emitido por uma autoridade certificadora dentro da ICP-Brasil e gerar a assinatura conforme uma das políticas estabelecidas por ela.

Já na **Listagem 2** mostramos o código exemplo para validação da assinatura gerada na **Listagem 1**. Considerando que o código da primeira listagem foi executado com sucesso, o comando para verificar a assinatura é: *java VerSig ChavePublica Assinatura Arquivo.txt*.

O processo consiste em:

- 1) importar a chave pública (linhas 14 a 20), que deve ser passada como primeiro parâmetro;
- 2) fazer a leitura do arquivo com a assinatura propriamente dita (linhas 23 a 26), passada como segundo parâmetro;

- 3) criar um objeto do tipo **Signature** (fornecido pelo SunJCE), que será inicializado com a chave pública (linhas 31 e 32);
- 4) ler os dados do arquivo assinado e recebido como terceiro parâmetro (*Arquivo.txt*) (linhas 35 e 36), que por sua vez serão incluídos no objeto **Signature** (linhas 38 a 44);
- 5) por fim, validar a assinatura através do método **verify()**, na linha 45.



Exemplo com componente que atende os requisitos da ICP-Brasil

Nos exemplos a seguir utilizaremos o componente chamado Demoiselle Certificate, fornecido pela comunidade que mantém o projeto do Framework Demoiselle e que já implementa os requisitos exigidos pela ICP-Brasil.

O primeiro exemplo, apresentado na **Listagem 3**, serve para a geração da assinatura digital. Neste código, mostramos como a classe **FileSystemKeyStoreLoader**, fornecida pelo componente do Demoiselle, permite realizar a leitura de um arquivo do tipo KeyStore (vide linhas 16 e 17) – também conhecido como chaveiro. Este representa uma estrutura que armazena o par de chaves criptográficas juntamente com uma cadeia de certificados, fazendo parte de uma família de padrões denominada PKCS (*Public-Key Cryptography Standards*). Para atender à legislação brasileira, a cadeia de certificados deve estar no padrão ICP-Brasil. Após a leitura do chaveiro, as informações obtidas serão utilizadas para criar um objeto do tipo **KeyStore**, que é viabilizado pela JCE em conformidade com o formato PKCS. A partir desse objeto, recuperamos os dados da chave privada (linha 21).

Em seguida, é criado um objeto do tipo **Certificate** (linha 23), também fornecido pela JCE, que armazenará a cadeia dos certificados das autoridades que garante que o certificado contido no arquivo lido foi emitido pela ICP-Brasil. Depois chamamos o método **PKCS7Factory.getInstance().factoryDefault()**, para criar um objeto do tipo **PKCS7Signer** (linha 27). Este, por sua vez, receberá os dados da cadeia de certificados armazenada

no objeto **Certificate** (linha 29) e da chave privada (linha 31), que estão no objeto do tipo **KeyStore**. Logo após, também será atribuído ao objeto **PKCS7Signer** o tipo do algoritmo (linha 37) que será usado para gerar a assinatura; neste exemplo, o **SHA256withRSA**, que é o padrão recomendado pela ICP-Brasil. Ainda a este mesmo objeto, vamos atribuir o valor para informar qual é a política de assinatura (linha 35) – dentre as definidas pela ICP-Brasil – que será usada.

Quando utilizamos o componente do Demoiselle, temos a nosso dispor a classe **PolicyFactory**, que já armazena todas as políticas definidas pela ICP-Brasil. Deste modo, resta-nos apenas escolher uma delas conforme os requisitos relacionados à assinatura a ser gerada como, por exemplo, se terão carimbo de tempo ou não. Neste exemplo, escolhemos a primeira e mais simples opção definida pela ICP-Brasil, que é identificada pela sigla **AD_RB_CADES_2_1** e que se refere à política de assinatura digital com referência básica no formato CAdES versão 2.1. As demais opções disponíveis são: Assinatura digital com Referências para Arquivamento (AD-RA), Assinatura digital com Referências Completas (AD-RC), Assinatura digital com Referência de Tempo (AD-RT) e Assinatura digital com Referências para Validação (AD-RV). Além disso, também atribuímos o valor verdadeiro (**true**) no método **setAttached()** do objeto **PKCS7Signer** (linha 33), o que indicará que a assinatura será do tipo anexada, fazendo com que o arquivo gerado armazene a assinatura e também o conteúdo que foi assinado.

Listagem 3. Geração de assinatura digital com Demoiselle Certificate.

```
01 import java.io.*;
02 import java.security.*;
03 import java.security.cert.Certificate;
04 import br.gov.frameworkdemoiselle.certificate.keystore.loader.factory
    .KeyStoreLoaderFactory;
05 import br.gov.frameworkdemoiselle.certificate.keystore.loader.implementation
    .FileSystemKeyStoreLoader;
06 import br.gov.frameworkdemoiselle.certificate.signer.SignerAlgorithmEnum;
07 import br.gov.frameworkdemoiselle.certificate.signer.factory.PKCS7Factory;
08 import br.gov.frameworkdemoiselle.certificate.signer.pkcs7.PKCS7Signer;
09 import br.gov.frameworkdemoiselle.policy.engine.PolicyFactory;
10
11 public class GenSigDemoiselle {
12
13     public static void main() throws UnrecoverableKeyException,
        KeyStoreException, NoSuchAlgorithmException, IOException
14    {
15        // Acesso ao certificado armazenado em arquivo (.p12)
16        FileSystemKeyStoreLoader loader = (FileSystemKeyStoreLoader)
            KeyStoreLoaderFactory
17            .factoryKeyStoreLoader(new File("certificado_icp_brasil.p12"));
18        KeyStore keystore = loader.getKeyStore("senha");
19        String alias = "alias";
20        // Acesso a chave privada
21        PrivateKey pk = (PrivateKey) keystore.getKey(alias, "senha".toCharArray());
22        // Leitura da cadeia de certificados
23        Certificate[] chain = keystore.getCertificateChain(alias);
24
25        // Gerando informação para ser assinada, mas poderia ser lida de um
        // arquivo também
26        byte[] content = "hello world".getBytes();
27        // Cria objeto de assinatura
28        PKCS7Signer signer = PKCS7Factory.getInstance().factoryDefault();
29        // Carrega a cadeia de certificados
30        signer.setCertificates(chain);
31        // Carrega a chave privada
32        signer.setPrivateKey(pk);
33        // Define assinatura com anexação do conteúdo assinado
34        signer.setAttached(true);
35        // Define a política da assinatura
36        signer.setSignaturePolicy(PolicyFactory.Policies.AD_RB_CADES_2_1);
37        // Define o algoritmo para geração da assinatura
38        signer.setAlgorithm(SignerAlgorithmEnum.SHA256withRSA);
39        // Assina o conteúdo e verifica
40        byte[] signed = signer.doSign(content);
41        signer.check(content, signed);
42        // Grava o arquivo com assinatura e conteúdo anexado
43        BufferedOutputStream bos = null;
44        FileOutputStream fos = new FileOutputStream(new File("/tmp/result.p7s"));
45        bos = new BufferedOutputStream(fos);
46        bos.write(signed);
47        bos.close();
48    }
}
```

Listagem 4. HTML com applet para geração de assinatura digital com Demoiselle Certificate.

```
01 <ui:define name="body">
02   <h:form id="mainForm" name="mainForm" method="post" action=""
03     prependId="false" enctype="multipart/form-data">
04       <table width="100%">
05         <tr>
06           <td><h1>Applet Exemplo de Assinatura Digital</h1></td>
07         </tr>
08         <tr>
09           <td><applet id="appletAssinador"
10             codebase="http://127.0.0.1:8080/estacionamento/"
11             code="br.gov.frameworkdemoiselle.certificate.applet.view.JPanelApplet"
12             mayscript="mayscript" style="width: 500px; height: 400px;"
13             archive="demoiselle-certificate-timestamp-2.0.0-SNAPSHOT-assinado.jar,
14               demoiselle-certificate-signer-2.0.0-SNAPSHOT-assinado.jar,
15               demoiselle-certificate-policy-engine-2.0.0-SNAPSHOT-assinado.jar,
16               demoiselle-certificate-cryptography-2.0.0-SNAPSHOT-assinado.jar,
17               demoiselle-certificate-core-2.0.0-SNAPSHOT-assinado.jar,
18               demoiselle-certificate-ca-icpbrazil-homologacao-2.0.0-
19                 SNAPSHOT-assinado.jar,
20                 demoiselle-certificate-ca-icpbrazil-2.0.0-SNAPSHOT-assinado.jar,
21                 demoiselle-certificate-applet-2.0.0-SNAPSHOT-assinado.jar,
22                 applet-customizada-1.0.0-assinado.jar,
23                 slf4j-api-1.6.1-assinado.jar,
24                 slf4j-log4j12-1.6.1-assinado.jar,
25                 log4j-1.2.17-assinado.jar,
26                 bcprov-jdk15on-1.51-assinado.jar,
27                 bcpkix-jdk15on-1.51-assinado.jar,
28                 bcmail-jdk15on-1.51-assinado.jar,
29                 htmlunit-2.15-assinado.jar"
30               alt="">
31             <param name="factory.applet.action"
32               value="org.demoiselle.applet_customizada.App" />
33             <param name="applet.javascript.postaction.failure" value="foo" />
34             <param name="permissions" value="all-permissions" />
35             No Applet
36           </applet></td>
37         </tr>
38       </table>
39       <tr> <th colspan="2">Escolha o arquivo a ser assinado</th></tr>
40       <tr> <td>Documento:</td>
41         <td><input type="file" name="documento" value="" /></td>
42       </tr>
43       <tr> <th colspan="2">Dados extraídos do certificado digital</th> </tr>
44       <tr>
45         <td>CPF:</td>
46         <td><input type="text" name="cpf" value="" size="11"
47           disabled="disabled" /></td>
48       </tr>
49       <tr>
50         <td>Nome:</td>
51         <td><input type="text" name="nome" value="" size="30"
52           disabled="disabled" /></td>
53       </tr>
54       <tr>
55         <td>Email:</td>
56         <td><input type="text" name="email" value="" size="30"
57           disabled="disabled" /></td>
58       </tr>
59       <tr>
60         <td>Nascimento:</td>
61         <td><input type="text" name="nascimento" value="" size="8"
62           disabled="disabled" /></td>
63       </tr>
64     </table>
65   </h:form>
66 </ui:define>
```

Nesta estratégia (anexada) não é necessário o arquivo original para validar a assinatura, mas o inconveniente é que o arquivo gerado será tão grande quanto o original a ser assinado. Esse tipo de abordagem é mais comum quando estamos assinando informações que são geradas dinamicamente (por exemplo, mensagens entre sistemas ou textos de e-mail). Para as informações que estão armazenadas em arquivos digitais (documentos de texto, planilhas, imagens), o mais comum é definir como desanexada e gerar um arquivo separado somente com a assinatura.

Neste exemplo é possível notar que boa parte da complexidade de entender como funcionam as técnicas de implementação de assinaturas digitais em Java, bem como as regras da ICP-Brasil, já foram abstraiadas pelo componente.

Há ainda outros requisitos referentes à geração de assinaturas digitais, não abordadas na listagem, que também já possuem implementações prontas no componente do Demoiselle. Por exemplo, a leitura de chaves e certificados armazenados em tokens criptográficos, algo bastante comum para uso comercial, como na geração de assinaturas para nota fiscal eletrônica ou na assinatura de processos jurídicos.

Exemplo de implementação para aplicações web

Quando a aplicação é instalada no equipamento do usuário, como as desenvolvidas com Swing, a implementação da assinatu-

ra digital é feita basicamente como demonstrado na **Listagem 3**. Já nos casos em que a aplicação será executada através de um navegador, a implementação das funcionalidades de geração e validação de assinaturas digitais deve ser feita a partir de uma página HTML. Neste cenário, por restrições de segurança do próprio navegador, é preciso o uso de Applets. Diante disso, tanto as bibliotecas (JAR) dos componentes que forem utilizados para suportar a geração e validação de assinaturas digitais, quanto o próprio applet, devem ser assinados, para que a JVM permita que elas sejam executadas pelo navegador, sendo que essa assinatura precisa ser feita por um tipo de certificado específico para esse fim.

A ICP-Brasil também normatiza esse tipo de certificado, que é chamado de certificado de assinatura de código. A diferença para os outros tipos de certificados normatizados pela ICP é que este possui o atributo code signing.

Na área do GitHub do Framework Demoiselle pode ser encontrado um projeto exemplo que armazena as bibliotecas fornecidas pelo componente Demoiselle Certificate já assinadas com certificados ICP-Brasil e prontas para serem executadas em uma aplicação web. São essas bibliotecas que foram usadas no exemplo da **Listagem 4** (vide linhas 12 a 27), onde demonstramos como criar uma página HTML para executar um applet para geração de assinaturas digitais no padrão nacional.

Listagem 5. Código do applet.

```
01 public class App extends AbstractAppletExecute {  
02  
03     @Override  
04     public void execute(KeyStore keystore, String alias, int policyselected,  
05                         Applet applet) {  
06         try {  
07             /* Carregando o conteúdo a ser assinado */  
08             String documento = AbstractAppletExecute.getFormField(applet,  
09                           "mainForm", "documento");  
10            if (documento.length() == 0) {  
11                JOptionPane.showMessageDialog(applet, "Por favor, escolha um documento  
12                    para assinar",  
13                    "Error", JOptionPane.ERROR_MESSAGE);  
14                return;  
15            }  
16            File file = new File(documento);  
17            String doc = documento;  
18            byte[] content = readContent(file.getCanonicalPath());  
19            /* Parametrizando o objeto signer */  
20            PKCS7Signer signer = PKCS7Factory.getInstance().factoryDefault();  
21            signer.setCertificates(keystore.getCertificateChain(alias));  
22            signer.setPrivateKey((PrivateKey) keystore.getKey(alias, null));  
23            switch (policyselected) {  
24                case 0:  
25                    signer.setSignaturePolicy(PolicyFactory.Policies.AD_RB_CADES_1_0);  
26                    break;  
27                /* Outras opções de políticas disponíveis no PolicyFactory */  
28                case 1:{  
29                    ...  
30                }  
31                signer.setAttached(false);  
32                /* Realiza a assinatura do conteúdo */  
33                byte[] signed = signer.doSign(content);  
34  
35                /* Grava o conteúdo assinado no disco */  
36                writeContent(signed, documento.concat(".p7s"));  
37                /* Valida o conteúdo */  
38                boolean checked = signer.check(content, signed);  
39                if (checked) {  
40                    JOptionPane.showMessageDialog(applet, "O arquivo foi assinado e validado  
41                        com sucesso.",  
42                        "Mensagem", JOptionPane.INFORMATION_MESSAGE);  
43                } else {  
44                    System.out.println("A assinatura não é válida!");  
45                }  
46                /* Exibe alguns dados do certificado */  
47                ICPBrasilCertificate certificado = super.getICPBrasilCertificate  
48                (keystore, alias, false);  
49                AbstractAppletExecute.setFormField(applet, "mainForm", "cpf",  
50                certificado.getCpf());  
51                AbstractAppletExecute.setFormField(applet, "mainForm", "nome",  
52                certificado.getNome());  
53                AbstractAppletExecute.setFormField(applet, "mainForm", "nascimento",  
54                certificado.getDataNascimento());  
55            } catch (KeyStoreException | NoSuchAlgorithmException |  
56            UnrecoverableKeyException | IOException e) {  
57                JOptionPane.showMessageDialog(applet, e.getMessage(), "Error",  
58                JOptionPane.ERROR_MESSAGE);  
59            }  
60            /* método para ler conteúdo de arquivos */  
61            private byte[] readContent(String arquivo) {  
62                byte[] result = null;  
63                try {  
64                    File file = new File(arquivo);  
65                    FileInputStream is = new FileInputStream(file);  
66                    result = new byte[(int) file.length()];  
67                    is.read(result);  
68                    is.close();  
69                } catch (IOException ex) {ex.printStackTrace();}  
70                return result;  
71            }  
72            /* método para gravar arquivos */  
73            private void writeContent(byte[] conteudo, String arquivo) {  
74                try {  
75                    File file = new File(arquivo);  
76                    FileOutputStream os = new FileOutputStream(file);  
77                    os.write(conteudo);  
78                    os.flush();  
79                    os.close();  
80                } catch (IOException ex) {ex.printStackTrace();}  
81            }  
82        }  
83    }  
84}
```



Note que todas as bibliotecas necessárias, tanto do Demoiselle (linhas 13 a 19) quanto de outras dependências (linhas 21 a 27), estão nomeadas com o sufixo *assinado*.

Observe ainda, na linha 20 da **Listagem 4**, a declaração de uma biblioteca Java chamada *applet-customizada*. Como pode ser verificado na **Listagem 5**, o código desse applet foi construído para gerar a assinatura, mas poderia ser apenas para ler as informações de um certificado para executar uma autenticação (login) em um sistema. A customização do applet deve atender às necessidades de uso de cada aplicação.

Atualmente, o componente Demoiselle Certificate, já de acordo com as especificações da ICP-Brasil, oferece as seguintes funcionalidades: ler os certificados, gerar e validar assinaturas digitais, criptografar e descriptografar informações ou arquivos.

Podemos verificar que esta classe deve estender **AbstractAppletExecute**, classe abstrata fornecida pelo componente Demoiselle-Certificate que disponibiliza os métodos para interação entre a aplicação e navegador como, por exemplo, ler um token criptográfico.

Com o intuito apenas de demonstração, optamos também por exibir alguns atributos que serão lidos do certificado do assinante (linhas 48 a 50), alguns deles específicos do padrão da ICP-Brasil, como o CPF do portador (linha 48). Comparando o que foi implementado na **Listagem 5** com o que está na **Listagem 3**, apenas os métodos para leitura (linhas 56 a 66) e gravação (linhas 68 a 76) de arquivos são diferentes. O restante é exatamente como já foi descrito.

Vale ressaltar que o uso do componente também ajuda a padronizar a implementação. Por exemplo: se nosso objetivo fosse desenvolver um serviço para criptografia, faríamos a customização de um applet praticamente da mesma forma que expomos na **Listagem 5**. Ou seja, estenderíamos a classe **AbstractAppletExecute** e implementaríamos os mesmos métodos para leitura e gravação de arquivos, assim como foi feito para o carregamento do par de chaves e da cadeia de certificados. No entanto, para criptografar um arquivo, em vez de criar um objeto do tipo **PKCS7Signer** (como na linha 18), seria criado um objeto do tipo **Cryptography**, também fornecido pelo Demoiselle Certificate e que já possui os métodos para cifrar e decifrar.

Para saber mais sobre as funcionalidades oferecidas pelo Demoiselle Certificate, consulte a documentação no site do componente, cujo endereço encontra-se na seção **Links**.

A **Figura 4** mostra o applet em execução. Neste exemplo, como estamos utilizando um certificado do tipo A3 para pessoa física, um dos campos que serão exibidos é o CPF, pois a classe **ICPBrasilCertificate** (vide linha 47) já possui métodos para isso, inclusive para identificar qual é o tipo do certificado. Por sua vez, caso estivéssemos utilizando um certificado para pessoa jurídica, optaríamos por exibir o CNPJ.

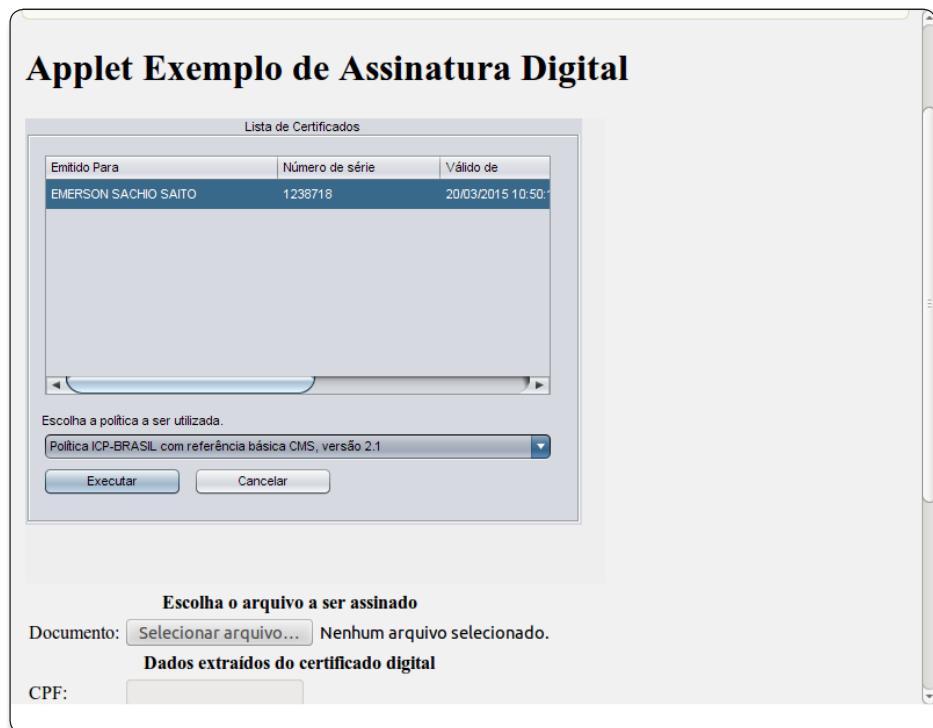


Figura 4. Applet utilizando o componente Demoiselle-Certificate

Figura 5. Validador on-line mantido pelo ITI

Conforme os documentos da ICP-Brasil, atualmente são 10 os tipos de certificados digitais para usuários finais, separados por finalidade de uso. Dentre as opções, seis são para gerar assinaturas digitais e quatro para gerar sigilo (criptografar informações). Essa divisão é uma norma da ICP-Brasil, mas do ponto de vista técnico não há diferença entre o certificado de assinatura e o certificado de sigilo, pois ambos podem fazer criptografia. O que muda é apenas



Assinador Digital de Documentos

Utilize esta ferramenta para assinar documentos com certificado digital ou validar documentos já assinados.

Para acessar o tutorial do assinador digital clique [aqui](#).

Orientações de instalação da cadeia de certificado em máquinas Windows



Figura 6. Assinador on-line mantido pelo SERPRO

uma propriedade do certificado que identifica o tipo e seu nível, conforme veremos a seguir. Ademais, para ter validade jurídica, é preciso usar o tipo correspondente para cada finalidade, conforme definido pela norma. Assim, se utilizar um certificado que foi emitido para ser utilizado como de sigilo para gerar uma assinatura digital, essa não terá validade perante a justiça brasileira.

Por isso, temos a seguinte divisão por tipo:

- Certificados para Assinatura Digital:

1. A1;
2. A2;
3. A3;
4. A4;
5. T3;
6. T4.



- Certificados para Sigilo:

1. S1;
2. S2;
3. S3;
4. S4.

E para cada conjunto de tipos (Assinatura ou Sigilo), há também uma divisão com os seguintes níveis:

- **Nível 1 (A1 e S1):** A geração das chaves é feita por software; terão o tamanho mínimo de 1024 bits; o armazenamento é feito em formato de arquivo digital; e a validade máxima de um ano;

- **Nível 2 (A2 e S2):** A geração das chaves é feita por software; terão o tamanho mínimo de 1024 bits; o armazenamento é feito em cartão inteligente (com chip) ou token (dispositivo semelhante a um pen drive); e a validade máxima de dois anos;

- **Nível 3 (A3, S3, T3):** A geração das chaves é feita por hardware; terão o tamanho mínimo de 1024 bits; o armazenamento é feito em cartão inteligente ou token; e a validade máxima de cinco anos;

- **Nível 4 (A4, S4 e T4):** A geração das chaves é feita por hardware; terão o tamanho mínimo de 2048 bits; o armazenamento é feito em cartão inteligente ou token; e a validade máxima de cinco anos.

Certificados dos tipos T3 e T4 podem ser emitidos apenas para equipamentos das Autoridades de Carimbo do Tempo (ACT) credenciadas na ICP-Brasil.

Além das questões técnicas, há uma classificação referente ao tipo do portador do certificado, pois no Brasil temos a distinção formal entre pessoa física e pessoa jurídica.

Diante disso, convencionou-se tratar os certificados usando essa mesma formalidade, e assim, comercialmente adota-se o termo e-CPF para certificados emitidos para pessoas físicas e e-CNPJ para aqueles emitidos para pessoas jurídicas.

Serviços on-line para geração e validação de assinaturas no padrão ICP-Brasil

Com o intuito de validar as assinaturas e comprovar se estão mesmo no padrão exigido, o ITI disponibiliza um serviço on-line (veja o endereço na seção [Links](#)). Esta é a melhor opção para validar se a implementação feita em um sistema atende os requisitos definidos pela ICP-Brasil.

A Figura 5 mostra o resultado da validação de uma assinatura gerada com o código apresentado nas [Listagens 4 e 5](#).

Como um facilitador ao usuário, existem alguns serviços on-line que fornecem assinadores digitais. Estes podem ser utilizados por qualquer pessoa que possua um certificado no padrão da ICP-Brasil devidamente configurado no navegador. Daqueles que encontramos nos mecanismos de buscas, indicamos o que é mantido pelo SERPRO (veja o endereço na seção [Links](#)), e que

utiliza o componente Demoiselle-Certificate. Na **Figura 6** expomos a sua interface.

Apesar de parecer simples a implementação de assinaturas digitais em Java, os requisitos referentes às questões legais e normativas trazem um pouco de complexidade para a efetividade desta.

Contudo, as iniciativas, tanto em software livre quanto proprietário, podem minimizar bastante esta complexidade.

Ainda assim, tão importante quanto saber como funciona o processo do ponto de vista técnico, é compreender os procedimentos jurídicos e as normas que garantem a veracidade e confiabilidade do processo como um todo. Para isso, existe bastante material teórico disponível na internet, principalmente no site da ICP-Brasil, que complementará o conhecimento que foi passado neste artigo.

Por fim, recomendamos a leitura do conteúdo que está disponível no site do Framework Pinhão (veja a seção **Links**), que apesar de não ser muito recente, também fornece informações relacionadas a aspectos técnicos e legais sobre a implementação de assinaturas digitais.

