



Edição 152 :: R\$ 14,90

 DEVMEDIA

DESTAQUE:
Introdução ao Eclipse Che
Conheça a IDE que está redefinindo a forma
como desenvolvemos software

**Gerenciando bancos de dados
com Liquibase e JUnit**
**Controle os scripts do banco e
veja como realizar testes integrados**

IoT com Java 8 e Raspberry Pi

Desenvolva mais um protótipo para a Internet das Coisas



BOAS PRÁTICAS NO JSF

Como melhorar o desempenho de suas aplicações



MVP

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

APlique esse investimento
na sua carreira...

E MOSTRE AO MERCADO
QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

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

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

 **DEVMEDIA**



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 – Introdução ao Eclipse Che

[Pedro E. Cunha Brigatto]

Conceúdo sobre Novidades, Artigo no estilo Solução Completa

22 – Desenvolvimento IoT com Java 8 e Raspberry Pi

[Paulo M. D. Bordin]

Destaque – Boas Práticas

33 – Como melhorar o desempenho de suas aplicações JSF

[Ronaldo Ronie Nascimento]

Conceúdo sobre Boas Práticas

42 – Gerenciando e testando seu banco de dados com Liquibase e JUnit

[Rafael Campos Lima]

Programador Java: Por onde começar?

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



DEVMEDIA

Introdução ao Eclipse Che

Como a redefinição do conceito de workspace pode levar a um novo patamar de colaboração em torno do software

Durante os últimos anos, o que se nota no mercado de Tecnologia da Informação é o esforço para diminuir um vazio histórico entre as áreas de Operação e Desenvolvimento. Inúmeras tecnologias foram — e continuam a ser — anunciadas no sentido de viabilizar, tecnicamente, a realização desse objetivo.

Plataformas como Docker e Vagrant, por exemplo, estão cada dia mais sólidas e difundidas entre os profissionais de nossa área, e já cumprem o importante papel de tornar o provisionamento de recursos computacionais mais simples e gerenciável. Da mesma forma, os IDEs e as linguagens de programação, assim como ferramentas periféricas (como as de revisão de código, versionamento e gestão de projetos), seguem em marcha ascendente de evolução, adequando-se às transformações do mercado. Vivemos, indubitavelmente, um momento muito especial, em que a ânsia por maior produtividade e eficiência tem ditado um expressivo progresso em praticamente todas as disciplinas associadas à Engenharia de Software.

Entretanto, por mais que surjam poderosos arsenais tecnológicos para tornar o cotidiano do desenvolvimento e operação de software mais produtivo, o velho e conhecido problema da configuração de ambiente ainda é um dilema a ser superado. Não há dúvidas que as tecnologias de provisionamento de recursos já citadas simplificam e aceleram muito a configuração de ambientes, mas o fato é que essa ainda é uma atividade que, especialmente no dia a dia dos desenvolvedores, é exercida de forma muito individualizada. O popular discurso do “funciona na minha máquina” continua muito frequente, evidenciando que ainda há espaço para evoluirmos nesse campo.

Eclipse Che é uma plataforma que, embora semelhante a outras em inúmeros aspectos (tais como Eclipse Oomph ou Eclipse Flux, cujas referências se encontram na seção [Links](#)), diferencia-se por incluir os *runtimes* na composição de um *workspace*. Ao longo deste artigo, entenderemos o quanto essa característica em particular é fundamental, e aprenderemos, também, como instalar, configurar e utilizar essa plataforma, tanto em contexto local quanto remoto.

Fique por dentro

Desenvolvimento de software é uma ciência em constante evolução. Ao longo dos últimos anos, presenciamos transformações significativas nesse mercado, impulsionadas principalmente pelo fenômeno do DevOps. Nesse contexto, o Eclipse Che é uma plataforma que, embora incipiente, apresenta um enorme potencial para redefinir o conceito de colaboração em projetos de software, englobando não apenas equipes técnicas, mas todos os demais profissionais envolvidos. Neste artigo, abordaremos os fundamentos desse novo IDE e sua arquitetura, e avaliaremos, por meio de um tutorial, os principais recursos por ela oferecidos.

A história por trás do Eclipse Che

A origem da jornada, que culminou na ideia e construção do Eclipse Che, está na iniciativa de um cientista da computação chamado Tyler Jewell, motivado especialmente por uma inquietação pessoal. Em suas apresentações em eventos como o EclipseCon (veja referências na seção [Links](#)), Jewell relata que, entre 2003 e 2010, alcançara uma posição de destaque em sua carreira, inicialmente na área técnica e, em seguida, gerencial. Entretanto, precisou se afastar de suas atividades profissionais durante nove meses por motivos de saúde.

Ao recuperar-se, sentiu o desejo de resgatar o período em que atuava exclusivamente na área técnica. Frustrou-se, segundo ele, ao não ser capaz de, após uma semana de trabalho, montar um ambiente relativamente simples, composto por um projeto web e um servidor de aplicações. Esse sentimento foi suficiente para provocar nele o pensamento de que, talvez, os principais problemas associados à colaboração em projetos de software fossem resolvidos caso qualquer membro, não necessariamente técnico, conseguisse dar sua parcela de contribuição sem precisar realizar qualquer setup de ambiente por conta própria.

O próximo passo, no raciocínio de Jewell, foi buscar a resposta para o que seria o conjunto mínimo de elementos para que uma colaboração ocorra. Ao final desse exercício, a lista era composta por: um IDE, arquivos do projeto em questão e, naturalmente, ambientes que permitissem validar a contribuição em si. Outro ponto importante é que todos esses itens, combinados no contexto de uma colaboração, deveriam ser entendidos como o

workspace em si, cujas características e estado fossem armazenáveis e controláveis para, em seguida, ficarem acessíveis aos demais membros do projeto.

O caminho seguido por Tyler apontava requisitos que já eram cobertos por algumas plataformas existentes àquela época, como o Eclipse Oomph ou o Eclipse Flux (veja a seção [Links](#)). Entretanto, havia algo de muito inovador em seu conceito, que tornava essas mesmas soluções citadas incompletas. Ainda que todas apresentassem propostas convincentes para atividades como sincronização e compartilhamento de workspaces, nenhuma considerava *runtimes* como integrante do workspace em si. Portanto, embora workspaces já pudessem ser compartilhados, a complexidade de setup de ambientes ainda era delegada a cada colaborador.

Eclipse Che surgiu, portanto, como uma proposta de tornar workspaces realmente universais, extinguindo as barreiras naturais de desenvolvimento e operação de projetos de software com as quais convivemos até os dias de hoje. Ao longo das próximas seções deste artigo, entenderemos melhor esse conceito de universalidade e o modo com que ele é colocado em prática na arquitetura e no uso da plataforma em si.

Antes de encerrarmos esta seção, no entanto, acabaremos com a curiosidade que, provavelmente, perdura na cabeça da maioria dos leitores até aqui: afinal, de onde vem o nome **Che**? Antes de darmos asas à nossa imaginação, antecipamos que a razão é bastante simples e não carrega consigo nada de lúdico: trata-se, apenas, de uma referência à cidade de Cherkasy, na Ucrânia, na qual boa parte do desenvolvimento da plataforma é realizada.

O conceito do workspace universal

Conforme vimos na seção anterior, o aspecto inovador do Eclipse Che está no modo como ele define o workspace. Sua universalidade, citada por Tyler Jewell, materializa-se essencialmente através dos pontos listados a seguir:

- **Runtime embutido:** ambientes usados para executar o projeto são embutidos no

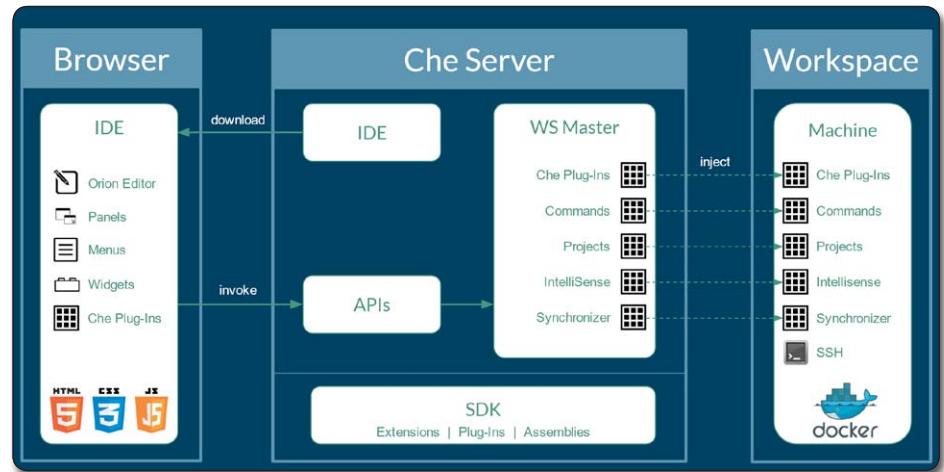


Figura 1. Diagrama com arquitetura do Eclipse Che – Fonte: Eclipse

workspace, retirando do colaborador a responsabilidade de configurá-los;

- **Colaborativo:** Che é, sobretudo, um servidor de workspaces. Ele não apenas permite que armazenemos e gerenciamos workspaces em servidores, mas possibilita seu compartilhamento, dando a outros colaboradores a chance de utilizá-los para fins como desenvolvimento, demonstração (*live demos*), revisão de código, entre outros;

- **Programável:** tanto o servidor quanto cada workspace possui a sua própria API RESTful, empregada tanto no uso quanto na administração de cada componente da plataforma. Geradas a partir de um framework extremamente popular chamado Swagger, as APIs RESTful são o meio pelo qual são invocados os inúmeros microsserviços implantados no servidor;

- **Versionável:** a partir do uso de tecnologias como Docker, a configuração de ambientes se torna declarativa, por meio de documentos denominados *Dockerfile*. Como o conteúdo desses arquivos é puro texto, versioná-los torna-se uma tarefa simples, habilitando-nos não somente a controlar a evolução do produto em si, mas também o histórico de todos os ambientes usados desde o desenvolvimento até a homologação de todas as versões do produto;

- **Extensível:** o SDK do Eclipse Che nos possibilita escrever extensões (plug-ins) tanto para o IDE quanto para o servidor ou, ainda, workspaces. Podemos, portanto, desenvolver tanto agentes que atuam diretamente no gerenciamento de work-

spaces e projetos quanto novas janelas e perspectivas para enriquecer a experiência de usuários por meio do IDE. O desenvolvimento de extensões não será, entretanto, abordado ao longo do texto;

Alguns termos que acabamos de usar podem, neste momento, soar um tanto confusos, mas é natural que seja assim. Na próxima seção, estudaremos as principais características da arquitetura da plataforma e, então, todos esses conceitos serão devidamente apresentados e contextualizados, fechando o raciocínio em torno de seus significados e importância.

A arquitetura do Eclipse Che

A arquitetura do Eclipse Che se baseia em três entidades fundamentais, sendo elas:

- **Browser** (ou cliente), que apresenta o IDE e se utiliza da API oferecida pelo servidor;
- **Servidor**, que hospeda a parte servidora do IDE e outras aplicações web (tais como o Dashboard), além de administrar os workspaces;
- **Workspaces**, que são os diversos espaços de trabalho existentes no domínio.

Cada um desses elementos citados, ilustrados na **Figura 1**, será brevemente introduzido nos tópicos seguintes.

Browser

O IDE do Eclipse Che é, em linhas gerais, uma aplicação web projetada para se comportar da mesma forma que um

IDE padrão de mercado, tanto do ponto de vista de seu uso quanto de sua extensibilidade. A ideia é que a experiência padrão de colaboração seja preservada (a partir de recursos como assistência à criação de código, integração com as variadas ferramentas usadas cotidianamente em desenvolvimento de sistemas, entre outros) e a criação de extensões (visuais ou não) não fuja, também, da experiência oferecida por tecnologias como o RCP (*Rich Client Platform*), empregado em versões padrão do IDE Eclipse.

No entanto, pela própria natureza do Che, o SDK precisava ser projetado sobre um framework que suportasse a geração de API REST e, também, aplicações cliente (em JavaScript) para serem executadas no contexto de navegadores web (como o Dashboard e o próprio IDE, que serão detalhados adiante). Esse requisito levou a equipe responsável pela concepção do Eclipse Che a adotar um framework chamado GWT (*Google Web Toolkit*), que oferece uma boa compatibilidade com praticamente todas as máquinas de renderização disponíveis no mercado e um modelo de desenvolvimento relativamente simples.

Com relação à edição de conteúdo e suporte a temas, o Eclipse Che baseia-se em outro projeto open source, muito interessante, chamado Orion, que permite a entrega de uma experiência muito fluida na edição de arquivos em geral.

Inúmeras funcionalidades que estamos acostumados a ver em IDEs muito populares (como NetBeans, IntelliJ IDEA ou, ainda, o próprio Eclipse, em suas versões standalone), estão presentes também no IDE do Che: *code assistance (auto-complete, syntax highlighting, detecção de erros de compilação, mensagens de warning)*, integração com ferramentas periféricas, como Maven, Git e Subversion, entre tantas outras. Todos esses recursos são disponibilizados através de microserviços implantados no servidor Che, acessados por meio de uma API RESTful e injetados, sob demanda, nos workspaces, sob a forma do que se denominou *serviços ou agentes*.

Há, também, perspectivas tanto para desenvolvimento quanto para operações, sendo possível gerenciar não apenas o projeto em si, mas máquinas e workspaces em um nível de granularidade bastante razoável. Podemos, por exemplo, editar receitas de máquinas (*containers Docker*, normalmente), bem como manipular o estado de cada workspace (parando ou iniciando suas execuções), além de trabalhar diretamente dentro de cada máquina, a partir de terminais de comando.

A aplicação em si, do lado cliente (navegadores web), é baixada como uma *single-page app*, e seu conteúdo engloba tecnologias como CSS3, HTML5 e JavaScript, com altíssimo grau de compatibilidade com praticamente todos os clientes existentes no mercado.

Servidor Che

O servidor Che, componente central de toda a arquitetura, é constituído de quatro elementos, sendo eles: uma API, o Workspace Master, o IDE e o SDK. Os principais papéis desempenhados por ele são:

- Hospedar e administrar o IDE: toda a coordenação de eventos e a definição do conteúdo a ser exibido em clientes — como navegadores web — será realizada dentro do contexto do servidor Che;
- Permitir que, a partir de sua API, atividades de operação e desenvolvimento sejam desempenhadas através, por exemplo, do IDE, tais como: criação, configuração e exportação de workspaces, criação/importação/compilação/execução de projetos, gerenciamento de máquinas via terminais de comando (SSH), entre outras. A comunicação do IDE com o servidor, por exemplo, se dá tipicamente como uma sequência de chamadas a essa API, as quais normalmente estarão associadas a modificações que um usuário deseja realizar em um projeto ou no workspace, diretamente;
- Controlar todo o ciclo de vida dos workspaces a partir de uma unidade extremamente importante, intitulada *Workspace master*. Esse controle envolve a injeção e remoção de recursos (projetos, plug-ins, comandos, serviços como *JDT Intellisense* e outros) em cada workspace sempre que necessário, além da sincronização de arquivos modificados no contexto dos workspaces, garantindo que o estado atual de um workspace seja sempre persistido para usos posteriores;

O SDK oferecido pelo servidor é um conjunto de bibliotecas que podem ser usadas para desenvolver extensões (leia-se plug-ins) nos três níveis: IDE propriamente dita, servidor ou workspaces. Extensões podem, ainda, ser agrupadas e distribuídas em pacotes customizados da plataforma, formato que foi denominado, na documentação, de *assemblies*.

Workspaces

Workspaces são unidades compostas por zero ou mais projetos, zero ou mais comandos e, no mínimo, um ambiente (usado para desenvolvimento). Todo ambiente, por sua vez, é composto por unidades denominadas *máquinas*, cuja implementação padrão se dá a partir de *containers*, usando como tecnologia base o Docker. Máquinas, por fim, são o runtime propriamente dito, e é nelas que aplicações são efetivamente executadas. Sua configuração se dá por meio de *receitas*, que são documentos no formato *Dockerfile* e que podem ser editadas a partir do IDE do Eclipse Che.

A cada ambiente, por definição, sempre haverá no mínimo uma máquina associada, e outras poderão ser criadas sob demanda. Podemos, por exemplo, criar máquinas que replicam as características de ambientes de produção ou QA (*Quality Assurance*), ou, ainda, máquinas para hospedar, isoladamente, sistemas periféricos à aplicação em si, como sistemas gerenciadores de bancos de dados.

A orquestração e o gerenciamento de workspaces — e de todos os seus componentes, introduzidos brevemente nos parágrafos anteriores — são realizados pelo servidor Che por meio do Workspace Master. É ele que, de modo contínuo, garante que tanto a injeção e remoção de serviços quanto a sincronização de todas as modificações realizadas por usuários em um projeto sejam continuamente processadas e nada se perca ao longo dessa experiência. Essa comunicação ocorre a partir de um agente

injetado pelo servidor Che em todo workspace assim que esse workspace é iniciado.

Agora que já vimos um pouco sobre a organização interna da plataforma, é chegado o momento de experimentá-la na prática. Iniciaremos com uma instalação local com o objetivo de explorar um pouco os seus recursos para, em seguida, navegarmos um pouco pela solução comercial oferecida pela empresa Codenvy, da qual Tyler Jewell é fundador e CEO. Por fim, encerraremos o artigo com nossas conclusões a respeito da plataforma e da experiência descrita ao longo do tutorial.

Instalando o Eclipse Che em uma máquina local

A instalação do Eclipse Che começa pelo cumprimento de alguns pré-requisitos. O primeiro deles é o Java 8, sendo suficiente o JRE. Além disso, essa plataforma depende do Docker e da Docker Toolbox para operar, sendo ambos usados no gerenciamento e execução dos containers — “máquinas” — geradas pelo Che. Outro ponto ao qual o leitor deve estar atento diz respeito à habilitação dos recursos de virtualização do computador no qual o Eclipse Che será instalado. Essa é uma configuração do BIOS, e maiores detalhes acerca do procedimento podem ser encontrados na página oficial do Che (cuja referência se encontra na seção [Links](#)).

O Eclipse Che é compatível com os principais sistemas operacionais: Windows, Mac OS e Linux. Para o primeiro deles, Windows, há um instalador do tipo *all-in-one*, que já se encarrega de instalar todos os pré-requisitos enunciados de forma automática, simplificando consideravelmente a instalação por meio de um assistente. Para os outros dois sistemas operacionais, no momento em que este artigo foi escrito, essa configuração era manual. No entanto, no site oficial do projeto já havia uma mensagem indicando que instaladores *all-in-one* também estavam a caminho, em breve para Mac OS (10.9+).

Há, entretanto, outras formas de se configurar a plataforma, por meio de imagens pré-configuradas que podem ser montadas via Vagrant. Recomendamos, portanto, que o leitor visite e página de downloads do projeto (veja a seção [Links](#)), pois há muita informação interessante por lá.

Neste tutorial, usaremos a versão disponível para Windows, que se encontra disponível na página de downloads do projeto. Trata-se de um arquivo executável de aproximadamente 397 MB que, quando aberto, inicia o assistente de instalação já mencionado. A primeira tela exibida por esse assistente é a que vemos na [Figura 2](#). O fluxo de instalação é bastante trivial e, portanto, omitiremos maiores detalhes a seu respeito.

Configurando o acesso a uma conta GitHub via OAuth

Para estudarmos o Eclipse Che, ao longo deste artigo usaremos como base um projeto pré-existente, hospedado em uma conta no GitHub (veja a seção [Links](#)). Isso exigirá que esse projeto seja, inicialmente, clonado no workspace do Che para, então, ser manipulado. Para viabilizar essa experiência, em uma instalação local como a que estamos realizando, será necessário configurar tanto a plataforma quanto o GitHub em si, de modo que:

- A conta do GitHub identifique e aceite requisições cuja origem é o nosso servidor Che;
- O Eclipse Che conheça a aplicação OAuth instalada no GitHub, para invocá-la.

Essas são, basicamente, as atividades que veremos a partir de agora. Recomendamos que, ao seguir o conteúdo das próximas seções, o leitor utilize um projeto de sua própria autoria, em uma conta pessoal no GitHub, para garantir uma total autonomia de ajuste tanto das configurações do projeto em si quanto na manipulação do projeto selecionado para aplicar a este tutorial.

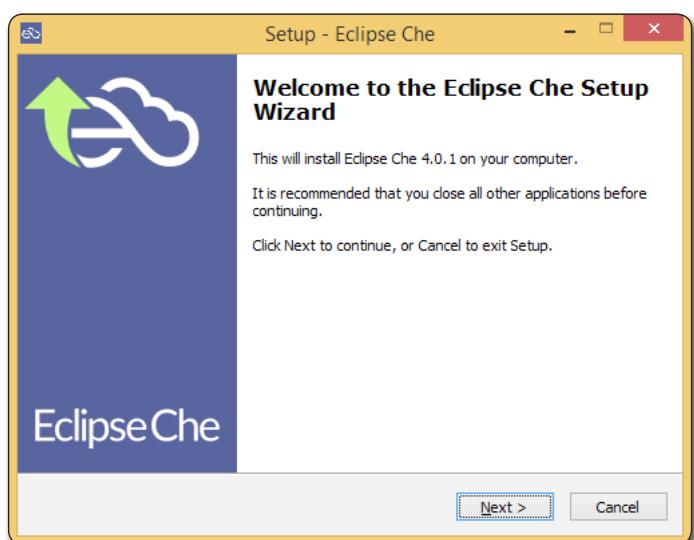


Figura 2. Primeira tela do assistente de instalação do Eclipse Che para Windows

Configuração de uma aplicação OAuth no GitHub

Para que o Eclipse Che se comunique adequadamente com o GitHub, precisamos, inicialmente, configurar uma nova aplicação OAuth em nossa conta. Essa configuração é realizada em *Settings > OAuth applications*. Na página que se abre, precisamos selecionar a aba *Developer applications* e clicar no botão *Register new application*, no canto superior direito do painel exibido.

No formulário que se abre, ilustrado na [Figura 3](#), preenchemos todos os campos de acordo com o conteúdo da [Tabela 1](#). Quando submetemos esses dados para o GitHub, clicando no botão *Register application*, serão geradas, enfim, as informações que nos interessarão para passarmos para a segunda etapa da configuração.

Campo do formulário	Valor do campo
Application name	DevmediaChe
Homepage URL	http://localhost/chevaadinapp
Application description	Aplicação de exemplo para exercitar a comunicação entre o Eclipse Che e o GitHub
Authorization callback URL	http://localhost:8080/ide/api/oauth/callback

Tabela 1. Dados da aplicação OAuth para comunicação entre o GitHub e o Eclipse Che

Introdução ao Eclipse Che

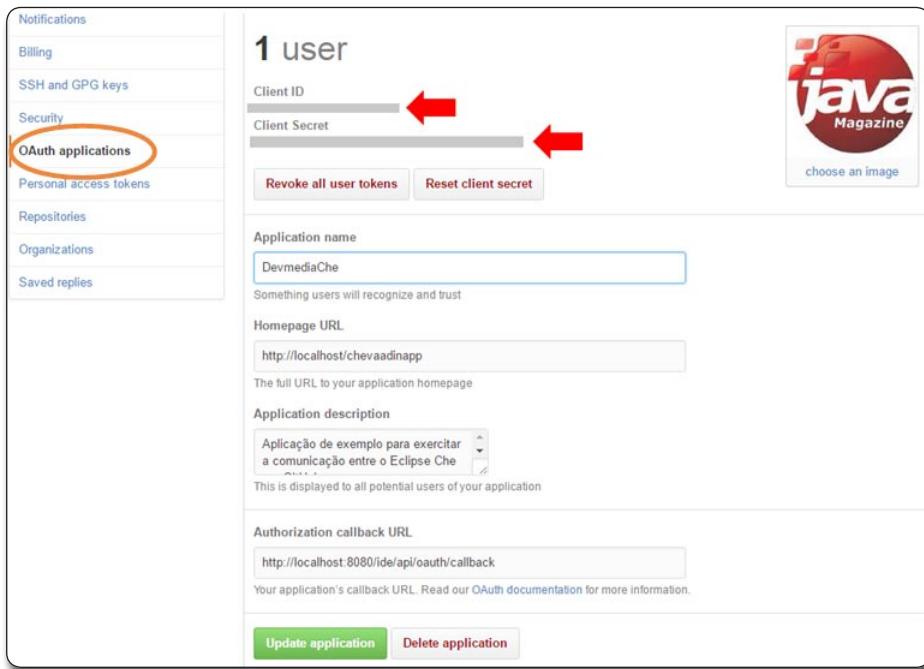


Figura 3. Configuração do GitHub para se comunicar com o nosso workspace no Eclipse Che

Listagem 1. Configuração do Eclipse Che para comunicação remota com uma conta do GitHub.

```
...
### OAuth configuration. You can setup GitHub OAuth to automate authentication to remote
### repositories. You need to first register this application with GitHub OAuth.
# GitHub application client ID
oauth.github.clientid=86b1xxxxxxxxxe2fd17
oauth.github.clientsecret=d015722b8xxxxxxxxxx9b4d5cf026f1
oauth.github.authuri= https://github.com/login/oauth/authorize
oauth.github.tokenuri= https://github.com/login/oauth/access_token
oauth.github.redirecturi= http://localhost:${SERVER_PORT}/ide/api/oauth/callback
auth.oauth.access_denied_error_page=/error-oauth
...
```

Listagem 2. Análise do diretório raiz de instalação do Eclipse Che.

```
C:\Personal\DevEnv\EclipseChe\eclipse-che-4.0.0-RC13>echo %cd%
C:\Personal\DevEnv\EclipseChe\eclipse-che-4.0.0-RC13

C:\Personal\DevEnv\EclipseChe\eclipse-che-4.0.0-RC13>cd bin

C:\Personal\DevEnv\EclipseChe\eclipse-che-4.0.0-RC13\bin>dir
O volume na unidade C é Sistema
O Número de Série do Volume é 92B9-41C1

Pasta de C:\Personal\DevEnv\EclipseChe\eclipse-che-4.0.0-RC13\bin

10/04/2016 00:54 <DIR> .
10/04/2016 00:54 <DIR> ..
29/03/2016 20:59 817 che-install-plugin.bat
29/03/2016 20:59 11.811 che-install-plugin.sh
29/03/2016 20:59 802 che.bat
29/03/2016 20:59 27.162 che.sh
4 arquivo(s) 40.592 bytes
2 pasta(s) 929.462.071.296 bytes disponíveis

C:\Personal\DevEnv\EclipseChe\eclipse-che-4.0.0-RC13\bin>
```

Esses dados são uma chave secreta (*Client secret*) e um ID de cliente (*Client ID*), que, na próxima seção, serão usados para completar o setup.

