



Edição 148 :: R\$ 14,90

 DEV MEDIA

Primeiros passos no mundo da IoT

Como iniciar o seu protótipo
colocando o dispositivo na web

Apache Spark: Processando grafos com Big Data

Implemente funcionalidades escaláveis
e garanta o alto desempenho

JavaServer Faces 2 e BOOTSTRAP

Design responsivo com o melhor de Java e JavaScript



**Gestão de projetos e Integração
Contínua com DevOps**
Aprenda a operar e administrar
software com qualidade e eficiência

**Dominando o Selenium
Web Driver na prática**
Integrando testes automatizados
e unitários com Selenium e TestNG



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!

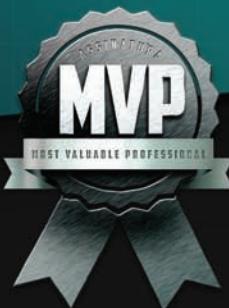
CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

- + 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 – Apache Spark: Processando grafos com Big Data

[Eduardo Felipe Zambom Santana e Luiz Henrique Zambom Santana]

Artigo no estilo Solução Completa, Conteúdo sobre Boas Práticas

17 – Gestão de projetos e Integração Contínua com DevOps

[Pedro E. Cunha Brigatto]

Artigo do tipo Mentoring

34 – Web design responsivo com Bootstrap e JSF 2

[Carlos Antônio Martins]

Conteúdo sobre Novidades, Artigo no estilo Curso

49 – Primeiros passos no mundo da Internet das Coisas – Parte 2

[Rômero Ricardo de Sousa Pereira]

Conteúdo sobre Boas Práticas

62 – Dominando o Selenium Web Driver na prática

[Sueila Sousa]



Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/javamagazine/feedback

REVISTAS DIGITAIS



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

São mais de 4.000 artigos publicados!

Uma verdadeira biblioteca online perfeita para seus estudos!



Para mais informações :

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



Apache Spark: Processando grafos com Big Data

Aprenda como desenvolver aplicações baseadas em grafos com o GraphX do Apache Spark

O Spark tem ganhado bastante destaque nos últimos meses, se tornando, inclusive, o projeto da Apache que mais recebeu commits em 2015. Este resultado se deve principalmente ao excelente desempenho das aplicações escritas com essa ferramenta, que tem como objetivo processar grandes conjuntos de dados de forma paralela e distribuída. O framework mais conhecido com esse propósito é o Hadoop, porém, testes mostram que o Spark possui um desempenho superior ao concorrente em diversas situações. Além disso, possui várias ferramentas que facilitam seu uso, como Spark SQL, Spark Streaming, MLlib e GraphX.

Um tipo de aplicação que pode ser tratado utilizando ferramentas Big Data são os grafos, teoria da Matemática comumente adotada para a solução de um grande número de problemas. Isso porque, se for viável modelar o problema em vértices (ou nós) e arestas (ou arcos), é possível fazer uso de todo o conhecimento já construído sobre esse conceito. Pensando nisso, o Spark disponibiliza diversos algoritmos prontos para serem executados, tanto os clássicos de busca e distância, quanto os focados em problemas atuais, como a verificação de conexão entre usuários de redes sociais e o algoritmo de relevância de páginas criado pelo Google, o PageRank.

Visto que os algoritmos de grafos, tradicionalmente, dependem de grande poder computacional, o Spark também leva esse fato em consideração, sendo uma solução que pode ajudar no processamento de aplicações desse tipo através do uso do GraphX, um componente construído sobre o Spark Core que facilita a análise de

Fique por dentro

Esse artigo é útil para aprendermos como fazer a análise de grafos utilizando a ferramenta Spark, solução Big Data para processamento de grandes volumes de dados. Grafos são utilizados para a solução de uma grande quantidade de problemas em computação como, por exemplo, representar o relacionamento de usuários em uma rede social, desenvolver algoritmos de roteamento em redes de computadores e a implementação de diversos algoritmos de logística (como a roteirização de veículos e a definição do melhor caminho). Porém, desses problemas, diversos demandam enorme poder computacional, como achar a melhor sequência de cidades a serem percorridas ou decidir qual o usuário mais influente de uma rede social, ainda mais quando lidamos com Big Data. Por isso é importante a existência de ferramentas que possibilitem a análise de grafos muito grandes. Pensando nisso, o Apache Spark disponibiliza o componente GraphX, recurso que viabiliza o processamento de algoritmos relacionados a grafos de forma paralela e distribuída.

grafos. Como um dos seus diferenciais, o Spark é executado de forma paralela e distribuída, permitindo a escalabilidade das soluções, e isso com pouca ou nenhuma necessidade de alteração no código fonte da aplicação.

Para apresentar a utilização do GraphX, esse artigo trará a implementação de dois exemplos. O primeiro se preocupa em abordar conceitos básicos para mostrar um grafo com aeroportos e os voos entre eles, assim como ensinar como realizar alguns cálculos sobre o mesmo. Esta solução será construída apenas em Scala, linguagem da API do GraphX. O segundo exemplo, por sua vez, terá o objetivo de simular uma rede social com usuários e suas conexões e também fará diversos processamentos sobre

o grafo. No entanto, terá parte do seu desenvolvimento em Java e parte em Scala, para demonstrar como integrar aplicações escritas nessas duas linguagens.

Teoria dos Grafos

Grafos é um conceito matemático muito popular em computação, pois muitos problemas do mundo real podem ser modelados com vértices (ou nós) e arestas. Por causa disso, existem diversos algoritmos desenvolvidos sobre essa estrutura e que podem ser utilizados independente do domínio da aplicação. Por exemplo, o algoritmo de Dijkstra, um dos mais famosos, pode calcular a melhor rota entre dois roteadores na Internet, entre dois pontos em um mapa e também o custo do voo entre dois aeroportos sem nenhuma alteração em seu código. A **Figura 1** traz um exemplo de grafo que representa o caminho entre algumas cidades. Na imagem é possível perceber que de Lins existe um caminho direto para São Paulo ou Campinas. Porém, se você quiser ir para o Rio de Janeiro, deverá passar por uma dessas duas cidades.

Atualmente, um dos principais exemplos da teoria dos grafos pode ser verificado na implementação de redes sociais, pois um grafo pode ser usado para modelar as pessoas e seus contatos em uma rede, qual usuário compartilhou uma postagem ou uma foto, entre muitas outras opções. Através de outros algoritmos da teoria dos grafos, como a busca em largura ou a busca em profundidade, essas redes sociais conseguem calcular quantos amigos em comum duas pessoas têm, a distância em número de contatos que estamos de outras pessoas, entre outras possibilidades.

Porém, a teoria dos grafos tem um grande problema: normalmente os algoritmos são computacionalmente caros. Um exemplo é o algoritmo do caixeiro viajante, também conhecido como TSP (*Travel Sales Problem*), que consiste em decidir a melhor sequência de cidades que uma pessoa deve visitar, de forma a percorrer a menor distância possível. Esse algoritmo, até hoje, não possui uma solução ótima para grafos de qualquer tamanho.

Em teoria dos grafos, uma questão importante é saber se as arestas dos grafos têm direção ou não. Isto porque em alguns grafos não existe orientação na aresta, ou seja, ambos os vértices que a aresta liga podem ser destino ou origem. Um exemplo onde isso acontece é em redes sociais como o Facebook, na qual não faz sentido ter orientação na aresta. Se duas pessoas estão conectadas na rede, não existe um usuário que é origem e outro que é destino. Os usuários estão ligados da mesma forma. Por sua vez, em uma rede social como Twitter ou Instagram, os vértices precisam ser diferentes, visto que uma pessoa é o “seguidor” e a outra é o “seguido”. Neste caso, faz sentido ter a orientação no grafo para diferenciar o tipo de vértice e saber qual a relação entre eles. Esse conceito é importante porque no GraphX todos os grafos são direcionados, isto é, todas as arestas têm uma orientação, um vértice de origem e um vértice de destino. A **Figura 2** demonstra essas duas opções.

No gráfico da esquerda é possível perceber que todas as pessoas estão relacionadas e não há a ideia de origem e destino. Os vértices são ligados da mesma forma e é possível navegar de

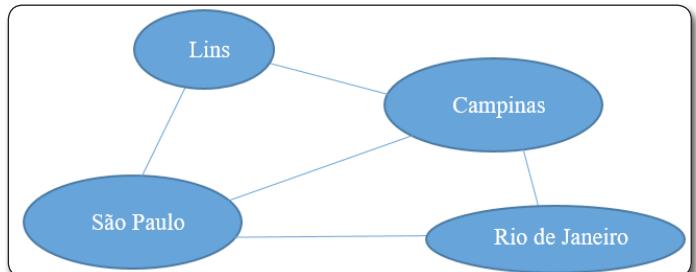


Figura 1. Exemplo de grafo que mostra a conexão de estradas entre algumas cidades

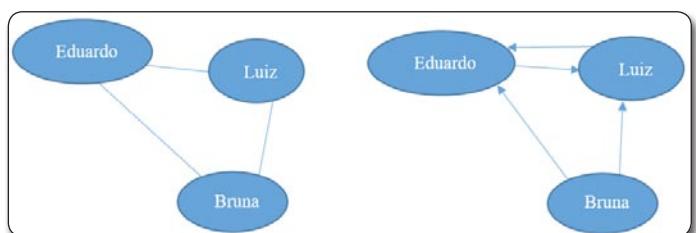


Figura 2. Grafos direcionados e não direcionados

um vértice para outro sem problema. Já no gráfico da direita, o vértice Eduardo “segue” o vértice Luiz, mas não o vértice Bruna. Nesse caso, não é possível navegar desse vértice para o outro. Além disso, note que existe a ideia de origem e destino. Assim, o vértice Bruna possui duas arestas em que ele é a origem e que têm como destino os vértices Eduardo e Luiz.

Apache Spark

O Apache Spark é uma ferramenta para Big Data com foco em baixa latência nas respostas e que oferece um modelo de programação bastante simples, além de possuir diversos componentes para realizar diferentes tipos de processamento de grandes conjuntos de dados de forma paralela e distribuída. A **Figura 3** expõe os principais componentes do Spark. Entre eles se destacam o Spark Core (Apache Spark na figura), núcleo da ferramenta e que oferece os componentes básicos para o processamento em larga escala; Spark MLlib, responsável por implementar os algoritmos de aprendizado de máquina; Spark Streaming, para processamento em tempo real; Spark SQL, para consulta de dados em cluster utilizando a linguagem SQL; e Spark GraphX, responsável pelo processamento de grafos e foco desse artigo.

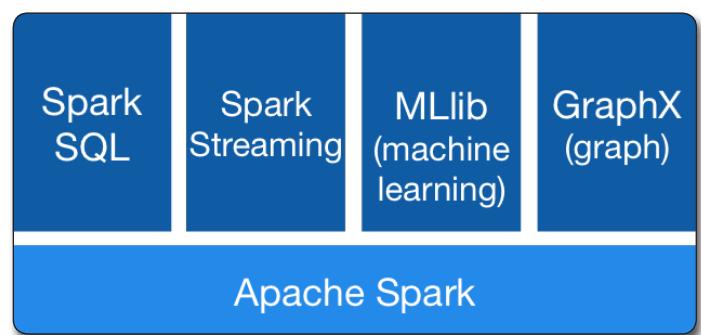


Figura 3. Componentes do Spark

Apache Spark: Processando grafos com Big Data

A arquitetura do Spark é bastante simples, conforme demonstra a **Figura 4**. Além da arquitetura, no entanto, também é importante conhecer os seguintes conceitos:

- **Resilient Distributed Datasets (RDD):** É a abstração para a manipulação de um conjunto de objetos distribuídos no cluster, geralmente armazenados em memória principal. É o principal componente no modelo de programação do Spark;
- **Operações:** representam as transformações (como agrupamentos, filtros e mapeamentos entre os dados) ou ações (como contagens e persistências);
- **Contexto Spark (Spark Context):** como ilustrado anteriormente, representa o principal ponto de entrada ao cluster Spark para uma aplicação que o está utilizando; e
- **Discretized Stream (DStream):** abstrai conjuntos de RDDs que recebem um fluxo contínuo de dados. O DStream é o mais importante conceito quando se utiliza o Spark para streaming, pois facilita o tratamento dos dados que chegam de distintas fontes através de mecanismos simples, consistentes e tolerantes a falhas.

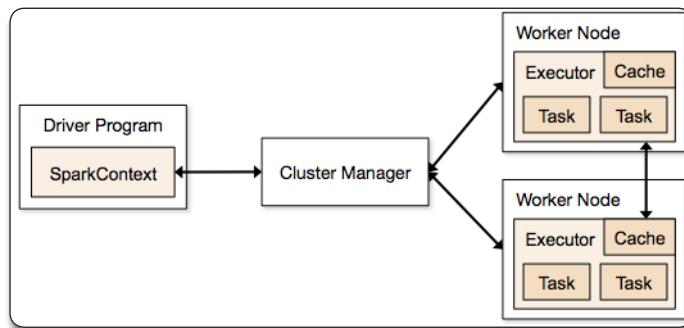


Figura 4. Visão geral da arquitetura do Spark

Com o GraphX a execução das aplicações também segue a arquitetura mostrada na **Figura 4**. Os vértices e arestas do grafo são representados com os RDDs e cada operação no grafo é uma tarefa que pode ser distribuída em um ou mais nós. Uma grande vantagem do Spark é que para o programador é indiferente se a aplicação será executada em um computador apenas ou em um cluster com várias máquinas. O próprio Spark é responsável por realizar as tarefas necessárias para a distribuição, como o escalonamento de tarefas, a comunicação entre as máquinas e o tratamento de possíveis falhas.

Configuração do Eclipse

Como nesse artigo iremos trabalhar com Java e Scala em um mesmo projeto, e utilizaremos o Maven para a gerenciar as dependências, é necessário realizar algumas configurações no Eclipse. Para isso, primeiro, faça o download da última versão desta IDE. Depois, no Eclipse Marketplace, acessado pelo menu *Help > Eclipse Marketplace*, instale o Scala IDE 4.2.x. A **Figura 5** apresenta o item que deve ser instalado.

Com isso, já é possível criar uma aplicação Scala. Porém, ainda é necessário acrescentar o plug-in do Maven para trabalhar



Figura 5. Instalação do IDE Scala

com essa linguagem, caso contrário o Maven sempre criará um projeto Java, que não permite a inclusão de arquivos Scala. Esse plug-in, de nome *m2e-scala*, deve ser instalado utilizando a ferramenta *Install new Software* do Eclipse, disponível no menu *Help > Install New Software*. Para isso, clique no botão *Add* e informe o endereço do repositório da ferramenta (vide seção **Links**). A **Figura 6** expõe a tela com o repositório adicionado. Depois disso, basta selecionar o item *Maven Integration for Eclipse* e clicar no botão *Finish*.

GraphX

Como já mencionado, o GraphX é o componente do Spark para processamento de grafos. Para isso, ele estende os componentes básicos que são disponibilizados no Spark Core. Com o GraphX é possível criar RDDs específicos, como VertexRDD e EdgeRDD, que são instanciados a partir de uma lista de vértices e arestas para manipular esses dados. Além disso, também é possível fazer diversas operações, como encontrar os vizinhos de um vértice, encontrar subgrafos, entre outras.

Para apresentar os conceitos básicos de como trabalhar com esta solução Spark, vamos demonstrar pequenas operações em uma classe Scala e depois veremos como construir uma aplicação completa envolvendo esta linguagem e Java.

A **Listagem 1** mostra como configurar o Maven para utilizar o GraphX. Nesse projeto são necessárias duas dependências: a biblioteca do Scala, na versão 2.10.X, que é a que o GraphX suporta (as versões 2.11.x não são suportadas); e a dependência do próprio GraphX.

Primeiros passos com o GraphX

Nesse exemplo será apresentada uma aplicação simples, que apenas cria um grafo que representa aeroportos e os voos entre eles e executa algumas operações sobre esse grafo (vide **Listagem 2**). Os aeroportos são os vértices do grafo e os voos, as arestas.

Como podemos verificar, inicialmente são definidas as variáveis `conf` e `sc` para definir o contexto do Spark. Em seguida, na variável `aeroportos` são especificados os vértices do grafo, com dois atributos: o ID, que todo vértice deve ter e que é do tipo `Long` (por isso o L depois do número); e uma tupla com duas `Strings`, sendo uma para representar o nome do aeroporto e outra para identificar a cidade em que este fica.

Por sua vez, a variável `voos` representa as arestas do grafo e utiliza a classe `Edge` do GraphX. No construtor dessa classe devem ser passados três parâmetros: o ID do vértice origem, o ID do vértice destino e o custo da viagem entre as cidades. Logo após, dentro do método `main()` é criado o grafo com as variáveis `vertices` e `arestas`. Para a instanciação dessas variáveis, note que são passados os vetores com os aeroportos e os voos utilizando o método `parallelize()` do Spark, que transforma um `array` em um `RDD` (neste caso, `VertexRDD` e `EdgeRDD`).

Com as arestas e vértices definidos, é criado o `graph`, instância da classe `Graph` do GraphX que armazena as informações do grafo e implementa diversas operações como, por exemplo, a contagem de vértices e nós, filtro nos vértices, entre outros. Nessa listagem as operações utilizadas foram: `numVertices`, que retorna o número de vértices do grafo; `numEdges`, para retornar o número de arestas; `filter`, que possibilita filtrar vértices por parâmetro (no caso, o estado do aeroporto); `collectNeighboorId`, responsável por montar um `Map` definindo a chave como o ID dos vértices e o valor como uma lista dos IDs dos vértices que são vizinhos; `reverse`, que inverte a orientação de todas as arestas do grafo; e `pageRank`, que executa o algoritmo PageRank no grafo e retorna a importância de cada um dos aeroportos.

A **Listagem 3** mostra o resultado da execução desse programa (foram retirados os logs do Spark e do GraphX para simplificar a leitura). Primeiro é exibido o número de aeroportos, de voos e de aeroportos em São Paulo. Em seguida é apresentado o número de vizinhos de cada vértice (com o grafo em seu estado normal

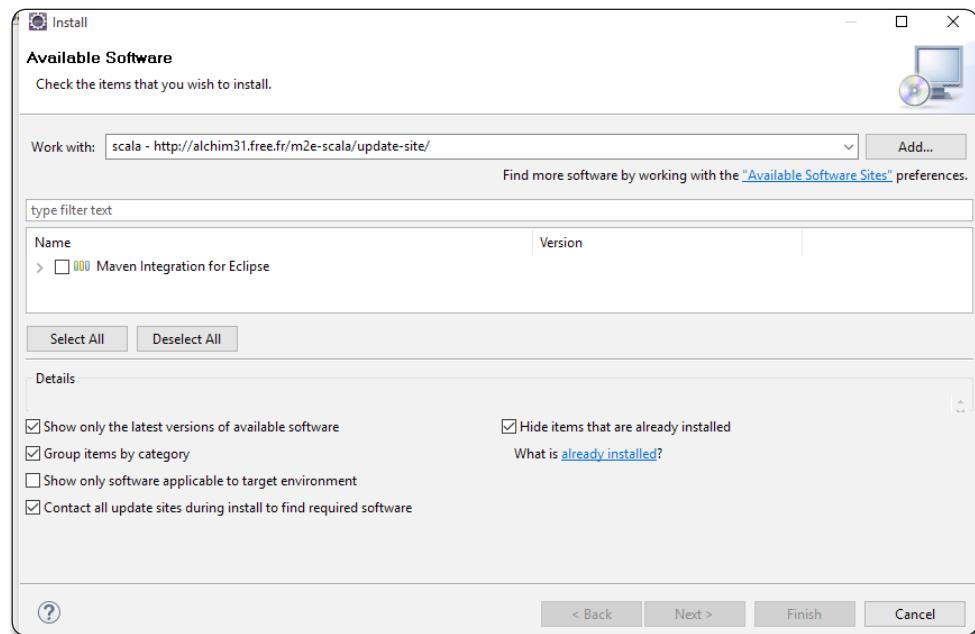


Figura 6. Configurando o Maven para criar projetos Scala

Listagem 1. Dependências para utilizar o GraphX – pom.xml.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.santana</groupId>
  <artifactId>graphx-example</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.scala-lang</groupId>
      <artifactId>scala-library</artifactId>
      <version>2.10.5</version>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-graphx_2.10</artifactId>
      <version>1.5.0</version>
    </dependency>
  </dependencies>

</project>
```

e depois o reverso). Por último, pode-se verificar o resultado da execução do algoritmo PageRank.

Entre as operações realizadas, se destaca a execução de uma implementação do PageRank, que é baseada no algoritmo desenvolvido pelo Google para seu buscador. Sua ideia é simples: quanto mais vértices apontam para outro, isto é, quanto mais arestas estão direcionadas para um vértice, maior a probabilidade desse vértice ser importante. Além disso, cada vértice pode ter um peso associado, e quanto maior o peso dos vértices que apontam para outro, maior será, também, a probabilidade desse vértice ser importante.

Apache Spark: Processando grafos com Big Data

Listagem 2. Aplicação com exemplos básicos do GraphX.

```
package com.santana.graphx_example

import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

object Grafo {

    val conf = new SparkConf()
        .setMaster("local")
        .setAppName("Exemplo GraphX")

    val sc = new SparkContext(conf)

    //lista de aeroportos que serão transformados em vértices
    val aeroportos = Array(
        (1L, ("Guarulhos", "São Paulo")),
        (2L, ("Congonhas", "São Paulo")),
        (3L, ("Galeão", "Rio de Janeiro")),
        (4L, ("Santos Dumont", "Rio de Janeiro")),
        (5L, ("Confins", "Belo Horizonte")),
        (6L, ("Pampulha", "Belo Horizonte")),
        (7L, ("Salgado Filho", "Porto Alegre")))

    //lista de voos que serão transformadas em arestas
    val voos = Array(
        Edge(3L, 1L, 100),
        Edge(1L, 3L, 100),
        Edge(2L, 4L, 150),
        Edge(4L, 1L, 120),
        Edge(5L, 1L, 140),
        Edge(7L, 1L, 180),
        Edge(7L, 1L, 200),
        Edge(7L, 3L, 150),
        Edge(3L, 6L, 150),
        Edge(3L, 7L, 150),
        Edge(6L, 1L, 200))

    def main(args: Array[String]) {
        val vertices: RDD[(VertexId, (String, String))] =
            sc.parallelize(aeroportos)

        val arestas: RDD[Edge[Int]] =
            sc.parallelize(voos)

        val graph = Graph(vertices, arestas)
    }

    println("Num aeroportos: " + graph.numVertices)
    println("Num vôos: " + graph.numEdges)

    //Conta o número de aeroportos do estado de São Paulo utilizando a
    //operação filter
    println("Num aeroportos São Paulo: " + graph.vertices.filter { case
        (id, (aeroporto, estado)) => estado == "São Paulo" }.count)

    //Cria um Map onde a chave é o id de um vértice e o valor para cada chave é uma
    //lista com o id de todos os vizinhos do vértice
    val noVizinhos = graph.collectNeighborhoods(EdgeDirection.In).collect;
    println("Vizinhos Grafo:")
    for (i < 0 to (noVizinhos.length - 1)) {
        println("ID do No: " + noVizinhos(i)._1)
        val vizinhos = noVizinhos(i)._2
        for (j < 0 to (vizinhos.length - 1)) {
            println("ID Vizinho: " + vizinhos(j))
        }
    }

    //Como as arestas no GraphX são direcionadas, inverte a direção das arestas
    val grafoReverso = graph.reverse

    //mostra os vizinhos do grafo reverso
    println("Vizinhos Grafo Reverso:")
    val noVizinhosReverso = grafoReverso.collectNeighborhoods(EdgeDirection.In).collect;
    for (i < 0 to (noVizinhosReverso.length - 1)) {
        println("ID do No: " + noVizinhosReverso(i)._1)
        val vizinhos = noVizinhosReverso(i)._2
        for (j < 0 to (vizinhos.length - 1)) {
            println("ID Vizinho: " + vizinhos(j))
        }
    }

    //executa o algoritmo PageRank
    println("Page Rank:")
    val ranks = graph.pageRank(0.0001).vertices
    val ranksByNomeAeroporto = vertices.join(ranks).map {
        case (id, (nomeAeroporto, rank)) => (nomeAeroporto, rank)
    }

    //imprime o resultado do PageRank
    println(ranksByNomeAeroporto.collect().mkString("\n"))
}
}
```

O parâmetro passado no método `pageRank()` – 0.00001 – é uma constante que indica ao algoritmo quando parar, pois o PageRank é uma solução iterativa que escolhe aleatoriamente um vértice inicial e então percorre o grafo. Assim, a cada iteração pode ser encontrado um resultado diferente, pois o caminho percorrido no grafo será diferente. Quando a diferença entre a média dos pesos dos vértices em duas iterações for menor que o valor passado, o algoritmo deve parar e retornar o resultado encontrado no momento.

A Figura 7 expõe, de forma gráfica, o resultado do algoritmo PageRank para o grafo da aplicação desenvolvida. Essa imagem foi gerada a partir do trecho final da Listagem 3, que apresenta o nome do aeroporto e sua relevância. Assim, quanto maior o vértice, maior a importância do aeroporto.

Note que Galeão (GIG) e Guarulhos (GRU) são os maiores vértices, pois têm ranking de 2,44 e 2,38, respectivamente, seguidos de Confins (CNF), Salgado Filho (POA), Santos Dumont (SDU) e, por fim, os aeroportos de Pampulha (PLU) e Congonhas (CGH). Para estabelecer esses valores, neste exemplo duas características são utilizadas para definir a importância do nó: a quantidade de arestas que chegam a esse nó e a relevância dos nós que apontam para ele.

GraphX com Java e Scala

Com os conceitos básicos demonstrados, a partir de agora será implementada uma aplicação um pouco maior, que integra as linguagens Java e Scala. Para isso, serão utilizados dados de uma rede social baseada no Twitter, onde um usuário pode ter vários seguidores.

Listagem 3. Resultado da execução do programa.

```
Num aeroportos: 8
Num voos: 11
Num aeroportos São Paulo: 2
Vizinhos Grafo:
ID do No: 4
ID Vizinho: 2
ID do No: 1
ID Vizinho: 3
ID Vizinho: 4
ID Vizinho: 5
ID Vizinho: 6
ID Vizinho: 7
ID Vizinho: 8
ID do No: 6
ID Vizinho: 3
ID do No: 3
ID Vizinho: 1
ID Vizinho: 2
ID Vizinho: 3
ID Vizinho: 4
ID Vizinho: 5
ID Vizinho: 6
ID Vizinho: 7
ID Vizinho: 8
ID do No: 7
ID Vizinho: 1
ID do No: 8
ID Vizinho: 1
ID Vizinho: 2
ID Vizinho: 3
ID Vizinho: 4
ID Vizinho: 5
ID Vizinho: 6
ID Vizinho: 7
ID Vizinho: 8
ID do No: 8
ID Vizinho: 1
ID do No: 5
ID Vizinho: 1
ID do No: 2
ID Vizinho: 4
Page Rank:
((Santos Dumont,Rio de Janeiro),0.2774999999999997)
((Guarulhos,São Paulo),2.282758722495344)
((Pampulha,Belo Horizonte),0.8438133987081985)
((Galeão,Rio de Janeiro),2.448753171911289)
((Salgado Filho,Porto Alegre),0.8438133987081985)
((Confins,Belo Horizonte),0.15)
((Congonhas,São Paulo),0.15)

println("Vizinhos Grafo Reverso:")
ID do No: 4
ID Vizinho: 1
```

Esses dados poderiam ser carregados do próprio Twitter, mas por restrições de espaço não faremos isso nesse artigo. Sendo assim, os dados serão carregados de dois arquivos texto: um com os usuários da rede e outro com todas as conexões que existem entre eles. Neste ponto vale ressaltar que o programa funcionará para qualquer configuração feita nesses arquivos. Um exemplo de como esses dados estão organizados pode ser analisado na **Listagem 4**. No repositório onde está o código fonte deste artigo existe um arquivo com o mesmo formato, mas com muito mais dados para que o leitor possa verificar o GraphX funcionando com um exemplo mais próximo do real (vide seção **Links**).

Implementando o código Java

A parte da aplicação escrita em Java é responsável por recuperar as informações da rede social de um arquivo texto. Para isso, são definidas as classes **Usuario** e **Conexao**, representando os objetos que serão manipulados, e a classe que faz a leitura do arquivo, **LeitorArquivo**.

A **Listagem 5** mostra o código da classe **Usuario**. Com conteúdo simples, esta declara o atributo **id**, que representa o identificador do usuário e é necessário para a criação de um vértice no grafo – como demonstrado na criação dos vértices na **Listagem 2** – e o nome do usuário, assim como os métodos de acesso, que foram omitidos.

A classe **Conexao** também é bastante simples (vide **Listagem 6**) e declara apenas dois ids, sendo um chamado **origem**, para o usuário seguidor, e outro chamado **destino**, representando o

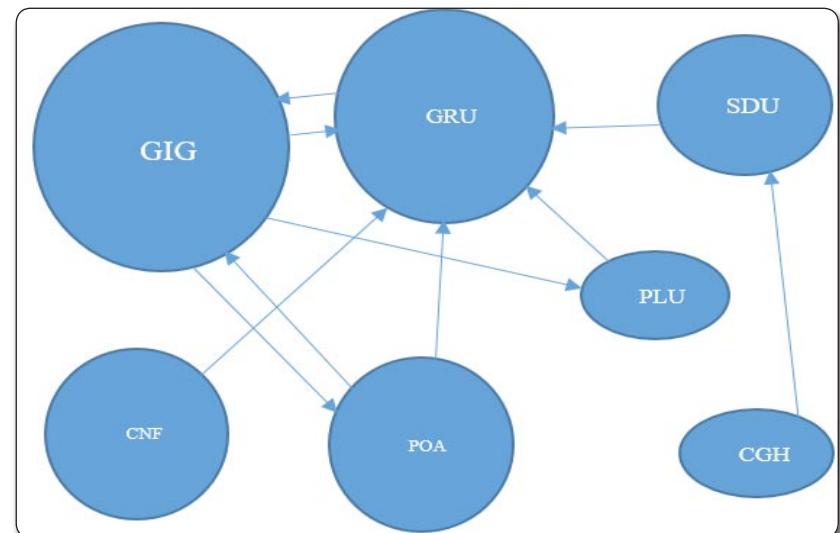


Figura 7. Importância dos aeroportos de acordo com o algoritmo PageRank

usuário que é seguido, além dos métodos de acesso, novamente omitidos.

Para recuperar os dados dos arquivos e criar os objetos das classes **Conexao** e **Usuario**, foi implementada a classe **LeitorArquivo**. A **Listagem 7** traz seu código. Nela, o método **lerArquivoUsuarios()** é responsável por ler os dados do arquivo **usuarios.txt** percorrendo todas as linhas e dividindo a **String** com o método **split()**, utilizando como caractere separador o **";"**. Logo após, cria cada usuário e o adiciona em um array retornado ao final.

Apache Spark: Processando grafos com Big Data

Ainda nessa classe, tem-se o método `lerArquivoConexoes()`, que recupera as conexões entre os usuários. Com código semelhante ao explicado anteriormente, a diferença é que como em uma mesma linha podem existir diversas conexões entre o primeiro nó da linha e os demais, além do `while` que itera sobre as linhas do arquivo, é necessário um `for` sobre o array de Strings que é retornado pelo método `split()`.

Listagem 4. Conteúdo dos arquivos usuarios.txt e conexoes.txt.

```
// usuários da rede – usuários.txt
1;Eduardo
2;Luiz
3;Bruna
4;Sonia
5;Brianda
6;João
7;Maria
8;Carlos

// conexões entre os usuários da rede – conexões.txt
1;2;3;4;5;6
2;1;3;4;8
3;2;4
4;5;8;1
5;4
6;1;3;5
7;4;6;7
8;1;4;7
```

Listagem 5. Código da classe Usuario.

```
package com.santana.devmedia.graphx.classes;

public class Usuario {

    private Long id;
    private String nomeUsuario;

    // gets e sets
}
```

Listagem 6. Código da classe Conexao.

```
package com.santana.devmedia.graphx.classes;
public class Conexao {

    private Long origem;
    private Long destino;

    // gets e sets
}
```

A última classe a ser implementada é a que tem o método `main()`, conforme o código da **Listagem 8**. Este método cria as duas listas, a de usuários e a de conexões, chamando os métodos definidos anteriormente. Em seguida, a classe `Grafo` é instanciada, classe Scala onde serão desenvolvidas todas as operações para a análise do grafo da rede social. Com essa instância em mãos, chamamos seus métodos para realizar as operações.

Listagem 7. Código da classe para leitura dos arquivos com dados de usuários e conexões.

```
package com.santana.devmedia.graphx.classes;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class LeitorArquivo {

    public static List<Usuario> lerArquivoUsuarios() throws IOException {
        List<Usuario> usuarios = new ArrayList<Usuario>();

        // Abre o arquivo com os usuários da rede
        FileReader arq = new FileReader("c:/dev/rede_social/usuarios.txt");
        BufferedReader lerArq = new BufferedReader(arq);
        String linha = lerArq.readLine();
        while (linha != null) {
            System.out.printf("%s\n", linha);

            String[] dados = linha.split(",");
            // Cria objeto usuário para cada linha do arquivo e o coloca na lista de usuários
            Usuario usuario = new Usuario();
            usuario.setId(Long.parseLong(dados[0]));
            usuario.setNomeUsuario(dados[1]);
            usuarios.add(usuario);
            linha = lerArq.readLine();
        }

        return usuarios;
    }

    public static List<Conexao> lerArquivoConexoes() throws IOException {
        List<Conexao> conexoes = new ArrayList<Conexao>();

        // Abre o arquivo com as conexões entre os usuários da rede
        FileReader arq = new FileReader("c:/dev/rede_social/seguidores.txt");
        BufferedReader lerArq = new BufferedReader(arq);
        String linha = lerArq.readLine();
        while (linha != null) {
            System.out.printf("%s\n", linha);

            String[] dados = linha.split(",");
            Long origem = Long.parseLong(dados[0]);
            // para cada linha podem existir diversas conexões
            for (int i = 1; i < dados.length; i++) {
                Conexao conexao = new Conexao();
                conexao.setOrigem(origem);
                conexao.setDestino(Long.parseLong(dados[i]));
                conexoes.add(conexao);
            }
            linha = lerArq.readLine();
        }

        return conexoes;
    }
}
```

Implementando o código Scala

Com todo o código Java apresentado, podemos desenvolver o código Scala utilizando o GraphX, o que será feito na classe Scala chamada **Grafo**. Para facilitar a explicação, este código será dividido em algumas listagens.

Listagem 8. Classe principal da aplicação.

```
package com.santana.devmedia.graphx.classes;

import java.io.IOException;
import java.util.List;

import com.santana.devmedia.graphx.Grafo;

public class RedeSocial {

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

        //instância um objeto da classe grafo, escrita em Scala
        Grafo grafo = new Grafo();

        //carrega os usuários da rede social
        List<Usuario> usuarios = LerArquivo.lerArquivoUsuarios();
        //carrega as conexões entre os usuários da rede social
        List<Conexao> conexoes = LerArquivo.lerArquivoConexoes();

        //executa os métodos que fazem o processamento sobre o grafo
        grafo.criaGrafo(usuarios.toArray(new Usuario[usuarios.size()]),
            conexoes.toArray(new Conexao[conexoes.size()]));
        grafo.contarNumeroUsuarios();
        grafo.verificarUsuarioExiste("Eduardo");
        grafo.verificarUsuariosMaisInfluentes();
        grafo.verificaSeguidores();
        grafo.encontraUsuarioMaisSeguidores();
        grafo.encontraUsuarioMaisSegue();

    }
}
```

A primeira operação que iremos implementar é a criação do grafo (vide **Listagem 9**). Como podemos verificar, essa classe é parecida com o exemplo da **Listagem 2**. Ela possui o array de vértices e arestas e os objetos para a configuração e inicialização do Spark. Depois, no método **criarGrafo()**, a partir dos arrays de usuários e conexões recebidos como parâmetro, são criados os RDDs de vértices e arestas com o comando **for**, e com eles sendo passados como parâmetros no construtor, é criado o objeto **Graph** do GraphX.

No exemplo anterior não foi necessário iterar sobre a lista para criar vértices e arestas, pois o array montado com os dados já estava no formato esperado pelo GraphX. Nesse exemplo, no entanto, os dados são objetos da classe **Usuario** e **Conexao** e por isso é necessário transformá-los para o padrão do GraphX.

Com o grafo montado podemos realizar as operações na rede social. Uma delas é a operação de listar os seguidores de cada usuário, funcionalidade esta implementada no método **encontraSeguidoresPorUsuario()**, apresentado na **Listagem 10**. Para isso, é utilizado o método **collectNeighbors()**, que recupera o id de todos os usuários da rede e o id de seus seguidores.

Nota

Como vamos apresentar os resultados da execução da aplicação no console do Eclipse, no início da classe são desabilitados os logs do Spark para não poluir a saída.

Listagem 9. Criação do grafo com os dados dos usuários e conexões.