Autor



Fabiano Castro Pereira

fabiano.pereira@serpro.gov.br

Concluiu Bacharelado e Mestrado em Ciências da Computação pela Universidade Federal de Santa Catarina (UFSC). Foi analista de sistemas no Laboratório de Segurança em Computação (LabSEC/UFSC), onde trabalhou com projetos de pesquisa na área de Segurança em Computação. Atualmente trabalha como analista na Coordenação Estratégica de Tecnologia do Serpro, em Florianópolis, onde atua em projetos voltados para inovação tecnológica, padrões de interoperabilidade, também contribuindo para o Framework Demoiselle no desenvolvimento de componentes relacionados à certificação digital.



Links:

Site da ICP-Brasil.

<http://www.iti.gov.br/icp-brasil>

Demoiselle Certificate.

<http://demoiselle.sourceforge.net/docs/components/certificate/reference/>

Endereço para download da aplicação exemplo.

<https://github.com/demoiselle/example>

Material teórico do projeto Tabelião-Pinhão.

<http://www.frameworkpinhao.pr.gov.br/modules/conteudo/conteudo.php?conteudo=16>

Java Cryptography Architecture (JCA)

<https://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html>

Verificador de assinaturas do ITI.

<https://verificador.iti.gov.br/verificador.xhtml>

Assinador do SERPRO.

<https://www.serpro.gov.br/assinador-digital/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Autor



Emerson Sachio Saito

emerson.saito@serpro.gov.br - <http://blogoosfera.cc/esaito/>
Bacharel em Análise de Sistemas pela PUC-PR, com especialização em Tecnologia da Informação pela UFPR. Trabalha há mais de 12 anos com a tecnologia Java. Trabalhou como analista de informática na CELEPAR (Companhia de informática do Paraná) de 2001 a 2009, onde entre outras diversas atividades, foi um dos desenvolvedores da plataforma de desenvolvimento Java/WEB chamada Pinhão e responsável pelo projeto chamado Tabelião, que implementa a parte de certificação digital desta plataforma. Atualmente é analista de desenvolvimento no SERPRO (Serviço Federal de Processamento de Dados), onde faz parte da equipe dedicada ao desenvolvimento do Framework Demoiselle e também está atuando no componente de certificação digital.



Open Data: desenvolvendo uma aplicação para monitoramento de ônibus

Veja nesse artigo como utilizar dados e padrões abertos para criar soluções de serviço público para monitoramento de ônibus com Open Data

Diversas empresas, sejam elas públicas ou privadas, vêm disponibilizando dados com licença livre. Essa nova opção, permite que os mesmos sejam utilizados da maneira que as pessoas desejarem. Segundo o portal de dados abertos do governo brasileiro, os principais motivos para a liberação das informações são transparência na gestão, contribuição da sociedade com serviços inovadores ao cidadão, aprimoramento na qualidade dos dados governamentais, viabilização de novos negócios e obrigatoriedade por lei.

Apesar de já ser uma ideia antiga, o conceito de dados abertos vem ganhando bastante destaque nos últimos anos, principalmente pela cobrança de uma maior transparência na administração pública. No Brasil, já existem diversas iniciativas no sentido de liberar o acesso a dados, antes restritos à população. Alguns exemplos de dados abertos são os dados de ônibus de algumas cidades do Brasil, como São Paulo e Rio de Janeiro, e alguns portais, como os do governo do estado e da cidade de São Paulo.

Um exemplo de fonte de dados abertos é a API Olho Vivo da SPTrans, órgão que gerencia o transporte público sobre rodas na cidade de São Paulo. Essa API permite que desenvolvedores acessem diversos dados sobre o sistema de ônibus, obtendo informações sobre as linhas,

Fique por dentro

Este artigo demonstra como desenvolver aplicações utilizando dados abertos, o que é viável quando empresas disponibilizam dados com licença para livre acesso da população. A partir disso, é possível utilizá-los para várias finalidades como, por exemplo, para que os cidadãos possam monitorar o que políticos e órgãos públicos estão fazendo e, também, para a criação de novas soluções ou serviços que melhoram a vida das pessoas. Para exemplificar isso, este artigo apresentará como desenvolver um sistema web que permite monitorar a posição dos ônibus da cidade de São Paulo em tempo real, com base em dados que são disponibilizados pela prefeitura.

as paradas e, principalmente, a posição de todos os ônibus em tempo real.

No desenvolvimento de sistemas com dados abertos, uma das opções é utilizar padrões e APIs também abertas como, por exemplo, o JSON, que é um padrão para troca de mensagens entre diferentes aplicações, e também a API do Google Maps ou do OpenStreetMap, que permitem que sejam implementadas soluções utilizando os mapas dessas plataformas.

Para mostrar como utilizar dados abertos, a partir de agora iremos implementar uma aplicação que monitora a posição dos ônibus de São Paulo em tempo real. Essa solução será dividida

em duas partes: a primeira será responsável por recuperar as posições dos ônibus em tempo real e as armazenar no Elasticsearch (utilizaremos uma ferramenta de Big Data pelo grande volume de dados que será salvo e pela necessidade de busca rápida a partir do nome de uma linha de ônibus); e a segunda será uma página web que mostrará a posição dos ônibus no Google Maps, sendo atualizada de tempos em tempos para acompanhar o trajeto dos mesmos. Ademais, como a API da SPTrans apenas informa a posição atual do veículo, isto é, não possibilita a recuperação de dados históricos, será necessário salvar esses dados todas as vezes que os solicitarmos.

A API da SPTrans

Assim como outras APIs públicas, a API Olho Vivo fornece diversos dados sobre os ônibus da cidade de São Paulo. Algumas das informações que podem ser recuperadas são os dados das linhas, a posição geográfica dos ônibus de uma linha, a posição geográfica (latitude e longitude) de um ponto de parada e a previsão de chegada do ônibus nesses pontos. Para que um desenvolvedor possa utilizar a API, no entanto, é necessário um cadastro no site da SPTrans. A URL para acesso a esse site está na seção [Links](#).

Realizado o cadastro, cada desenvolvedor recebe um token único, que deve ser passado para todas as requisições que forem feitas na API. No desenvolvimento da aplicação, mostraremos como esse token deve ser informado à API. Além disso, existe uma boa documentação da API, que descreve todos os serviços que ela disponibiliza e todos os dados que devem ser especificados como parâmetro, assim como o formato de todos os dados que são retornados. A URL para essa documentação também está na seção [Links](#).

Na API, todas as requisições são feitas através de uma URL simples como, por exemplo, <http://api.olhovivo.sptrans.com.br/v0/Linha/Buscar?termoBusca=715M-10>, onde *Buscar* é o nome do serviço da API que estamos utilizando, *termoBusca* é o nome do parâmetro que estamos passando para a API e 715M-10 é o valor passado no parâmetro, que nesse caso, é o letreiro de uma linha de ônibus de São Paulo. A **Listagem 1** mostra o retorno da requisição à URL exemplo.

Na aplicação que será desenvolvida nesse artigo, utilizaremos dois serviços da API, o *Buscar*, já mostrado, e também o serviço *Posicao*. O serviço *Buscar* retorna os dados gerais sobre a linha como, por exemplo, o código, o letreiro, o sentido do ônibus (no caso de São Paulo, Bairro/Centro ou Centro/Bairro), e o nome da linha, que pode ser diferente se o ônibus está no sentido Bairro/Centro ou Centro/Bairro. Já o serviço *Posicao* retorna, em tempo real, a posição dos ônibus de uma determinada linha. As informações recuperadas são o código do veículo (cada veículo tem um identificador único), a latitude e a longitude. Porém, é sempre necessário chamar o serviço *Buscar* antes, pois na chamada do serviço *Posicao*, é preciso informar o código da linha, que é um identificador da SPTrans recuperado pelo serviço *Buscar*.

A documentação da API não indica o intervalo em que realiza a atualização dos dados com a posição dos ônibus. Por nossa experiência, o tempo médio para isso é de 20 segundos.

Listagem 1. JSON de resposta da API Olho Vivo.

```
{  
    "CodigoLinha":33258,  
    "Circular":false,  
    "Letreiro":"715M",  
    "Sentido":2,  
    "Tipo":10,  
    "DenominacaoTPTS":"LGO. DA POLVORA",  
    "DenominacaoTSTP":"JD. MARIA LUIZA",  
    "Informacoes":null  
}
```

Instalação e configuração do ambiente de desenvolvimento

Para a implementação do nosso exemplo vamos utilizar o Eclipse e o Maven. Além disso, para a aplicação que mostrará a posição dos ônibus em tempo real, é necessário ter acesso – local ou remoto – a um servidor web. Neste caso, optamos pelo Tomcat, versão 8. Note que também poderíamos utilizar servidores de aplicação, como GlassFish, WildFly, entre outros.

O primeiro passo é baixar e instalar essas ferramentas (veja a seção [Links](#)). Para tanto, basta descompactar os arquivos e colocá-los em algum diretório. Depois é necessário adicionar o Tomcat ao Eclipse, o que pode ser feito pela aba *Servers* desta IDE, ao clicar em *Add Server* e selecionar o caminho da instalação do Tomcat. Assim, será possível executar os exemplos que serão apresentados diretamente a partir deste ambiente de desenvolvimento.

Para a persistência dos dados será utilizado o Elasticsearch. Até poderíamos utilizar um banco de dados relacional para isso, mas pela quantidade de dados que serão salvos e pela quantidade de buscas que serão realizadas, avaliamos como mais interessante adotar uma ferramenta Big Data. Para configurar o Elasticsearch é necessário criar um índice nessa ferramenta para descrever como os dados serão salvos. A **Listagem 2** mostra o código responsável por criar o índice chamado **onibus**. Uma boa maneira de executar esse código é utilizar o plugin Sense para o Chrome, que permite o envio de dados no padrão JSON para o Elasticsearch. O endereço para download dessa ferramenta está na seção [Links](#). Para mais informações sobre o Elasticsearch, leia o artigo “Elasticsearch: realizando buscas no Big Data”, publicado na Java Magazine 137.

Esse índice contém todos os dados que serão armazenados, a saber: o id do ônibus (código do ônibus recuperado através do serviço *Buscar* da API Olho Vivo), o letreiro do ônibus, o campo *nome*, onde será salvo o nome da linha do ônibus, a data e hora em que o registro foi salvo e a posição do ônibus. É válido ressaltar ainda que nesse índice serão armazenadas todas as leituras de posição. Por isso, é importante definir o id como sendo o código do ônibus, para que no momento de realizar a busca no Elasticsearch, ele retorne a última posição lida de cada ônibus. Deste modo, bastará ao Elasticsearch obter a última leitura para cada id salvo. Ademais, a partir dessa decisão, também será possível recuperar dados de leituras anteriores, caso seja solicitado.

Listagem 2. Índice do Elasticsearch para salvar os dados dos ônibus.

```
POST /sptrans/onibus/_mapping
{
  "onibus": {
    "_id": {
      "path": "busCode"
    },
    "properties": {
      "name": {
        "type": "string"
      },
      "fields": {
        "raw": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    },
    "busCode": {
      "type": "string"
    },
    "lineCode": {
      "type": "string"
    },
    "letters": {
      "type": "string"
      "index": "not_analyzed"
    },
    "dateBus": {
      "type": "date"
    },
    "location": {
      "type": "geo_point"
    }
  }
}
```

Aplicação que recupera os dados dos ônibus

Depois de tudo configurado, podemos começar a implementação das aplicações. Primeiro, vamos desenvolver a solução que se conecta na API Olho Vivo e monitora, de 30 em 30 segundos, a posição dos ônibus de São Paulo. Para simplificar, nesse exemplo vamos monitorar apenas a linha 715M-10, mas a aplicação pode ser facilmente estendida para acompanhar todas as linhas da cidade.

Dito isso, podemos começar criando um projeto Maven no Eclipse. As dependências necessárias para esse projeto são o *json-simple*, que é o framework utilizado para interpretar arquivos JSON, e o driver para conexão com o Elasticsearch. A **Listagem 3** mostra o *pom.xml* do projeto.

Para manipular os dados dos ônibus na aplicação, será necessário implementar uma classe Java simples, que representará os dados recuperados através da API. Para isso, foi criada a classe **Bus**, mostrada na **Listagem 4**. Os atributos dessa classe são o nome da linha, o código do ônibus (código único que o identifica), as letras da linha e a latitude e longitude da posição onde o ônibus se encontra.

Com a aplicação configurada e a classe que representa os ônibus criada, podemos implementar a conexão com a API para recuperar os dados. Antes disso, no entanto, é preciso realizar

a autenticação, o que é feito ao recuperar um cookie com o token que foi gerado pela API no momento do cadastro no site da SPTrans.

Listagem 3. Arquivo pom.xml para configuração do projeto.

```
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.santana.sc</groupId>
  <artifactId>sptrans</artifactId>
  <version>0.0.1</version>
  <dependencies>
    <dependency>
      <groupId>com.googlecode.json-simple</groupId>
      <artifactId>json-simple</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>org.elasticsearch</groupId>
      <artifactId>elasticsearch</artifactId>
      <version>1.4.0</version>
    </dependency>
  </dependencies>
</project>
```

Listagem 4. Código da classe Bus.

```
package com.santana.sc.sptrans;

public class Bus {

  private int code;
  private String name;
  private String letters;
  private double lat;
  private double lon;

  // getters e setters omitidos...
}
```

A **Listagem 5** mostra o código do método **getCookie()**. Nele, inicialmente é estabelecida uma conexão HTTP simples utilizando a classe **HttpURLConnection**. Em seguida, através do método **setRequestMethod()**, especifica-se que a requisição será do tipo GET. Note que para essa requisição funcionar, deve ser informado como parâmetro o **token** recebido no momento do cadastro feito no site da API. Depois, com a classe **DataOutputStream**, é recuperada a resposta da requisição, que contém o cookie que devemos passar em todas as requisições futuras que forem feitas à API. Por último, o cookie é salvo na **String cookie** e retornada no final do método.

Com o cookie em mãos, podemos executar as requisições à API da SPTrans. Nesse exemplo, como já mencionado, faremos chamadas a dois serviços da API. O primeiro deles, *Buscar*, procura, a partir do letreiro de uma linha de ônibus, as informações sobre ela, como seu nome nos dois sentidos (ida e volta) e seu código. Com o código da linha, é possível fazer a chamada ao serviço *Posicao*, que é o mais importante neste nosso exemplo. Este recupera a posição de todos os ônibus de determinada linha em tempo real.

Listagem 5. Recuperação do cookie de autenticação da API Olho Vivo.

```
public String getCookie() throws Exception {  
  
    String cookie = "";  
  
    String url = "http://api.olhovivo.sptrans.com.br/v0/Login/Autenticar?token=substituir_pelo_token";  
    URL c = new URL(url);  
    HttpURLConnection connection = (HttpURLConnection) c.openConnection();  
  
    connection = (HttpURLConnection) c.openConnection();  
    connection.setRequestMethod("GET");  
  
    // Send request  
    DataOutputStream wr = new DataOutputStream(connection.getOutputStream());  
    wr.flush();  
    wr.close();  
  
    Map<String, List<String>> headerFields = connection.getHeaderFields();  
  
    Set<String> headerFieldsSet = headerFields.keySet();  
    Iterator<String> headerFieldsIter = headerFieldsSet.iterator();  
  
    while (headerFieldsIter.hasNext()) {  
        String headerFieldKey = headerFieldsIter.next();  
  
        if ("Set-Cookie".equalsIgnoreCase(headerFieldKey)) {  
            List<String> headerFieldValue = headerFields.get(headerFieldKey);  
            for (String headerValue : headerFieldValue) {  
                String[] fields = headerValue.split(";");  
                cookie = fields[0];  
            }  
        }  
    }  
  
    return cookie;  
}
```

Listagem 6. Código para recuperar a posição dos ônibus em tempo real.

```
public List<Bus> getBusesByLine(String cookie, String busLine) throws Exception {  
  
    List<Bus> buses = new ArrayList<Bus>();  
  
    StringBuffer buffer = new StringBuffer();  
    buffer.append("http://api.olhovivo.sptrans.com.br/v0/Linha/Buscar?termosBusca=");  
    buffer.append(busLine);  
  
    URL url = new URL(buffer.toString());  
  
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();  
    connection.setRequestProperty("Cookie", cookie);  
  
    BufferedReader in = new BufferedReader(new InputStreamReader(  
        (connection.getInputStream())));  
    String inputLine;  
    String total = "";  
  
    while ((inputLine = in.readLine()) != null) {  
        total = total + inputLine;  
    }  
    System.out.println(total);  
  
    JSONParser parser = new JSONParser();  
  
    JSONArray linhas = (JSONArray) parser.parse(total);  
  
    Iterator<JSONObject> linhasIterator = linhas.iterator();  
    while (linhasIterator.hasNext()) {  
        JSONObject linha = (JSONObject) linhasIterator.next();  
  
        Long codigo = (Long) linha.get("CodigoLinha");  
        String letreiro = (String) linha.get("Letreiro");  
        String denominacaoTPTS = (String) linha.get("DenominacaoTPTS");  
        String denominacaoTSTP = (String) linha.get("DenominacaoTSTP");  
        Long sentido = (Long) linha.get("Sentido");  
  
        url = new URL("http://api.olhovivo.sptrans.com.br/v0/  
Posicao?codigoLinha=" + codigo);  
  
        connection = (HttpURLConnection) url.openConnection();  
        connection.setRequestProperty("Cookie", cookie);  
  
        connection.toString();  
  
        in = new BufferedReader(new InputStreamReader(  
            (connection.getInputStream())));  
        inputLine = "";  
        total = "";  
  
        while ((inputLine = in.readLine()) != null) {  
            total = total + inputLine;  
        }  
        JSONObject units2 = (JSONObject) parser.parse(total);  
  
        String hr = (String) units2.get("hr");  
        JSONArray onibusLista = (JSONArray) units2.get("vs");  
  
        Iterator onibusIterator = onibusLista.iterator();  
        while (onibusIterator.hasNext()) {  
            JSONObject onibus = (JSONObject) it.next();  
            Double py = (Double) onibus.get("py");  
            Double px = (Double) onibus.get("px");  
  
            String denominacao = sentido == 1 ? denominacaoTPTS : denominacaoTSTP;  
  
            Bus bus = new Bus();  
            bus.setCode((Integer.parseInt((String) onibus.get("p"))));  
            bus.setLat(py);  
            bus.setLon(px);  
            bus.setName(denominacao);  
            bus.setLetters(letreiro);  
  
            buses.add(bus);  
  
            System.out.println("Codigo: " + bus.getCode()  
                + " Letreiro: " + letreiro + " Denominacao:  
                + denominacao + " Hora: " + hr + " lat: " + py  
                + " long: " + px);  
        }  
        in.close();  
        return buses;  
    }  
}
```

A **Listagem 6** mostra o código para realizar as operações supracitadas. Primeiro, note que adicionamos ao final do endereço da requisição a **String** do parâmetro **busLine**, parâmetro que deve conter as letras de uma linha de ônibus da cidade de São Paulo. Em seguida, é estabelecida a conexão com o serviço *Buscar* através do objeto **connection**, da classe **HttpURLConnection**. Para a requisição funcionar, ainda é necessário informar o cookie que foi recuperado, o que é feito passando o mesmo como parâmetro para o método **setRequestProperty()**, também do objeto **connection**.

Como o resultado da chamada ao serviço é dado no formato JSON, para ler esses dados utilizamos o framework **JSON.simple**, mais precisamente um objeto da classe **JSONParser**. A partir dele recuperamos a lista dos ônibus, e nessa lista é feita uma iteração, momento em que os dados são extraídos utilizando o identificador de cada campo.

Depois de obter os dados do serviço *Buscar*, principalmente o código da linha, é possível chamar o serviço *Posicao*, que recupera os dados de todos os ônibus desta linha que estão na rua, entre eles a latitude e a longitude. Como esperado, esse serviço retorna outro JSON, e para ler seus dados é feito o mesmo processamento da chamada ao serviço *Buscar*. Nesse JSON, os principais dados são o **p**, que representa o código do ônibus, e **py** e **px**, que representam a latitude e a longitude.

Ao final, os dados recuperados com as chamadas a esses serviços são adicionados a um objeto do tipo **Bus**, que, por sua vez, é incluído em uma lista de ônibus, a lista **buses**, retornada pelo método.

Finalmente, com os dados de todos os ônibus recuperados, é possível armazená-los no Elasticsearch. A **Listagem 7** mostra o código que recebe os dados de um ônibus e os salva. Como podemos verificar, esse código é bastante simples. Inicialmente, é estabelecida a conexão com o servidor local do Elasticsearch através da classe **Client**, e depois são colocados em um **HashMap** os dados que serão salvos.

Com os dados já preparados, indicamos no método **prepareIndex()**, da classe **Client**, o nome do índice no qual os dados serão armazenados. E por último, chamamos o método **execute()**, que envia o comando *put* para o Elasticsearch, que é o comando responsável pela inserção dos dados. Para facilitar o nosso trabalho, a API do Elasticsearch abstrai a conversão do **HashMap** em um JSON.

Ainda na **Listagem 7**, note que os dados de latitude e longitude são colocados em um vetor chamado **posição**. Esse é o modo adotado pelo Elasticsearch para entender que estas informações se tratam dos dados de uma posição geográfica. Voltando à **Listagem 1**, verifique que esses dados foram definidos no mapeamento como sendo do tipo **geo_point**. Tais dados podem até ser salvos diretamente como **String** ou **float**, mas algumas ferramentas, como é o caso do Kibana, uma solução de visualização de dados para o Elasticsearch, não relacionará automaticamente que eles representam posições geográficas.

Listagem 7. Código para salvar os dados de um ônibus no Elasticsearch.

```
public void putDataBus(int busCode, int lineCode, String name, String letters,
String lat, String lon, Date date) {

    Client client = new TransportClient().addTransportAddress
    (new InetSocketAddress("localhost", 9300));

    double posicao [] = new double[2];
    posicao[0] = Double.parseDouble(lat);
    posicao[1] = Double.parseDouble(lon);

    Map<String, Object> data = new HashMap<String, Object>();

    data.put("busCode", "" + busCode);
    data.put("lineCode", "" + lineCode);
    data.put("location", posicao);
    data.put("name", name);
    data.put("letters", letters);
    data.put("dateBus", date);

    IndexResponse response = client.prepareIndex("sptrans", "onibus")
    .setSource(data).execute().actionGet();

    client.close();
}
```

Como a posição dos ônibus muda a todo instante, é necessário criar um mecanismo que permaneça em execução e periodicamente envie uma solicitação à API da SPTrans para recuperar as novas posições. A **Listagem 8** mostra o código que realiza essa função, utilizando todos os métodos que criamos anteriormente.

Listagem 8. Aplicação que recupera os dados dos ônibus e os salvo no Elasticsearch

```
package com.santana.sc.esclient;

import java.util.Date;
import java.util.List;

import com.santana.sc.sptrans.Bus;
import com.santana.sc.sptrans.Crawler;

public class Main {

    public static void main(String[] args) throws Exception {

        while (true) {
            Crawler c = new Crawler();
            String cookie = c.getToken();
            List<Bus> buses = c.getBusesByLine(cookie, "715M-10");

            ESClient client = new ESClient();
            for (Bus b : buses) {
                client.putDataBus(b.getCode(), 0, b.getName(), b.getLetters(),
                "" + b.getLatitude(), "" + b.getLongitude(), new Date());
            }

            // executa de 30 em 30 segundos
            Thread.sleep(30000);
        }
    }
}
```

Listagem 9. Configuração do arquivo pom.xml para o projeto web.

```
<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/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.santana.sc</groupId>
<artifactId>monitor</artifactId>
<packaging>war</packaging>
<version>0.0.1-SNAPSHOT</version>
<name>monitor Maven Webapp</name>
<url>http://maven.apache.org</url>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.5</version>
  </dependency>
  <dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>1.2</version>
  </dependency>
  <dependency>
    <groupId>com.santana.sc</groupId>
    <artifactId>sptrans</artifactId>
    <version>0.0.1</version>
  </dependency>
</dependencies>
<build>
  <finalName>monitor</finalName>
</build>
</project>
```

Inicialmente, é chamado o método `getCookie()` para recuperar o cookie da API. Depois, é chamado o método `getBusesByLine()`, que retorna a lista de ônibus para uma linha passada como parâmetro. Nesse exemplo estamos utilizando diretamente a linha “715M-10”, mas podemos utilizar outras linhas, e até mesmo pedir para um usuário da aplicação digitar a linha desejada. Finalmente, os dados desses ônibus são salvos no Elasticsearch com a invocação do método `putDataBus()`. Como a API demora um pouco para atualizar a posição dos ônibus, esperamos 30 segundos para realizar uma nova requisição.

Assim, a aplicação para recuperar os dados dos ônibus e armazená-las no Elasticsearch está pronta. A partir disso, é possível desenvolver diversos tipos de soluções e serviços como, por exemplo, uma aplicação para calcular dados sobre os ônibus como velocidade e tempo de uma rota, uma aplicação para verificar o trânsito nas linhas de ônibus da cidade e uma aplicação para monitorar a posição dos ônibus da cidade em tempo real.

Aplicação que exibe a posição dos ônibus em tempo real

Para demonstrar a utilização dos dados recuperados pela primeira aplicação, vamos desenvolver uma aplicação que exibe esses dados no Google Maps. Para isso, de 30 em 30 segundos, ela realizará a leitura dos dados mais recentes dos ônibus que foram salvos no Elasticsearch

Com os dados no Elasticsearch, é possível também calcular outras informações, como a rota que um ônibus percorreu e o tempo que ele ficou parado em determinado lugar. Por questão de espaço, no entanto, iremos focar apenas em mostrar a posição dos ônibus.

Dito isso, primeiramente criamos um novo projeto web e o configuramos com o Maven. Para o nosso caso, as dependências serão a `servlet-api`, pois o projeto será uma aplicação web; o `jstl`, pois usaremos JSTL na implementação da página que mostra os ônibus no Google Maps; o driver de conexão do Elasticsearch para recuperar os dados dos ônibus; e também a dependência da aplicação já desenvolvida, para utilizar a classe `Bus`.



Open Data: desenvolvendo uma aplicação para monitoramento de ônibus

A **Listagem 9** mostra a configuração do *pom.xml* para essa aplicação.