Configuração do Eclipse Che para comunicação com o GitHub

O próximo passo será completar a configuração do acesso, pelo Eclipse Che, ao repositório de código. Para isso, precisamos editar um arquivo de configuração da plataforma, chamado *che.properties*, localizado dentro da pasta *conf* do diretório raiz da plataforma. A porção que nos interessa desse arquivo foi reunida na **Listagem 1**.

Os dados que, efetivamente, mudam nesse arquivo de configuração são apenas a chave secreta e o ID do cliente. Todos os outros podem continuar com os valores originais. Preenchidas as informações, precisamos não apenas salvar o arquivo, mas garantir que o Che seja iniciado com essas novas configurações. Caso ele já estivesse em execução, por exemplo, seria necessário reiniciá-lo para que essa configuração fosse aplicada.

Executando o Eclipse Che localmente

Com o Eclipse Che instalado e sua comunicação com o GitHub configurada, chega o momento de colocá-lo em execução. A única ressalva a ser feita é que essa inicialização só ocorrerá caso o JRE reconhecido no ambiente seja qualquer distribuição na versão 8. Certifique-se, portanto, de ter editado a variável *JAVA_HOME* corretamente, pois é a partir dela que essa informação é obtida.

A inicialização da plataforma se dá a partir da execução de um script, localizado na pasta *bin* no diretório raiz de instalação (vide **Listagem 2**). O nome desse arquivo é *che*, e há disponíveis uma versão para Windows (*che.bat*) e outra para Mac OS e Linux (*che.sh*).

Como nosso tutorial se baseia em um ambiente Windows, executamos o arquivo *.bat*. Ao fazê-lo, uma série de mensagens é exibida no console, das quais destacamos as principais, reunindo-as na **Listagem 3**. Note que, inicialmente, é criada uma máquina virtual sobre o Oracle VM

Listagem 3. As principais mensagens que devemos entender sobre a inicialização do Eclipse Che.

```
C:\Personal\DevEnv\EclipseChe\eclipse-che-4.2.0\bin>che.bat
...
Docker machine named che already exists...
Setting environment variables for machine che...
Docker is configured to use docker-machine named che with IP 192.168.99.100...

##### HOW TO CONNECT YOUR CHE CLIENT #####
After Che server has booted, you can connect your clients by:
1. Open browser to http://localhost:8080, or:
2. Open native chromium app.
#####

2016-05-03 17:23:23,421[main]      [INFO ] [o.a.c.core.StandardService 435]  -
Starting service Catalina
2016-05-03 17:23:23,421[main]      [INFO ] [o.a.c.core.StandardEngine 259]  -
Starting Servlet Engine: Apache Tomcat/8.0.32
2016-05-03 17:23:23,478[ost-startStop-1] [INFO ] [o.a.c.startup.HostConfig 910]
- Deploying web application archive C:\Personal\DevEnv\EclipseChe\eclipse-che-4.2.0\tomcat\webapps\dashboard.war
2016-05-03 17:23:23,841[ost-startStop-1] [INFO ] [o.a.c.startup.HostConfig 974]
Deployment of web application archive C:\Personal\DevEnv\EclipseChe\eclipse-che-4.2.0\tomcat\webapps\dashboard.war has finished in 362 ms
2016-05-03 17:23:23,846[ost-startStop-1] [INFO ] [o.a.c.startup.HostConfig 910]
- Deploying web application archive C:\Personal\DevEnv\EclipseChe\eclipse-che-4.2.0\tomcat\webapps\ide.war
2016-05-03 17:23:29,240[ost-startStop-1] [WARN ] [p.DockerExtConfBindingProvider 51]
-DockerExtConfBindingProvider
```



```
2016-05-03 17:23:31,070[ost-startStop-1] [INFO ] [.c.p.d.c.DockerRegistryChecker 43]
- Probing registry 'http://localhost:5000'
2016-05-03 17:23:32,084[ost-startStop-1] [WARN ] [.c.p.d.c.DockerRegistryChecker 50]
- Docker registry http://localhost:5000 is not available, which means that you won't be able to save snapshots of your workspaces.
How to configure registry?
    Local registry -> https://docs.docker.com/registry/
    Remote registry -> set up docker.registry.auth.* properties
2016-05-03 17:23:32,438[ost-startStop-1] [INFO ] [o.a.c.startup.HostConfig 974]
- Deployment of web application archive C:\Personal\DevEnv\EclipseChe\eclipse-che-4.2.0\tomcat\webapps\ide.war has finished in 8,592 ms
2016-05-03 17:23:32,440[ost-startStop-1] [INFO ] [o.a.c.startup.HostConfig 910]
- Deploying web application archive C:\Personal\DevEnv\EclipseChe\eclipse-che-4.2.0\tomcat\webapps\swagger.war
2016-05-03 17:23:32,556[ost-startStop-1] [INFO ] [o.a.c.startup.HostConfig 974]
Deployment of web application archive C:\Personal\DevEnv\EclipseChe\eclipse-che-4.2.0\tomcat\webapps\swagger.war has finished in 116 ms
2016-05-03 17:23:32,558[ost-startStop-1] [INFO ] [o.a.c.startup.HostConfig 1030]
- Deploying web application directory C:\Personal\DevEnv\EclipseChe\eclipse-che-4.2.0\tomcat\webapps\ROOT
2016-05-03 17:23:32,659[ost-startStop-1] [INFO ] [o.a.c.startup.HostConfig 1142]
- Deployment of web application directory C:\Personal\DevEnv\EclipseChe\eclipse-che-4.2.0\tomcat\webapps\ROOT has finished in 101 ms
2016-05-03 17:23:32,665[main]      [INFO ] [o.a.c.http11.Http11NioProtocol 470]
- Starting ProtocolHandler ("http-nio-8080")
2016-05-03 17:23:32,676[main]      [INFO ] [o.a.catalina.startup.Catalina 642]
- Server startup in 9300 ms
```

VirtualBox (instalada a partir da *Docker Toolbox*), rodando Linux, cujos detalhes foram capturados e exibidos na **Figura 4**. Em seguida, é instalado o *Docker* nessa VM, pois essa será a plataforma usada pelo Che para o provisionamento de máquinas.

Logo após, um container Docker é criado e, nele, iniciada uma instância do Tomcat (um web container bastante popular) para hospedar as aplicações web da plataforma, como: Dashboard, IDE e Swagger (esse último responsável por disponibilizar a API RESTful já citada em seções anteriores).

Assim que o Che entra em operação, como as próprias mensagens da **Listagem 3** indicam, tanto o Dashboard quanto o IDE já se encontram acessíveis pelo endereço <https://localhost:8080>. O Dashboard, aplicação para a qual somos inicialmente redirecionados, está ilustrado na **Figura 5**. É a partir dele que podemos criar, excluir e administrar workspaces e projetos.

Esses workspaces e projetos, uma vez criados, podem ser abertos no IDE do Che. Enquanto o Dashboard se destina mais ao gerenciamento dos workspaces e projetos, o IDE será a ferramenta usada para o desenvolvimento e configuração desses

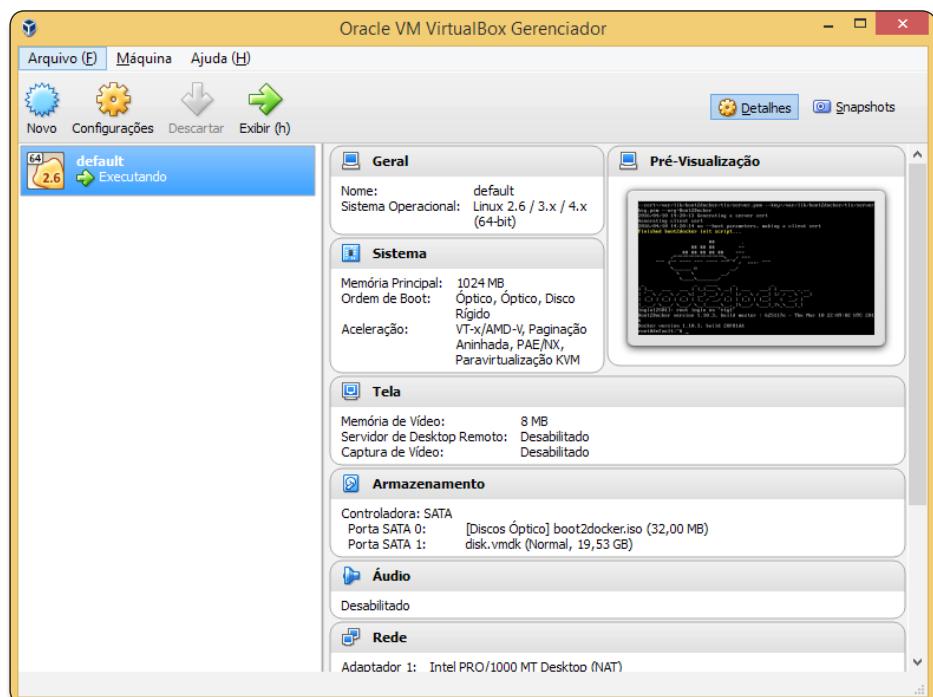


Figura 4. Oracle VM Virtual Box exibindo a máquina virtual criada para hospedar o Eclipse Che

componentes. Ao longo da próxima seção, experimentaremos cada uma dessas aplicações, tornando um pouco mais evidente as operações que cada uma delas oferece para o usuário.

Criando nosso workspace e nosso projeto no Eclipse Che

O Dashboard e o IDE do Eclipse Che, como já comentado em seção anterior, são aplicações web compostas de duas partes:

Introdução ao Eclipse Che

a primeira, hospedada no servidor, e a segunda, uma aplicação do tipo *single-page app*, baixada e executada no contexto de navegadores web. Essa última é gerada a partir do compilador GWT, que, através de um mecanismo conhecido como *deferred binding* (veja detalhes no **BOX 1**), constrói a aplicação de forma otimizada,

desenhada especificamente para o cliente que a executará.

O Dashboard do Eclipse Che organiza seus principais elementos através de um menu lateral, à esquerda. A partir dessa primeira tela, podemos criar e gerenciar o ciclo de vida de workspaces, bem como criar projetos dentro desses workspaces.

BOX 1. GWT e o mecanismo de Deferred Binding

Deferred binding é uma das principais características que colocam o GWT como uma boa escolha em contextos como o do Eclipse Che. O pacote final a ser baixado pelo navegador web, além de mais compacto, contém uma lógica otimizada para explorar a máquina de renderização do navegador-alvo de uma forma bem mais eficiente. Apenas para efeito de ilustração, pense que o GWT, por meio desse mecanismo, é capaz de gerar versões diferentes de uma mesma aplicação para navegadores diferentes. Mais que isso, é possível que versões distintas sejam geradas até em casos em que o navegador web é o mesmo, mas suas configurações (como idioma) são diferentes entre si.

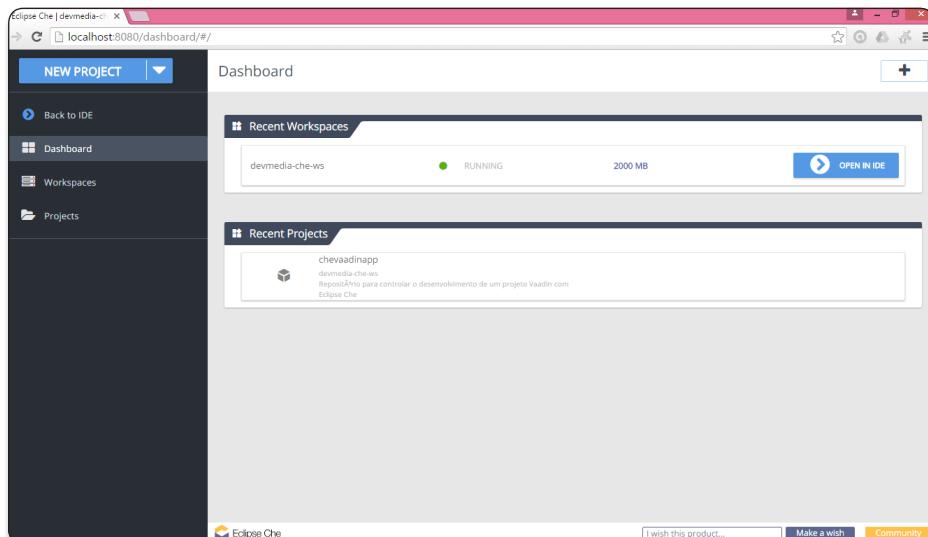


Figura 5. Dashboard do Eclipse Che, acessado via web browser

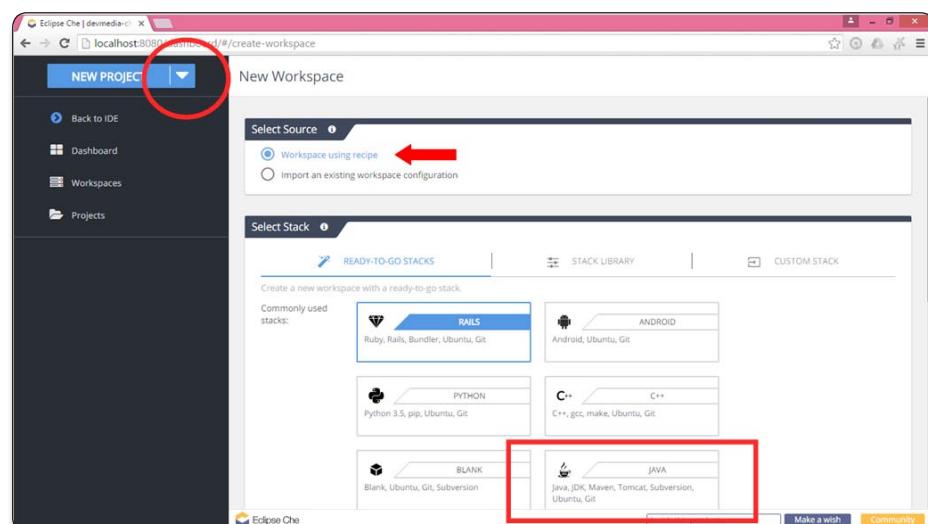


Figura 6. A criação de workspaces no Eclipse Che

Para começar, criaremos o nosso primeiro workspace passo a passo.

Nosso primeiro workspace

Há duas formas de se criar workspaces. Em ambas, é necessário que, primeiramente, cliquemos no item *Workspaces*, no menu lateral do Dashboard. Quando fazemos isso, no painel à direita (**Figura 5**), é exibida uma lista com todos os workspaces criados por nós até o momento. Observe que, no canto superior direito desse painel, há um botão intitulado "+", que, ao ser clicado, apresenta o formulário de criação de workspaces ilustrado na **Figura 6**. Essa é, portanto, a primeira forma.

A segunda, um pouco menos intuitiva, é através de um botão no canto superior esquerdo do Dashboard, ilustrado na **Figura 6**, que contém o texto *NEW PROJECT*. Ao clicarmos no símbolo de seta, circulado na figura, abre-se uma caixa de seleção com apenas um item: *NEW WORKSPACE*. Um clique nesse item é suficiente para que o mesmo formulário que acabamos de citar seja apresentado.

Workspaces podem ser criados, como vemos nesse formulário, a partir de uma receita ou importados a partir de uma configuração pré-existente. Ao usarmos receitas, podemos nos beneficiar de uma lista pré-existente, oferecida pela plataforma, que cobre basicamente todos os casos mais típicos de ambientes — *stacks* — usados nos dias de hoje.

Essas receitas são, na prática, arquivos Dockerfile que reúnem toda a stack de software necessária para que projetos de uma determinada natureza possam ser não apenas desenvolvidos, mas testados e gerenciados. Em nosso tutorial, usaremos uma receita chamada **JAVA**, destacada em um retângulo vermelho na **Figura 6**, que contém um ambiente composto pelas seguintes tecnologias/plataformas/serviços: Apache Maven, Apache Tomcat, JDK 8, Subversion, Git; tudo rodando sobre o sistema operacional Ubuntu.

Embora o uso de stacks pré-configuradas acelere significativamente nosso trabalho, nada impede que editemos nossa própria receita. O Eclipse Che, através de seu IDE, oferece um editor para que escrevamos

nossos próprios documentos *Dockerfile* e, dessa forma, configuremos o exato ambiente que precisamos.

Além da definição de um *stack*, a criação de um workspace passa pela definição da quantidade de memória a ser alocada para ele. Em nosso tutorial, definimos uma quantidade de 2 GB. Por fim, precisamos dar um nome ao workspace e, neste caso, o escolhido foi *devmedia-che-ws*. Preenchidos todos os campos, basta que cliquemos no botão *CREATE WORKSPACE*, aguardando o fim do processo.

Ao ser criado, um workspace ainda não se encontra em execução. Precisamos, manualmente, iniciá-lo. Fazemos isso selecionando o workspace em questão e, em seguida, clicando no botão *RUN*. Já com o status *RUNNING*, como podemos ver a partir da **Figura 5**, podemos abri-lo no IDE clicando no botão intitulado *OPEN IN IDE*. O resultado será parecido com o ilustrado na **Figura 7**. A única diferença que o leitor notará é que, para o seu workspace recém-criado, ainda não haverá nenhum projeto listado (será um workspace ainda “em branco”).

Observe que, assim que o IDE é aberto, uma mensagem é exibida no canto superior direito, indicando que o agente do workspace já se encontra em operação. Esse é um passo fundamental, pois, como vimos no início do artigo, ao discutir a arquitetura do Che, esse agente é o principal ponto de contato do workspace master com workspaces, e a partir de onde toda a administração de recursos presentes em um workspace se realiza.

Vimos, até aqui, algumas das características de workspaces no Eclipse Che. Há, no entanto, outros detalhes sobre os quais falaremos no decorrer do texto. Nesse instante, faremos uma breve pausa neste assunto para começar a trabalhar na criação de nosso primeiro projeto.

Nosso primeiro projeto

Ao longo desta seção, usaremos o workspace criado para construir nosso primeiro projeto no Eclipse Che. A criação de projetos pode ser realizada, nessa plataforma, de duas formas. São elas:

1. Gerar um projeto novo, a partir de um modelo ou exemplo provido pela plataforma;

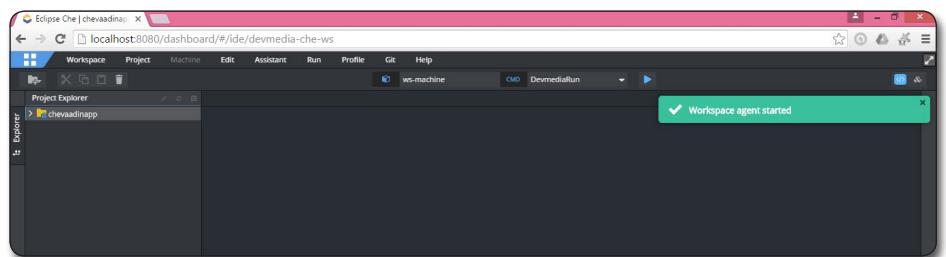


Figura 7. Workspace aberto no IDE, logo depois de ser criado

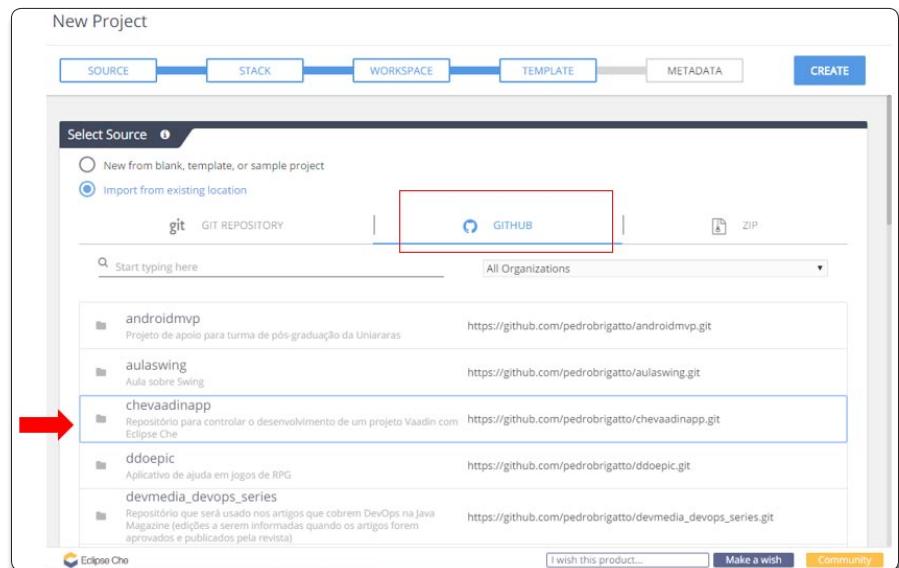


Figura 8. Criação de um projeto a partir de repositório hospedado no GitHub

2. Importar um projeto existente, de alguma fonte conhecida.

Em nosso tutorial, usaremos o segundo modo, a partir da opção *Import from existing location*. Há três modos pelos quais podemos importar um projeto para um workspace. O primeiro deles é usar um projeto compactado (ZIP). O segundo, que é o que aplicaremos, é apontar para um projeto hospedado em uma conta GitHub. E o terceiro, também relacionado ao sistema de versionamento Git, permite que importemos projetos hospedados em servidores de outras soluções baseadas nesse sistema de versão, como o Bitbucket.

Analisemos a **Figura 8**. Neste tutorial, sinalizamos a importação de um projeto e, em seguida, selecionamos a aba *GITHUB*, para, enfim, definir o repositório *chevaadinapp* como a fonte de onde importaremos o projeto. É importante lembrar que, para que essa lista de repositórios seja exibida quando clicamos na aba *GITHUB*, é preciso que

sua conta tenha sido configurada conforme orientamos no início do tutorial. Se isso não acontecer, volte para a seção *Configurando o acesso a uma conta GitHub via OAuth* e repita o procedimento nela detalhado.

Selecionado o projeto, basta que salvemos essa configuração pressionando o botão *CREATE*.

Dentro de poucos instantes, o projeto estará criado, salvo e associado ao workspace selecionado. Mais uma vez, o leitor poderá observar que, tanto no Dashboard quanto na lista de projetos do Che, esse projeto estará listado e haverá um botão com os dizeres *OPEN IN IDE*, a partir do qual podemos abri-lo no IDE. A **Figura 9** ilustra os detalhes, que, a partir de agora, serão destacados.

A primeira característica importante tem a ver com a estrutura do projeto em si. Perceba que, em termos visuais, o leiaute, as janelas e o editor lembram muito o IntelliJ IDEA. O editor de código também lembra muito o que todos os outros IDEs usam,

Introdução ao Eclipse Che

mas com a diferença de que, neste caso, a tecnologia por trás é toda herdada do projeto Orion.

Recursos como o JDT Intellisense, que sugere opções para completar instruções quando usamos o atalho *Ctrl + Espaço*, também estão disponíveis e são, tecnicamente falando, o resultado de um trabalho do Workspace Master, que, ao identificar que se trata de um projeto Java, injetou esse recurso no workspace. Observe, ainda, que fora aplicado *syntax highlighting* ao código, recurso que facilita imensamente o trabalho do desenvolvedor.

Não apenas esses recursos citados, mas todos os outros, estão implementados na forma de microserviços hospedados no servidor, e são injetados sob demanda nos workspaces, a partir de ações de usuários e do tipo do projeto.

Outro serviço que desenvolvedores usam frequentemente é o de depuração de código. O depurador é outro serviço importante que pode ser injetado em nossos workspaces.

Na **Figura 9**, podemos ver um ponto de parada (*breakpoint*) definido em uma classe chamada *MyUI*, exibida no editor. Esse breakpoint fica indicado por uma seta e, neste caso, foi inserido na linha 36.

Resumidamente, todo o conteúdo destacado até aqui, referente ao conteúdo da **Figura 9**, ilustra alguns pontos que havíamos introduzido no começo do artigo:

1. Todo projeto tem um tipo a ele associado;
2. O tipo de um projeto, bem como ações tomadas por usuários, diretamente no IDE, determinam quais tipos de serviços devem ser injetados no workspace para que a experiência do colaborador,

junto a um projeto, seja tão suave quanto a que já tinha ao usar seu IDE preferido.

Executando um projeto

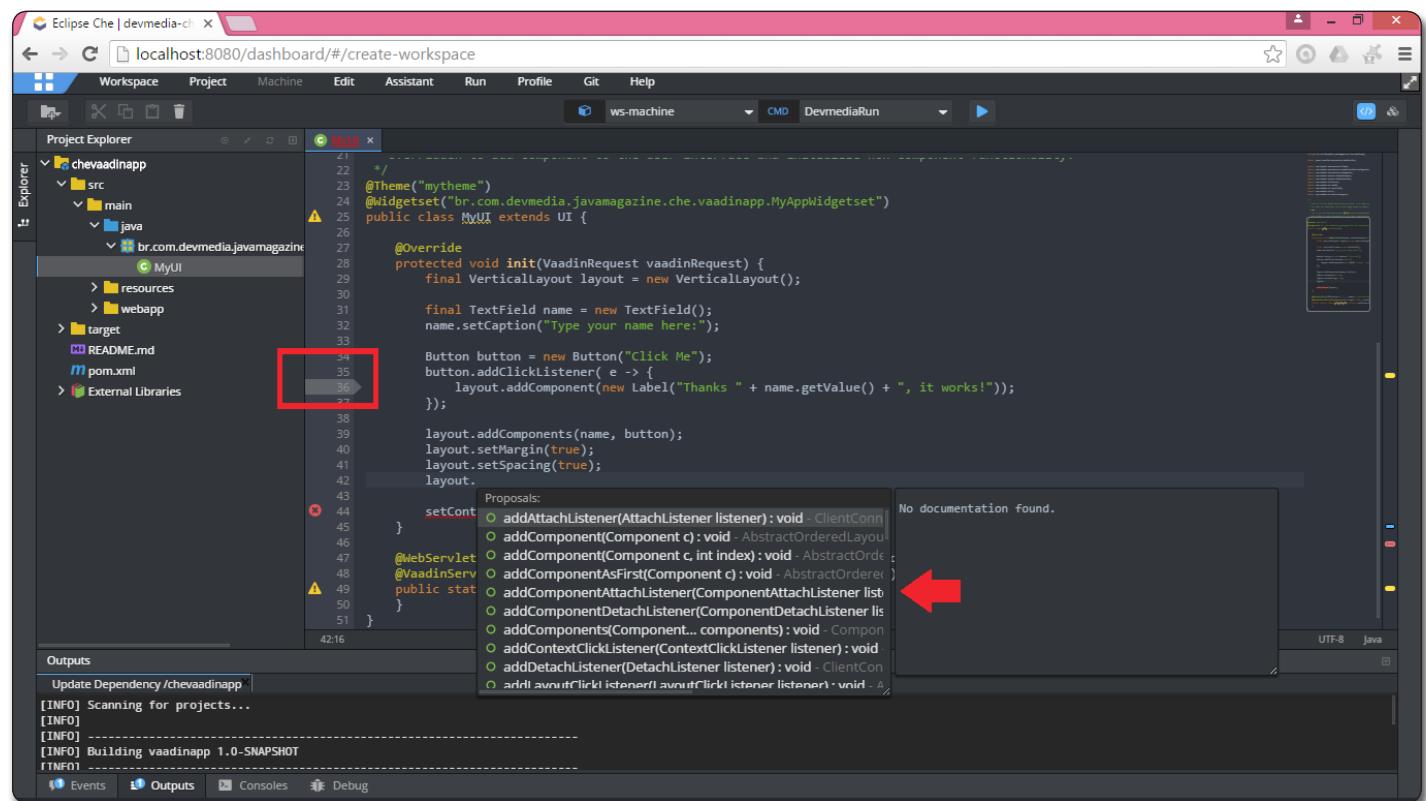
Para executar o projeto que acabamos de importar, usaremos o assistente gráfico disponibilizado pelo IDE. O primeiro passo é selecionar a raiz do projeto. Em seguida, para executá-lo, usamos o item *Run > Edit Commands* da barra de menu, que exibe um pop-up para criarmos ou editarmos comandos. Nessa janela, note uma categoria chamada *JAVA*, à esquerda, e um botão com o sinal “+”. Quando clicamos nesse botão, a porção à direita desse pop-up exibirá um formulário que usaremos para configurar um novo comando. Para este tutorial, preencheremos os campos solicitados de acordo com os dados apresentados na **Tabela 2**.