```
package com.santana.devmedia.graphx

//imports omitidos...

class Grafo {

    //desabilita logs do Spark
    Logger.getLogger("org").setLevel(Level.OFF)
    Logger.getLogger("akka").setLevel(Level.OFF)

    var vertices: Array[(Long, (String))] = null
    var edges: Array[Edge[String]] = null

    val conf = new SparkConf()
        .setMaster("local[2]")
        .setAppName("Simple Application")
    val sc = new SparkContext(conf)

    var g: Graph[(String), String] = null

    def criaGrafo(usuarios: Array[Usuario], conexoes: Array[Conexao]) {

        vertices = new Array[(Long, (String))](usuarios.length)
        edges = new Array[Edge[String]](conexoes.length)

        //cria os vértices a partir da lista de usuários
        for (i <- 0 to (usuarios.length - 1)) {
            vertices(i) = (usuarios(i).getId, (usuarios(i).getNomeUsuario))
        }

        //cria as arestas a partir da lista de conexões
        for (i <- 0 to (conexoes.length - 1)) {
            edges(i) = Edge(conexoes(i).getOrigem, conexoes(i).getDestino, "segue")
        }

        val usersRDD: RDD[(Long, (String))] = sc.parallelize(vertices)
        val relationshipsRDD: RDD[Edge[String]] = sc.parallelize(edges)

        g = Graph(usersRDD, relationshipsRDD)

    }

    //outros métodos com os cálculos sobre o grafo.

}
```

Como este método recupera apenas os ids dos usuários, é necessário executar um **join** do resultado da coleta de vizinhos com o RDD dos vértices do grafo. Em seguida, para exibir os seguidores de cada usuário, é feita uma iteração para percorrer todos eles e, no **for** interno, é realizada uma iteração para listar os nomes dos seguidores de cada um. É importante notar que na chamada do método **collectNeighbors()** é passado o parâmetro **EdgeDirection.In**. Isso indica que queremos verificar os vizinhos que apontam para o vértice que está sendo analisado. Existe também a opção **EdgeDirection.Out**, que recupera os vizinhos que são apontados pelo vértice que está sendo analisado.

Apache Spark: Processando grafos com Big Data

Listagem 10. Método para listagem de seguidores dos usuários.

```
def encontraSeguidoresPorUsuario() {  
  
    // coleta o map com o id de cada vértice e a lista de seus vizinhos  
    val ids = g.collectNeighbors(EdgeDirection.In);  
    val usuarios = g.vertices.join(ids).map {  
        case (id, (nomeUsuario)) => (nomeUsuario)  
    }.collect()  
  
    //vértice por vértice, escreve o nome do usuário e quem o segue  
    for (i <- 0 to (usuarios.length - 1)) {  
        println("Usuário:" + usuarios(i)._1)  
        val vizinhosIds = usuarios(i)._2  
        for (j <- 0 to (vizinhosIds.length - 1)) {  
            val nomeVizinho = vizinhosIds(j)._2  
  
            println("Nome Seguidor:" + nomeVizinho)  
        }  
    }  
}
```

Algumas operações simples, mas que podem ser importantes para a verificação dos dados das redes sociais é saber o número de usuários e o número de conexões entre eles. Para isso, basta conhecer o número de vértices e de arestas do grafo, através do método **count**. A **Listagem 11** mostra o código para fazer essa contagem. O método **contarNumeroUsuarios()** imprime o número de vértices do grafo e o método **contarNumeroConexoes()** imprime o número de arestas.

Listagem 11. Como obter o número de usuários e de conexões.

```
def contarNumeroUsuarios() {  
    println("Número de usuários da rede:" + g.vertices.count)  
}  
  
def contarNumeroConexoes() {  
    println("Número de usuários da rede:" + g.edges.count)  
}
```

Outra opção interessante é verificar se um usuário existe na rede social. A **Listagem 12** apresenta o código dessa operação, onde é utilizado o nome do usuário para realizar a filtragem dos vértices. O retorno do método **filter** é um RDD com os vértices que respeitam a restrição passada. Como apenas queremos verificar se o usuário existe, simplesmente chamamos o método **count**, que retorna o número de elementos do RDD. Caso esse número seja maior que 0, o usuário existe; caso contrário, não.

É possível descobrir também qual usuário tem mais seguidores. Para isso, podemos utilizar o grau do vértice, que representa o número de arestas que estão ligadas a esse vértice. Aqui, saiba que existem dois tipos de graus: de entrada e de saída. O grau de entrada de um vértice é o número de arestas em que ele é o destino, e o grau de saída é o número de arestas em que o vértice é a origem. Assim, para verificar o usuário que tem mais seguidores, precisamos comparar o grau de entrada de cada vértice.

A **Listagem 13** contém o código da função **encontraUsuarioMaisSeguidores()**. Inicialmente, é definida a função interna

max, que compara o grau de dois vértices do grafo e retorna qual é o maior. Essa função será passada como parâmetro (em Scala, funções são objetos e podem ser passadas como parâmetros para outras funções) para fazer uma operação de redução que vai verificar todos os vértices do grafo, encontrar qual tem mais seguidores e retornar o seu id e qual o grau de entrada dele. O código responsável por realizar essa tarefa é o **g.inDegrees.reduce(max)**. Com isso é encontrado o id do usuário que tem mais seguidores. Para melhorar a saída da aplicação e apresentar o nome do usuário, é realizada uma busca para encontrar esta informação pelo id. Por fim, é impresso o nome do usuário e o número de seguidores que ele possui.

Listagem 12. Código que verifica existência de usuário.

```
def verificarUsuarioExiste(nomeUsuario: String) {  
    //filtra os vértices pelo nome do usuário passado como parâmetro.  
    val num = g.vertices.filter { case (id, (nome)) => nome == nomeUsuario }.count  
    println("Usuário existe:" + (num > 0))  
}
```

Quase da mesma forma, é possível descobrir o usuário que segue mais pessoas na rede. A **Listagem 14** contém o código da função **encontraUsuarioMaisSegue()**. Como podemos notar, a diferença é que ao invés de utilizar o grau de entrada, é utilizado o grau de saída dos vértices (**outDegrees**).

Listagem 13. Código para recuperar usuário com mais seguidores.

```
def encontraUsuarioMaisSeguidores() {  
  
    //função para comparar o grau de cada usuário da rede social  
    def max(a: (Long, Int), b: (Long, Int)): (Long, Int) = {  
        if (a._2 > b._2) a else b  
    }  
  
    //Verifica o usuário que tem mais seguidores  
    val maxInDegree: (Long, Int) = g.inDegrees.reduce(max)  
  
    val usuario = g.vertices.filter { case (id, (nome)) => id == maxInDegree._1 }.collect  
    println("Usuário:" + usuario(0)._2)  
    println("Número Seguidores:" + maxInDegree._2)  
}
```

Ainda seguindo a proposta dos últimos dois métodos, é possível encontrar o usuário que tem mais relacionamentos em geral, isto é, aquele que é mais seguido e mais segue outros usuários. A **Listagem 15** mostra o código para esse cálculo. A diferença dessa implementação para as duas listagens anteriores é que ela utiliza a propriedade **degrees**, que retorna o número de arestas ligadas ao vértice, independente de este ser origem ou destino na relação.

Finalmente, a última operação implementada é o cálculo para obtenção dos usuários mais influentes, o que pode ser feito com o algoritmo PageRank. A **Listagem 16** exibe o código do método **verificarUsuariosMaisInfluentes()**.

Listagem 14. Código para recuperar usuário que segue mais pessoas.

```
def encontraUsuarioMaisSegue() {  
  
    //função para comparar o grau de cada usuário da rede social  
    def max(a: (Long, Int), b: (Long, Int)): (Long, Int) = {  
        if (a._2 > b._2) a else b  
    }  
  
    //Verifica o usuário que segue mais pessoas  
    val maxOutDegree: (Long, Int) = g.outDegrees.reduce(max)  
  
    val usuario = g.vertices.filter { case (id, (nome)) =>  
        id == maxOutDegree._1}.collect  
    println("Usuário:" + usuario(0)._2)  
    println("Número de usuários seguidos:" + maxOutDegree._2)  
  
}
```

Como pode ser verificado, a execução deste é bastante simples: é chamado o método **pageRank()** da classe **Graph** e este retorna o id de cada usuário e a sua “nota”. Para expor o nome do usuário e não apenas o id, é feito um join do retorno do PageRank (variável **ranks**) com os vértices do grafo e o resultado dessa operação é armazenado na variável **ranksPorUsuario**. Por fim, são impressos as notas e os nomes dos usuários.

Listagem 15. Código que encontra usuário com mais conexões

```
def encontraUsuarioMaisConexoes() {  
  
    //função para comparar o grau de cada usuário da rede social  
    def max(a: (Long, Int), b: (Long, Int)): (Long, Int) = {  
        if (a._2 > b._2) a else b  
    }  
  
    //Verifica o usuário que tem mais conexões no grafo.  
    val maisConexoes: (Long, Int) = g.degrees.reduce(max)  
  
    val usuario = g.vertices.filter { case (id, (nome)) =>  
        id == maxOutDegree._1}.collect  
    println("Usuário:" + usuario(0)._2)  
    println("Número Seguidores:" + maxOutDegree._2)  
  
}
```

A Listagem 17 mostra o resultado da execução do programa **RedeSocial**, onde podemos encontrar o número de usuários da rede, a verificação se o usuário “Eduardo” existe, o resultado da execução do algoritmo PageRank, o nome de todos os usuários e o nome de seus seguidores, o usuário que tem mais seguidores e o que mais segue outros usuários.

Entre as ferramentas de processamento de grafos, o GraphX se destaca por possibilitar a análise de grafos de forma paralela e distribuída, permitindo a execução de algoritmos complexos em grafos com muitos vértices e arestas. Existem benchmarks que mostram o GraphX analisando grafos com mais de 100 milhões de arestas e 6 milhões de vértices.

Além dos exemplos abordados nesse artigo, outros casos onde o GraphX pode ser empregado são: aplicações de logística, para

Listagem 16. Código do algoritmo PageRank para a rede social.

```
def verificarUsuariosMaisInfluentes() {  
    //calcula o PageRank para todos os usuários, o pageRank é um algoritmo  
    //iterativo, que fica analisando a  
    val ranks = g.pageRank(0.0001).vertices  
    //faz o join do nome do usuário com o rank desse usuário  
    val ranksPorUsuario = g.vertices.join(ranks).map {  
        case (id, (nomeUsuario, rank)) => (nomeUsuario, rank)  
    }  
  
    //imprime os dados na tela  
    println(ranksPorUsuario.collect().mkString("\n"))  
}
```

Listagem 17. Saída gerada com a execução do programa de análise da rede social.