Agora, para iniciar de fato a implementação, crie a classe **ES-Client** e codifique o método **getBuses()**, responsável por buscar os dados no Elasticsearch (vide **Listagem 10**). Como parâmetro ele deve receber o letreiro da linha que desejamos recuperar.

Listagem 10. Código do método **getBuses()** - Recupera os dados no Elasticsearch.

```
public List<Bus> getBuses(String letters) {
    Client client = new TransportClient()
        .addTransportAddress(new InetSocketAddress("localhost", 9300));

    SearchResponse response = client.prepareSearch("sptrans")
        .setQuery(QueryBuilders.termQuery("letters", letters))
        .setFrom(0).setSize(60).execute().actionGet();

    SearchHit[] results = response.getHits().getHits();
    List<Bus> buses = new ArrayList<Bus>();
    for (SearchHit hit : results) {
        Bus bus = new Bus();
        Map<String, Object> result = hit.getSource();
        bus.setLetters((String) result.get("letters"));
        ArrayList lista = (ArrayList) result.get("location");
        bus.setLat((Double) lista.get(0));
        bus.setLon((Double) lista.get(1));
        buses.add(bus);
    }

    client.close();
    return buses;
}
```

Após definir a assinatura, logo no início do método, é estabelecida a conexão com o servidor do Elasticsearch, através da classe **Client**, e depois é feita a busca pelo letreiro do ônibus utilizando a classe **SearchResponse**. Como parâmetros dessa consulta são passados o nome do índice, com o método **prepareSearch()**, e o termo para consulta, que neste caso é o letreiro do ônibus. Além disso, o método **setSize()** indica a quantidade máxima de respostas, e o método **actionGet()** indica ao Elasticsearch que será utilizada uma requisição do tipo **GET**. Assim, todos os ônibus que correspondem ao letreiro passado como parâmetro serão retornados.

Depois disso, com o método **getHits()** é criado um vetor de objetos da classe **SearchHit**, que contém todos os ônibus retornados pelo Elasticsearch. A partir desse vetor, durante uma

iteração, com as informações presentes em cada posição é criado um objeto **Bus**, que é adicionado em uma lista a ser retornada ao final do método.

Com o método para recuperar os dados implementado, vamos desenvolver a parte web da aplicação, a qual possibilitará a visualização dos dados. Para isso, precisamos criar um Servlet, que será responsável por atender à requisição, chamar o método **getBuses()** para recuperar os dados dos ônibus e os encaminhar para a página que contém o mapa (*listaBus.jsp*). Apesar de a API de Servlet não ser a melhor opção para o desenvolvimento de aplicações web complexas, avaliamos que para este exemplo ela é suficiente. A **Listagem 11** mostra o código de **MapServlet**.

Listagem 11. Servlet que recupera os dados do Elasticsearch.

```
public class MapServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        ESClient client = new ESClient();
        List<Bus> lista = client.getBuses("715M");
        request.setAttribute("lista", lista);
        RequestDispatcher rd = getServletContext().getRequestDispatcher(
            "/listaBus.jsp");
        rd.forward(request, response);
    }
}
```

Considerando que o Tomcat foi configurado na porta padrão 8080 e o nome da aplicação é *monitor*, esse servlet deve atender a todas as requisições feitas à URL <http://localhost:8080/monitor/MapServlet>.

O nosso último passo é apresentar os dados dos ônibus no Google Maps. Para isso, será utilizada a API Google Maps V3, desenvolvida em JavaScript.

A **Listagem 12** mostra, além do código HTML, o código que recupera as posições dos ônibus que foram passados na requisição para a página *listaBus.jsp* e os adiciona no mapa. Essas ações serão realizadas na função JavaScript **initialize()** que, logo nas primeiras linhas, define o ponto inicial para exibição do mapa com uma latitude e longitude do centro de São Paulo. Depois, para iterar sobre os dados que estão na requisição, utilizamos JSTL. Com seus recursos, para cada ônibus que está na lista chamamos a função JavaScript **mark()**, que com a latitude e longitude de cada objeto coloca um ponto no mapa utilizando a classe **google.maps.Marker**, pertencente à API do Google Maps.



Para concluir, no final da página é criada uma tabela HTML, que reúne todos os ônibus que estão sendo acompanhados e expõe o nome da linha e a latitude e longitude de cada um.

A Figura 1 mostra como fica o mapa com os pontos identificando onde cada ônibus se encontra. E para atualizar o mapa de acordo com os dados recuperados da API Olho Vivo, essa tela também é atualizada a cada 30 segundos. O código <meta http-

equiv="refresh" content="30"> pode ser verificado nas primeiras linhas da Listagem 12.

Já a **Figura 2** mostra a tabela que contém as informações dos ônibus que estão sendo exibidos no mapa.

A utilização de dados e padrões abertos é uma tendência no mundo todo e será muito importante para que os cidadãos possam, por exemplo, acompanhar serviços públicos, como a posição dos

Listagem 12. Código que exibe os dados dos ônibus no Google Maps.

```
<html>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<head>
<title>Lista e Mapa dos ônibus da cidade de São Paulo</title>
<meta name="viewport" content="initial-scale=1.0, user-scalable=no">
<meta charset="utf-8">
<meta http-equiv="refresh" content="30">
<style>
html, body, #map-canvas {
height: 100%;
margin: 0px;
padding: 0px
}
</style>
<script src="https://maps.googleapis.com/maps/api/js?v=3.exp"></script>
<script>
function initialize() {
var myLatlng = new google.maps.LatLng(-23.57374162500004, -46.726239);
var mapOptions = {
zoom: 11,
center: myLatlng
}
var map = new google.maps.Map(document.getElementById('map-canvas'),
mapOptions);
var text = "";

<c:forEach var="bus" items="${lista}">
    mark(map, <c:out value="${bus.lat}" />, <c:out value="${bus.lon}" />,
    '<c:out value="${bus.letters}" />');
</c:forEach>
}

function mark(map, lat, lon, text) {
```

```
var myLatlng = new google.maps.LatLng(lat, lon);
var marker = new google.maps.Marker({
  position: myLatlng,
  map: map,
  title: text
});
}

google.maps.event.addDomListener(window, 'load', initialize);
</script>
</head>
<body>
  <div id="map-canvas"></div>

<c:out value="Lista de Ônibus">
  <table>
    <tr>
      <td>Nome</td>
      <td>Longitude</td>
      <td>Latitude</td>
    </tr>
  </table>
  <c:forEach var="bus" items="${lista}">
    <tr>
      <td><c:out value="${bus.letters}"></c:out></td>
      <td><c:out value="${bus.lon}"></c:out></td>
      <td><c:out value="${bus.lat}"></c:out></td>
    </tr>
  </c:forEach>
  </table>

</body>
</html>
```

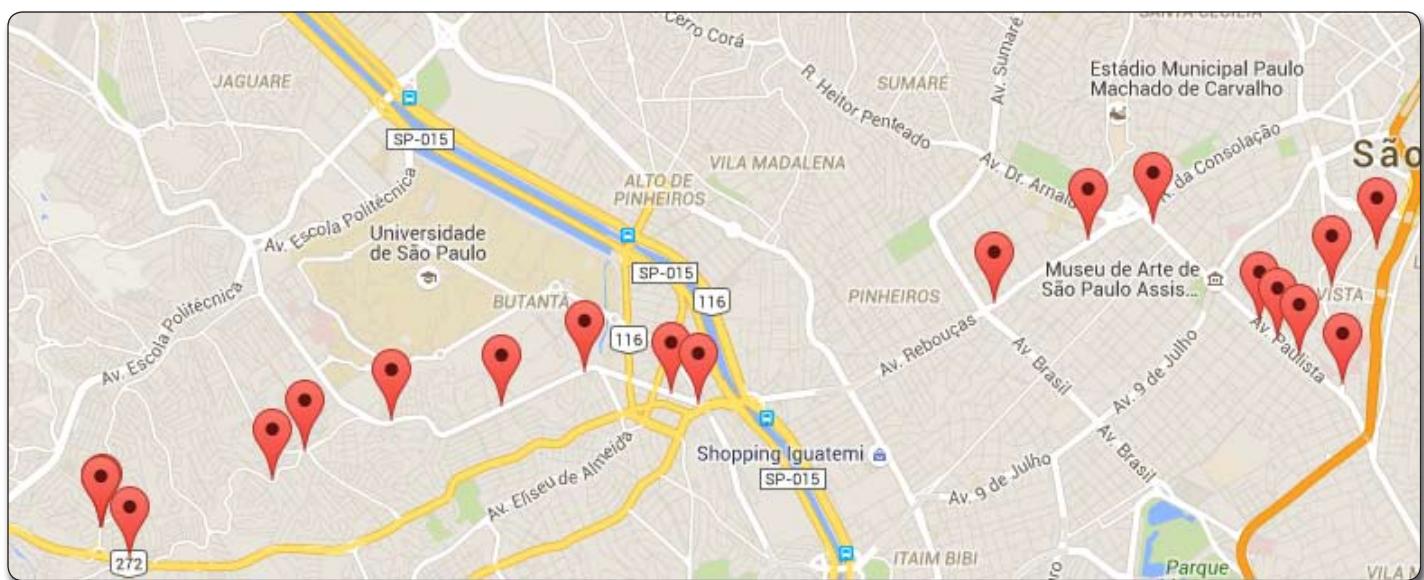


Figura 1. Mapa da cidade de São Paulo com a última posição dos ônibus monitorados

Open Data: desenvolvendo uma aplicação para monitoramento de ônibus

ônibus de uma cidade e a quantidade de leitos livres em hospitais. Outro ponto em que os dados abertos podem ser úteis é o acompanhamento das decisões tomadas pelos políticos, o que pode ser feito com os dados de portais de transparência já implementados em diversas cidades do país.

Nome	Long	Lat
715M	-46.73421899999996	-23.573938
715M	-46.742559	-23.5765575
715M	-46.66796625	-23.558115625
715M	-46.7618985	-23.5832605
715M	-46.70763050000001	-23.571608250000004
715M	-46.649944250000004	-23.5667458125
715M	-46.7618985	-23.5832605

Figura 2. Lista com os ônibus que estão sendo exibidos no mapa

Acreditamos que essa tendência ficará ainda mais forte com a ideia de cidades inteligentes e internet das coisas, o que permitirá a geração e monitoração de muito mais dados que existem hoje como, por exemplo, dados sobre a situação do trânsito, sobre o consumo de água e energia de casas e de edifícios governamentais, sobre a geração e coleta de lixo nas cidades, entre muitas outras opções.

Nesse contexto, a utilização de padrões abertos é fundamental, porque é provável que todos esses recursos sejam combinados para a criação das mais variadas aplicações. E para isso, é necessária alguma forma de facilitar a interoperabilidade entre esses diferentes dados e serviços.

Por fim, destacamos ainda que com os dados coletados de APIs como a Olho Vivo, outros diferentes serviços podem ser elaborados, sendo possível verificar, por exemplo, problemas no transporte público da cidade, como descobrir quais ônibus estão atrasados, calcular informações como velocidade média, identificar veículos que saem da rota, entre outros.

Autor



Eduardo Felipe Zambom Santana

ezambomsantana@gmail.com

É bacharel e mestre em Ciéncia da Computação pela UFSCar.

Possui mais de 10 anos de experiência em programação com Java.

Possui as certificações OCPJP, OCPJWCD e IBM DB2 DBA. Atualmente é aluno de doutorado na USP e professor na Universidade Anhembi Morumbi.



Autor



Luiz Henrique Zambom Santana

lhzsantana@gmail.com

É bacharel e mestre em Ciéncia da Computação. Possui mais de 10 anos de experiência em programação Java e há dois anos

atua com tecnologias de Big Data. Trabalhou em projetos para grandes empresas no Brasil, Argentina e Alemanha. Atualmente é consultor, startupper na 33! e doutorando na UFSC.



Links:

API da SPTrans.

<http://www.sptrans.com.br/desenvolvedores/APIOlhoVivo.aspx>

Documentação da API Olho Vivo.

<http://www.sptrans.com.br/desenvolvedores/APIOlhoVivo/Documentacao.aspx?>

GitHub do projeto JSON.simple.

<https://github.com/fangyidong/json-simple>

Documentação da Google Maps API.

<https://developers.google.com/maps/?hl=en>

Plugin Sense do Google Chrome para manipulação de dados no Elasticsearch.

<https://goo.gl/3YSmH2>

Site oficial do Elasticsearch.

<https://www.elastic.co/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Desenvolva aplicações web escaláveis com Crux Framework

Entenda como o Crux framework funciona e porquê utilizá-lo para construir aplicações web escaláveis que exploram os recursos do HTML5 e exigem o máximo de desempenho

Antes de desenvolver uma solução web, toda empresa deve avaliar a seguinte pergunta: para onde a web está caminhando? Ao responder esta pergunta, a empresa pode (e deve) convergir a aplicação em construção para este caminho.

Atualmente, dentre as várias tendências na web, vamos destacar duas:

- Aplicações que rodam diretamente no cliente, consultando o servidor apenas quando realmente for necessário, como o Gmail do Google, onde grande parte da sua lógica de controle está no cliente;
- Aplicações multiplataformas e multidispositivos, como aplicativos que têm como requisito serem utilizados por usuários de todo o mundo, como o Google Maps que, por exemplo, pode ser executado em um computador desktop utilizando uma plataforma Linux, bem como em um dispositivo celular utilizando uma plataforma iOS.

Ambas as tendências sinalizam que a web está deixando aquele conceito básico de páginas que rodam somente em um browser desktop e se concentrando em novas plataformas e dispositivos. Agora, não construímos mais páginas web, mas sim aplicativos web! E estes devem ser robustos, portáveis, fáceis de desenvolver e manter.

Diante deste cenário, o Crux Framework provê ferramentas que facilitam ao desenvolvedor trabalhar com estas tendências de forma natural e intuitiva.

Fique por dentro

Neste artigo vamos conhecer o Crux Framework, solução nacional e open source construída para o desenvolvimento front-end de aplicações web. No entanto, antes de nos aprofundarmos no funcionamento e nos conceitos do Crux, analisaremos o que tem sido considerado por muitos como o futuro das aplicações web e então, como o Crux está em sintonia com esta visão. Em seguida, vamos conhecer um pouco sobre as funcionalidades e a arquitetura do framework, e ao final, será apresentada uma pequena sequência de passos para que você possa aprender de uma forma simples e rápida como desenvolver e executar um projeto Crux.

É um framework construído sobre o GWT (*Google Web Toolkit*) voltado para acelerar o desenvolvimento do *front-end* de aplicações web que exigem o máximo de desempenho e que sejam adaptáveis aos diversos tipos de dispositivos existentes.

Crux é um termo em inglês que significa o ponto central de uma argumentação ou um problema muito complexo. No jargão dos esportistas de escalada, “crux” é a parte mais difícil de um percurso. Esse nome representa muito bem o mesmo, pois a sua proposta é justamente resolver os problemas mais complexos do desenvolvimento de aplicações, oferecendo a seus usuários uma forma simples de transformar ideias em soluções.

Utilizado em grandes empresas dos ramos de Telecom, Saúde, Cosméticos e Auditoria, o Crux oferece um modelo de desenvolvimento que permite escrever o código da camada front-end em

Java e XHTML, sendo então compilado para resultar em artefatos JavaScript e HTML, formatos interpretados pelo browser. Neste momento, vale ressaltar que quaisquer otimizações no código são realizadas em tempo de compilação, o que possibilita gerar um código específico e customizado para cada tipo de dispositivo, como smartphones, tablets, desktops e até TVs.

Como o Crux aborda as tendências da web?

Saber desenvolver aplicações que rodam diretamente no cliente e são multiplataformas e multidispositivos é o diferencial para qualquer empresa de desenvolvimento que deseja se manter no mercado, principalmente no mercado mobile, que está em grande ascensão. Diante desse cenário, veremos a seguir como o Crux lida com estas tendências e como o desenvolvedor pode se nutrir disso para construir suas aplicações.

Aplicações que rodam diretamente no cliente

Esse tipo de aplicação está diretamente relacionado ao que chamamos de *rich client*, conceito que identifica soluções que possuem grande parte do código executada no cliente. Tal abordagem permite alguns ganhos bastante significativos, uma vez que desafoga os servidores ao processar informações diretamente no cliente. Como os clientes estão ficando cada vez mais poderosos, esta prática vem se tornando comum.

Outro conceito importante é a utilização de uma *Single Page Application* (ou aplicação de página única). Esta opção sugere que a aplicação seja renderizada em apenas uma página web, que jamais é recarregada completamente durante as requisições. Para que a atualização do conteúdo ocorra, são utilizadas requisições AJAX, que trazem informações para modificar porções da tela. No caso de uma aplicação onde apenas partes do conteúdo são atualizadas, esta abordagem permite o reaproveitamento de várias estruturas, garantindo um cenário de otimização de recursos e, consequentemente, desempenho.

Esta abordagem é muito promissora, porém também oferece certos desafios. Um deles é que grande parte do conteúdo da página seria trazido de uma vez, o que pode ser um volume grande de dados dependendo do tamanho da aplicação. No entanto, este é um problema que pode ser contornado através da utilização eficiente e regrada dos recursos a serem trazidos. Por exemplo, se estamos requisitando um recurso de um cliente desktop, não é recomendado trazer do servidor recursos relacionados a um cliente mobile. O Crux oferece uma maneira simples de tratar este problema, enviando somente o código necessário para o dispositivo. Na seção “A expansão do conceito de design responsivo”, veremos mais informações sobre este assunto.

Se ainda assim o volume de dados for muito grande, pode-se considerar a utilização de cache de recursos, que pode ser habilitado manipulando os cabeçalhos HTTP das requisições. Por fim, caso a busca inicial pelo recurso (antes de ocorrer um *cache hit*) estiver lenta, a utilização de um CDN (veja o **BOX 1**) pode resolver o problema de forma definitiva.

BOX 1. CDN

Acrônimo para Content Delivery Network, termo que representa uma grande rede de computadores instalada em vários data centers na Internet. O objetivo do CDN é servir conteúdo com alta disponibilidade e desempenho aos usuários finais.

Outro desafio ao utilizar uma aplicação corporativa rica em conteúdo e executando em uma única página é que no mundo web este código geralmente é desenvolvido com HTML e JavaScript, o que, particularmente, torna o código custoso de manter e otimizar. Lembre-se que o JavaScript não possui, de forma nativa, conceitos que permitem um maior grau de abstração na linguagem (como anotações, pacotes, interfaces, etc.). Com o Crux, no entanto, desenvolvemos a aplicação em Java, uma linguagem mais expressiva, que melhora a manutenibilidade e organização do código. Além disso, o Crux oferece ao desenvolvedor uma série de ferramentas que auxiliam a construção de aplicações que rodam diretamente no cliente, as quais iremos conhecer no decorrer do artigo.

Aplicações Multiplataformas e Multidispositivos

Aplicações crossdevice são aquelas construídas para funcionar de forma similar em diversos dispositivos, como PCs, tablets, celulares, TVs, etc. Já as aplicações crossplatform são aquelas construídas para funcionar nas diversas plataformas existentes, como Windows, Linux, Android, etc. Quando dizemos que uma aplicação é crossdevice e crossplatform, esta executará em todos os dispositivos e plataformas previstas, geralmente produzindo as mesmas funcionalidades e alcançando o mesmo objetivo final.

Dificilmente uma empresa se dará ao luxo de produzir aplicações específicas para cada um de seus dispositivos e plataformas alvo. Neste cenário, optando por minimizar o custo do projeto, geralmente são adotados frameworks que possibilitam ao desenvolvedor escrever apenas um código e compilá-lo (ou interpretá-lo) para os demais ambientes. Por exemplo, no caso das aplicações mobile (que serão executadas em dispositivos com iOS, Android, Windows Phone, etc.), podemos contar com diversos frameworks como, por exemplo, Appcelerator, Adobe AIR, Sencha, Unity, Corona, Mono, dentre outros.

Embora estes frameworks sejam relativamente recentes, se paramos para pensar, já possuímos tecnologias crossdevice e crossplatform sendo desenvolvidas há mais de 20 anos! Uma aplicação web que executa sobre browsers como o Chrome, Firefox, Internet Explorer, Opera, etc. é, por definição, crossdevice e crossplatform. Note que estes browsers funcionam nos mais variados dispositivos (incluindo TVs!) e também nas mais variadas plataformas. Então, para que reinventar a roda se esforçando para construir ferramentas que permitem o desenvolvimento multiplataforma e multidispositivo se o próprio browser faz este papel? De fato, ele provê uma camada de abstração entre o dispositivo e a aplicação, oferecendo a possibilidade de desenvolvêrmos para esta camada e abstrairmos grande parte das informações sobre onde a aplicação estará rodando.

Um dos problemas desta abordagem é que constantemente precisamos ultrapassar esta camada e acessar alguma funcionalidade/recurso específico do dispositivo como, por exemplo, a câmera ou a agenda de contatos. Para isso, o browser deve acessar esta funcionalidade no dispositivo onde está sendo executado e permitir seu acesso através de uma API JavaScript.

A definição sobre o que o browser pode acessar no dispositivo é determinada por especificações regidas pela W3C, que é a principal organização de padronização da Internet. A partir disso, as empresas mantenedoras dos browsers podem decidir se vão seguir a especificação e implementá-la (completa ou parcialmente), criar sua própria especificação (desvinculando-se da W3C) ou mesmo não implementar nenhum recurso previsto. Atualmente, no entanto, as funcionalidades que normalmente são utilizadas em aplicações corporativas já se encontram implementadas na maioria dos browsers, e em conformidade com as especificações da W3C. Deste modo, já conseguimos construir aplicações web que funcionem off-line, persistem dados no próprio browser, vibrem quando o usuário tocar em algum botão, acessem a câmera do usuário e ainda detectem a sua localização.

Estas funcionalidades fazem parte do HTML5, a grande especificação que está tornando a web cada vez mais poderosa e funcional. Através do HTML5 temos acesso a opções que nos permitem concentrar grande parte do processamento da aplicação no cliente, construindo uma aplicação web que se comporta de forma semelhante a uma aplicação nativa. Para auxiliar nesta tarefa, o Crux oferece suporte às mais variadas bibliotecas do HTML5, trabalhando de forma inteligente para facilitar a utilização de cada uma ou a integração entre elas.

Conhecendo os recursos do Crux

O Crux oferece um rico conjunto de recursos para o desenvolvimento de aplicações que rodam no cliente e precisam ser multiplataformas e multidispositivos. Os próximos tópicos oferecem um overview destes recursos e permitem aprofundarmos nosso conhecimento um pouco mais no framework.

Simplificando o uso das APIs do HTML5

Normalmente, quando vamos utilizar uma API do HTML5 como, por exemplo, FileStorage, AppCache, WebSQL, IndexedDB, Canvas, etc., escrevemos suas chamadas diretamente no código JavaScript da aplicação. No entanto, esse código pode se tornar extenso, complexo e de difícil manutenção porque o JavaScript não oferece suporte a recursos que geralmente as linguagens compiladas oferecem, como reflexão e anotações, que nos permitem escrever menos código e deixá-lo mais organizado.

O Crux permite o desenvolvimento do nosso código seja feito em Java, em vez de JavaScript. Para exemplificar as vantagens disso, vamos imaginar que nosso cliente queira persistir dados no browser para trabalhar de forma off-line em sua aplicação. Logo, devemos recorrer às APIs de persistência de dados do HTML5, como a IndexedDB (focada no modelo orientado a objetos)

ou a WebSQL (focada no modelo relacional). Um problema disso é que, infelizmente, alguns browsers não implementam ambas as APIs. Com isso, caso queiramos que nossa aplicação funcione em todos eles, devemos escrever código para lidar com o IndexedDB e com o WebSQL. Com o Crux, podemos escrever nosso código em Java utilizando uma API única de persistência Java (fundamentada na JPA) e deixar o trabalho de detectar a API que o browser suporta e fazer a tradução do código Java para o JavaScript da API alvo por conta do framework.

Quando trabalhamos com JPA, estamos acostumados a utilizar anotações em nossas classes de persistência para denotar comportamentos específicos. Por exemplo, podemos anotar nossas classes com a `@Entity` para indicar que aquela é uma classe de persistência, ou mesmo anotar nossos atributos com `@NotNull` para sinalizar que aquele atributo não deve ser nulo no banco. Estas opções trazem um poder de abstração muito grande e nos permite construir a camada de persistência de uma maneira simples e rápida. Imagine, agora, que podemos utilizá-las também no lado cliente da aplicação, de uma forma semelhante à que estamos acostumados. Isto vai contribuir para melhorarmos o tempo de desenvolvimento, uma vez que possibilita reaproveitarmos conceitos que já estamos acostumados a trabalhar.

Para completar o exemplo, durante o mapeamento dos DAOs, ainda podemos aproveitar os mecanismos de interface e herança do Java e escrever classes abstratas que já implementam comportamentos de inserção, remoção e alteração de registros, bastando apenas estendê-los.

Com o Crux, o que mencionamos até aqui passa a ser possível para a camada de front-end, através do uso de um mecanismo conhecido como amarração tardia (em inglês, *Deferred Binding*), do GWT. Este é uma alternativa à reflexão e oferece um poder de expressão enorme se combinarmos com as anotações do Java.

Vale destacar que a utilização de persistência na camada de front-end é apenas um dos benefícios que o Crux oferece através das APIs IndexedDB e WebSQL. Ou seja, o Crux não se limita a isto, explorando também a maioria das APIs do HTML5, principalmente as utilizadas no meio corporativo, possibilitando assim uma ótima maneira de se trabalhar com todas elas, seja através de anotações, interfaces, entre outras possibilidades.

A expansão do conceito de design responsivo

Quando pensamos em Design Responsivo logo nos vem à cabeça que o layout das páginas se altera dependendo do tamanho da tela do dispositivo onde estão sendo exibidas. Este quesito geralmente é suficiente para garantirmos que nossa aplicação funcionará de maneira adequada em cada dispositivo, correto? Errado.

Mais que o layout, o modo de interação dos componentes pode/deve variar em cada dispositivo. Por exemplo, em um dispositivo desktop, um slideshow de imagens deve exibir botões de avançar e retroceder a imagem que está sendo exibida. Já em um dispositivo mobile (com funcionalidades de *touchscreen*), estes botões são desnecessários.