Ao salvarmos essa configuração, ela já aparecerá selecionada no topo do IDE, logo acima do editor de código.

Então, basta clicar no botão *Run*, que fará com que o container web (*Jetty*, neste caso) seja iniciado e a aplicação seja implantada e iniciada. Observe, pela **Figura 10**, que, logo acima do console, sob o editor de código, tanto o comando executado (*jetty:run*, neste caso) quanto uma URL da aplicação são exibidos. Ao clicarmos na URL em questão, a aplicação será aberta em outra aba do navegador, pronta para uso (seja em contexto de execução ou depuração).

A linha de comando (Terminal)

Além de todas as visões e o editor vistos até aqui, o Eclipse Che oferece também toda a flexibilidade da linha de comando para



The screenshot shows the Eclipse Che IDE interface. On the left is the Project Explorer view, displaying a project named 'chevaadinapp' with subfolders 'src', 'main', 'java', and 'br.com.devmedia.javamagazine'. Inside 'src/main/java', there is a file named 'MyUI.java'. The code editor shows the following snippet:

```
21
22
23 @Theme("mytheme")
24 @Widgetset("br.com.devmedia.javamagazine.che.vaadinapp.MyAppWidgetset")
25 public class MyUI extends UI {
26
27     ...
28
29     @Override
30     protected void init(VaadinRequest vaadinRequest) {
31         final VerticalLayout layout = new VerticalLayout();
32
33         final Textfield name = new Textfield();
34         name.setCaption("Type your name here:");
35
36         Button button = new Button("Click Me");
37         button.addClickListener(e -> {
38             layout.addComponent(new Label("Thanks " + name.getValue() + ", it works!"));
39
40             layout.addComponents(name, button);
41             layout.setMargin(true);
42             layout.setSpacing(true);
43             layout.
44         }
45     }
46
47     @WebServlet
48     @VaadinServlet
49     public static
50     }
51 }
```

A red box highlights the code area between lines 34 and 45. A red arrow points to the Intellisense dropdown menu that appears below the cursor, listing various methods like `addAttachListener`, `addComponent`, etc. The status bar at the bottom right shows 'UTF-8 Java'.

Figura 9. Projeto exibido no IDE do Eclipse Che

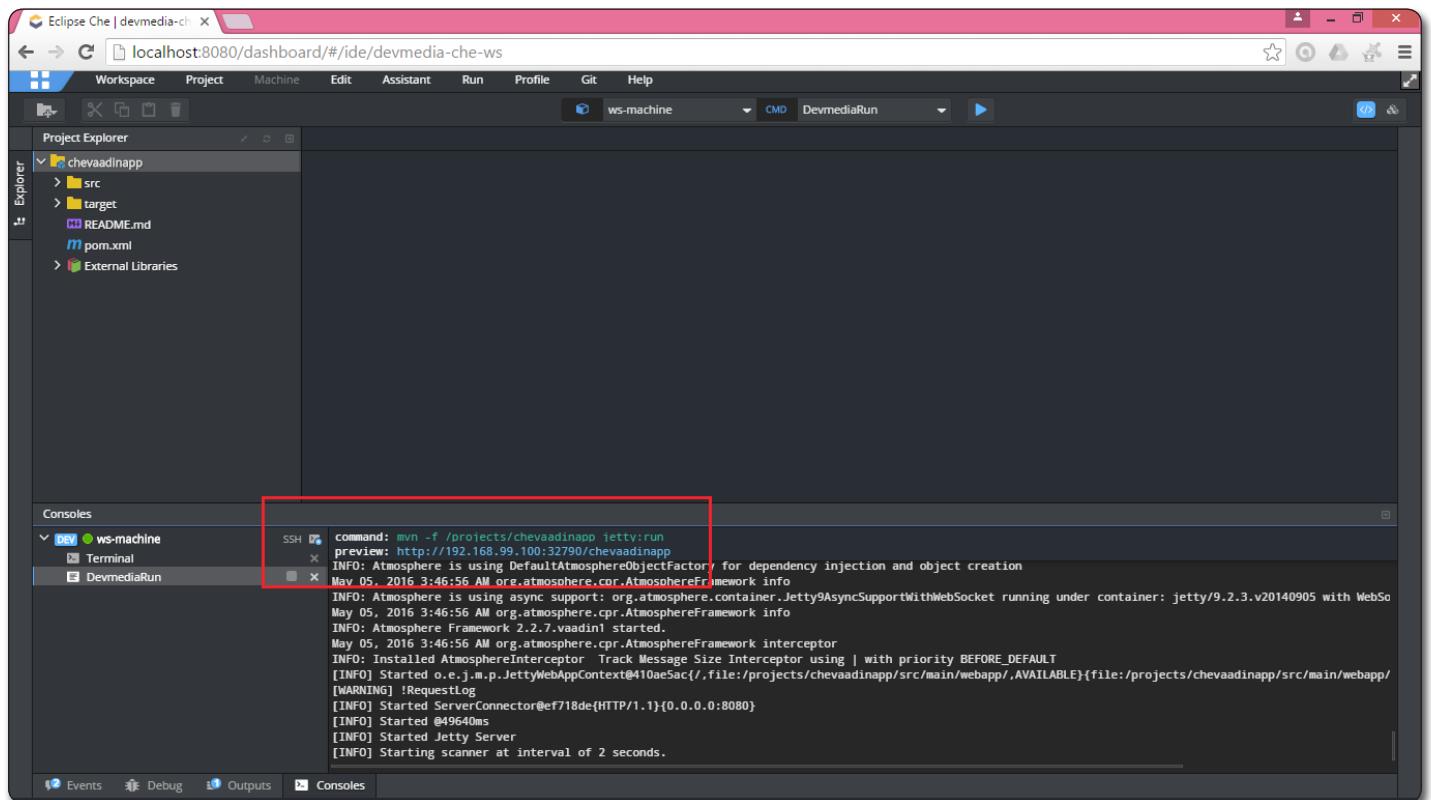


Figura 10. Projeto em execução, com URL de preview indicada acima do console

Campo do formulário	Valor do campo
Name	DevmediaRun
Working directory	\${current.project.path}
Command line	jetty:run
Preview URL	http://\${server.port.8080}/\${current.project.relpah}

Tabela 2. Configuração de um comando para execução do projeto

aqueles desenvolvedores — ou operadores — que se sentem mais confortáveis com essa abordagem. Esse terminal pode ser acessado pela opção *Run > Terminal* da barra de menu, ou diretamente pela aba *Consoles*, na borda inferior do IDE. O acesso à máquina que hospeda o workspace é feito via SSH, via usuário *root*, garantindo livre acesso ao container.

Para ambientar o leitor com a estrutura básica do sistema de arquivos da máquina que usamos em nosso tutorial, executamos alguns comandos no terminal e exibimos o resultado na **Figura 11**. O primeiro comando executado, *ls -a*, lista todo o conteúdo do diretório raiz. De todos eles, o que merece destaque neste momento é o *projects*, no qual os projetos associados a esse workspace serão salvos.

Em seguida, entramos no nosso projeto, *chevaadinapp*, e, nele, usamos o Apache Maven para inicialmente limpá-lo, e em seguida instalá-lo na máquina (*mvn clean install*). Esse comando, cabe lembrar, também poderia ser configurado graficamente, por meio da opção *Run > Edit Commands*, e sua execução seria exatamente igual à realizada por linha de comando.

Ao longo da próxima seção, continuaremos usando o terminal para, desta vez, submeter modificações realizadas no código-fonte ao repositório remoto. Como já frisado há pouco, todos os comandos que executaremos poderiam ser configurados a partir dos assistentes visuais, todos disponíveis na barra de menu do IDE. Como as funções em questão são bastante intuitivas no modo gráfico, acabamos optando por executá-las diretamente da linha de comando para mostrar, na prática, o nível de liberdade que a plataforma nos oferece.

Submitendo código ao repositório remoto

Agora que já fomos introduzidos ao terminal de comandos e sentimos a liberdade que ele nos traz, continuaremos a utilizá-lo para trabalhar, desta vez, com versionamento de código. A mudança que faremos no projeto será muito simples, apenas para gerar o material necessário para explorar o uso do Git no workspace.

Para começar, observe o conteúdo da **Listagem 4**. Esta é a única classe do projeto, **MyUI**, e representa a interface de usuário exibida quando a aplicação for executada. Apenas para o leitor entender um pouco a filosofia do framework Vaadin, a classe UI, superclasse de **MyUI**, representa o conteúdo que será exibido na tela inteira do navegador (em caso de aplicações normais) ou, então, em uma porção dela (quando a aplicação Vaadin em questão é desenvolvida seguindo o modelo de *portlets*). O projeto usado neste tutorial segue o primeiro modelo, e, portanto, o conteúdo implementado em **MyUI** será exibido em toda a tela do navegador.

Introdução ao Eclipse Che

O desenvolvimento em Vaadin segue um modelo muito semelhante ao de outros frameworks Java bem conhecidos, como SWT ou Swing. Vaadin, inclusive, é um framework que, assim como o SDK e as aplicações web do Eclipse Che, baseia-se em GWT. Essa foi a principal motivação que norteou nossa escolha de projeto-alvo para este tutorial. Para mais informações sobre o framework, há, na seção **Links**, uma referência para a sua página oficial.

Nosso projeto foi gerado a partir de um arquétipo Maven chamado *vaadin-archetype-application* (veja como utilizá-lo no próprio web site do Vaadin, disponível na seção **Links**) e, então, submetido

ao controle de versão do GitHub, no repositório *chevaadinapp* usado neste tutorial. A única tela dessa aplicação é composta por um campo de texto, um botão, e labels gerados dinamicamente, a cada interação do usuário com o botão.

A única modificação feita no código da classe **MyUI** foi uma pequena alteração na mensagem gerada a cada clique de botão, passando a imprimir o seguinte texto: “*Conteúdo do campo de texto: <valor inserido pelo usuário>*”. A linha de código-fonte modificada encontra-se em negrito na **Listagem 4**, para facilitar sua identificação.

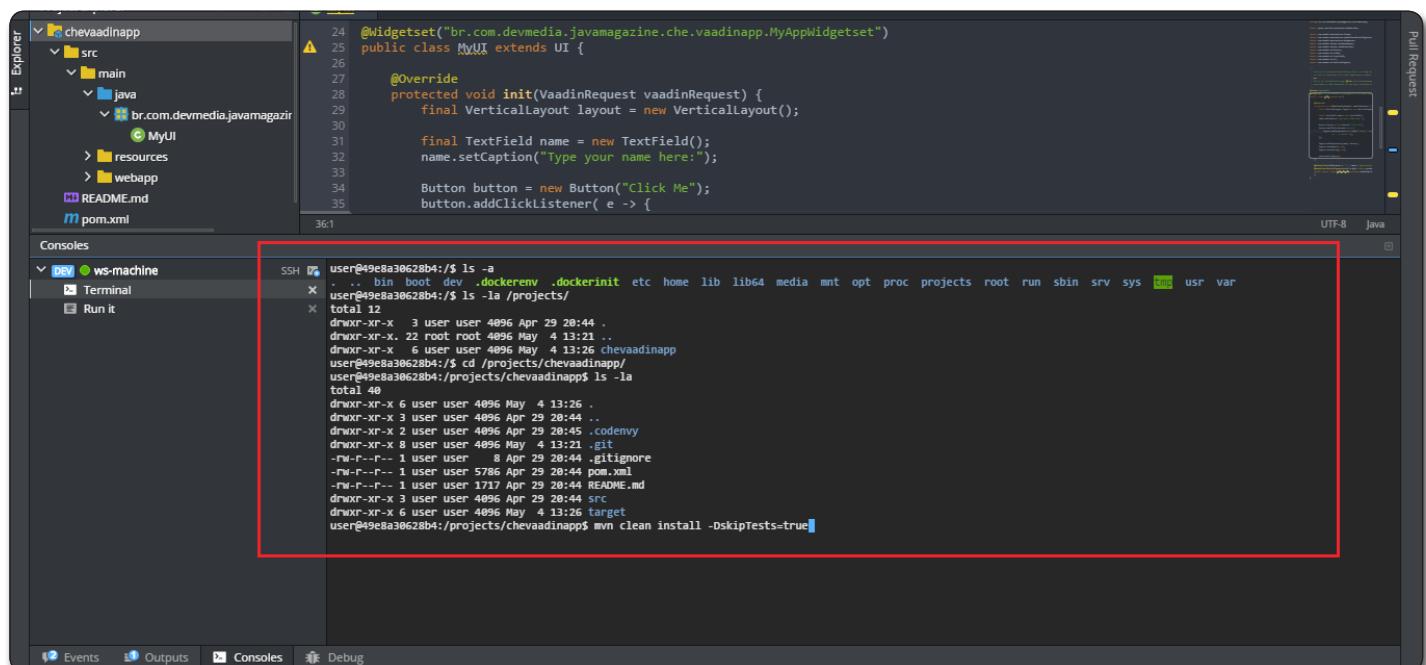


Figura 11. O terminal de comandos do IDE

Listagem 4. MyUI.java – realizando alterações nesta classe para posterior submissão ao repositório no GitHub.

```
package br.com.devmedia.javamagazine.che.vaadinapp;

import javax.servlet.annotation.WebServlet;
...
import com.vaadin.ui.UI;
import com.vaadin.ui.VerticalLayout;

/**
 * This UI is the application entry point. A UI may either represent a browser window
 * (or tab) or some part of a html page where a Vaadin application is embedded.
 */
@Theme("mytheme")
@Widgetset("br.com.devmedia.javamagazine.che.vaadinapp.MyAppWidgetset")
public class MyUI extends UI {

    @Override
    protected void init(VaadinRequest vaadinRequest) {
        final VerticalLayout layout = new VerticalLayout();
```

```
        final TextField name = new TextField();
        name.setCaption("Type your name here:");

        Button button = new Button("Click Me");
        button.addClickListener(e -> {
            layout.addComponent(new Label("Conteúdo do campo de texto:
                " + name.getValue()));
        });

        layout.addComponents(name, button);
        layout.setMargin(true);
        layout.setSpacing(true);

        setContent(layout);
    }

    @WebServlet(urlPatterns = "/*", name = "MyUIServlet", asyncSupported = true)
    @VaadinServletConfiguration(ui = MyUI.class, productionMode = false)
    public static class MyUIServlet extends VaadinServlet {
    }
```

Assim que verificarmos que as alterações estão funcionando conforme o desejado, chega o momento de submetermos as alterações ao repositório central. Para isso, usaremos o terminal de comandos e um pouco de nosso conhecimento em *git*, que, como já sabemos, fora instalado e disponibilizado neste workspace pelo fato de estar contido na especificação da *stack Java* oferecida pela plataforma.

Observe a **Listagem 5**. Nela, apresentamos todas as instruções executadas no terminal para que o conteúdo seja, enfim, submetido. Em linhas gerais, o trabalho realizado pode ser resumido conforme os itens listados a seguir:

- Configuração do nome de usuário a ser usado para a submissão de código;
- Configuração do e-mail associado ao usuário configurado no passo anterior;
- Modificação do mecanismo padrão de *push* do Git para “simple” (veja detalhes no **BOX 2**);
- Adição das modificações realizadas no repositório Git local (operação de *commit*);
- Submissão das alterações também ao repositório remoto.

Logo após cumprirmos todos os passos da **Listagem 5**, já vemos a submissão refletida no repositório do GitHub. Se quiséssemos agregar nessa cadeia de operações atividades como as de integração e entrega contínuas, enriqueceríamos ainda mais todo o processo. Caso o leitor se interesse por isso, há material suficiente publicado nas edições 147, 148, 149 e 150 da Java Magazine, mas vale mencionar, também, que o próprio Che oferece a possibilidade de integração com outras plataformas, por exemplo a OpenShift. Veja mais sobre isso em uma referência que deixamos na seção **Links**.

Por fim, apenas para mantermos uma associação de tudo o que

BOX 2. Os mecanismos de push do Git

O Git oferece alguns mecanismos possíveis para o procedimento de *push*, que nada mais é do que “empurrar” para repositórios remotos alterações feitas em um contexto local. No exemplo deste artigo, o repositório remoto encontra-se hospedado em uma conta GitHub, mas o serviço em si poderia ser qualquer outro, desde que baseado no sistema Git. Quanto às estratégias de *push* disponíveis para uso, até o momento em que este artigo foi escrito, dispúnhamos das seguintes opções: *nothing*, *current*, *upstream*, *simple* e *matching*.

Até a versão 2.0 do Git, seu mecanismo padrão era o *matching*. A partir de então, passou a ser o *simple*. Essa opção trabalha de forma muito similar à *upstream*, com a diferença de agregar um nível maior de segurança pelo fato de não permitir que submissões de código — os *pushes* — sejam realizadas caso a *upstream* configurada no contexto da máquina de desenvolvimento (de onde a submissão parte) seja distinta daquela determinada no repositório ao qual o conteúdo está sendo submetido.

Caso o leitor queira obter informações mais detalhadas a respeito dessas estratégias de submissão de conteúdo a repositórios *git*, deixamos na seção **Links** uma referência para a página oficial do projeto.

fizemos via linha de comando com recursos gráficos disponibilizados pelo IDE do Eclipse Che, disponibilizamos as devidas correspondências na **Tabela 3**. Nela, o leitor encontrará o modo visual de processar todos os comandos que executamos a partir da **Listagem 5**.

A perspectiva OPS

Ao longo de todas as seções anteriores, abordamos exclusivamente operações e assuntos relacionados ao contexto do desenvolvedor de software. No entanto, o Eclipse Che também oferece uma perspectiva dedicada ao universo dos operadores, que veremos brevemente ao longo dos próximos parágrafos.

Para isso, tomaremos como base o conteúdo da **Figura 12**. A perspectiva “Ops” é exibida quando clicamos no ícone destacado em azul, indicado por uma seta vermelha nessa figura. As principais funções oferecidas por ela são:

- A criação e atualização de receitas Docker;

Comando realizado	Caminho a partir da barra de menu do IDE
git config --global user.name/email	Profile > Preferences > Git > Committer
git commit	Git > Commit
git push	Git > Remotes > Push

Tabela 3. Comandos Git executados via terminal

Listagem 5. Comandos executados no terminal do Eclipse Che.

1. git config --global user.name pedrobrigatto
2. git config --global user.email pedrobrigatto@gmail.com
3. git config --global push.default simple
4. git commit -m "Alterando mensagem de apresentacao da aplicacao"
5. git push -u origin master

- Visualização da configuração de todas as máquinas associadas a um workspace;
- Acesso — via SSH — ao terminal de comandos de uma máquina;

Não nos prolongaremos nas atividades associadas à Operação de Software por não ser, esse, o foco de nosso artigo. Entretanto, não poderíamos deixar de, ao menos, citá-la, para que o leitor, de acordo com seu interesse nas operações que ela oferece, possa explorá-la melhor.

Exportando nosso workspace

Workspaces, como já vimos, são a peça central do Eclipse Che, e a incorporação de runtimes tem como principal objetivo resolver o velho dilema do setup de ambientes que, por ser delegado a cada indivíduo envolvido no projeto, abre margem para a diversidade e, consequentemente, torna-se foco de muitos dos problemas enfrentados no dia a dia de projetos.

Uma vantagem da plataforma é que podemos não apenas construir e evoluir nossos workspaces, mas salvá-los e exportá-los de modo que, quando precisarmos retomar nossas atividades, continuemos exatamente de onde havíamos parado. Além disso, essa função permite que compartilhemos estados específicos de um workspace com outros desenvolvedores, que, para acessá-los, precisariam apenas visitar um link gerado e compartilhado por nós para, em questão de minutos, tê-los prontos para uso, a partir de um navegador web.

A opção de exportação pode ser encontrada a partir do item **WORKSPACES** do menu lateral do Dashboard do Eclipse Che.

Introdução ao Eclipse Che

Ao selecionarmos um workspace, vemos um painel com as suas características e funções, como a de inicialização e parada. Note que, além dessas duas funções, há também um botão com os dizeres *EXPORT*, que, uma vez pressionado, abre o pop-up exibido na **Figura 13**.

Por essa figura, vemos que existem duas formas de se exportar um workspace. Na primeira (“*AS A FILE*”), o conteúdo é salvo em um arquivo JSON e baixado em nossa máquina. Esse documento pode, em seguida, ser compartilhado ou mesmo versionado, para que fique acessível a qualquer outro colaborador. A segunda, que corresponde mais à realidade ditada pelas novas tecnologias que vêm sendo

adotadas atualmente, é salvá-lo em uma nuvem privada. Nesse caso, precisamos fornecer a URL e as credenciais dessa nuvem para que a conexão seja estabelecida, e o processo é realizado automaticamente.

Caso recordemos o início do tutorial, ainda na **Figura 6** (quando falávamos sobre a criação de workspaces), mencionamos uma aba intitulada *Import an existing workspace configuration*. O conteúdo exibido no pop-up da **Figura 13** é exatamente o tipo de informação que deve ser inserida na área de texto. Uma vez confirmada a operação, o workspace, exatamente com as características e estados determinados no documento JSON, será carregado.

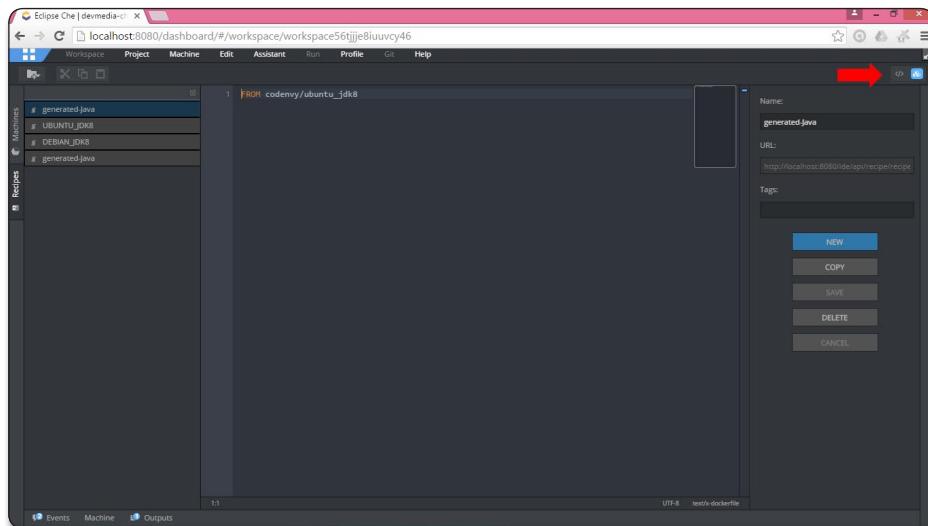


Figura 12. A perspectiva OPS do IDE

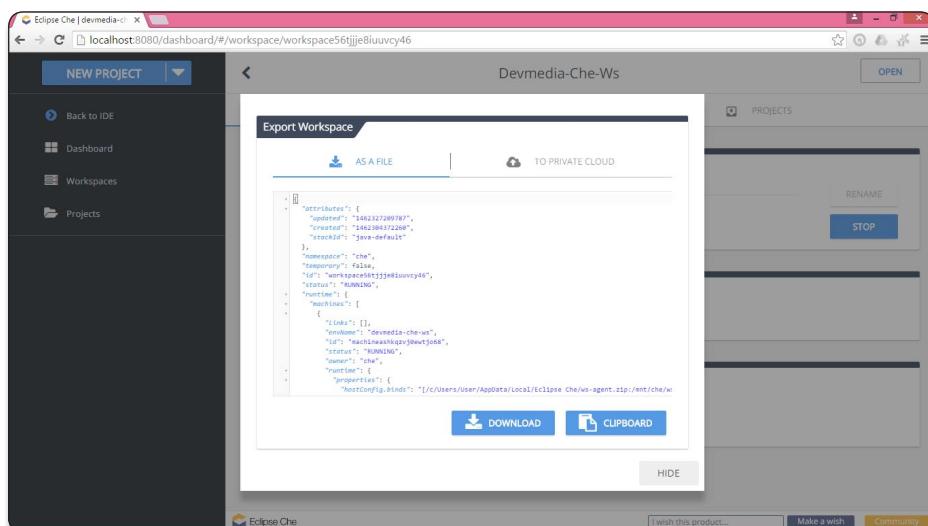


Figura 13. Exportando um workspace no Eclipse Che

Agora que já discutimos sobre praticamente todos os principais recursos da plataforma, estudaremos uma solução que apresenta o Eclipse Che em uma versão remota, distribuída e, portanto, muito mais próxima do que corporações devem adotar para, efetivamente, sentir os benefícios que essa nova cultura pode proporcionar.

A solução comercial da Codenvy

Os recursos vistos até aqui nos dão uma boa ideia de todo o arsenal tecnológico que o Eclipse Che oferece, bem como o grande potencial que a plataforma possui. Trata-se, certamente, de uma tecnologia com grandes chances de transformar, ao longo dos próximos anos, a cultura de colaboração em torno de projetos de software.