```
Número de usuários da rede: 8  
Usuário existe: true  
verificarUsuariosMaisInfluentes  
(Sonia,0.1900992670285877)  
(João,0.5354747673489915)  
(Carlos,0.9101405276317055)  
(Luiz,0.6570964437073549)  
(Eduardo,1.3196015334348719)  
(Bruna,0.6655668995305231)  
(Maria,0.5689812755952409)  
(Brianda,1.146482623215017)  
verificaSeguidores  
Usuário: Sonia  
Nome Seguidor: Eduardo  
Nome Seguidor: Luiz  
Nome Seguidor: Bruna  
Nome Seguidor: Brianda  
Nome Seguidor: Maria  
Nome Seguidor: Carlos  
Usuário: João  
Nome Seguidor: Eduardo  
Nome Seguidor: Maria  
Usuário: Carlos  
Nome Seguidor: Luiz  
Nome Seguidor: Sonia  
Usuário: Luiz  
Nome Seguidor: Eduardo  
Nome Seguidor: Bruna  
Usuário: Eduardo  
Nome Seguidor: Luiz  
Nome Seguidor: Sonia  
Nome Seguidor: João  
Nome Seguidor: Carlos  
Usuário: Bruna  
Nome Seguidor: Eduardo  
Nome Seguidor: Luiz  
Nome Seguidor: João  
Usuário: Maria  
Nome Seguidor: Maria  
Nome Seguidor: Carlos  
Usuário: Brianda  
Nome Seguidor: Eduardo  
Nome Seguidor: Sonia  
Nome Seguidor: João  
encontraUsuarioMaisSeguidores  
Usuário: Sonia  
Número Seguidores: 6  
encontraUsuarioMaisSegue  
Usuário: Eduardo  
Número de usuários seguidos: 5
```

Apache Spark: Processando grafos com Big Data

definir a melhor sequência de lugares a visitar; sistemas para definir as preferências de compras de usuários, para relacionar produtos que tenham as mesmas características ou o mesmo público alvo em uma loja virtual.

Por fim, vale ressaltar que assim como o GraphX, existem outras ferramentas para o processamento de grafos com Big Data, como o Apache Giraph, que funciona com o Apache Hadoop, e GraphLab. Contudo, o GraphX é o que demonstra o melhor desempenho, além da vantagem de o modelo de programação ser muito parecido com o do Spark, o que facilita o aprendizado e a migração de aplicações para o modelo de grafos.

Autor



Luiz Henrique Zambom Santana

lhzsantana@gmail.com

É bacharel e mestre em Ciência da Computação. Atualmente cursa doutorado também em Ciência da Computação na UFSC. Possui mais de 10 anos de experiência em programação Java e há dois anos atua com tecnologias de Big Data. Trabalhou em projetos para grandes empresas no Brasil, Argentina e Alemanha. Atualmente é consultor de Elasticsearch.



Autor



Eduardo Felipe Zambom Santana

ezambomsantana@gmail.com

É bacharel e mestre em Ciência da Computação pela UFSCar. Possui mais de 10 anos de experiência em programação. Atualmente é aluno de doutorado na USP e é professor na Universidade Anhembi Morumbi.



Links:

Endereço para download do Spark.

<http://spark.apache.org/downloads.html>

GitHub do projeto Apache Spark.

<https://github.com/apache/spark>

Página do GraphX dentro do projeto Spark.

<http://spark.apache.org/graphx/>

GitHub do autor, com os códigos apresentados no artigo.

<https://github.com/lhzsantana/rt-javamagazine>

Comparação entre as ferramentas GraphX, Giraph e GraphLab.

http://www.cs.berkeley.edu/~kubitron/courses/cs262a-F13/projects/reports/project21_report_ver2.pdf

Página do projeto Apache Giraph.

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

M2-Scala para criar projetos Maven com Scala.

<http://alchim31.free.fr/m2e-scala/update-site/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Gestão de projetos e Integração Contínua com DevOps

Aprenda nesse artigo como operar e administrar software com alta qualidade e eficiência

Em artigo publicado na Edição 147 da Java Magazine, abordamos alguns dos pontos mais relevantes sobre desenvolvimento de software à luz de um paradigma que vem agitando o mercado no Brasil e no mundo: DevOps.

Naquela ocasião, algumas plataformas e ferramentas foram aplicadas – sempre sob a perspectiva ‘Dev’ desta nova cultura – na evolução de um projeto Java até o ponto ilustrado pela **Figura 1**. Por esta imagem, notamos que o trabalho que será realizado neste artigo será baseado em um projeto de software com uma estratégia sólida de versionamento de código, através do Git. Percebemos, ainda, que todo o seu ciclo de vida é gerenciado pelo Apache Maven, e que todo o trabalho é viabilizado a partir de uma IDE muito popular, o Eclipse.

Esta combinação de tecnologias, somada a uma equipe madura e bem qualificada, caracteriza um excelente ponto de partida na busca pela excelência na ‘camada’ de desenvolvimento, e nos permite abordar, ao longo do texto que se segue, aspectos fundamentais dentro de outra frente igualmente importante: operações.

Enquanto desenvolvimento é um termo que remete à construção do software em si, a frente de operações engloba aspectos vinculados à sua validação e utilização. Operadores estão, frequentemente, preocupados com o provisionamento e monitoramento de toda a infraestrutura necessária para todo o trabalho de criação e, também, homologação do software. São profissionais que atuam em contato tanto com a equipe de desenvolvimento quanto com clientes, com o principal objetivo de garantir a fluidez do projeto por meio do provisionamento de toda a infraestrutura necessária em ambas as pontas.

Fique por dentro

Este artigo ajudará o leitor a entender algumas das principais atividades relacionadas à administração e operação de projetos baseados na cultura DevOps. O texto cobre tanto o conceito quanto a aplicação de atividades como Integração Contínua e Gestão de projetos, por meio de ferramentas e plataformas como Trello, Redmine e Jenkins, e será útil para o leitor que desejar não apenas conhecer tais tecnologias, mas adotá-las em seu dia a dia.

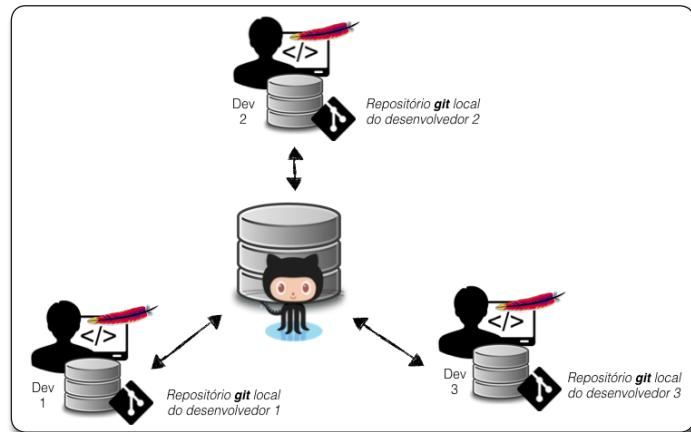


Figura 1. Situação do projeto tema ao início deste artigo

A primeira questão que abordaremos neste artigo é a da Integração Contínua de software. Ao implementá-la, garantiremos que toda alteração realizada em nosso repositório de código resulte, imediatamente, em uma atividade de construção (*build*) do projeto com o objetivo de identificar os impactos das alterações submetidas, tanto sob a perspectiva da consistência estrutural

(por meio de operações como a compilação de todo o código) quanto de comportamento (através da execução de suítes de testes automáticos).

A Integração Contínua de Software

Ao aplicar um conjunto de tecnologias no desenvolvimento de um projeto de software de qualidade em nosso ambiente de trabalho, estamos nos prevenindo contra uma série de fatores de risco. Um bom sistema de versionamento, por exemplo, nos torna capazes de administrar a evolução do código-fonte com muito mais precisão. Ao estabelecermos o Maven na gestão do ciclo de vida de um sistema, delegamos a execução de atividades como controle de dependências, compilação, empacotamento e testes – dentre outras – para uma plataforma de software, tornando-as muito menos propensas a falhas.

Entretanto, manter toda esta configuração distribuída, sob a responsabilidade individual de cada desenvolvedor no projeto, é insuficiente. É importante que essas mesmas características sejam espelhadas em um ambiente considerado oficial, do qual se possa extrair versões de um produto, submetê-las à avaliação de equipes de testes e, em última instância, aos próprios clientes. A integração contínua é o primeiro grande passo neste sentido, e é sobre isto que passaremos a conversar nesta seção.

Mas o que é integração contínua, afinal?

Integração contínua é um dos temas mais populares em Engenharia de Software na atualidade. Mas, afinal de contas, esta integração remete a que? E por que contínua? O que isto quer dizer? De que estamos falando, mais especificamente?

Estas podem ser perguntas recorrentes que, principalmente profissionais mais novatos – inundados por uma imensidão de novidades divulgada diariamente em nosso mercado – podem estar se fazendo. Vamos, então, avaliar o conceito por trás do tema.

Quando se fala em integração contínua (ou simplesmente CI), os holofotes estão direcionados essencialmente para o código-fonte. Integração, portanto, remete à prática de se avaliar os impactos diretos da incorporação de qualquer conteúdo, submetido por

um desenvolvedor dentro de uma equipe, em um repositório de código-fonte centralizado.

A motivação para o surgimento desta prática é que, à medida que desenvolvedores trabalham para transformar documentação de requisitos – funcionais e não funcionais – em algo executável/concreto, é natural que colisões de código possam surgir. É bem provável que, em algum momento, um mesmo módulo de um projeto seja alterado, simultaneamente, por dois ou mais profissionais, e um episódio como esse pode, eventualmente, resultar em problemas.

A integração contínua atua exatamente antecipando tais situações, permitindo que o time reaja imediatamente e colabore para garantir uma evolução muito mais segura do produto. No trato popular, portanto, adotar integração contínua é simplesmente seguir a velha máxima de que ‘é melhor prevenir que remediar’.

E como faço para começar a trabalhar com integração contínua?

A natureza preventiva da prática de integração contínua parece boa, não? Agora que vimos o que significa integrar software continuamente, a próxima pergunta que o leitor pode estar se fazendo é como, enfim, torná-la realidade em seu projeto. Para respondê-la, configuraremos este processo sobre o repositório de código-fonte de nosso projeto tema.

Inicialmente, devemos identificar, dentre as principais soluções disponíveis no mercado, a que melhor atenda nossas necessidades. Ao fazer isso, o próximo passo é decidirmos sobre qual infraestrutura nosso processo de integração contínua será instalado. Por fim, deve-se verificar os passos a executar, para implementar e publicar este serviço.

Antes de darmos o primeiro passo, porém, é importante que destaquemos dois aspectos fundamentais associados à prática de integração contínua: o primeiro, de ordem cultural, trata de uma questão comportamental; enquanto o segundo, mais técnico, refere-se ao risco da dependência tecnológica.

Boas práticas de versionamento: não represe seu código!

Uma das más práticas que podem prejudicar o andamento de qualquer projeto de software – principalmente aqueles que buscam



se beneficiar das vantagens da cultura DevOps – é o represamento de código em repositório privado. Tal comportamento pode resultar no acúmulo de muito conteúdo na máquina do desenvolvedor e, ao mesmo tempo, ignorar muitas submissões de código que outros desenvolvedores podem estar realizando, diariamente, conforme o projeto evoluí. Quando, finalmente, uma submissão de todo o material acumulado é feita, haverá grande probabilidade do surgimento de conflitos de versão, sendo necessário um bom tempo até que tudo seja resolvido e, enfim, volte-se a ter código estável.

A dica que fica, portanto, é submeter código o mais frequentemente possível, de modo a evitar este tipo de pesadelo que, quase sempre, prejudica consideravelmente o desempenho da equipe. O consenso atual entre profissionais ativos de comunidades e fóruns pela Internet é de que o intervalo razoável para cada submissão é de uma vez ao dia.

O cliente para versionamento de código não deve ser impedimento para o processo

Outra característica que devemos sempre ter em mente diz respeito ao cliente de versionamento que utilizamos. Como sabemos, o Git é o sistema em si e temos, à nossa disposição, algumas implementações populares como GitHub e Bitbucket, que o oferecem como um serviço. Em nosso projeto tema, decidimos utilizar o GitHub, mas é importante que o leitor saiba que não há, nesta escolha, nenhuma especificidade de natureza técnica; tudo o que estamos fazendo através do GitHub é, também, absolutamente possível de ser realizado com Bitbucket ou qualquer serviço similar. Este é um detalhe fundamental para quem deseja manter seus projetos flexíveis, pois a independência de plataformas abre espaço para mudanças estruturais sem comprometer a sua essência.

Integração contínua na prática: uma introdução ao Jenkins

Agora que vimos o que é CI em sua essência, é chegado o momento de colocá-lo para funcionar. Escolheremos, para tanto, uma das tantas ofertas de mercado e, em seguida, a estratégia que usaremos para o provisionamento e configuração de todo o ambiente. Os critérios que utilizamos para selecionar o servidor de CI para nosso projeto estão listados a seguir:

- Modelo de licenciamento flexível;
- Natureza *open-source*;
- Vasta documentação, oficial e extraoficial; e
- Alta popularidade.

Ao confrontar todos esses requisitos com todas as ofertas de mercado, chegamos à conclusão de que o melhor candidato para o nosso projeto é o Jenkins, e é sobre ele que passaremos a conversar de agora em diante.

Jenkins, o servidor do povo

O Jenkins é uma solução de integração e entrega contínua mais popular no mercado de software. De código aberto e fácil instalação

e administração, pode ser implantado tanto *in-house* – utilizando a própria infraestrutura da empresa – quanto consumido como serviço a partir de plataformas na web. Em ambos os casos, sua administração sempre será realizada a partir de uma interface web, bastante amigável, e um grande acervo de vídeos, documentos e análises por toda a Internet garantem respaldo no esclarecimento de dúvidas e dificuldades que venhamos a enfrentar.

Jenkins, o servidor flexível

Distribuído sobre a licença MIT, o Jenkins é uma solução de CI altamente flexível. Assim, é permitido, de acordo com o conteúdo deste modelo de licenciamento, que sejam feitas modificações inclusive nas funcionalidades centrais do produto em questão. Além disso, permite-se utilizá-lo em soluções comerciais sem a necessidade de se pagar nada por isso. Tudo isto, aliado aos inúmeros recursos que podem ser incorporados à versão *pura* do servidor – por meio de plug-ins –, faz com que o Jenkins seja uma das soluções mais atraentes do mercado quando o assunto é integração contínua.

Jenkins runtime: *in-house* ou PaaS?

Para instalar o Jenkins, não precisamos mais do que um container web e o WAR da aplicação em si. Isto já seria suficiente para colocá-lo para executar em nosso ambiente de desenvolvimento ou, ainda, em um servidor dentro de nossa própria infraestrutura. Esta seria a abordagem popularmente conhecida como *in-house*, adotada ainda hoje por muitas empresas, em diversas situações, devido a uma série de fatores que abririam uma discussão que não cabe neste artigo. De uma maneira sucinta, podemos vincular a adoção deste modelo a uma única palavra: conservadorismo.

A alternativa à estratégia *in-house* é substituirmos nossa infraestrutura através da contratação de empresas que nos forneçam o mesmo *setup* de servidores, na forma de serviço. Isto é o que se convencionou chamar como PaaS (Plataforma como Serviço) e tem, gradualmente, abandonado a condição de tendência para a de realidade, principalmente devido à evolução das tecnologias de computação em nuvem (orquestradores, virtualização, containers, dentre outras).

Neste artigo, aproveitaremos a oportunidade para trabalhar somente com tecnologias oferecidas na “nuvem” e, para o caso específico do Jenkins, utilizaremos uma plataforma muito bem-concebida no mercado, chamada OpenShift, sobre a qual passaremos a conversar um pouco mais detalhadamente na próxima seção.

Criando um servidor Jenkins com a OpenShift

OpenShift é uma plataforma da Red Hat que oferece, através de uma interface web intuitiva, a possibilidade de criar e hospedar aplicações em uma infraestrutura na nuvem. Ao adotá-la, a primeira grande vantagem para as empresas e desenvolvedores em geral é a liberdade adquirida com a delegação do provisionamento das máquinas necessárias para manter nossas soluções no ar.

Gestão de projetos e Integração Contínua com DevOps

Nesta plataforma, boa parte dos recursos disponíveis pode ser usada em caráter gratuito, à distância de alguns cliques e alguma customização. Conforme nossas soluções passam a se tornar mais profissionais ou exigem maior poder computacional e/ou memória/disco, há dois pacotes oferecidos

para nos atender – Silver e Gold. Além disso, há a possibilidade de se expandir nossa infraestrutura a partir da contratação de mais containers – chamados de *gears* – em um modelo de especificação por demanda. Todas essas informações podem ser obtidas detalhadamente na página da OpenShift.

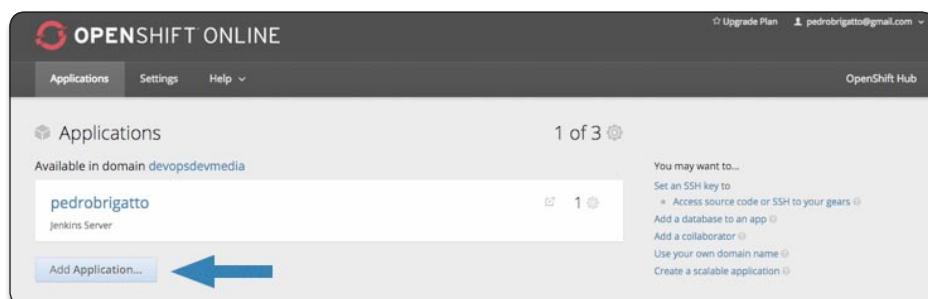


Figura 2. Página inicial da área de usuário de uma conta OpenShift

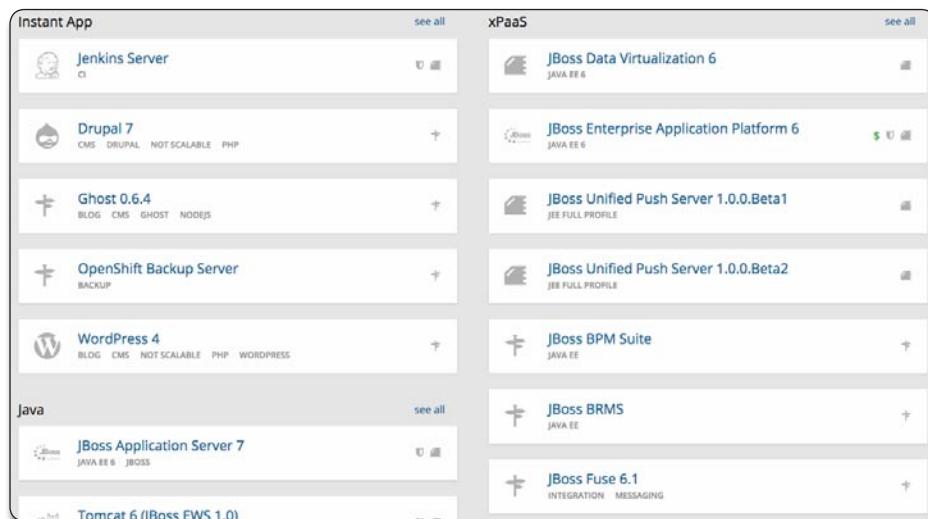


Figura 3. Alguns dos tipos de aplicação suportados pela OpenShift

Based On: Jenkins Server Cartridge

Jenkins is a continuous integration (CI) build server that is deeply integrated into OpenShift. See the Jenkins info page for more. Based on Jenkins 1.509+

<http://www.jenkins-ci.org>

OpenShift maintained

Receives automatic security updates

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

Because this is your first application, you need to provide a domain under which your applications will be grouped

OpenShift will automatically register this domain name for your application. You can add your own domain name later.

Source Code: https://github.com/pedrobrigatto/dev master

We'll create a Git code repository in the cloud, and populate it with a set of reasonable defaults. If you provide a Git URL, your application will start with an exact copy of the code and configuration provided in this Git repository.

Figura 4. Configuração do Jenkins na OpenShift

Criar uma conta na OpenShift é uma tarefa bastante simples. Ao entrar no site da OpenShift, escolha a opção *SIGN UP FOR FREE* e complete o cadastro com suas informações pessoais. Ao concluir este passo, é necessária a confirmação de sua conta por meio de um link enviado via e-mail, o que a ativará imediatamente e deixá-la pronta para uso.

Quando acessamos nossa conta, deparamo-nos com a página ilustrada na Figura 2. Neste caso, o projeto que estamos usando como base para este artigo já havia sido gerado e está, portanto, listado. No caso de uma conta recém-criada, nada haverá para ser listado em seu primeiro acesso.

Para adicionar uma aplicação a partir desta página, precisamos clicar no botão *Add Application*, indicado por uma seta azul na Figura 2. Ao fazermos isto, uma nova tela se abre e, nela, devemos informar o tipo de aplicação em que estamos interessados. Esta página é apresentada na Figura 3.

Notemos, antes de prosseguir com o trabalho, a variedade de aplicações que podem ser criadas e hospedadas a partir da OpenShift. É impressionante a quantidade de serviços atualmente disponibilizados na nuvem, e tudo da forma mais simplificada possível, tirando do desenvolvedor e, também, de operadores, boa parte da complexidade que sabemos que existe na configuração de plataformas desta natureza.

Nesta página, ao selecionarmos a opção *Jenkins Server* (indicada com uma seta vermelha, na Figura 3), somos encaminhados para a criação do projeto propriamente dito. Durante o cadastro, devemos definir a URL que será associada ao portal de administração de nossa instância do Jenkins. Em nosso projeto tema, utilizamos o endereço <http://pedrobrigatto-devopsdevmedia.rhcloud.com>, como podemos observar na Figura 4.

Outra informação muito importante a ser fornecida é o endereço do repositório de código-fonte que o Jenkins utilizará, incluindo o *branch*. Neste caso, como também podemos observar na Figura 4, faremos uso do código-fonte disponível no *branch master*, de nosso repositório remoto hospedado no GitHub, localizado



Figura 5. Credenciais para acesso ao portal de administração do Jenkins

S	W	Name ↓	Last Success	Last Failure	Last Duration
		devmedia_devops_job2	3 days 2 hr - #9	4 days 11 hr - #5	16 sec
		devops_devmedia_job	N/A	4 days 12 hr - #6	0.87 sec

Figura 6. Página principal do portal de administração do Jenkins

em https://github.com/pedrobrigatto/devmedia_devops_series.

Fornecidas todas as informações no cadastro da aplicação, basta que confirmemos sua criação e, dentro de poucos instantes, estaremos aptos a acessá-la. O processo é concluído com a exibição das credenciais criadas pela OpenShift, utilizadas para acessar o Jenkins e administrá-lo por sua interface web. Esta mensagem de configuração está ilustrada na **Figura 5**.

Quando isto acontece, já podemos abrir uma nova aba em nosso navegador e acessar nosso servidor na URL que definimos durante o cadastro – e informada, também, na mensagem exibida na **Figura 5**. Assim que efetuarmos o login, seremos apresentados ao painel principal de administração do Jenkins, exposto na **Figura 6**.

Um detalhe bem interessante de todo este processo de criação da instância do Jenkins é que ele toma por base algumas informações que fornecemos – como o repositório de código-fonte – para

pré-configurar o servidor de CI com os recursos necessários (como o plug-in para comunicação com o GitHub). Caso o usuário tivesse optado pela instalação *in-house*, recursos como esse teriam que ser instalados manualmente.

Dentro do painel de administração do Jenkins, duas funções – ambas indicadas com uma seta azul na **Figura 6** – merecem nossa atenção especial. A primeira, *New Item*,

será usada para criar novos itens (leia-se *jobs*); enquanto a segunda, *Manage Jenkins*, permite que verifiquemos e alteremos todas as configurações do servidor. Ambas serão muito acessadas ao longo das próximas seções.

Jenkins instalado. Como configurá-lo?

Antes de vermos como foi criado nosso primeiro *job*, há algumas características



Gestão de projetos e Integração Contínua com DevOps

que precisamos editar nas configurações do servidor. Algumas têm caráter puramente corretivo, contornando defeitos conhecidos da versão atual do Jenkins; enquanto outras remetem à instalação de componentes a serem utilizados no processo de build que configuraremos em instantes. Ao longo das próximas seções, exploraremos em detalhes tudo o que precisou ser feito em nosso servidor de CI para que, enfim, pudéssemos executar nossos builds da forma que esperávamos.

0 serviço de download do Jenkins

O primeiro item que veremos refere-se a uma falha conhecida no serviço de download de componentes do Jenkins. Sua configuração original não gerencia a

identificação de versões apropriadamente, solicitando esta informação através de um campo de texto (ao invés de uma lista com valores pré-definidos, como era de se esperar). A consequência direta disso é uma maior complexidade na configuração de alguns elementos, delegando para o usuário a responsabilidade de encontrar o texto exato que corresponda a uma versão válida – para o Jenkins – do item cuja instalação é necessária. Normalmente, o resultado prático acaba sendo uma série de tentativas e erros, tornando a experiência bastante frustrante.

Para entender melhor do que estamos falando aqui, observe a **Figura 7**. Nesta imagem, o problema já fora corrigido, e a seleção da versão do JDK é feita por meio de uma *combo box*. Antes de aplicarmos

Listagem 1. Script de correção para builds automáticos para o Jenkins.

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

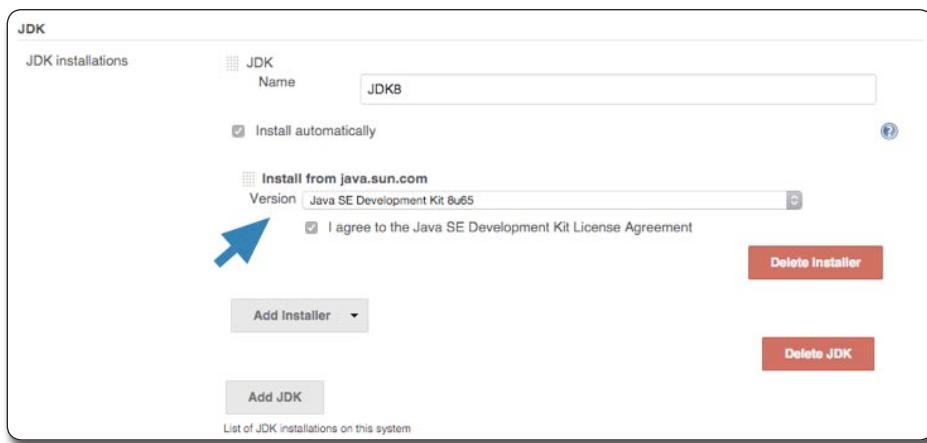


Figura 7. Configuração do JDK para instalação automática via Jenkins

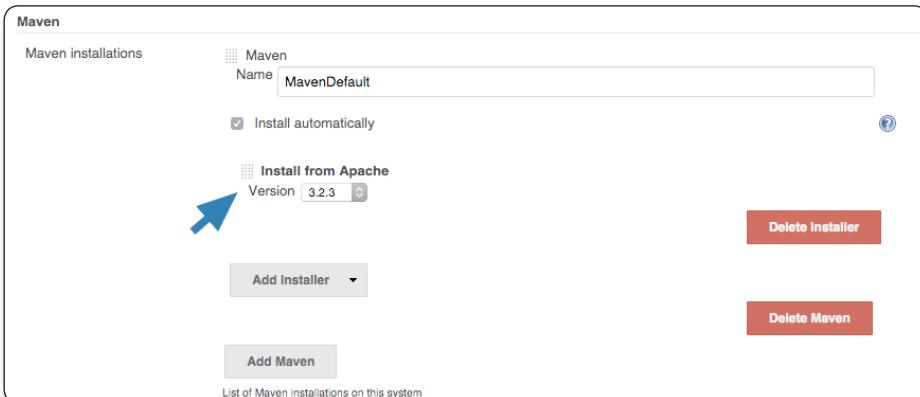


Figura 8. Configuração do Maven para instalação automática via Jenkins

a correção que veremos a seguir, porém, este mesmo campo era apresentado como uma caixa de texto. Todo este problema é contornado por meio da execução das instruções exibidas na **Listagem 1**.

O conteúdo desta listagem deve ser executado a partir de uma janela chamada *Script Console*, disponível no painel de administração em *Jenkins > Manage Jenkins > Script Console*. Assim que a operação for concluída, por garantia, devemos também recarregar as configurações do servidor acessando a função *Jenkins > Manage Jenkins > Reload Configuration from Disk*.

A partir daí os formulários de instalação de componentes que exijam a seleção de uma versão por parte do operador apresentarão todas as opções suportadas por meio de *combo boxes*, e não mais caixas de texto. Além disso, o serviço de download, nos bastidores, será capaz de mapear essas versões, identificá-las em suas origens, baixá-las e instalá-las automaticamente. A história completa deste defeito pode ser acessada por meio de uma referência listada na seção **Links**, ao final do artigo.

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

Definindo o número de executores

O pré-requisito essencial para um job entrar em execução é a existência de um executor disponível. Este é o nome dado aos agentes, no Jenkins, que coordenam todo o trabalho configurado em um job. Segundo a documentação oficial da OpenShift, o número padrão de executores definido é de dois por cada instância do Jenkins, o que nos leva a concluir que não deveríamos nos preocupar em checar tal configuração.

Na prática, entretanto, colocar um job para executar em uma instância recém-criada do Jenkins na OpenShift resulta em uma condição de permanente inércia, na qual o processamento jamais chega a ser iniciado. Ao procurar a causa raiz para este comportamento, verificamos que a documentação não traduz a realidade, e

o número padrão de executores definido em nosso servidor é 0.

Ao estabelecermos um valor maior do que zero para este campo, haverá agentes disponíveis para processar o job e, portanto, veremos toda a execução ser realizada conforme o esperado. Portanto, recomendamos ao leitor que preste atenção especial nesta característica quando estiver configurando o seu servidor, observando os passos listados a seguir:

1. Vá até *Jenkins > Manage Jenkins > Configure System*;
2. Edite o campo *# of executors* com um número maior que 0; e
3. Salve o trabalho, clicando no botão *Save*.

Configurando o instalador do JDK

Corrigido o número de executores e, também, o comportamento do serviço de download do Jenkins, passaremos agora para a configuração dos componentes necessários para que o build de nosso projeto tema seja realizado.

O primeiro desses elementos é o JDK, necessário para que plataformas como o Maven possam executar corretamente. Neste tutorial, delegaremos sua instalação para o próprio Jenkins, configurando um instalador com as características ilustradas na **Figura 7**. Como vemos, trata-se de um instalador, de nome *JDK8*, que instalará automaticamente (note o check box *Install automatically* selecionado), direto do site da Oracle, a versão *8u65* do JDK.

Configurando o instalador do Maven

Assim como o procedimento configurado para o JDK, na seção anterior, a instalação do Maven é realizada pelo serviço de download do Jenkins. Nosso papel, na administração da plataforma, é definir os instaladores com os quais desejamos trabalhar, e é isto que faremos nesta seção. Na tela de configuração do sistema, acessada a partir do caminho *Jenkins > Manage Jenkins > Configure System*, devemos navegar até a seção intitulada *Maven*, ilustrada na **Figura 8**. Nesta seção, encontramos um botão *Add Installer*, que, quando clicado, apresentará as abordagens de instalação suportadas pelo serviço. Em nosso exemplo,



Figura 9. Configuração do Git para instalação automática via Jenkins

selecionamos a opção *Install from Apache*, que procurará pelo instalador diretamente no site do projeto *Maven*, hospedado pela fundação *Apache*. Ao selecionarmos esta opção, um pequeno formulário será carregado dinamicamente, exibindo todos os campos necessários para a configuração do instalador em criação: um campo para seu nome, uma *combo box* que indicará a versão do Maven e um *check box* indicando se a instalação deverá ocorrer automaticamente. No exemplo ilustrado na **Figura 8**, criamos um instalador de nome *MavenDefault*, que instalará automaticamente a versão 3.2.3 do *Maven*.

Configurando o instalador do Git

Nosso próximo passo é garantirmos que o Jenkins seja capaz de se comunicar com o repositório hospedado no GitHub. Para isto, devemos garantir que ao menos uma versão do Git esteja instalada e configurada no servidor. Para nos certificarmos que esta condição é válida, ainda na configura-

ção global do sistema (em *Jenkins > Manage Jenkins > Configure System*), navegue até a subseção de título *Git*, ilustrada na **Figura 9**. Observe, então, que existe um botão *Add Installer* que, quando clicado, carregará um formulário de configuração dinamicamente. Neste formulário, devemos preencher as seguintes informações:

- O nome do instalador (que, em nosso caso, é *Default*);
- O caminho até o executável. Normalmente, este campo é preenchido automaticamente com o valor *git*. Como, ao longo da instalação, a variável de ambiente *PATH* é configurada para reconhecer o programa, basta que preenchamos este campo com seu nome – *git*;
- A habilitação – ou não – da instalação automática.

Em nosso exemplo, como visto na **Figura 9**, definimos um instalador de nome *Default*, que instalará e configurará, automaticamente, o Git no ambiente servidor.



Gestão de projetos e Integração Contínua com DevOps

Jenkins instalado e configurado. Hora do job?

Neste momento nosso servidor já está devidamente configurado. O que fizemos até esta seção é tudo o que precisamos para poder dar início à criação de um job para automatizar nosso processo de build.

Para começar, selecione a opção *Jenkins > New Item*. A tela apresentada será semel-

hante à ilustrada na **Figura 9**. O Jenkins nos oferece alguns templates que podem acelerar um pouco esta etapa. Entretanto, optar por um desses modelos pode não ser um bom negócio, e este é exatamente o caso em que nos encontramos. Como o leitor pode perceber na figura, há um item chamado *Maven project* que, ao que tudo

indica, atua como acelerador na criação de jobs para projetos gerenciados via Maven. Parece, portanto, ser exatamente o que precisamos, correto? Não exatamente.

O ponto é que este template geralmente nos traz mais dor de cabeça do que agilidade. Trata-se de um problema relativamente antigo no Jenkins, relacionado a uma falha na definição do endereço do agente Maven que realizará o processo de build propriamente dito. Caso se interesse por informações detalhadas sobre este problema, verifique a referência deixada na seção **Links**.

Nossa saída é ignorar este template e partir para a criação de nosso job baseado na primeira opção exibida na **Figura 10**: *Freestyle project*. Esta será a abordagem utilizada em nosso tutorial, exposta em todos os seus detalhes mais relevantes a partir da próxima seção.

Assim que atribuímos um nome ao nosso job, mantendo como modelo a opção *Freestyle project*, devemos confirmar a operação clicando no botão *OK*. O próximo passo é começar a editar suas características, como veremos a partir da seção seguinte.

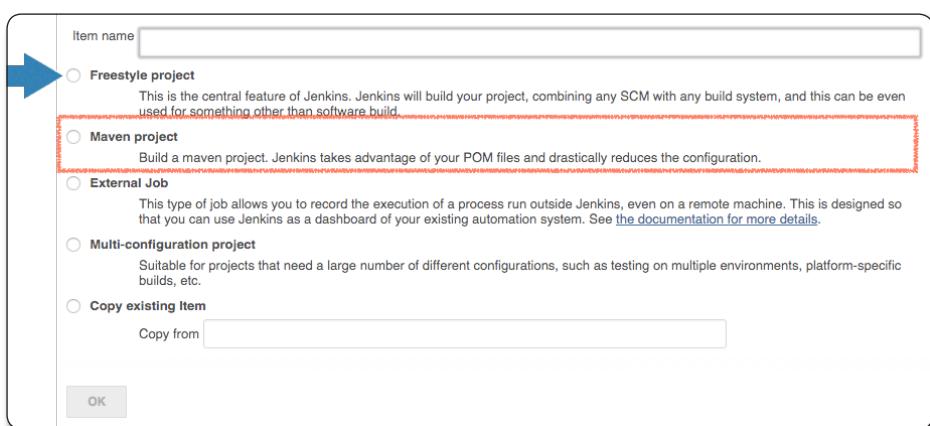


Figura 10. Modelos (templates) de configuração de jobs no Jenkins

A screenshot of the Jenkins 'Source Code Management' configuration dialog for a 'Freestyle project'. Under the 'Repositories' section, 'Git' is selected. The 'URL of repository' field contains 'https://github.com/pedrobrigatto/devmedia_devops_series.git'. Below it, 'Branches to build' is set to 'Branch Specifier (blank for default): */master'. There are 'Add' and 'Delete Branch' buttons. At the bottom, there are 'Save' and 'Apply' buttons.

Figura 11. Configuração do repositório Git usado pelo job

A screenshot of the Jenkins 'Build' configuration dialog. It shows a single step named 'Invoke top-level Maven targets'. Under 'Maven Version', 'MavenDefault' is selected. In the 'Goals' field, the value '-f \$WORKSPACE/devops/pom.xml clean install' is entered. There is an 'Advanced...' button and a 'Delete' button at the bottom right.

Figura 12. Configuração dos passos do build do projeto via job

As propriedades de nosso repositório de código

Ao selecionarmos a opção *Freestyle project* e confirmar a operação clicando no botão *OK*, será aberta uma página em que o job passará a ser configurado. O primeiro aspecto que cobriremos é a comunicação com o repositório de código-fonte, editando as propriedades agrupadas abaixo do subitem *Source code management*, conforme indica a **Figura 11**. Por esta imagem, vemos um formulário no qual devemos informar os seguintes dados:

- O tipo de sistema de versionamento utilizado (em nosso caso, Git);
- A URL do repositório (neste caso, a URL de nosso projeto no GitHub); e
- O *branch* em que está o projeto a ser clonado pelo Jenkins (neste exemplo, **/master*).

Essas são as propriedades que devemos definir em termos de repositório. Ao preenchermos todos os campos indicados pela **Figura 11**, é chegado o momento de passarmos para o próximo aspecto do job: seus passos.

Os passos do processo de build

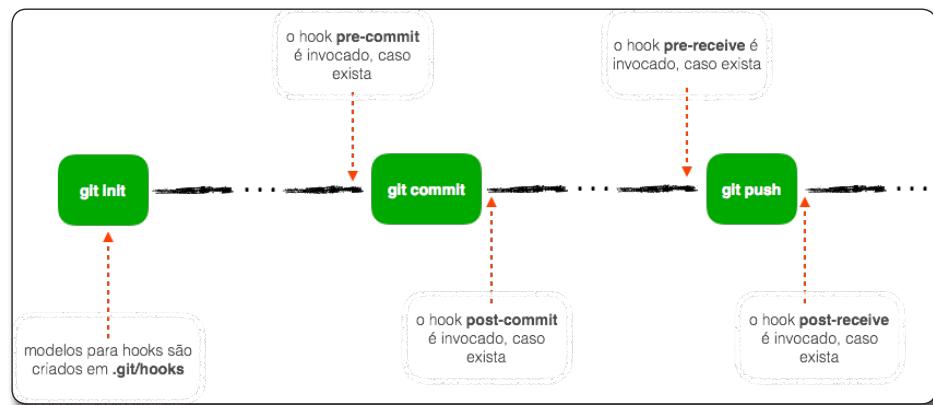
O job que estamos configurando terá como responsabilidade coordenar o processo de build de nosso projeto tema que, como vimos em seções anteriores, está todo baseado sobre o Maven. Aprendemos, portanto, a compor um plano de build baseado inteiramente em recursos oferecidos por esta plataforma.

A edição de um plano de build de um job, no Jenkins, é realizada em uma subseção identificada pelo título *Build*, ilustrada na Figura 12. Observe, nesta imagem, a existência de um botão com o título *Add build step*. É através dele que especificamos todos os passos de build necessários em nosso processo. Ao clicarmos nesse botão, são listadas todas as opções suportadas pelo Jenkins como passos de um build. Em nosso caso, o tipo a ser selecionado é *Invoke top-level Maven targets*, que carregará um formulário dinâmico solicitando todas as informações sobre o que desejamos fazer neste passo. Como o leitor pode ver na Figura 12, o nosso único passo do build deve ser assim definido:

- **Maven version:** o instalador *Maven-Default*, criado por nós, deve ser selecionado;
- **Goals:** os objetivos pré-definidos pelo Maven – somente aqueles que nos interessam – devem ser indicados neste campo.

O build de nosso projeto tema envolverá uma limpeza inicial, que removerá quaisquer arquivos gerados por execuções anteriores e, logo em seguida, as seguintes operações: compilação, testes, empacotamento e instalação. Em termos práticos, os *goals* do Maven que nos interessam são apenas dois: *clean* e *install*. Enquanto o goal *clean* caracteriza a limpeza do projeto, *install* engloba todas as outras etapas citadas.

Outro ponto importante que merece destaque no campo *Goals* ilustrado na Figura 12 é a indicação do caminho completo até o arquivo descriptor do projeto, especificado pelo parâmetro *-f*. Ao utilizá-lo, estamos informando o caminho completo até o arquivo descriptor do projeto (*pom.xml*), garantindo que o Maven encontre o que precisa para iniciar a execução dos goals estabelecidos.



Note que, para identificar o caminho completo deste arquivo, fazemos uso de uma variável de ambiente chamada **WORKSPACE**, que guarda uma referência para o workspace do Jenkins – ou, em outras palavras, o diretório dentro do qual são clonados os projetos gerenciados pelo servidor. Ao utilizá-la na definição do caminho até o POM de nosso projeto, livraremos da responsabilidade de saber o caminho deste workspace, que é definido nos bastidores pelo próprio Jenkins.

Preenchidos todos os campos do formulário de build, já temos condições de rodar este job manualmente, de dentro da própria interface web do Jenkins. Entretanto, queremos e precisamos de algo além disso. Para podermos dizer que nosso processo é, definitivamente, de integração de contínua, devemos garantir que ele possa ser executado automaticamente, a cada submissão de código-fonte no repositório.

Para isso, é necessário entender um mecanismo muito importante do Git e, por último, um recurso de autenticação do próprio Jenkins. Estas duas características, que veremos em detalhes a partir da próxima seção, são os dois últimos detalhes que temos que ajustar para, enfim, concluirmos todo este processo.

Git e seus hooks

Olhando um pouco mais a fundo para a arquitetura do Git, percebemos o quanto poderoso este sistema de versionamento é. Há características que, normalmente, não nos despertam a atenção até que precisemos delas. Nossa objetivo, nesta

seção, é identificar se há alguma forma de customizarmos nosso repositório para controlar eventos específicos como:

- Submissão de código;
- Clonagens; e
- Atualização de repositórios privados (*pull requests*).

Felizmente, para nós, este tipo de customização é possível e, melhor ainda, simples de fazer. Este conceito é denominado *hook* e é implementado por meio de arquivos mantidos dentro do diretório *.git/hooks* de todo repositório Git. A única premissa para esses arquivos é que recebam nomes pré-definidos dentro da arquitetura do Git, para serem devidamente lidos e processados, tão logo os estados associados a ele sejam atingidos.

Para ilustrar este cenário, vejamos a Figura 13. Esta imagem mostra, ainda que de forma simplificada, situações em que hooks podem ser programados:

1. Imediatamente antes de se realizar um *commit*, podemos verificar se, por exemplo, a mensagem de submissão de código atende a convenções estabelecidas pela empresa. Podemos, ainda, validar a formatação de código (via ferramentas como *Checkstyle* ou similares), dentre tantas outras coisas; e
2. Assim que todo o procedimento de submissão tiver sido realizado, o Git se encontrará em um estado conhecido como *post-receive*. Em uma situação como esta, podemos iniciar um processo de build a partir de um servidor de CI, dentre outras operações.

Gestão de projetos e Integração Contínua com DevOps

A lista de todos os hooks Git, bem como o seu significado e possíveis casos de uso, estão devidamente documentados no site oficial da ferramenta, que pode ser conferido a partir da seção **Links**.

No nosso estudo de caso, estamos interessados em disparar, automaticamente, um build a cada atualização ocorrida no repositório de código. Vejamos como isto foi feito em nosso projeto.

Para habilitar o que acabamos de ver, serão necessários três passos:

1. Criar um token, no Jenkins, para viabilizar a invocação de builds automáticos a partir de uma URL, diretamente do hook do Git;
2. Configurar as permissões de acesso do usuário anônimo do Jenkins, para que a comunicação entre o Git – por meio do hook – e o servidor possa ocorrer; e
3. Configurar o hook propriamente dito, para disparar o build automático sempre

que algum material for submetido ao repositório remoto do projeto.

Agora que sabemos o que fazer, em relação aos hooks, vejamos como esta configuração ocorre.

Configurando gatilhos de build e tokens de acesso no Jenkins

Veremos, nesta seção, como configurar um token e facilitar, desta forma, a comunicação entre o Jenkins e o Git. A configuração em si é bastante simples: dentro do job que criamos no tópico “Jenkins instalado e configurado. Hora do job?”, vá até a opção *Build Triggers* e selecione *Triggers build remotely*. Feito isso, o leitor observará que é aberto um campo de título *Authentication Token*, e é aí que devemos definir o token que será utilizado pelo Git para disparar o build. Em nosso exemplo, conforme ilustrado

na **Figura 14**, utilizamos *devmedia_devops_token*. Observe, ainda, que logo abaixo do campo, há uma descrição que explica como utilizar este token em uma eventual chamada ao servidor. Basta que substituamos *JENKINS_URL* pela URL do nosso servidor Jenkins e *TOKEN_NAME* pelo token que definirmos, e realizemos a chamada ao servidor. No nosso estudo de caso, a URL ficaria da seguinte forma: https://pedrobrigatto-devopsdevmedia.rhcloud.com/job/devmedia_devops_job2/build?token=devmedia_devops_token.

Alterando as permissões de acesso do usuário Jenkins

O segundo passo que devemos dar em direção ao build automático passa pela configuração do usuário anônimo do Jenkins para a chamada ao servidor. Aqui, cabe ressaltar ao leitor que poderíamos trabalhar com outro usuário qualquer, configurando-o exatamente para este propósito e nada mais. Entretanto, por questões de espaço e simplicidade, concluímos que o mais importante aqui é entender a funcionalidade em si, e não as suas especificidades. Por este motivo, adotaremos o usuário padrão utilizado nestes casos, que é o anônimo.

O problema a ser resolvido aqui é que este usuário não tem as permissões necessárias para realizar uma chamada ao servidor e disparar, por exemplo, um processo de build. Quando tentamos fazer isto apenas com a configuração original do usuário Jenkins, verificamos que a chamada falha e nada ocorre. Devemos, portanto, revisar as permissões deste usuário em *Jenkins > Manage Jenkins > Configure Global Security*, no servidor.

Quando fazemos isto, encontramos uma tela similar à exibida na **Figura 15**. Perceba que, neste caso, já selecionamos a opção *Administer* para o usuário *Anonymous*. No caso do leitor, assim que abrir as configurações de seu servidor, verá que esta opção não se encontra ativada. Basta, portanto, que a ative e clique no botão *Save* da página para confirmar as alterações.

Caso o leitor tenha preferido configurar o Jenkins localmente ao invés de utilizar a OpenShift, é importante também verificar alguns pontos para que a **Figura 15** reflita,

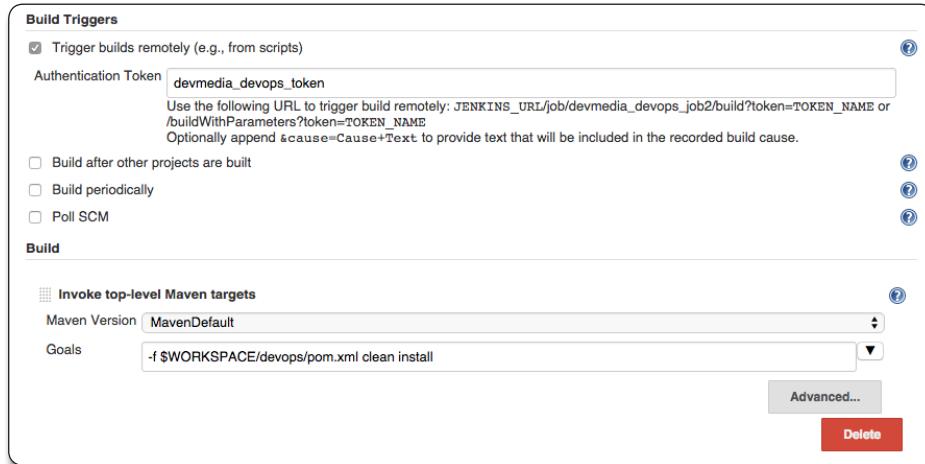


Figura 14. Preparação para o uso de triggers na realização de builds automáticas via Jenkins

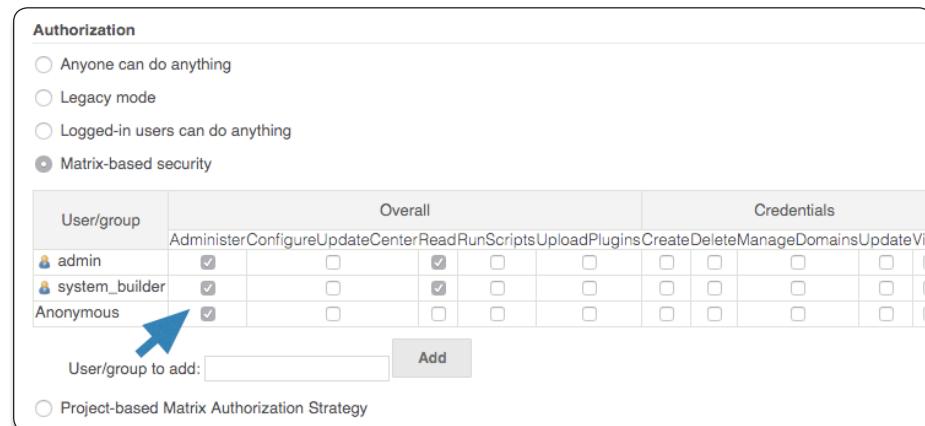


Figura 15. Configuração de permissões de acesso no Jenkins

de fato, o que verá na seção *Authorization* das configurações globais de segurança do Jenkins. São eles:

- Em *Jenkins > Manage Jenkins > Configure Global Security > Access Control > Authorization*, certifique-se que a opção *Matrix-based security* está selecionada; e
- Quando esta opção é selecionada, o esperado é que o usuário anônimo já esteja configurado. Verifique, portanto, se isto se aplica em seu caso.

Depois que salvarmos as alterações feitas nesta seção, é importante que nos lembremos de recarregar as configurações do disco, para que o Jenkins aplique a nova configuração ao ambiente em execução. Para fazer isto, basta que o leitor vá até *Jenkins > Manage Jenkins* e clique no item *Reload Configuration from Disk*.

Configurando o hook em nosso repositório

Agora que o usuário anônimo já está configurado corretamente e temos um token criado, é hora de configurar o hook Git para disparar builds automáticas a cada vez que o repositório central, remoto, for modificado.

O trigger que utilizaremos neste caso é o *post-update* que, conforme a documentação, é disparado sempre que todo o processo de submissão de código está completo, sendo o momento mais apropriado para notificar usuários e/ou invocar outros serviços.

Para usar este recurso, o primeiro passo que daremos é renomear o arquivo, removendo sua extensão *.sample*, originalmente inclusa. Assim que iniciamos o repositório através do comando *git init*, já vimos que uma série de templates é criada para nos dar uma base sobre como funciona cada hook. Todos esses arquivos são criados com esta extensão *.sample*, bastando que a removamos para que sejam hooks, a partir de então, ativos.

Removida a extensão do arquivo *.git/hooks/post-receive*, é hora de editá-lo. Deste modo, abra-o em seu editor preferido e insira o código exibido na **Listagem 2**. Lembre-se, no entanto, de alterar a URL em negrito seguindo as recomendações listadas a seguir:

Listagem 2. *post-receive – hook do Git invocado quando todo o processo de submissão de código está completo.*

```
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ pwd  
/Users/pedrobrigatto/Personal/Projects/DevOpsArticle/devmedia_devops_series  
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ cd .git/hooks/  
Pedro-Brigattos-MacBook:hooks pedrobrigatto$ ls -la  
total 80  
drwxr-xr-x 11 pedrobrigatto staff 374 Dec 7 09:48 .  
drwxr-xr-x 14 pedrobrigatto staff 476 Dec 7 09:46 ..  
-rwxr-xr-x 1 pedrobrigatto staff 452 Nov 7 11:47 applypatch-msg.sample  
-rwxr-xr-x 1 pedrobrigatto staff 896 Nov 7 11:47 commit-msg.sample  
-rwxr-xr-x 1 pedrobrigatto staff 313 Dec 7 09:48 post-update  
-rwxr-xr-x 1 pedrobrigatto staff 398 Nov 7 11:47 pre-applypatch.sample  
-rwxr-xr-x 1 pedrobrigatto staff 1642 Nov 7 11:47 pre-commit.sample  
-rwxr-xr-x 1 pedrobrigatto staff 1348 Nov 7 11:47 pre-push.sample  
-rwxr-xr-x 1 pedrobrigatto staff 4951 Nov 7 11:47 pre-rebase.sample  
-rwxr-xr-x 1 pedrobrigatto staff 1239 Nov 7 11:47 prepare-commit-msg.sample  
-rwxr-xr-x 1 pedrobrigatto staff 3611 Nov 7 11:47 update.sample  
Pedro-Brigattos-MacBook:hooks pedrobrigatto$ cat post-update  
#!/bin/sh  
#  
# An example hook script to prepare a packed repository for use over  
# dumb transports.  
#  
# To enable this hook, rename this file to "post-update".  
  