Mais importante do que isto é que a aplicação no dispositivo mobile, uma vez que não possui os botões de avançar e retroceder, não deveria sequer tê-los recebido do servidor, mesmo que os oculte durante a exibição. No caso de uma aplicação que não se preocupe com isto, o botão de avançar e retroceder (e o código relacionado com o comportamento que deve ser acionado caso o usuário os toque) são enviados para o cliente e lá é realizada uma lógica para verificar se eles devem ser exibidos dependendo do tamanho da tela. Supondo que nosso cliente seja um cliente mobile, estes botões nunca serão exibidos para o usuário, mas estarão presentes na aplicação.

O Crux se preocupa com esta otimização e a executa de forma automática e em tempo de compilação, enviando para o cliente somente o código que o dispositivo processará, evitando, portanto, que imagens, CSS, JavaScript, entre outros artefatos sejam carregados de forma desnecessária. Se enviarmos somente o que for preciso, minimizamos a necessidade de banda de rede e aumentamos a velocidade da aplicação, uma vez que não precisaremos executar lógicas em tempo de execução, através de JavaScript, para descobrir como determinado componente da tela deve ser exibido dependendo do tamanho da tela.

Os dispositivos atendidos e a otimização do código JavaScript

Quando acessamos uma URL de uma aplicação web construída em Crux, primeiramente é carregado um script simples, responsável por identificar o ambiente que está executando a aplicação. Uma vez identificado, o Crux requisita os próximos recursos HTML e JavaScript correspondentes ao ambiente identificado. Estes recursos variam em função de quatro propriedades:

- **Idioma:** Português, inglês, etc.;
- **Browser:** Chrome, Safari, Firefox, Internet Explorer, etc.;
- **Tamanho da tela:** Dispositivos grandes (como desktops, tablets e televisões) ou dispositivos pequenos (como celulares);
- **Modo de interação:** Através do mouse, touchscreen ou setas.

A partir da combinação destas propriedades o Crux gera recursos únicos. Assim, um usuário que esteja acessando a aplicação de um tablet utilizando o Safari, por exemplo, receberá somente os recursos correspondentes ao seu ambiente, e nada mais além disso. Ou seja, as especificidades de cada propriedade são tratadas caso a caso e isto garante que o código final a ser enviado para o cliente seja único e otimizado. Estes recursos são gerados no momento da compilação e ficam disponíveis no servidor na forma de arquivos estáticos. Como são várias as combinações, dependendo do tamanho da aplicação final, este código pode ultrapassar a ordem de 500 MB, mas isto não é um problema, uma vez que eles ficarão armazenados no servidor e o cliente somente vai requisitar o que lhe interessa.

Ademais, em se tratando de código JavaScript a ser enviado para o cliente, existem diversas otimizações que somente desenvolvedores experientes conhecem ou sabem aplicar. Com o Crux não precisamos nos preocupar com isto, pois o próprio framework (juntamente com o compilador do GWT) se encarrega de otimizar

(veja o **BOX 2**), deixando-o o mais performático possível. Vale lembrar que estas otimizações são essenciais para rodar a aplicação em um dispositivo mobile, uma vez que o poder de processamento destes muitas vezes é limitado.

BOX 2. Exemplos de otimizações GWT na compilação

O compilador GWT realiza diversas otimizações sobre o código, estas que podem ser classificadas em dois tipos:

- Otimizações que garantem um código final menor, permitindo que esteja disponível apenas o que realmente for utilizado pela aplicação. Por exemplo: remoção de classes, interfaces, métodos e campos sem referência;
- Otimizações para melhorar o desempenho da aplicação, garantindo que o ponteiro de execução realize menos passos para se chegar ao resultado desejado. Por exemplo, eliminar partes irrelevantes em condições e expressões lógicas (tais como “|| true”, ou mesmo “a + 0”), que não trazem qualquer efeito sobre a execução do programa.

Conhecendo um pouco sobre a arquitetura do Crux

Até aqui, vimos alguns dos benefícios de se utilizar o Crux. A partir de agora, vamos nos aprofundar um pouco mais na arquitetura do framework e no código fonte de uma aplicação construída com ele.

Com relação à arquitetura, o Crux emprega o padrão MVC, solução bastante comum às aplicações hoje em dia e que permite separar a representação da informação da interação do usuário com ela. Diferentemente do GWT, que utiliza o padrão MVP (Modelo-Visão-Apresentação), o Crux adotou essa opção por se tratar de um modelo mais simples e usual, permitindo ao desenvolvedor (muitas vezes) utilizar apenas um Controlador para uma View, enquanto no MVP é comum utilizarmos várias Apresentações para uma mesma View.

No Crux, construímos as nossas telas em arquivos com extensão .xhtml, os quais podem representar uma view ou uma screen. Em seguida, utilizamos arquivos CSS para customizar a apresentação e, por fim, programamos os estados das telas através de controllers. Este modelo pode ser resumido pela **Figura 1** e será explicado em detalhes nos tópicos a seguir.

Screen e Views

A camada de visão é definida através dos artefatos screen e view. O artefato screen representa uma página HTML, enquanto uma view representa uma porção desta página, que geralmente a utilizamos para construir uma tela como, por exemplo, a tela de cadastro de usuários, a tela de listagem de itens do estoque, etc. Cada tela dessa é especificada através de um arquivo XHTML, sendo este uma estrutura HTML convertida para o modelo XML, o que nos permite inserirmos também informações sobre como o Crux vai trabalhar com cada componente da interface.

Por falar em componentes (ou widgets, como os chamamos no contexto do GWT), é importante mencionar que o Crux apoia a larga utilização destes para a composição das telas, uma vez que isto acelera e deixa o desenvolvimento mais robusto. Deste modo, quando observamos uma aplicação construída em Crux,

é comum notarmos, por exemplo, em uma tela de cadastro de informações, um componente de formulário, um componente de tabela, outro de popup, e assim por diante.

A diferença deste cenário para outras aplicações é que no Crux estes componentes são como caixas pretas, isto é, a semântica deles é mais valorizada do que a estrutura HTML que eles geram. Por exemplo, no GWT temos um componente de painel flutuante (**FlowPanel**) que (na renderização apresentada no browser) será uma simples `div`, porém, o painel é mais do que isto, ele possui uma semântica própria, podendo ter tipos de propriedades e eventos específicos. Isto colabora para construirmos páginas que valorizam a semântica em detrimento da simples disposição dos elementos HTML.

Para compor nossas telas, além dos componentes do GWT, o Crux possui suas próprias bibliotecas de componentes, que podem ser vistas no showcase do framework (vide seção **Links**). Elas se dividem em duas: a denominada **widgets**, mais direcionada para browsers antigos (como o Internet Explorer 8 e 9 ou o Chrome abaixo da versão 30); e **smartfaces**, voltada para os browsers mais modernos, com ampla utilização de CSS3 e HTML5.

Cabe destacar que o framework também permite a utilização de outras bibliotecas de componentes JavaScript, como os da biblioteca **jQuery UI**. No entanto, como no Crux desenvolvemos em Java e, portanto, a interface de acesso aos componentes também é implementada em Java, caso queiramos nos comunicar diretamente com um código JavaScript, devemos utilizar o recurso de funções nativas do GWT, conhecido como **JSNI**. Através dele, escrevemos o contrato de um método em Java, mas em seu corpo podemos inserir código JavaScript.

Note que isto pode gerar um trabalho adicional, uma vez que esta conversão deve ser realizada para cada funcionalidade que desejarmos acessar do componente JavaScript. Para evitar esse trabalho, em sua próxima versão o Crux terá suporte nativo a **Web Components**, solução que reúne um conjunto de padrões (regidos por uma especificação da W3C e atualmente produzidos pelo Google) que permite ao desenvolvedor reutilizar componentes construídos por terceiros de forma mais facilitada, bastando invocar as suas interfaces diretamente no código da página HTML (ou XHTML, no caso do Crux).

Ao contrário do que estamos acostumados, estes componentes representam mais do que simples trechos de código JavaScript, sendo compostos também por elementos HTML, CSS e, inclusive, um DOM próprio, denominado **Shadow DOM**. Estes componentes são uma das apostas do Google em se tratando do futuro das aplicações web e vêm sendo largamente dis-

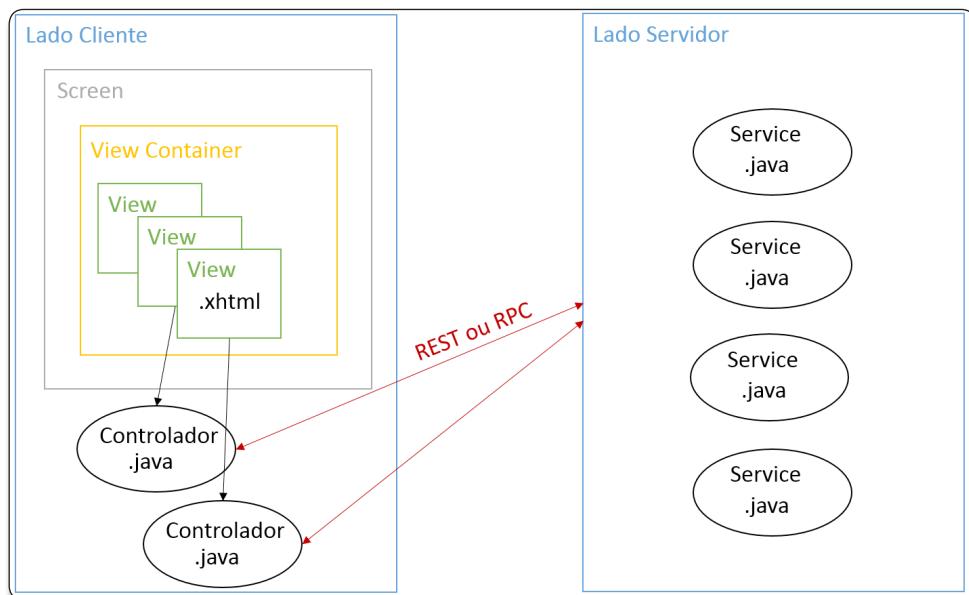


Figura 1. Resumo dos componentes envolvidos na arquitetura do Crux Framework

cutidos no contexto das aplicações GWT, principalmente nas conferências *GWT.create*.

Retornando ao conceito de screen, caso desejarmos construir uma página para toda a aplicação (lembre-se do *Single Page Application*), utilizaremos apenas um artefato deste tipo, o qual tem como extensão `.crux.xml`. Como exemplo de screen, observe o código apresentado na **Listagem 1**.

Listagem 1. Exemplo de código de uma Screen (index.crux.xml).

```

01 <!DOCTYPE html>
02 <html>
03   xmlns="http://www.w3.org/1999/xhtml"
04   xmlns:c="http://www.crxframework.org/crx"
05   xmlns:faces="http://www.crxframework.org/crx/smarts-faces"
06   xmlns:gwt="http://www.crxframework.org/crx/gwt">
07
08   <head>
09     <title>Crux Framework</title>
10   </head>
11
12   <body>
13     <script type="text/javascript" src="..sample-crx/sample-crx.nocache.js">
14       </script>
15
16     <c:screen
17       smallViewport="user-scalable=no, width=320"
18       largeViewport="user-scalable=no, width=device-width, height=device-height"
19       useResource="smartsFacesResources"/>
20
21     <faces:simpleViewContainer id="views">
22       <faces:view name="main"/>
23     </faces:simpleViewContainer>
24   </body>
25 </html>

```

Desenvolva aplicações web escaláveis com Crux Framework

Neste exemplo, podemos notar:

- **Linha 13:** Inclusão do arquivo `sample-crux.nocache.js`. Este é o script que efetivamente carrega o módulo GWT na página. Tal JavaScript é produzido pela compilação do GWT;
- **Linha 15:** Uso da tag `<cc:screen>`, que define as configurações da screen. No exemplo, ela está informando os *ViewPorts* da tela, recurso que determina as configurações de renderização e comportamento da nossa aplicação (como o tamanho da página e a forma como o zoom será processado ao pinçar a tela do dispositivo). Além disso, o atributo `useResource` determina os recursos que serão utilizados (como arquivos CSS e imagens);
- **Linha 20:** Uso da tag `<crux:simpleViewContainer>`. Esta representa o container de views desta screen, local onde as views serão carregadas. No exemplo, estamos utilizando um *Simple View Container*, que é responsável por exibir apenas uma view de cada vez, e a primeira a ser exibida possui o nome “**main**”. Caso desejarmos, no decorrer do código da aplicação, podemos acessar este container e alterar esta view para outras. Por se tratar de um container simples, a view anterior será descarregada e dará lugar a outra.

Além do *Simple View Container*, existem outras opções, cada qual responsável por um comportamento diferente. Outros dois deles, e também bastante utilizados são:

- **Multiple Views Container:** este container gerencia várias views e as exibe diretamente na screen, sendo uma de cada vez. A diferença deste para o *Simple View Container* é que este não precisa descarregar uma view para carregar outra;
- **Dialog View Container:** este container é responsável por exibir uma view através de uma caixa de diálogo flutuante.

Dentro de um arquivo de view podemos inserir os componentes das bibliotecas widgets e smartfaces, os componentes nativos do GWT ou quaisquer outros que estiverem integrados ao Crux, através da utilização do JSNI. Como exemplo de view, observe o código apresentado na **Listagem 2**.

Neste trecho de código, podemos notar:

- **Linha 7:** Utilização da tag `useController`, que determina qual será o controller da view. No próximo tópico falaremos sobre os controllers;
- **Linhas 10 a 13:** Utilização das tags `<gwt:flowPanel>`, `<gwt:textBox>` e `<faces:button>`, componentes responsáveis por compor a nossa tela;
- **Linha 12:** Definição do botão `okButton`. Note que ele possui um atributo chamado `onSelect` que indica a ação que será executada ao ser acionado. No caso do exemplo, ao clicar (ou tocar) sobre este botão, será acionado o método `sayOk()` do controller `mainController`.

Controllers

Os controllers são classes escritas em Java responsáveis por interagir com os componentes da tela. A partir destas classes, também podemos invocar as chamadas a funções do servidor e

realizar as demais lógicas que seriam comumente definidas em um código JavaScript.

Um dos grandes benefícios de se utilizar o Crux é que a camada controller está localizada no cliente e não no servidor, como é usual nos demais frameworks. Esta estratégia garante um desempenho consideravelmente superior às arquiteturas convencionais, uma vez que após o carregamento da página, na rede somente são trafegados os dados da aplicação e não mais páginas HTML inteiras. Como exemplo de um controller, observe o código apresentado na **Listagem 3**.

Listagem 2. Exemplo de código de uma View (`main.view.xml`)

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <v:view
03   xmlns="http://www.w3.org/1999/xhtml"
04   xmlns:v="http://www.cruxframework.org/view"
05   xmlns:gwt="http://www.cruxframework.org/crx/gwt"
06   xmlns:faces="http://www.cruxframework.org/crx/smarts-faces"
07   useController="mainController"
08   onLoad="mainController.onLoad">
09
10   <gwt:flowPanel id="mainPanel">
11     <gwt:textBox id="nameTextBox" />
12     <faces:button id="okButton" text="Go!" onSelect="mainController.sayOk()"/>
13   </gwt:flowPanel>
14
15 </v:view>
```

Listagem 3. Exemplo de código de um Controller.

```
01 package br.com.globo.sim.sim.client;
02 import org.cruxframework.crux.core.client.controller.Controller;
03 import org.cruxframework.crux.core.client.controller.Expose;
04 import org.cruxframework.crux.core.client.ioc.Inject;
05 import org.cruxframework.crux.core.client.screen.views.BindView;
06 import org.cruxframework.crux.core.client.screen.views.WidgetAccessor;
07 import org.cruxframework.crux.widgets.client.dialog.FlatMessageBox;
08 import org.cruxframework.crux.widgets.client.dialog.FlatMessageBox
09 .MessageType;
09 import com.google.gwt.user.client.ui.TextBox;
10
11 @Controller("mainController")
12 public class MainController
13 {
14   @Inject
15   public MainView view;
16
17   @Expose
18   public void onLoad()
19   {
20     //do something
21   }
22
23   @Expose
24   public void sayOk()
25   {
26     FlatMessageBox.show("OK :" + view.nameTextBox().getText(),
27     MessageType.INFO);
27   }
28
29   @BindView("main")
30   public static interface MainView extends WidgetAccessor
31   {
32     TextBox nameTextBox();
33   }
34}
```

Neste trecho de código, podemos notar:

- **Linha 11:** A anotação `@Controller`, que define que esta classe representa um controller de nome `mainController`.
- **Linha 14:** A utilização de IoC através da anotação `@Inject`. Este recurso permite ao desenvolvedor injetar nos controllers objetos geridos por um container de injeção, que é responsável pelo controle de instanciação de cada um destes objetos;
- **Linha 15:** Injetamos automaticamente um objeto do tipo `WidgetAcessor` (declarado na linha 30). Este objeto será utilizado na linha 26 para acessar o método `nameTextBox()`;
- **Linha 17:** A utilização da anotação `@Expose` indica que este método será invocado por um arquivo de extensão XML, como uma view ou uma screen;
- **Linha 32:** Mostra a amarração (bind) do método `nameTextBox()` da interface `MainView` com o componente da view `main.view.xml` cujo id é `nameTextBox`. Este elo é realizado observando que o id do componente corresponde ao nome do método da interface. Note que neste exemplo este é o único método declarado, mas podemos criar outros, caso queiramos acessar os demais componentes presentes em nossa view;
- **Linha 29:** A utilização da anotação `@BindView` indica que os componentes declarados em um `WidgetAcessor` estão amarrados com a view `main`.

Services

Services são classes Java responsáveis pela lógica de negócios. Elas constituem parte do *back-end* da aplicação e processam os dados no lado do servidor. No Crux, podemos enviar dados do cliente para estas classes de duas maneiras: utilizando RPC (*Remote Procedure Call*) ou REST (*Representational State Transfer*). Ambas têm suas aplicações e contextos para adoção, mas quando trabalhamos com uma aplicação *rich client* (foco principal do Crux), recomendamos também construir-la de forma *stateless*, isto é, não mantemos o estado dos dados no lado do servidor, apenas no lado cliente. Deste modo, informações como o estado do formulário da página, os dados da seção do usuário, etc., são todos geridos e mantidos no cliente. Em termos gerais, isto quer dizer que o servidor desconhece o cliente. Como este tipo de solução se enquadra exatamente com o que a arquitetura RESTful propõe, a adoção do REST para realizarmos chamadas ao servidor é aconselhada.

Dado que escolhemos a maneira como vamos nos comunicar com o servidor, o Crux suporta quaisquer tecnologias que o arquiteto do sistema optar para aderir ao contexto e necessidade da aplicação. Por exemplo, o servidor pode utilizar Java (com CDI, Spring, etc.), Node.js ou mesmo pode nem haver um servidor, caso desejemos uma aplicação off-line, com todos os seus recursos no próprio cliente.

Construindo a primeira aplicação com Crux

Agora que já sabemos por onde estamos caminhando, vamos desenvolver um “Hello World” em Crux que vai servir de base para exercitarmos alguns conceitos mencionados até aqui e nos

familiarizarmos um pouco mais com o framework. Para isso, execute os passos a seguir:

1. Baixe o arquivo referente ao Archetype Maven de criação de projetos Crux, cujo endereço está disponível na seção **Links**;
2. No menu do Eclipse, localize e selecione a opção `File > New > Project`;
3. Em seguida, clique na opção `Maven Project`;
4. Na janela que será aberta, de nome `New Maven Project`, clique no botão `Next`;
5. Clique no botão `Configure`, localizado no canto superior direito, e na tela seguinte, no botão `Add Local Catalog`;
6. Selecione o arquivo XML previamente salvo, após acionar o botão `Browse`;
7. No campo de descrição, digite um nome para o catálogo como, por exemplo, *Catálogo Crux*;
8. Clique em `Ok` nas duas telas seguintes, retornando à tela `New Maven Project`;
9. Selecione a opção `crux-helloworld-war`;
10. Na próxima janela, informe, respectivamente, o pacote e o nome da aplicação como, por exemplo: **Group Id** = `meus.testes.crux` e **Artifact Id** = `HelloCrux`;
11. Altere as seguintes propriedades do archetype para informar, respectivamente, o nome e o nome curto do módulo da aplicação, por exemplo: **module-name** = `HelloCrux`; **module-short-name** = `hellocrux`. A **Figura 2** demonstra os valores no formulário;

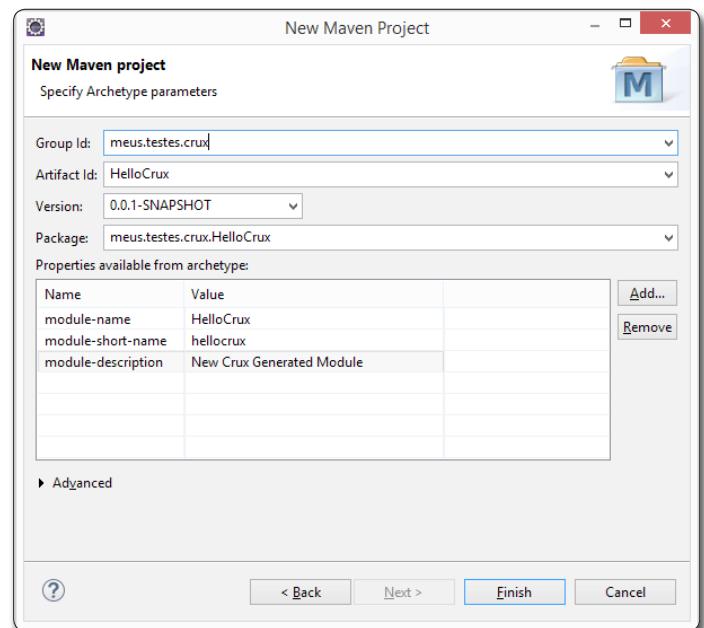


Figura 2. Exemplo de preenchimento dos campos na criação de um projeto Crux com o Maven

12. Pressione o botão `Finish` e pronto! Um projeto Crux será criado e estará pronto para você começar a explorar os recursos. Na **Figura 3** podemos ver a estrutura do projeto;
13. Localize os arquivos `Start Jetty.launch` e `Start CodeServer.launch` na pasta raiz do projeto. Para iniciar o servidor de aplicação, clique sobre o primeiro arquivo com o botão direito do mouse, selecione a opção

Desenvolva aplicações web escaláveis com Crux Framework

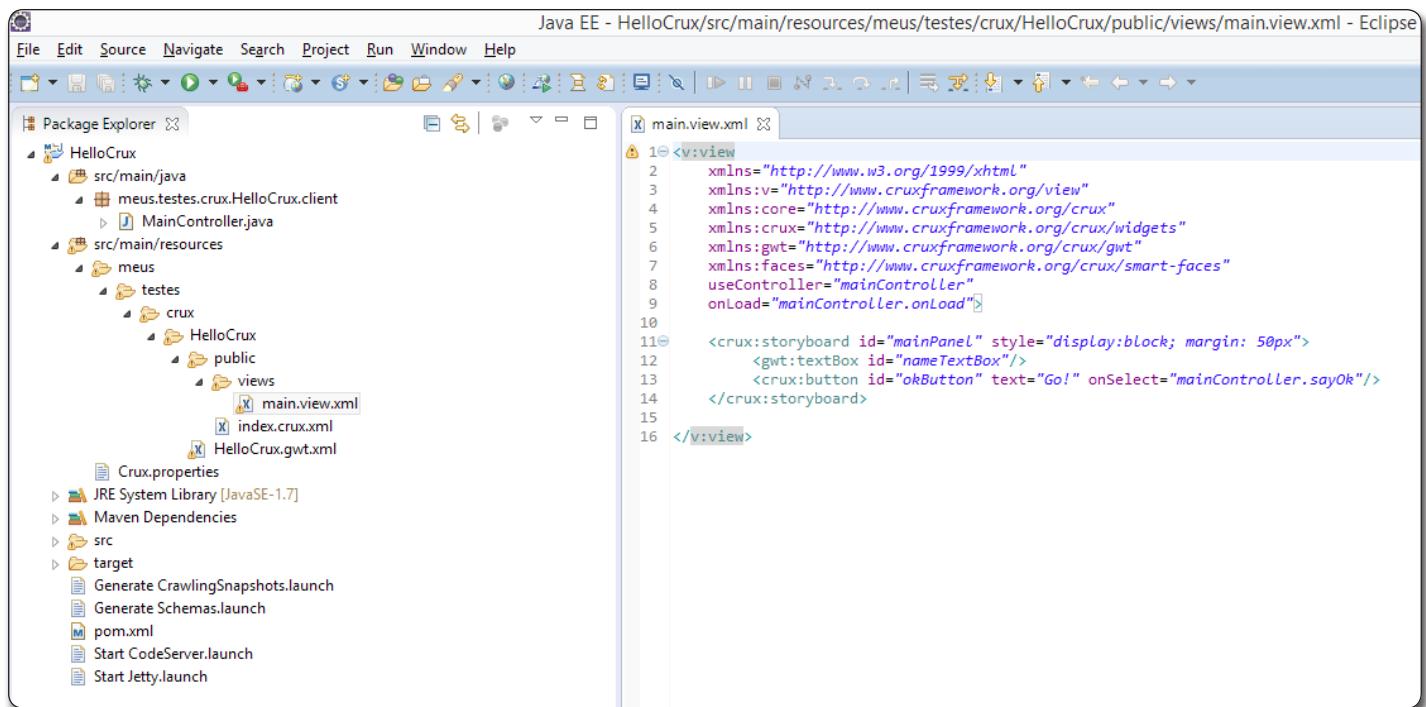


Figura 3. Projeto exemplo criado através do passo a passo

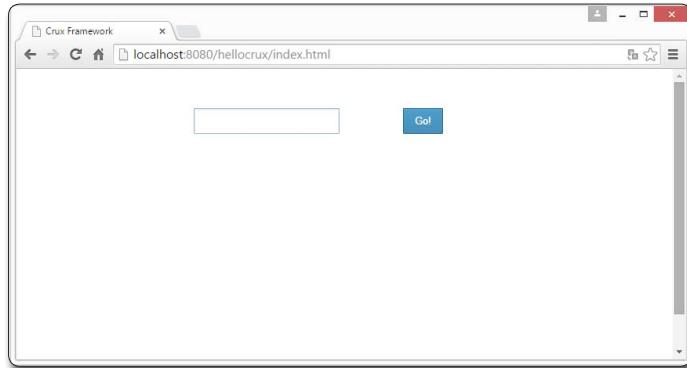


Figura 4. Aplicação exemplo rodando em Crux

Listagem 4. Código de um Controller.