Analisaremos, ao longo desta seção, como todo este potencial da plataforma foi explorado e estendido em uma solução comercial oferecida por uma empresa chamada Codenvy, cujo fundador e CEO é ninguém menos que o próprio Tyler Jewell, idealizador do Che. Estudaremos, especialmente, um recurso adicional intitulado *fábrica*, mecanismo cuja proposta é automatizar o processo de configuração e disponibilização de workspaces.

Para ilustrar como uma fábrica funciona, vejamos a **Figura 14**. Esta é a página inicial de uma conta que criamos para este tutorial. Para criar a sua própria conta e experimentar a tecnologia, basta acessar o portal da Codenvy, disponibilizado na seção **Links**, e se cadastrar.

Assim que entramos no portal, notamos que o menu lateral disponibiliza um item a mais, intitulado *Factories*. Ao clicarmos nele, o painel à direita exibe nossa lista de fábricas e um botão, localizado no canto superior direito da tela, que nos permite criar novas fábricas. Ao pressionarmos esse botão, abre-se um formulário de cadastro, ilustrado na **Figura 14**.

Em nosso exemplo, a fábrica criada foi vinculada ao mesmo repositório usado ao longo de todo o tutorial, *chevaadinapp*. Selecionado o repositório, devemos clicar no botão *NEXT*, no canto inferior da tela, e preencher o restante das informações, conforme ilustrado na **Figura 15**. Essas informações estão listadas a seguir:

- O nome da fábrica;
- A receita e o workspace a serem usados;
- A quantidade máxima de memória a ela alocada;
- Os comandos que devem ser associados a ela;
- As ações que devem ser tomadas, tão logo o workspace seja colocado em execução.

Nessa nossa fábrica, definimos dois comandos, vistos na **Figura 16** (que é uma continuação da tela de criação de uma fábrica, conforme rolamos o formulário para baixo). O primeiro deles, chamado **PackageIt**, usa o Maven para empacotar o projeto. Já o segundo, **RunIt**, é usado para executar a aplicação. Esses dois comandos estarão listados para nós, prontos para uso no workspace, assim que o abrirmos em um navegador web.

Um ponto relevante, contido tanto na **Figura 15** quanto na **Figura 16**, tem a ver com duas URLs geradas para nós, assim que a fábrica é criada. A primeira é obtida a partir de um botão com os dizeres *Copy URL*, enquanto a segunda é disponibilizada a partir de outro botão, com o título *Copy Markdown*. Enquanto o primeiro nos informa a URL do workspace em si, o segundo já é uma string que podemos adicionar, por exemplo, aos nossos repositórios de código, permitindo que outros colaboradores possam carregar, a partir de um único clique, o workspace no navegador web de sua preferência. Em nosso caso, usamos essa string no arquivo *README.md* do repositório, e o resultado pode ser visto na **Figura 17**.

Ao pressionarmos o botão com os dizeres *Developer Workspace*, o workspace associado à fábrica será aberto em um navegador, totalmente configurado e pronto para uso. Portanto, sem a necessidade de software algum instalado em sua máquina, qualquer pessoa poderia se tornar um potencial colaborador do projeto, pois todo o stack de software seria automaticamente preparado, bem como o projeto e o runtime no qual o projeto tem de ser testado antes de qualquer submissão de código.

E, já que workspaces podem ser exportados para uma nuvem privada, como vimos há pouco, podemos ainda criar

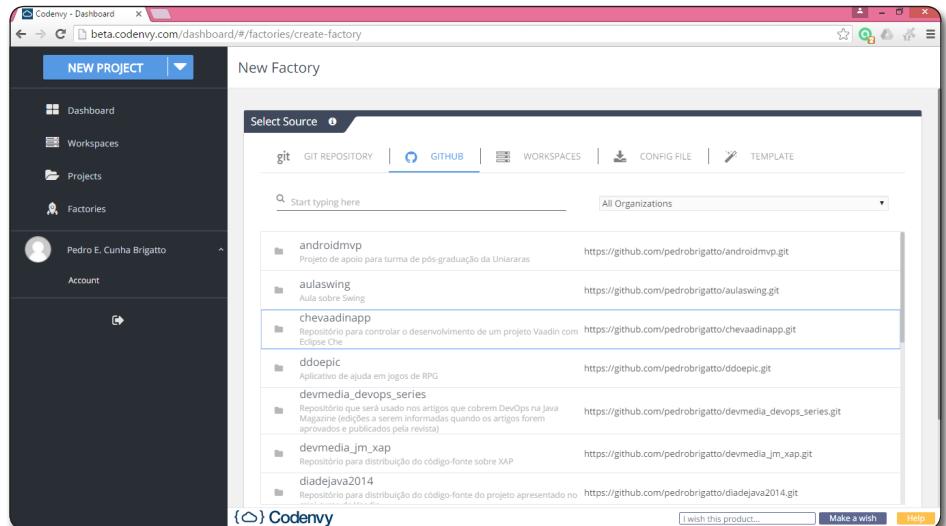


Figura 14. Configuração de uma fábrica no ambiente da Codenvy

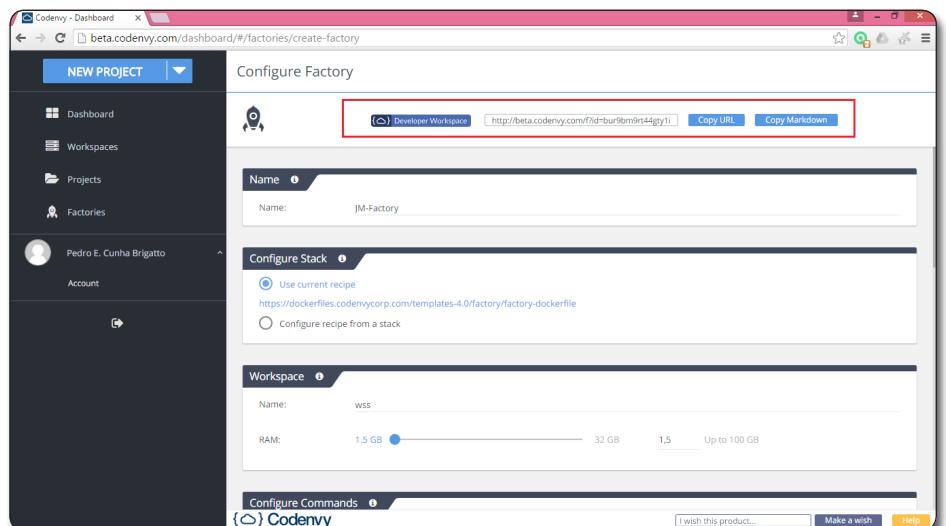


Figura 15. Detalhe para as URLs oferecidas para uso de workspaces via fábricas

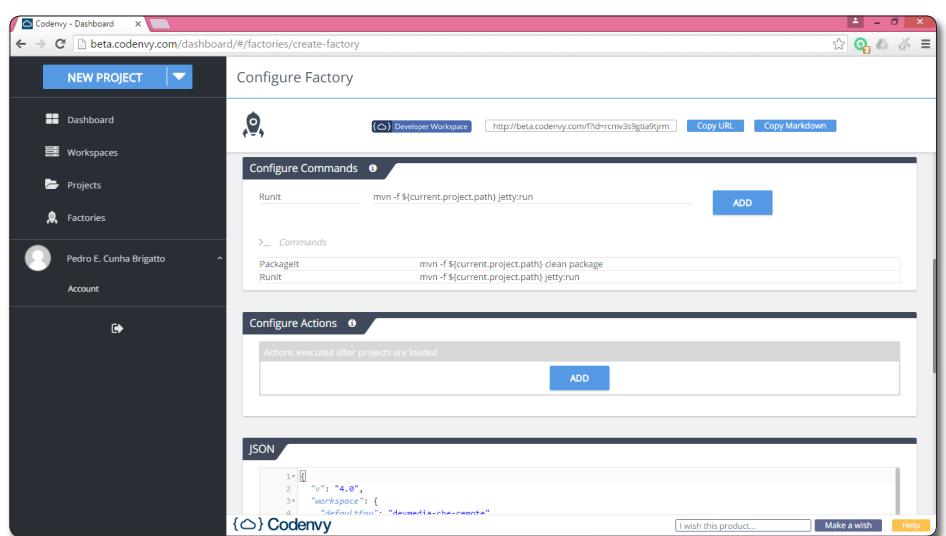


Figura 16. Continuação da edição de uma fábrica, com a definição de comandos

Introdução ao Eclipse Che

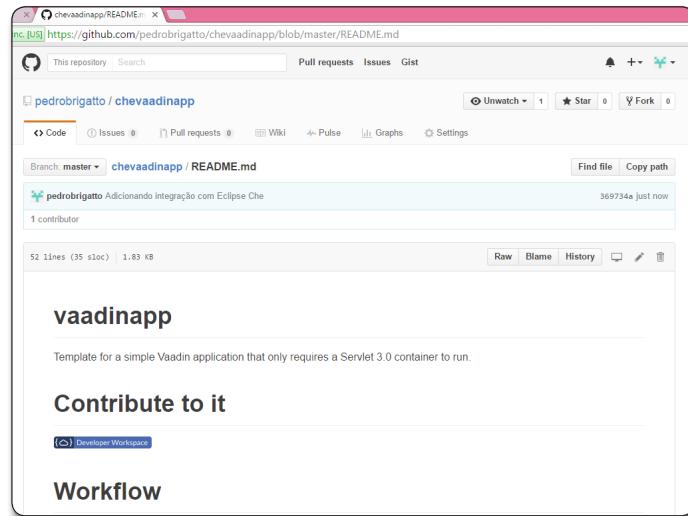


Figura 17. Repertório no GitHub, com um link para acesso ao workspace do Che

outros botões de compartilhamento como esse que vimos agora e, novamente, compartilhar estados específicos de um workspace, relativos a mudanças bastante específicas no código, permitindo que colegas possam enxergar não apenas as modificações realizadas, mas experimentá-las exatamente no contexto em que devem ser avaliadas: ambientes, comandos e ações.

Em termos práticos, imagine que, dada uma colaboração de um desenvolvedor, em um projeto versionado em uma conta no GitHub, todo o conteúdo desenvolvido tenha de ser incorporado ao *branch* principal. Antes de fazer isso, entretanto, é necessário certificar-se que o projeto funciona adequadamente com essa mudança. Neste momento, normalmente a validação das modificações se dá por meio de atividades como revisões de código, mas imagine quanto mais poderoso pode ser esse processo caso tenhamos a possibilidade de não apenas analisar o código em si, mas executá-lo e experimentá-lo naquela versão específica, no runtime em que foi desenvolvido.

Como podemos imaginar, as possibilidades que ganhamos com esse conceito de exportação de workspaces aumentam bastante, principalmente em termos da qualidade final da colaboração.

O Eclipse Che é uma plataforma com um enorme potencial para redefinir o conceito de colaboração em projetos de software. O que vimos ao longo do artigo foi uma pincelada sobre os principais fundamentos e funcionalidades que a plataforma oferece, mas as possibilidades que ela abre para nós são ainda maiores.

O Che pode ser usado puramente como um servidor de workspaces e abrigar uma IDE totalmente customizada para casos específicos, como já feito em uma parceria entre Codenvy e a SAP, para desenvolvimento de aplicações centralizadas em dados, tendo como sistema de informações o SAP HANA.

Pode, também, integrar-se com plataformas poderosíssimas e bastante influentes no cenário de DevOps, como a OpenShift. Outro caso significativo é da colaboração da Microsoft com

plataformas como Azure, ou mesmo a combinação do Che com ferramentas e tecnologias muito promissoras da IBM, ligadas principalmente às áreas de DevOps e IoT, como o BlueMix.

Todos esses cases citados aqui estão em uma apresentação feita por Tyler Jewell no EclipseCon deste ano, e cuja URL se encontra na seção **Links**.

Há, ainda, a possibilidade de nós mesmos desenvolvermos ou evoluirmos cada componente do Che, pois temos em mãos muito mais que um IDE ou um servidor de workspaces. Temos um poderoso SDK, baseado em GWT, que diminui drasticamente a curva de aprendizado (afinal, estamos falando de API Java) para o desenvolvimento de novas extensões tanto para o servidor quanto para workspaces e o IDE.

Por fim, saiba que o Eclipse Che ainda se encontra em uma versão Beta, mas, dado tudo o que vimos ao longo do artigo e a velocidade com que o projeto vem crescendo, estamos, sem dúvidas, diante de um potencial protagonista no cenário de desenvolvimento de software para os próximos anos.

Autor



Pedro E. Cunha Brigatto

pedrobrigatto.devmedia@gmail.com

Engenheiro da Computação graduado pela Universidade Federal de São Carlos, desenvolvedor certificado SAP Netweaver (Java Stack) e programador certificado SCJP. Especialista em Engenharia de Software graduado pela UNIMEP e pós-graduado em Administração pela Fundação Bl-FGV, atua com desenvolvimento de software desde 2005. Atualmente é consultor de serviços profissionais da Avaya Brasil.



Links:

Página principal do projeto Eclipse Che.

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

Apresentação de Tyler Jewell no EclipseCon 2015: "An introduction to Eclipse Che".

<https://www.eclipsecon.org/na2015/session/introduction-eclipse-che>

Apresentação de Tyler Jewell no EclipseCon 2016: "Eclipse: The evolution and future of IDEs".

<http://tinyurl.com/huq92ho>

Documentação oficial sobre os mecanismos de push do Git.

<https://git-scm.com/docs/git-push>

Projetos de exemplo disponibilizados pela plataforma.

<https://github.com/che-samples>

Página oficial do framework Vaadin.

<https://vaadin.com/home>

Integrando Eclipse Che e OpenShift.

<http://tinyurl.com/hy2e2dc>

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>

Desenvolvimento IoT com o Java 8 e Raspberry Pi

Aprenda neste artigo o que é, quais são os principais conceitos e como desenvolver um projeto voltado para a Internet das Coisas

AInternet das Coisas é hoje uma das áreas mais estudadas por entusiastas de tecnologia da informação. Assim, além de estar presente em vários artigos, tem despertado o interesse de várias empresas do setor de tecnologia, como Google, Oracle, Qualcomm e Microsoft.

O conceito de IoT, como veremos a partir de agora, é um pouco amplo e, portanto, difícil de resumir em uma sentença. De forma simples, entende-se por Internet das Coisas um ambiente que reúne diversos tipos de hardware, as “coisas” (things), que se comunicam entre si (M2M, ou *Machine to Machine*) com pessoas (através de displays de LED, por exemplo) ou ainda com outros sistemas computacionais (via Wi-Fi, Bluetooth, etc.). A Internet das Coisas, no entanto, transcende a conectividade entre sensores, motores, micro controladores, gadgets vestíveis, entre outras opções, e tem como propósito principal elevar o nível de comunicação entre dispositivos e seres humanos, transformando pulsos elétricos em dados que podem ser manipulados e enviados para sistemas maiores, com interface mais amigável.

Segundo a Gartner, referência de consultoria em tecnologia, hoje temos cerca de cinco bilhões de dispositivos interconectados que compõem a IoT e esse número deve saltar para 25 bilhões até 2020. Com essa quantidade de dispositivos gerando informações que trafegam na rede, alguns desafios precisam ser superados, tais como:

1. Como processar essa quantidade de informação e transformá-la em algo útil?
2. Como desenvolver soluções de integração entre bilhões de equipamentos e sistemas de grandes corporações?
3. Quem será o desenvolvedor de tais sistemas? Um engenheiro da computação, eletrônico ou um analista de sistemas?
4. Um protocolo de comunicação entre os dispositivos tem que ser estabelecido, bem como APIs de acesso

Fique por dentro

Este artigo tem como objetivo principal a introdução ao conceito de Internet das Coisas, bem como explorar suas potenciais aplicações, e, como o Java 8 consegue suprir as demandas de desenvolvimento originárias dessa nova tecnologia. Além dos conceitos de IoT, veremos como o Raspberry Pi, um microprocessador de baixo consumo e custo, pode ser utilizado como interface para a prototipação de projetos para a IoT.

a esses protocolos devem ser suportadas pelas linguagens de programação.

Atualmente já existem tecnologias que podem auxiliar na mineração e processamento de dados, tais como as soluções de Big Data de empresas como SAS, Oracle e IBM, superando assim o primeiro desafio. Porém, a resolução desse ponto não é o foco desse artigo.

O segundo desafio também já possui mais de uma solução, e entre as principais podemos citar as plataformas Arduíno e Raspberry Pi. O Arduíno é uma plataforma de prototipagem composta de uma placa microcontroladora, uma IDE e uma linguagem de programação própria, baseada em C e C++. O hardware segue os princípios do “hardware open source”, ou seja, tem uma especificação que pode ser fabricada por qualquer pessoa que estiver disposta a construir sua própria placa. A concepção dessa plataforma surgiu como um projeto da universidade italiana Ivrea Interaction Design Institute, com o objetivo de fornecer uma solução de baixo custo para que estudantes de engenharia pudessem desenvolver seus trabalhos acadêmicos mais facilmente.

O Arduíno pode ser conectado a diversos sensores, estabelecendo uma comunicação de duas vias com eles, de tal maneira que é possível ler dados dos sensores e escrever comandos (ligar e desligar, por exemplo) e parâmetros de configuração, tais como precisão de amostra, leitura contínua ou não, sentido de rotação de um motor, etc.

Devido às características supracitadas, o processo de desenvolvimento de uma aplicação para a plataforma Arduíno é um pouco diferente do processo de desenvolvimento de software para um microcomputador. O código fonte é compilado da mesma forma, no entanto o produto desse processo é gravado no microcontrolador e a aplicação é executada sempre que o dispositivo é ligado (não há um sistema operacional com janelas e ícones para o usuário iniciar/encerrar a aplicação).

Já o Raspberry Pi (RPi) é um microcomputador completo, com processador, memória RAM e ROM (basicamente um cartão de memória MicroSD), portas USB, HDMI e Ethernet. Esse computador tem aproximadamente o tamanho de um cartão de crédito, opera com apenas 5 volts e possui interfaces que podem ser conectadas a diversos dispositivos.

Um dos diferenciais do Raspberry Pi é que ele roda um sistema operacional. Originalmente, apenas uma versão do Debian customizada para a arquitetura de processadores ARM, mas outros SOs também passaram a ser suportados na versão 2 do Pi, tais como o Windows 10, Ubuntu, entre outros. Devido a isso, temos várias opções de linguagens de programação disponíveis para essa solução, como C, C++, Java, Python, etc. Além disso, o Raspberry viabiliza diversas opções de conectividade com sensores, tais como os protocolos I²C, SPI e UART. Pode-se dizer até que todos os sensores compatíveis com o Arduíno são compatíveis com o Raspberry Pi.

Com todos esses diferenciais, por que escolher o Arduíno, se o Raspberry Pi pode me oferecer mais? Em favor do Arduíno, temos que levar em consideração que além de ser mais barato e consumir menos energia, conta também com portas de entrada analógicas (capazes de ler informações mais precisas e não apenas 0s e 1s, como as digitais), que podem ser úteis em algumas situações.

A escolha de qual plataforma utilizar, portanto, deve ser feita de acordo com as necessidades do projeto.

O terceiro desafio – quem vai desenvolver para IoT – é algo a se pensar, pois exigirá de um analista de sistemas com conhecimento básico em eletrônica. Palavras pouco comuns para um desenvolvedor, como resistores, transistores e voltagem, passam a fazer parte do vocabulário de quem se interessa por IoT. Desse modo, o perfil de profissional, ou entusiasta, que deseja adentrar nesse universo de sensores, motores, etc., é o de um engenheiro? À primeira vista, a resposta é sim. Um engenheiro da computação ou eletrônico, além do conhecimento de eletrônica, tem uma carga horária de faculdade bem interessante em linguagens de programação como C++ e Java, o que lhes dá boas condições para desenvolver uma integração entre sensores e um sistema computacional. Se você não tem esse perfil, no entanto, não se preocupe e não desista de conhecer esse novo mundo que se apresenta através da IoT. Provavelmente você terá um pouco mais de dificuldade, mas não precisa ser “cientista de foguetes” para fazer um LED acender quando a temperatura do ambiente atingir um valor muito alto, por exemplo.

O último ponto levantado, sobre protocolos e APIs de acesso ao hardware, é onde entra em cena o Java. Além de ser uma linguagem consagrada no mercado e com uma vasta quantidade de desenvolvedores, ainda tem uma característica fundamental para a adoção em um cenário tão heterogêneo quanto o da IoT, o fato de ser multiplataforma. Hoje em dia temos o Java rodando em artefatos simples como cartões, a poderosos computadores, passando por placas embarcadas, dispositivos móveis e outros aparelhos de menor poder de processamento e memória. Em sua essência, o Java mantém as mesmas características em todas as plataformas na qual funciona, apenas adaptando-se às particularidades de cada uma. Por isso temos alguns sabores de Java e



Desenvolvimento IoT com o Java 8 e Raspberry Pi

dentre essas variações, duas podem ser utilizadas para IoT: Java Micro Edition e Java Standard Edition Embedded.

Até meados da década passada, o Java Micro Edition (Java ME) era a principal solução tecnológica para o desenvolvimento de softwares para dispositivos móveis (leia-se celulares e PDAs), contudo, perdeu espaço com o avanço dos smartphones Android e iPhone. Nesse período de ascensão dos smartphones a SUN já tinha deixado o Java ME meio de lado e a Oracle não fez muito esforço para trazê-la de volta aos holofotes. No entanto, quando foi anunciado o lançamento do Java 8, foi lançada também uma nova versão do Java ME, atualizada para suportar as novas features da linguagem (Generics, por exemplo), mas tendo um novo foco, os dispositivos embarcados.

No Java ME 8, boa parte dos conceitos permaneceram (midlets, pacotes opcionais, etc.), mas classes de interface gráfica, por exemplo, deixaram de existir. Desse modo, essa nova versão não tem utilidade alguma para smartphones, o que reforça a inviabilidade de ter uma versão do Java ME 8 para Android, iPhone ou qualquer outro modelo. Em outras palavras, essa versão do Java é um subconjunto de classes da plataforma Java SE com o adendo de algumas classes pertinentes a dispositivos embarcados.

Por sua vez, o Java Standard Edition Embedded foi lançado juntamente com a versão 5 do Java Standard Edition e diferentemente do Java ME, representa o próprio Java SE otimizado para computadores de baixo consumo e menor performance (arquitetura de processadores ARM, por exemplo).

Enfim, as diferenças entre Java ME e Java SE Embedded são:

- Java ME trabalha com MIDlets, Java SE Embedded, com classes regulares que possuem o método **public static void main(String args[])**;
- Java ME tem classes para acesso às portas GPIO (*Generic Port Input Output*), protocolos I2C (*Inter-Integrated Circuit*), UART (*Universal Asynchronous Receiver Transmitter*) e SPI (*Serial Peripheral Interface*), fundamentais para a comunicação com sensores. O Java SE Embedded precisa de APIs de terceiros para tal;

- A versão Embedded é destinada a aparelhos com maior capacidade de memória e processamento, se comparados ao público alvo do Java ME.

O Raspberry Pi, em suas versões mais antigas (A+, B e B+), suporta tanto o Java ME quanto o Java SE Embedded. No entanto, a partir da versão mais recente, o Raspberry Pi 2 Model B, apenas a versão Embedded é suportada, pelo menos por enquanto, pois houve uma mudança na arquitetura do microprocessador.

Dito isso, para o projeto de exemplo apresentado neste artigo, utilizaremos a plataforma Raspberry Pi 2 Model B, o Java SE Embedded, bem como a biblioteca Pi4J, que leva a essa versão do Java algumas das funcionalidades do Java ME. Antes de começarmos com o projeto propriamente dito, no entanto, precisamos passar por alguns conceitos fundamentais para possibilitar sua compreensão e implementação.

As principais siglas da IoT

Algo comum nas áreas da Tecnologia da Informação, a IoT também exige um conhecimento multidisciplinar; neste caso, envolvendo eletrônica, mecânica e desenvolvimento de software. Para cada uma dessas disciplinas, tem-se um arsenal de siglas relacionadas e algumas, como as citadas anteriormente, merecem um pouco mais de atenção, pois o seu entendimento é fundamental para que seja possível compreender como é feita a comunicação entre os dispositivos integrantes de qualquer solução IoT.

Generic Port Input Output

A porta genérica de entrada e saída, ou GPIO, nada mais é do que um pino em um Raspberry Pi, um conector macho. Esse pino pode ser utilizado para a leitura ou escrita de dados e ter um valor a ele atribuído: ligado ou desligado. Ao conectar a GPIO a outro dispositivo, ela se torna uma interface com a qual podemos, por exemplo, ligar ou desligar um LED, configurar a direção da rotação de um motor, ler um conjunto de pinos e transformar o

valor dessa leitura binária em um valor decimal (temperatura, por exemplo), ou em qualquer outra coisa que seja comprehensível pela maioria das pessoas.

A manipulação das portas GPIO pode ser feita através de software e para isso o Java ME fornece classes para ler e escrever valores. Já o Java Embedded não fornece tais mecanismos e para suprir essa ausência, faz uso da biblioteca Pi4J.

Como mais um diferencial, o Raspberry Pi fornece até 25 portas de GPIO, identificadas por um número de 2 a 26, de acordo com o modelo do aparelho. A Figura 1 mostra os pinos em um Raspberry Pi 2, enquanto a Figura 2 mostra a identificação de cada GPIO, bem como os pinos de energia (3,3v e 5v) e aterrimento.



Figura 1. Os pinos de um Raspberry Pi 2 Model B

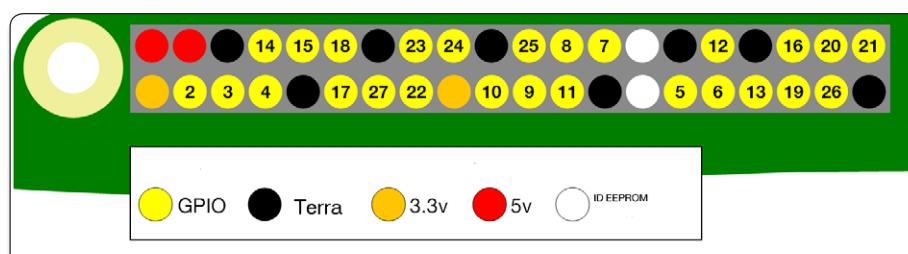


Figura 2. Identificação das portas GPIO

I²C

O Inter-Integrated Circuit (I²C) é um protocolo de comunicação que pode ser utilizado entre um Raspberry Pi e sensores conectados a ele. A conversação entre os dispositivos se dá através de um canal de comunicação de duas vias. Assim, tanto o Raspberry Pi quanto os sensores podem enviar e receber informações. A coordenação de quem envia e quem recebe informações é feita pelo Raspberry Pi, que assume o papel de Mestre do canal de comunicação, sendo os sensores conectados a ele, os escravos.