exec git update-server-info  
curl https://pedrobrigatto-devopsdevmedia.rhcloud.com/job/devmedia_devops_job2/  
build?token=devmedia_devops_token&delay=0sec  
Pedro-Brigattos-MacBook:hooks pedrobrigatto$
```

- Substitua o trecho *https://pedrobrigatto-devopsdevmedia.rhcloud.com* pelo endereço informado pela OpenShift para a sua aplicação Jenkins; e
- Substitua o trecho *?token=devmedia_devops_token* utilizando o token definido em sua própria configuração do Jenkins.

Validando o hook e o job

Para testar todas essas configurações na prática, temos algumas possibilidades. A primeira delas, mais para efeito de observação do Jenkins isoladamente, é executar o comando destacado em negrito na **Listagem 2** diretamente em um terminal (console). Isto nos permitiria entender se o comando, de fato, está funcionando. Caso funcione em um terminal, as chances de falhar dentro do hook que configuramos é praticamente nula.

Podemos, ainda, digitar somente a URL em um browser e verificar se, no Jenkins, um build é automaticamente iniciado. Esta é outra boa maneira para verificar se o token criado fez efeito e se tudo está devidamente configurado em nosso servidor.

É possível verificar, também, o resultado do build, detalhadamente, a partir do console. Para acessar este console, devemos selecionar o nosso job na página principal do Jenkins e, em seguida, clicar no item *Console Output* do menu lateral. Igualmente, mas em um caráter bem mais resumido, podemos ver a ‘saúde’ de nossos builds na página inicial de administração do Jenkins, conforme ilustra a **Figura 16**. Este é um trecho normalmente exibido no canto inferior direito desta página e, além do status dos últimos builds concluídos, exibe também qualquer build que esteja pendente ou em execução no momento. Cada item neste pequeno painel é um link que nos leva à página principal de um build específico, de onde podemos extrair mais informações.

A gestão de projetos em DevOps

Para que DevOps seja aplicado com sucesso, é necessário que haja uma boa gestão de todas as atividades envolvidas. Dentre todas elas, uma das mais importantes é, sem dúvida, o processo ao qual o projeto está submetido – e a palavra de ordem, nos últimos anos, tem sido a

Gestão de projetos e Integração Contínua com DevOps

adoção de variantes ágeis como o *Scrum* ou *Scrumban*. Entretanto, este trabalho só será efetivamente facilitado à medida que boas ferramentas sejam disponibilizadas, pois só assim será possível planejar e administrar atividades com qualidade, além de gerar e visualizar informações importantes acerca da saúde do projeto, ao longo de sua execução.

Analisaremos, nesta seção, duas ferramentas de mercado que atendem a este propósito, iniciando com um breve estudo sobre o Redmine e as possibilidades que esta plataforma nos dá.

Introdução ao Redmine

Redmine é uma aplicação web muito flexível e bastante popular no mercado. Distribuída sob a licença GPL, é uma so-

lução de código aberto que oferece uma interface web bastante amigável e simples de usar, apresentando-se como uma ótima plataforma para a gestão de projetos.

Algumas das principais funcionalidades do Redmine são:

- *Issue tracking* (e criação de issues, inclusive, por e-mail);
- Boa integração com sistemas de versão como Subversion, Git e Mercurial;
- *Wiki* e *fórum* individualizados por projeto; e
- Suporte a múltiplas bases de dados.

Assim como fizemos com Jenkins para a implementação de nosso servidor de integração contínua, evitaremos a instalação do Redmine em plataforma própria, optando por utilizar esta plataforma como

serviço, pela web. O provedor que adotamos, neste caso, é o *Hosted Redmine*.

Acessando o Hosted Redmine

O primeiro passo para criarmos nosso projeto no *Hosted Redmine* é criarmos a nossa conta de acesso a este provedor. O processo é bastante simples, bastando ao leitor que acesse a página principal da *Hosted Redmine* (veja na seção **Links**) e realize o seu cadastro. Com o cadastro já realizado e confirmado, devemos utilizar as credenciais que escolhemos para entrar no sistema. A primeira página que vemos quando entramos na *Hosted Redmine* é algo parecido com o que está ilustrado na **Figura 17**. No exemplo da imagem, podemos ver algumas issues já criadas e atribuídas a um membro do projeto. Estas são algumas tarefas que criamos, de antemão, para servir como pano de fundo para o material que abordaremos desta seção em diante. No caso de uma conta recém-criada, no entanto, esta página estaria vazia e ainda teríamos que criar o nosso projeto. Passaremos, a partir de agora, a descrever os passos necessários para isso.

Criando um projeto no Hosted Redmine

Para criar um projeto no *Hosted Redmine*, basta que cliquemos no item *Projects*, na barra de menu superior e, então, no link *New project*. A **Figura 18** nos mostra a disposição desses dois itens citados na página principal de nossa conta, indicados por setas azuis.

The screenshot shows a 'Build History' section with the following data:

#	Date
9	Dec 7, 2015 6:51 AM
8	Dec 7, 2015 6:47 AM
7	Dec 7, 2015 5:58 AM
6	Dec 5, 2015 10:08 PM
5	Dec 5, 2015 9:57 PM
4	Dec 5, 2015 9:39 PM
3	Dec 5, 2015 9:36 PM
2	Dec 5, 2015 9:34 PM
1	Dec 5, 2015 9:32 PM

At the bottom, there are links for 'RSS for all' and 'RSS for failures'.

Figura 16. Painel de status dos últimos builds realizados

The screenshot shows the 'My page' section with two main tables:

Issues assigned to me (3)

#	Project	Tracker	Subject
499548	devmedia_devops	User Story	Criar a representação de um mapa de cores para ser usado ... (New)
499546	devmedia_devops	Task	Criar a representação de um resistor (New)
499544	devmedia_devops	User Story	[SPRINT-1] Desenvolver primeira versão do modelo de dados (New)

Reported issues (3)

#	Project	Tracker	Subject
499548	devmedia_devops	User Story	Criar a representação de um mapa de cores para ser usado ... (New)
499546	devmedia_devops	Task	Criar a representação de um resistor (New)
499544	devmedia_devops	User Story	[SPRINT-1] Desenvolver primeira versão do modelo de dados (New)

Figura 17. Página inicial de uma conta na Hosted Redmine

A edição de um novo projeto se dá em uma página como a ilustrada na **Figura 19**. É nela que definimos todos os recursos desejados no projeto, tais como Wiki, Fórum, gerenciamento de issues, dentre inúmeros outros. Vemos também que todos os recursos suportados pelo Redmine estão organizados em duas subáreas: *Modules* e *Trackers*. Em nosso projeto de exemplo, incluímos todos os elementos selecionados nesta figura, embora boa parte deles não seja coberto nas seções subsequentes. Como efeito do espaço disponível no artigo, daremos foco exclusivo para apenas dois elementos, a saber:

- Issue tracking; e
- Repository.

Administrando requisitos com Redmine

A administração de requisitos é uma disciplina da Engenharia de Software que vem sofrendo mudanças profundas ao longo dos últimos anos, muito em consequência do surgimento das metodologias ágeis. O contexto atual, caracterizado por propostas mais leves e dinâmicas como o Scrum, vem ocasionando a substituição de alguns processos muito populares no passado, como UP e cascata (*waterfall*), e esta transição vem transformando, aos poucos, a forma com que projetos de software evoluem.

Toda esta transformação, analisada à luz do mapeamento de requisitos, exige que os sistemas de gestão de projetos se tornem capazes de atender às características e aos comportamentos introduzidos pelas novas práticas – uma história de usuário do *Scrum*, por exemplo, deve ser modelada de uma forma totalmente diferente da que são registrados requisitos em um projeto *waterfall*.

Se voltarmos à **Figura 19**, veremos que um dos itens que selecionamos abaixo da categoria *Trackers* é denominado *User story*. O *Redmine*, em sua versão padrão, trabalha com um modelo mais *tradicional* de gestão de requisitos, o que significa que encontraremos nesta ferramenta um estilo mais orientado a processos como UP ou waterfall quando formos editar os requisitos de nossos projetos. Entretanto, ao selecionar esta opção, trazemos para o nosso ambiente a possibilidade de tra-

The screenshot shows the Redmine interface with the title "Online Project Management :: Free Redmine Hosting". The top navigation bar includes "Home", "My page", "Projects", and "Help". On the right, it says "Logged in as pedrobrigido". Below the navigation is a search bar and a "PROJECTS" section with options like "View closed projects" and "Apply". The main content area is titled "Projects" and lists three projects: "TestCreditSystem" (status: bla bla bla), "Megonet Operating System" (status: The Megonet Operating System (MGOS) is an Open Source project to create an Operating system aimed at 32/64 Bit Intel/AMD Computers.), and "lookingformaps" (status: Web portal to index and allow user search online published maps: OGC WMS, KML, KMZ, Etc). At the bottom left, there's a "NOK" link. A blue arrow points to the "New project" button in the top right corner of the main content area.

Figura 18. Página de projetos de uma conta na Hosted Redmine

The screenshot shows the "New project" creation form. It has fields for "Name" (devmedia_devops), "Description" (Projeto criado para estudar os principais pontos a serem considerados quando estamos entrando no mundo de DevOps. Essas características e práticas vão desde atividades puramente associadas a desenvolvedores (DEV), outras relacionadas somente a operadores (OPS) e todo um outro conjunto de itens que precisam ser resolvidos de modo a encurtar o gap entre as duas áreas e dar mais dinamismo, velocidade e eficiência ao processo como um todo.), "Identifier" (devmedia_devops), "Homepage" (Public), "Inherit members" (checkbox), and "Project Type" (Personal Use). Below these, there are two sections: "Modules" and "Trackers". The "Modules" section contains checkboxes for Issue tracking, Time tracking, News, Documents, Files, Wiki, Repository, Calendar, Gantt, Charts, Feature, User Story, Bug, Module, and Support. The "Trackers" section contains checkboxes for Task, Deliverable, Tweak, and Step. A red box highlights the "Modules" and "Trackers" sections.

Figura 19. Detalhes das funcionalidades que podemos selecionar para o nosso projeto Redmine

lhar, também, com parte dos elementos introduzidos a partir das metodologias ágeis, como o já citado Scrum. Há, ainda, a possibilidade de estender os recursos da plataforma, a partir do uso de plug-ins como o *RedmineCRM*.

Para criar um novo requisito em nosso projeto, devemos clicar no item *New Issue*, disponível na barra de menu indicada

na **Figura 20**, com uma seta vermelha. Em seguida, um formulário será aberto e o novo item deverá ser criado. Na **Figura 21**, apresentamos o conteúdo que utilizamos para criar uma nova história de usuário, com o título *[SPRINT-1] Desenvolver a primeira versão do modelo de dados*. Definimos este identificador porque, embora não estejamos utilizando nenhum plug-in



Gestão de projetos e Integração Contínua com DevOps

The screenshot shows the 'Settings' page of a Redmine project. The top navigation bar includes links for Overview, Activity, Issues, New issue, Charts, Gantt, Calendar, News, Documents, Wiki, Files, Settings (which is highlighted in blue), and Delete Project. Below the navigation is a sub-menu with tabs for Information, Modules, Members, Versions, Issue categories, Wiki, Repositories (which is highlighted in blue), Forums, and Activities (time tracking). A large red arrow points upwards from the 'Settings' link in the main menu towards the 'Repositories' tab in the sub-menu. The main content area displays a message 'No data to display'.

Figura 20. Abas de um projeto no Redmine

The screenshot shows the 'New issue' creation form. It includes fields for Tracker (set to 'User Story'), Subject ('[SPRINT-1] Desenvolver primeira versão do modelo de dados'), and a large Description area with a rich text editor. The Description text discusses the role of a software architect and the requirements for a resistor model. Below the Description are fields for Status ('New'), Priority ('Normal'), Assignee ('Pedro Brigatto'), and various tracking details like Parent task, Start date, Due date, Estimated time, and % Done (0%). Two red arrows point from the text in the Description field to the 'Parent task' and 'Due date' fields.

Figura 21. Criação de um issue em nosso projeto Redmine

que suporte todos os elementos de uma metodologia ágil como Scrum, desejamos dividir o nosso projeto em ciclos – *sprints*. Por isso, estamos definindo um prefixo que indica exatamente em qual Sprint cada história de usuário deve ser alocada. Isto remete ao fato que apresentamos a alguns instantes, em que uma eventual ausência de recursos em uma plataforma não deve servir de motivo para que não organizemos os nossos projetos da forma que desejamos/precisamos.

Pela Figura 21, observamos que todo item que criamos pode, ainda em sua criação, ser associado a um membro do projeto. Fazemos isso selecionando um dos colaboradores listados no campo *Assignee*.

Ao concluir a edição desta história de usuário, o leitor poderá observar que o Redmine permite a criação de subitens a ela associada. Devemos fazer isto para, aos poucos, recheá-la com todas as tarefas planejadas para cumprir os objetivos da história. Por questões de espaço e, também, por ser uma atividade organizada de forma bastante intuitiva pela ferramenta, deixaremos esta atividade a cargo do leitor. O resultado final desta experiência deve ser algo bastante parecido com o que vemos ilustrado na Figura 22.

Neste caso, mostramos a mesma história de usuário já citada e exibida a partir da Figura 21, com duas tarefas a ela associadas (indicadas por setas vermelhas).

Associando o repositório de código ao Redmine

Além de criar e gerenciar itens – histórias de usuário, tarefas, dentre outros – o Redmine nos permite que associemos nossos repositórios de código a ele. É o que veremos, brevemente, nesta seção.

Veja, a partir da página inicial do nosso projeto – exibida na Figura 20 – que existe um item na barra de menu com o título *Settings*. Ao acessarmos este painel, outro menu é apresentado e, nele, há uma opção denominada *Repositories*. É aí que o Redmine nos permite apontar para os nossos repositórios. Logo que selecionamos este item no menu, o painel que se abre oferece uma opção chamada *New repository*. Quando clicamos neste item, veremos abrir uma tela como a exibida na Figura 23.

Um detalhe importante que o leitor perceberá quando estiver configurando seu repositório no Redmine é que o campo *SCM* oferece apenas *Subversion* como opção. No entanto, ignore esta informação, selecionando este valor e entrando, normalmente, com

os dados do seu repositório no GitHub. O resultado final será o seu repositório devidamente configurado e associado ao seu projeto. Na imagem ilustrada na **Figura 23**, exibimos a configuração do repositório criado para servir como base para o artigo. Perceba que, embora o tipo de sistema de versionamento selecionado tenha

sido Subversion (o único disponível), a URL é de um repositório Git, bem como as credenciais de acesso. Quando salvamos esta configuração, o Redmine realiza a vinculação entre as duas contas e, assim que a operação é concluída, o leitor poderá ver algo como o exibido na **Figura 24**.

Figura 22. Detalhes de uma história de usuário do Redmine

Figura 23. Associação de um repositório Git em uma conta na Hosted Redmine

Gestão de projetos e Integração Contínua com DevOps

Introdução ao Trello

Uma alternativa ao Redmine, mais orientada ao modelo introduzido pelas metodologias ágeis, é o Trello. Ao contrário do primeiro, esta ferramenta organiza requisitos de um projeto em torno do conceito de cartões. Além disso, permite que criemos listas diferentes para estruturar o fluxo do projeto, permitindo mover itens de uma lista para outra sempre que desejarmos. Um exemplo que ilustra essas características pode ser visto na **Figura 25**, onde temos três listas definidas:

- EM ABERTO, indicando as tarefas que ainda não foram iniciadas;
- EM ANDAMENTO, com as atividades que estão em progresso; e
- CONCLUÍDOS, com tudo aquilo que já foi finalizado na iteração atual do projeto.

No contexto oferecido pelo Trello (veja como acessar o serviço na seção **Links**), todo o trabalho de um projeto é organizado na forma de *Quadros* que, em um paralelo com processos ágeis como Scrum, pode ser comparado com uma *Sprint*. Portanto, assim como fizemos com o projeto Redmine criado em seções anteriores (a partir do uso de prefixos para identificar o Sprint em questão), podemos definir, no Trello, um quadro para cada Sprint consumido pelo projeto.

Todo quadro é composto por uma lista de itens, que correspondem às tarefas que precisam ser realizadas. E todo item, ao ser clicado, pode ser detalhado a partir de um campo de descrição e ser, também, associado a documentos anexos que ajudem a esclarecer seu propósito (documentos, imagens, dentre outros). Há, ainda, outras

características que podem ser editadas, como podemos ver a partir da **Figura 26**.

De certa forma, a gestão de projetos sempre estará associada aos seguintes elementos:

- **Requisitos:** mapeados como histórias de usuário ou em formatos mais tradicionais, são nada mais que tudo aquilo que precisa ser coberto no projeto; e
- **Membros:** lista de colaboradores que atuam no desenvolvimento do projeto.

A estrutura utilizada por cada sistema – e cada metodologia – pode até variar, mas o importante é garantir que o conteúdo seja claro e bem descrito, fornecendo as direções necessárias para que o trabalho seja bem executado.

DevOps, em linhas gerais, não exige que a sua metodologia de desenvolvimento seja ágil. No fundo, o que se espera é organização em torno da prática e das ferramentas, de tal modo que a eficiência do time seja a máxima possível, e a fluidez na comunicação e colaboração sejam potencializadas.

Integração contínua é essencial para que um projeto DevOps seja bem-sucedido. A partir desta prática, conseguimos eliminar do processo boa parte das atividades mais sensíveis a erros humanos e, ao mesmo tempo, inserimos no nosso dia a dia uma prática que nos permitirá avaliar a saúde de um projeto continuamente. Desta forma, tornamo-nos capazes de reagir muito mais prontamente a eventuais problemas que podem surgir ao longo de um ciclo de desenvolvimento.

Com o advento da computação em nuvem e o provisionamento facilitado – tanto no aspecto do processo quanto no aspecto do custo e segurança – de máquinas remotas, todo este potencial da integração contínua pode ser, inclusive, aproveitado fora de nossa infraestrutura, por meio de soluções oferecidas no modelo de PaaS (plataforma como serviço). É o caso da OpenShift, utilizada ao longo do artigo para avaliarmos os recursos do Jenkins.

Em DevOps, a importância de uma boa gestão de projetos é máxima, pois somente controle automatizado de atividades de desenvolvimento e operação não são

The screenshot shows a GitHub repository named 'devmedia_devops'. The main page displays a table of 'Latest revisions' with columns for #, Date, Author, and Comment. Below the table is a list of recent commits by 'pedro.brigatto' with details like date, author, and commit message. At the bottom, there are links for 'View differences' and 'View all revisions'.

Figura 24. Detalhes do repositório hospedado em uma conta GitHub, já vinculado à conta na Hosted Redmine

The screenshot shows a Trello board titled 'DevMedia_Devops'. It has three columns: 'EM ABERTO', 'EM ANDAMENTO', and 'CONCLUÍDOS'. Each column contains several cards with descriptions of tasks. A sidebar on the left lists 'EM ABERTO' tasks: 'Implementar testes unitários para mapa de cores', 'Adicionar cobertura do Sonar', 'Integrar gestão de projeto com repositório de código', and 'Adicionar um cartão...'. A sidebar on the right shows a list of cards under 'CONCLUÍDOS': 'Implementar lógica de cálculo de resistência', 'Implementar lógica de representação de mapa de cores de resistores', 'Criar repositório centralizado de código (Github)', 'Testar comunicação com repositório centralizado de código', 'Adicionar ferramenta de cobertura de código no Eclipse', 'Configurar Maven', 'Testar Maven localmente', 'Configurar Jenkins', and 'Testar integração Jenkins-Github'. A button 'Adicionar uma lista...' is visible at the top right of the board area.

Figura 25. Página de apresentação de um quadro do Trello

suficientes para garantir a qualidade de um projeto e a coesão de uma equipe. Logo, é fundamental que tenhamos um bom sistema em mãos, que nos permita traduzir requisitos e metodologias de trabalho em informação muito bem organizada, que possa ser consumida de forma absolutamente intuitiva por qualquer membro do projeto.

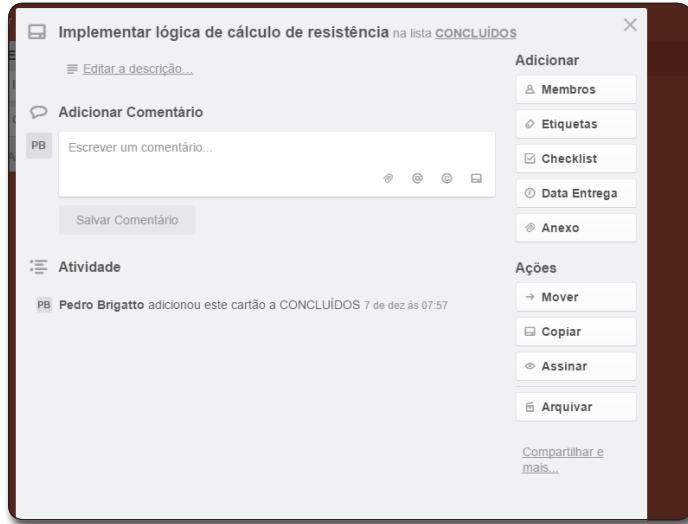


Figura 26. Edição de um item de uma lista no Trello

DevOps não se consegue sem um ambiente de alta sinergia entre as pessoas e clareza máxima de objetivos, e garantir que essas características estejam presentes é o maior papel que sistemas de gestão de software devem desempenhar.

Mais uma vez, o uso de computação em nuvem e PaaS figura como um grande facilitador, com soluções de alta qualidade e segurança disponíveis a um preço bastante acessível e fora de nossas dependências físicas. Foi o que ilustramos, ao longo do texto, com o uso da *Hosted Redmine*.

Autor



Pedro E. Cunha Brigatto

pdrobrigatto@gmail.com

Engenheiro da Computação graduado pela Universidade Federal de São Carlos, desenvolvedor certificado SAP Netweaver (Java Stack) e programador certificado SCJP. Especialista em Engenharia de Software graduado pela Unimep e pós-graduado em Administração pela Fundação BI-FGV, atua com desenvolvimento de software desde 2005. Atualmente atua como consultor técnico no desenvolvimento de soluções de alta disponibilidade na Avaya.



Links:

Código do projeto tema no GitHub.

https://github.com/pdrobrigatto/devmedia_devops_series

Correção para o problema relacionado ao serviço de download do Jenkins.

<https://issues.jenkins-ci.org/browse/JENKINS-26780>

Trabalhando com hooks no Git.

<https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks>

Questão relacionada a permissões de acesso ao Jenkins.

<http://tinyurl.com/oh2w3k3>

Solução Redmine como serviço.

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

Problema relacionado ao agente Maven no Jenkins.

<https://forums.openshift.com/jenkins-failed-to-build-maven-project>

Página oficial do Trello.

<https://trello.com/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Web design responsivo com Bootstrap e JSF 2

Veja neste artigo como viabilizar um design responsivo em aplicações web com Bootstrap e JSF 2

ESTE ARTIGO É DO TIPO MENTORING

SÁIBA MAIS: WWW.DEVMEDIA.COM.BR/MENTORING-SAIAMAI

Segundo relatório do Google Analytics, do total de acessos ao seu sistema de busca, 60% são feitos através de telefones celulares. Demonstrando essa mesma tendência, em um levantamento realizado pelo IBGE (Instituto Brasileiro de Geografia e Estatística) em 2013, verificou-se que 57,3% das residências no Brasil acessam a internet através de dispositivos móveis. Foi observado, também pelo IBGE, que nas navegações pela rede somente por esses equipamentos, houve, em 2014, um acréscimo de 7,2 milhões de usuários se comparado ao ano anterior, e tudo indica que em 2015 esse percentual deverá aumentar.

Além disso, dados divulgados pela FGV (Fundação Getúlio Vargas) em 2015 apontaram a existência de 306 milhões de dispositivos conectados à internet no Brasil. A maioria deles, cerca de 154 milhões, são smartphones, ou seja, temos uma média três dispositivos para cada dois habitantes. Somente no segmento de celulares, também chamado de *feature phone*, a penetração desses equipamentos no mercado brasileiro foi de 162%, ou seja, 1,62% aparelho por habitante. Esse índice supera a média mundial, que é de 117%, e até o percentual verificado nos Estados Unidos, de 148%.

Dessa forma, verificamos que esses novos dispositivos chegaram para ficar e tudo indica que por causa deles os computadores (Desktops e Notebooks) se tornarão obsoletos no sentido que os conhecemos hoje, isto é, para o acesso à internet, leitura de documentos ou a execução de aplicativos. Tal fato pode ser observado no dia a dia, quando notamos que em diversos lugares nos deparamos com pessoas com seus smartphones e tablets realizando ações que até pouco tempo eram disponíveis apenas aos computadores.

Cenário

Neste artigo serão apresentados os conceitos relacionados a Design Responsivo e como utilizá-los em uma aplicação web. Para isso, analisaremos o Bootstrap e o JSF e como adotá-los em conjunto para alcançar esta nova meta de grande parte das aplicações atuais. Nesse contexto, veremos também como configurar o ambiente de desenvolvimento para a construção de um projeto simples com interface responsiva voltada para o cadastro de alunos, pois com a grande quantidade de dispositivos móveis, a capacidade de ajuste dos elementos de um site ao tamanho das telas desses equipamentos torna-se cada vez mais importante.

Diante desse cenário surge a necessidade de desenvolver sites que sejam capazes de atender às características desses novos equipamentos quanto ao tamanho da tela e à forma como o conteúdo deve ser apresentado ao visitante. É certo que a forma como as informações são apresentadas a quem utiliza um smartphone deve ser diferente daquela apresentada a quem utiliza um computador, visto que são equipamentos completamente diferentes quanto ao tamanho e resolução da tela, modo de interação e capacidade de processamento. Por exemplo, determinado conteúdo pode ficar pequeno e centralizado quando visualizado com um computador ou ficar muito grande e descentralizado quando visualizado por meio de um smartphone. Além disso, a forma que a informação é exibida para o visitante pode ser crucial para o sucesso do site, pois uma navegação mais confortável pode significar a permanência ou retorno ao mesmo. Por exemplo, uma boa navegação de consumidores em uma loja virtual pode significar uma compra e a aceitação da loja pelo cliente para compras futuras.

Portanto, o desenvolvimento de sites deve possuir uma abordagem bem elaborada em sua concepção e codificação, de modo a garantir aos visitantes a capacidade de terem uma experiência visualmente rica e funcional. Com base nisso, surgiu a necessidade de uma web que suporte esses novos dispositivos, pois o cenário dominante até então, onde há um navegador instalado em um computador controlado por mou-

se e teclado em uma tela confortável, não atende mais à nova realidade de telas pequenas, sensíveis ao toque, redes móveis e muitas outras diferenças.

Assim, surge uma nova maneira de pensar a web, a qual visa a adaptação das páginas a todo tipo de dispositivo, abandonando as telas de tamanhos previsíveis e pensando em sites flexíveis, que suportem os mais diferentes tipos de interação e ferramentas de acesso. Nesse contexto, passa a ter destaque o Design Responsivo, a “chave” para essa nova web e solução que define um conjunto de técnicas com o intuito de melhorar a experiência do usuário na navegação independentemente do dispositivo utilizado.

Design Responsivo

Diante do crescimento do número de dispositivos móveis através dos quais a maioria dos acessos à internet são realizados, é natural que todas as empresas começem a direcionar seus esforços para atender a esse novo ambiente da melhor maneira. Contudo, devido à variação de tamanho e qualidade das telas, não seria adequado o desenvolvimento de múltiplas versões de um mesmo site. Sendo assim, uma das soluções seria a programação do conteúdo de forma que os elementos nele existentes, tais como imagens, textos, layout, entre outros, pudessem se adaptar automaticamente a essas variáveis. Nesse contexto, inspirado por conceitos de arquitetura e no livro *Interactive Architecture*, de Michael Fox e Miles Kemp, Ethan Marcotte publicou em meados de 2010, no site *A LIST APART*, um artigo intitulado “*Responsive Web Design*”, onde mostrava as técnicas que poderiam garantir uma correta visualização de páginas independente do dispositivo utilizado.

Devido à grande e positiva repercussão e o grande número de questionamentos da comunidade por mais explicações sobre o tema, no ano seguinte foi lançado um livro –também escrito por Ethan – que recebeu o mesmo nome do artigo e que mudou, definitivamente, a forma de fazer design para a web. Antes disso, a maior parte das pessoas que trabalhavam com conteúdo para web construía seus sites criando uma divisão principal de largura fixa que funcionava como um contêiner, sendo todo conteúdo colocado dentro dele. Assim, tal abordagem permitia o controle do design criado, pois limitava-o pela divisão fixa definida e impedia a expansão desordenada da página pelo monitor. Como a maior parte dos computadores possuía a resolução 1024x768, foi padronizado 960 pixels como a largura ideal para um site. Entretanto, tal abordagem começou a gerar problemas quando um tipo único de equipamentos que acessava a internet deu lugar a uma rica variedade: os tablets, smartphones e até TVs. Nesse momento, a divisão principal de largura fixa ficaria pequena para uma televisão de 42 polegadas, mas muito grande para telas com resolução menor, sinalizando que a adoção de um valor fixo não é mais adequada.

A solução proposta por Ethan Marcotte é criar um design que permita ao site comportar-se de maneira diferente de acordo com o dispositivo, ajustando todo o conteúdo conforme o tamanho e resolução da tela. Para isso, Ethan utilizou três ingredientes técnicos: *fluid grids* (layout fluido), *flexible images* (imagens e recursos

flexíveis) e *media queries*. Vejamos uma análise sobre cada um deles nos tópicos a seguir.

Layout Fluido

Layouts fluídos são conhecidos também como grids flexíveis e têm como principal característica a proibição do uso de medidas absolutas ou fixas no CSS da página, tais como a definição de pixels ou pontos. Assim, quando especificamos tamanhos de espaçamentos, paddings, margens, enfim, qualquer medida para os elementos em uma página, devemos sempre especificar valores variáveis ou relativos, como *porcentagens* ou *ems*. Os tipos de medidas para CSS são:

- **Pixel (px):** a unidade pixel é uma unidade de medida fixa e geralmente representa a menor parte de uma imagem digital à qual é possível atribuir uma cor. O número deles em uma imagem determina sua resolução. Da mesma forma, um pixel na tela de um dispositivo é a menor unidade endereçada nela, e a resolução é representada pela quantidade de pixels que cada dimensão pode exibir. Por exemplo: uma tela de um computador com resolução 1366x768 tem a capacidade de exibir 1366 pixels na largura por 768 pixels no comprimento, totalizando 1.049.088 pixels, ou seja, mais de um 1 megapixel. O pixel é a unidade de medida fixa mais usada em arquivos CSS para organizar os elementos na tela;
- **Ponto (pt):** a unidade ponto é uma unidade de medida fixa tradicionalmente utilizada para declaração do tamanho da fonte. É empregada em CSS geralmente para impressão de qualquer tipo de documento em papel, tais como jornais, revistas, outdoors, entre outros. Por exemplo, o CSS `h2 {font-size: 12pt;}` definiria um cabeçalho com tamanho de fonte igual a 12 pontos (12pt);
- **Porcentagem (%):** a unidade porcentagem é uma unidade de medida variável ou relativa e é parecida com a unidade “*em*” (analisada a seguir), embora apresente algumas diferenças fundamentais que não serão apresentadas aqui por entender que



este artigo não carece de tal aprofundamento. Com porcentagem, para representar o tamanho da fonte correspondente ao padrão 12pt, utiliza-se 100%, e o uso dessa medida permite ao texto permanecer totalmente escalável para diferentes tamanhos de tela. Outra questão a ser observada é que quando a medida *porcentagem* é utilizada em um atributo *font-size*, o tamanho do texto estará relacionado ao *font-size* padrão do dispositivo, que geralmente é 12px; e quando utilizado para outras medidas, o tamanho do elemento estará relacionado à largura do elemento pai. Por exemplo, o CSS `h3{font-size: 100%;}` definiria o texto do cabeçalho com um tamanho de fonte igual 12px (100% do *font-size* padrão) e o CSS `nav{width: 25%;}` definiria o tamanho da barra de navegação como sendo 25% do tamanho do seu elemento pai;

- **Ems (em):** a unidade *em* é uma unidade de medida variável ou relativa que nasceu na tipografia e também é chamada de quadratim. Na tipografia, 1em corresponde ao tamanho da letra M em caixa alta e era utilizado pelos tipógrafos como medida de referência para outras letras. Em CSS, 1em corresponde a 16px e o uso desta unidade de medida em texto torna-o também escalável para tamanhos de tela diferentes. Além disso, ele sempre estará relacionado ao tamanho do *font-size* do equipamento quando utilizado para determinar o tamanho do texto ou o tamanho de outro elemento HTML.

Como visto, as unidades de medida *Pixel* e *Ponto* representam unidades fixas, enquanto *Porcentagem* e *Ems* representam unidades variáveis, as quais permitem escalabilidade e adaptação de tamanho a um elemento em relação aos demais em um documento. Assim, fica evidente que a utilização de *px* e *pt* não é adequada para a construção de designs responsivos, uma vez que a escalabilidade e a adaptação dos elementos ficariam prejudicadas. Essas medidas existentes nos CSS das páginas devem ser convertidas para medidas variáveis ou relativas utilizando a relação apresentada a seguir:

$$1\text{em} = 12\text{pt} = 16\text{px} = 100\%$$



Nesse momento o leitor pode estar se perguntando: "Diante das duas unidades de medida variável, qual devo usar ao constatar que tanto *em* quanto *porcentagem*, teoricamente, nos traria o mesmo resultado? A resposta é: depende do objetivo. Após muitos testes e experiência com o desenvolvimento de sites responsivos, os desenvolvedores verificaram que apesar das semelhanças, o uso de *porcentagem* se apresentou mais adequado para lidar com o tamanho do layout (larguras, margens, espaçamentos, entre outros) e o uso de *em* se apresentou mais adequado para uso ao lidar com o tamanho das fontes do texto. Deste modo, para o desenvolvimento de sites que tenham como objetivo contar com um design responsável, o consenso é que se adote para o tamanho dos elementos as unidades de medida variáveis ou relativas, sendo *em* para definição dos tamanhos de fontes de textos e % para o tamanho do layout (larguras, margens, espaçamentos, entre outros).

Imagens e recursos flexíveis

Por mais que sejam definidas a altura e largura das imagens de um site, elas sempre apresentarão problemas quando visualizadas em diferentes dispositivos e resoluções, pois o tamanho em pixels adotado pode resultar em imagens pequenas para computadores com alta resolução e grandes para dispositivos móveis com baixa resolução. Portanto, podemos optar pela medida variável *porcentagem* para lidar com essas imagens. No entanto, ainda assim é preciso ter cuidado, pois se aumentarmos a altura e largura originais, podemos deixar o layout "pixelizado" e carregar uma imagem com tamanho original de alta resolução e depois reduzi-la para baixa resolução pode deixar a página lenta para os casos onde há limitação de banda. Desse modo, o problema da visualização de imagens apresenta-se de forma a exigir muita atenção durante o desenvolvimento, principalmente porque para o sucesso do projeto hoje em dia devemos considerar que o site pode ser acessado por diversos equipamentos.

Dito isso, um layout que seja realmente fluido precisa de imagens que se adaptem a todo tipo de resolução, seja a de um computador ou de um dispositivo móvel. Embora não haja uma solução padronizada para tratar essa questão, já existem várias técnicas ou conjuntos de técnicas que podem ser adotadas para um ambiente responsável de modo a superar esse problema. A seguir citaremos algumas delas, e como verificaremos, cada uma possui suas particularidades, envolvendo desde soluções server-side a front-end. Ao final, lembre-se: a escolha de uma delas dependerá da real necessidade de cada projeto. As técnicas mais comuns são:

- Responsive images (*Filament Group*);
- Adaptive Images;
- HiSRC;
- Picturefill;
- Responsive Enhance;
- Sencha.IO;
- rwdImages;
- Foresight.js.

Ademais, caso não deseje utilizar nenhuma das técnicas apresentadas, seria uma solução muito boa a criação de imagens com tamanhos diferentes para intervalos de resoluções. Por exemplo:

- Servir a imagem 1 para resoluções até 320px;
- Servir a imagem 2 para resoluções maiores que 1024px e menores ou igual a 1600px;
- Servir a imagem 3 para resoluções maiores que 1600px.

Dessa forma, cada imagem seria exibida na página de acordo com a resolução do equipamento visualizador, respeitando os intervalos apresentados. Com isso, melhoraremos significativamente a percepção do usuário nos três casos apresentados, uma vez que faríamos uso das imagens mais adequadas conforme o tamanho da tela.

Relacionado a isso, atualmente existe uma discussão no grupo *Responsive Images Workgroup*, na W3C, que estuda incluir nativamente recursos que possibilitem às imagens se ajustar conforme o tamanho do layout da página nas próximas versões do HTML.

Agora o leitor pode estar se perguntando: Como servir imagens diferentes para cada resolução identificada? A resposta é simples: usando Media Queries, tópico que veremos a seguir.

Media Queries e Media Types

Antes de começarmos a falar sobre *Media Queries*, no entanto, iremos abordar os conceitos referentes a *Media Type*, pois os dois são totalmente relacionados. *Media Type* define basicamente qual CSS será aplicado à página para um dado tipo de equipamento. Elas podem ser declaradas em arquivos CSS ou diretamente na própria página, usando o atributo **media**, da tag `<link/>`. Desse modo, podemos especificar uma apresentação diferenciada através de CSS para cada dispositivo, seja ele um computador, uma TV, uma impressora ou um smartphone, renderizando a página web conforme a definição mais apropriada para o equipamento em questão. A identificação do dispositivo se dá utilizando o atributo **media** conforme mencionado, o qual pode receber os seguintes valores:

- **All**: utilizada para todos os dispositivos;
- **Braille**: para dispositivos táteis;
- **Embossed**: quando se trata de dispositivos que imprimem em braile;
- **Handheld**: usada para dispositivos portáteis com telas pequenas e banda limitada;
- **Print**: quando se deseja imprimir em papel;
- **Projection**: destinado a apresentações como PPS;
- **Screen**: utilizado principalmente para monitores coloridos de computadores;
- **Speech**: utilizada para sintetizadores de voz;
- **Tty**: utilizada para dispositivos que possuem uma grade fixa para exibição de caracteres, tais como: teletypes, terminais e também dispositivos portáteis com display limitados;
- **Tv**: utilizada para dispositivos como televisores, ou seja, com baixa resolução, quantidade de cores e scroll limitados.

A partir disso, quando desejarmos definir uma media type para impressão de um documento, poderíamos definir a tag `<link/>`, presente na tag `<header/>` da página, da seguinte maneira:

```
<link rel="stylesheet" type="text/css" media="print" href="print_style.css">
```

Assim, as configurações da folha de estilo `print_style.css` serão aplicadas sempre que uma solicitação de impressão da página for executada. Podemos, então, definir o valor do atributo **media** conforme a necessidade e a aplicação da folha de estilo dependerá dos dispositivos nele definidos. Ou seja, pode-se especificar mais de uma **media** para a mesma tag `<link/>`, separando os valores por vírgula, conforme apresentado a seguir:

```
<link rel="stylesheet" type="text/css" media="print, handheld" href="style.css">
```

Deste modo, tal configuração fará com que a folha de estilo `style.css` seja executada sempre que o dispositivo for um *handheld* ou se desejar imprimir a página. No entanto, diante da alta qualidade da tela dos dispositivos atuais, onde as páginas são renderizadas pelos navegadores com a mesma qualidade da tela dos computadores tradicionais, não faz muito sentido trabalharmos com layouts em CSS baseados unicamente em *media types*, uma vez que um **media handheld** comporta-se como um **screen**, mesmo não sendo um **screen** e vice-versa. Portanto, não devemos definir um CSS somente para **handheld**, nem um CSS totalmente **screen**, pois apesar da renderização semelhante, o comportamento do usuário e a forma de navegação são diferentes. É neste ponto que faz-se necessário o uso de *Media Queries*, recurso que possibilita tratar os dispositivos com hardwares parecidos de modo a separá-los não apenas pelo seu tipo (**screen**, **handheld**, etc.), mas também pelo tamanho da tela onde o conteúdo será exibido. Logo, o uso de *Media Queries* define condições específicas para um CSS e caso todas as condições estabelecidas no atributo **media** forem aprovadas, o mesmo será aplicado. Por exemplo, podemos especificar um CSS para dispositivos que possuam tela com até 320px da seguinte maneira:

```
<link rel="stylesheet" media="screen and (max-width: 320px)" href="device_320.css">
```

Como podemos observar, as *media queries* são aplicadas dentro do mesmo atributo **media** da tag `<link/>`, definindo qual condição deve ser satisfeita para aplicação do CSS. No caso do último exemplo, a folha de estilo `device_320.css` será aplicada para todos os computadores (**screen**) que possuírem uma tela menor ou igual a 320px (**max-width: 320px**). Assim, além de definir a *media type*, podemos especificar quais características dela devem ser consideradas na avaliação, conforme demonstrado com o uso de **max-width**.

Neste momento é válido informar que existem várias características que podem ser utilizadas para cada *media type*, tais com **width**, **height**, **device-width**, **device-height**, entre outras, e a maioria delas pode ser prefixada com **min** ou **max**.

Além disso, você pode compor complexas *Media Queries* através da combinação delas utilizando operadores lógicos. Os operadores existentes são:

- **or** (representado pela vírgula): é usado para combinar várias características *media* dentro de uma única media query, requerendo que pelo menos uma condição seja verdadeira para o retorno ser verdadeiro;
- **and**: é usado para combinar várias características *media* dentro de uma única media query, mas requer que todas as condições sejam verdadeiras para o retorno ser verdadeiro;
- **not**: é usado para negar o resultado da avaliação;
- **only**: é usado para prevenir que navegadores antigos, que não dão suporte a certas funcionalidades, apliquem as folhas de estilos selecionadas. Além disso, tal operador pode ser utilizado para restringir o CSS a algum tipo de *media*. Por exemplo, o código `<link rel="stylesheet" media="only screen" href="only.css" />` restringiria o arquivo *only.css* apenas para dispositivos *screen*.

Para mais informações sobre os tipos de *media type* e como criar *Media Queries*, veja a documentação no site da W3C.

Portanto, como preconizava *Ethan*, se um projeto seguir todos os três ingredientes técnicos apresentados (*Fluid Grids*, *Flexible Images* e *Media Queries*) para construção do design do site, ele será considerado responsivo e se comportará de maneira diferente de acordo com o tamanho de tela e resolução identificadas em cada dispositivo. Entretanto, a elaboração de um projeto responsivo desde o início pode demandar um tempo considerável na criação de componentes, layout e folhas de estilos necessários para a aplicação. Para amenizar esse problema é indicada a utilização de um dos princípios da Engenharia de Software, o reuso. Através da reutilização obtém-se aumento da qualidade (devido ao reaproveitamento de artefatos já testados) e a óbvia redução do esforço no desenvolvimento. Por isso, existem frameworks que implementam o design responsivo e fornecem toda a base já construída, de onde a implementação de um site pode partir. O *Skeleton*, o *Foundation* e o *Bootstrap*, que inclusive é um dos objetos de estudo desse artigo, são bons exemplos.

Bootstrap

O *Bootstrap* é um framework front-end para desenvolvimento de websites que contém um conjunto de ferramentas que auxiliam na criação de códigos HTML, CSS e JavaScript. Criado em 2010 pelos engenheiros do Twitter *Mark Otto* e *Jacob Thornton* como uma ferramenta interna da empresa – inicialmente chamado de *Twitter Blueprint* –, foi publicado no GitHub em 2011 como uma ferramenta open source. A partir de então o projeto obteve a contribuição de milhares de desenvolvedores na comunidade, tornando-se

leve, intuitivo e um dos frameworks front-end mais populares do mercado, disponibilizando um conjunto de componentes úteis à grande maioria dos sites atuais, como grid, tabelas, formulários, botões, dropdowns, navigation, pagination, carousel, entre outros, os quais permitem o desenvolvimento de forma rápida e fácil. Outro diferencial do *Bootstrap* é que o solidifica entre as principais opções é seu conjunto de características, que analisaremos a seguir:

1. Abordagem Mobile First: o *Bootstrap* se baseia no princípio que devemos começar o desenvolvimento e planejamento de um projeto web, desde um pequeno site até um grande sistema, pensando primeiramente nos dispositivos móveis e somente depois nos computadores;
2. Suporte para todos os navegadores populares: o framework dá suporte para todos os navegadores populares do mercado. Para os navegadores nas versões mais antigas a apresentação da página pode demonstrar pequenas variações quando os componentes são renderizados, mas sem perder a funcionalidade;
3. Fácil aprendizagem: para iniciar no aprendizado sobre o framework é necessário ter conhecimento apenas de HTML e CSS. Além disso, a ferramenta possui uma boa documentação em seu site oficial;
4. Design responsivo: a implementação de seu CSS ajusta os componentes do layout a todos os tamanhos de tela dos dispositivos atuais, tais como desktop, tablets e smartphones.

Bootstrap Grid System

Para a organização dos elementos das páginas na tela do equipamento visualizador o *Bootstrap* utiliza um sistema de grids (grades) que funciona como uma tabela abstrata, onde esses elementos são colocados e organizados conforme variações detectadas de acordo com o tamanho/orientação da tela. Essa tabela é composta por 12 colunas, as quais podem ser agrupadas para criar colunas mais largas e ajustadas ao tamanho da tela dos dispositivos através da utilização de classes CSS predefinidas no framework. Por outro lado, não existe um número fixo de linhas, podendo o desenvolvedor utilizar quantas forem necessárias. Por exemplo, podemos ter um layout com quatro linhas e um diferente número de colunas em cada uma, como: doze de tamanho unitário na primeira, três de tamanho quatro na segunda, duas de tamanho seis na terceira e uma de tamanho doze na quarta, conforme expõe a **Figura 1**.

Uma vez definida a quantidade de linhas e colunas, verificamos que a interseção entre elas forma um grid retangular onde podemos colocar os elementos HTML que farão parte do conteúdo do website. Tais elementos são posicionados dentro do grid através de classes CSS.

Dessa forma, para que o *Bootstrap Grid System* organize esses elementos, ele faz uso de três estruturas: duas já mencionadas (linhas e colunas) e uma nova, o container. Este último é representado pela tag HTML `<div>` com a classe CSS `.container` e é responsável por prover a largura e alinhamento do layout na página, bem como conter as linhas e colunas criadas. Por exemplo:

1	2	3	4	5	6	7	8	9	10	11	12
4		4				4					
6			6								
12											

Figura 1. Exemplo de um layout Grid System

```
<div class="container">  
...  
</div>
```

Já as linhas definem as divisões horizontais do layout e são criadas através do elemento `<div>` dentro do container com a classe CSS `.row`, como verificado na **Listagem 1**.

Listagem 1. Exemplo de criação de uma linha.