```
01 package br.com.globo.sim.sim.client;
02 import org.cruxframework.crux.core.client.controller.Controller;
03 ...
04 import com.google.gwt.user.client.ui.TextBox;
05
06 @Controller("mainController")
07 public class MainController
08 {
09     @Expose
10     public void onLoad()
11     {
12         Window.alert("Welcome!");
13     }
14 ...
15 ...
16 }
```

Run As e, em seguida, a opção em Start Jetty. De forma semelhante, para iniciar o servidor de código estático (responsável por prover ao browser os arquivos estáticos como HTML, JavaScript, CSS, etc.), clique sobre o segundo arquivo com o botão direito do mouse, selecione a opção Run As e, depois, a opção Start CodeServer;

14. Quando ambos os processos completarem o processo de inicialização (observe as saídas geradas na aba console do Eclipse), a aplicação já estará disponível para ser acessada pelo endereço <http://localhost:8080/hellocrux/index.html>. Portanto, acesse-o através do Chrome, por exemplo, e veja uma tela semelhante à apresentada na Figura 4;

A partir daí nossa aplicação (que exibe em um alerta o conteúdo digitado no TextBox, após o botão Go! ser acionado) estará pronta para ter seu código fonte modificado à vontade pelo desenvolvedor. Graças ao recurso de deploy automático do Crux, todas as alterações serão compiladas pelo servidor de código estático e o resultado será exibido no browser de forma automática.

Para demonstrar este processo, vamos adicionar no método **onLoad()** da classe **MainController** o código **Window.alert("Welcome!")**, que faz com que uma mensagem de alerta seja exibida quando a view **main** for carregada no container. Veja a linha 12 da Listagem 4.

Após esta modificação, salve o arquivo e logo em seguida observe o processo de compilação da aplicação em seu navegador, como expõe a Figura 5. Durante este processo, o código Java será transformado em código JavaScript.

Caso esteja interessado em acompanhar esta etapa mais atentamente, você pode observar a aba **Console** do Eclipse, onde é apresentado

um log de todo o processo. Ao final da compilação, a página será recarregada e o resultado final exibido, conforme a **Figura 6**.

Após esta breve introdução ao Crux, convidamos o leitor a conhecer mais sobre este interessante framework. A adoção de uma solução que viabiliza uma fácil manutenção do código e, ao mesmo tempo, consegue trazer um grande poder de abstração é de suma importância para trabalharmos em grandes aplicações corporativas.

Para que o leitor possa conhecer um pouco mais sobre o framework, recomendamos que realize os passos do Hello World em Crux e explore o código fonte da aplicação para ver como é fácil e até divertido construir projetos com esta tecnologia. Se você quiser ir mais adiante, sugerimos explorar recursos mais avançados do framework, como a comunicação com o servidor através de requisições REST, o acesso às APIs para persistência de dados no cliente, a utilização do cache de páginas para a construção de aplicações off-line, entre outros, e ao final do processo, inspecione a aplicação diretamente através do browser, para perceber a otimização e o tamanho reduzido do código JavaScript, bem como a pequena quantidade de dados que é trafegada durante a comunicação entre cliente e servidor.

Por fim, também convidamos o leitor a fazer parte da nossa comunidade, reportando problemas, sugerindo novas features, criticando ou elogiando. Participar é muito importante! Somente assim, podemos crescer cada vez mais e construir um produto de qualidade, pronto para abranger todos os desafios que a web constantemente nos proporciona.

Autor



Samuel Almeida Cardoso

samuel@cruxframework.org

Bacharel em Ciência da Computação pela UFMG, Pós-graduado em Arquitetura de Sistemas Distribuídos, Mestrando em Computação e apaixonado por tecnologia, atuante no mercado de trabalho há mais de 10 anos.



Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!

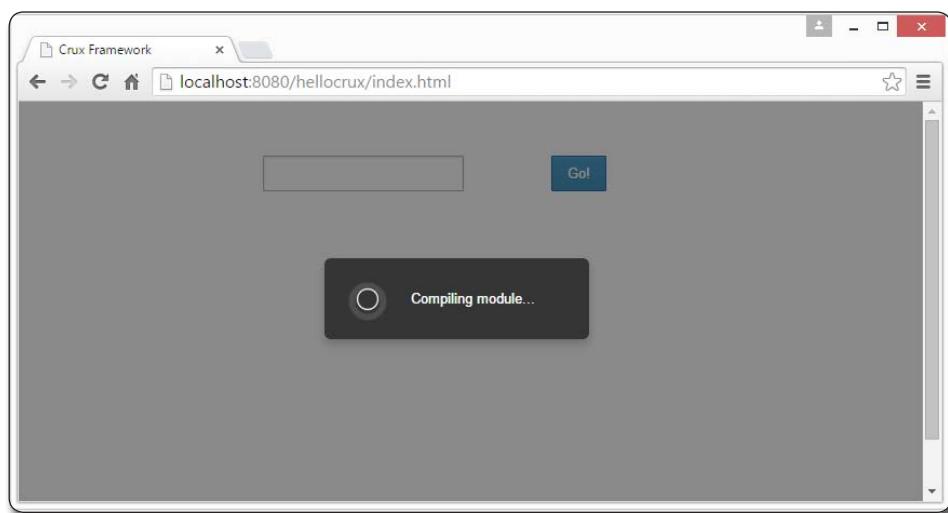


Figura 5. Aplicação sendo compilada automaticamente após modificação no código

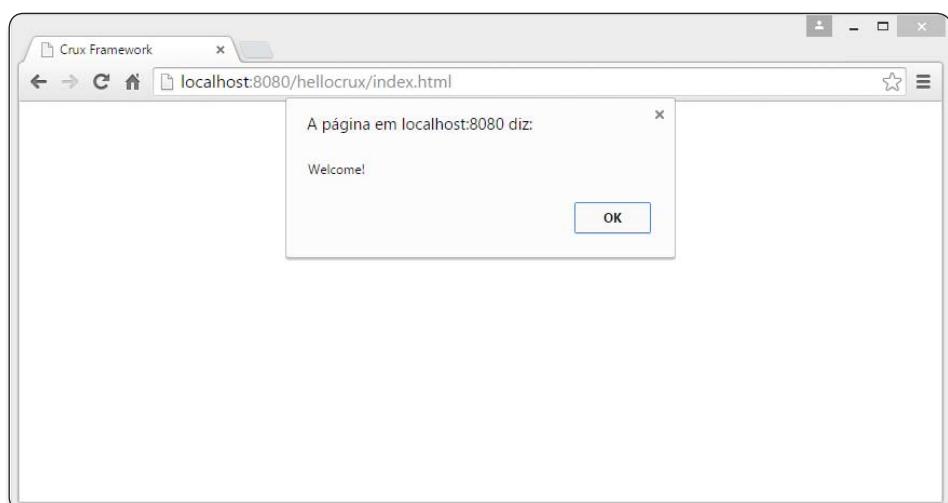


Figura 6. Resultado da compilação: mensagem de alerta exibida ao carregar view main

Links:

Site do GWT Web Toolkit.

<http://www.gwtproject.org>

Site oficial do Crux Framework.

<http://www.cruxframework.org>

Documentação das funcionalidades do Crux Framework.

<http://www.cruxframework.org/#!view=manual>

Showcase dos componentes do Crux Framework.

<http://showcase.cruxframework.org>

Blog da equipe de desenvolvimento do Crux Framework.

<http://blog.cruxframework.org>

Archetype Maven para criação de um projeto Crux Framework.

<http://repo1.maven.org/maven2/org/cruxframework/crux-archetypes/5.3.1/crux-archetypes-5.3.1-catalog.xml>

Catálogo do Crux.

<http://www.cruxframework.org/downloads/crux-archetypes-5-catalog.xml>

Morphia: Veja como simplificar o desenvolvimento de aplicações NoSQL

Conheça as principais características do banco de dados NoSQL MongoDB e como implementar aplicações que irão simplificar o seu desenvolvimento usando o Morphia

O Modelo de Dados Relacional é uma ferramenta de modelagem de dados que desde a década de 70 vem sendo adotado pelas empresas para resolver todo tipo de problema de armazenamento e manipulação de dados. A principal característica desse modelo está na responsabilidade de manter a integridade dos dados através da utilização de chaves primárias (*primary keys*), chaves estrangeiras (*foreign keys*), tipos de dados (*varchar*, *char*, *number*, *blob*, etc.) e, principalmente, a necessidade de garantir as propriedades ACID, acrônimo de Atomicidade, Consistência, Isolamento e Durabilidade.

A implementação de modelos relacionais é feita através dos SGBDRs (Sistemas Gerenciadores de Bancos de Dados Relacionais), que passaram a ser utilizados em grande escala pelas empresas por oferecerem como principais recursos o controle de transações, a integridade dos dados, otimização de consultas e utilização da linguagem SQL como padrão para consulta e manipulação de dados.

No entanto, levando em consideração o cenário apresentado de aumento no volume de dados gerados e no número de usuários por aplicação, os SGBDRs têm se

Fique por dentro

Este artigo é útil para arquitetos e programadores que projetam/ implementam sistemas que manipulam grandes volumes de dados e necessitam de escalabilidade e performance mantendo a qualidade do código. Levando em consideração esse cenário, primeiramente o artigo irá apresentar as principais características dos Bancos de Dados NoSQL, em especial o MongoDB, e suas vantagens em comparação com os bancos de dados relacionais para alguns problemas específicos. Como principal objetivo do artigo, será ensinado como implementar aplicações Java com MongoDB utilizando o framework ODM (Object Document Mapping) Morphia, que se propõe a facilitar as operações de inserção, atualização, remoção e consulta de documentos em coleções de bases de dados MongoDB.

mostrado ineficientes quanto à necessidade de uma melhor performance e extremamente complexos no que se refere à configuração de escalabilidade em aplicações em ambientes clusterizados, sem citar os custos cobrados por parte dos fornecedores, que aumentam significativamente.

Outro problema presente em aplicações que fazem uso de bancos de dados tradicionais é a incompatibilidade entre o paradigma

relacional e o orientado a objetos. Nestes bancos os dados são organizados em uma estrutura de tabelas, enquanto a aplicação normalmente adota o paradigma orientado a objetos. Com isso, torna-se necessário realizar uma conversão para representar e mapear as informações do banco de dados na aplicação, e para tal tarefa, utilizamos frameworks ORM (*Object Relational Mapping*), dentre os quais destacamos JPA e Hibernate.

Como solução para os problemas citados, este artigo apresenta, primeiramente, os bancos de dados NoSQL. Em seguida, daremos mais espaço ao MongoDB, uma solução NoSQL indicada principalmente para aplicações que necessitam de alta performance e manipulam grandes volumes de dados. Relacionado ao MongoDB, apresentaremos também o Morphia, um framework ODM (*Object Document Mapping*) que é utilizado para facilitar a implementação de funcionalidades em aplicações Java como inserção, atualização, remoção e consulta a documentos no MongoDB.

Na prática, iremos abordar a implementação de um sistema exemplo, onde serão demonstrados os recursos do framework Morphia. Para isso, dado um modelo de domínio, inicialmente faremos o mapeamento das suas entidades para documentos do MongoDB utilizando as anotações da API do framework. Na sequência, serão implementadas as funcionalidades de conexão, inserção, atualização, remoção e buscas no banco de dados, e, por fim, as testaremos e validaremos através de testes de integração.

Bancos de dados NoSQL

Por conta das limitações apresentadas pelos bancos de dados tradicionais, projetistas e arquitetos iniciaram estudos para a criação de um tipo de banco de dados que fornecesse uma alternativa às soluções relacionais e apresentasse como principais características:

- Alta performance;
- Escalabilidade;
- Facilidade em manipular grandes volumes de dados;
- Simplicidade na criação de clusters.

O resultado desses estudos foi a criação dos bancos de dados denominados de NoSQL, ou *Not Only SQL*. Esse termo começou a ser utilizado a partir de uma reunião realizada no dia 11 de junho de 2009, em São Francisco, Estados Unidos, organizada pelo desenvolvedor Johan Oskarson, tendo como foco bancos de dados não relacionais, distribuídos e open source. Johan queria um nome para a reunião que fosse curto, fácil de lembrar e que se tornasse uma boa hashtag para o Twitter. Foi indicado então, através do canal #cassandra do IRC, que a reunião se chamassem “NoSQL”, sugestão feita por Eric Evans, desenvolvedor da Rack Space – sem ligação com Eric Evans criador do DDD.

Os Bancos de Dados NoSQL trazem como principais características a não utilização do modelo de dados relacional, facilidade na escalabilidade horizontal, código fonte aberto, fácil configuração em ambientes clusterizados, não possuem esquema (o que possibilita a adição de novos campos aos registros sem ter que fazer qualquer mudança na estrutura) e a não utilização da linguagem

SQL para consultas e manipulação de dados, apesar de algumas opções possuírem uma linguagem de consulta bem parecida, com o objetivo de facilitar o aprendizado, como é o caso da CQL, linguagem de consulta do Cassandra, solução NoSQL voltada para o armazenamento em famílias de colunas.

Podemos organizar o NoSQL em quatro categorias de tipos de armazenamento de dados, a saber: Chave-Valor, Documentos, Armazenamento em famílias de colunas e grafos. O foco desse artigo está na categoria Documentos, mirando para isso o banco de dados MongoDB.

Bancos de dados de documentos e o MongoDB

Os bancos de dados orientados a documentos são opções que possibilitam que as informações, os dados, sejam armazenados em formatos como JSON, BSON ou XML. Outro diferencial desse tipo de armazenamento de dados é permitir a criação de novos atributos nas “tabelas” sem a necessidade de qualquer alteração nos documentos existentes, o que é viabilizado pela não existência de um schema, elemento comum e de grande importância para os bancos de dados relacionais.

Atualmente, as principais opções de bancos de dados que armazenam informações no formato de documentos são: MongoDB, CouchDB, Terrastore, OrientDB e RavenDB. Implementado com a linguagem C++, o MongoDB, traz uma boa documentação e é facilmente instalado nos sistemas operacionais Windows, Linux e Mac OS, disponibilizando, ainda, vários drivers para diversas linguagens de programação.

É importante citar também que o conceito de transação no MongoDB é executado apenas no nível de documento. Assim, não permite que sejam executados inserts, updates e deletes para vários documentos e, ao final, se decida se deseja manter ou desfazer as operações através de comandos como *commit* e *rollback*, mecanismos implementados por bancos relacionais.

A configuração da replicação de dados e sharding são mais dois recursos importantes que podemos destacar no MongoDB. Estes podem ser utilizados para garantir que o banco de dados tenha um bom desempenho quando ocorrer um aumento significativo no número de usuários com acesso para leitura/escrita. O conceito de sharding possibilita escalar horizontalmente o banco de dados através da adição de várias máquinas a um cluster, de modo que o banco de dados consiga aprimorar sua performance.

A utilização do MongoDB é indicada para sistemas com as seguintes características: não precisam de controle transacional, necessitam de alta escalabilidade e disponibilidade, e lidam com um grande volume de dados. Podemos citar, como casos reais apropriados para sua adoção: sistemas para controle de auditoria e registro de eventos, blogs e aplicações de comércio eletrônico.

Para realizar a instalação do MongoDB, primeiramente é necessário baixar a versão para o seu sistema operacional na área de downloads do site do MongoDB (veja a seção [Links](#)). Logo após, descompacte o arquivo baixado em uma pasta de sua preferência. Para começar a utilizar o banco, acesse o diretório `\bin` e execute o arquivo `mongod.exe`. Agora, ainda no diretório

\bin, execute o comando `mongo.exe`, que abrirá o cliente shell, onde será possível executar comandos para a criação de bancos de dados, realização de consultas, inserções, atualizações e remoções de dados.

Por default, ao abrir o shell de comandos administrativos, o MongoDB se conectará ao banco de dados `test`. Para acessar outro banco de dados, execute o comando `use nomeDoBancoDeDados`. A seguir, apresentamos os principais comandos que podem ser utilizados:

- Acessar um banco de dados e caso não exista, o mesmo será criado: `use nomeDoBancoDeDados;`
- Listar os bancos de dados: `show dbs;`
- Listar as coleções de um banco de dados: `show collections;`
- Criar uma coleção em um banco de dados: `db.createCollection ("TECNOLOGIA");`
- Inserir um documento em uma coleção: `db.TECNOLOGIA.insert ({nome:"C++"},{tipo:"LINGUAGEM_PROGRAMACAO"});`
- Atualizar um documento em uma coleção: `db.TECNOLOGIA .update({nome:"C++"},{nome:"C"},{tipo:"LINGUAGEM_PROGRAMACAO"});`
- Consultar documentos em uma coleção. Ao executar o comando a seguir, o resultado listará os documentos da coleção TECNOLOGIA armazenados no banco de dados `test`: `db.TECNOLOGIA find();` E ao executar o comando a seguir, o resultado listará os documentos do tipo LINGUAGEM_PROGRAMACAO da coleção TECNOLOGIA: `db.TECNOLOGIA.find({tipo:"LINGUAGEM_PROGRAMACAO"});`
- Remover documentos em uma coleção: `db.TECNOLOGIA .remove({tipo:"LINGUAGEM_PROGRAMACAO"});`
- Deletar uma coleção: `db.TECNOLOGIA.drop();`
- Deletar um banco de dados: `db.dropDatabase();`

Veja nas **Listagens 1** e **2** exemplos de documentos armazenados em coleções de um banco de dados no MongoDB.

Listagem 1. Coleção de documentos da entidade TECNOLOGIA.

```
{  
  "_id": ObjectId("123456789"), "nome": "JAVA",  
  "tipo": "LINGUAGEM_PROGRAMACAO"  
}  
  
{  
  "_id": ObjectId("987654321"), "nome": "MONGODB",  
  "tipo": "BANCO_DADOS"  
}
```

Listagem 2. Coleção de documentos da entidade PROFISSIONAL, que possui uma FORMACAO e pode ter várias FUNCOES.

```
{  
  "_id": ObjectId("123456789"), "nome": "Paulo Henrique da Silva",  
  "funcoes": ["ARQUITETO", "DESENVOLVEDOR"], "formacao":  
  {"nomelInstituicao": "UFMG",  
   "curso": "Ciencia da Computacao"  
   "duracao": "4 anos"}  
}
```

Morphia

O Morphia é um framework que tem como principal objetivo facilitar a interação entre a aplicação e o MongoDB, simplificando a implementação das operações de conexão, inserção, atualização, remoção e consulta de documentos em coleções no banco de dados.

Através de sua API, o Morphia permite a realização do mapeamento ODM (*Object Document Mapping*), em que dado um modelo de domínio, torna-se possível o mapeamento das entidades Java desse modelo para um documento do MongoDB. O framework possui uma proposta semelhante à dos frameworks ORMs (*Object Relational Mapping*) JPA e Hibernate, responsáveis pelo mapeamento de uma classe no modelo de domínio para uma tabela em um banco de dados.

Também de modo semelhante, as operações de inserção, atualização, remoção e buscas de documentos são realizadas através da manipulação das entidades Java mapeadas com as anotações da API do Morphia. Para essa manipulação, o Morphia disponibiliza a interface `org.mongodb.morphia.Datastore` e a classe `org.mongodb.morphia.Morphia`, que é responsável pela criação do `Datastore` dada uma instância do driver do MongoDB e um banco de dados, como veremos mais à frente.

A seguir são apresentadas as principais anotações do Morphia para realizar o mapeamento ODM. A utilização de todas elas pode ser verificada na parte prática deste artigo. Conheçamos as anotações:

- **@Entity:** Anotação construída no pacote `org.mongodb.morphia.annotations`, declara que uma classe será salva como um documento no banco de dados do MongoDB. O framework utiliza como padrão o nome da classe para criar a coleção;
- **@Id:** Sinaliza qual campo da entidade será utilizado na persistência da coleção como ID do documento;
- **@Transient:** Indica ao Morphia para não tornar o atributo persistente;
- **@PostPersist:** O método anotado com `@PostPersist` será chamado pelo Morphia logo após a entidade ser persistida;
- **@PrePersist:** O método anotado com `@PrePersist` será chamado pelo Morphia antes da entidade ser persistida;
- **@Embedded:** Permite que a lista ou atributo anotado com essa anotação seja tratado como um agregado, ou seja, fará parte da estrutura do documento;
- **@Reference:** Armazena um id como referência para outro documento de outra coleção;
- **@Indexed:** Aplica um índice ao campo que estiver anotado. O índice ajudará as consultas a ter um melhor desempenho e também poderá garantir que não existam registros duplicados através da propriedade `unique`;

Por fim, vale ressaltar que para todo atributo de entidade que não estiver anotado, o Morphia tentará armazená-lo na coleção como um campo persistente.

Sistema de controle de projetos e alocação de profissionais

A fim de apresentar uma aplicação Java que utiliza o banco de dados MongoDB e a API do Morphia, será criado um cenário

fictício referente a uma necessidade de software de uma empresa do segmento de planos de saúde, de nome GTY.

Esta empresa possui um setor de TI com gerentes que acumulam as seguintes responsabilidades: cadastrar tecnologias (Linguagens de Programação ou Banco de Dados) utilizadas pelo setor, cadastrar profissionais e cadastrar os projetos a serem desenvolvidos internamente.

Neste cenário, caso exista a necessidade de implementação de um software pelo setor de TI, um gerente realizará o cadastro do projeto informando o nome, uma breve descrição, a data prevista de início, as tecnologias utilizadas e os profissionais responsáveis pelo desenvolvimento. Para controle e auditoria dos projetos cadastrados, será preciso logar toda inclusão ou alteração que for realizada nas entidades em algum documento no banco de dados, gravando as seguintes informações: entidade que sofreu a alteração, data e hora.

A Figura 1 apresenta o modelo de domínio referente ao sistema de controle de projetos e alocação de profissionais da GTY. A partir disso, para demonstrar o Morphia na prática, serão mapeadas as principais entidades usando sua API e, logo em seguida, serão criados os serviços de infraestrutura para exemplificar as funcionalidades de inserção, atualização, remoção e consulta a documentos nas coleções de um banco MongoDB.

Implementação do sistema

Para a implementação do sistema, utilizaremos a IDE Eclipse e o plugin m2e, que simplificarão nossas tarefas de codificação e gerência do Maven, que será utilizado para controlar as dependências do projeto, através do arquivo *pom.xml* (vide Listagem 3). O sistema a ser construído é um projeto Java simples, e como nosso foco está na camada de acesso a dados, não mostraremos como construir a camada de apresentação.

Pensando em manter as boas práticas, o projeto terá suas classes organizadas em pacotes. Dentre eles, o pacote **br.com.gerenciamentoprojeto.entidades** será responsável por conter todas as entidades, que aqui foram mapeadas referenciando documentos de um banco de dados no MongoDB. Esta relação é estabelecida através da anotação **@Entity** da API do Morphia. Veja as Listagens 4, 5, 6 e 7, que apresentam, respectivamente, o código das entidades **Tecnologia**, **Profissional**, **Projeto** e **LogManutencao**. Além da anotação **@Entity**, o leitor mais atento notará nestas listagens outros recursos da API do Morphia, a saber:

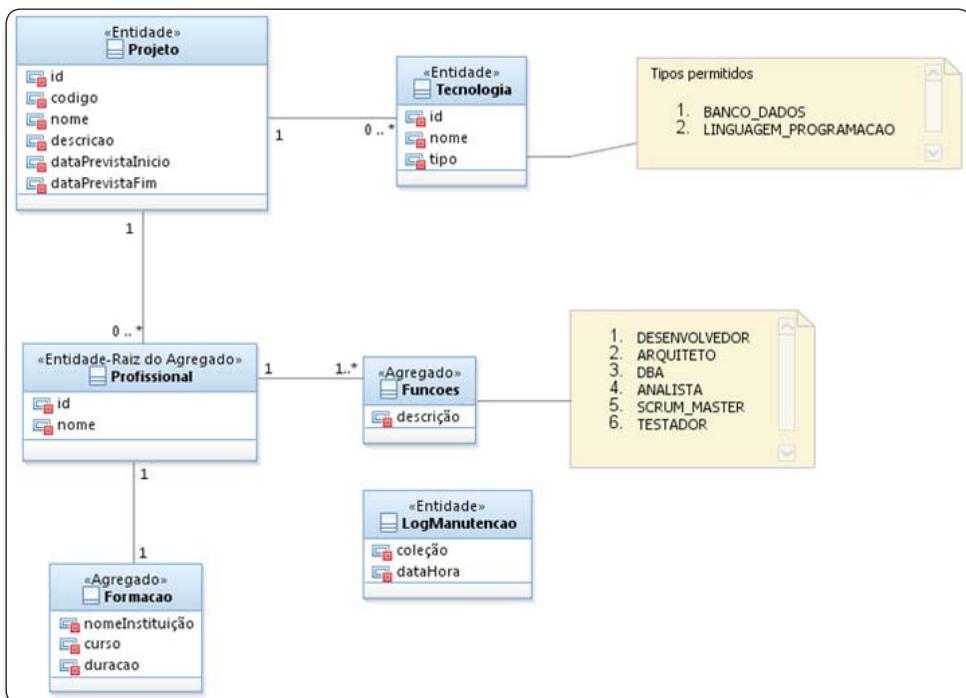


Figura 1. Modelo de domínio do sistema de controle de projetos e alocação de profissionais

Listagem 3. Arquivo *pom.xml* utilizado pelo Maven para gerenciar dependências e configurações da aplicação.