Os sensores que trabalham com o protocolo I²C têm, no mínimo, quatro pinos (pode haver mais, mas seria uma particularidade de um sensor A ou B), analisados a seguir:

- **Voltage Common Collector (VCC):** responsável pela alimentação de energia do sensor;
- **Ground (GND):** representa o aterramento, garantindo o isolamento elétrico do sensor;
- **Signal Clock (SCL):** representa o barramento responsável pela sincronização da comunicação. É ele quem dita o início e o fim de uma mensagem enviada; e
- **Signal Data (SDA):** barramento no qual trafegam as mensagens trocadas entre o mestre e os escravos.

Esses quatro pinos de um sensor compatível com o I²C são conectados a um Raspberry Pi através das GPIOs e a partir desse momento o sistema operacional detecta e atribui um identificador, um número hexadecimal, para o sensor. Para um melhor entendimento, pode-se fazer uma analogia entre o I²C e o protocolo de rede TCP/IP, onde temos o endereço IP composto por quatro octetos de números decimais (ex.: 192.168.100.1) para identificar uma máquina, enquanto no I²C temos apenas um octeto representado por um número hexadecimal (ex.: 4F).

Raio-X de uma protoboard

Antes de começarmos a implementar qualquer coisa para a IoT usando o Java, é muito importante entendermos como funciona a “Placa de Ensaio”, ou, do inglês, *protoboard*. Esse equipamento é fundamental para conectarmos as peças de nossos projetos entre si e ao Raspberry Pi. Mas, afinal, o que é uma *protoboard*? De modo simples, trata-se de uma placa com furos que são ligados por barramentos. É através desses furos que podemos ligar um componente a outro, ou ao RPi.

A Figura 3 mostra uma *protoboard*, onde podemos ver, destacados em azul, os barramentos horizontais. Esses são utilizados, geralmente, para conectar o protótipo a uma fonte de energia. Em vermelho, por sua vez, temos os barramentos verticais, onde cada furo em uma coluna é conectado aos demais furos das outras linhas dessa mesma coluna: de A a E e de F a J.

A título de exemplo, se ligarmos uma fonte de energia conectada nos barramentos horizontais a, digamos, o furo A1, os furos B1, C1, D1 e E1 também serão energizados.

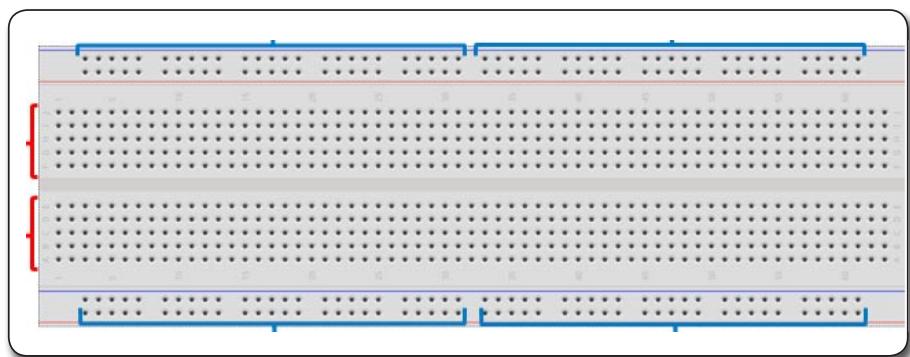


Figura 3. Exemplo de protoboard

Nota

O uso inadequado de uma protoboard pode levar a sérios danos em seus equipamentos, inclusive à “perda total” deles. Por exemplo, quando um projeto envolve diversos componentes, geralmente conta com uma fonte de energia auxiliar, como uma bateria de 9V. Se essa bateria for conectada erroneamente em um barramento, pode “fritar” o seu Raspberry Pi, que suporta no máximo 5V.

Pi4J

O Pi4J é uma biblioteca à parte do Java SE Embedded que leva a essa plataforma as interfaces necessárias para interagir com as portas GPIO. Essa biblioteca funciona com o auxílio de uma ferramenta do Java, o JNI (*Java Native Interface*), que permite à aplicação chamadas a funções instaladas no sistema operacional onde ela está sendo executada. Um exemplo de biblioteca que pode ser instalada no Raspberry Pi é o WiringPi, solução escrita em C e que provê métodos para manipular os pinos do GPIO.

Por motivos que não veem ao caso, o WiringPi utiliza uma nomenclatura diferente do padrão do Raspberry Pi para as portas GPIO.



Desenvolvimento IoT com o Java 8 e Raspberry Pi

Na **Tabela 1** podemos verificar essas diferenças. As colunas com o cabeçalho “Pino Físico” representam os pinos da placa do Raspberry Pi, os mesmos identificados na **Figura 1**. As colunas RPi contêm os “nomes” dos pinos físicos (GPIO02 à GPIO26) sob a perspectiva do Raspberry Pi, nomenclatura essa que é utilizada para identificar as portas GPIO em algumas linguagens de programação (como Python, Perl e Java ME). Já as colunas WiringPi apresentam os nomes das GPIOs conforme as mesmas serão identificadas pelo WiringPi. Podemos ver a diferença, por exemplo, no pino 3, que para o Raspberry é a GPIO 02, e para o WiringPi é a 8.

Em termos práticos, podemos dizer que se a aplicação utiliza o Java ME 8, ela seguirá o padrão do Raspberry Pi. Por sua vez, se a aplicação utilizar o Java SE Embedded, juntamente com o Pi4J, seguirá o padrão do WiringPi.

Meu primeiro dispositivo IoT

Para apresentarmos na prática o que foi estudado até aqui, a sequência deste artigo mostrará como criar um protótipo utilizando Raspberry Pi 2 e Java SE Embedded (com o Pi4J). O projeto resume-se a três LEDs de cores diferentes que irão acender de acordo com a luminosidade do ambiente.

A Lista de materiais para a construção desse protótipo é composta por:

- Um Raspberry Pi 2 (com a última versão do Raspian instalado – essa versão já vem com o Java SE Embedded);
- Uma protoboard;
- Dez fios (também conhecidos como jumpers, ou arames) Macho-Fêmea de aproximadamente 0,5 mm de diâmetro (encontrados em lojas de componentes eletrônicos);
- Três LEDs de 1,5v;
- Três resistores de 100Ω (para evitar que os LEDs queimem);
- Um sensor digital de luminosidade BH1750.

Nota

Uma boa fonte de fios (jumpers) é um cabo de rede que não esteja sendo utilizado. Você pode cortá-lo e desmontá-lo. Com isso, terá oito fios do comprimento que desejar. O único inconveniente é o contato dos fios com os pinos dos sensores ou do Raspberry Pi, que são conectores macho. Para evitar esse problema, o ideal é crimpá um conector fêmea em uma das pontas.

O ambiente de desenvolvimento embarcado

Com o atual poder de processamento do Raspberry Pi 2, não é de se estranhar que seja possível utilizar IDEs como o Eclipse no próprio dispositivo para viabilizar a implementação. No entanto, se o desenvolvedor optar por essa alternativa, a performance não será das melhores, pois por mais potente que seja o dispositivo, ele ainda não será páreo para processadores como um Intel Core i5 com alguns gigabytes de memória RAM. Portanto, a solução mais indicada para um ambiente de desenvolvimento embarcado é deixá-lo o mais simples possível, sem uma IDE, por exemplo, e utilizar um computador remoto, com mais recursos, para realizar a implementação propriamente dita.

Partindo do princípio que o código produzido em um desktop pode ser transportado para o ambiente embarcado via rede, se a plataforma utilizada para o sistema for o Java ME, existe um cliente para o Raspberry Pi (versões anteriores ao Pi 2) que sincroniza pela rede o programa gerado, podendo inclusive executá-lo remotamente.

WiringPi	RPi	Pino Físico		RPi	WiringPi
		1	2		
8	GPIO02	3	4		
9	GPIO03	5	6		
7	GPIO04	7	8	GPIO14	15
		9	10	GPIO15	16
0	GPIO17	11	12	GPIO18	1
2	GPIO27	13	14		
3	GPIO22	15	16	GPIO23	4
		17	18	GPIO24	5
12	GPIO10	19	20		
13	GPIO09	21	22	GPIO25	6
14	GPIO11	23	24	GPIO08	10
		25	26	GPIO07	11
30	ID_SD	27	28	ID_SC	31
21	GPIO05	29	30	GPIO12	26
22	GPIO06	31	32		
23	GPIO13	33	34	GPIO16	27
24	GPIO19	35	36		
25	GPIO26	37	38	GPIO20	28
		39	40	GPIO21	29

Tabela 1. Diferenças de nomenclatura para as portas GPIO entre o Raspberry Pi e o WiringPi



Na verdade, para o desenvolvedor, o Raspberry Pi aparece como um dispositivo acoplado ao computador. Já o Java SE Embedded não dispõe de tal artifício e os programas compilados no desktop devem ser transferidos através de outros protocolos (FTP, SCP, pasta compartilhada, etc.) e executados através de uma sessão SSH.

A distribuição do Linux para Raspberry, o Raspian, já traz um servidor SSH/SCP configurado, bem como uma versão do Java SE Embedded instalada. Porém, se a sua distribuição Linux não inclui a versão Embedded do Java instalada, basta fazê-lo acessando o Raspberry Pi via SSH e digitando:

```
sudo apt-get install oracle-java8-jdk
```

Uma vez instalado o Java, podemos testá-lo através do comando: `javac -version`.

O próximo passo para ter o ambiente configurado no Raspberry é instalar o WiringPi. Como essa instalação é um pouco mais complexa, exige a execução das seguintes etapas:

1) Instalar o Git-Core, que é um versionador de código. Para tanto, execute:

```
sudo apt-get install git-core
```

2) Baixar o código fonte da última versão do WiringPi, através do comando:

```
git clone git://git.drogon.net/wiringPi
```

3) Compilar e instalar o WiringPi, conforme o código:

```
cd wiringPi  
./build
```

Além disso, visto que no projeto deste artigo o sensor de luminosidade se comunica com o Raspberry Pi através do protocolo I²C, precisamos realizar alguns passos para poder utilizar esse protocolo. São eles:

1) Habilitar os módulos do kernel do Linux referentes ao I²C. Para isso, em um editor de textos de sua preferência, adicione as seguintes linhas ao arquivo `/etc/modules`:

```
i2c-bcm2708  
i2c-dev
```

2) Instalar os programas utilizados (`python-smbus` e `i2c-tools`) para a detecção de sensores I²C através do comando:

```
sudo apt-get install python-smbus i2c-tools
```

1) Editar o arquivo `/etc/modprobe.d/raspi-blacklist.conf` e certificar-se de que o mesmo esteja vazio;

2) Reiniciar o Raspberry utilizando o comando:

```
sudo reboot
```

Após a reinicialização do Raspberry Pi, teste se os devidos módulos foram habilitados, o que pode ser feito executando o comando `i2cdetect -y 1`. A Figura 4 ilustra o resultado, onde podemos notar que não existem dispositivos I²C conectados. As letras UU na coluna b, linha 10, indicam que o módulo do kernel foi carregado corretamente.

Com a configuração do Java, do WiringPi e do I²C, o ambiente embarcado está pronto para o desenvolvimento.

```
root@raspberrypi:/home/pi/pi4j# i2cdetect -y 1  
 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  
00: ---  
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- UU  
20: ---  
30: ---  
40: ---  
50: ---  
60: ---  
70: ---
```

Figura 4. Resultado da execução do comando `i2cdetect`

O ambiente de desenvolvimento no desktop

Como mencionado anteriormente, é possível programar para IoT utilizando apenas o Java SE Embedded, Raspberry Pi, as bibliotecas descritas na seção anterior e um editor de textos. No entanto, o poder de processamento de um Desktop e a agilidade propiciada pelas IDEs Java fazem com que essa seja uma opção mais interessante. Existe apenas um contraponto nesse modelo (Java SE Embedded sendo codificado de um desktop). Diferentemente do Java ME 8, que traz uma integração entre o Raspberry Pi e o desktop, para desenvolver utilizando o Java SE Embedded precisamos compilar o código no desktop e transferi-lo para o Raspberry Pi.

Nesse cenário pouco automatizado, uma boa opção é adotar o Maven. Além de fornecer uma forma de gerenciamento de bibliotecas, o Maven viabiliza plug-ins que podem empacotar a aplicação no formato desejado – no nosso caso, um JAR executável – e transferi-la para o destino – no nosso caso, o Raspberry Pi.

Para a criação de um projeto Maven que possa servir para o propósito deste artigo, basta utilizar o *wizard* de sua IDE preferida. A Figura 5 mostra a estrutura de um projeto Maven simples no Eclipse.

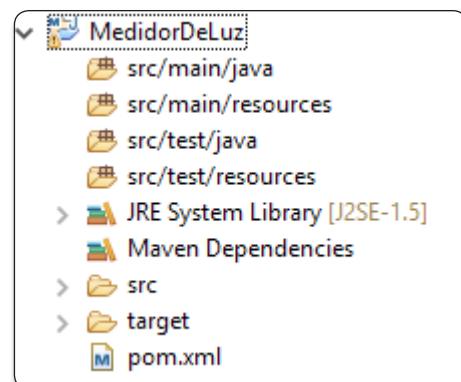


Figura 5. Estrutura de um projeto Maven sem archetype

Desenvolvimento IoT com o Java 8 e Raspberry Pi

Os principais componentes listados nessa figura são as pastas de código-fonte (*src/main/java* e *src/test/java*), as dependências do Maven e o principal, o arquivo *pom.xml*. Nele, conseguimos, dentre outras coisas, gerenciar plug-ins e bibliotecas das quais o projeto depende. A **Listagem 1** mostra o *pom.xml* configurado para o nosso exemplo.

Como o objetivo deste artigo não é destrinchar o Maven e seu funcionamento, vamos aos detalhes da **Listagem 1** pertinentes ao projeto. Após as linhas 3 a 5, que são referentes à identificação da aplicação, o primeiro conjunto de informações relevantes pode ser visto entre as linhas 6 e 22, que são responsáveis pelas dependências do projeto, no caso, as bibliotecas do Pi4J.

Listagem 1. Trecho do *pom.xml* com informações do projeto e declaração das dependências.

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                               http://maven.apache.org/xsd/maven-4.0.0.xsd">
02   <modelVersion>4.0.0</modelVersion>
03   <groupId>exemplo.java.magazine </groupId>
04   <artifactId>MedidorDeLuz</artifactId>
05   <version>0.0.1</version>
06   <dependencies>
07     <dependency>
08       <groupId>com.pi4j</groupId>
09       <artifactId>pi4j-core</artifactId>
10      <version>1.0.</version>
11    </dependency>
12    <dependency>
13      <groupId>com.pi4j</groupId>
14      <artifactId>pi4j-gpio-extension</artifactId>
15      <version>1.0.</version>
16    </dependency>
17    <dependency>
18      <groupId>com.pi4j</groupId>
19      <artifactId>pi4j-device</artifactId>
20      <version>1.0.</version>
21    </dependency>
22  </dependencies>
```

Já entre as linhas 23 e 96, apresentadas na **Listagem 2**, temos a configuração dos plug-ins do Maven para gerar o JAR executável. Normalmente, projetos utilizam um ou dois plug-ins (para compilação e empacotamento), mas para este caso, serão utilizados os três plug-ins listados a seguir:

- **Maven-Shade:** responsável por transformar o produto da compilação, um JAR não executável, em um JAR executável. Esse plug-in é configurado na linha 38 através da tag *<mainClass>*, que recebe o nome completo da classe que possui o método *main()*;
- **Maven-Compiler:** responsável por compilar a aplicação. Não requer qualquer configuração extra;
- **Maven-Antrun:** plug-in que faz uso da biblioteca Ant para executar tarefas do processo de build. No caso do *pom.xml* apresentado na **Listagem 1**, esse plug-in é responsável pela transferência do artefato gerado pelo Maven-Compiler para o Raspberry Pi.

Ainda sobre as configurações pertinentes aos plug-ins, as linhas 61 e 62 são utilizadas pelo Maven-Antrun para criar uma pasta no Raspberry Pi e em seguida copiar o recém-gerado arquivo JAR para ela. As variáveis declaradas nessas linhas, com exceção de *project.build.directory* e *project.build.finalName*, que são variáveis de ambiente do Maven, são definidas no formato de tags dentro da tag *<properties>*; no exemplo, entre as linhas 97 e 103.

Para refletir as configurações do seu ambiente, altere as variáveis de conexão SCP, descritas nas linhas 97 a 103, de acordo com a sua configuração de rede, usuário e senha para se conectar ao Raspberry Pi.

Ao salvar essa configuração do *pom.xml*, o Eclipse baixará as dependências e a seção *Maven Dependencies* deverá ficar semelhante à exposta na **Figura 6**.

Com o projeto do Eclipse devidamente configurado, uma vez que o código-fonte esteja pronto, basta executá-lo com o comando *Maven Install*, conforme o exemplo apresentado na **Figura 7**.

Assim, concluímos a configuração do ambiente de desenvolvimento no desktop.

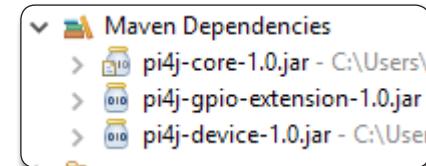


Figura 6. Bibliotecas do Pi4J baixadas pelo Maven

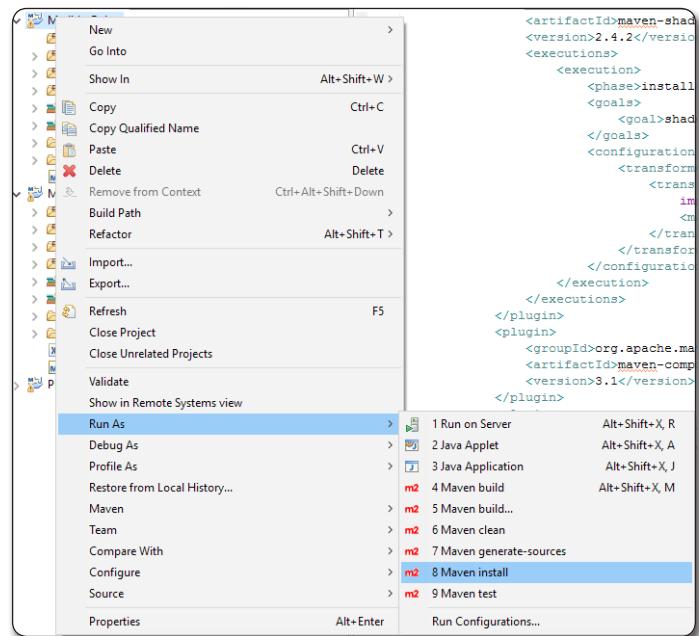


Figura 7. Executando o projeto com o Maven

O projeto eletrônico

Conforme mencionado na introdução deste artigo, para desenvolver um projeto de IoT deve-se ir além do software.

É necessário, também, um mínimo de conhecimento em eletrônica, para que seja possível compreender a estrutura montada na

Listagem 2. Trecho do pom.xml com as configurações dos plug-ins.

```

23 <build>
24   <plugins>
25     <plugin>
26       <groupId>org.apache.maven.plugins</groupId>
27       <artifactId>maven-shade-plugin</artifactId>
28       <version>2.4.2</version>
29     <executions>
30       <execution>
31         <phase>install</phase>
32         <goals>
33           <goal>shade</goal>
34         </goals>
35       <configuration>
36         <transformers>
37           <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
38             <mainClass>magazine.java.Principal</mainClass>
39           </transformer>
40         </transformers>
41       </configuration>
42     </execution>
43   <executions>
44   </plugin>
45   <plugin>
46     <groupId>org.apache.maven.plugins</groupId>
47     <artifactId>maven-compiler-plugin</artifactId>
48     <version>3.1</version>
49   </plugin>
50   <plugin>
51     <artifactId>maven-antrun-plugin</artifactId>
52   <executions>
53     <execution>
54       <id>id-qualquer</id>
55       <phase>install</phase>
56       <goals>
57         <goal>run</goal>
58       </goals>
59     <configuration>
60       <tasks>
61         <sshexec host="${pi.host}" port="${pi.port}"
62           username="${pi.user}" password="${pi.pwd}"
63           trust="true" command="mkdir -parents ${pi.pasta}" />
64         <scp file="${project.build.directory}/${project.build.finalName}.jar"
65           todir="${pi.user}:${pi.pwd}@${pi.host}:${pi.pasta}" />
66       </tasks>
67     </configuration>
68   </executions>
69   </plugin>
70 </build>
71 <properties>
72   <pi.host>192.168.137.246</pi.host>
73   <pi.port>22</pi.port>
74   <pi.user>pi</pi.user>
75   <pi.pwd>123</pi.pwd>
76   <pi.pasta>/home/pi/pi4j</pi.pasta>
77 </properties>
78 </project>

```

Figura 8, na qual podemos identificar uma Protoboard com os componentes ligados via cabos ao Raspberry Pi.

Ainda sobre essa figura, na Protoboard podemos identificar o sensor de luminosidade (BH1750), os LEDs (Vermelho, Amarelo e Verde) e três resistores.

O sensor BH1750 tem as conexões feitas da seguinte maneira:

- VCC: pino 3,3V;
- GND: pino Terra;
- SCL: GPIO 8;
- SDA: GPIO 9.

Um ponto que demanda um pouco mais de atenção está relacionado ao conjunto de LEDs, pois ele precisa de um certo cuidado. Note que cada led possui duas “pernas”, uma mais comprida que a outra. A mais curta deve estar conectada ao barramento do resistor e este, por sua vez, será conectado em um pino Terra. A razão pela qual precisamos de resistores neste projeto é que o resistor baixa a voltagem de 3,3v (fornecida pelo Raspberry) para 1,5v, voltagem que os LEDs utilizam. Já a “perna” mais comprida dos LEDs deve ser conectada diretamente ao Raspberry, na seguinte ordem:

Desenvolvimento IoT com o Java 8 e Raspberry Pi

- Vermelho: GPIO 0;
- Amarelo: GPIO 1;
- Verde: GPIO 2.

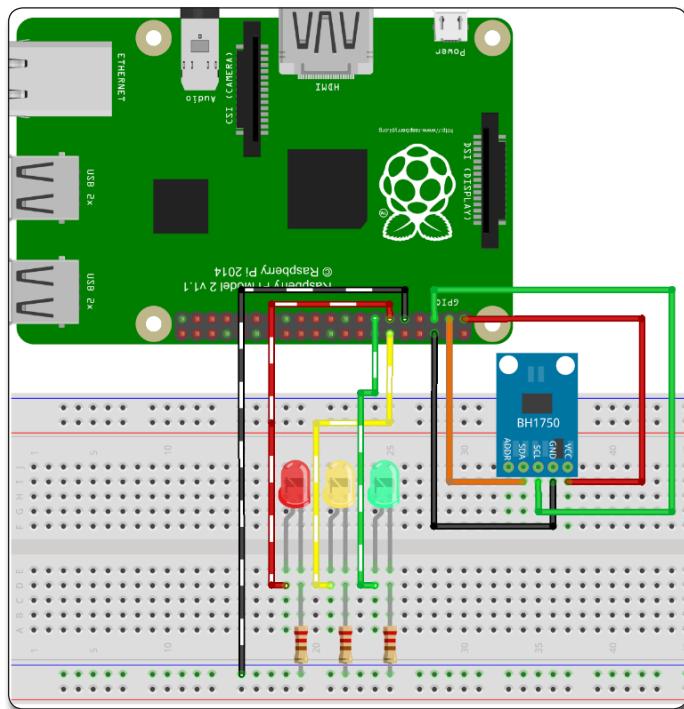


Figura 8. Projeto montado em uma Protoboard

Nota

O diagrama eletrônico é parte da documentação do seu projeto. Portanto, faça o desenho, como na **Figura 8**, para seus protótipos e guarde-o, pois ele é extremamente útil na hora de remontar seu projeto. Para criar um diagrama eletrônico podemos utilizar várias ferramentas. Neste artigo, optamos pela Fritzing, por ser gratuita, possuir uma série de componentes pré-instalados e permitir a importação de outros.

Com os LEDs devidamente conectados ao Raspberry Pi, é possível fazer um teste simples para checar se eles estão funcionando. Deste modo, abra uma conexão SSH (utilizando o seu software de SSH preferido, por exemplo, o Putty para Windows ou o SSH para Linux) com o aparelho. Uma vez estabelecida a conexão, execute os passos a seguir:

1. Configure a porta GPIO para escrita. No comando, a porta 0 é configurada para escrita:

```
sudo gpio mode 0 out
```

2. Ligue ou desligue a porta. No comando a seguir, o valor 0 representa a porta e o valor 1, o valor ligado (0 para desligar):

```
sudo gpio write 0 1
```

Se tudo correr bem, o LED vermelho será ativado.

Para verificar se o sensor BH1750 foi reconhecido pelo Linux, o comando `i2cdetect -y 1` deve ser executado em um terminal. O resultado dessa execução retorna o endereço utilizado pelo sensor, como demonstra a **Figura 9**. Neste comando, o parâmetro “1” representa o barramento no qual a busca será feita. Saiba, no entanto, que o Raspberry Pi trabalha com dois barramentos para o I²C, sendo que o mais utilizado geralmente é o 1. Porém, em algumas versões mais antigas esse número pode ser o 0.

O valor mostrado na linha 20, coluna 3, é a representação hexadecimal do endereço utilizado pelo sensor. Esse valor será reaproveitado no código-fonte da aplicação para referenciar o dispositivo e assim efetuar as leituras necessárias para o projeto.

Implementando a aplicação

Uma vez que temos o ambiente configurado e os componentes eletrônicos ligados, estamos prontos para a última etapa do projeto, o software.



Listagem 3. Código da classe Principal.