```
<div class="container">  
  <div class="row">  
    ...  
  </div>  
  <div class="row">  
    ...  
  </div>  
  ...  
</div>
```

Em seguida, podemos criar as colunas dentro do bloco que representa a linha declarando a tag `<div>` com uma das classes CSS para colunas disponíveis no framework. Assim, as colunas representariam divisões verticais do layout e teriam classes CSS definidas conforme o tamanho da tela de modo a criar layouts dinâmicos e flexíveis. Os nomes dessas classes são identificados através dos prefixos de cada tipo de equipamento e podem ser utilizados sozinhos ou combinados, conforme a necessidade do projeto. São eles:

- `.col-xs-*`: utilizado para smartphones que possuem tela menor do que 768px. O `xs` significa Extra Small;
- `.col-sm-*`: utilizado para tablets que possuem tela maior ou igual a 768px. O `sm` significa Small Devices;
- `.col-md-*`: utilizado para desktops que possuem tela maior ou igual a 992px. O `md` significa Medium Devices;
- `.col-lg-*`: utilizado para desktops que possuem tela maior ou igual a 1200px. O `lg` significa Large Devices.

Como podemos observar, no final de cada prefixo existe o símbolo `"*"`, o qual representa um número inteiro entre 1 e 12, pois como vimos a tabela abstrata é composta por 12 colunas. Assim, quando definimos `<div class="col-xs-6"></div>`, estamos dizendo que este CSS será aplicado para um dispositivo *Extra Small* (`xs`) e que a coluna terá tamanho 6 nesta tela. Portanto, o número no final do nome da classe representa o tamanho que cada coluna deve ter de acordo com a tela do dispositivo visualizador. Observe o exemplo da **Listagem 2**.

Como foi possível verificar, a utilização do Grid System Bootstrap permite criar tabelas abstratas em páginas de modo a ajustar os elementos do layout de acordo com o tipo de dispositivo visualizador. Esse processo é feito a partir de classes CSS nas tags `<div>` representantes do container, linhas e colunas. Para mais informações sobre o Bootstrap, veja a documentação no site oficial do framework.

Listagem 2. Exemplo de criação de colunas.

```
<div class="container">  
  <div class="row">  
    <div class="col-md-3 ...> </div>  
    ...  
  </div>  
  <div class="row">  
    <div class="col-md-6 ...> </div>  
    ...  
  </div>  
  <div class="row">  
    <div class="col-md-3 ...> </div>  
    ...  
  </div>  
  ...  
</div>
```

JavaServer Faces

O JSF é um framework MVC para sistemas web baseado em componentes que auxilia bastante a construção de aplicações desse tipo com a tecnologia Java. Esses componentes são colocados em formulários de páginas e ligados a objetos Java de forma a prover suporte a várias características presentes no framework, tais como: mecanismos de manipulação de eventos, validações de informações enviadas pelos clientes, navegação entre páginas, entre outras. Assim, com essa tecnologia torna-se mais fácil a construção de interfaces com o usuário pelo fato de o desenvolvedor poder implementar as aplicações simplesmente utilizando componentes visuais reusáveis.

Devido à existência destes componentes na interface, toda página JSF é um documento XML com namespaces, recurso utilizado para importar e possibilitar o uso de componentes HTML do JSF. Alguns exemplos de namespaces são: `xmlns:h`, `xmlns:f` e `xmlns:ui`.

A partir da declaração destes os componentes são utilizados na interface com o usuário e ligados a objetos Java – conhecidos como JavaBeans – através da utilização da tag `<h:form>` e dos métodos `get` e `set` das classes. Dessa forma, os componentes de UI que serão vinculados às propriedades de um JavaBean devem, necessariamente, fazer parte de um formulário JSF, pois é através dele que o JSF envia e recebe dados do servidor. A seguir é apresentada a declaração da tag `<h:form>`:

```
<h:form id="form1">  
  <!-- componentes JSF de interface com o usuário devem ser colocados aqui -->  
</h:form>
```

No lado do servidor, cada JavaBean que tenha atributo vinculado a alguma página deve possuir a anotação `@ManagedBean` sobre o nome da classe, para que seja gerenciado pelo framework JSF e permita a ligação entre as regras de negócio da aplicação e a camada de visualização. Essa anotação possui ainda um atributo chamado `name`, não obrigatório, que representa o nome dado ao bean gerenciado. Ademais, para definir o ciclo de vida de cada

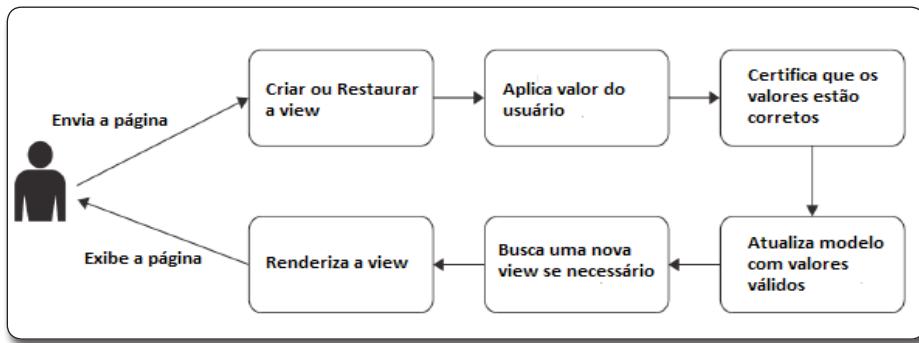


Figura 2. Ciclo de vida de uma requisição JSF

um dos beans conforme a necessidade do projeto, é possível especificar o seu escopo fazendo uso de uma das anotações analisadas a seguir:

- **@ApplicationScoped:** especifica que existirá somente uma instância da classe em toda a aplicação, ou seja, todos os usuários terão acesso à mesma instância;
- **@SessionScoped:** garante a existência de uma instância para cada usuário que utiliza a aplicação;
- **@ViewScoped:** relaciona a instância do bean gerenciado à página. Assim, enquanto essa página estiver sendo manipulada pelo usuário, o bean estará disponível;
- **@RequestScoped:** garante que existirá uma instância pelo tempo de duração da requisição feita pelo usuário;
- **@NoneScoped:** cada vez que houver uma chamada a uma propriedade ou método da classe bean, será criada uma nova instância. Neste caso a instância não terá ligação com qualquer escopo da aplicação;
- **@CustomScoped:** essa anotação permite a criação de escopos personalizados. Para isso, a instância de uma classe bean é armazenada em uma interface do tipo `java.util.Map<K,V>` e disponibilizada para uma página ou um conjunto de páginas da aplicação.

Ciclo de vida de uma requisição JSF

O ciclo de vida de uma requisição JSF compreende um conjunto de etapas realizadas pelo framework para a criação/restauração e exibição das páginas ao usuário. Essas etapas são responsáveis por capturar os dados enviados pela UI, verificar a validade dos mesmos, aplicar esses dados às regras de negócio e exibir o resultado do processamento. Assim, o entendimento deste ciclo é de extrema importância para o desenvolvedor que precisa lidar com este framework. A Figura 2 representa as etapas de uma requisição JSF.

Baseando-se no fluxo mostrado, podemos descrever as ações realizadas em cada fase da seguinte maneira:

- **RESTORE_VIEW (Criar ou Restaurar a view):** é a primeira fase do JSF e visa criar ou restaurar (no caso de uma página já requisitada) a árvore de componentes de uma página. Caso seja a primeira requisição à página, o framework cria uma nova árvore de componentes e associa a ela e caso não seja a primeira requisição, o framework restaura a árvore de componente já existente;

- **APPLY_REQUEST_VALUES (Aplica valores do usuário):** nesta fase são atribuídos os valores que foram enviados na requisição para seus respectivos componentes da árvore de componentes. Como os dados coletados no formulário HTML são transferidos na requisição como `String`, o JSF também armazena essas informações na árvore de componentes como `String`;
- **PROCESS_VALIDATIONS (Certifica que os valores estão corretos):** depois de completada a fase anterior, é iniciada a fase de processar conversões e validações como, por exemplo, validar uma formatação de uma data ou até mesmo verificar se determinado campo está sendo informado. Assim, caso ocorra algum erro, as informações são armazenadas no FacesContext e o fluxo é direcionado para a fase *Renderiza a view* para construir a resposta. Caso contrário, o fluxo segue para a próxima fase;
- **UPDATE_MODEL_VALUES (Atualiza o modelo com valores válidos):** se tudo ocorreu bem na fase anterior, o JSF já tem certeza que pode atualizar os valores dos atributos do bean gerenciado com os valores contidos na árvore de componentes através da utilização dos métodos `get` e `set`;
- **INVOKE_APPLICATION (Busca uma nova view se necessário):** depois de passar por todas as fases anteriores, é nesta fase que serão tratados as ações e os eventos da lógica de negócios da aplicação. É aqui que o desenvolvedor vai determinar o que será feito com o que foi aplicado ao modelo. Assim, podemos determinar nesta fase qual será a próxima página que será exibida para o usuário;
- **RENDER_RESPONSE (Renderiza a view):** finalmente o ciclo de vida chega ao fim. Nesta fase será feita a codificação da árvore de componentes, transformando-a em HTML (+ CSS e JS) e exibindo-a no navegador do cliente.

Navegação utilizando JSF

Para viabilizar a navegação entre páginas o JSF estabelece diferentes opções que especificam como as mudanças entre elas devem ocorrer de modo a definir o conteúdo que será visualizado pelo usuário conforme as regras de negócio no sistema. Basicamente existem três formas de defini-la, a saber:

1. **Navegação implícita e dinâmica:** quando se adota esse tipo de navegação, o retorno do método associado a uma ação na janela, como o clique em um botão, define a próxima página a ser apresentada para o cliente. Esse método deve estar presente em um *bean* gerenciado e retornar um objeto do tipo `String` para especificar a página de destino. Caso essa `String` seja `null` o JSF manterá a aplicação na própria página. Por exemplo:

```
public String retorno() {  
    return "index";  
}
```

Assim, um *bean* gerenciado com o método `retorno()` irá direcionar a aplicação para a página `index.xhtml`;

2. Navegação Estática: neste tipo de navegação o componente JSF define a nova página que deverá ser aberta colocando o nome dela diretamente no atributo `action`. Um exemplo é apresentado na **Listagem 3**. Assim, após o clique no link *Voltar para Login* a aplicação será direcionada para `index.xhtml`;

2. Navegação Explícita e Dinâmica: deste modo as regras de navegação são definidas em um arquivo XML, chamado `faces-config.xml`. Veja um exemplo na **Listagem 4**.

Listagem 3. Exemplo de configuração para navegação estática.

```
<h:body>
<h2>Página Principal</h2>
<h:form>
  <h:commandLink action="index" value="Voltar para Login" />
</h:form>
</h:body>
```

Listagem 4. Exemplo de configuração para navegação explícita e dinâmica.

```
<navigation-rule>
  <from-view-id>/login.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>logar</from-outcome>
    <to-view-id>/home.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>error</from-outcome>
    <to-view-id>/login.jsf</to-view-id>
  </navigation-case>
  ...
</navigation-rule>
```

Neste caso, a tag `<navigation-rule>` informa a existência de uma regra de navegação. Dentro dela, a tag `<from-view-id>` determina em qual página o usuário está quando executa a ação de envio de dados para o servidor. Logo após temos a tag `<navigation-case>`, que contém outras duas: `<from-outcome>` e `<to-view-id>`. Na primeira é onde definimos a String de saída ou retorno do método no bean gerenciado após a requisição ser executada e na segunda especificamos para qual página o usuário deve ser redirecionado caso o retorno seja igual ao valor informado em `<from-outcome>`.

Como apresentado, o framework JSF é uma API para desenvolvimento de soluções web bastante completa e onde o reuso de componentes permite rapidez e facilidade na construção de UI (*User Interface*). Nela, há um conjunto de componentes que vão desde recursos simples, como o input e botões, a outros mais sofisticados, como tabelas com ordenação e filtros, caixas de sugestão, menus e

Figura 3. Janela de cadastro de aluno

gráficos. Para mais informações sobre JSF 2 veja a documentação no site oficial.

Cenário exemplo para implementação da parte prática

Após conhecer os principais conceitos relacionados ao Bootstrap e JSF, bem como de Design Responsivo, iniciaremos a parte prática, a qual se baseia no desenvolvimento de uma tela de cadastro de alunos e possibilitará ao leitor visualizar o comportamento das tecnologias abordadas conforme o tamanho da tela do dispositivo visualizador, assim como alguns dos seus recursos. A **Figura 3** mostra a janela de cadastro da nossa aplicação, que será construída utilizando a IDE NetBeans 8.0.2, o JDK 7, JSF 2, Bootstrap 3 e o Tomcat 8.0.15.

Configuração do ambiente

Para desenvolver o exemplo faz-se necessário instalar os softwares que auxiliarão na implementação e também obter as bibliotecas que farão parte do projeto. A configuração do ambiente como um todo se baseia em duas etapas:

- 1) download e instalação das bibliotecas e,
- 2) criação do projeto.

Download e instalação das bibliotecas

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

- JDK 1.7 ou superior;
- NetBeans 8.0.2;
- Tomcat 8.

Além destas, é preciso realizar o download do Bootstrap 3 que, como vimos, será utilizado para a implementação do Design Responsivo. Para isso, basta acessar a seção de downloads na página do Bootstrap, baixar o arquivo `bootstrap-3.3.5-dist.zip` e descompactá-lo em um local de sua preferência. Este arquivo contém todas as bibliotecas referentes ao Bootstrap que desejamos para o nosso exemplo. Além disso, iremos precisar do plugin jQuery `jquery.min.js`, pois códigos JavaScript do Bootstrap necessitam do jQuery para funcionar. Para obter o jQuery basta realizar o download em sua página oficial.

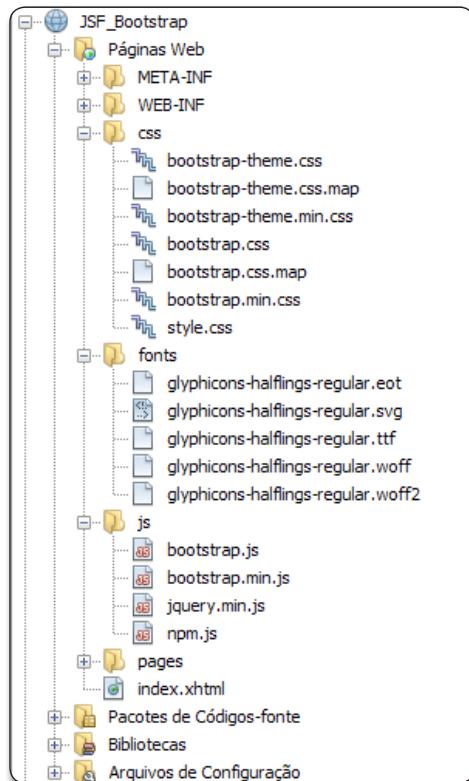


Figura 4. Bibliotecas Bootstrap necessárias para o projeto

Quanto ao JSF, iremos adotar a versão 2.2, que acompanha a IDE NetBeans.

Criação do projeto

Para configurar o ambiente onde será desenvolvida a aplicação, execute os seguintes passos:

1. No NetBeans, acesse a opção de menu *Arquivo > Novo Projeto*. Em seguida, escolha em *Categorias* a opção *Java Web*, em *Projetos* escolha *Aplicação Web* e clique em *Próximo*;
2. Informe o nome do projeto como “JSF_Bootstrap”, selecione o *Servidor* (neste caso Apache Tomcat 8.0.15) e a *Versão do Java EE (Java EE 7 Web)* para o projeto e clique em *Próximo*;
3. Marque a caixa de seleção com o framework *JavaServer Faces* e clique em *Finalizar*;
4. Na aba *Projetos*, clique com o botão direito sobre *Páginas Web* e acesse *Novo > Pasta*. Em *Nome da Pasta*, informe “css”. Repita esse procedimento para a criação das pastas: *css*, *js* e *pages*;
5. Agora, navegue até a pasta *bootstrap-3.3.5-dist* e copie os arquivos para as respectivas pastas criadas no passo quatro. Além disso, insira na pasta *js* o Plugin *jQuery jquery.min.js*;
6. No final do processo, a pasta *Páginas Web* conterá as bibliotecas do Bootstrap necessárias para a execução da aplicação e teremos o projeto configurado conforme a **Figura 4**.

Desenvolvendo a aplicação

Com as bibliotecas instaladas e o ambiente de desenvolvimento pronto, podemos iniciar, de fato, a codificação da aplicação. As-

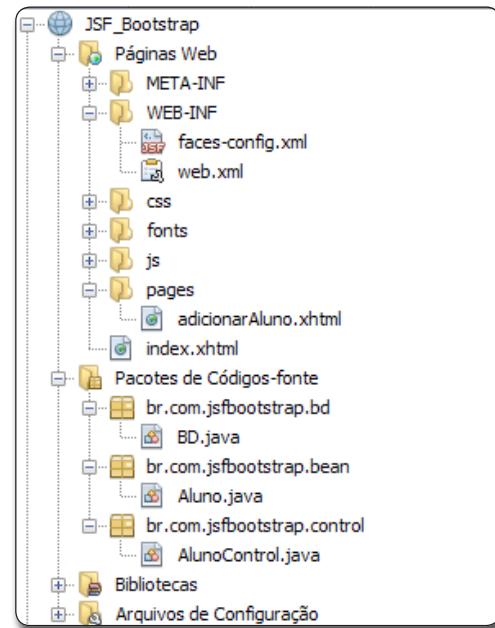


Figura 5. Visão final do projeto

sim, serão criadas as classes, arquivos de configuração e páginas HTML necessários ao funcionamento do cadastro, bem como realizados os devidos ajustes para o funcionamento da aplicação como um todo.

Implementação da aplicação

Após o download e instalação das bibliotecas e a criação do projeto, podemos iniciar a implementação da aplicação. A seguir listamos os passos para isso:

1. Crie três pacotes no projeto *JSF_Bootstrap*, chamados **br.com.jsfbootstrap.bd**, **br.com.jsfbootstrap.bean** e **br.com.jsfbootstrap.control**;
2. Crie uma página JSF com o nome *adicionarAluno* na pasta *pages*. Seu código é apresentado na **Listagem 5** e **6**;
3. Crie a classe **AlunoControl** dentro do pacote **br.com.jsfbootstrap.control** e a implemente conforme descrito na **Listagem 7**;
4. Crie a classe **Aluno** no pacote **br.com.jsfbootstrap.bean**. Seu código é apresentado na **Listagem 8**;
5. Crie a classe **BD** no pacote **br.com.jsfbootstrap.bd** conforme o código da **Listagem 9**;
6. Crie um arquivo com a extensão XML chamado *faces-config* na pasta *WEB-INF*. Seu código é apresentado na **Listagem 10**;
7. Substitua o conteúdo do arquivo *web.xml*, existente no diretório *WEB-INF*, pelo conteúdo da **Listagem 11**;
8. Salve todos os arquivos e clique em *Executar > Limpar e Construir Projeto* para que o código criado seja compilado;
9. Publique o projeto no servidor Tomcat clicando sobre o mesmo com o botão direito do mouse e escolhendo a opção *Executar*.

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

Listagem 5. adicionarAluno.xhtml: tela para cadastro de alunos – Parte 1.

```
01. <?xml version='1.0' encoding='UTF-8'?>
02. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
03. <html xmlns="http://www.w3.org/1999/xhtml"
04.   xmlns:h="http://xmlns.jcp.org/jsf/html"
05.   xmlns:f="http://xmlns.jcp.org/jsf/core">
06.   <h:head>
07.     <meta charset="utf-8"/>
08.     <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
09.     <meta name="viewport" content="width=device-width, initial-scale=1"/>
10.    <!-- As três meta tags acima devem vir em primeiro lugar no head da página
      e em seguida coloca-se as demais tags-->
11.   <title>Bootstrap</title>
12.   <!-- Folhas de estilo do Bootstrap -->
13.   <link href=".css/bootstrap.min.css" rel="stylesheet"/>
14.   <link href=".css/style.css" rel="stylesheet"/>
15. </h:head>
16. <h:body>
17.   <!-- Plugin jQuery necessário para o JavaScript do Bootstrap -->
18.   <script src=".js/jquery.min.js"></script>
19.   <!-- Plugin JavaScript do Bootstrap -->
20.   <script src=".js/bootstrap.min.js"></script>
21.   <nav class="navbar navbar-inverse navbar-fixed-top">
22.     <div class="container">
23.       <div class="navbar-header">
24.         <button type="button" class="navbar-toggle collapsed"
25.             data-toggle="collapse" data-target="#navbar" aria-expanded="false"
26.             aria-controls="navbar">
27.           <span class="sr-only">Toggle navigation</span> <span
28.             class="icon-bar"></span> <span class="icon-bar"></span> <span
29.             class="icon-bar"></span>
30.         </button>
31.         <a class="navbar-brand" href="#">Cadastro de Aluno</a>
32.       </div>
33.       <div id="navbar" class="navbar-collapse collapse">
34.         <ul class="nav navbar-nav navbar-right">
```



```
35.           <li><a href="#">Início</a></li>
36.           <li><a href="#">Opções</a></li>
37.           <li><a href="#">Perfil</a></li>
38.           <li><a href="#">Ajuda</a></li>
39.         </ul>
40.       </div>
41.     </div>
42.   </nav>
43.   <hr />
44.   <h:form>
45.     <div id="main" class="container">
46.       <h3 class="page-header">Adicionar Aluno</h3>
47.       <!-- Área de campos do form -->
48.       <div class="row">
49.         <div class="form-group col-sm-7 col-md-4">
50.           <h:outputLabel value="Nome" for="nome"/>
51.           <h:inputText id="nome" class="form-control"
52.             value="#{alunoController.aluno.nome}" title="Nome"/>
53.         </div>
54.         <div class="form-group col-sm-3 col-md-4">
55.           <h:outputLabel value="Data de Nascimento" for="dataNascimento"/>
56.           <h:inputText id="dataNascimento" class="form-control"
57.             value="#{alunoController.aluno.dataNascimento}"
58.             title="Data de Nascimento"/>
59.         </div>
60.         <div class="form-group col-sm-2 col-md-4">
61.           <h:outputLabel value="Sexo" for="sexo"/>
62.           <h:selectOneMenu id="sexo" value="#{alunoController.aluno.sexo}"
63.             class="form-control" title="Sexo">
64.             <f:selectItem itemValue="A" itemLabel="Selecione..."/>
65.             <f:selectItem itemValue="M" itemLabel="Masculino"/>
66.             <f:selectItem itemValue="F" itemLabel="Feminino"/>
67.           </h:selectOneMenu>
68.         </div>
69.       </div>
```

Listagem 6. adicionarAluno.xhtml: tela para cadastro de alunos – Parte 2.

```
01.   <div class="row">
02.     <div class="form-group col-sm-12 col-md-3">
03.       <h:outputLabel value="Endereço" for="endereco"/>
04.       <h:inputText id="endereco" class="form-control"
05.         value="#{alunoController.aluno.endereco}" title="Endereço"/>
06.     </div>
07.     <div class="form-group col-sm-4 col-md-3">
08.       <h:outputLabel value="Bairro" for="bairro"/>
09.       <h:inputText id="bairro" class="form-control"
10.         value="#{alunoController.aluno.bairro}" title="Bairro"/>
11.     </div>
12.     <div class="form-group col-sm-4 col-md-3">
13.       <h:outputLabel value="Cidade" for="cidade"/>
14.       <h:inputText id="cidade" class="form-control"
15.         value="#{alunoController.aluno.cidade}" title="Cidade"/>
16.     </div>
17.     <div class="form-group col-sm-4 col-md-3">
18.       <h:outputLabel value="Estado" for="estado"/>
19.       <h:inputText id="estado" class="form-control"
20.         value="#{alunoController.aluno.estado}" title="Estado"/>
21.     </div>
22.     <div class="row">
23.       <div class="form-group col-sm-4 col-md-2">
24.         <h:outputLabel value="CPF" for="cpf"/>
25.         <h:inputText id="cpf" class="form-control"
26.           value="#{alunoController.aluno.cpf}" title="CPF"/>
27.       </div>
28.     <div class="form-group col-sm-4 col-md-2">
```



```
29.       <h:outputLabel value="RG" for="rg"/>
30.       <h:inputText id="rg" class="form-control"
31.         value="#{alunoController.aluno.rg}" title="RG"/>
32.     </div>
33.     <div class="form-group col-sm-4 col-md-2">
34.       <h:outputLabel value="Telefone" for="fone"/>
35.       <h:inputText id="fone" class="form-control"
36.         value="#{alunoController.aluno.fone}" title="Telefone"/>
37.     </div>
38.     <hr />
39.     <div id="actions" class="row">
40.       <div class="col-md-3">
41.         <h:commandButton value="Salvar" type="submit"
42.           class="btn btn-primary btn-space" action="#{alunoController.criar}"/>
43.         <h:commandButton value="Cancelar" type="submit" class=
44.           "btn btn-primary btn-space" action="#{alunoController.limpar}"/>
45.       </div>
46.     </div>
47.   </h:form>
48. </h:body>
49. </html>
```

Entendendo o código

Caso todos os artefatos tenham sido criados corretamente, nossa página de cadastro estará funcionando e poderá ser utilizada sem problemas. Comecemos então a explicar o código do projeto pela página JSF *adicionarAluno.xhtml* (vide **Listagens 5 e 6**), que recebeu algumas configurações para comportar-se conforme o esperado de acordo com o dispositivo visualizador. Nas linhas 3 a 5 observamos a declaração de tags JSF que permitem a utilização de elementos HTML como *head*, *body*, *inputs*, *labels*, *combos*, entre outros. Em seguida, dentro do **head**, temos as seguintes meta tags:

1. **<meta charset="utf-8"/>**: permite determinar qual o encoding da página, ou seja, possibilita definir a codificação dos caracteres, corrigindo problemas de acentuação do texto presente na página;
2. **<meta http-equiv="X-UA-Compatible" content="IE=edge"/>**: estabelece o padrão de compatibilidade para o navegador Internet Explorer, informando que o mesmo deve executar o padrão de compatibilidade implementado na última versão do navegador (Edge) de modo a evitar erros na renderização da página;
3. **<meta name="viewport" content="width=device-width, initial-scale=1"/>**: estabelece o tamanho da parte visível em uma página renderizada pelo navegador (**name="viewport"**) como sendo o tamanho da largura de tela do dispositivo que acessa essa página, tendo como zoom inicial o valor 1 (**content="width=device-width, initial-scale=1"**).

Logo após são declaradas no código as importações das folhas de estilos (linhas 13 e 14) e dos arquivos JavaScript, tanto do plugin jQuery quanto do Bootstrap (linhas 18 e 20), que realizam todo o trabalho na página para organizar os componentes dentro do layout conforme o tamanho da tela. Já na linha 21 temos um elemento **Navbar**, que define uma barra de navegação ou de menus presente em muitas páginas e que é muito útil e comum em projetos web. Esse componente pode ser obtido no site oficial

do Bootstrap, mais precisamente na seção de componentes do framework, onde existem vários exemplos de como utilizá-lo.

Ainda nessa listagem, verificamos a tag JSF mais importante da página: **<form/>**, a qual é responsável por realizar a ligação dos componentes da tela com as propriedades do *bean* gerenciado. Assim, relacionaremos cada campo de texto (**h:inputText**) a uma propriedade (**nome**, **endereco**, **email**, entre outras) do *bean* **alunoController** (visto mais adiante) de modo a passar as informações do aluno para o lado servidor, para serem processadas. Entretanto, antes de coletarmos as informações para serem passadas ao servidor, devemos construir o layout da página de maneira que ele seja responsivo e se ajuste ao tamanho da resolução da tela de cada dispositivo, o qual é o propósito deste artigo. Para isso, criamos um Grid System Bootstrap para organização dos componentes através da tag **<div/>** com a classe CSS *container* (vide linha 45). Por sua vez, ela conterá outras quatro tags **<div/>**, representantes das linhas, todas utilizando a classe CSS *row* e contendo um conjunto definido de colunas, conforme demonstra a **Figura 3**.

Com isso, observamos que nos elementos que representam as colunas foram usadas as classes CSS presentes no framework, tais como **form-group**, **col-sm-*** e **col-md-***, de forma a obter o resultado esperado quanto ao arranjo dos componentes na página. Cada uma implementa uma funcionalidade útil para o layout responsivo, sendo a classe **form-group** utilizada basicamente para dimensionar os elementos **<input/>**, **<textarea/>** e **<select/>** com largura 100% e organizar melhor o espaçamento entre eles, e as outras duas para determinar o tamanho da coluna conforme o dispositivo.

The screenshot shows a mobile application interface titled "Cadastro de Aluno". Below it, a sub-section titled "Adicionar Aluno" is displayed. The form contains five input fields: "Nome", "Data de Nascimento", "Sexo" (with a dropdown placeholder "Selecione..."), "Endereço", and "Bairro". Each field is contained within a "form-group" element, which includes a "col-sm-12" class to ensure full width on smaller screens like mobile devices.

Figura 6. Abordagem Mobile First com resolução 320x499



Além disso, identificamos na página as mesmas classes **col-sm-*** e **col-md-*** com um número de colunas diferentes. Isso foi feito para dividir nosso layout em três comportamentos, conforme explicamos a seguir:

- **Tela menor que 768px:** aplica-se a abordagem *Mobile First* e os elementos serão agrupados verticalmente, com espaçamento entre eles, e o menu da página irá aglutinar-se formando uma lista suspensa, conforme mostrado na Figura 6;

- **Tela maior ou igual a 768px e menor que 992px:** usa-se apenas as classes CSS relacionadas aos equipamentos classificados como *Small Devices*, ou seja, o framework Bootstrap irá percorrer todos os elementos da página aplicando as classes **col-sm-*** conforme os tamanhos das colunas definidos em cada um deles. Por exemplo, na primeira linha da tabela (linha 48) o layout seria dividido em três colunas com tamanhos sete (**col-sm-7**), três (**col-sm-3**) e dois (**col-sm-2**), conforme mostrado na Figura 7;

- **Tela maior ou igual a 992px** aplica-se apenas as classes CSS relacionadas aos equipamentos classificados como *Medium Devices*. Assim, para todos os tamanhos de tela maiores que 992px, o layout se comportará como se fosse um *Medium Device*, pois não definimos em nossa página nenhuma coluna com **col-lg-***, a qual é responsável por tratar casos onde a tela é maior. Desse modo, podemos verificar que na terceira linha da tabela o layout seria dividido da seguinte forma: dois **col-md-2**, dois

col-md-2, dois **col-md-2** e seis **col-md-6**. A Figura 8 apresenta esse comportamento;

Além disso, em nosso exemplo temos a classe Java **AlunoControl**, apresentada na Listagem 7 e representando o controlador da aplicação, sendo responsável por capturar os dados da página e enviá-los para o servidor. Para tanto, definimos a anotação **@ManagedBean** sobre a classe e nomeamos o bean como **alunoController** (vide linha 1), bem como definimos seu escopo através da anotação **@RequestScoped**, na linha 2. Em seguida, criamos dois

atributos privados, **banco** e **aluno**, com seus respectivos métodos gets e sets. O primeiro representa a base de dados do sistema e o segundo o aluno sendo cadastrado.

Ainda analisando o código dessa classe, repare que no construtor (veja a linha 13) ocorre uma chamada ao método **getInstance()** da classe **BD** (vide Listagem 8) para instanciar um objeto representante da base de dados, assim como é criada uma instância de **Aluno**, a qual terá seus atributos ligados aos elementos da janela de cadastro.

Figura 7. Configuração para Small Device com resolução 980x1280

Listagem 7. AlunoControl: classe que representa o bean gerenciado da aplicação.

```

01. @ManagedBean(name = "alunoController")
02. @RequestScoped
03. public class AlunoControl {
04.     private BD banco;
05.     private Aluno aluno;
06.
07.     public AlunoControl() {
08.         banco = BD.getInstance();
09.         aluno = new Aluno();
10.    }
11.
12.    public Aluno getAluno() {
13.        return aluno;
14.    }
15.
16.    public void setAluno(Aluno aluno) {
17.        this.aluno = aluno;
18.    }
19.
20.    public String criar() {
21.        banco.addAluno(aluno);
22.        setAluno(new Aluno());
23.        System.out.println("Aluno adicionado com sucesso.");
24.        return "pages.aluno.criar";
25.    }
26.
27.    public String limpar() {
28.        setAluno(new Aluno());
29.        return "pages.aluno.limpar";
30.    }
31.
32. }
```

The screenshot shows a web application window titled "Cadastro de Aluno" (Student Registration). The main title bar has links for "Inicio", "Opções", "Perfil", and "Ajuda". Below the title, there's a section titled "Adicionar Aluno" with various input fields. The fields include "Nome" (Name), "Data de Nascimento" (Birthdate), "Sexo" (Sex) with a dropdown menu, "Endereço" (Address), "Bairro" (Neighborhood), "Cidade" (City), "Estado" (State), "CPF" (CPF), "RG" (RG), "Telefone" (Telephone), and "EMAIL". At the bottom left are two buttons: "Salvar" (Save) and "Cancelar" (Cancel).

Figura 8. Configuração para Medium Devices com resolução 1280x600

Listagem 8. BD: classe utilizada como banco de dados da aplicação.