```

<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.gerenciamentoprojeto</groupId>
  <artifactId>controleProjeto</artifactId>
  <version>1.0</version>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.3.2</version>
          <configuration>
            <source>1.6</source>
            <target>1.6</target>
            <compilerArgument></compilerArgument>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
    <dependency>
      <groupId>org.mongodb.morphia</groupId>
      <artifactId>morphia</artifactId>
      <version>1.0.1</version>
    </dependency>
  </dependencies>
</project>
  
```

Morphia: Veja como simplificar o desenvolvimento de aplicações NoSQL

- **@Entity:** As classes **Tecnologia**, **Profissional**, **Projeto** e **LogManutencao** receberam esta anotação e assim seus objetos podem ser persistidos como um documento no MongoDB. A propriedade **value** permite escolher o nome da coleção que será criada no banco de dados. Já a propriedade **noClassnameStored** possibilita definir se você deseja ou não armazenar o nome da classe anotada na coleção do banco;
- **@Indexed:** Anotação utilizada nas entidades **Tecnologia** e **Projeto**, mais precisamente sobre os campos **nome**. A propriedade **value** permite especificar como será a ordenação dos documentos, se ascendente ou descendente. A propriedade **name**, por sua vez, permite configurar o nome do índice. Por último, **unique** evita que documentos duplicados sejam armazenados na coleção;
- **@Transient:** Declarada no atributo exemplo **campoNaoPersistente** da entidade **Tecnologia**. Por isso, não será armazenado na coleção do MongoDB;
- **@PostPersist:** Declarada nas entidades **Tecnologia**, **Profissional** e **Projeto**. Os métodos assim anotados serão executados logo após o documento ser armazenado na coleção;
- **@Embedded:** O atributo com esta anotação (vide **Listagem 5**) fará parte da estrutura do documento como um anexo, possibilitando que se tenha outro documento ou uma coleção dentro de um documento;

Listagem 4. Código da entidade Tecnologia.

```
package br.com.gerenciamentoprojeto.entidade;

//importando dependências do Morphia
import org.bson.types.ObjectId;
import org.mongodb.morphia.annotations.Entity;
import org.mongodb.morphia.annotations.Id;
import org.mongodb.morphia.annotations.PostPersist;
import org.mongodb.morphia.annotations.Transient;
import org.mongodb.morphia.annotations.Indexed;
import org.bson.types.ObjectId;

@Entity(value = "TECNOLOGIA", noClassnameStored = true)
public class Tecnologia implements Serializable {

    @Id
    private ObjectId id;

    @Indexed(value = IndexDirection.ASC, name = "nome", unique = true)
    private String nome;

    @Transient
    private String campoNaoPersistente;

    private String tipo;
    GeradorDeLogManutencao geradorDeLogManutencao = new GeradorDeLogManutencao();

    @PostPersist
    public void postPersist() {
        geradorDeLogManutencao.gerarLogManutencao(this.getClass().getName());
    }
}
```

- **@Reference:** Utilizada na entidade **Projeto**, esta anotação possibilita que o atributo armazene uma referência para o id de um documento de outra coleção, de modo similar a uma Foreign Key no modelo de dados relacional.

Listagem 5. Mapeamento da entidade Profissional para o documento PROFISSIONAL.

```
package br.com.gerenciamentoprojeto.entidade;

@Entity(value = "PROFISSIONAL", noClassnameStored = true)
public class Profissional implements Serializable {

    @Id
    private ObjectId id;
    private String nome;

    @Embedded
    private List<String> funcoes = new ArrayList<String>();

    @Embedded
    private Formacao formacao;
    GeradorDeLogManutencao geradorDeLogManutencao = new GeradorDeLogManutencao();

    @PostPersist
    public void postPersist() {
        geradorDeLogManutencao.gerarLogManutencao(this.getClass().getName());
    }
}
```

Listagem 6. Mapeamento da entidade Projeto para o documento PROJETO.

```
package br.com.gerenciamentoprojeto.entidade;

@Entity(value = "PROJETO", noClassnameStored = true)
public class Projeto implements Serializable {

    @Id
    private ObjectId id;

    private String codigo;

    @Indexed(value = IndexDirection.ASC, name = "nome", unique = true)
    private String nome;

    private String descricao;
    private Date dataPrevistaInicio;
    private Date dataPrevistaFim;

    @Reference("Tecnologia")
    private List<Tecnologia> tecnologias = new ArrayList<Tecnologia>();

    @Reference("Profissional")
    private List<Profissional> profissionais = new ArrayList<Profissional>();

    GeradorDeLogManutencao geradorDeLogManutencao =
    new GeradorDeLogManutencao();

    @PostPersist
    public void postPersist() {
        geradorDeLogManutencao.gerarLogManutencao(this.getClass().getName());
    }
}
```



Listagem 7. Mapeamento da entidade LogManutencao para o documento LOG_MANUTENCAO.

```
package br.com.gerenciamentoprojeto.entidade;

@Entity(value = "LOG_MANUTENCAO", noClassnameStored = true)
public class LogManutencao implements Serializable {

    @Id
    private ObjectId id;
    private String colecao;
    private Date dataHora;

    public String getColecao() {
        return colecao;
    }

    public void setColecao(String colecao) {
        this.colecao = colecao;
    }

    public Date getDataHora() {
        return dataHora;
    }

    public void setDataHora(Date dataHora) {
        this.dataHora = dataHora;
    }
}
```

Já no pacote `br.com.gerenciamentoprojeto.util` será implementada a classe `Conexao` (vide **Listagem 8**), responsável por abrir e fechar as conexões com o banco. Para isso, através da classe `MongoClient` é informado o IP da máquina (vide **Listagem 8**) onde está rodando o banco de dados e a classe `Morphia` criará a interface `Datastore`, utilizada para gerenciar todas as entidades no MongoDB. Neste mesmo pacote será desenvolvido o serviço `GeradorDeLogManutencao` (vide **Listagem 9**), que irá logar qualquer alteração realizada nas entidades do sistema. Para tanto, todas as entidades gerenciadas pelo Morphia terão um método `postPersist()` anotado com `@PostPersist` (vide **Listagens 4, 5 e 6**), sendo chamado pelo Morphia logo após a entidade ser persistida.

No pacote `br.com.gerenciamentoprojeto.servico` será implementado o serviço `GerenciadorDeProjetos` (vide **Listagem 10**). Nele estarão implementadas as funcionalidades de inclusão e obtenção dos dados no MongoDB. Para isso, esta classe fará uso da interface `Datastore` da API do Morphia.

Por fim, no pacote `br.com.gerenciamentoprojeto.test` será criada a classe de teste `GerenciadorDeProjetosTest` (vide **Listagem 11**), que através da classe de serviço `GerenciadorDeProjetos` e todas as outras classes implementadas utilizando as APIs do Morphia e MongoDB, incluirá no banco de dados um projeto completo com as tecnologias a serem empregadas e os profissionais que estarão alocados para desenvolver a aplicação e, em seguida, realizará os testes e validações necessários com o JUnit para verificar se as funcionalidades estão executando corretamente.

Os testes implementados, a princípio, irão verificar se o código do projeto incluído no MongoDB é `CADATRO_CLIENTES_GTY`.

Listagem 8. Serviço de infraestrutura responsável por criar o datastore e manipular as conexões com o MongoDB.

```
package br.com.gerenciamentoprojeto.util;

import org.mongodb.morphia.Datastore;
import org.mongodb.morphia.Morphia;
import com.mongodb.MongoClient;

public class Conexao {

    private static String IP_CONEXAO_MONGODB = "localhost";
    private static String NOME_BANCO_MONGODB = "gty";

    public static Datastore abrirConexao() {
        MongoClient mongo = new MongoClient(IP_CONEXAO_MONGODB);
        Morphia morphia = new Morphia();
        Datastore datastore = morphia.createDatastore(
            mongo, NOME_BANCO_MONGODB);
        return datastore;
    }
}
```

```
public static void fecharConexao(Datastore datastore) {
    datastore.getMongo().close();
}
```

Listagem 9. Serviço de infraestrutura responsável por gerar os logs de alterações realizadas nas coleções.

```
package br.com.gerenciamentoprojeto.util;

public class GeradorDeLogManutencao {

    public void gerarLogManutencao(String colecao) {
        Datastore datastore = Conexao.abrirConexao();
        LogManutencao logManutencao = new LogManutencao();
        logManutencao.setColecao(colecao);
        logManutencao.setDataHora(new Date());
        datastore.save(logManutencao);
        Conexao.fecharConexao(datastore);
    }
}
```



Morphia: Veja como simplificar o desenvolvimento de aplicações NoSQL

Listagem 10. Serviço de domínio responsável por manter as coleções do sistema no MongoDB.

```
package br.com.gerenciamentoprojeto.servico;

public class GerenciadorDeProjetos implements IGerenciadorDeProjetos {

    public void incluirTecnologia(Tecnologia tecnologia) {
        Datastore datastore = Conexao.abrirConexao();
        datastore.save(tecnologia);
        Conexao.fecharConexao(datastore);
    }

    public Tecnologia obterTecnologiaPorNome(String nome) {
        Datastore datastore = Conexao.abrirConexao();
        Query<Tecnologia> query = datastore
            .createQuery(Tecnologia.class).field("nome").equal(nome);
        List<Tecnologia> listTecnologia = query.asList();
        Conexao.fecharConexao(datastore);

        if (listTecnologia != null && !listTecnologia.isEmpty())
            return listTecnologia.get(0);
        else
            return null;
    }

    public void incluirProfissional(Profissional profissional) {
        Datastore datastore = Conexao.abrirConexao();
        datastore.save(profissional);
        Conexao.fecharConexao(datastore);
    }

    public List<Profissional> obterProfissionalPorFormacao(String formacao) {
        Datastore datastore = Conexao.abrirConexao();

        Query<Profissional> query = datastore
            .createQuery(Profissional.class)
            .field("formacao.curso").equal(formacao);
        List<Profissional> listProfissional = query.asList();
    }
}

    Conexao.fecharConexao(datastore);
}

public void incluirProjeto(Projeto projeto) {
    Datastore datastore = Conexao.abrirConexao();
    datastore.save(projeto);
    Conexao.fecharConexao(datastore);
}

public List<Projeto> obterTodosProjetos() {
    Datastore datastore = Conexao.abrirConexao();

    Query<Projeto> query = datastore.find(Projeto.class);
    List<Projeto> listProjeto = query.asList();

    Conexao.fecharConexao(datastore);
    return listProjeto;
}

public Projeto obterProjetoPorCodigo(String codigo) {
    Datastore datastore = Conexao.abrirConexao();

    Query<Projeto> query = datastore
        .createQuery(Projeto.class)
        .field("codigo").equal(codigo);
    List<Projeto> listProjeto = query.asList();
    Conexao.fecharConexao(datastore);

    if (listProjeto != null && !listProjeto.isEmpty())
        return listProjeto.get(0);
    else
        return null;
}
```

```
K:\workspaces\controleProjetos\controleProjeto>mvn test
olha o java K:\Lincoln\softwares\jdk-7u40-hotspot-x64
[INFO] Scanning for projects...
[INFO]
[INFO] Building Unnamed - br.com.gerenciamentoprojeto:controleProjeto:jar:0.0.1-SNAPSHOT
[INFO]   task-segment: [test]
[INFO]
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources {execution: default-testResources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: K:\workspaces\controleProjetos\controleProjeto\target\surefire-reports

TESTS

Running br.com.gerenciamentoprojeto.repositorio.GerenciadorDeProjetosTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 6.255 sec
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 8 seconds
[INFO] Finished at: Sat Oct 31 07:25:39 BRST 2015
[INFO] Final Memory: 19M/229M
[INFO] -----
```

Figura 2. Prompt de comando com a execução e resultados dos testes

Na sequência, o teste verificará se o Morphia armazenou corretamente o profissional formado em Ciência da Computação, e por último, se as Tecnologias JAVA e MONGODB foram cadastradas corretamente.

Para iniciar o teste, no prompt de comando, acesse o diretório onde se encontra o arquivo *pom.xml* e execute *mvn test*. Este comando fará com que os testes da classe **GerenciadorDeProjetosTest** (vide Listagem 11) sejam executados e os resultados apresentados, conforme a Figura 2.

Para sistemas e módulos de software que têm como característica o processamento e a manipulação de grandes volumes de dados, a utilização de bancos de dados relacionais deixou de ser a solução “bala de prata” por conta dos altos custos e da complexidade para escalar a aplicação através da configuração de um cluster com várias máquinas.

Neste contexto, a opção por adotar bancos de dados NoSQL como o MongoDB para atender a necessidade de alta performance e desempenho

Listagem 11. Testes de integração responsáveis por validar as funcionalidades de manutenção de documentos no MongoDB.

```
package br.com.gerenciamentoprojeto.test;

public class GerenciadorDeProjetosIT {

    IGerenciadorDeProjetos gerenciadorDeProjetos = new GerenciadorDeProjetos();

    @Test
    public void incluirProjetoDeCadastroDeClienteDaGTY() {
        Tecnologia tecnologiaJava = new Tecnologia();
        tecnologiaJava.setNome("JAVA");
        tecnologiaJava.setTipo("LINGUAGEM_PROGRAMACAO");
        gerenciadorDeProjetos.incluirTecnologia(tecnologiaJava);

        Tecnologia tecnologiaMongoDB = new Tecnologia();
        tecnologiaMongoDB.setNome("MONGODB");
        tecnologiaMongoDB.setTipo("BANCO_DADOS");
        gerenciadorDeProjetos.incluirTecnologia(tecnologiaMongoDB);

        //Incluindo profissionais
        Profissional profissionalRaquel = new Profissional();
        profissionalRaquel.setNome("Raquel Queiros");
        profissionalRaquel.setFuncoes(new ArrayList<String>(Arrays.asList(new String[]{"DESENVOLVEDOR"})));
        profissionalRaquel.setFormacao(new Formacao("UEMG", "Ciencia da Computacao", "4 anos"));

        Profissional profissionalMaria = new Profissional();
        profissionalMaria.setNome("Maria das Graças");
        profissionalMaria.setFuncoes(new ArrayList<String>(Arrays.asList(new String[]{"SCRUM_MASTER", "DBA"})));
        profissionalMaria.setFormacao(new Formacao("PUC", "Sistemas de Informacao", "4 anos"));

        gerenciadorDeProjetos.incluirProfissional(profissionalRaquel);
        gerenciadorDeProjetos.incluirProfissional(profissionalMaria);

        //Incluindo projeto de cadastro de clientes da GTY
        Projeto projetoDeCadastroDeClienteDaGTY = new Projeto();
        projetoDeCadastroDeClienteDaGTY.setCodigo("CADASTRO_CLIENTES_GTY");
        projetoDeCadastroDeClienteDaGTY
            .setNome("Sistema de Cadastramento de Clientes da GTY");
        projetoDeCadastroDeClienteDaGTY
            .setDescricao("Realizar a manutenção e cadastro de todos os Clientes da GTY");
        projetoDeCadastroDeClienteDaGTY.setDataPrevistaInicio(new Date());
    }

    //Tecnologia
    Tecnologia tecnologiaJavaCadastrada = gerenciadorDeProjetos
        .obterTecnologiaPorNome("JAVA");
    Tecnologia tecnologiaMongoDBCadastrada = gerenciadorDeProjetos
        .obterTecnologiaPorNome("MONGODB");

    List<Tecnologia> tecnologiasDoProjeto = new ArrayList<Tecnologia>();
    tecnologiasDoProjeto.add(tecnologiaJavaCadastrada);
    tecnologiasDoProjeto.add(tecnologiaMongoDBCadastrada);
    projetoDeCadastroDeClienteDaGTY.setTecnologias(tecnologiasDoProjeto);

    List<Profissional> listProfissionaisDeCienciaDaComputacao =
        gerenciadorDeProjetos
            .obterProfissionalPorFormacao("Ciencia da Computacao");

    projetoDeCadastroDeClienteDaGTY
        .setProfissionais(listProfissionaisDeCienciaDaComputacao);

    //Salvando o projeto no MongoDB
    gerenciadorDeProjetos.incluirProjeto(projetoDeCadastroDeClienteDaGTY);

    //Testando os documentos persistidos no MongoDB
    Projeto projetoBD = gerenciadorDeProjetos
        .obterProjetoPorCodigo("CADASTRO_CLIENTES_GTY");
    assertEquals(projetoBD.getCodigo(), "CADASTRO_CLIENTES_GTY");
    assertEquals(projetoBD.getNome(),
        "Sistema de Cadastramento de Clientes da GTY");
    assertEqualsIgnoreCase("Sistema de Cadastramento de Clientes da GTY");