```
01 package magazine.java;
02
03 import java.math.BigInteger;
04 import com.pi4j.io.i2c.I2CBus;
05 import com.pi4j.io.i2c.I2CDevice;
06 import com.pi4j.io.i2c.I2CFactor;
07 import com.pi4j.wiringpi.Gpio;
08
09 public class Principal {
10     public static void main(String[] args) {
11         Gpio.wiringPiSetup();
12         try {
13             final int ledR = 0;
14             final int ledY = 1;
15             final int ledG = 2;
16             Gpio.pinMode(ledR, Gpio.OUTPUT);
17             Gpio.pinMode(ledY, Gpio.OUTPUT);
18             Gpio.pinMode(ledG, Gpio.OUTPUT);
19             Gpio.digitalWrite(ledR, false);
20             Gpio.digitalWrite(ledY, false);
21             Gpio.digitalWrite(ledG, false);
22
23             I2CBus bus = I2CFactor.getInstance(I2CBus.BUS_1);
24             I2CDevice device = bus.getDevice(0x23);
25
26             Runtime.getRuntime().addShutdownHook(new Thread() {
27                 public void run() {
28                     Gpio.digitalWrite(ledR, false);
29                     Gpio.digitalWrite(ledY, false);
30                     Gpio.digitalWrite(ledG, false);
31                 }
32             });
33             while (true) {
34                 byte buffer[] = new byte[2];
35                 device.read(0x23, buffer, 0, 2);
36                 BigInteger bigInt = new BigInteger(buffer);
37                 int valor = bigInt.intValue();
38                 Gpio.digitalWrite(ledR, valor > 10);
39                 Gpio.digitalWrite(ledY, valor > 1000);
40                 Gpio.digitalWrite(ledG, valor > 10000);
41                 System.out.print("\r" + valor);
42                 Thread.sleep(200);
43             } catch (Exception e) {
44                 e.printStackTrace();
45             }
46         }
47     }
}
```

Por ser um exemplo pequeno, o código-fonte desse projeto é relativamente simples. Excluindo o trecho relacionado à inicialização dos componentes do Pi4J, o restante é apenas leitura de valores do sensor de luminosidade e, a partir disso, ligar ou desligar os LEDs. A **Listagem 3** mostra o código da classe **Principal**.

O primeiro ponto que chama a atenção nessa listagem é a linha 11. Nela podemos observar a chamada ao método **wiringPiSetup()** da classe **Gpio**, classe que representa o conjunto de portas GPIO como um todo e tem métodos e constantes estáticas para configurar e manipular as portas de comunicação. O método **wiringPiSetup()** faz a inicialização da ponte com o WiringPi e, consequentemente, com os mecanismos de manipulação das portas do Linux. Caso esse método não seja invocado na aplicação, ela se comportará de maneira inadequada, apresentando erros de I/O, falhas de segmentação, entre outros. Saiba que a chamada ao método **wiringPiSetup()** deve ser feita apenas uma vez.

Após a inicialização da interface com o WiringPi, já é possível utilizar qualquer porta GPIO, como o demonstrado nas linhas 16 a 18, onde, através de chamadas ao método **pinMode()**, é feita a configuração das portas (correspondentes à numeração utilizada pelo WiringPi) para o modo Output, ou seja, escrita. Uma vez que é permitida a escrita para as GPIO correspondentes aos LEDs, seus valores são prontamente inicializados como desligados, conforme nos mostram as linhas 19 a 21 com chamadas ao método **digitalWrite()** da classe **Gpio**.

Com os LEDs devidamente configurados, o próximo passo é a leitura do sensor de luminosidade. Para isso, na linha 23, temos a inicialização de um barramento de I²C (conforme o comando utilizado com o programa *i2cdetect*, pode ser o barramento 1 ou 0), e na linha seguinte, 24, a conexão com o dispositivo, criando-

se uma instância da classe **I2CDevice**. Observe que o parâmetro passado para o método **getDevice()** é o valor **0x23**, o mesmo que foi retornado como resultado da execução do *i2cdetect*.

Nas linhas 25 a 31, não há nada de diferente do Java SE. Em suma, temos um tratamento para o comando CTRL+C, ou um comando de interrupção da aplicação no console. Quando a aplicação é terminada dessa maneira, é uma boa ideia desligar os LEDs, caso contrário, os mesmos permanecerão ligados.

O próximo e último passo da aplicação é a leitura propriamente dita dos valores do sensor e a configuração dos LEDs de acordo com essa leitura. Na linha 34, o valor é lido para um array de bytes através da chamada ao método **read()**. Neste ponto, diferentemente do que você pode estar pensando, o parâmetro **0x23**, passado para esse método, não é o endereço do dispositivo, mas sim uma configuração do sensor para definir a sua resolução de leitura; neste caso, baixa resolução (lê de 4 em 4 unidades de medida). Para mais detalhes, deve-se consultar a documentação do sensor (veja a seção **Links**). Em seguida, na linha 36, o valor carregado no array de bytes é convertido para um valor inteiro, que corresponde a quantos lux (unidade de medida que indica a luminosidade por metro quadrado) o sensor conseguiu ler. Se o resultado for maior que 10 lux, a luz vermelha acenderá, se for maior que 1.000, a amarela, e se for maior que 10.000, a verde.

Com o código pronto, execute o programa através do Maven conforme ilustrado na **Figura 7**. Se tudo correr bem, uma mensagem de sucesso será exibida no console, como mostra a **Figura 10**.

Essa figura, além da resposta de sucesso, nos informa para onde foi copiado o arquivo JAR contendo a aplicação. No caso desse projeto, o artefato foi copiado para **/home/pi/pi4j**, conforme especificado no **pom.xml**.

Desenvolvimento IoT com o Java 8 e Raspberry Pi

```
<terminated> MedidorLuzInstall [Maven Build] C:\Program Files\Java\jdk1.8.0_60\bin\javaw.exe
[sshexec] Connecting to 192.168.100.3:22
[sshexec] cmd : mkdir --parents /home/pi/pi4j
[scp] Connecting to 192.168.100.3:22
[scp] done.
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 7.795 s
```

Figura 10. Resultado da execução do install do Maven

```
pi@raspberrypi:~/pi4j $ sudo java -jar MedidorLuz-0.0.1.jar
44
```

Figura 11. Aplicação leu 44 luxes

Agora, para testar se a aplicação realmente funciona, conecte-se ao Raspberry Pi usando SSH e execute os seguintes passos:

1) Acesse a pasta onde foi copiado o JAR:

```
cd pi4j
```

2) Execute a aplicação como super-usuário (root):

```
sudo java -jar MedidorLuz-0.0.1.jar
```

Como definido pelas configurações no *pom.xml*, o nome do JAR é composto pelo nome do artefato (tag **artifactId**) e pela versão (tag **version**).

O resultado da execução da aplicação, além dos LEDs acesos, é a impressão no console da quantidade de luxes lidos (vide Figura 11).

Neste artigo foi apresentado um exemplo simples de Internet das Coisas com o objetivo de servir como uma introdução a um vasto campo que abrange projetos de monitoração, automação, interação com humanos, com máquinas e uma série de outras aplicações. Além da quantidade de diferentes projetos, o número de dispositivos que são considerados como parte da IoT faz com que existam inúmeras maneiras de se resolver os mesmos problemas.

Assim como este conteúdo baseou-se no Raspberry Pi e no Java SE Embedded, poderíamos ter utilizado o Java ME 8 no mesmo Raspberry, ou mesmo ter desenvolvido o projeto todo em Arduíno.

Para que pudéssemos expor os conceitos básicos de IoT, o exemplo desenvolvido assemelha-se a um “Olá Mundo”. Agora, com os conhecimentos adquiridos, é possível ir além e criar seus próprios protótipos explorando os recursos de outros sensores, motores e demais componentes. Enfim, as possibilidades que a Internet das Coisas nos traz são “infinitas” e a limitação vai mais da criatividade de quem desenvolve o projeto do que propriamente do domínio da tecnologia, pois esta é, relativamente, simples.

Autor



Paulo M. D. Bordin

paulobordin@gmail.com

Graduado em Tecnologia em Informática pelo CEFET-PR, Pós-Graduado em Tecnologia Java pela UTF-PR. Certificado SCJP (1.4), SCWCD (1.5) e SCMAD. Atualmente é líder de projetos no HSBC Global Technologies e professor do curso de Especialização em Tecnologia Java e Desenvolvimento para Dispositivos Móveis da UTF-PR.



Links:

The Internet of Things is a Revolution Waiting to Happen.

<http://www.gartner.com/smarterwithgartner/the-internet-of-things-is-a-revolution-waiting-to-happen/>

Página da ferramenta Fritzing.

<http://fritzing.org/>

Página da biblioteca WiringPi.

<http://wiringpi.com/>

Endereço do projeto Pi4J.

<http://pi4j.com/>

Página do sensor BH1750.

<http://www.instructables.com/id/BH1750-Digital-Light-Sensor/>

Como melhorar o desempenho de suas aplicações JSF

Aprenda neste artigo o que fazer para deixar sua aplicação JSF muito mais rápida

Aplicações web são essenciais para praticamente qualquer empresa. Ao oferecer uma solução prática e eficaz para realizar a interface com o usuário, sem a necessidade de instalação de um aplicativo do lado do cliente, tais aplicações possibilitam uma maior mobilidade, facilidade de acesso e menor custo.

Buscando trazer essas vantagens para a linguagem Java, a tecnologia JavaServer Faces (JSF) surgiu como uma alternativa ao JSP, tecnologia até então padrão para a implementação de aplicações web em Java.

Ao viabilizar recursos até o momento inexistentes e uma API de fácil utilização, o JSF se firmou no mercado como uma tecnologia de referência, se tornando uma ótima opção para o desenvolvimento web. Indo mais além, o framework JSF implementa o modelo MVC (*Model-View-Controller*), ou seja, permite que seu código seja organizado em três camadas: modelo, visão e controle, garantindo assim um desacoplamento e uma simplicidade maior para as tarefas de manutenção e evolução.

Porém, mesmo com todas essas vantagens, ainda é necessário um bom domínio dessa tecnologia e um desenvolvimento cuidadoso, com muita atenção aos detalhes, para que se consiga implementar um sistema otimizado e com boa performance.

Neste momento é válido ressaltar que muitos dos problemas de performance das primeiras versões foram resolvidos na versão 2 da especificação. Essas melhorias foram possíveis graças a novos recursos, como *Partial State Saving*, *Tree Visiting* e *Project Stage*, o que tornou mais simples implementar um código realmente otimizado, que resulta em:

- Menor consumo de memória e CPU;
- Melhor gerenciamento do estado de cada componente;
- Otimização no algoritmo de busca de componentes;
- Otimização na serialização da árvore de componentes.

Fique por dentro

Este artigo descreve como otimizar o desempenho de aplicações web que utilizam a tecnologia JavaServer Faces, melhorando o tempo de resposta para que elas consigam suportar um número maior de requisições. Para isso, serão analisados detalhes das configurações da tecnologia e quando se deve optar por uma configuração ou outra. No artigo, será utilizada a versão 2.2 do JSF, pois foram introduzidos novos conceitos, como visões sem estado e fluxo de página, que também podem ajudar significativamente na melhora do desempenho.



Dito isso, o objetivo deste artigo será orientar como resolver os principais problemas de performance enfrentados pelos desenvolvedores e expor soluções que a tecnologia JSF oferece, a partir de pequenos exemplos de código. Inicialmente, no entanto, iremos analisar alguns conceitos fundamentais do JSF, referentes à árvore de componentes, para que o desenvolvedor possa ficar a par de como esse framework funciona.

Árvore de componentes

Em JSF, a árvore de componentes é um conceito fundamental. A construção dessa árvore é baseada no código de uma página JSF (*view*), código esse que utiliza a linguagem de marcação XHTML. Após o código ser analisado, uma instância de cada componente JSF é criada e a árvore de componentes é construída. A partir disso, essa estrutura será utilizada ao longo de todo o ciclo de vida do JSF, ciclo esse que é definido pelas seguintes fases:

- **Restaurar View:** Nesta fase a árvore de componentes é criada ou restaurada, caso já tenha sido criada em uma requisição anterior;
- **Aplicar valores:** Os componentes JSF são atualizados com os valores enviados na requisição, por exemplo entradas de formulário;

- **Conversão / Validação:** Conversores e validadores são aplicados sobre os valores dos componentes de entrada (*inputs*) da página;

- **Atualização do modelo:** Se a validação for bem-sucedida, os valores dos componentes de entrada são atribuídos às propriedades do *bean*. Isso significa que os métodos *setters* dos *beans* serão chamados;

- **Invocar aplicação:** Esta fase é onde a lógica da aplicação será executada, isto é, onde os métodos dos *managed beans* são executados;

- **Criar a resposta:** Fase responsável por encontrar a página (*view*) de resposta, percorrer a árvore de componentes e processar cada componente JSF para criar um HTML de resposta para o cliente.

Como vimos, a árvore de componentes está envolvida em todas as fases do ciclo de vida, sendo percorrida várias vezes. Isso sugere que a quantidade de componentes JSF presente em cada uma tem um impacto significativo sobre o tempo de resposta, e consequentemente, sobre o desempenho da aplicação. Com essas informações, podemos assumir que uma árvore de componentes de pequeno porte é benéfica para ter um tempo de resposta mais curto. Tal afirmação pode ser confirmada no gráfico da **Figura 1**, que mede o tempo de duração de cada

fase, com diferentes tamanhos de árvores. Note que quanto maior a quantidade de componentes JSF em uma página, maior é o tempo gasto em cada fase do ciclo de vida.

Otimizando sua aplicação

A primeira e significativa mudança que pode ser realizada, caso sua aplicação ainda não a utilize, é alterar a implementação do JSF para o Apache MyFaces versão 2.2.9 (última versão até o momento da escrita deste artigo), pois essa é a opção que possui melhor performance. Podemos confirmar isso analisando um comparativo de desempenho — apresentado na **Figura 2** — entre o Apache MyFaces e o Mojarra. Para utilizar essa versão do Apache MyFaces, saiba que é necessário no mínimo o Java 1.6, JSP 2.1, JSTL 1.2 e Java Servlet 2.5.

Otimizando o MyFaces

Existem muitas ações que podem ser realizadas para fazer com que o Apache MyFaces apresente um melhor desempenho. A primeira que iremos apresentar está relacionada à definição dos recursos utilizados em uma página. Recurso, nesse caso, pode ser compreendido como qualquer arquivo referenciado na página, como arquivos JavaScript, CSS, imagens, etc.

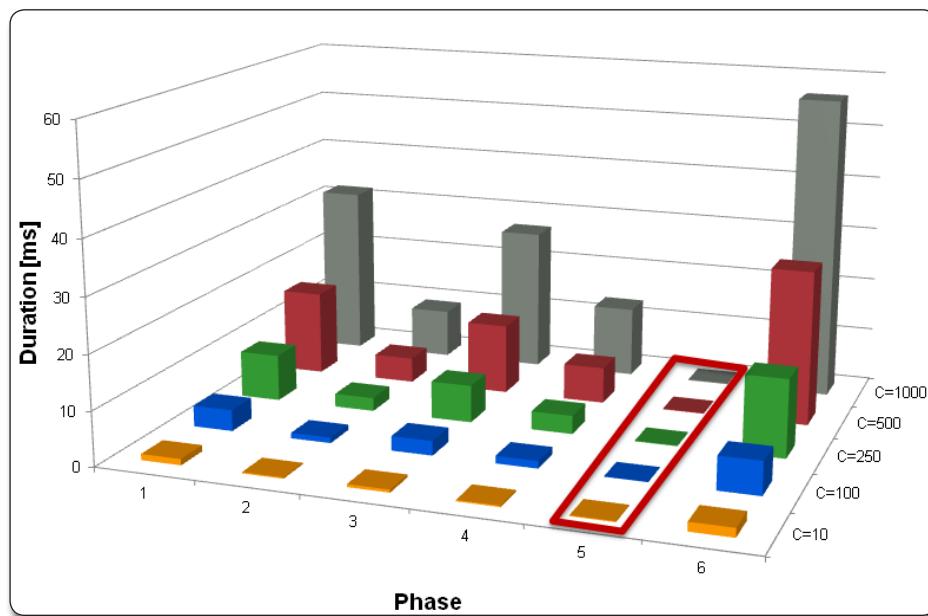


Figura 1. Duração das fases em relação ao tamanho da árvore. Fonte: blog.ocio.de



Para evitar que um mesmo recurso seja referenciado mais de uma vez em uma página e consequentemente sejam realizadas requisições desnecessárias, o MyFaces oferece a classe **MyFacesExtensionsFilter**, que é um *servlet filter* (veja o **BOX 1**) responsável por realizar o *buffer* e *parse* das respostas. Entre os objetivos da **MyFacesExtensionsFilter** estão:

1. Recuperar recursos estáticos, como arquivos de imagem, JavaScript, CSS, etc.;
2. Gerar as URLs desses recursos e injetá-los na tag HTML `<head>`, de forma transparente para o desenvolvedor, garantindo que um mesmo recurso seja referenciado uma única vez na página.

Infelizmente, essa solução também tem suas desvantagens. Para o objetivo 1, o *filter* funciona muito bem. Por outro lado, para o objetivo 2, ele pode causar um efeito colateral — o uso excessivo de memória — ao carregar toda a resposta em memória para realizar o processamento necessário. Consequentemente, isso acaba influenciando no desempenho do sistema, além de aumentar o tempo de resposta das requisições.

BOX 1. Servlet Filter

Na API de Servlets há um tipo de componente chamado *filter*. Um *filter* (ou filtro, em português) é responsável por interceptar requisições e respostas para transformar ou utilizar as informações contidas nas mesmas. Filtros normalmente não são responsáveis por criar respostas, mas sim prover funções universais, que podem ser utilizadas por toda a aplicação. Para criar um *Servlet Filter* deve-se implementar a interface `javax.servlet.Filter`.

Com base nisso, caso sua aplicação utilize o **MyFacesExtensionsFilter** e passe a apresentar problemas de performance, podemos realizar uma otimização para remover esse filtro da configuração do `web.xml` e ainda assim manter todos os benefícios. Para isso, é necessário realizar dois passos:

1. Adicionar o código da **Listagem 1** ao `web.xml`. Esse código configura um parâmetro de contexto do MyFaces chamado `org.apache.myfaces.ADD_RESOURCE_CLASS` com o valor `org.apache.myfaces.component.html.util.StreamingAddResource`.

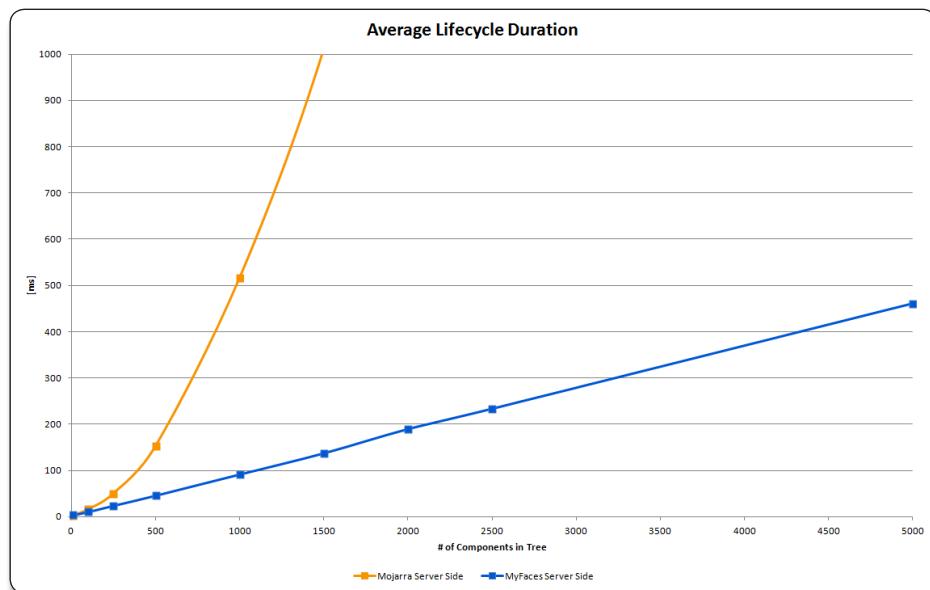


Figura 2. Comparativo do tempo de execução do ciclo de vida do JSF entre Apache MyFaces e Mojarra. Fonte: blog.io.de

Listagem 1. Configuração no web.xml para utilizar a classe StreamingAddResource.

```
<context-param>
    <param-name>org.apache.myfaces.ADD_RESOURCE_CLASS</param-name>
    <param-value>
        org.apache.myfaces.component.html.util.StreamingAddResource
    </param-value>
</context-param>
```

Listagem 2. Exemplo de um código XHTML que utiliza a tag <t:documentHead>.

```
<f:view>
    <t:document>
        <t:documentHead>
            ...
            <title>example</title>
            ...
        </t:documentHead>
        <t:documentBody>
            ...
        </t:documentBody>
    </t:document>
</f:view>
```

source. `StreamingAddResource` é uma classe utilitária que permite que um componente JSF registre os recursos estáticos (como arquivos JavaScript, CSS, imagens etc.) que ele necessita, para que esses recursos sejam adicionados à página. Um dos diferenciais dessa opção é que quando vários componentes registram a necessidade para o mesmo recurso, somente uma referência para o mesmo é inserida na página;

2. Substituir a tag HTML `<head>` do seu arquivo JSF pela tag `<t:documentHead>`, lembrando que essa nova tag deve estar

dentro de `<f:view>` para que funcione corretamente. Veja um exemplo na **Listagem 2**.

Realizadas essas alterações, conseguimos reduzir a utilização de memória e eliminar requisições desnecessárias ao servidor.

Alterando o método de salvar estado

Em aplicações JSF, o estado das páginas que o usuário está acessando, por padrão, é sempre armazenado, seja do lado servidor (*server-side*) ou do lado cliente (*client-side*). Alterar o mecanismo de salvar o estado de *server-side* para *client-side* move os dados

salvos do servidor para o cliente. Isso reduz a memória necessária no servidor, em contrapartida, aumenta a necessidade de largura de banda, devido à troca de estado a cada requisição. Ademais, salvar o estado do lado cliente também exige que a resposta seja desserializada para cada requisição HTTP, o que significa um tempo de resposta maior ao adotar essa opção. Com base nisso, levando em consideração o tempo de resposta, é interessante configurar sua aplicação para utilizar o método de salvar estado no servidor. Porém, ao fazer isso, mais memória do servidor será necessária para suportar a aplicação.

Então, é essencial fazer uma análise e verificar qual dos dois modos se encaixa melhor no cenário da sua aplicação. Basicamente, você deve levar em conta se o mais importante é o tempo de resposta (utilizar o modo *server-side*) ou se o essencial é reduzir a quantidade de recursos necessários no servidor para suportar sua aplicação (utilizar o modo *client-side*).

A **Listagem 3** demonstra como realizar essa configuração, que é feita definindo o parâmetro de contexto **javax.faces.STATE_SAVING_METHOD** no *web.xml* com o valor **server** ou **client**.

Listagem 3. Configuração do *web.xml* para utilizar o método server side para salvar o estado.

```
<context-param>
<param-name>javax.faces.STATE_SAVING_METHOD</param-name>
<param-value>server</param-value>
</context-param>
```



Ainda relacionado a isso, deve-se ficar atento com relação à configuração da quantidade máxima de páginas salvias na sessão para cada usuário que acessa o sistema. Essa configuração pode ser alterada adicionando o parâmetro de contexto **org.apache.myfaces.NUMBER_OF_VIEWS_IN_SESSION**, mudando o valor para um número que seja mais adequado ao seu contexto (vide **Listagem 4**). Por padrão, o valor é **20**.

Listagem 4. Configuração do *web.xml* para definir a quantidade de views salvias na sessão.

```
<context-param>
<param-name>
org.apache.myfaces.NUMBER_OF_VIEWS_IN_SESSION
</param-name>
<param-value>10</param-value>
</context-param>
```

Sessão do usuário

Outro ponto importante, com relação ao desempenho de um sistema que utiliza JSF, é a serialização do estado do usuário na sessão, pois, como citado anteriormente, a serialização e desserialização da árvore de componentes tem um grande impacto na performance e no uso de recursos do servidor (lembrando que isso é válido somente quando configurarmos o método de salvar estado como **server**, pois, caso contrário, os dados serão serializados do lado do cliente). Quando habilitada essa opção, o estado será serializado em memória na sessão do usuário. Porém, para tentar reduzir o tempo de processamento e utilização de memória do servidor, é possível desabilitar esse mecanismo. Para isso, basta adicionar outro parâmetro de contexto no arquivo *web.xml*, chamado **org.apache.myfaces.SERIALIZE_STATE_IN_SESSION**, e defini-lo com o valor **false**, como expõe a **Listagem 5**. Ao optar por esse caminho, o desenvolvedor deve ficar ciente que a sessão do usuário não vai ser persistida em disco, e caso o servidor onde a aplicação executa seja reiniciado, a sessão será perdida.

Listagem 5. Configuração do *web.xml* para desabilitar a serialização do estado na sessão.

```
<context-param>
<param-name>
org.apache.myfaces.SERIALIZE_STATE_IN_SESSION
</param-name>
<param-value>false</param-value>
</context-param>
```

Como já mencionado ao longo deste artigo, o JSF realiza várias serializações durante o ciclo de vida, para cada página (*view*) requisitada. Sabendo disso, constatamos que é muito importante que esse processo seja realizado o mais rápido e eficientemente possível. Pensando nisso, o MyFaces oferece a possibilidade de o desenvolvedor alterar o mecanismo de serialização da aplicação. Neste ponto vale ressaltar que essa solução não pode ser aplicada se optarmos pelo método de salvar estado do lado do cliente

(client-side), pois nesse caso nenhuma serialização é realizada pela aplicação.

Neste artigo vamos mostrar uma opção de como melhorar a performance do processo de serialização do JSF utilizando a biblioteca **JBoss Serialization**. Essa é uma opção simples e eficiente que informa ser possível obter um desempenho pelo menos duas vezes melhor, se comparado com o mecanismo padrão de serialização do Java (Fonte: serialization.jboss.org), padrão esse que pode provocar picos de utilização de CPU e causar degradação e redução da capacidade de escalabilidade do sistema.

Para definir um novo mecanismo de serialização, é preciso criar uma classe que implemente a interface **SerialFactory** do MyFaces, interface essa que, por contrato, torna necessário implementar dois métodos: **getOutputStream()** e **getInputStream()**. O primeiro, responsável por realizar a serialização, recebe um objeto do tipo **InputStream** como parâmetro e retorna um objeto do tipo **ObjectInputStream**. Já o segundo, responsável pela desserialização, recebe um objeto **OutputStream** e retorna um objeto do tipo **ObjectOutputStream**.

A **Listagem 6** apresenta um exemplo de código onde foi criada uma classe chamada **MySerialFactory** para realizar a serialização e desserialização de objetos.

Listagem 6. Código da classe **MySerialFactory**, implementação da interface **SerialFactory**.