```
01. public class BD {
02.     private static BD instancia;
03.     private List<Aluno> alunos;
04.     private BD() {
05.         alunos = new ArrayList<Aluno>();
06.     }
07.     public static BD getInstance()
08.     {
09.         if(instancia == null)
10.             instancia = new BD();
11.         return instancia;
12.     }
13.     public List<Aluno> getAlunos() {
14.         return alunos;
15.     }
16.     public void addAluno(Aluno aluno)
17.     {
18.         alunos.add(aluno);
19.     }
20. }
```

Por último, temos os métodos **criar()** e **limpar()**, que são responsáveis por adicionar a instância do aluno ao banco e alterar a ligação do objeto aluno com a página através da criação de uma nova instância do mesmo. Assim, quando o usuário preencher as informações na página e clicar no botão *Salvar*, o JSF executará seu ciclo e suas etapas para associar os dados fornecidos à arvore de componentes e validá-los. Em seguida, executará o método **criar()**, que então adicionará o aluno à base de dados, imprimirá no console a mensagem sobre a operação e irá retornar uma **String** com

o valor “`pages.aluno.criar`”.

A partir desse retorno o framework acessará o arquivo de configuração `faces-config.xml` para procurar pela regra de navegação (`<navigation-rule>`) relacionada à janela que executou a ação de clique e encontrar o caso de navegação da página (`<navigation-case>`) que coincide com o valor de retorno do método. Feito isso, a nova janela será identificada pelo valor contido em `<to-view-id>`, criada e exibida para o usuário.

De maneira semelhante, quando o usuário clicar no botão *Cancelar*, o framework executará o método **limpar()** do bean gerenciado, onde será criada uma nova instância de **aluno** com os atributos vazios e que serão associados aos elementos da página de cadastro. Feito isso, o método retornará a **String** “`pages.aluno.limpar`” que, devido ao mapeamento no `faces-config.xml`, manterá a aplicação na mesma página.

Como mencionado, a classe **AlunoControl** contém um atributo privado do tipo **Aluno** (vide **Listagem 9**) que é utilizado na janela de cadastro e depois salvo na base de dados. Por se tratar de uma classe simples que possui apenas gets e sets, sua explicação será omitida.

Como curiosidade, note que a classe **BD** representa uma versão simplificada de um banco de dados em memória. Optamos por fazê-la dessa forma para manter o foco na responsividade das interfaces. Ainda assim, observe que ela foi implementada para garantir a existência de apenas uma instância por toda a aplicação. Para isso, foram criados dois atributos: um **static**, referenciando a instância do banco; e uma lista de objetos **aluno**, associada aos alunos já armazenados. Ademais, temos os métodos `getInstance()`, `getAlunos()` e `addAluno()`. O primeiro é responsável por criar uma instância de **BD** e retorná-la para o objeto chamador, o segundo retorna a lista de alunos e o terceiro adiciona um aluno ao banco.

Por fim, encerrando a implementação do nosso projeto, temos ainda dois arquivos XML. O primeiro deles, de nome `faces-config.xml`, é usado para viabilizar a navegação explícita e dinâmica entre as páginas (vide **Listagem 10**). Nele, temos a regra de navegação da página `/pages/adicionarAluno.xhtml` com duas saídas: `pages.aluno.limpar` e `pages.aluno.criar`, as quais direcionam a



aplicação para a mesma página que executou a ação (*adicionar-Aluno.xhtml*). O segundo XML é o arquivo *web.xml* (vide **Listagem 11**), que declara a configuração do servlet do JSF 2 para que tudo funcione na aplicação. Para tanto, no elemento **<servlet-name>** informamos um nome para o servlet, que neste caso será *Faces Servlet*, e em **<servlet-class>** informamos ao container de servlet (neste caso o Tomcat) qual a classe do JSF ele deve executar quando a aplicação iniciar. Finalmente, em **<url-pattern>**, dentro de **<servlet-mapping>**, especificamos a URL que o servlet *Faces Servlet* irá interceptar quando chegar uma requisição do tipo *http://<host:port>/JSF_Bootstrap/* ao servidor.

Listagem 9. Aluno: classe utilizada para representar um aluno.

```

01. public class Aluno {
02.     private String nome;
03.     private String dataNascimento;
04.     private String sexo;
05.     private String endereco;
06.     private String bairro;
07.     private String cidade;
08.     private String estado;
09.     private String cpf;
10.     private String rg;
11.     private String fone;
12.     private String email;
13.
14.     public Aluno() {
15. }
16.
17.     public String getNome() {
18.         return nome;
19.     }
20.
21.     public void setNome(String nome) {
22.         this.nome = nome;
23.     }
24.
25.     // Demais métodos get's e set's
26.
27.     public String toString()
28.     {
29.         return this.nome + "-" + this.dataNascimento + "-" + this.sexo +
30.             " " + this.endereco + "-" + this.bairro + "-" + this.cidade + "-" +
31.             this.estado + "-" + this.cpf + "-" + this.rg + "-" + this.fone + "-" + this.email;
32.     }

```

Implementados todos os artefatos, a aplicação cliente pode ser executada clicando com o botão direito sobre o projeto e acessando a opção *Executar*. Feito isso, ao informar a URL da página de cadastro no navegador (*pages/adicionarAluno.jsf*), uma janela semelhante à **Figura 1** será aberta para que as informações do aluno sejam fornecidas.

Para que seja visualizado o comportamento da página em diferentes tamanhos de tela, faz-se necessária a utilização de um navegador que possua a funcionalidade *Modo Design Adaptável*, recurso presente no Firefox. A partir deste, basta abrir a janela de cadastro de aluno e executar a ação *CTRL + SHIFT + M* para que a

Listagem 10. faces-config.xml: arquivo com as configurações de navegação nas páginas.

```

<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
    <navigation-rule>
        <from-view-id>/pages/adicionarAluno.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>pages.aluno.limpar</from-outcome>
            <to-view-id>/pages/adicionarAluno.jsf</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-outcome>pages.aluno.criar</from-outcome>
            <to-view-id>/pages/adicionarAluno.jsf</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>

```

Listagem 11. web.xml: arquivo de configuração da aplicação.

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.jsf</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>pages/adicionarAluno.xhtml</welcome-file>
    </welcome-file-list>
</web-app>

```

funcionalidade de design adaptável seja ativada. Ao realizar esse procedimento o leitor poderá notar que a janela irá se comportar de forma diferente para cada resolução escolhida. Por exemplo, caso seja escolhida a resolução 320x499, a janela terá a forma da **Figura 6** e caso a escolha seja por 980x1280, teremos um resultado semelhante ao exposto na **Figura 7**. O mesmo vale para a **Figura 8**, que será apresentada quando a resolução 1280x600 for escolhida no combo com os valores das resoluções.

Como a diversidade de equipamentos que acessam a internet é muito grande e aumenta a cada dia, a construção de sites

responsivos se tornou praticamente uma obrigação para os desenvolvedores e para o sucesso dos projetos web. Deste modo, sua adoção é considerada não mais como um novo recurso web, mas sim como uma necessidade que provoca grandes impactos no crescimento dos negócios online, os quais são muito dependentes do acesso destes diferentes tipos de equipamentos. Lembre-se que a implementação da responsividade melhora consideravelmente a experiência dos clientes ou visitantes, tornando a navegação mais rica e interessante quando se utiliza esse recurso nas páginas.

Para o leitor que deseja aprofundar-se no assunto, a seção **Links** fornece os subsídios necessários.

E se você está interessado em dominar o Bootstrap, mas ainda tem alguma dúvida sobre a relevância deste, vale ressaltar que hoje ele é o projeto mais popular do GitHub, com alto índice de aceitação entre os desenvolvedores e disponibilizando um rico conjunto de componentes em CSS, HTML, extensões em JavaScript e modos de customização, o que possibilita ser utilizado em qualquer projeto web.

Autor



Carlos Antônio Martins

carlosmartinsufu@yahoo.com.br



É formado em Ciência da Computação pela Universidade Federal de Uberlândia (UFU) com especialização em Desenvolvimento Java pelo Centro Universitário do Triângulo (UNITRI) e Análise e Desenvolvimento de Sistemas Aplicados a Gestão Empresarial no Instituto Federal do Triângulo Mineiro (IFTM). Trabalha na empresa Algar Telecom como Analista de Sistemas.

Enfim, o desenvolvimento de aplicações responsivas com JSF e Bootstrap se mostrou bastante simples e produtivo, valendo ao leitor, portanto, a análise desta nova opção ao iniciar o planejamento de novos projetos web.

Links:

Site oficial do Bootstrap.

<http://getbootstrap.com>

Página de Ethan Marcotte.

<http://ethanmarkotte.com/>

Site oficial do Mojarra, implementação do JSF.

<https://javaserverfaces.java.net/>

Endereço da página “A list apart”.

<http://alistapart.com/>

Site oficial do Tomcat.

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

Página da W3C Community and Business Groups.

<https://www.w3.org/community/>

Endereço com informações sobre Media Type e Media Queries.

<http://www.w3.org/TR/css3-mediaqueries/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Primeiros passos no mundo da Internet das Coisas – Parte 2

Veja neste artigo como “dar vida” aos seus protótipos colocando seu dispositivo IoT na web

ESTE ARTIGO FAZ PARTE DE UM CURSO

No artigo que deu início aos nossos primeiros passos em IoT concluímos que conhecer a Internet das Coisas, bem como saber onde essas novas tecnologias podem nos levar é extremamente importante, pois percebemos que IoT é o modelo de construção de soluções na internet do futuro e que já se apresenta muito promissora no presente.

Neste cenário, a internet ganha uma tendência de descentralização ao passo que os dispositivos começam a se intercomunicar independentemente de servidores, alimentando outros sistemas de forma mais automática, com menor dependência de interação humana. Ao mesmo tempo, as pessoas passam a estar mais conectadas não somente a outras pessoas, mas também a outros dispositivos e ambientes, como a própria casa ou aparelhos eletrodomésticos. Neste paradigma, ambientes e coisas ganham a possibilidade de gerenciarem a si próprios com pouca ou nenhuma interação humana. Esta é uma visão futurística, mas que, na verdade, já faz parte da realidade em países mais evoluídos tecnologicamente; uma pesquisa rápida pelo termo “Internet of Things” em vídeos na internet, demonstra que existe uma quantidade enorme de organizações já visualizando IoT como o próximo passo de evolução das tecnologias modernas.

Seguindo o caminho de descoberta, neste artigo daremos continuidade ao experimento iniciado na publicação anterior, quando iniciamos o desenvolvi-

Fique por dentro

Na primeira parte que iniciou esta jornada, vimos o que é e como funciona a Internet das Coisas, bem como porquê ela é importante e se apresenta como um presente já promissor. Para isso, criamos um protótipo de hardware que funciona como medidor de altura e agora chegou a vez de conectá-lo à internet e dar sentido ao termo IoT. Deste modo, vamos transformar o nosso dispositivo limitado em um aparelho que pode exibir essas medidas de altura a qualquer usuário na internet e com potencial, inclusive, de se conectar a redes sociais com poucas alterações.

mento de um dispositivo capaz de medir a altura de um corpo que se posicione abaixo do sensor ultrassônico. Já criamos toda a camada de controle do Arduino através da instalação do JArduino como firmware e da criação do Sketch Java, porém ainda não desenvolvemos a solução web que colocará o nosso aparelho na internet, para que o seu estado esteja disponível como informação em uma aplicação. Para esta solução, vamos utilizar WebSockets e ligar o dispositivo à camada de visualização web, promovendo interatividade entre o usuário e o aparelho. Portanto, mãos à obra!

Conectando coisas à Internet

Habilitar o nosso medidor de altura para a Internet é uma parte importante do processo de criar um dispositivo IoT. Sem que haja conectividade e disponibilidade do equipamento para o “mundo”, não há também a Internet das Coisas. Em nosso experimento, o objetivo é torná-lo disponível para que qualquer interface, seja através de um dispositivo móvel como um smartphone ou através de uma página web, possa apresentar o estado do sensor e também possa interagir com o disposi-

tivo a qualquer momento solicitando a sua calibragem. Neste experimento abordaremos a implementação de uma página web que se comunique com um WebSocket para obter e enviar informações ao equipamento, porém, uma vez desenvolvida a aplicação WebSocket, qualquer outro meio de diálogo com usuário poderá se conectar ao dispositivo simultaneamente, permitindo que diferentes modelos de apresentação da informação do sensor sejam integrados. Um exemplo positivo desta flexibilidade pode ser verificado em uma situação em que o desenvolvedor queira dar suporte a deficientes auditivos, por exemplo. Neste caso, uma aplicação conectada a um alto-falante e a um microfone pode proporcionar o mesmo nível de interatividade ao usuário com tal deficiência, sem a necessidade de reconstrução de toda a aplicação, dependendo somente da criação de mais um cliente WebSocket.

Como verificado na **Listagem 1**, o cliente WebSocket é criado através da classe **br.com.pontoclass.iot.hmeter.webcosket**.**WebSocket**, que atua como um Endpoint – ou seja, um ponto de destino do WebSocket – encapsulando toda a complexidade de conexão e mensageria. É através dessa classe que o Sketch Java se comunica com o “mundo” para informar sobre sua alteração de estado e para receber o pedido de calibragem. Nesta classe, o método **onMessage(String message)** recebe as mensagens que chegam do servidor web que criaremos mais adiante e

as redireciona para o método **byName(String message)**, do enum **br.com.pontoclass.iot.hmeter.strategy.Strategy** (vide **Listagem 2**). Este, por sua vez, retorna um valor de enum dependendo do conteúdo da mensagem. Tal resultado é utilizado para reconfigurar o Sketch representado pela instância **ultrassonic** através do seu método **setStrategy(Strategy strategy)**.

Desta forma, conforme o diálogo com o servidor WebSocket se desenvolve, o comportamento do Sketch **Ultrassonic** (visto no artigo publicado na Java Magazine 147) pode mudar devido à atualização do **Strategy** configurado para ser executado no método **loop()**. Para tornar mais clara a interatividade neste fluxo, o leitor deve se lembrar que na classe **Ultrassonic** o método **loop()** era acionado periodicamente para executar o **Strategy** configurado. O **Strategy** padrão tratava de enviar uma mensagem ao servidor WebSocket a fim de informá-lo sobre a disponibilidade do dispositivo para ser utilizado. A partir desta primeira comunicação, este servidor WebSocket (que criaremos logo mais) tratará de intermediar a comunicação entre o Endpoint WebSocket da camada front-end (também a ser criado ainda neste artigo) e o **WebSocket** da **Listagem 1**.

Neste experimento o enum **Strategy** é essencial, uma vez que é ele o responsável por encapsular todos os comportamentos do Sketch **Ultrassonic**. Optamos por um enum para facilitar o processo de transformação da mensagem no formato **String** para o

Listagem 1. Código da classe WebSocket.

```
01. package br.com.pontoclass.iot.hmeter.websocket;
02.
03. import java.io.IOException;
04. import java.net.URI;
05. import java.net.URISyntaxException;
06. import javax.websocket.ClientEndpoint;
07. import javax.websocket.ContainerProvider;
08. import javax.websocket.DeploymentException;
09. import javax.websocket.OnMessage;
00. import javax.websocket.OnOpen;
01. import javax.websocket.Session;
02. import javax.websocket.WebSocketContainer;
03. import br.com.pontoclass.iot.hmeter.sketch.Ultrassonic;
04. import br.com.pontoclass.iot.hmeter.strategy.Strategy;
05.
06. @ClientEndpoint
07. public class WebSocket {
08.
09.     private URI uri;
10.     private Session session;
11.     private Ultrassonic ultrassonic;
12.
13.     public WebSocket(Ultrassonic ultrassonic, String uri) {
14.         this.ultrassonic = ultrassonic;
15.         try {
16.             this.uri = new URI(uri);
17.         } catch (URISyntaxException e) {
18.             throw new RuntimeException(e.getMessage());
19.         }
20.     }
21.
22.     public void connect() {
23.         WebSocketContainer container = ContainerProvider.getWebSocketContainer();
24.         try {
25.             this.session = container.connectToServer(this, getURI());
26.             this.session.getBasicRemote().sendText("CLIENT:HW");
27.         } catch (DeploymentException | IOException e) {
28.             throw new RuntimeException(e.getMessage());
29.         }
30.     }
31.
32.     public URI getURI() {
33.         return uri;
34.     }
35.
36.     public void setUrl(URI uri) {
37.         this.uri = uri;
38.     }
39.
40.     public @OnOpen void onOpen(Session session) {
41.         this.session = session;
42.     }
43.
44.     public @OnMessage void onMessage(String message) {
45.         ultrassonic.setStrategy(Strategy.byName(message));
46.     }
47.
48.     public void sendMessage(String value) throws IOException {
49.         session.getBasicRemote().sendText(value);
50.     }
51. }
```

valor correto de enum, já que pela utilização de enum estamos limitando as instâncias possíveis e forçando que esses valores não tenham “estado” que possa gerar efeitos colaterais durante a execução. Esses valores, então, determinam como o Sketch irá se comportar durante a execução do método `loop()` da classe `Ultrassonic`, dado que nesta classe existe apenas o redirecionamento da execução para a variável de instância `strategy`, que pode ser modificada dependendo da mensagem recebida pelo método `onMessage(String message)` de `WebSocket`.

Criamos este mecanismo para que durante a troca de mensagens o sensor possa receber e responder aos comandos do `WebSocket`. Com isto, estabelece-se um padrão de troca de mensagens em que a classe `WebSocket` modifica o `Strategy` em execução em função do valor recebido em formato de `String` (pedindo para que o sensor seja calibrado através do valor `Calibrate`, por exemplo), ao

passo que durante a execução do próprio `Strategy`, ele pode enviar mensagens informando sobre o que está acontecendo (dizendo que o processo de calibragem foi iniciado, por meio da mensagem `“HW:BeginCalibrate”`, por exemplo).

No enum `Strategy` temos os três valores representando as ações possíveis dentro do Sketch: `Available`, `Calibrate` e `KeepAlive`. `Available` não executa nenhuma ação diretamente no sensor, mas sim na comunicação com o socket, através da chamada `ultrassonic.getWebSocket().sendMessage("HW:HWAvailable")`, nas linhas 60 e 61, enviando a informação com o valor “`HW:HWAvailable`” para informar aos outros nós – ou clientes – do `WebSocket` que o sensor está disponível para ser utilizado.

Como veremos mais tarde, tais clientes são as interfaces web acessadas via browser para que haja a interação do usuário com o dispositivo. Esta e todas as outras mensagens seguem um

Listagem 2. Declaração do enum `Strategy`.

```

01. package br.com.pontoclass.iot.hmeter.strategy;
02.
03. import java.io.IOException;
04. import java.util.Optional;
05. import java.util.logging.Logger;
06. import java.util.stream.IntStream;
07. import br.com.pontoclass.iot.hmeter.sketch.Ultrassonic;
08.
09. public enum Strategy {
10.     Calibrate() {
11.
12.         private static final int PRECISION = 5;
13.
14.         @Override
15.         public float execute(Ultrassonic ultrassonic) {
16.             try {
17.                 ultrassonic.getWebSocket()
18.                     .sendMessage("HW:BeginCalibrate");
19.                 Float result = IntStream.range(1, PRECISION)
20.                     .mapToObj(i -> ultrassonic.average())
21.                     .reduce((a, b) -> a+b+0f)
22.                     .map(f -> f/PRECISION)
23.                     .orElse(-1f);
24.                 ultrassonic.getWebSocket()
25.                     .sendMessage(String.format("HW:Calibrate:%.2f",
26.                         result));
27.             return result;
28.         } catch (IOException e) {
29.             throw new RuntimeException(e.getMessage());
30.         } finally {
31.             ultrassonic.setStrategy(KeepAlive);
32.         }
33.     }
34.
35. },
36. KeepAlive() {
37.
38.     @Override
39.     public float execute(Ultrassonic ultrassonic) {
40.         try {
41.             Float result = ultrassonic.average();
42.             ultrassonic.getWebSocket()
43.                 .sendMessage(String.format("HW:KeepAlive:%.2f",
44.                     result));
45.             return result;
46.         } catch (IOException e) {
47.             throw new RuntimeException(e.getMessage());
48.         }
49.     },
50. },
51. Available() {
52.
53.     private Logger LOGGER = Logger.getLogger(this.getClass().getName());
54.
55.     @Override
56.     public float execute(Ultrassonic ultrassonic) {
57.
58.         LOGGER.info("No action set up...");
59.         try {
60.             ultrassonic.getWebSocket()
61.                 .sendMessage("HW:HWAvailable");
62.         } catch (IOException e) {
63.             throw new RuntimeException(e.getMessage());
64.         }
65.         try {
66.             Thread.sleep(3000);
67.         } catch (InterruptedException e) {
68.             LOGGER.warning(e.getMessage());
69.         }
70.         return -1;
71.     }
72. };
73.
74. public String getName() {
75.     return this.toString();
76. }
77.
78. public abstract float execute(Ultrassonic ultrassonic);
79.
80. public static Strategy byName(String message) {
81.     return Optional.ofNullable(Strategy.valueOf(message))
82.         .orElse(KeepAlive);
83. }
84. }
```

pequeno protocolo criado neste experimento, que consiste no padrão “ID:AÇÃO[:PARAMS]”, onde ID é o identificador do cliente (aqui utilizamos HW, uma vez que o servidor WebSocket compreenderá este valor como sendo uma ação do hardware), e AÇÃO é o nome da ação a ser realizada pelos receptores. Caso existam outros valores separados por dois-pontos (:) – [:PARAMS] – eles serão considerados como parâmetros a serem enviados ao receptor. **Calibrate**, por sua vez, executa a ação de calibração do medidor de altura. Para isso, inicialmente é enviada uma mensagem ao socket por meio da chamada **ultrassonic.getWebSocket().send Message("HW:BeginCalibrate")**, nas linhas 17 e 18, com o valor “HW:BeginCalibrate”, informando aos clientes do WebSocket sobre o início da calibração, para que eles tenham a oportunidade de exibir o estado do medidor. Em seguida, após o término da calibração realizada no código da linha 19 a 23, é enviada uma nova mensagem, “HW:Calibrate:VALOR”, onde VALOR é o valor obtido através da média de leituras no medidor. Neste envio, o método **String.format()** é utilizado para transformar e formatar o resultado da calibração do tipo **float** em uma **String** com duas casas decimais (“%.2f”), desta forma permitindo que o padrão “ID:AÇÃO[:PARAMS]” seja mantido no envio da mensagem.

No final da calibração, o próprio enum **Calibrate** configura o Sketch **Ultrasonic** para que ele passe a executar a ação **KeepAlive**, através da chamada **ultrassonic.setStrategy(KeepAlive)**. Com isso, o Sketch passa a realizar consultas periódicas no sensor, enviando o resultado dessa consulta no formato “HW:KeepAlive:VALOR”, onde VALOR é o resultado da média obtida através da chamada **ultrassonic.average()**.



Listagem 3. Código da classe Bootstrap.

```
01. package br.com.pontoclass.iot.hmeter.main;
02.
03. import br.com.pontoclass.iot.hmeter.sketch.Ultrasonic;
04.
05. public class Bootstrap {
06.
07.     private Ultrasonic ultrasonic;
08.
09.     public static void main(String[] args) throws InterruptedException {
10.         Bootstrap instance = new Bootstrap();
11.         instance.start();
12.         try {
13.             instance.wait();
14.         } finally {
15.             instance.stop();
16.         }
17.     }
18.
19.     public Bootstrap() {
20.         ultrasonic = new Ultrasonic();
21.     }
22.
23.     private void start() {
24.         ultrasonic.runArduinoProcess();
25.     }
26.
27.     private void stop() {
28.         ultrasonic.stopArduinoProcess();
29.     }
30. }
```

Como podemos perceber, a maior vantagem de utilizar Java para controlar o Arduino é a possibilidade de mudar o comportamento do Sketch em tempo de execução, da mesma forma que fazemos em uma aplicação Java qualquer. Esta opção é sutil, mas extremamente poderosa, pois a partir desta implementação é possível adicionar comportamentos ao Sketch através da criação de novos valores no enum **Strategy** e de sua utilização no fluxo do loop no Sketch **Ultrasonic**. Por exemplo, imagine se o desenvolvedor decide de que também vai adicionar um comportamento que transforme o medidor de altura em um alarme quando detectar a abertura da porta em um determinado horário. Este comportamento seria muito fácil de ser adicionado. Seria possível, inclusive, alterar a implementação do **Strategy** para a utilização de uma interface com implementações em classes concretas ao invés de em um enum e com isso permitir a transferência de novos códigos Java que alterem o comportamento do Sketch – e consequentemente do sensor – sem necessidade de recompilar o código em execução.

Após a implementação do Sketch, é possível criar uma classe que o inicie e permaneça rodando, como o que foi feito na **Listagem 3**, que apresenta o código de **br.com.pontoclass.iot.hmeter.main.Bootstrap**.

A classe **Bootstrap** é uma implementação simples de uma classe com o método **main()** que instancia e inicia o Sketch **Ultrasonic** para que ele entre em loop durante todo o tempo de execução, comunicando-se diretamente com o Arduino para enviar e

receber instruções. Com esta classe será possível salvar a aplicação em um arquivo JAR e executá-lo individualmente no Raspberry Pi ou em qualquer outro computador. Porém, como o Sketch Ultrasonic conecta-se a um WebSocket, precisamos de uma aplicação Web com a implementação do lado servidor do socket. Tal implementação pode ser criada separadamente como um web-app que será executado no Raspberry Pi em um Web container Tomcat – ou qualquer outro container Web – como veremos no próximo tópico.

Listagem 4. Declaração do projeto websocket com Maven.

```

01. <?xml version="1.0"?>
02. <project xmlns="http://maven.apache.org/POM/4.0.0"
03.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                           http://maven.apache.org/xsd/maven-4.0.0.xsd">
05.   <modelVersion>4.0.0</modelVersion>
06.
07.   <parent>
08.     <groupId>br.com.pontoclass</groupId>
09.     <artifactId>iot</artifactId>
10.   <version>1.0.0-SNAPSHOT</version>
11. </parent>
12.
13. <groupId>br.com.pontoclass.iot</groupId>
14. <artifactId>websocket</artifactId>
15. <packaging>war</packaging>
16. <name>Internet of Things - Web Socket</name>
17.
18. <dependencies>
19.   <dependency>
20.     <groupId>junit</groupId>
21.     <artifactId>junit</artifactId>
22.     <scope>test</scope>
23.   </dependency>
24.
25.   <dependency>
26.     <groupId>javax.websocket</groupId>
27.     <artifactId>javax.websocket-api</artifactId>
28.     <version>1.1</version>
29.   </dependency>
30.
31.   <dependency>
32.     <groupId>javax</groupId>
33.     <artifactId>javaee-web-api</artifactId>
34.     <version>7.0</version>
35.   </dependency>
36. </dependencies>
37.
38. <build>
39.   <finalName>websocket</finalName>
40. </build>
41. </project>
```

WebSocket das Coisas

No exemplo deste artigo, a implementação da camada web será realizada através da utilização da API WebSocket, especificada na JSR 356, da plataforma Java EE. A opção pela utilização de Web-Sockets é bastante conveniente devido à possibilidade

de fazer push – isto é, chamadas do servidor diretamente à camada front-end –, diferentemente de outras tecnologias como Servlets, que necessitariam de monitoração ativa por parte dos clientes no front-end. Esta camada é criada em outro módulo Maven, cuja declaração pode ser conferida na **Listagem 4**. A declaração é bem simples e basicamente adiciona as dependências principais `javax.websocket:javax.websocket-api:1.1` e `javax:javaee-web-api:7.0`. Neste módulo criaremos tanto o WebSocket (server) quanto a página para visualização do estado no sensor, bem como a implementação Angular.js que possibilitará a interação entre as duas camadas.

A classe `br.com.pontoclass.iot.websocket.HMeterServer` (vide **Listagem 5**), cria um WebSocket de forma simplificada através da utilização das anotações previstas pela especificação. A sua função é realizar o tratamento de sessões e o recebimento das mensagens dos clientes. Visto que queremos ver o resultado da medição de altura do nosso dispositivo em um navegador web, torna-se necessário abrir uma conexão WebSocket também da camada de visualização web com o servidor WebSocket, que, por sua vez, como visto na primeira parte do artigo, já tem sua conexão estabelecida com a implementação Sketch controladora do dispositivo.

Essa classe possui a função básica de limpar as sessões que forem fechadas nos métodos `onClose()` e `onError()` (quando houver perda de conexão) e, principalmente, realizar o mapeamento das mensagens recebidas para as ações correspondentes, como podemos ver no método `onMessage()`. As primeiras linhas deste método verificam se a mensagem recebida é a identificação de um novo dispositivo (HW) ou de um novo cliente web (WEB). Desta forma, como podemos verificar no código anterior, da **Listagem 1**, a classe `WebSocket` envia a mensagem “CLIENT:HW” após o sucesso na conexão com o servidor WebSocket e esta mensagem



Primeiros passos no mundo da Internet das Coisas – Parte 2

Listagem 5. Implementação do WebSocket – classe HMeterServer.

```
01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.ArrayList;
04. import java.util.Collections;
05. import java.util.List;
06. import java.util.Optional;
07. import java.util.logging.Level;
08. import java.util.logging.Logger;
09. import javax.websocket.OnClose;
10. import javax.websocket.OnError;
11. import javax.websocket.OnMessage;
12. import javax.websocket.Session;
13. import javax.websocket.server.ServerEndpoint;
14.
15. public @ServerEndpoint("/hmeter") class HMeterServer {
16.     private static final Logger LOGGER = Logger.getLogger(
17.         HMeterServer.class.getName());
18.     private static Optional<Session> hWSession = Optional.empty();
19.     private static List<Session> webSessions = Collections.synchronizedList(
20.         new ArrayList<>());
21.     public @OnClose void onClose(Session session) {
22.         LOGGER.log(Level.INFO, "Conexão finalizada com o cliente: {0}", session.getId());
23.         if(s.getId().equals(session.getId())) {
24.             ActionFactory.factory("HW")
25.                 .handle("HW:HWLost".split(":"), hWSession, webSessions);
26.         }
27.     });
28.     webSessions.remove(session);
29. }
30.
31. public @OnError void onError(Throwable exception, Session session) {
32.     LOGGER.log(Level.INFO, "Erro de conexão com o cliente: {0}", session.getId());
33.     Optional.ofNullable(session)
34.         .map(Session::isOpen)
35.         .filter(Boolean.FALSE::equals)
36.         .ifPresent(webSessions::remove);
37. }
38.
39. public @OnMessage void onMessage(String message, Session session) {
40.     LOGGER.log(Level.INFO, "Mensagem recebida do cliente [{0}]: {1}",
41.         new Object[]{session.getId(), message});
42.     String[] protocol = message.split("\\:");
43.     if("CLIENT".equalsIgnoreCase(protocol[0])) {
44.         if("HW".equalsIgnoreCase(protocol[1])) {
45.             hWSession = Optional.of(session);
46.         } else if("WEB".equalsIgnoreCase(protocol[1])) {
47.             webSessions.add(session);
48.         }
49.     } else {
50.         Action action = ActionFactory.factory(protocol[0]);
51.         LOGGER.log(Level.INFO, "Nova mensagem recebida do cliente [{0}]: {1}",
52.             new Object[]{session.getId(), message});
53.         action.handle(protocol, hWSession, webSessions);
54.     }
55. }
56. }
```

Listagem 6. Código da classe ActionFactory.

```
01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.HashMap;
04. import java.util.Map;
05.
06. public class ActionFactory {
07.
08.     private static final Map<String, Action> actions = new HashMap<>();
09.     private static final Action DEFAULT_ACTION = new NoAction();
10.
11.     static {
12.         actions.put("HW", new HardwareAction());
13.         actions.put("WEB", new WebAction());
14.     }
15.
16.     public static Action factory(String actionPerformed) {
17.         return actions.getOrDefault(actionPerformed, DEFAULT_ACTION);
18.     }
19. }
```

é processada pela condição inicial do método `onMessage()` aqui citado, realizando a identificação dos clientes – Hardware ou Web. Para todas as outras situações em que a mensagem siga o padrão “ID:AÇÃO[:PARAMS]”, existe a transformação da mensagem em uma ação através da chamada `ActionFactory.factory(protocol[0])`. O retorno deste método é uma instância da interface `br.com.pontoclass.iot.websocket.Action`, que será responsável pela interpretação do comando recebido.

Listagem 7. Código da interface Action.

```
01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.List;
04. import java.util.Optional;
05.
06. import javax.websocket.Session;
07.
08. public interface Action {
09.
10.     void handle(String[] protocol, Optional<Session> hWSession,
11.                 List<Session> webSessions);
12. }
```



Listagem 8. Declaração da classe HardwareAction.

```
01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.HashMap;
04. import java.util.List;
05. import java.util.Map;
06. import java.util.Optional;
07. import java.util.logging.Logger;
08. import javax.websocket.Session;
09.
10. public class HardwareAction implements Action {
11.
12.     private static final Logger LOGGER = Logger.getLogger(
13.         HardwareAction.class.getName());
14.     private static Map<String, Command> commands = new HashMap<>();
15.
16.     static {
17.         commands.put("Calibrate", new HWCalibrateCommand());
18.         commands.put("BeginCalibrate", new HWBeginCalibrateCommand());
19.         commands.put("KeepAlive", new HWKeepAliveCommand());
20.         commands.put("HLLost", new HWLostCommand());
21.         commands.put("HWAvaliable", new HWAvaliableCommand());
22.     }
23.
24.     @Override
25.     public void handle(String[] protocol, Optional<Session> hWSession,
26.             List<Session> webSession) {
27.         Command command = commands.get(protocol[1]);
28.         if(command == null) {
29.             LOGGER.warning(String.format("Um comando desconhecido foi solicitado pelo Hardware: [%s]", protocol[1]));
30.         } else {
31.             command.execute(protocol, hWSession, webSession);
32.         }
33.     }
34. }
```

Listagem 9. Declaração da interface Command.

```
01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.List;
04. import java.util.Optional;
05. import javax.websocket.Session;
06.
07. public interface Command {
08.
09.     public void execute(String[] protocol,
10.             Optional<Session> hWSession,
11.             List<Session> webSessions);
12. }
```

Listagem 10. Declaração da classe HWCalibrateCommand.

```
01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.List;
04. import java.util.Optional;
05. import java.util.logging.Logger;
06. import javax.websocket.Session;
07.
08. public class HWCalibrateCommand implements Command {
09.
10.     @Override
11.     public void execute(String[] protocol,
12.             Optional<Session> hWSession,
13.             List<Session> webSessions) {
14.             webSessions.stream()
15.                 .forEach(session -> {
16.                     try {
17.                         session.getBasicRemote()
18.                             .sendText(String.format("Calibrate:%s", protocol[2]));
19.                     } catch (Exception e) {
20.                         Logger.getLogger(this.getClass().getName())
21.                             .warning(String.format("Something went wrong by trying to answer session [%s]: [%s]", session.getId(), e.getMessage()));
22.                     }
23.                 });
24.     }
25. }
```

Nas Listagens 6 e 7 observamos as declarações da classe `br.com.pontoclass.iot.websocket.ActionFactory` e da interface `br.com.pontoclass.iot.websocket.Action`.

Na classe `ActionFactory`, os comportamentos são divididos em ações recebidas do Hardware (HW) e ações recebidas da Web (WEB), pois o front-end pode solicitar ao servidor WebSocket que o medidor de altura realize a calibragem, enquanto o Hardware enviará vários eventos como início de calibragem, execução de calibragem, valores lidos do sensor, entre outros, fornecendo informações sobre todas as etapas necessárias para que o dispositivo realize sua função e com isso o front-end possa dar feedbacks a respeito do estado do sensor.

A `Action` responsável por receber tais eventos é a classe `br.com.pontoclass.iot.websocket.HardwareAction`, que tem sua declaração na Listagem 8. Sua função é a de agrupar os comportamentos relacionados ao hardware, realizando o direcionamento da requisição ao comando correto de acordo com a mensagem recebida. Tais comandos estão separados em objetos do tipo `Command`, associados no bloco estático da classe, como podemos conferir no código das linhas 15 a 21 (ainda na Listagem 8). Neste bloco são associadas todas as ações já vistas no enum `Stategy` e mais a ação "HLLost", que será utilizada para informar ao front-end que a conexão com o hardware foi perdida.

É claro que esta ação não poderia ser enviada pelo hardware, uma vez que com a perda da conexão isso se torna impossível. Por esse motivo, o método `onClose()` da classe `HMeterServer` (Listagem 5) simula o recebimento desta mensagem para que o comando responsável por informar ao front-end tal evento tenha a sua execução garantida.