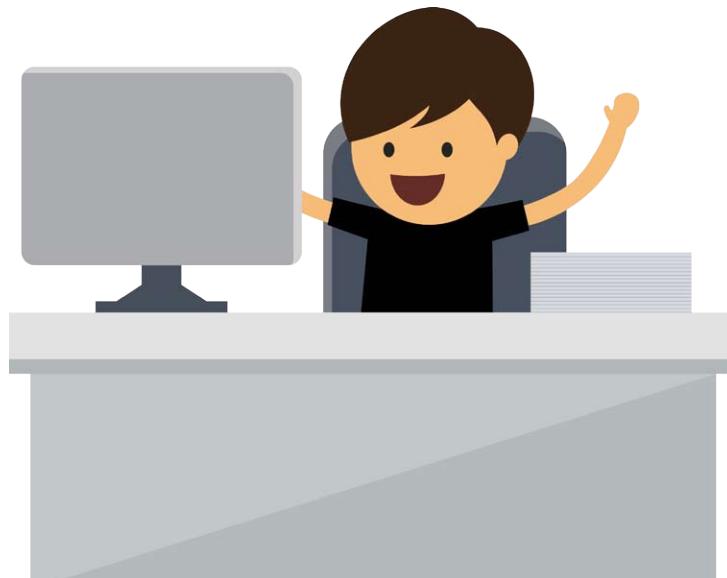
    boolean persistiuProfissionalFormadoEmCienciaDaComputacao = false;
    for (Profissional profissional : projetoBD.getProfissionais()) {
        persistiuProfissionalFormadoEmCienciaDaComputacao = true;
        assertTrue(profissional.getFormacao()
            .getCurso());
        assertEqualsIgnoreCase("Ciencia da Computacao");
    }
    assertTrue(persistiuProfissionalFormadoEmCienciaDaComputacao);
    boolean persistiuTecnologias = false;
    for (Tecnologia tecnologia : projetoBD.getTecnologias()) {
        persistiuTecnologias = true;
        assertEquals(tecnologia.getNome(),
            "JAVA" || tecnologia.getNome()
            .equalsIgnoreCase("MONGODB"));
    }
    assertTrue(persistiuTecnologias);
}
```

em sistemas que não precisam de um controle transacional avançado, se torna cada vez mais viável.

A plataforma Java, sendo a mais importante tecnologia para desenvolvimento de sistemas corporativos, disponibiliza drivers e frameworks como o Morphia para a interação da aplicação com a solução NoSQL e conforme demonstrado no artigo, a implementação de aplicativos Java com bancos NoSQL, em especial o MongoDB, é realizada de forma bastante simples.

Dentre as diversas opções disponíveis, um dos principais diferenciais do Morphia é a possibilidade de mapeamento das entidades para os documentos do MongoDB. Isso garante que a estrutura da aplicação tenha um design fácil de manter, realizar correções e evoluções.

Ainda assim, os SGBDRs continuam sendo uma ótima opção de armazenamento de dados para muitas empresas e aplicações por conta da sua consistência, maturidade, padronização da



linguagem de consulta, entre outras características. Ou seja, o modelo de dados relacional continua e continuará atendendo a sistemas para o armazenamento de banco de dados. Agora, o que temos com as opções NoSQL, são mais alternativas para criar uma solução com a arquitetura ideal para suas reais necessidades.

Enfim, a escolha por um tipo de banco de dados relacional ou NoSQL depende, principalmente, do problema e domínio de negócio a ser resolvido, do volume de dados trafegado na aplicação, do número de usuários simultâneos e da necessidade de controle transacional. Levando em consideração essas questões, podemos realizar a escolha adequada do tipo de banco a ser adotado na aplicação, o que não impede, também, de se implementar uma estrutura poliglota, híbrida, na qual, após dividir a aplicação em subdomínios, é possível definir para cada subdomínio o tipo de banco de dados que melhor atende as necessidades para o armazenamento.

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Autor



Lincoln Fernandes Coelho

lincolnfcoelho@gmail.com

Bacharel em Ciéncia da Computaçao, trabalha com desenvolvimento de software há 12 anos, sendo especialista na plataforma Java, design, boas práticas e qualidade de código-fonte. Possui as certificações SCJP, SCWCD e SCBCD.



Links e Referências:

Site do banco de dados NoSQL MongoDB.

<http://www.mongodb.org>

Endereço para download do MongoDB.

<http://www.mongodb.org/downloads>

Site do projeto Morphia.

<https://mongodb.github.io/morphia/>

NoSQL – Essencial. Pramod J. Sadalage e Martin Fowler, Novatec, 2013.

Introdução ao MongoDB – Hows, David; Membrey, Peter; Plugge, Eelco, Novatec, 2015.

UML – Essencial. Martin Fowler, 3º Edição, Bookman, 2005.

TDD – Desenvolvimento Guiado Por Testes. Kent Beck, Bookman, 2010.



Dominando os tipos de testes automatizados

Aprenda neste artigo técnicas para escrever testes automatizados considerando a especificidade de cada problema e domine-os de uma vez por todas

A automatização de testes é um tema de grande relevância quando se fala em qualidade de software e por isso, o uso desta disciplina deve ser considerado em todos os projetos de software. Com testes automatizados consegue-se entender melhor os problemas, já que o desenvolvedor, pela prática, valida sua hipótese considerando diferentes cenários. Além disso, reduz-se o stress e aumenta-se a satisfação, pois com um bom conjunto – ou suíte – de testes, bugs são detectados mais cedo no ciclo de desenvolvimento e menos problemas chegam ao cliente, diminuindo com isso o custo na criação de novos produtos, visto que o código com testes automatizados é construído com mais cuidado, o que sugere menos bugs e, consequentemente, menos gastos com manutenção. Consequentemente, reduz-se também o custo com evoluções no sistema, já que uma alteração que possa causar efeitos colaterais é rapidamente evidenciada pelos testes, permitindo identificar os pontos falhos de forma clara e objetiva, alcançando assim correções ágeis e entregas com menos erros. Constatase, portanto, que a adoção de testes automatizados oferece ganhos em diversas etapas da construção de um sistema.

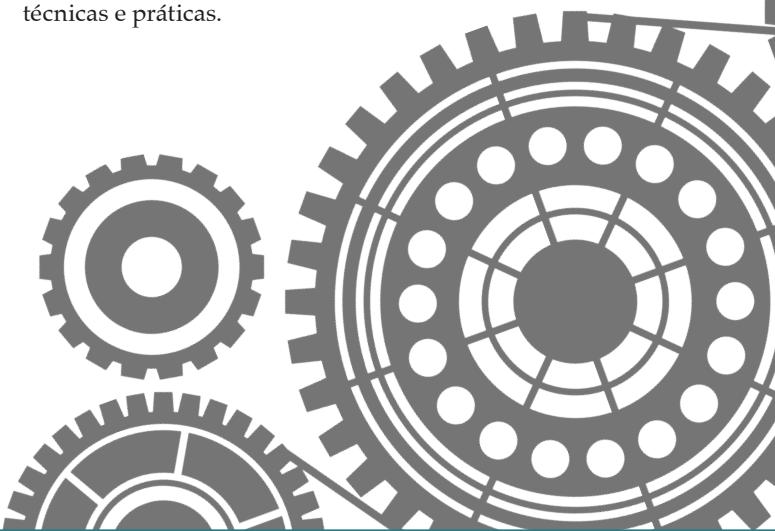
Contudo, sua aplicação eficaz é uma atividade longe do trivial. Não apenas pelo uso de ferramentas e frameworks, mas principalmente por causa do entendimento dos conceitos envolvidos. Sabe-se que existem diferentes tipos de testes automatizados, específicos para cada situação (testes unitários para testar unidade, testes de integração para testar componentes, testes de aceitação para testar funcionalidades), mas comumente observa-se que a distinção nem sempre é considerada. O problema com isso é que um mal entendimento dos

Fique por dentro

Este artigo é útil por explicitar e analisar as diferentes formas de adotar testes automatizados. Para isso, serão analisados aspectos teóricos, como os conceitos relacionados aos diferentes tipos de testes, e práticos, como a automatização de tais conceitos utilizando técnicas e frameworks. Além disso, o papel dos testes automatizados em times ágeis e em sistemas legados também será explorado. O objetivo com isso é fornecer uma base conceitual, que permita ao leitor aplicar testes automatizados alinhados à necessidade de forma eficaz.

conceitos relacionados pode levar à lentidão desnecessária na execução dos testes, dificuldades na otimização do processo de integração contínua, falhas na comunicação sobre os testes dentro do próprio time, entre outros.

A partir dessa contextualização, este artigo analisará os diferentes tipos de testes automatizados, considerando algumas técnicas e práticas.



Dominando os tipos de testes automatizados

Entre quadrantes e pirâmides

Idealmente, o desenvolvedor deveria garantir que todo código-fonte entregue possui testes automatizados que o validam. Essa proposta parte do princípio de que não se sabe se algo realmente funciona como esperado até que seja efetivamente testado. Diante disso, nos cenários onde o desenvolvimento de software é estruturado de acordo com métodos ágeis e onde os times fazem uso de

BOX 1. User Stories

User Stories, ou Histórias de Usuário, são uma prática comum em times que fazem uso de métodos ágeis, como XP e Scrum. Elas podem ser encaradas como uma forma clara, direcionada, menos ambígua e mais voltada à construção de software do que a escrita tradicional de requisitos. A User Story define uma funcionalidade de forma abrangente, e sobre ela são criadas as tarefas necessárias para sua realização. Por exemplo, para uma funcionalidade de login de usuário, a técnica de User Story poderia apresentar alguns dos requisitos no seguinte formato:

Usuário efetua login no sistema com sucesso

Dado um usuário com permissão

QUANDO o mesmo efetua login no sistema

ENTÃO é redirecionado para a página principal

Usuário efetua login no sistema com falha

Dado um usuário com permissão

QUANDO tenta efetuar login com credenciais inválidas

ENTÃO recebe mensagem indicando erro na autenticação

E não consegue efetuar login.

User Stories (vide **BOX 1**), espera-se que cada tarefa de codificação possua também um esforço para criação dos respectivos testes automatizados. Dessa forma, garante-se a construção de código com melhor qualidade.

A noção de que cada requisito de software precisa ter um teste associado aumenta consideravelmente a cobertura de código sendo testado. Contudo, também é importante identificar qual tipo de teste é mais adequado para cada situação, pois não se deve utilizar o mesmo para tudo. Para que se saiba que teste automatizado escrever, um passo fundamental é conhecer as opções existentes. No livro *Agile Testing: A Practical Guide for Testers and Agile Teams*, as autoras Lisa Crispin e Janet Gregory apresentam um quadrante organizando os diferentes tipos de testes (vide **Figura 1**).

O enfoque deste quadrante é direcionado ao trabalho com times ágeis, mas não significa que os tipos dispostos são relevantes apenas a times com esta organização. Como pode-se verificar, cada setor da figura é numerado, de Q1 a Q4, de acordo com sua característica (e cada setor do quadrante possui um balão, que define a forma de realização dos testes). O quadrante Q1, com o balão *Automated*, apresenta tipos de testes que podem ser feitos de forma automatizada, enquanto Q2, com o balão *automated & manual*, apresenta tipos que podem ser implementados tanto de forma manual quanto automatizada. Já o quadrante Q3 lida com testes que são feitos manualmente, e Q4 está relacionado a testes que devem ser feitos com ferramentas especializadas (*tools*).

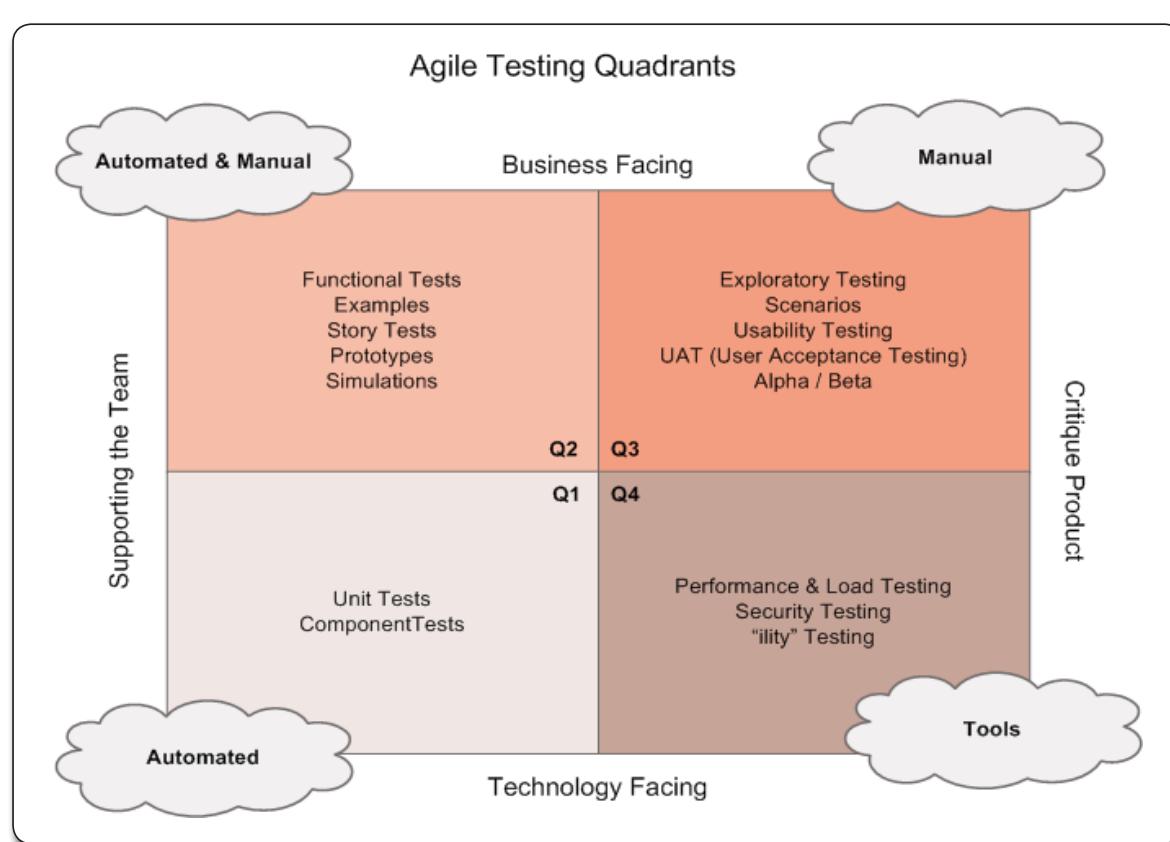


Figura 1. Quadrante de testes de Crispin e Gregory

Visto que este artigo trata especificamente de testes automatizados, focaremos nossa análise nos tipos apresentados nos quadrantes Q1 e Q2. Iniciemos, então, nossa análise, sobre os tipos de testes presentes em Q1, os testes unitários (*Unit Tests*) e os testes de componentes (ou *Component Tests*, também conhecidos como testes de integração).

Testes unitários

Testes unitários são aqueles que, como o nome indica, testam uma unidade. Esta afirmação, contudo, não é muito precisa, levantando algumas dúvidas. Afinal, o que é uma unidade? É uma classe? Um método? É uma tela?

Certamente não é uma tela, e também não se trata, necessariamente, de uma única classe. No livro *Continuous Integration: Improving Software Quality and Reducing Risk*, os autores Duvall, Matyas e Glover afirmam que testes unitários validam o comportamento de pequenos elementos em um sistema, elementos esses que, na orientação a objetos, costumam ser chamados de métodos. Assim sendo, o teste unitário é aquele que testa os métodos de uma classe de produção (ver **BOX 2**).

BOX 2. Classe de produção e classe de testes

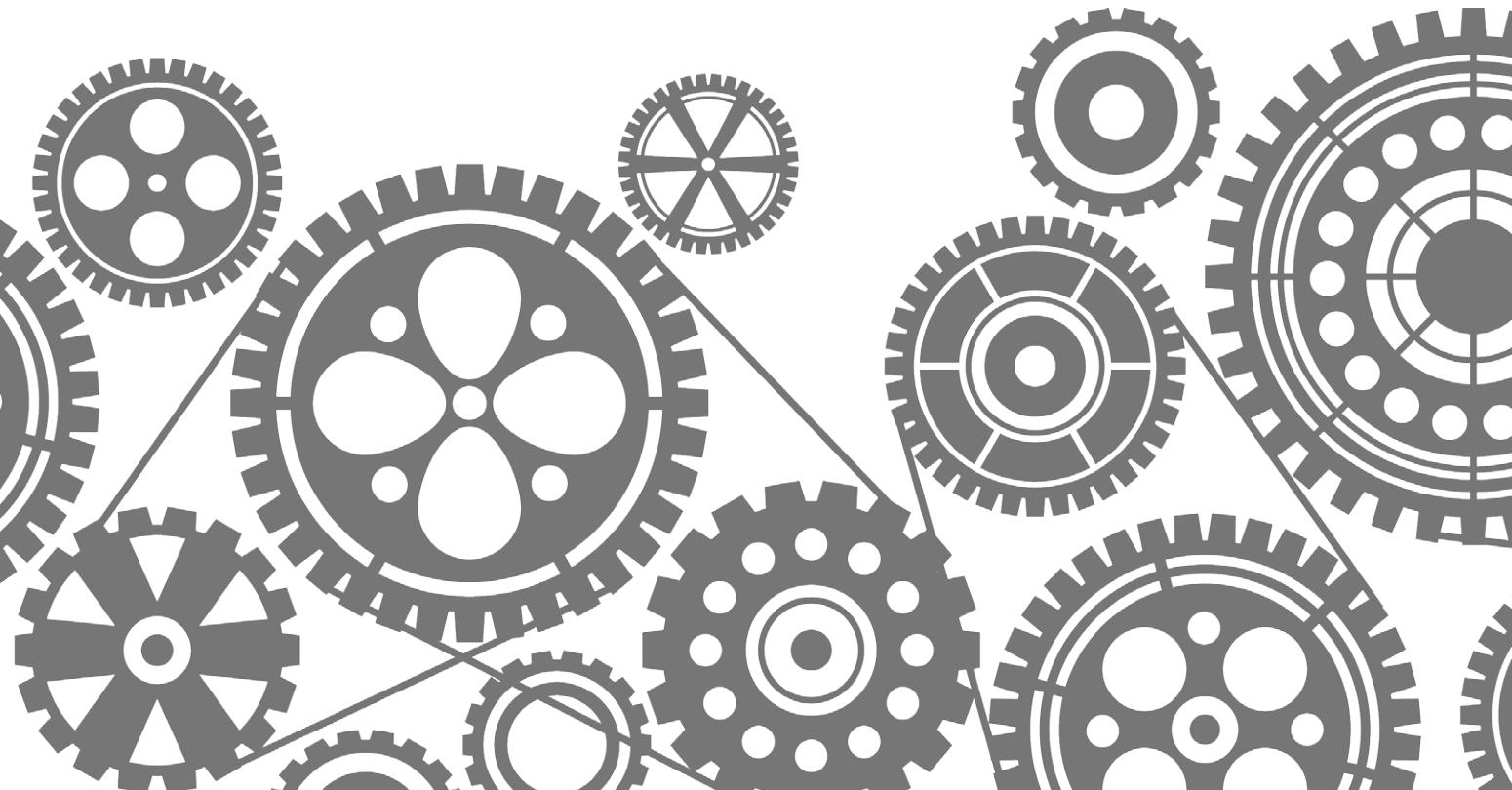
Uma classe de produção é aquela que é efetivamente testada pela classe de teste. Ela recebe esse nome porque é a classe que é entregue junto com a aplicação (diferentemente da classe de testes, que existe apenas durante o desenvolvimento e não entra no empacotamento do artefato JAR ou WAR a ser executado em produção). A classe de produção também é chamada de classe sobre teste, ou classe sendo testada (CUD – Class Under Test). Uma classe de testes, por sua vez, tem a função exclusiva de testar uma CUD.

Mas não é só isto. A definição de teste unitário deve considerar ainda o nível de acoplamento das dependências do código de produção. Quando um código de produção está acoplado a recursos externos (como banco de dados, web services e disco rígido), o teste deixa de ser unitário e passa a ser de integração (analisado posteriormente). Em suma, para um teste ser unitário, os métodos da classe sendo testada (e suas dependências) não podem ter relação com recursos externos.

Em suma, um teste unitário testa uma unidade, e uma unidade é uma classe de produção que pode ou não possuir dependências. Caso as possua, tais dependências devem ser desacopladas de recursos externos.

Por exemplo, se uma classe precisa de um container para injecção de dependência (como o Spring), ou se faz acesso direto a um banco de dados, ou implementa um cliente para consumo de um web service, ela depende de recursos externos e, segundo a definição, não pode ser testada de forma unitária. Contudo, vale ressaltar que é possível isolar esta dependência, possibilitando que a classe seja testada de forma unitária. Para isso, algumas técnicas podem ser consideradas:

- **Refatoração: Extract Method/Class** – Se uma classe ou método sobre teste possui dependência de recursos externos, pode-se realizar o *Extract Method* ou *Extract Class*. Estes padrões, descritos no Livro *Refactoring: Improving the Design of Existing Code*, de Martin Fowler, sugerem formas de isolar o código para que seja mais facilmente testável. Os padrões podem ser usados para separar o código que possui dependência de recursos externos do código que não possui. Assim, o que não possui esse tipo de dependência pode ser testado de forma unitária;



- **Mock:** O conceito de mock (*similar*) também trata de isolar a dependência externa, mas de forma mais intrusiva. Para tanto, programaticamente, define-se como determinado código deve se comportar, simulando seu comportamento quando ele é chamado. Por exemplo, se alguém deseja testar de forma unitária uma classe que depende de uma injeção de dependência em um atributo **ProdutoService**, é possível simular este atributo, de modo que quando o mesmo for referenciado, em vez de realizar seu comportamento padrão, realize um comportamento específico. Assim, o teste pode rodar sem a necessidade de um container para injeção de dependência, como o Spring.

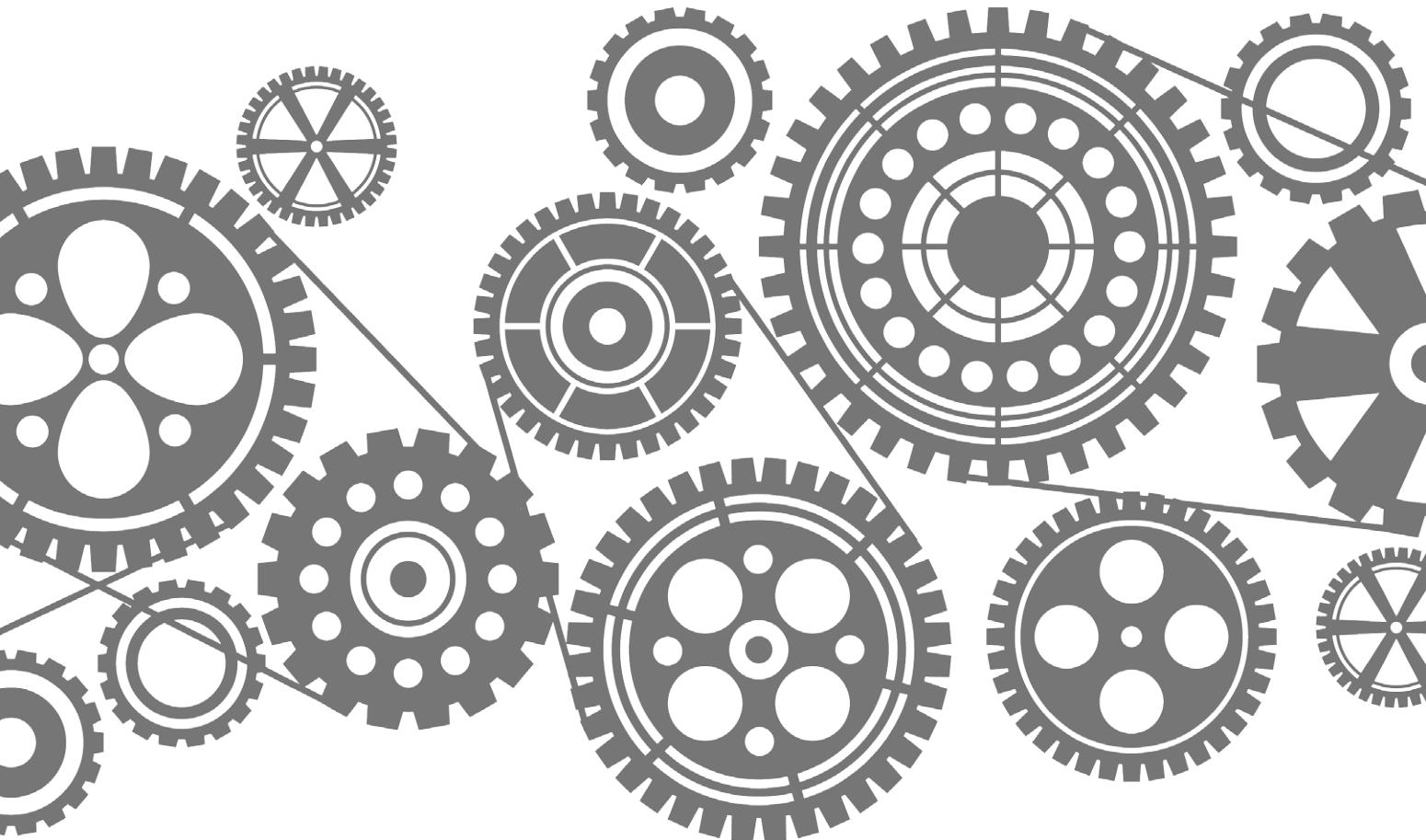
A escolha por uma técnica ou outra pauta-se numa decisão de design: deseja-se alterar o código de forma a tornar as dependências a recursos externos mais isoladas do código Java puro? Se a resposta for sim, a refatoração faz-se adequada. Caso contrário, pode-se utilizar mock. Apenas uma ressalva ao uso de mock: seu ponto negativo é que ele expõe a implementação da classe sobre teste. Para simular um comportamento, é preciso saber como a classe sobre teste se comporta por dentro, e levar esta lógica para o teste. Note que neste modo o teste está programando o comportamento interno da classe. O problema é que, quando a classe mudar internamente, o teste também precisará ser alterado. No entanto, isso não deveria ser necessário, afinal, o teste deveria validar o comportamento da classe

sem saber como a mesma funciona por dentro, preservando o encapsulamento.

Uma ótima opção para elevar o nível dos testes unitários é adotar o TDD (*Test Driven Development*). Nesta técnica, muito utilizada principalmente em ambientes ágeis, a classe de testes é criada antes da classe de produção, de forma que os testes guiam o código a ser implementado, testado e posteriormente colocado em produção.

Além de oferecer alta cobertura aos testes (ao deixar para fazer os testes depois, podemos esquecer de implementá-lo ou ficar sem tempo para isso), essa abordagem também pode ser vista como uma técnica de *design emergente*, onde o design evolui de acordo com a construção do software, em vez de ser uma etapa anterior à construção. Algo bem diferente do tradicional modelo *Big Design Up Front*, que sugere uma análise de todos os cenários do sistema antes que o mesmo seja construído. Mais detalhes sobre TDD podem ser encontrados no livro de Kent Beck, intitulado *Test Drive Development: By Example*.

Outro conceito importante a se considerar: os testes unitários são testes de caixa branca. Isto significa que eles testam o comportamento interno do sistema e, portanto, devem ser escritos por desenvolvedores. Analistas de requisitos ou testadores podem realizar outros tipos de testes, mas não têm o perfil para escrever testes unitários. Lembre-se que os testes unitários são classes escritas em Java com o propósito de realizar testes



especificamente sobre classes Java de produção. Em outro artigo, demonstraremos como utilizar os frameworks de testes unitários JUnit e Hamcrest.

Em suma, as principais vantagens dos testes unitários são: ser simples de construir; rápidos de rodar (quando o conceito aqui descrito é entendido e aplicado corretamente); fáceis de executar via integração contínua, o que permite que funcionem como testes de regressão, apontando rapidamente problemas de efeitos colaterais por alterações indevidas; e viabilizam a prática do TDD, o que possibilita um design claro e uma grande cobertura de testes.

Testes de Integração/Componente

Os testes de integração são caracterizados, segundo Duvall, Matyas e Glover, pela verificação de partes maiores do sistema que dependem de recursos externos, como banco de dados, sistemas de arquivos ou *endpoints* de rede, para citar alguns. Estes testes verificam se os componentes em análise realmente produzem o comportamento esperado.

Um teste de integração comum a muitas aplicações é aquele que valida as ações executadas contra um banco de dados. Considerando que o acesso a recursos externos sempre demanda mais tempo do que um acesso direto à memória (por questões diversas, como latência de rede, leitura de disco, dentre outras), é comum que os testes de integração demorem mais do que os unitários.

Esse tipo de teste também é mais difícil (e demorado) de implementar. O principal motivo para isso está relacionado ao estado das dependências externas, que deve ser preparado e garantido. Um teste, por definição, precisa ser independente e determinístico, ou seja, ao ser executado múltiplas vezes, deve apresentar o mesmo resultado. Por exemplo, um teste que dependa de um banco de dados precisa que o banco de dados esteja sempre no mesmo estado consistente no qual o teste se baseia. De outra forma, o teste falharia pelo motivo errado (não por identificar um problema na classe de produção, mas porque os dados esperados para a realização do teste não estavam consistentes). Neste cenário, é comum a necessidade de preparar o banco de dados antes da execução do teste e garantir que o mesmo volte ao estado original após a execução.

A maior dificuldade na implementação dos testes de integração encontra-se exatamente neste ponto: o uso adequado de recursos externos. Para demonstrar essa questão, imagine que desejamos testar uma funcionalidade de alteração de uma entidade que dependa de mais cinco (todas mapeadas em tabelas distintas). Neste cenário, precisamos de seis tabelas populadas da forma correta para que possamos realizar o teste e ter a certeza de que a classe de produção funciona conforme o esperado. Agora, imagine que entre uma execução e outra um desenvolvedor da equipe altere uma dessas tabelas. Com isso, o teste que até então estava funcionando, para de funcionar sem motivo aparente. O que ocorreu é que a mudança de outra pessoa provocou um efeito colateral, que levou o teste a

falhar – apesar do código Java não ter sido alterado. A alteração da entidade continua a ocorrer corretamente, mas o teste falha mesmo assim. Isto é, o teste não falhou porque a alteração da entidade tem um *bug*, mas porque o pré-requisito para a realização do teste parou de ser atendido.

Para que este problema não aconteça, o teste automatizado precisa garantir o estado do banco de dados antes e após sua execução. O teste precisa inserir o dado do qual depende (pré-condição), realizar o teste e então remover os dados alterados (pós-condição). Este requisito de atrelar massa de dados ao teste pode ser realizado pelo uso de alguns frameworks, como o DBUnit ou mesmo o Spring-Test, com alguma configuração, conforme será demonstrado ainda neste artigo.

Martin Fowler analisa problemas relacionados a este de forma clara e sucinta em seu artigo *Eradicating Non-Determinism in Tests*, onde sugere que os testes deixam de ser confiáveis quando param de ser determinísticos, o que leva a serem abandonados posteriormente. Para ele, o não-determinismo surge por cinco causas principais, que precisam ser evitadas, a saber:

1. Falta de isolamento: Trata da questão já exemplificada, onde o estado de um banco de dados (ou outros recursos externos) interfere na correta execução do teste;

2. Comportamento assíncrono: A utilização de *sleep* em threads pode tanto tornar os testes lentos quanto interromper a execução do método por causa do timeout. Portanto, em cenários com processamento assíncrono, isto é, com threads executando em paralelo, use callbacks para garantir que o teste seja executado no momento correto;

3. Serviços remotos: Em alguns casos é comum o uso de serviços remotos reais para testes de integração entre sistemas, já que nem sempre certos serviços estarão disponíveis para teste. No entanto, sistemas reais podem não prover as respostas determinísticas do teste, pois mudam frequentemente. Por exemplo, um serviço que retorna a temperatura de uma cidade pode retornar resultados diferentes a cada consulta. Desta forma, é preciso fazer uso de um *Test Double* (vide **BOX 3**), simulando o comportamento do serviço real;

4. Tempo: Depender do relógio é algo claramente não-determinístico. Testes que dependem de tempo (hora corrente) não irão gerar resultados iguais. Portanto, não se deve depender de horário em cenários de teste;