```
01. package org.example.jsf;
02.
03. import java.io.IOException;
04. import java.io.InputStream;
05. import java.io.ObjectInputStream;
06. import java.io.ObjectOutputStream;
07. import java.io.OutputStream;
08.
09. import org.apache.myfaces.shared_impl.util.serial.SerialFactory;
10. import org.jboss.serial.io.JBossObjectInputStream;
11. import org.jboss.serial.io.JBossObjectOutputStream;
12.
13. public class MySerialFactory implements SerialFactory {
14.
15.     @Override
16.     public ObjectInputStream getObjectInputStream(InputStream stream)
17.             throws IOException {
18.         return new JBossObjectInputStream(stream);
19.     }
20.
21.     @Override
22.     public ObjectOutputStream getObjectOutputStream(
23.             OutputStream stream) throws IOException {
24.         return new JBossObjectOutputStream(stream);
25.     }
}
```

Feito isso, é necessário definir um parâmetro de contexto no *web.xml* da aplicação, parâmetro esse que diz qual é o novo método de serialização a ser utilizado. O nome desse parâmetro é **org.apache.myfaces.SERIAL_FACTORY**, e o valor deve ser o nome da classe que acabamos de criar: **MySerialFactory**. A **Listagem 7** apresenta

essa configuração. O arquivo JAR do JBoss Serialization pode ser encontrado no próprio site da JBoss (veja a seção **Links**).

Nesse caso, utilizamos um mecanismo de serialização da JBoss, mas você pode ficar à vontade para escolher outro mecanismo que considere mais eficiente.

Buffer Size

Também podemos tentar reduzir o tempo de renderização de uma página aumentando o tamanho do buffer de resposta.

Essa decisão diminui a quantidade de *flushes* necessários durante o processo de renderização, o que pode gerar um ganho de performance no tempo de criação da resposta. Para isso, deve-se definir dois parâmetros de contexto, chamados **com.sun.faces.responseBufferSize** e **facelets.BUFFER_SIZE** (esse segundo parâmetro só é válido caso você utilize o componente facelets), conforme a **Listagem 8**.

Listagem 7. Configuração do *web.xml* para utilizar a nova classe de serialização

```
<context-param>
<param-name>org.apache.myfaces.SERIAL_FACTORY</param-name>
<param-value>org.example.jsf.MySerialFactory</param-value>
</context-param>
```

Listagem 8. Configuração do *web.xml* para definir o tamanho do buffer de resposta.

```
<context-param>
<param-name>com.sun.faces.responseBufferSize</param-name>
<param-value>50000</param-value>
</context-param>
<context-param>
<param-name>facelets.BUFFER_SIZE</param-name>
<param-value>50000</param-value>
</context-param>
```

Saiba, no entanto, que não existe um número mágico que possa ser declarado para todas as aplicações. Portanto, é importante que o desenvolvedor realize algumas medições para definir o valor ideal. Para isso, é aconselhado realizar várias medições de tempo de resposta, com diferentes tamanhos de *buffer*, começando com um valor bem pequeno e aumentando gradativamente. O número ideal é identificado quando o tempo de resposta começa a diminuir.

Compress State in Session

Por fim, também é possível desabilitar a compressão dos dados ao salvar o estado, pois esse processo consome um tempo considerável. Para desabilitar esse recurso é necessário configurar o parâmetro de contexto **org.apache.myfaces.COMPRESS_STATE_IN_SESSION** com o valor igual a **false** (veja a **Listagem 9**).

Assim, a aplicação terá um ganho no tempo de resposta, porém tome bastante cuidado, pois essa alteração possui um efeito colateral no consumo de memória, principalmente se sua aplicação armazena grandes quantidades de informação na sessão. Portanto, só faça isso se você tiver certeza que o servidor em que a sua aplicação está tem memória suficiente para comportar esse aumento.

Listagem 9. Configuração do web.xml para não realizar a compressão do estado na sessão.

```
<context-param>
<param-name>org.apache.myfaces.COMPRESS_STATE_IN_SESSION
</param-name>
<param-value>false</param-value>
</context-param>
```

Facelets

O Facelets é uma linguagem de descrição de páginas (*PDL — Page Description Language*) criada especificamente para JSF. Ele estabelece uma linguagem de *templates* que suporta a criação da árvore de componentes das telas JSF.

Buscando otimizar ainda mais nossa aplicação, uma opção é tentarmos melhorar o desempenho do componente Facelets, e, para isso, podemos diminuir a quantidade de estados armazenados por ele (no servidor ou no cliente).

A solução, neste caso, é permitir que o Facelets crie as páginas (*views*) antes do processamento da requisição HTTP. Para que isso ocorra, utilize o parâmetro de contexto **facelets.BUILD_BEFORE_RESTORE** no *web.xml* com o valor igual a **true**, conforme a **Listagem 10**. Ao fazer isso, como efeito colateral, provavelmente ocorrerá um *overhead* inicial, porém ao longo do tempo seu sistema terá um ganho significativo no desempenho.

Listagem 10. Configuração do web.xml para que o facelets crie as páginas previamente.

```
<context-param>
<param-name>facelets.BUILD_BEFORE_RESTORE</param-name>
<param-value>true</param-value>
</context-param>
```

Listagem 11. Configuração do web.xml para desabilitar o refresh do Facelets.

```
<context-param>
<param-name>facelets.REFRESH_PERIOD</param-name>
<param-value>-1</param-value>
</context-param>
```

Listagem 12. Configuração do web.xml para desabilitar o modo DEVELOPMENT.

```
<context-param>
<param-name>facelets.DEVELOPMENT</param-name>
<param-value>false</param-value>
</context-param>
```

Outra característica de Facelets é que ele possui um mecanismo de atualização automática das *views* no qual um processo fica verificando se a página sofreu alguma alteração. Como desvantagem, esse processo pode levar a um consumo excessivo de recursos do servidor. Para evitar esse problema é possível desabilitar esse mecanismo adicionando o parâmetro **facelets.REFRESH_PERIOD** com o valor **-1** (menos um), conforme expõe a **Listagem 11**.

Por fim, certifique-se que o Facelets não seja executado em produção no modo debug, pois nesse modo são armazenados cinco vezes mais informações sobre componentes e *beans* para cada usuário que esteja utilizando a aplicação. Para evitar isso, configure o parâmetro **facelets.DEVELOPMENT** com o valor **false**, conforme demonstra a **Listagem 12**.

Stateless com JSF

O conceito de *stateless* (sem estado), a princípio, pode parecer difícil de utilizar, visto que a maioria das aplicações têm, pelo menos, algum tipo de estado armazenado, principalmente quando faz uso do JSF, que, como vimos, armazena várias informações ao longo de cada requisição.

Em uma aplicação *stateless*, a camada web não mantém estado, por mais tempo que o processamento de uma requisição possa durar. Isso significa a não utilização das sessões web clássicas. Assim, todas as informações de estado, necessárias para processar uma requisição, precisam estar na própria requisição. As informações adicionais, por sua vez, podem ser recuperadas de um armazenamento secundário, com base em um identificador, enviado na requisição.

Quando optado pelo armazenamento secundário, visto que essa operação precisa ser realizada a cada requisição, é importante que o acesso ao armazenamento seja rápido o suficiente para não prejudicar a performance. Por isso, o uso de bases de dados NoSQL ou simplesmente caches em memória são preferidos, ao invés dos tradicionais bancos de dados relacionais.



O conceito de *stateless* se tornou muito popular em aplicações web e sempre foi uma ótima opção quando se deseja melhorar a performance. Sendo assim, utilizar páginas *stateless* em aplicações JSF parece uma alternativa muito boa. Contudo, por padrão, o JSF é um framework *stateful* (guarda estado) e isso sempre foi um ponto de críticas a essa tecnologia. Pensando nisso, a versão 2.2 introduziu um novo recurso, chamado “*stateless views*”. Ao adotá-lo, o estado dos componentes JSF não será salvo e nem restaurado a cada requisição HTTP, trazendo ganhos significativos à performance do sistema. Vale reforçar que esse recurso só deve ser utilizado caso sua aplicação não necessite que o estado dos componentes seja armazenado.

Por que é um estado em JSF?

Primeiramente, é importante não confundir e saber distinguir os dois tipos de estado de uma aplicação JSF: o estado da aplicação e o estado da página. Toda aplicação, provavelmente, precisa manter algum tipo de estado, por exemplo os dados do usuário que está acessando o sistema ou quais campos o usuário já preencheu do formulário.

Da perspectiva do desenvolvedor, o estado da aplicação é armazenado em atributos de *managed beans*, que, por sua vez, estão associados a um tipo de escopo (como *request* ou *session scope*). De modo simples, o escopo define o ciclo de vida de um *managed bean*, interferindo diretamente na quantidade de recursos necessários no servidor. O estado da página, por sua vez, é representado pela sua árvore de componentes, onde cada componente possui atributos e valores que estão associados a uma classe do tipo *entity bean*.

Por que o JSF precisa manter o estado da página?

Como o HTTP é um protocolo *stateless* e muitas vezes é desejável manter o estado da árvore de componentes do JSF para que ele consiga identificar o estado atual de uma página, faz-se necessário o armazenamento dessa informação. Assim, caso o usuário clique no botão voltar do navegador e, em seguida, reenvie um formulário que foi criado anteriormente, ao manter o estado da página é possível identificar que essa operação já foi realizada e evitar uma provável duplicação de dados.

Por que devemos optar por uma aplicação *stateless*?

Salvar o estado em uma aplicação web normalmente significa que o estado do usuário necessita estar em algum lugar da memória do lado servidor, por exemplo na sessão, criando uma dependência entre um cliente específico e a instância da aplicação que contém o estado. Note que essa dependência não é um problema em um ambiente de servidor único, supondo que a demanda de requisições é tranquilamente suportada.

Os problemas ficam evidentes quando o aplicativo é implantado em várias máquinas, por exemplo em um ambiente em cluster (vide **BOX 2**). Nesses casos, devem ser tomadas medidas para que o estado dos usuários esteja disponível em cada nó. Por isso, replicação de sessão e sessões persistentes são conceitos comuns a qualquer sistema com vários nós (instâncias). A desvantagem

disso é que a configuração e manutenção dos servidores em *cluster* se tornarão mais difíceis, e a replicação da sessão pode levar a um enorme desperdício de memória, já que o estado também é replicado para nós que provavelmente nunca precisarão dessa informação.

BOX 2. Cluster

Cluster (ou clustering) é, em poucas palavras, o nome dado a um sistema que relaciona dois ou mais servidores para que esses trabalhem em conjunto com o intuito de processar uma ou mais tarefas. Essas máquinas dividem entre si as tarefas e executam esse trabalho de maneira simultânea.

Em contraste, ao desenvolver uma aplicação *stateless* os nós do cluster podem ser implantados sem a necessidade de se conhecerem, pois não é preciso compartilhar qualquer estado entre os nós. Assim, é possível que os pedidos subsequentes de um cliente sejam atendidos a partir de diferentes nós, uma vez que cada pedido contém toda a informação solicitada para processar a requisição. Com base nisso, arquiteturas em nuvem elásticas permitem que aplicações *stateless* lidem com grandes quantidades de requisições, uma vez que os novos nós podem facilmente ser gerados sob demanda.

Como citado anteriormente, o JSF armazena o estado das páginas, por padrão, na sessão do usuário. Isso significa que certa quantidade de memória do servidor será utilizada para essa finalidade.

Por fim, o “maior prejuízo” de uma aplicação que não é *stateless* é que não teremos tanta facilidade para explorar o melhor da computação elástica (*autoscaling*) e todos os benefícios que ela traz com relação ao ganho de performance e disponibilidade.

Stateless em JSF

Aplicar o conceito de *stateless* em JSF significa que algumas ou todas as páginas da aplicação não salvarão o estado. Contudo, é válido lembrar que alguns componentes JSF foram projetados para serem *stateful*. Desse modo, há uma boa chance de alguns componentes não funcionarem conforme o esperado. Antes de definir uma página como *stateless*, portanto, analise com muito cuidado os possíveis efeitos colaterais.

Quando *stateless* em JSF é recomendado?

Respondendo de forma breve: sempre que possível, pois aplicações *stateless* geralmente são fáceis de escalar. Portanto, projetar uma aplicação JSF para executar no modo *stateless* pode ser a melhor escolha. Em contrapartida, com JSF isso não é fácil, mas pode ser facilmente justificado para resolver problemas de escalabilidade e desempenho, tornando válido o esforço para alterar sua aplicação para essa opção.

Definindo uma página em JSF como *stateless*

Definir uma página como *stateless* é muito simples: basta adicionar o atributo **transient** na tag **<f:view>** e especificar seu valor como **true**. Esse comportamento impede que o estado da página seja salvo. Um exemplo é apresentado na **Listagem 13**.

Listagem 13. Exemplo de código XHTML que utiliza a tag <f:view> com o atributo transient.

```
<f:view transient="true">
<t:document>
<t:documentHead>
...
<title>example</title>
...
</t:documentHead>
<t:documentBody>
...
</t:documentBody>
</t:document>
</f:view>
```

Atualmente, infelizmente, não há nenhuma possibilidade de ativar essa funcionalidade para toda a aplicação. O conceito de *stateless views* é indiferente aos métodos de salvar estado (*cliente-side* ou *server-side*) vistos anteriormente, pois todas as páginas que não são definidas como *stateless*, por padrão, não irão utilizar esse novo recurso.

Menos requisições, sistemas mais rápidos

Uma abordagem comum para otimização do desempenho de aplicações web, independente da tecnologia utilizada, é reduzir o número de requisições HTTP ao servidor, o que pode ser feito através da combinação de recursos do mesmo tipo em um único arquivo.



Essa ação pode ser realizada manualmente, porém não é desejável colocar todo o código JavaScript, por exemplo, em um único arquivo, pois isso quebra o conceito de modularidade e torna muito difícil manter uma base de código sustentável.

A combinação de recursos em JSF pode ser feita substituindo a solução padrão do componente de renderização de recursos (**ResourceHandler**) por uma implementação personalizada, que combina todos os scripts (JavaScript) e folhas de estilo (CSS), gerando novos recursos (arquivos) “agrupados”.

Felizmente, existem várias implementações prontas para as principais bibliotecas de componentes do JSF, como acontece com o ICEfaces, RichFaces, PrimeFaces e OmniFaces. Neste artigo, no entanto, vamos utilizar e descrever a implementação oferecida pelo OmniFaces, chamada **CombinedResourceHandler**.

O OmniFaces vem com uma implementação de **ResourceHandler** muito simples de utilizar. Desse modo, tudo o que é necessário fazer é configurá-lo no *faces-config.xml*, conforme a **Listagem 14**. Para fazer o download do JAR do OmniFaces, acesse o endereço indicado na seção **Links**.

Listagem 14. Configuração do arquivo *faces-config.xml* para utilizar a classe **CombinedResourceHandler**.

```
<application>
<resource-handler>
org.omnifaces.resourcehandler.CombinedResourceHandler
</resource-handler>
</application>
```

Boas práticas para obter melhor desempenho com JSF

Existem algumas boas práticas que devemos adotar, sempre que possível, para evitar problemas de performance. A primeira delas é evitar lógica complexa ou pesada nos métodos *getters*. Eles são chamados várias vezes durante o ciclo de vida de uma requisição e só devem retornar algo previamente definido.

Outro ponto importante é utilizar o atributo **ajaxSingle="true"** nos componentes JSF, visando reduzir o tráfego de informações entre o cliente e o servidor. Isso faz com que somente os dados do componente sejam enviados, isto é, evita que todo o formulário seja transmitido desnecessariamente. Só não utilize essa opção se você realmente quiser enviar toda a informação da página de volta para o servidor.

Considere, também, o uso do atributo **immediate="true"**. Para os casos em que você não precisa de validação, isso faz com que uma etapa do ciclo de vida da requisição não seja executada, reduzindo o tempo de renderização.

Quando tiver que escolher entre um componente e outro, não faça a opção pelo mais “rico” só porque você considera legal. Dê preferência ao componente mais simples, pois provavelmente será mais eficiente.

Ademais, evite exibir grandes tabelas para o usuário. Em casos assim, opte pela paginação. E principalmente, nunca retorne todas as páginas da tabela em uma única requisição, isto é, realize a paginação do lado do servidor, fazendo com que somente as

informações da página que o usuário requisitou sejam transmitidas pela rede.

Por fim, não compleique desnecessariamente as *expression languages* (ELs), removendo, sempre que possível, qualquer tipo de lógica complexa. O ideal é que tais comportamentos sejam implementados no código dos *managed beans*. E também, deixe o código o mais simples possível nos getters e setters, pois são esses os métodos acessados ao se processar uma EL.

Neste artigo foram apresentadas algumas medidas que visam melhorar o desempenho geral de uma aplicação JSF. No entanto, é válido ressaltar que não necessariamente devemos aplicar todas elas, pois isso não garante que todos os problemas de desempenho serão resolvidos. A depender do projeto, pode até impactar de forma negativa, o que sinaliza que para realizar qualquer mudança é importante avaliar todas as possibilidades. Lembre-se, ainda, que as aplicações também dependem de inúmeros outros fatores, como banco de dados, web services, rede, entre outros.

Não seria surpresa se em futuras versões do JSF novas abordagens que vão em direção ao *stateless mode* (sem estado) sejam adotadas. Com o modo *stateless* não haverá a necessidade de criar uma árvore de componentes a cada requisição (existirá um cache de cada *view*), o que diminuirá o uso de memória e CPU e trará um ganho gigantesco na quantidade de requisições por segundo que uma aplicação conseguirá atender.

Por fim, saiba que além do conteúdo apresentado, há uma série de outros tópicos relacionados a desempenho em sistemas web muito interessantes e que valem a pena o leitor pesquisar e se aprofundar. Portanto, não pare os estudos por aqui. Busque por esse conteúdo em outras edições da Java Magazine e em outros materiais de referência da área.

Autor



Ronaldo Ronie Nascimento

ronaldo.ronie@gmail.com - <https://br.linkedin.com/in/ronaldorоние>

Engenheiro de Software Sênior/Arquiteto na empresa Semsedia, é bacharel em Engenharia de Computação pela PUC-Campinas e tem pós-graduação em Engenharia de Software Orientada a Serviços — SOA. Trabalha com as tecnologias do mundo Java e Web desde 2007, com experiência em desenvolvimento de sistemas distribuídos de alta performance e disponibilidade.



Links:

Endereço para download do JBoss-Serialization.

<http://serialization.jboss.org/downloads>

Comparação entre Apache MyFaces e Mojarra.

<http://blog.ocio.de/2013/04/08/jsf-comparison-myfaces-vs-mojarra/>

Endereço para download do componente OmniFaces.

<http://central.maven.org/maven2/org/omnifaces/omnifaces/2.2/omnifaces-2.2.jar>

Gerenciando e testando seu banco de dados com Liquibase e JUnit

Aprenda neste artigo a utilizar o Liquibase e saiba como gerenciar os scripts de banco, assim como realizar testes integrados com a base de dados

Para quem já programa há algum tempo sabe que gerenciar a estrutura e os dados do banco de dados de uma aplicação é uma das partes mais complicadas e sujeitas a erros em um processo de desenvolvimento. A fim de mitigar esse problema, foram criadas diversas soluções com o intuito de deixar essa etapa mais simples, cada uma com suas vantagens e desvantagens.

Uma dessas opções é compartilhar a responsabilidade da gestão do banco com outros profissionais, como os DBAs. Nesses casos, quando o desenvolvedor percebe que precisará criar uma nova coluna em uma tabela, ele prepara o comando SQL e o envia para o DBA, que irá validar e, às vezes, até otimizar o comando, fazendo a gestão de todos os comandos que deverão ser executados no momento da entrega de uma nova release do sistema. Uma das vantagens dessa maneira de trabalhar é que os desenvolvedores ficam mais focados no código e os DBAs podem gerenciar todo o processo de mudança na base de dados, mas ao mesmo tempo exige um maior alinhamento entre as equipes na entrega de uma release, pois também serão os DBAs que executarão todos os scripts necessários antes que os desenvolvedores subam a nova versão para produção, que espera encontrar aquela nova coluna na base de dados.

Como sincronizar o trabalho de diferentes equipes pode ser muito custoso e envolver muitas atividades manuais, alguns desenvolvedores tentaram automatizar a atualização das estruturas das tabelas do banco de dados criando e melhorando frameworks já existentes, como o Hibernate. Além de controlar as consultas e

Fique por dentro

Este artigo analisa e discute as vantagens de utilizar uma das mais fáceis bibliotecas de gerenciamento de mudanças em bancos de dados relacionais encontradas atualmente, o Liquibase. Com exemplos práticos, você aprenderá a configurar um projeto Java tendo o Liquibase como gerenciador de scripts de banco, além de criar testes integrados com Spring e JUnit que acessam uma base de testes, também gerenciada por essa solução.

inserções nas tabelas que são realizadas pela aplicação, o Hibernate também pode atualizar a estrutura dessas tabelas sem a necessidade de você criar e executar scripts SQL. Isso é feito colocando o valor `update` na propriedade `hibernate.hbm2ddl.auto`, o que faz o framework alterar as estruturas da base de acordo com as classes de entidade do projeto no momento em que a aplicação estiver sendo iniciada. Caso uma nova tabela, ou coluna, precise ser criada, o Hibernate executa os comandos necessários para isso. O problema dessa funcionalidade é que se a aplicação começar a ficar lenta e um DBA precisar otimizar algo, o Hibernate pode não validar essas otimizações e, às vezes, até desfazê-las, dependendo de como estão suas configurações.

Ao final, essas opções que discutimos ficam, de certa forma, nos extremos de como é possível gerenciar um banco de dados: na primeira opção, a gestão é completamente manual; já na segunda, totalmente automatizada, o que dificulta a implantação de otimizações que a ferramenta talvez não seja capaz de realizar. Diante disso, será que existe um jeito de unir as vantagens das duas em uma só solução?

Encontrando o meio-termo: o Liquibase

Vamos imaginar uma maneira um pouco diferente para gerenciar uma base de dados. Que tal os desenvolvedores continuarem criando os scripts SQL do banco, só que de uma forma mais padronizada e estruturada, permitindo que alguma ferramenta os execute automaticamente no momento da entrega de uma release? E, além disso, mantermos os DBAs validando e otimizando esses scripts? Essa opção agora é uma realidade, pois já existe um framework que faz esse meio-termo entre a automatização e as atividades manuais: o Liquibase.

De forma simples, podemos definir o Liquibase como uma biblioteca de código aberto que tem como objetivo gerenciar as alterações realizadas no banco de dados. Criado em 2006, ele pode se integrar aos principais gerenciadores de projetos, como o Maven, Gradle e Ant, facilitando bastante sua utilização. Essa biblioteca possui diversos comandos que permitem realizar a gestão do banco, como fazer um *rollback* até determinada alteração, atualizar a base executando todos os scripts que ainda faltam e até criar um arquivo com toda a estrutura atual do banco de dados. O Liquibase consegue, ainda, simultaneamente, gerenciar outra base de dados, só que essa sendo exclusiva para testes. Assim, a criação e execução de testes integrados com o banco se torna uma tarefa menos complicada.

Um dos diferenciais dessa solução é que ela suporta arquivos de script nos formatos XML, YAML, JSON ou SQL, sempre utilizando o conceito de *changeSets*, que nada mais é do que um conjunto de comandos que o Liquibase deve executar na base de dados como, por exemplo, um comando de criação de tabela seguido por outro de inserção de um registro. Nesse contexto, um ou mais conjuntos de comandos, ou *changeSets*, podem ser colocados em um mesmo arquivo, chamado de *changeLog*. Para isso, no entanto, cada *changeSet* precisa ter um identificador, preferencialmente único, para que o Liquibase possa identificar quais conjuntos de comandos já foram ou precisam ser executados no banco.

Quando o Liquibase é executado, ele verifica todos os *changeSets* existentes nos arquivos de *changeLog* e começa a aplicá-los na base de dados, pela ordem do nome dos arquivos. A gestão de quais *changeSets* foram executados acontece na tabela *databasechangelog*, criada pelo próprio Liquibase na base de dados, permitindo assim que ele não aplique o mesmo *changeSet* duas vezes na base. Outra característica importante é que o framework não permite a alteração ou inclusão de novos comandos em um *changeSet* que já foi executado no banco, pois o Liquibase não o executará novamente, lançando assim um erro para avisar sobre o problema. Caso um *changeSet* tenha sido criado de forma incorreta, é necessário criar um novo corrigindo o problema ou realizar um *rollback* dessa alteração para então modificar o *changeSet* em questão.

O Liquibase também permite, a qualquer momento, que o banco de dados seja “marcado” com alguma *tag*, da mesma forma que acontece quando se deseja marcar um código-fonte que utiliza um sistema de versionamento como o SVN.

Assim, as boas práticas de controle de versão do código fonte também podem ser adotadas para gerenciar o banco de dados.

Com a possibilidade de marcação de um estado da base de dados, operações como o *rollback* para um determinado estado anterior ficam muito mais fáceis de serem efetuadas.

Outra característica do framework é a possibilidade de utilizá-lo em um banco que estava sendo gerenciado por outra ferramenta, como o Hibernate, ou em alguma base em que a gestão da execução dos scripts já foi perdida há algum tempo. Para isso, ele possui o comando *generateChangeLog*, que gera um arquivo com os scripts de criação das estruturas existentes na base atual. Esse comando, no entanto, possui algumas limitações, pois não consegue recuperar todos os tipos de estruturas do banco, como *procedures*, *functions* e *triggers*. Finalmente, depois que o arquivo é gerado, é necessário apenas executar o comando *changeLogSync*, para marcar os *changeSets* criados como já executados. Assim o Liquibase não os executará novamente.

Ademais, o Liquibase disponibiliza o comando *diff*, que gera um relatório comparando as tabelas e outros componentes de duas bases de dados, mostrando o que há de diferente entre elas. Outra funcionalidade interessante é a opção de criar, automaticamente, a documentação da base de dados, no mesmo formato dos javadocs, com as informações dos *changeSets* executados, os que estão pendentes, as últimas alterações aplicadas, a estrutura das tabelas, entre outras.

Dito isso, para analisar o Liquibase na prática, utilizaremos um projeto simples para demonstrar como modificar uma aplicação standalone gerenciada pelo Maven e que utiliza o Spring e o Hibernate. Quando executada, essa aplicação expõe no console os produtos cadastrados em uma tabela da base de dados.

Utilizando o Liquibase em um projeto

O projeto em que vamos empregar o Liquibase acessa uma base de dados MySQL para consultar os produtos cadastrados em uma tabela de nome *Product*. No exemplo, utilizaremos também o Hibernate, mas não o deixaremos configurado para atualizar automaticamente as tabelas, justamente para evitar que ele execute modificações que não estejam listadas nos scripts do Liquibase. Dessa forma, precisaremos ter o script de criação dessa tabela em um arquivo.

A **Figura 1** mostra a estrutura do projeto Maven com a estrutura de pastas sugerida para organizar os scripts SQL. O projeto completo está disponível para download na página desta edição.

Configurando o Liquibase

O primeiro passo para poder utilizar o Liquibase é declarar o plugin *liquibase-maven-plugin* no arquivo de configurações do Maven. A **Listagem 1** apresenta o trecho do *pom.xml* com essa definição.

Nele é possível especificar as configurações e bibliotecas do banco de dados que o Liquibase irá manipular, assim como em

Gerenciando e testando seu banco de dados com Liquibase e JUnit

quais fases do processo de execução do Maven ele deve rodar. Entre as linhas 14 e 27, definimos que no momento em que os testes integrados estiverem sendo executados, o Liquibase deverá aplicar os comandos SQL no banco de dados de teste, e entre as linhas 28 e 41 especificamos que durante a fase de *install* do Maven, ele deve executar os scripts SQL no banco principal.

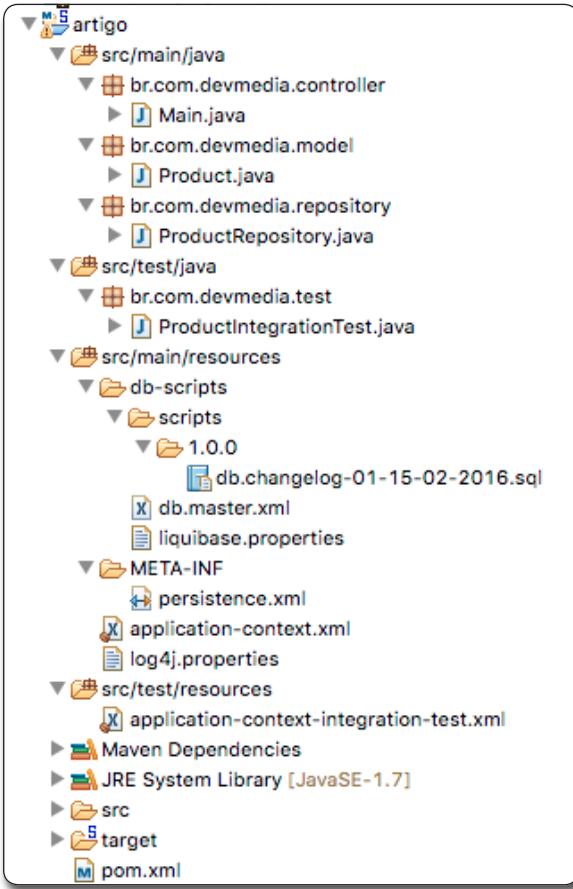


Figura 1. Projeto Maven de exemplo

Note que algumas propriedades do plugin do Liquibase estão recebendo valores que ainda não foram configurados. É o caso das opções *username* e *password*. Esses valores são de propriedades específicas do Liquibase que serão informadas na seção de *profiles* no arquivo de configuração do Maven. O recurso de *profiles* do Maven é extremamente útil quando a aplicação precisa ser executada em ambientes com características diferentes.

O segundo passo, então, é justamente especificar os valores das propriedades do Liquibase nos *profiles* do Maven. A **Listagem 2** exibe esse trecho do *pom.xml*. Nele estão definidas as propriedades necessárias a cada um dos ambientes em que a aplicação será executada. Deste modo, dependendo do valor do argumento *profile*, que é passado no momento da execução do comando *install* do Maven, o plugin do Liquibase utilizará as informações do banco de dados que foram configuradas para aquele perfil de ambiente.

Continuando nossa análise da seção *profiles*, a propriedade *liquibase.verbose* diz ao plugin se ele deve ou não mostrar mais

informações quando estiver executando algum script no banco. Já a propriedade *liquibase.driver*, nas linhas 10 e 29, especifica qual driver de banco de dados será utilizado (no caso, o driver do MySQL). As propriedades que terminam com “-test” são utilizadas quando os testes integrados são executados pelo Maven. Por default, deixamos desabilitada a execução de testes integrados definindo a propriedade *skip.integration.tests* com o valor *true*, o que diz ao Maven que ele deve “pular” os testes integrados.

Listagem 1. Declarando o plugin do Liquibase no pom.xml.