No código da interface `Action`, a declaração do parâmetro `hwSession` é feita através da utilização do tipo `java.util.Optional<Session>`, a fim de evitar erros com ponteiros nulos (`NullPointerException`), já que o código chamador do método `handle()` pode passar o valor `Optional.empty()`, enquanto a implementação do método pode utilizar o método `hwSession.ifPresent()` para verificar a existência de valor na variável

Optional. Esta prática não somente evita erros com ponteiros, mas também prevê a possibilidade de que o front-end se conecte ao WebSocket antes do hardware, o que implica, nesse caso, em um parâmetro vazio de sessão com o hardware. Nas **Listagens 9 a 14**, a interface **br.com.pontoclass.iot.websocket.Command** e todas as suas implementações utilizadas na classe **HardwareAction** podem ser conferidas.

Podemos observar que todas as classes são implementações da interface **Command** e funcionam de forma muito semelhante. A classe **HWCalibrateCommand**, por exemplo, utiliza a lista **webSessions** para percorrer todas as sessões web ativas com o objetivo de enviar uma mensagem aos clientes informando sobre o evento **Calibrate**, com o seu valor – de calibragem – parametrizado, como podemos ver nas linhas 17 e 18 da **Listagem 10**. Já a classe **HWKeepAliveCommand**, apresentada na **Listagem 12**, utiliza o mesmo método de envio de mensagem às sessões web para informar sobre o evento **KeepAlive**, que indica a altura do corpo que acabou de se posicionar abaixo do sensor. Desta forma, entendemos que, para cada evento recebido do hardware, um *multicast* do mesmo evento é direcionado a todos os clientes web.

Listagem 11. Declaração da classe HWBeginCalibrateCommand.

```
01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.List;
04. import java.util.Optional;
05. import java.util.logging.Logger;
06. import javax.websocket.Session;
07.
08. public class HWBeginCalibrateCommand implements Command {
09.
10.     @Override
11.     public void execute(String[] protocol,
12.                         Optional<Session> hWSession,
13.                         List<Session> webSessions) {
14.         webSessions.stream()
15.             .forEach(session -> {
16.                 try {
17.                     session.getBasicRemote().sendText("BeginCalibrate");
18.                 } catch (Exception e) {
19.                     Logger.getLogger(this.getClass().getName())
20.                         .warning(String.format("Something went wrong by trying to answer
21. session [%s]: [%s]", session.getId(), e.getMessage()));
22.                 }
23.             });
24. }
```

Neste experimento foram mapeados os seguintes eventos de Hardware:

- **Calibrate:** informa a altura máxima que pode ser medida pelo sensor, de acordo com a calibragem realizada;
- **BeginCalibrate:** indica o início da calibragem do hardware, dando a oportunidade aos clientes web de atualizarem a página informando o acontecimento;
- **KeepAlive:** indica o último valor da altura do corpo que acabou de passar pelo sensor;

- **HWLost:** Informa aos clientes web sobre a indisponibilidade do hardware (este é o único evento provocado pelo próprio serviço WebSocket e simula o evento como se fosse do próprio hardware);

Listagem 12. Declaração da classe HWKeepAliveCommand.

```
01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.List;
04. import java.util.Optional;
05. import java.util.logging.Logger;
06. import javax.websocket.Session;
07.
08. public class HWKeepAliveCommand implements Command {
09.
10.     @Override
11.     public void execute(String[] protocol,
12.                         Optional<Session> hWSession,
13.                         List<Session> webSessions) {
14.         webSessions.stream()
15.             .forEach(session -> {
16.                 try {
17.                     session.getBasicRemote()
18.                         .sendText(String.format("KeepAlive:%s", protocol[2]));
19.                 } catch (Exception e) {
20.                     Logger.getLogger(this.getClass().getName())
21.                         .warning(String.format("Something went wrong by trying to answer
22. session [%s]: [%s]", session.getId(), e.getMessage()));
23.                 }
24.             });
25. }
```

Listagem 13. Declaração da classe HWAvailableCommand.

```
01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.List;
04. import java.util.Optional;
05. import java.util.logging.Logger;
06. import javax.websocket.Session;
07.
08. public class HWAvailableCommand implements Command {
09.
10.     @Override
11.     public void execute(String[] protocol,
12.                         Optional<Session> hWSession,
13.                         List<Session> webSessions) {
14.         webSessions.stream()
15.             .forEach(session -> {
16.                 try {
17.                     session.getBasicRemote().sendText("HWAvailable");
18.                 } catch (Exception e) {
19.                     Logger.getLogger(this.getClass().getName())
20.                         .warning(String.format("Something went wrong by trying to answer
21. session [%s]: [%s]", session.getId(), e.getMessage()));
22.                 }
23.             });
24. }
```

- **HAvailable:** informa aos clientes web que o hardware está disponível para ser utilizado e pronto para receber o primeiro pedido de calibragem.

Listagem 14. Declaração da classe HWLostCommand.

```

01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.List;
04. import java.util.Optional;
05. import java.util.logging.Logger;
06. import javax.websocket.Session;
07.
08. public class HWLostCommand implements Command {
09.
10.     @Override
11.     public void execute(String[] protocol,
12.             Optional<Session> hWSession,
13.             List<Session> webSessions) {
14.         webSessions.stream()
15.             .forEach(session -> {
16.                 try {
17.                     session.getBasicRemote().sendText("HWLost");
18.                 } catch (Exception e) {
19.                     Logger.getLogger(this.getClass().getName())
20.                         .warning(String.format("Something went wrong by trying to answer
session [%s]: [%s]", session.getId(), e.getMessage()));
21.                 }
22.             });
23.     }
24. }
```

Visualizando os dados do sensor

Uma vez que a finalidade do módulo web em nosso experimento é fornecer a medida da altura de alguém, precisamos de uma página para visualizar os resultados da leitura do dispositivo. Até agora, no entanto, abordamos apenas a camada back-end do WebSocket e sua comunicação com o hardware. Deste modo, criaremos uma camada front-end que tratará a comunicação com o socket tanto para enviar quanto para receber informações relacionadas ao sensor ultrassônico, permitindo assim a exibição das mudanças de estado detectadas.

Neste exemplo, para controlar a camada de visualização e sua interação com o back-end, utilizaremos o AngularJS. Na **Listagem 15** podemos observar o código HTML com a página que irá exibir a altura do visitante que se posicionar abaixo do sensor.

Essa página é declarada em um JSP no mesmo webapp em que o WebSocket é criado. Nela, os scripts importados mais relevantes a serem observados são *services.js* e *controllers.js*, que podem ser conferidos nas **Listagens 16** e **17**. Nestas listagens estamos declarando, respectivamente, o serviço **\$websocket** responsável pela comunicação com o WebSocket da aplicação web e o controlador **HMeterController**, utilizado para realizar o controle daquilo que é exibido ao usuário na página de visualização do estado do dispositivo, em função dos comandos recebidos pelo serviço **\$websocket**.

O serviço **\$websocket** encapsula toda a complexidade de comunicação do WebSocket, implementando o protocolo simples criado neste artigo em sua função **onMessage()**, enquanto o controlador

Listagem 15. Código da página HTML (JSP) para exibição das informações.

```

01. <%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
02. <!DOCTYPE html>
03. <html ng-app="iot">
04.     <head>
05.         <title page-title></title>
06.         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
07.         <meta name="viewport" content="width=device-width, initial-scale=1.0">
08.
09.         <link rel="stylesheet" href="css/style.css" />
10.
11.         <!-- bower:js -->
12.         <script src="bower_components/jquery/dist/jquery.js"></script>
13.         <script src="bower_components/angular/angular.js"></script>
14.         <script src="bower_components/angular-ui-router/release/
angular-ui-router.js"></script>
15.         <!-- endbower -->
16.
17.         <script src="js/script.js"></script>
18.         <script src="js/config.js"></script>
19.         <script src="js/services.js"></script>
20.         <script src="js/controllers.js"></script>
21.     </head>
22.
23.     <body ng-controller="HMeterController">
24.         <!--[if lt IE 7]>
25.         <p class="browsehappy">Você está utilizando um navegador
desatualizado</strong>.
Por favor, <a href="http://browsehappy.com/">actualize o seu navegador</a>
para uma melhor experiência de usuário.</p>
26.         <![endif]-->
27.
28.         <!-- Wrapper-->
29.
30.         <div ui-view="main"></div>
31.
32.
33.         <!-- End wrapper-->
34.     </body>
35. </html>
```

HMeterController realiza o controle da camada de visualização de acordo com os comandos recebidos através do serviço, de acordo com a mudança de estado do sensor. Para cada alteração neste estado, o servidor WebSocket envia uma mensagem ao serviço AngularJS **\$websocket**, atualizando o seu estado no front-end. Repare que, na declaração do **\$websocket**, temos métodos como **beginCalibrate()**, **calibrate()**, **keepAlive()**, entre outros, sendo declarados. Esses métodos encapsulam as funções reais através da variável **callback**, fornecida ao serviço pelo controlador na linha 54 da **Listagem 17**. Com isso, todas as mudanças de estado na página serão regidas pelas chamadas de funções declaradas nessa variável **callback** e acionadas pelo **\$websocket** através do mecanismo de recebimento de mensagens.

Na **Figura 1** temos o estado inicial da visualização do medidor de altura no navegador.

Enquanto o sensor estiver desconectado do WebSocket, a camada de visualização não sofrerá nenhuma alteração.

Primeiros passos no mundo da Internet das Coisas – Parte 2

Listagem 16. Declaração do serviço AngularJS \$websocket.

```
01.'use strict';
02.
03.angular.module('iot')
04..service('$websocket', [function(){
05.    return {
06.        hmeter: null,
07.        callbacks: null,
08.        calibration: new Number("0.0"),
09.        restartCalibration: function() {
10.            this.calibration = new Number("0.0");
11        },
12.        init: function(callbacks) {
13.            this.callbacks = callbacks;
14.            this.hmeter = new WebSocket("ws://localhost:8080/websocket/hmeter");
15.            this.hmeter.onopen = this.onOpen;
16.            this.hmeter.onclose = this.onClose;
17.            this.hmeter.onmessage = this.onMessage;
18.            this.hmeter.socketHandler = this;
19.            var _this = this;
20.            angular.element(document).bind("HSHC", function(evt) {
21.                _this.hmeter.send("CLIENT:WEB");
22.            });
23.            angular.element(document).bind("BGNC", function(evt) {
24.                _this.sendCalibrateMessage();
25.            });
26.        },
27.        beginCalibrate: function() {
28.            this.callbacks.beginCalibrate();
29.        },
30.        calibrate: function(height) {
31.            this.callbacks.calibrate(height);
32.            this.calibration = new Number(height.replace(",","."));
33.        },
34.        keepAlive: function(height) {
35.            if(new Number(height.replace(",",".")) < this.calibration - 0.03) {
36.                this.callbacks.keepAlive((this.calibration - new Number((height.replace(",","."))).toFixed(2));
37.            } else {
38.                this.callbacks.waitForVisitors();
39.            }
40.        },
41.        hwLost: function(){
42.            this.callbacks.hwLost();
43.        },
44.        connectionClosed: function() {
45.            this.callbacks.connectionClosed();
46.        },
47.        sendCalibrateMessage: function() {
48.            this.hmeter.send("WEB:Calibrate");
49.        },
50.        onOpen: function() {
51.            console.log("conexão aberta...");
52.            angular.element(document).trigger('HSHC');
53.        },
54.        onClose: function() {
55.            console.log("conexão encerrada...");
56.            this.socketHandler.connectionClosed();
57.        },
58.        onMessage: function(evt) {
59.            console.log("mensagem recebida: " + evt.data);
60.            var protocol = evt.data.split(":");
61.            if("BeginCalibrate" == protocol[0]) {
62.                this.socketHandler.beginCalibrate();
63.            } else if("Calibrate" == protocol[0]) {
64.                this.socketHandler.calibrate(protocol[1]);
65.            } else if("KeepAlive" == protocol[0]) {
66.                if(this.socketHandler.calibration == null ||
67.                    this.socketHandler.calibration == 0) {
68.                    angular.element(document).trigger('BGNC');
69.                } else {
70.                    this.socketHandler.keepAlive(protocol[1]);
71.                }
72.            } else if("HWLost" == protocol[0]) {
73.                this.socketHandler.hwLost();
74.            } else if("HWAvalable" == protocol[0]) {
75.                this.socketHandler.sendCalibrateMessage();
76.            }
77.        }
78.    };
79.    this.callbacks.waitForVisitors();
80.    this.socketHandler = new SocketHandler();
81.    this.socketHandler.connectionClosed();
82.    this.socketHandler.beginCalibrate();
83.    this.socketHandler.calibrate("0.0");
84.    this.socketHandler.keepAlive("0.0");
85.    this.socketHandler.sendCalibrateMessage();
86.    this.callbacks.hwLost();
87.    this.callbacks.connectionClosed();
88.});
89.});
```

Listagem 17. Declaração do controlador AngularJS HMeterController.

```
01.'use strict';
02.
03.angular.module('iot')
04..controller('HMeterController', ['$scope',
05.    '$state',
06.    '$websocket',
07.    function($scope, $state, $websocket) {
08.        $scope.sendCalibrate = function (){
09.            $websocket.restartCalibration();
10.        };
11.        var callback = {
12.            beginCalibrate: function() {
13.                $scope.title = "Calibrando...";
14.                $scope.message = "Aguarde...";
15.                $scope.cssColor = "yellow";
16.                $scope.$apply();
17.            },
18.            calibrate: function(height) {
19.                $scope.title = "Equipamento calibrado!";
20.                $scope.message = "Altura Máxima:" + height;
21.                $scope.cssColor = "blue";
22.                $scope.$apply();
23.            },
24.            keepAlive: function(height) {
25.                $scope.title = "Olá visitante!";
26.                $scope.message = "Sua altura:" + height;
27.                $scope.cssColor = "green";
28.                $scope.$apply();
29.            },
30.            hwLost: function(){
31.                $scope.title = "Impossível medir a altura";
32.                $scope.message = "Fora de Serviço";
33.                $scope.cssColor = "red";
34.                $scope.$apply();
35.            },
36.            connectionClosed: function() {
```

Continuação: Listagem 17. Declaração do controlador AngularJS HMeterController.

```
37.   $scope.title = "Impossível medir a altura!";
38.   $scope.message = "Fora de Serviço";
39.   $scope.cssColor = "red";
40.   $scope.$apply();
41. },
42. waitForVisitors: function() {
43.   $scope.title = "Aguardando por visitantes!";
44.   $scope.message = "Aguardando...";
45.   $scope.cssColor = "gray";
46.   $scope.$apply();
47. }
48. );
49. angular.element(document).ready(function() {
50.   $scope.title = "Procurando o serviço...";
```



```
51.   $scope.message = "Aguarde...";
52.   $scope.cssColor = "yellow";
53.   try {
54.     $websocket.init(callback);
55.   } catch(e) {
56.     console.log(e);
57.     $scope.title = e.message;
58.     $scope.message = "#ERRO";
59.     $scope.cssColor = "red";
60.   }
61.   $scope.$apply();
62. });
63.]);
```



Figura 1. Visualização inicial do medidor de altura



Figura 2. Visualização de dispositivo calibrando



Figura 3. Visualização de altura máxima lida pelo dispositivo

No entanto, uma vez que a aplicação com o Sketch seja executada e consiga se comunicar com o WebSocket, uma mensagem é enviada do hardware para o front-end informando que o dispositivo está disponível ("HW-Available").

O front-end, então, envia uma mensagem ao WebSocket solicitando a calibragem do dispositivo. Deste modo o diálogo é estabelecido, de forma que em todas as interações a camada de visualização seja atualizada para informar aos usuários sobre o que está acontecendo. A atualização seguinte no front-end (vide **Figura 2**) informa ao usuário que o dispositivo está sendo calibrado, e depois da calibragem, é enviada a mensagem com a altura máxima que o dispositivo consegue informar (vide **Figura 3**).

Após a calibragem do dispositivo, o evento `KeepAlive` entra em loop até que alguma outra ação seja solicitada. Com isso, a aplicação no navegador fica em modo de espera, aguardando o posicionamento de alguém abaixo do dispositivo (vide **Figura 4**) e após a passagem do corpo pelo sensor, a tela exibida informa a altura do indivíduo (vide **Figura 5**).

Ainda na camada de visualização, o usuário tem a opção de clicar na mensagem no canto inferior direito para solicitar que o sensor inicie novamente o processo de calibração, útil quando o sensor se perde por algum motivo. Nesta ação ocorre o envio de uma mensagem do cliente WebSocket do navegador ao servidor. Assim, através da chamada à função `sendCalibrateMessage()` do serviço Angular `$websocket`, a mensagem "WEB:Calibrate" é enviada.

De volta ao servidor, na classe **ActionFactory** vista anteriormente na **Listagem 7**, podemos verificar que ela também associa uma **Action** às mensagens do tipo web por meio da classe **br.com.pontoclass.iot.websocket.WebAction**, apresentada na **Listagem 18**. Essa classe possui apenas um comando: **WebCalibrateCommand**, exposto na **Listagem 19**, responsável por fazer o caminho inverso da comunicação, enviando ao hardware o pedido de Calibração. Isto inicia novamente todo o fluxo de mensagens, resultando na nova calibragem do dispositivo. Ademais, esse processo também permite que a visualização do estado do dispositivo possa ser atualizada no navegador.

Carregando o projeto no Raspberry Pi

Após todo o processo de desenvolvimento e testes em um computador comum, é hora de instalar o servidor WebSocket como uma aplicação web no Raspberry Pi e também o Sketch JArduino, como um programa que executará ininterruptamente por meio da execução da classe **Bootstrap**, vista na **Listagem 3**. Para tanto, é necessário que seja instalada a versão do Java Embedded Suite no dispositivo, uma vez que esta versão do Java é apropriada para a arquitetura do Raspberry Pi.

Se o Sistema Operacional instalado no dispositivo for uma distribuição Linux, é muito provável que a instalação do JDK possa ser feita de forma simples, através da execução do comando *sudo apt-get install oracle-java8-jdk*. Em seguida, todo o processo de instalação do container web pode ser feito como se fosse em uma distribuição Linux de um computador qualquer. É possível também realizar o download do pacote JDK apropriado diretamente no site da Oracle e acompanhar os passos de instalação quando o Sistema Operacional não possuir gerenciadores de pacotes ou quando os mesmos estiverem desatualizados.

Após a instalação do JDK e do container Web, basta seguir o processo de instalação do WebSocket (geralmente a instalação é tão simples quanto jogar o pacote “.war” da aplicação na pasta *webapp* do servidor) e iniciar o programa .jar controlador do Arduino (normalmente a execução se dá através do comando *java -jar nomeDoPacote.jar*). Este último procedimento necessita da conexão direta entre o Arduino e o Raspberry Pi para que não haja falha na sua inicialização. Tal conexão é realizada via porta USB e a alimentação elétrica desta porta já é suficiente para a execução do Arduino. Já o Raspberry Pi precisará de uma fonte externa de eletricidade de pelo menos 2A (dois amperes) na versão mais nova do dispositivo (Raspberry Pi 2 Model B, até a data de escrita deste artigo). Essa fonte elétrica pode ser adquirida facilmente (provavelmente no mesmo local de venda aonde o Raspberry Pi tenha sido comprado) e ser ligada ao dispositivo através de sua entrada universal.



Figura 4. Visualização de dispositivo aguardando por visitantes



Figura 5. Visualização da altura do visitante

Listagem 18. Código da classe WebAction.

```
01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.HashMap;
04. import java.util.List;
05. import java.util.Map;
06. import java.util.Optional;
07. import java.util.logging.Logger;
08.
09. import javax.websocket.Session;
10.
11. public class WebAction implements Action {
12.
13.     private static final Logger LOGGER = Logger.getLogger(
14.             WebAction.class.getName());
15.
16.     static {
17.         commands.put("Calibrate", new WebCalibrateCommand());
18.     }
19.
20.     @Override
21.     public void handle(String[] protocol,
22.                         Optional<Session> hWSession,
23.                         List<Session> webSession) {
24.         Command command = commands.get(protocol[1]);
25.         if(command == null) {
26.             LOGGER.warning(String.format("Unknown command was asked by" +
27.                     " the web client: [%s]", protocol[1]));
28.         } else {
29.             command.execute(protocol, hWSession, webSession);
30.         }
31.     }
32. }
```

Como podemos verificar, a utilização de Java para controle de componentes eletrônicos já é uma realidade e está madura o suficiente para que qualquer desenvolvedor bem-aventurado a explore. Esta possibilidade abre espaços para que o programador Java possa mergulhar no universo da Internet das Coisas (IoT) sem a necessidade de aprender novas linguagens ou eletrônica avançada; bastam alguns sensores em mãos, um dispositivo controlador, Java e um pouco de criatividade para que o leitor possa se tornar um inventor de novos dispositivos utilitários que mais tarde possam até se tornar produtos inovadores.

Listagem 19. Declaração da classe WebCalibrateCommand.

```

01. package br.com.pontoclass.iot.websocket;
02.
03. import java.util.List;
04. import java.util.Optional;
05. import java.util.logging.Logger;
06.
07. import javax.websocket.Session;
08.
09. public class WebCalibrateCommand implements Command {
10.
11.     @Override
12.     public void execute(String[] protocol,
13.             Optional<Session> hWSession,
14.             List<Session> webSession) {
15.         hWSession.ifPresent(session -> {
16.             try {
17.                 session.getBasicRemote().sendText("Calibrate");
18.             } catch (Exception e) {
19.                 Logger.getLogger(this.getClass().getName())
20.                     .warning("Something went wrong by trying " +
21.                             "to send the Calibrate message to the Hardware...");
22.             }
23.         });
24.     }
25. }
```

Com os conhecimentos apresentados no artigo, o aprofundamento nos detalhes não abordados se tornará mais fácil e intuitivo. O leitor poderá buscar a compreensão de quais são os próximos passos para transformar um protótipo em um produto acabado e terminar descobrindo-se como criador de uma nova startup que pode nascer a partir de uma ideia com testes em protótipos; mas para seguir em frente é preciso profissionalizar e otimizar não somente o processo, como também o próprio modelo (dispositivo).

Muitas destas prototipações, inclusive como a realizada neste artigo, podem ser feitas sem a necessidade de Arduino e Sketches, pois é possível comunicar-se diretamente com a GPIO do próprio Raspberry Pi configurando outro modelo de comunicação e controle de sensores via Java de forma mais simplificada, quando a solução não requerer a utilização de muitos pinos. O emprego do Arduino neste artigo serviu para mostrar que mesmo placas micro controladas através de códigos nativos, que não utilizam Java em seu Firmware, podem ser facilmente integradas a soluções

desta linguagem através de bibliotecas intermediárias, como o JArduino.

A aplicação do JArduino nos projetos de Internet das Coisas facilita bastante a descoberta deste novo nicho para os programadores Java, mas pode se tornar dispensável com o tempo, quando o desenvolvedor perceber que mesmo as aplicações intermediárias, escritas em outras linguagens, podem ser facilmente escritas para se integrar através de protocolos criados pelo próprio desenvolvedor para a troca de dados entre dispositivos.

Autor



Rômero Ricardo de Sousa Pereira

javeiro@uninove.edu.br e rpereira@atex.com



É formado em Ciência da Computação pela Universidade Nove de Julho, desde 2009, onde atuou como professor em cursos extensivos de Java durante dois anos. Possui nove anos de experiência como desenvolvedor de sistemas, sendo oito atuando com Java, passando por experiências desde o desenvolvimento de aplicações Standalone com Java SE, até soluções Web e Enterprise de alta demanda com Java EE. Atualmente é consultor Java na Atex Digital Media do Brasil. Possui as certificações OCJP e OCWCD.

Links:

Designing the Internet of Things (Livro).

<http://www.wiley.com/WileyCDA/WileyTitle/productCd-111843062X.html>

Explained: The ABCs of the Internet of Things.

<http://www.computerworld.com/article/2488872/emerging-technology-explained-the-abcs-of-the-internet-of-things.html>

JSR 356, Java API for WebSocket.

<http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html>

Micro-Electro-Mechanical Systems.

<http://www.tecmundo.com.br/nanotecnologia/3254-o-que-sao-mems-.htm>

Como funcionam os sensores ultrassônicos (ART691).

<http://www.newtoncbraga.com.br/index.php/como-funciona/5273-art691>

IoT: Do you really should have knowledge in electronics?

<http://blogs.msdn.com/b/cdndevs/archive/2015/04/17/iot-do-you-really-should-have-knowledge-in-electronics.aspx>

Eletrônica Didática - Protoboard.

<http://www.eletronicadidatica.com.br/protoboard.html>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Dominando o Selenium Web Driver na prática

Crie testes automatizados paralelos integrando à API do TestNG, e converta-os para várias linguagens de programação

OSelenium é uma ferramenta utilizada para automatização de testes de sistemas que permite ao usuário reproduzi-los rapidamente no ambiente real da aplicação, em função da sua integração direta com o navegador. Através de uma extensão podemos criar os scripts de testes de forma simples e utilizar nas mais diversas linguagens de programação. No geral, o script é gerado em HTML, porém pode ser exportado e editado para testes em Java, C#, etc.

Neste artigo trataremos de entender como funciona o Selenium Web Driver, um recurso adicionado recentemente ao framework que se acopla à API do Selenium, permitindo a execução direta de testes via código fonte no servidor. Ele faz uso de uma API JavaScript extensiva que, aliada ao motor JavaScript do próprio navegador, permite acesso ao documento DOM completo da página HTML, assim como a manipulação de suas propriedades e funções. Em outras palavras, ele permite traduzir o código de servidor para o código de cliente e executar nossos testes automatizados no universo front-end dos diversos browsers suportados.

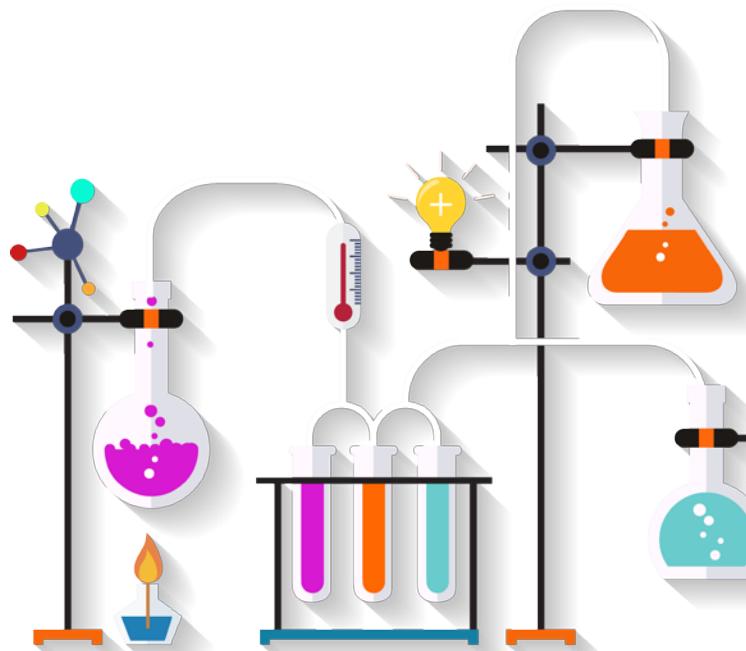
Além disso, uma outra grande vantagem é a possibilidade de execução das suítes de testes de forma paralelizada, criando várias threads e executando-as concorrentemente, algo que não temos na versão gráfica do Selenium. Dessa forma, os desenvolvedores podem criar testes unitários e atrelá-los aos automatizados, dentro de loops que abrem várias instâncias dos drivers dos navegadores e permitem testar diferentes recursos das aplicações de forma muito mais rápida e performática.

Criando um círculo de feedback rápido

Muitas pessoas reclamam do quanto tempo levam para executar todos os seus testes, que vão de algumas horas a alguns dias. Vamos ver como podemos acelerar

Fique por dentro

Este artigo é útil a todo desenvolvedor de testes automatizados que deseja conhecer mais a fundo os recursos do Selenium Web Driver, os quais, dentre outras coisas, permitem comunicação direta com as APIs de drivers dos diferentes navegadores, bem como fácil integração com frameworks de testes unitários como o jUnit e o TestNG. Veremos desde a configuração do ambiente, criação dos primeiros testes, até o gerenciamento de suítes completas em conformidade com o modelo de execução paralela, que possibilita acesso direto aos recursos do DOM HTML, além de converter todo o conteúdo para diversas linguagens de programação.



as coisas e colocar aqueles que estamos escrevendo em execução de forma regular e rápida.

O segundo problema que podemos nos deparar em trazer outras pessoas para executar seus testes é a dificuldade para configurar o projeto para trabalhar em sua máquina e o grande esforço para que eles façam o mesmo.

Isso cria um círculo rápido de feedback, isto é, se os seus desenvolvedores estão rodando todos os testes antes de cada check-in eles vão saber as alterações feitas no código antes que o mesmo deixe a sua máquina.

Facilitando os testes para desenvolvedores

O ideal é que nossos testes sejam executados sempre que alguém inserir novos códigos ao repositório de código central. Partindo desse princípio, alguém pode simplesmente ir à base de código, executar um comando e ter todos funcionando.

Nós vamos fazer isso facilmente utilizando o **Apache Maven**, um poderoso framework para integração e geração de builds de projetos, além de gerenciador de dependências (o usaremos para mapear as dependências do Selenium Web Driver no tutorial).

Todavia, o Maven não é a única solução para este problema (por exemplo, o Gradle está rapidamente ganhando popularidade e aceitação da comunidade), mas possui uma probabilidade maior de utilização na área, já que a maioria dos desenvolvedores o terão utilizado em algum momento de sua carreira.

Um dos principais pontos positivos é que ele incentiva os desenvolvedores a usar um padrão na estrutura do projeto que torna mais fácil navegar ao redor do código fonte; ele também torna mais fácil a conexão em um sistema de CI (*Continuous Integration*, ou Integração Contínua), como o *Jenkins* ou o *TeamCity*, por exemplo.

Ao fazer uso do Maven somos capazes de verificar o nosso código no teste, simplesmente executando o comando *mvn clean install* em uma janela do terminal. Este comando irá baixar automaticamente todas as dependências mapeadas para o projeto, configurar o caminho de classes e executar todos os testes.

Files

Maven is distributed in several formats for your convenience. Simply pick a ready-made binary distribution archive and follow the [instructions](#) in the Maven yourself.

In order to guard against corrupted downloads/installations, it is highly recommended to [verify the signature](#) of the release bundles.

Link	Checksum
Binary tar.gz archive	apache-maven-3.3.9-bin.tar.gz
Binary zip archive	apache-maven-3.3.9-bin.zip
Source tar.gz archive	apache-maven-3.3.9-src.tar.gz
Source zip archive	apache-maven-3.3.9-src.zip

- [Release Notes](#)

Figura 1. Selecionando arquivo de download do Maven

Construindo projeto de teste com o Maven

Para fazer o devido uso do Maven precisamos primeiramente baixá-lo no site oficial. Portanto, acesse o link disponível da seção **Links** e efetue o download do arquivo em formato zip, dentro da categoria **Link** do mesmo site (veja na **Figura 1** o arquivo correto a baixar).

Se estiver utilizando o sistema operacional Linux, pode fazer a mesma instalação usando a ferramenta *apt-get*, através do seguinte comando:

```
sudo apt-get install maven
```

Ou, se estiver utilizando um Mac, pode fazer o mesmo procedimento com o utilitário *homebrew*, via comando:

```
brew install maven
```

O próximo passo é adicionar o diretório `/bin` à variável de ambiente PATH do SO. Para isso acesse o menu Windows e digite “variáveis de ambiente”, selecione a opção “Editar Variáveis de Ambiente do Sistema” e, na janela que abrir, clique no botão “Variáveis de Ambiente...”. Em seguida, na nova janela, dentro das opções da categoria “Variáveis do sistema” procure pela variável *Path*, clique em “Editar” e no final do conteúdo insira um ponto-e-vírgula (`:`). Adicione também o caminho da pasta `/bin` completo de onde descompactamos o zip do Maven, tal como temos na **Figura 2**. Clique em OK três vezes e pronto, seu ambiente estará pronto para trabalhar com a ferramenta.

Depois de ter o Maven instalado e funcionando, vamos começar nosso projeto Selenium preenchendo o nosso arquivo básico de configuração do POM (*pom.xml*). O leitor pode ficar à vontade para usar qualquer IDE de sua preferência ou configurar os arquivos de forma manual no Windows Explorer. Para este artigo faremos uso da IDE Eclipse (vide seção **Links** para download), por sua simplicidade em se integrar com o Maven e manusear seus recursos.

Dentro da IDE selecione a opção *File > New > Java Project*, dê um nome ao seu projeto (*selenium-web-driver*) e clique em *Finish*. Isso será o suficiente para criar uma estrutura básica para executar as bibliotecas do Selenium Web Driver, assim como as suas dependências. Entretanto, antes temos que converter nosso projeto para ser reconhecido como um projeto Maven no ambiente. Clique com o botão direito sobre o projeto e selecione a opção *Configure > Convert to Maven Project*. Uma janela com as propriedades do POM aparecerá para informar os ids de artefato e grupo, além do tipo final de empacotamento que usaremos. Preencha todas as informações tal como mostramos na **Figura 3** e clique em *Finish*.

Dominando o Selenium Web Driver na prática

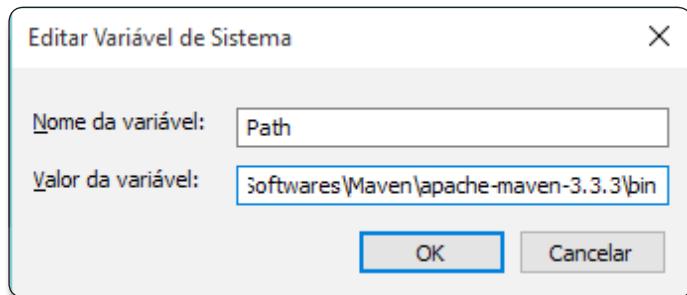


Figura 2. Configurando variável de ambiente Path para o Maven

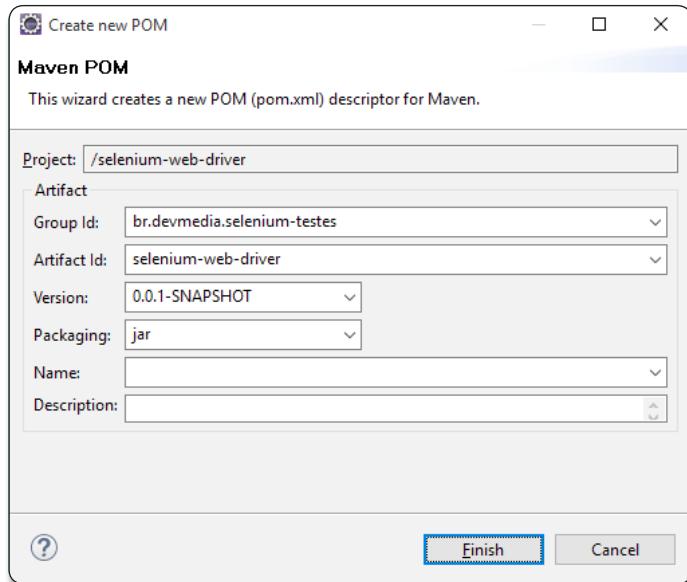


Figura 3. Configurando parâmetros do POM no Maven

Para efetuar os testes automatizados precisamos de um framework auxiliar de testes unitários para efetuar as operações de lógica de negócio na camada do servidor. Existem dois principais frameworks que podemos usar que se comunicam muito bem com o Selenium Web Driver: o **jUnit** (o mais famoso) e o **TestNG** (mais simples e didático). Optaremos por usar o **TestNG**, principalmente por ser mais fácil de obter o famoso *up-and-running out-of-the-box* (ambiente configurado e funcional de forma rápida), sendo o **jUnit** mais extensível para essa opção. Quando o assunto é Selenium, o **TestNG** é sempre tido como uma melhor opção nas listas de discussão da comunidade, com várias *threads* de perguntas sobre o assunto.