5. Vazamento de recursos: Quando os recursos são mal gerenciados, os testes podem passar a falhar por motivos errados, apresentando comportamento não-determinístico. Falta de memória é um dos problemas mais comuns. Para lidar com vazamentos, uma solução comumente adotada é a implementação de um pool de recursos (um mecanismo para reaproveitar objetos, de forma que novos objetos não precisem ser criados constantemente, o que leva ao potencial vazamento). Deste modo, sempre utilize um pool de recursos em cenários relevantes.

Com o uso de *Test Double* o desenvolvedor pode remover a dependência a recursos externos. Isto significa que o uso de *Test*

Doubles permite que funcionalidades que até então deveriam ser avaliadas por testes de integração sejam avaliadas por testes unitários. Esta afirmação é especialmente relevante dentro do contexto da integração contínua, onde a execução de testes unitários e de integração precisa ser separada, por conta, principalmente, do tempo de execução.

BOX 3. Test Double

Gerard Meszaros, em seu livro *xUnit Test Patterns*, se refere ao termo *Test Double* como um conjunto de objetos que pode ser utilizado para substituir uma classe de produção ou um conjunto delas durante os testes.

Os Test Doubles são categorizados em tipos, a saber:

- **Dummy:** São objetos que servem simplesmente para preencher parâmetros de métodos. Atendem aos cenários onde deseja-se chamar dado método, mas os objetos sendo passados por parâmetro são irrelevantes, não precisando ser corretamente construídos. Por exemplo, considere o cenário onde se quer chamar um método calcular(usuario), mas, por causa de um design ruim, o objeto usuario não é relevante para o cálculo. Neste caso, criamos um objeto usuario qualquer (Dummy), apenas para passá-lo por parâmetro, de forma a atender os objetivos do teste. Objetos dummy são passados por parâmetro, mas nunca são utilizados;
- **Fake:** É uma implementação do código real, mas que não atende aos propósitos de produção, servindo apenas para teste. Por exemplo, considere um serviço que obtenha uma lista de todas as contas de um usuário. O serviço real vai até o banco de dados obter as contas. O serviço Fake, por sua vez, simplesmente traz uma lista pré-determinada de contas. Desta forma, realiza uma ação básica e simplificada, diferente da ação real, sofisticada, que é realizada pelo código de produção. Esta opção é útil principalmente para simular a ação de serviços de terceiros quando os mesmos não estão disponíveis para testes de forma determinística;
- **Stub:** Provê retorno determinístico de chamadas a métodos. Para isso, retorna objetos que possuem sempre o mesmo valor, eliminando a necessidade de construir objetos que dependam de recursos externos, como bancos de dados e web services;
- **Spy:** Também provê retorno determinístico, mas, diferentemente do Stub (que sempre retorna o mesmo valor), apresenta valores diferentes dependendo da forma pela qual foi chamado;
- **Mock:** Esta opção permite que dado método em teste possa ser programado para, durante a execução do teste, realizar um comportamento diferente do original (simular um comportamento). Isso é útil, por exemplo, para ignorar chamadas a recursos externos quando se está testando um código que não faz uso de tais recursos, mas cujo acoplamento existe por questões de design da classe.

Contudo, dúvidas podem surgir a partir dessa constatação: se estou testando dois módulos de um sistema, isto não é um teste de integração, independentemente de utilizar recursos externos ou não? Para responder a esta questão é preciso considerar a arquitetura da aplicação: estes dois módulos são duas aplicações que dependem de um container para executar, efetuam sua integração via web services ou via banco de dados? Caso positivo, certamente trata-se de um teste de integração, mas se a dependência entre dois módulos for puramente de biblioteca (como um módulo que referencia uma classe POJO de outro módulo), um teste unitário dá conta do recado. E se um módulo A depende de um web service de um módulo B, mas o web service do módulo B é simulado por meio de um Test Double (um mock, por

exemplo), observa-se que o módulo A passa a não depender mais do módulo B para o teste, pois a função do Test Double é justamente a de viabilizar este desacoplamento. Temos, então, um único módulo, que pode ser testado de forma unitária, sem depender do segundo.

Para que testes automatizados sejam confiáveis, precisam gerar o mesmo resultado independentemente do número de vezes que forem executados. Esse determinismo é mais fácil de atingir quando a aplicação é construída respeitando as boas práticas da orientação a objetos, como a alta coesão e o baixo acoplamento. Quando a aplicação não considera esses valores, a implementação de testes de qualquer tipo pode se tornar muito difícil. Em cenários onde existe código acoplado e há interesse no uso de testes automatizados que o validem, além do uso de *Test Doubles*, outra alternativa para simplificar os testes é proposta por Meszaros: o *Humble Object Pattern*. Este padrão sugere que um código altamente acoplado precisa ser desacoplado do seu ambiente através da criação de objetos menores, de forma a se tornar mais testável.

Por exemplo, se deseja-se testar um serviço que possui muitas dependências a outros serviços e muitos métodos que realizam cálculos diversos, sugere-se com este padrão que os métodos de cálculos sejam extraídos para classes à parte (o mesmo princípio de Refatoração: *Extract Method/Class*, descrito na seção de testes unitários), especializadas e que podem ser testadas de forma isolada e unitária. Desta forma, reduz-se o acoplamento, o que apresenta duas vantagens: a) menos código será testado como teste de integração e mais código será testado como teste unitário; b) simplifica-se a escrita dos testes (testes de integração são mais difíceis e demorados de escrever).

Garantido o determinismo, os testes de integração apresentam-se como uma categoria muito útil na verificação do correto funcionamento do sistema. Quando planejados e implementados adequadamente, os testes entre diferentes componentes apresentam ganhos valiosos na identificação prematura de bugs no sistema. Inclusive, observa-se que testes unitários e de integração se complementam neste sentido: enquanto o primeiro ajuda na detecção de bugs em aspectos pontuais de classes, o segundo identifica bugs nestas mesmas classes avaliando a execução das mesmas em grupo. Ademais, ambos os testes são de caixa branca, o que significa que são construídos por meio de código Java, exigindo perfil de desenvolvedor.

Testes e critérios de aceitação

O quadrante Q2 também é conhecido como o quadrante dos testes de aceitação, e assim será referenciado neste artigo. Nele, os testes que podem ser automatizados são os funcionais e de histórias; os demais são tratados como manuais. Em nosso estudo, no entanto, optamos por analisar apenas os testes de histórias.

Como o leitor deve lembrar, as user stories já foram mencionadas, mas nos testes de aceitação ganham uma nova dimensão: podem ser vinculadas a testes automatizados, pelo uso da técnica de BDD (*Behavior Driven Development*, ou desenvolvimento voltado a comportamento). Isso ocorre porque o BDD define um padrão de escrita de histórias que é pragmático e suficiente para ser automatizável. JBehave e Concordion são alguns dos frameworks que fazem uso direto do conceito de BDD para automatizar as histórias.

O modelo de escrita de histórias do BDD é o seguinte:

DADO [uma pré-condição] (opcional)
QUANDO [um problema ocorrer]
E [outro problema ocorrer] (opcional)
ENTAO [uma ação deve ser realizada]
E [outra ação deve ser realizada] (opcional)

Por exemplo, suponha que o cliente solicitou uma funcionalidade para cadastro de CPFs. Algumas histórias para este cenário podem ser escritas da seguinte forma:

User Story: Usuário cobra CPF com sucesso

DADO um usuário editando seus dados pessoais
QUANDO o CPF informado é válido
ENTAO o CPF é devidamente atualizado
E o sistema informa que o CPF foi cadastrado com sucesso

User Story: Usuário informa CPF incorreto no cadastro

DADO um usuário editando seus dados pessoais
QUANDO o CPF informado é inválido
ENTAO o sistema informa que o CPF não pode ser cadastrado

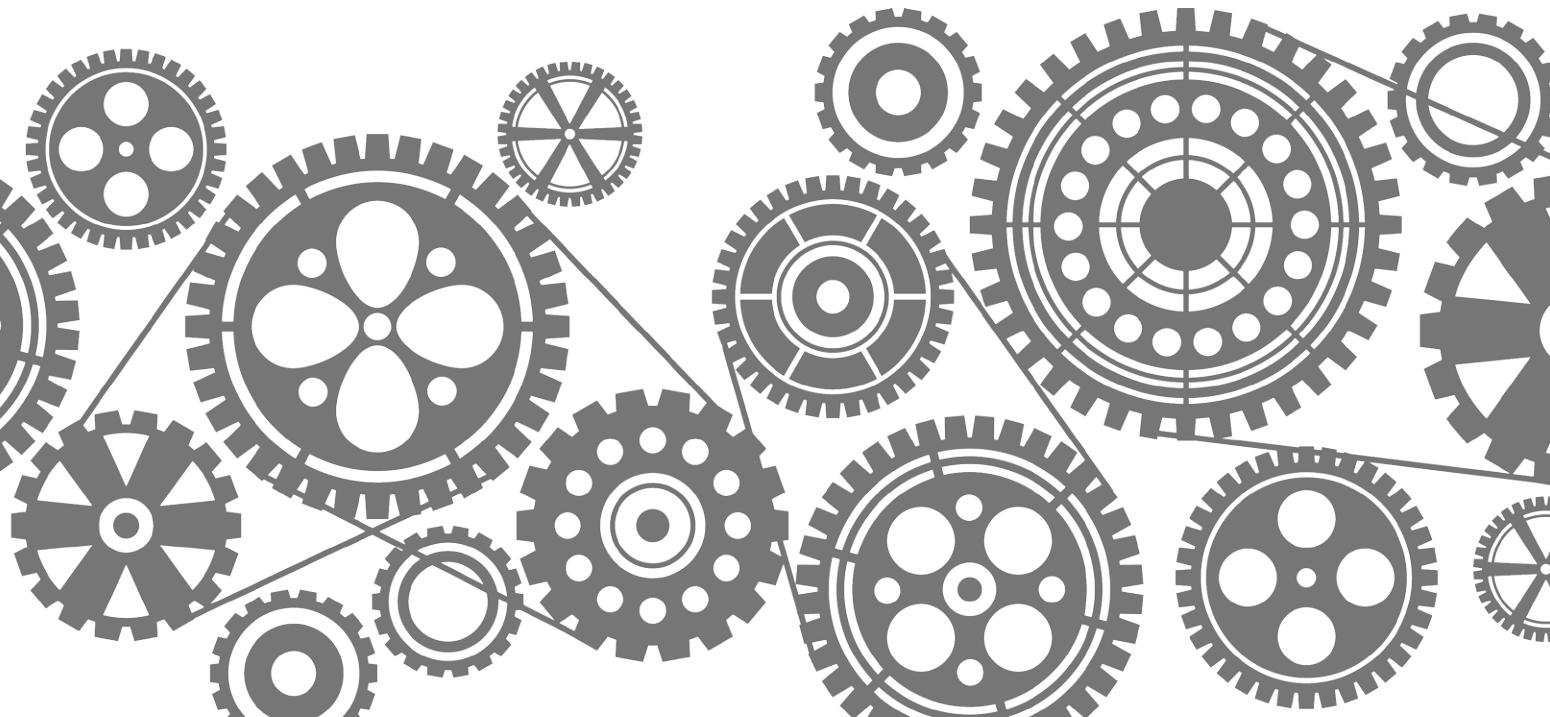
Este é o formato utilizado no BDD. De forma simples, rápida, sucinta e não-ambígua, foram escritos requisitos que atendem à necessidade de entendimento tanto por parte do cliente quanto por parte dos desenvolvedores, sem o *overhead* conceitual típico dos documentos de muitas páginas. Os detalhes não devem estar escritos nas histórias, pois elas servem para guiar o desenvolvimento dizendo *o que* deve ser feito em um nível suficientemente genérico. A partir disso, espera-se que o time ágil se organize para obter os detalhes complementares, o que é feito dependendo do contexto da empresa, utilizando meios relevantes ao ambiente e à cultura organizacional.

Ainda sobre a escrita de histórias, vale ressaltar que existem diferentes formas de fazê-la. Uma opção diferente e igualmente válida de escrever o mesmo requisito através de *User Stories* é apresentada a seguir:

User Story: Usuário cobra CPF

- O CPF pode ser alterado na edição de dados pessoais;
- O CPF deve ser validado;
- Quando o CPF é válido, o sistema deve informar que a atualização ocorreu com sucesso;
- Quando o CPF é inválido, o sistema deve informar que a atualização não ocorreu.

Neste formato, a história é apresentada por meio de um conjunto de critérios de aceitação organizados em tópicos. Quando todos os tópicos são atendidos, então a história está cumprida e a funcionalidade pode ser entregue para o usuário. Embora possa ser considerada uma forma mais natural de apresentar requisitos ao cliente do que a primeira (do BDD), o formato do BDD possui a vantagem de suportar automatização mais naturalmente.



Dominando os tipos de testes automatizados

Mas qual o ganho de automatizar uma *User Story*, afinal? Por que isto é importante?

A automatização de uma *User Story* garante que o teste seja capaz de validar o requisito do cliente de forma focada e assertiva. Quando um teste de aceitação falha, sua falha é relacionada ao requisito, e não a um detalhe de implementação, que é o que os testes unitários e de integração detectam. Trata-se de uma forma diferente de validação. Ao mesmo tempo, o uso dessa prática aumenta as chances de o desenvolvedor programar exatamente aquilo que o cliente espera, pois reduz possíveis falhas de comunicação. O BDD é a técnica comumente empregada nestes casos.

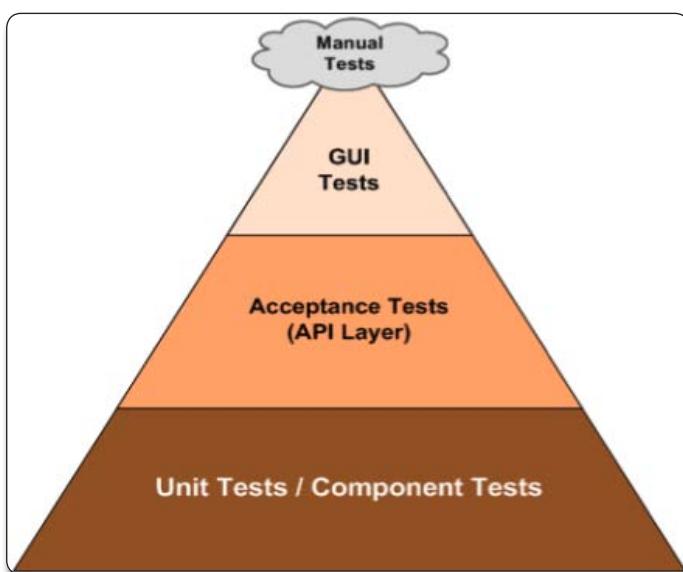


Figura 2. Pirâmide de automação de testes de Crispin e Gregory

Dan North, criador do BDD, em seu artigo *Introducing BDD*, explica como sua técnica evoluiu do TDD apoiada no DDD, consolidando-se como um tipo de representação de requisitos que pode ser automatizado com testes de aceitação. Esse mesmo autor, em outro artigo (*What's in a Story?*, também de leitura obrigatória àqueles que desejam se aprofundar nesta abordagem), apresenta o modelo de histórias do BDD aqui citado.

Na parte prática deste artigo, a ser publicada na próxima edição, será apresentado como as histórias são vinculadas aos testes automatizados pelo uso dos conceitos de BDD e do framework Concordion.

Quando e quanto utilizar?

Uma dúvida natural ao considerar os diferentes tipos de teste é: quando devo usar cada teste e quanto de cada teste devo utilizar? Para responder a esta pergunta, considere inicialmente a pirâmide da Figura 2.

Em relação às proporções de escrita de testes automatizados, esta imagem sugere que deve-se construir mais testes unitários e de componentes, seguidos pelos testes de aceitação e, em menor escala, testes funcionais. Isto porque os testes na base da pirâmide são mais simples e fáceis de implementar, e permitem que problemas sejam encontrados mais cedo durante o desenvolvimento. Além disso, considerando as características de ambos, entende-se que os testes unitários devem ser feitos em maior escala, pois são mais simples de se criar e mais rápidos de executar.

Em suma: a) testes unitários devem ser utilizados quando se deseja testar objetos sem dependência com recursos externos (no caso de haver objetos com tais dependências, as mesmas podem ser simuladas por meio de *Test Doubles*); b) testes de componentes ou integração devem ser utilizados quando se

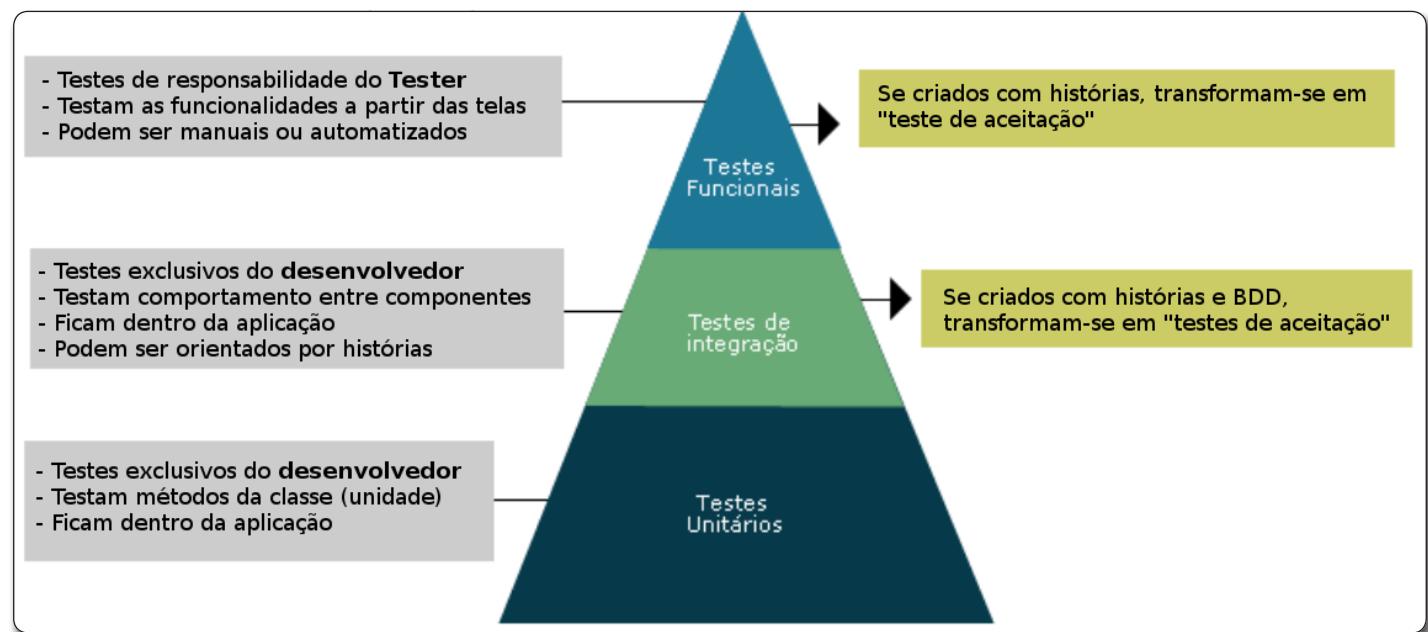


Figura 3. Pirâmide de testes simplificada para adequar-se aos conceitos deste artigo

deseja testar conjuntos de objetos que possuem dependências com o ambiente, como a relação entre uma rotina e o banco de dados, ou entre uma rotina e um web service; e c) testes de aceitação devem ser escritos quando se deseja validar se uma *User Story* está implementada corretamente.

Saiba que nenhum dos três tipos de teste exclui os demais, pois são complementares. E se a proporção aqui apresentada for seguida, a tendência é que se alcance uma suíte de testes confiável, pois esta cobrirá problemas em diferentes níveis, e eficaz, pois os ganhos observados na aplicação poderão justificar o tempo gasto na escrita dos testes.

Ao escolher o tipo de teste a implementar, também deve-se considerar o tempo investido para construir e executá-lo. Se houver pouco tempo para criar testes em um código muito acoplado, sugere-se aplicar o *Humble Object Pattern* como técnica de refatoração para isolar o código dependente de recursos externos daquele que pode ser executado sem tais dependências, e testar de forma unitária o que for possível. Caso haja maior oportunidade para melhoria, sugere-se, além de aplicar *Humble Object Pattern* e criar os testes unitários, investir na preparação de um teste de integração. Por fim, se houver histórias e tempo, sugere-se escrever também os testes de aceitação.

Para melhor descrever a relação entre testes automatizados apresentada neste artigo, foi elaborada uma segunda pirâmide, derivada do trabalho de Crispin e Gregory, pelo autor deste artigo (vide **Figura 3**).

Observe que os testes de aceitação (posicionados no meio da pirâmide) são um tipo especial de teste de integração, com a diferença de que o primeiro tipo é atrelado a histórias, enquanto o segundo não. Já no topo da pirâmide, observa-se que testes funcionais podem ser automatizados como testes de aceitação, por meio de ferramentas como Selenium. Nestes casos, o teste valida o sistema por meio do conceito de caixa preta, guiado por histórias. A validação funcional é feita pela navegação nas telas da aplicação, de forma que detalhes internos de implementação não são considerados.

Como visto até agora, diversas vantagens dos testes automatizados já foram apresentadas. Contudo, existem outras cuja menção faz-se importante. São elas:

- Testes automatizados de regressão trazem mais confiança ao realizar alterações;
- A automatização libera o tempo das pessoas para atuar em atividades menos repetitivas;
- Testes automatizados são mais rápidos (e baratos);
- Testes automatizados estão menos sujeitos a erros.

Automatização de testes em diferentes casos

Até aqui, os conceitos relacionados à automatização de testes apresentados foram analisados apenas na teoria. Neste tópico vamos começar a mudar essa perspectiva e analisar como os testes automatizados poderiam ser aplicados em dois cenários comuns do mundo real.

Testes automatizados em sistemas legados

Grande parte do trabalho de desenvolvimento reside na evolução de sistemas existentes, que muitas vezes não foram construídos com testes automatizados e nem mesmo consideraram as boas práticas da orientação a objetos. Nestes cenários, as seguintes práticas podem ser adotadas:

- Sempre que for alterar um código sem testes, lembre-se do *Humble Object Pattern*. Extraia um trecho de código de dentro de um método acoplado para uma classe ou método isolado (IDEs como IntelliJ e Eclipse fazem isso automaticamente) e escreva o teste unitário para ele. Vale ressaltar que no momento da extração do código você pode encontrar dificuldades por conta da dependência de atributos que recebem seu valor por injeção de dependência. Neste caso, considere a passagem de objetos por parâmetro para esta nova classe ou método sendo criado. Ainda na escrita de testes unitários, procure evitar o uso de mocks, o que é recomendado apenas em casos onde não é possível isolar o código;
- Se houver tempo, analise se é possível preparar uma massa de dados para o teste. Caso positivo, implemente o teste de integração;
- Caso a prática de User Stories exista e haja tempo disponível, implemente os testes de aceitação com BDD.

Testes automatizados em times ágeis

Times ágeis, atuando com métodos baseados em iterações (como Scrum), geralmente possuem autonomia para definir sua forma de trabalho e fazem uso de User Stories. Num cenário como esse, algumas ações podem ser realizadas visando alcançar um maior nível de qualidade no software em construção, a saber:

- Insista para que as histórias sejam escritas no formato do BDD (ou que a tradução para o formato do BDD seja tarefa do time durante a iteração). Dessa forma torna-se mais fácil adotar testes de aceitação automatizados, já que tais testes dependem da existência de histórias no formato do BDD;
- Insista para que a *definição de pronto* do time considere a criação de testes para cada história a ser entregue. Com isso, o time será capaz de planejar e estimar a escrita de testes durante a iteração. Os testes unitários devem ser adotados sempre que possível, e os testes de integração devem ser utilizados para testar as dependências mais importantes. Já os testes de aceitação vinculados às histórias devem ser criados com maior atenção aos cenários mais importantes para o cliente, de modo a aplicar o esforço do teste de forma eficaz.

Testes automatizados e a integração contínua

Uma prática comum em empresas que desenvolvem software é o uso de ferramentas para Integração Contínua. Estas ferramentas automatizam a geração de builds e a execução de testes automatizados em servidores específicos, integrando o código constantemente. A integração contínua é especialmente útil em projetos com muitos desenvolvedores, viabilizando um feedback rápido a todos os envolvidos sobre a saúde do código.

em construção. Neste ambiente, a execução dos testes unitários, de integração e de aceitação pode ser separada por motivos diversos, a saber:

- Como testes unitários são mais rápidos, podem ser executados a cada build. Já os testes de integração e execução são mais lentos, podendo ser agendados para rodar uma vez ao dia;
- O servidor de integração contínua pode não ter acesso aos recursos externos que os testes de integração dependem. Com o intuito de evitar erros identificados pelo fato do ambiente não estar preparado para o teste, entre outros motivos, a ferramenta pode “desativar” os testes de integração/aceitação.

A análise apresentada neste artigo foi idealizada com o intuito de aprimorar a prática de desenvolvimento de software nas empresas. Diretamente relacionada a ela, sabe-se que o entendimento dos diferentes tipos de teste é de grande utilidade para as tarefas do dia a dia, onde o tempo é escasso, mas a qualidade é fundamental. Essa distinção permite elaborar uma estratégia que possibilita montar uma suíte de testes adequada às necessidades do projeto, assim como pode ajudar àqueles que trabalham com desenvolvimento ágil a automatizar suas histórias e àqueles que lidam com integração contínua e visam a redução da lentidão de processos de build.

Autor



Daniel Medeiros de Assis

daniel.medeiros.assis@gmail.com
<https://br.linkedin.com/in/dmassis>



Mestrando em Engenharia de Software no IPT/USP. Possui mais de 15 anos de experiência em desenvolvimento de software, com especialização em tecnologias Web, design e qualidade de código, e processos ágeis de desenvolvimento.

No próximo artigo esta análise será apresentada na prática, por meio da criação de um projeto em Java que utilizará os três tipos de teste abordados neste artigo: unitário, de integração e de aceitação.

Links:

Artigo “Eradicating Non-Determinism in Tests”, de Martin Fowler.

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

Padrão “Humble Object”, do livro “xUnit Design Patterns”.

<http://xunitpatterns.com/Humble%20Object.html>

Livro “xUnit Test Patterns”.

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

Livro “Continuous Integration: Improving Software Quality and Reducing Risk”.

<http://www.amazon.com/Continuous-Integration-Improving-Software-Reducing/dp/0321336380>

Livro “Refactoring: Improving the Design of Existing Code”.

<http://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672>

Padrão de Refatoração “Extract Method”.

<http://refactoring.com/catalog/extractMethod.html>

Padrão de Refatoração “Extract Class”.

<http://www.refactoring.com/catalog/extractClass.html>

Você gostou deste artigo?

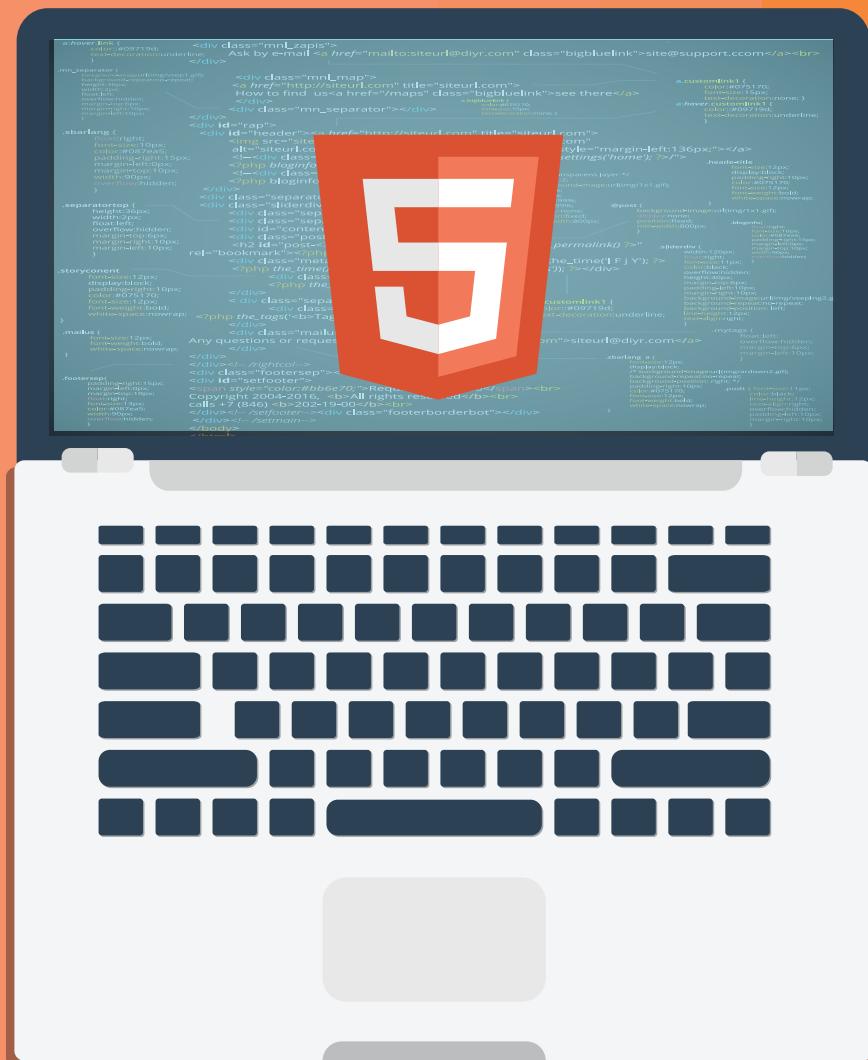
Dê seu voto em www.devmedia.com.br/javamagazine/feedback

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.



Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.



Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.



1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486