```
01 <plugin>
02   <groupId>org.liquibase</groupId>
03   <artifactId>liquibase-maven-plugin</artifactId>
04   <version>3.1.0</version>
05 
06   <configuration>
07     <changeLogFile>src/main/resources/db-scripts/db.changelog.xml
08     </changeLogFile>
09     <verbose>${liquibase.verbose}</verbose>
10     <dropFirst>${liquibase.dropFirst}</dropFirst>
11     <promptOnNonLocalDatabase>
12       ${liquibase.promptOnNonLocalDatabase}
13     </promptOnNonLocalDatabase>
14     <driver>${liquibase.driver}</driver>
15   </configuration>
16   <executions>
17     <execution>
18       <id>liquibase-before-integration-test</id>
19       <phase>pre-integration-test</phase>
20       <goals>
21         <goal>update</goal>
22       </goals>
23       <configuration>
24         <skip>${skip.integration.tests}</skip>
25         <url>${liquibase.url-test}</url>
26         <username>${liquibase.username-test}</username>
27         <password>${liquibase.password-test}</password>
28         <contexts>${liquibase.contexts}</contexts>
29       </configuration>
30     </execution>
31     <execution>
32       <id>liquibase-install</id>
33       <phase>install</phase>
34       <goals>
35         <goal>update</goal>
36       </goals>
37       <configuration>
38         <skip>${liquibase.skip}</skip>
39         <url>${liquibase.url}</url>
40         <username>${liquibase.username}</username>
41         <password>${liquibase.password}</password>
42         <contexts>${liquibase.contexts}</contexts>
43       </configuration>
44     </execution>
45   </executions>
46   <dependencies>
47     <dependency>
48       <groupId>mysql</groupId>
49       <artifactId>mysql-connector-java</artifactId>
50       <version>5.0.4</version>
51     </dependency>
52   </dependencies>
53 </plugin>
```

Ainda na **Listagem 2**, a linha 5 marca o profile *default* como o padrão. Assim, se nenhum profile for informado na execução de algum comando Maven, o *default* será utilizado. Por segurança, esse profile tem as configurações do plugin *liquibase-maven-plugin* ajustadas para rodar no ambiente local, evitando uma execução não desejada em outro ambiente. Já o profile *live* tem as informações do banco de dados do ambiente de produção.

Listagem 2. Definindo os profiles no pom.xml.

```

01 <profiles>
02   <profile>
03     <id>default</id>
04     <activation>
05       <activeByDefault>true</activeByDefault>
06     </activation>
07     <properties>
08       <argLine />
09       <liquibase.url>jdbc:mysql://127.0.0.1/artigo</liquibase.url>
10      <liquibase.driver>com.mysql.jdbc.Driver</liquibase.driver>
11      <liquibase.username>root</liquibase.username>
12      <liquibase.password></liquibase.password>
13      <liquibase.contexts>local</liquibase.contexts>
14      <liquibase.verbose>true</liquibase.verbose>
15      <liquibase.skip>false</liquibase.skip>
16      <liquibase.url-test>jdbc:mysql://127.0.0.1/artigotest</liquibase.url-test>
17      <liquibase.username-test>root</liquibase.username-test>
18      <liquibase.password-test></liquibase.password-test>
19      <liquibase.verbose-test>false</liquibase.verbose-test>
20      <skip.integration.tests>true</skip.integration.tests>
21    </properties>
22  </profile>
23
24  <profile>
25    <id>live</id>
26    <properties>
27      <argLine />
28      <liquibase.url>jdbc:mysql://mysql.devmedia.com.br/artigo</liquibase.url>
29      <liquibase.driver>com.mysql.jdbc.Driver</liquibase.driver>
30      <liquibase.username>root</liquibase.username>
31      <liquibase.password></liquibase.password>
32      <liquibase.contexts>live</liquibase.contexts>
33      <liquibase.verbose>false</liquibase.verbose>
34      <liquibase.skip>false</liquibase.skip>
35      <liquibase.url-test>jdbc:mysql://mysql.devmedia.com.br/artigotest</liquibase.url-test>
36      <liquibase.username-test>root</liquibase.username-test>
37      <liquibase.password-test></liquibase.password-test>
38      <liquibase.verbose-test>true</liquibase.verbose-test>
39    </properties>
40  </profile>
41 </profiles>
```

Configurados os valores das propriedades do Liquibase em cada *profile*, é necessário criar o seu arquivo de configuração, o *liquibase.properties*.

A **Listagem 3** mostra o conteúdo desse arquivo. Nele devem ser declaradas as informações do banco de dados que o Liquibase irá manipular. Como podemos notar, algumas dessas informações recebem os nomes de propriedades definidas no *pom.xml*. Dessa forma, o Maven as substituirá pelo valor correto de acordo com o *profile* selecionado durante a execução do comando *install*.

Além de especificar as informações do banco de dados no arquivo do Liquibase, é necessário também passá-las para o

Hibernate. Uma forma de fazer isso é configurando unidades de persistência no arquivo *persistence.xml*. A **Listagem 4** mostra o conteúdo desse arquivo.

Listagem 3. Conteúdo do arquivo liquibase.properties.

```
#liquibase.properties
changeLogFile = db-scripts/db.changelog.xml
```

```
url = ${liquibase.url}
driver = ${liquibase.driver}
username = ${liquibase.username}
password = ${liquibase.password}
contexts = ${liquibase.contexts}
verbose = ${liquibase.verbose}
dropFirst = ${liquibase.dropFirst}
```

Listagem 4. Conteúdo do arquivo persistence.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="mySqlDefault"
    transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="hibernate.archive.autodetection" value="class, hbm"/>
      <property name="hibernate.hbm2ddl.auto" value="validate"/>
      <property name="hibernate.connection.driver_class"
        value="com.mysql.jdbc.Driver"/>
      <property name="hibernate.connection.url" value="${liquibase.url}"/>
      <property name="hibernate.connection.username"
        value="${liquibase.username}"/>
      <property name="hibernate.connection.password"
        value="${liquibase.password}"/>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL5InnoDBDBDialect"/>
    </properties>
  </persistence-unit>
  <persistence-unit name="mySqlTest" transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="hibernate.archive.autodetection" value="class, hbm"/>
      <property name="hibernate.hbm2ddl.auto" value="validate"/>
      <property name="hibernate.connection.driver_class"
        value="com.mysql.jdbc.Driver"/>
      <property name="hibernate.connection.url" value="${liquibase.url-test}"/>
      <property name="hibernate.connection.username"
        value="${liquibase.username-test}"/>
      <property name="hibernate.connection.password"
        value="${liquibase.password-test}"/>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL5InnoDBDBDialect"/>
    </properties>
  </persistence-unit>
</persistence>
```

Observe que para esse exemplo foram definidas duas unidades de persistência, uma para cada ambiente: *mySqlDefault* e *mySqlTest*. Em cada uma delas as propriedades do Hibernate recebem como

Gerenciando e testando seu banco de dados com Liquibase e JUnit

valores os nomes das propriedades do Liquibase declaradas no trecho de *profiles* do *pom.xml*, pois, da mesma forma que no arquivo *liquibase.properties*, esses valores serão trocados pelo Maven enquanto ele estiver sendo executado.

Para que a aplicação utilize a unidade de persistência correta, é necessário especificar qual delas deve ser adotada no arquivo de configuração do Spring, o *application-context.xml*. A **Listagem 5** mostra o conteúdo desse arquivo. Aqui, saiba que é a propriedade **persistenceUnitName**, do objeto **entityManagerFactory**, que deve receber o nome da unidade de persistência padrão, definida no *persistence.xml*.

Listagem 5. Conteúdo do arquivo *application-context.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jpa="http://www.springframework.org/schema/data/jpa"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="mySqlDefault"/>
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor
                .HibernateJpaVendorAdapter"></bean>
        </property>
    </bean>

    <bean id="transactionManager" class="org.springframework.orm.jpa
        .JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
    </bean>

    <tx:annotation-driven transaction-manager="transactionManager"/>

    <jpa:repositories base-package="br.com.devmedia.repository"
        query-lookup-strategy="create-if-not-found" />
    <context:component-scan base-package="br.com.devmedia" />
</beans>
```

Para finalizar as configurações do Liquibase, é necessário criar um arquivo chamado *db.changelog.xml*, como mostra a **Listagem 6**. Esse arquivo serve para definir o tipo do banco de dados que será manipulado e o nome da pasta que terá os arquivos de *changeLog*. No nosso caso, o tipo de banco é o MySQL e os arquivos de *changeLog* serão agrupados em pastas de acordo com a versão da aplicação que for sendo lançada, o que facilita saber o que foi executado no banco em cada release.

Agora, dentro da pasta configurada na **Listagem 6**, podemos criar o arquivo com os comandos SQL propriamente ditos. Uma sugestão é adotar uma nomenclatura que contenha o número do *changeLog* (no exemplo, o número é 01) e a data em que ele foi criado (15-02-2016). Isso porque o Liquibase executará todos os arquivos contidos nessa pasta seguindo a ordem alfabética do nome.

Como podemos notar, na **Listagem 7**, que apresenta o código do nosso *changelog*, os comandos SQL estão agrupados em *changeSets*. Por padrão, a primeira linha do arquivo deve conter um comentário com o texto *liquibase formatted sql*. Já o início de um *changeSet* deve possuir uma linha de comentário com a palavra *changeSet*. Para especificar o fim do conjunto de comandos, o Liquibase entende o ";" da última instrução SQL do *changeSet* ou uma linha de comentário com a palavra *rollback* e um comando SQL que reverte a ação executada pelo comando do *changeSet*.

Além disso, cada *changeSet* deve ter um identificador, normalmente especificado com o nome da pessoa que criou o comando e um número sequencial.

Listagem 6. Conteúdo do arquivo *db.changelog.xml*.

```
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
    http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-2.0.xsd
    http://www.liquibase.org/xml/ns/dbchangelog-ext
    http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd">

    <preConditions>
        <dbms type="mysql" />
    </preConditions>
    <includeAll path="src/main/resources/db-scripts/scripts/1.0.0/" />
</databaseChangeLog>
```

Listagem 7. Conteúdo do arquivo *db.changelog-01-15-02-2016.sql*.

```
1 --liquibase formatted sql
2
3 --changeset rafael:1
4
5 DROP TABLE IF EXISTS `product`;
6 CREATE TABLE `product` (
7     `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
8     `name` varchar(100) NOT NULL,
9     `description` varchar(200) NOT NULL,
10    `price` double(10,2) NOT NULL,
11    PRIMARY KEY (`id`)
12 ) ENGINE=InnoDB AUTO_INCREMENT=100 DEFAULT CHARSET=utf8;
13
14 --rollback drop table `product`;
15
16 --changeset rafael:2 context:local
17 INSERT INTO `product` VALUES (1, "TV", "Samsung 42\"", 1200);
18
19 -- rollback delete from `product` where id = 1;
```

É possível também informar outros argumentos na linha de início de um *changeSet* como, por exemplo, o argumento *context*, na linha 16. Ele tem a mesma utilidade do *profile* no Maven, permitindo que determinado conjunto de comandos seja executado apenas em determinados ambientes, ou contextos. No nosso exemplo, o *changeSet* “2”, na linha 16, será executado somente no banco de dados local e caso seja solicitado um rollback desse código, o comando da linha 19 será aplicado.

Criando as classes do projeto

Configurado o projeto respeitando os passos anteriores, a primeira classe a ser criada é a classe de modelo, que representa a entidade **Product** no banco de dados. Como podemos notar na **Listagem 8**, essa classe possui a anotação `@Entity` para especificar ao Hibernate que ela é uma entidade a ser armazenada em uma tabela na base de dados.

Listagem 8. Código da classe Product.

```
@Entity
@SuppressWarnings("serial")
public class Product implements Serializable {

    private Long id;
    private String name;
    private String description;
    private Double price;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public Double getPrice() {
        return price;
    }
    public void setPrice(Double price) {
        this.price = price;
    }
    @Override
    public String toString() {
        return "Product [id=" + id + ", name=" + name + ", description=" +
               + description + ", price=" + price + "]";
    }
}
```

Em seguida, precisamos definir as operações que serão realizadas com a entidade **Product** no banco de dados. Para isso, criaremos a interface **ProductRepository**, que estende a interface **JpaRepository** e, por isso, recebe todos os métodos de inserção, atualização, remoção e consulta necessários para um projeto padrão. Como diferencial, acrescentaremos um método para uma consulta específica, que utilizará o nome do produto. Deste modo, adicionamos a anotação `@Query`, acima da declaração do método, com o comando que deve ser executado. A **Listagem 9** mostra o código da interface **ProductRepository**.

Listagem 9. Código da interface ProductRepository.

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    @Query("select p from Product p where p.name like ?")
    List<Product> findByName(String name);
}
```

Feito isso, nos resta criar a classe principal do projeto, que chama a interface que acessa o repositório para consultar os produtos cadastrados na base pesquisando pelo nome. A **Listagem 10** mostra o código dessa classe. Como podemos notar, o primeiro método é o `getProductsByName()`, que chama o método `findByName()` da interface **ProductRepository** e recupera uma lista de produtos que contêm o texto informado como parâmetro. Em seguida, o principal método dessa classe, o `main()`, instancia o application context do Spring, obtém uma instância da própria classe **Main** e então chama o método `getProductsByName()` com o parâmetro “TV”, o que retornará para o console os produtos que tiverem a palavra “TV” no nome.

Listagem 10. Código da classe Main.

```
@Component
public class Main {

    @Autowired(required = true)
    private ProductRepository productRepository;

    private void getProductsByName(String name) {
        List<Product> products = productRepository.findByName(name);
        for(Product product: products){
            System.out.println(product);
        }
    }

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext
        ("application-context.xml");
        Main main = context.getBean(Main.class);

        main.getProductsByName("TV");
    }
}
```

Gerenciando e testando seu banco de dados com Liquibase e JUnit

Executando o Liquibase

Com o plugin e os arquivos do Liquibase configurados, basta executar o comando correto do Maven para fazer o Liquibase rodar os scripts SQL no banco de dados adequado. Como configuramos o plugin para ele ser executado na fase de *install* do Maven, o comando a ser informado é:

```
mvn clean install -Dliquibase.dropFirst=true
```

O argumento **liquibase.dropFirst=true** serve para indicar ao Liquibase que ele deve apagar todo o banco e executar todos os scripts SQL novamente. Se esse argumento for igual a **false**, o Liquibase executará apenas os scripts que ainda não foram executados, pois ele controla os *changeSets* que foram executados em uma tabela própria. Outra opção para fazer o Liquibase atualizar o banco é executar o comando *update* do próprio plugin, como demonstrado a seguir:

```
mvn liquibase:update -Dliquibase.dropFirst=true
```

Mais um comando disponível no plugin é o de marcação de estado do banco de dados. Esse resulta na criação de uma *tag* para demarcar a estrutura atual da base. No nosso exemplo, após a execução do segundo *changeSet*, a primeira versão do banco estaria pronta, e assim poderíamos chamá-la de versão “1.0.0”. Diante do exposto, para criar a tag “1.0.0” no banco, poderíamos executar o seguinte comando:

```
mvn liquibase:tag -Dliquibase.tag=1.0.0
```

Neste momento, após o Liquibase atualizar a base, podemos executar a classe **Main** e verificar se os dados inseridos pelo script realmente estão no banco de dados.

Por fim, um dos comandos mais úteis do Liquibase é o *rollback*. Com ele você pode voltar o banco de dados para determinada situação se baseando pelos *changeSets* ou pelo nome de uma *tag*. Por exemplo, se você deseja que a tabela de produtos não tenha o item inserido pelo segundo *changeSet*, basta executar:

```
mvn liquibase:rollback -Dliquibase.rollbackCount=1
```

Nesse comando informamos ao Liquibase que ele deve realizar o *rollback* dos comandos de um *changeSet*, contando a partir do último que foi aplicado ao banco.

Testes integrados com a base de dados

Como sabemos, testes de integração são testes que têm como objetivo testar as diversas partes de um software de forma combinada, ou integrada. Por exemplo, se há a necessidade de testar se os métodos de consulta ao banco de dados estão trazendo corretamente as informações esperadas, criar um teste integrado desse método com o banco é uma boa saída. Antes disso, no entanto, precisamos adicionar o plugin *maven-surefire-plugin* no arquivo de

configurações do Maven para especificar quais classes do projeto são de testes unitários e quais são de testes integrados.

Na **Listagem 11** temos o trecho do *pom.xml* com a declaração do plugin *maven-surefire-plugin*, responsável por executar as classes de teste enquanto o Maven estiver rodando. Nas linhas 22 e 35 são definidos os padrões de nomenclatura das classes que o plugin deve considerar como testes unitários ou como testes integrados. Dessa forma, deixamos claro para o Maven quais classes ele deve chamar durante a execução de cada um desses tipos de teste.

Como o objetivo é permitir que as classes de testes integrados validem se os dados inseridos na base estão corretos, uma boa prática é criar um banco de dados exclusivo para esses testes. Visto que na seção de *profiles* do *pom.xml* já especificamos as informações do banco de teste, devemos apenas utilizá-lo nas classes de teste.

Listagem 11. Declarando o plugin do Surefire no *pom.xml*.

```
01 <plugin>
02   <groupId>org.apache.maven.plugins</groupId>
03   <artifactId>maven-surefire-plugin</artifactId>
04   <version>${maven-surefire-plugin.version}</version>
05   <configuration>
06     <skip>true</skip>
07     <trimStackTrace>false</trimStackTrace>
08     <forkCount>1</forkCount>
09     <argLine>${argLine} -Xmx1024m -XX:MaxPermSize=512m</argLine>
10     <reuseForks>true</reuseForks>
11   </configuration>
12   <executions>
13     <execution>
14       <id>unit-tests</id>
15       <phase>test</phase>
16       <goals>
17         <goal>test</goal>
18       </goals>
19       <configuration>
20         <skip>${skip.unit.tests}</skip>
21         <excludes>
22           <exclude>**/*IntegrationTest*.java</exclude>
23         </excludes>
24       </configuration>
25     </execution>
26     <execution>
27       <id>integration-tests</id>
28       <phase>integration-test</phase>
29       <goals>
30         <goal>test</goal>
31       </goals>
32       <configuration>
33         <skip>${skip.integration.tests}</skip>
34         <includes>
35           <include>**/*IntegrationTest*.java</include>
36         </includes>
37       </configuration>
38     </execution>
39   </executions>
40 </plugin>
```

Portanto, para testar os produtos cadastrados no banco, criaremos a **ProductIntegrationTest**, e nela testaremos se as consultas de produtos por Id e por nome estão funcionando.

A **Listagem 12** mostra o código da classe **ProductIntegrationTest**, que testa o método **findByName()** da interface **ProductRepository** fazendo uma inserção de um registro na tabela e verificando se ele é retornado após a consulta. É importante salientar que após o método de teste ser executado, o registro inserido será removido da tabela, pois ele foi executado em um contexto de transação (um *rollback* será executado em seguida). Deste modo, esse teste sempre vai inserir uma nova linha, que deixará de existir depois que o teste terminar.

Listagem 12. Código da classe ProductIntegrationTest.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:/application-context-integration-test.xml")
public class ProductIntegrationTest {

    @Autowired
    private ProductRepository productRepository;

    @Test
    @Transactional
    public void findByNameTest() {
        Product newProduct = new Product();
        newProduct.setName("Computer");
        newProduct.setDescription("Mac");
        newProduct.setPrice(1880.99);
        productRepository.save(newProduct);

        List<Product> products = productRepository.findByName("Computer");
        Assert.assertTrue(products.size() == 1);
    }

    @Test
    public void findByIdTest() {
        Product product = productRepository.findOne(1l);
        Assert.assertTrue(product != null);
    }
}
```

Já o segundo método de teste verifica se o método **findOne()** da interface **ProductRepository** traz o produto com Id 1 que foi inserido pelo *changeSet* 2 do Liquibase. Repare, no entanto, que esse método não está em um contexto de transação, já que isso não é necessário, pois ele não insere nada novo no banco que deva ser removido após o fim dos testes.

Para finalizar, vamos criar o arquivo de configuração do Spring especificado na anotação **@ContextConfiguration** da classe de teste, o *application-context-integration-test.xml*. Esse arquivo, mostrado na **Listagem 13**, é específico para testes integrados com o banco de dados. Note que a única diferença em relação ao arquivo *application-context.xml* é o valor da propriedade **persistenceUnitName**, que recebe o nome da unidade de persistência

de teste definida no arquivo *persistence.xml*, garantindo assim que qualquer inserção ou consulta sejam feitas sempre no banco de dados de testes.

Criada a classe com os testes integrados e o arquivo de configuração do Spring, precisamos apenas executar o seguinte comando para fazer o Maven rodar o Liquibase no banco de teste e, em seguida, rodar os testes de integração:

```
mvn integration-test -Dskip.integration.tests=false -Dliquibase.dropFirst=true
```

Listagem 13. Conteúdo do arquivo *application-context-integration-test.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/data/jpa
                           http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="mySqlTest"/>
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
        </property>
    </bean>

    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
    </bean>

    <tx:annotation-driven transaction-manager="transactionManager"/>

    <jpa:repositories base-package="br.com.devmedia.repository"
                    query-lookup-strategy="create-if-not-found"/>
    <context:component-scan base-package="br.com.devmedia"/>
</beans>
```

Essa instrução informa ao Maven que ele deve executar as classes de testes integrados configuradas na fase **integration-test** do plugin *maven-surefire-plugin*. Nesse comando, foi informado o argumento **skip.integration.tests** com o valor **false** para especificar que os testes integrados não devem ser “pulados”.

Gerenciar o banco de dados de uma aplicação sempre foi uma tarefa trabalhosa e suscetível a erros.

Ultimamente, no entanto, novas soluções têm surgido para solucionar esse problema e uma delas é o Liquibase. Tentando agregar a automatização do controle e a possibilidade de ainda conseguir customizar o banco, ele é uma opção que deve ser levada em consideração tanto para novas aplicações quanto para aquelas antigas, que podem se mostrar bem complicadas quando se trata do controle dos scripts SQL da base de dados.

Ademais, a facilidade de trabalhar com o Maven faz com que o Liquibase seja utilizado não só para controlar o banco de dados principal, mas também os bancos de testes, permitindo que o desenvolvedor crie mais testes com dados reais e em uma estrutura idêntica à da base de produção. Lembre-se que criar testes integrados reduz significativamente as chances de encontrar bugs em produção e contar com uma ferramenta como o Liquibase para auxiliar na gestão das bases de dados pode trazer ganhos tanto na performance da equipe quanto na qualidade do projeto.

Autor



Rafael Campos Lima

<http://rafaelcamposl.blogspot.com.br>

Desenvolvedor Java Sênior, graduado em Ciências da Computação pela USP São Carlos, com certificação SCJP e SCWCD.



Links:

Site do projeto Liquibase.

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

Site do projeto Apache Maven.

<https://maven.apache.org/>

Site do plugin do Liquibase para o Maven.

<http://www.liquibase.org/documentation/maven/index.html>

Site do banco de dados MySQL.

<https://www.mysql.com/>

Site do Spring Framework.

<https://projects.spring.io/spring-framework/>



REVISTAS DIGITAIS



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

São mais de 4.000 artigos publicados!

Uma verdadeira biblioteca online perfeita para seus estudos!



Para mais informações :

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

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