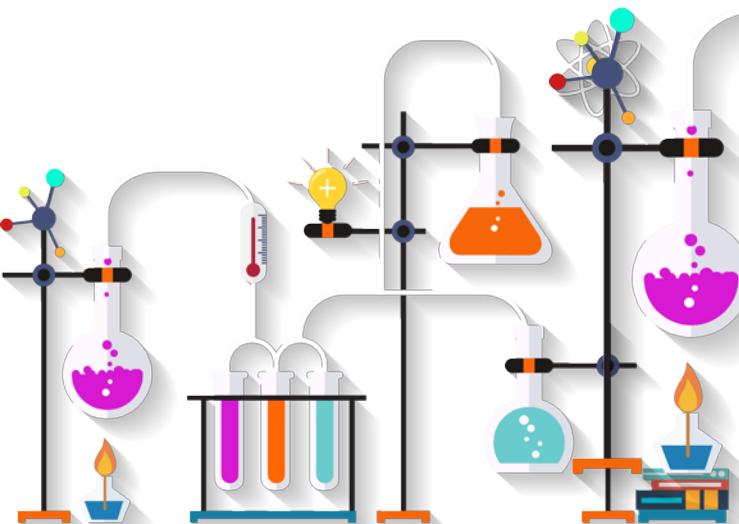
Portanto, para começar, vamos modificar o conteúdo do nosso arquivo gerado do pom.xml para o demonstrado na **Listagem 1**.

Temos aqui é um código XML puro do Maven. O *groupId*, *artifactId*, *version* e propriedades estão sujeitos às convenções de nomenclatura padrão:

- **groupId**: Este deve ser um domínio\controle a ser introduzido ao contrário (como um nome de pacote no Java ou a URL de um site ao contrário);
- **artifactId**: Este é o nome que será atribuído ao seu arquivo executável no final, por isso lembre-se de torná-lo no que você quer que seja chamado.

Listagem 1. Conteúdo inicial do arquivo pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>br.devmedia.selenium-testes</groupId>
<artifactId>selenium-web-driver</artifactId>
<version>0.0.1-SNAPSHOT</version>
<build>
  <sourceDirectory>src</sourceDirectory>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
</build>
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <!-- Versão das dependências -->
  <selenium.version>2.48.2</selenium.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-remote-driver</artifactId>
    <version>${selenium.version}</version>
  </dependency>
  <dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.9.9</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

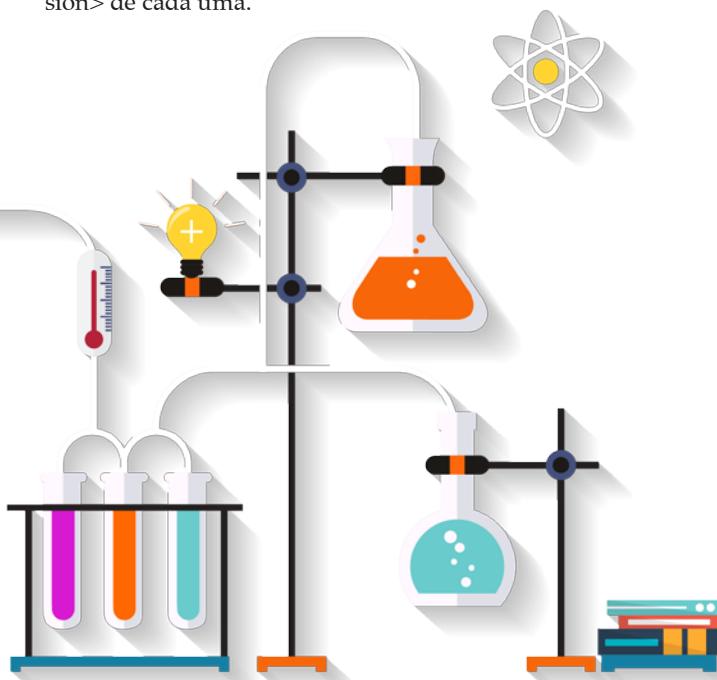


- **version:** Esta deve ser sempre um número com `-SNAPSHOT` (uma imagem temporária) anexado ao fim. Isso mostra que ele atualmente é um trabalho em processo.

A tag `<build>` foi gerada automaticamente com as informações referentes à versão do compilador do Maven, bem como o nível do JDK que será usado pelo projeto. É importante que o leitor não modifique nenhuma dessas propriedades sob o risco de tornar o projeto não-funcional. A tag `<properties>` configura uma lista de propriedades em forma de constantes que podem ser usadas nas demais configurações. Nela incluímos a propriedade que define o `encoding` (codificação de caracteres) dos nossos arquivos de código (UTF-8), bem como a versão das bibliotecas do Selenium Web Brower (conhecido comumente como *SeleniumHQ*) que incluem as bibliotecas de suporte à API do Java, bem como as libs do driver remoto do Selenium em si.

Para que o nosso projeto funcione em conformidade com o encoding declarado no pom.xml precisamos converter sua formatação para o mesmo. Para isso, clique com o botão direito no projeto e selecione a opção *Properties*. Em *Resource* mude a opção *Text file encoding* para *Other* e a opção da combo para *UTF-8*.

É muito importante que o leitor se atente às versões das bibliotecas que está usando, e isso pode ser identificado através do `groupId` daquele lib em questão. Por exemplo, perceba que as dependências declaradas em seguida ao *selenium-java* e do *selenium-remote-driver* tem em seu atributo `<groupId>` o mesmo valor. Isso acontece porque ambas pertencem ao mesmo projeto Java publicado na central online de dependências do Maven, logo, sempre que uma versão nova for lançada do projeto (`groupId`), ambas as bibliotecas também terão suas versões atualizadas. É por isso que criamos uma propriedade em nível global e a inserimos no atributo `<version>` de cada uma.



É provável que ao fazer este tutorial uma versão mais recente que a 2.48.2 já tenha sido lançada. Para checar isso, basta acessar o site do *mvnrepository* (seção **Links**) e pesquisar pelo `groupId` em questão. A mesma lógica vale para a biblioteca do TestNG, que pertence ao grupo `org.testng`.

O atributo `<scope>` configura qual nível daquela API será usado, isto é, em qual ambiente do Java. No caso testes, suas classes se farão úteis, assegurando que as dependências só sejam carregadas no `classpath` quando os mesmos forem executados.

Salve o arquivo e aguarde até que o Maven faça o build do projeto, baixando todas as dependências (precisaremos de uma conexão estável com a internet, sem proxy, para efetuar este passo). Para verificar se todas as libs foram baixadas com sucesso, clique com o direito sobre o projeto e selecione *Build Path > Configure Build Path....* Na aba *Libraries > Maven Dependencies*, veja a lista de bibliotecas.

Nota

O leitor talvez se questione em relação à grande quantidade de bibliotecas adicionadas ao projeto, quando só configuramos duas dependências no pom.xml. Isso acontece porque algumas libs têm outras dependências próprias, o que acaba gerando um ciclo de dependência, resultando nessa quantidade de libs (49 no total). Essa é mais uma vantagem em usar o Maven, já que ele gerencia tudo isso sozinho e de forma automática.

Precisamos ainda organizar nossos pacotes de código fonte para dividir o que será classe de fonte comum e classe de teste. Para isso, acesse novamente o menu de *Build Path* e, na aba *Source*, clique sobre a pasta `src` apresentada, em seguida no botão *Remove*. Depois, clique em *Add Folder... > Create New Folder...* e dê o nome `src/main/java` ao mesmo, clicando em *OK* para finalizar. Faça o mesmo para o pacote `src/test/java`. No final, teremos dois novos diretórios de código fonte que poderão ser usados para separar suas classes. Quando fizer essa configuração, o Maven passará a dar um erro informando que não consegue encontrar mais o diretório de código fonte do projeto. Isso se deve à seguinte linha que foi inserida automaticamente quando da criação do projeto:

```
<sourceDirectory>src</sourceDirectory>
```

Remova-a e o Maven voltará a funcionar normalmente.

Já temos então a base do nosso projeto Selenium. O próximo passo é criar um teste básico que possamos executar usando o Maven. Comece então criando uma nova classe de nome `PrimeiroTeste.java`, clicando com o botão direito na pasta `src/test` e selecionando *New > Class*. Preencha também o campo *Package* com o valor `br.edu.devmedia.selenium`. Insira o conteúdo da **Listagem 2** à mesma.

Repare que o Selenium tem suas estruturas semelhantes às da linguagem HTML usada no Selenium Web Brower (observe através do plugin do Firefox ou de outras implementações). Os dois métodos de teste, efetivamente, estão no fim da classe anotados com a `annotation` do TestNG `@Test` (atente-se para importar o pacote correto desta classe – `org.testng.annotations.Test` – pois

Dominando o Selenium Web Driver na prática

Listagem 2. Primeiro teste com o Selenium Web Driver.

```
package br.edu.devmedia.selenium;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.ExpectedCondition;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.testng.annotations.Test;

public class PrimeiroTeste {

    private void exemploGoogleQuePesquisaPor(final String stringPesquisa) {
        WebDriver driver = new FirefoxDriver();
        driver.get("http://www.google.com");

        WebElement campoPesquisado = driver.findElement(By.name("q"));
        campoPesquisado.clear();
        campoPesquisado.sendKeys(stringPesquisa);

        System.out.println("O título da página é: " + driver.getTitle());
        campoPesquisado.submit();
    }

    @Test
    public void googleExemploQueijo() {
        exemploGoogleQuePesquisaPor("Queijo!");
    }

    @Test
    public void googleExemploLeite() {
        exemploGoogleQuePesquisaPor("Leite!");
    }
}
```

o jUnit, bem como outros frameworks, usam a mesma). Ambos fazem uma chamada ao método `exemploGoogleQuePesquisaPor()` passando uma String com valores distintos.

O objetivo do teste, basicamente, é simular uma pesquisa à página do `google.com` (que funciona com requisições HTTP do tipo GET) passando no parâmetro “p” (reconhecido pela engina do Google) o valor a ser pesquisado.

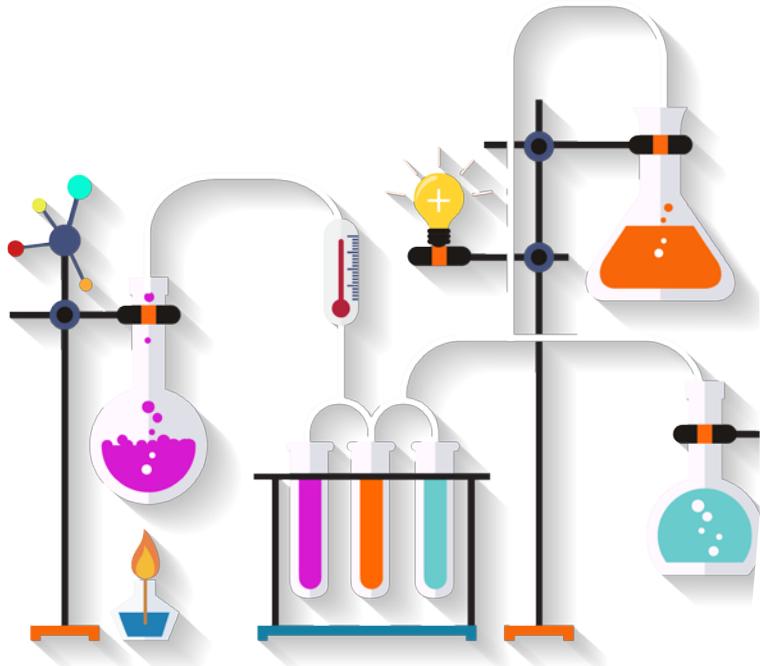
O método `exemploGoogleQuePesquisaPor()` recebe a String de pesquisa, instancia o driver do Firefox (via classe `FirefoxDriver`, que vem dentro das libs do Selenium importadas) e submete a página a qual o mesmo deve acessar “virtualmente”. Em seguida, criamos um objeto do tipo `WebElement` que busca na página aberta um elemento HTML via regra de condição feita no método `findElement()` (no nosso caso procuramos o elemento pelo atributo `name` dele; o valor “q” se refere ao nome do campo de texto da caixa de pesquisa na página do Google). Uma vez com o elemento em mãos, podemos modificar seu valor através do método `sendKeys()`, bem como dar um `Enter` no mesmo via método `submit()` (neste caso, ele não é aplicado somente ao input de tipo `submit`, mas qualquer input que esteja dentro de um formulário na página).

A implementação seguinte trata de um tempo de espera de 10 segundos (via classe `WebDriverWait`), o suficiente para que a requisição retorne com a resposta. E quando isso acontecer, pegamos nosso objeto de driver e modificamos o título (`tag <title>`) para o valor da String enviada na pesquisa. Assim, podemos imprimir no Console da IDE o novo valor do mesmo. Não esqueça de sempre finalizar o driver (fechar o navegador), via método `quit()`.

Para executar esse exemplo não basta tentar executar a classe como uma `Java Application` convencional porque não temos o método `main()`. Como se trata de testes unitários precisamos configurar o Maven para fazer o trabalho. Portanto, clique com o botão direito sobre o projeto e selecione a opção `Run As > Run`

Configurations.... Depois, clique com o direito na opção `Maven Build` e selecione `New`. Na janela que abrir na lateral, preencha o campo `Name` com o valor `Selenium Testes` e o campo `Goals` com o valor `clean install` (comando para limpar e instalar o projeto no Maven). Em seguida, clique no botão `Browse Workspace...`, selecione o nosso projeto e clique em `OK` (veja na **Figura 4** como suas configurações devem ficar).

Pronto, agora é só clicar em `Run` e o projeto será compilado e o build gerado. Sempre que quiser executar novamente basta clicar no botão de seta ao lado do `Run` no topo da IDE e selecionar a opção `Selenium Testes`.



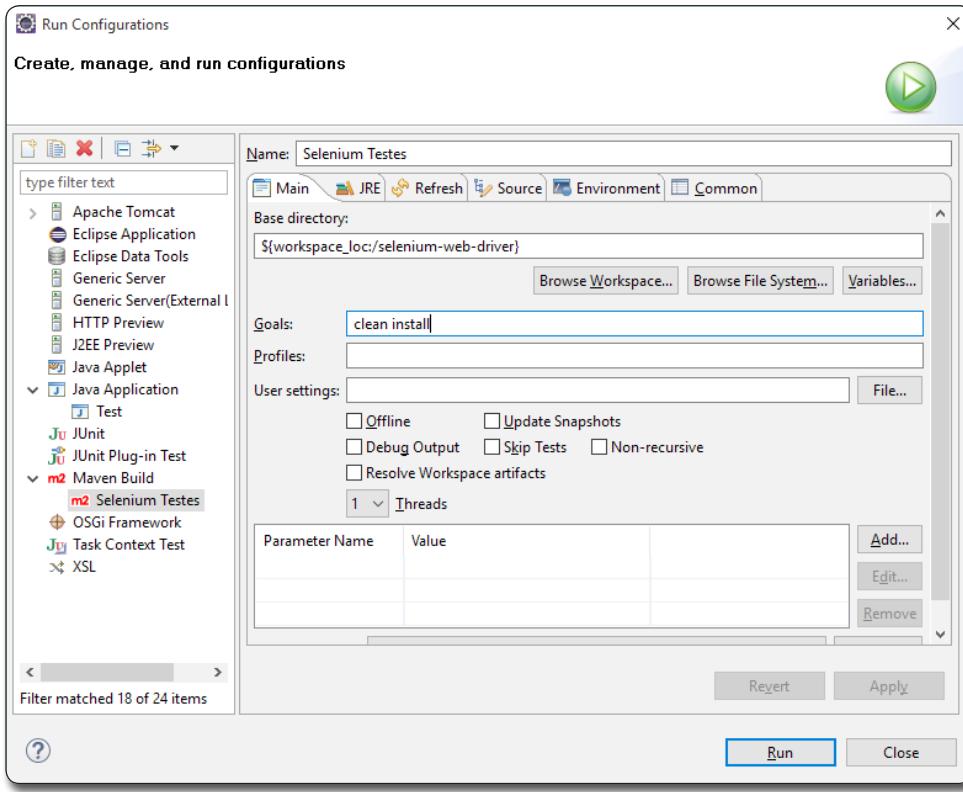


Figura 4. Configurações do Maven para executar testes

Possíveis problemas no Maven

Dependendo de como seu ambiente tenha sido configurado, pode ser que seja preciso configurar um JDK (*Java Development Kit*) que o Maven tem dependência direta. Se a execução dos seus testes retornar um erro semelhante ao ilustrado na **Listagem 3**, precisamos executar os seguintes passos:

1. Acesse o menu *Window > Preferences > Java > Installed JREs*;
2. Verifique se existe algum JDK disponível;
3. Caso só tenha JRE, clique no botão *Add...* > *Standard VM*, clique em *Next*;
4. Em seguida, clique em *Directory...* e procure pela pasta do JDK instalado na máquina (geralmente em *C:\Arquivos de Programas\Java*);
5. Clique em *Finish*, marque a checkbox do JDK importado e depois clique em *OK*.

Após isso, volte novamente à tela de *Run Configurations* e, na aba *JRE*, selecione a nova JDK na combo de *Alternate JRE*. Clique em *Run* e o código fonte, bem como o build do projeto, serão gerados com sucesso. Para saber se tudo correu bem, verifique se a mensagem exibida na **Listagem 4** aparece no final do seu Console.

Executando os testes: Plugin Surefire

Todos os passos até aqui apenas criaram a estrutura de projeto e build para os nossos testes. Todavia, ainda precisamos inserir um componente que se encarregue de executar os testes finais, que por sua vez, devem estar inseridos em uma suíte de testes.

Esta é um conjunto de testes organizados em uma ou mais classes que nos possibilita organizar e dividir os mesmos, considerando que podemos ter inúmeros testes no projeto e não queremos executar todos de uma vez sempre que usarmos os comandos do Maven.

Para tal, faremos uso de um plugin do próprio Maven, criado exclusivamente para essa finalidade: o **maven-surefire**. Ele recebe um arquivo XML num formato pré-definido para suítes de testes, que contém a(s) classe(s) de teste que deve(m) ser executada(s) no projeto. Portanto, criemos primeiramente o arquivo de suíte. Na raiz do projeto crie um novo arquivo XML clicando com o botão direito e selecionando a opção *New > Other....* Digite “xml” na caixa de pesquisa que aparecer e selecione a opção *XML > XML File*. Clique em *Next*, dê o nome *testng.xml* para o arquivo e clique em *Finish*. O arquivo deve conter a estrutura demonstrada na **Listagem 5**.

Listagem 3. Erro de JDK no Maven.

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.3:compile (default-compile) on project selenium-web-driver: Compilation failure
[ERROR] No compiler is provided in this environment. Perhaps you are running on a JRE rather than a JDK?
[ERROR] --> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
```

Listagem 4. Execução do build do projeto com sucesso.

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.162 s
[INFO] Finished at: 2015-12-07T09:13:45-03:00
[INFO] Final Memory: 18M/157M
[INFO] -----
```

Listagem 5. Conteúdo do arquivo testng.xml.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Exemplo de Execução de testes com TestNG">
<test name="Teste Simples - Google Page">
<classes>
<class name="br.edu.devmedia.selenium.PrimeiroTeste" />
</classes>
</test>
</suite>
```

Dominando o Selenium Web Driver na prática

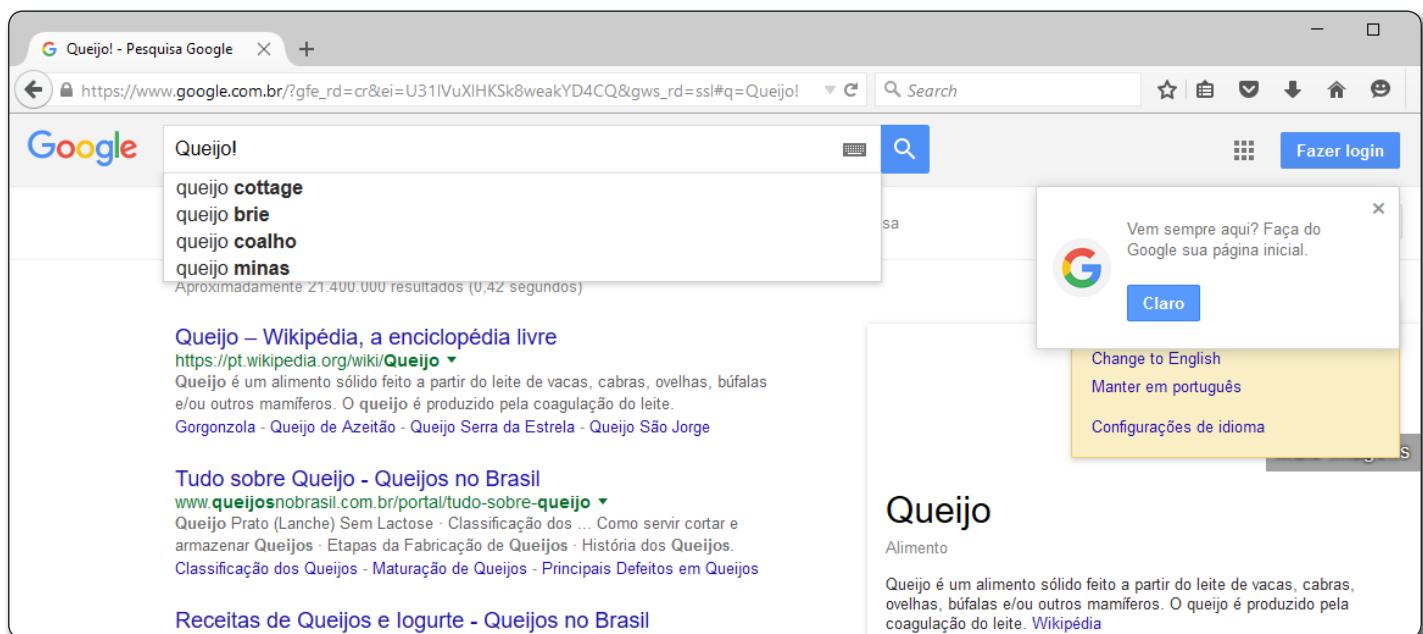


Figura 5. Print da tela do Firefox quando texto é consultado

A estrutura desse tipo de arquivo deve sempre obedecer à hierarquia: `suite > test > classes > class`. Nela, podemos ter quantas suítes de teste quisermos no mesmo arquivo, assim como vários testes em uma suíte, várias classes em um teste, e assim por diante. Para cada suíte ou teste podemos opcionalmente dar um nome à mesma, que será exibido no Console quando o teste estiver em execução. Por fim, informe o nome da(s) classe(s) de teste do seu projeto que devem, obrigatoriamente, estar contidas dentro do diretório `src/test/java`.

Em seguida, abra o `pom.xml` e inclua o plugin definido na [Listagem 6](#), logo após o plugin já incluso do `maven-compiler-plugin`.

Veja que informamos na tag `<version>` a versão 2.19, via consulta à página do `mavenrepository`, bem como o nome do arquivo `testng.xml` que criamos na tag `<suiteXmlFile>`. Se tiver mais arquivos de suítes a serem considerados pelo plugin, insira-os em tags subsequentes abaixo umas das outras.

Listagem 6. Inserindo novo plugin do Surefire.

```
<!-- Plugin do Maven para executar os testes -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.19</version>
  <configuration>
    <suiteXmlFiles>
      <!-- Arquivo(s) XML da TestNG suíte -->
      <suiteXmlFile>testng.xml</suiteXmlFile>
    </suiteXmlFiles>
  </configuration>
</plugin>
```

Para testar basta executar novamente a task `Selenium Testes` e aguardar o processo finalizar. Veremos o navegador do Firefox (se instalado) abrir duas vezes, acessar a URL do `google.com`, digitar os textos `Leite!` e `Queijo!` na caixa de pesquisa e fechar o navegador em seguida. Veja na [Figura 5](#) um print da tela do navegador no momento em que o texto é consultado.

Veremos também no Console do Eclipse o log de execução dos testes, com seu resultado final, conforme mostra a [Listagem 7](#). Veja como o valor do título da página é exibido antes da pesquisa, apenas com a palavra “Google”, e depois da pesquisa com o valor digitado acrescido do texto “ – Pesquisa Google”.

Listagem 7. Log final com resultado da execução dos testes.

TESTS

Running `TestSuite`

O título da página é: Google
O título da página é: Leite! - Pesquisa Google
O título da página é: Google
O título da página é: Queijo! - Pesquisa Google
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 30.589 sec - in `TestSuite`

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

Nota

Se tiver problemas para baixar as dependências, tente adicionar um `-U` no fim do comando de `clean` do Maven. Isso vai forçá-lo a verificar os repositórios da central para buscar as bibliotecas atualizadas.

Se os testes não funcionam, provavelmente as versões do seu **Selenium** e Firefox estão fora de sincronia. Verifique os pré-requisitos e refaça os passos até aqui para se certificar de que seu ambiente esteja configurado corretamente.

Agora temos um projeto básico configurado para executar um par de testes também básicos usando o Maven. Levando em consideração um cenário simples, com poucos testes leves, tudo vai funcionar rapidamente. Entretanto, quanto mais adicionamos testes ao seu projeto, mais as coisas vão começando a se debilitar em quesito performance. Para tentar suavizar este problema vamos utilizar todo o poder da sua máquina para executar os testes em paralelo.

Executando os testes em paralelo

Executar testes em paralelo significa muitas coisas diferentes em um ambiente que se comporta de maneira síncrona até o momento, a saber:

- Executar todos os testes em vários navegadores ao mesmo tempo;
- Executar os testes em várias instâncias do mesmo navegador.

Isso são coisas bem diferente do ponto de vista programático. Devemos utilizar nossos testes em paralelo para aumentar a abrangência do nosso site, isto é, quando estamos escrevendo testes automatizados para garantir que as coisas funcionam com o site que estamos testando, certamente considerará, de início, que seu site deve funcionar em todos os navegadores. A realidade é que isto não somente é uma máxima, como também uma difícil realidade de encarar: já que muitos navegadores lá fora suportarão todo tipo de recurso como, por exemplo, sites

que utilizam AJAX de forma intensa, ou o Flash no navegador web *Lynx*, baseado em texto, que pode ser usado em uma janela no terminal Linux.

Criar testes para todos os navegadores suportados pelo Selenium é bom, mas temos alguns problemas aqui. Algo que a maioria das pessoas não sabem é que o suporte ao navegador oficial do núcleo do Selenium é a atual versão do navegador e a versão anterior ao momento do lançamento de uma nova versão do Selenium. Na prática, ele pode muito bem trabalhar em navegadores mais antigos, já que a equipe do framework faz vários trabalhos para tentar garantir que eles não quebrem tal suporte. No entanto, se desejar executar uma série de testes no Internet Explorer 6, 7 ou 8, saiba que está executando testes em navegadores que não são oficialmente suportados pelo Selenium.

Em seguida, vem o nosso próximo conjunto de problemas. O Internet Explorer é suportado apenas em máquinas Windows, e podemos ter apenas uma única versão dele instalada na máquina por vez. Existem alguns *hacks* que podemos usar para instalar várias versões do Internet Explorer em uma mesma máquina, mas se fizer isso não irá conseguir testes precisos. O Safari é suportado apenas em máquinas OS X e, como no IE, só poderemos ter uma versão instalada de cada vez.

Logo, fica evidente que, mesmo que desejemos executar os testes em todos os navegadores suportados pelo Selenium, não seremos capazes de fazê-los em uma única máquina.

Neste ponto os desenvolvedores tendem a modificar a estrutura dos testes para que aceitem apenas uma lista de navegadores. Eles escrevem algum código que detecta, ou especifica os navegadores disponíveis em uma máquina. Depois de fazerem isso, começam a executar os testes em mais algumas máquinas em paralelo, mais ou menos como mostra a **Figura 6**.

Outra preocupação recorrente é em relação à engine de JavaScript do navegador escolhido. O Internet Explorer é notoriamente conhecido pelas suas dificuldades em conciliar a execução de scripts (principalmente paralelos) com performance em aplicações web. Além disso, diferentes navegadores executam testes em velocidades diferentes porque os seus motores de JavaScript não são iguais.

Quando um teste falhar, devemos descobrir em qual navegador ele está sendo realizado, bem como o porquê da falha. Isso pode levar apenas um minuto do seu tempo, mas todos os minutos se somam. Então, por que não executamos os testes em apenas um tipo de navegador no momento? Vamos proceder da seguinte forma: faremos os testes em um navegador bom e rápido, e depois nos preocuparemos com a compatibilidade *cross-browser*.

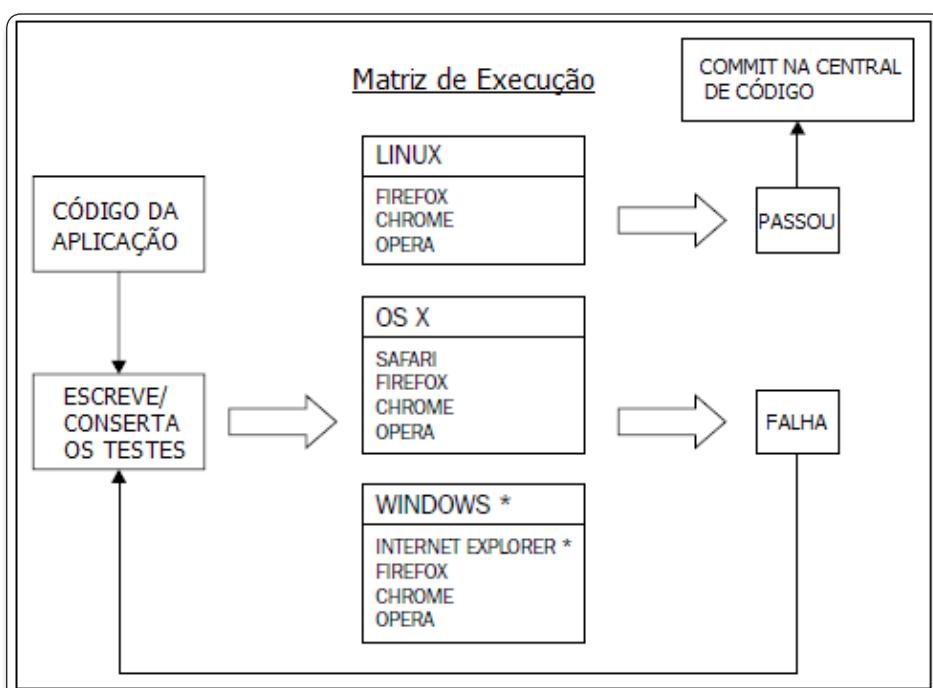


Figura 6. Matriz de execução de testes por browsers

Nota

É uma boa ideia escolher somente um navegador para rodar os testes nas máquinas de desenvolvimento. Podemos usar um servidor de CI se preocupar com a cobertura do navegador como parte da nossa construção de pipeline. Procure focar sempre em navegadores mais usados no mercado, que tenham comprovadamente bons motores JavaScript.

Testes paralelos com TestNG

O TestNG suporta paralelamente threads *out-of-the-box*, só precisamos dizer onde usar. No nosso caso, estamos interessados nas definições de configuração em paralelo com a propriedade *threadCount*. Criaremos conjuntos paralelos de execução de métodos. Isto irá pesquisar em nosso projeto os métodos que possuem a anotação *@Test* e irá agrupá-los em um grande *pool* de testes. O plugin do Maven que adicionamos, então, leva os testes para fora deste *pool* e executa-os. O número de testes que serão executados simultaneamente dependerá de quantas threads estarão disponíveis. Usaremos a propriedade *threadCount* para controlar isso.

É importante notar que isso não garante a ordem que os testes serão executados. Estamos usando as definições da configuração da *threadCount* para controlar como muitos testes serão executados em paralelo, mas, como já deve ter notado, não especificamos um número. Em vez disso, usamos a variável *\${threads}* no Maven; isso vai levar o valor à propriedade *threads* do plugin Maven.

Então, começemos executando o seguinte comando:

```
mvn clean install
```

Esse comando irá utilizar o valor padrão 1 no arquivo do **POM**. No entanto, se desejar uma quantidade maior de threads, é só executar:

```
mvn clean install -Dthreads=2
```

Ele agora irá substituir o valor de 1 armazenado no arquivo **POM** para o valor 2. Isso nos dá a capacidade de ajustar o número de *threads* que usamos para executar nossos testes sem fazer quaisquer alterações de código físico.

Agora é preciso modificar o nosso código para tirar vantagem disso. Anteriormente, instanciamos uma instância do *FirefoxDriver* em cada um dos nossos testes. Vamos levar esta classe para fora do teste e colocar a instanciação do navegador em sua própria classe chamada *WebDriverThread*. Mas antes disso, vamos criar uma classe chamada *DriverFactory* que tratará da triagem das threads de uma forma geral. Siga as instruções da **Listagem 8** para criar a referida classe.

A classe mantém dois atributos globais:

- *webViewThreadPool*, referente ao pool de threads com os drivers que usaremos para executar em paralelo os testes. O tipo genérico dessa lista é *WebDriverThread* (que ainda criaremos a seguir) e ela é instanciada a partir do método *synchronizedList()* de *Collections*, o qual retorna uma lista síncrona de dados;
- o segundo atributo se refere à lista de threads que usaremos dentro do método de inicialização dos dados. Este método está

declarado com a anotação *@BeforeSuite*, que funciona como um construtor de testes, executando antes de qualquer outra estrutura da suíte.

Listagem 8. Classe *DriverFactory* – fábrica de drivers.

```
package br.edu.devmedia.selenium;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.openqa.selenium.WebDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.AfterSuite;
import org.testng.annotations.BeforeSuite;

public class DriverFactory {

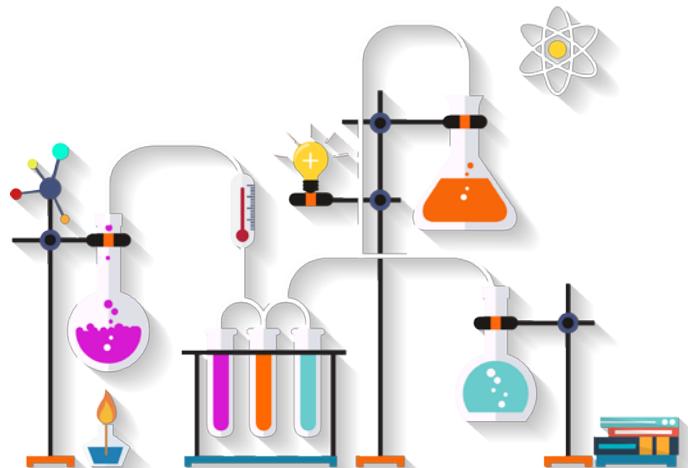
    private static List<WebDriverThread> webViewThreadPool =
        Collections.synchronizedList(new ArrayList<WebDriverThread>());
    private static ThreadLocal<WebDriverThread> driverThread;

    @BeforeSuite
    public static void instantiateDriverObject() {
        driverThread = new ThreadLocal<WebDriverThread>() {
            @Override
            protected WebDriverThread initialValue() {
                WebDriverThread webViewThread = new WebDriverThread();
                webViewThreadPool.add(webViewThread);
                return webViewThread;
            }
        };
    }

    public static WebDriver getDriver() throws Exception {
        return driverThread.get().getDriver();
    }

    @AfterMethod
    public static void clearCookies() throws Exception {
        getDriver().manage().deleteAllCookies();
    }

    @AfterSuite
    public static void closeDriverObjects() {
        for (WebDriverThread webViewThread : webViewThreadPool) {
            webViewThread.quitDriver();
        }
    }
}
```



Estamos fazendo isso para isolar cada instância de WebDriver se certificando de que nenhum cruzamento de dados aconteça entre os testes. Quando eles começarem em paralelo, não queremos que diferentes testes disparem comandos para a mesma janela do navegador. Cada instância do WebDriver agora está bloqueada com segurança em sua própria thread.

No método estático getDriver() acessamos o método get() da lista de threads que retorna a próxima disponível, retornando o driver em si via método getDriver(). Também usamos as anotações @AfterMethod para executar após o método estático, cuja função será limpar os cookies do navegador para a próxima suíte de testes; e @AfterSuite, que se encarrega de fechar os drivers (os navegadores) fisicamente. Veja que neste último método estamos fazendo uso do método quit() do WebDriver que, por sua vez, trata de matar a instância do navegador que está aberta. Existe um outro método chamado close() que pode ser usado para essa finalidade, mas ele não fecha o navegador, mas apenas a aba atual que estiver aberta. Esse método é mais usado para quando abrimos várias abas no navegador e desejamos fechar apenas a atual.

Criemos agora uma nova classe de nome WebDriverThread, tal como mostra a **Listagem 9**.

Listagem 9. Classe WebDriverThread – thread individual.

```
package br.edu.devmedia.selenium;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.remote.DesiredCapabilities;

public class WebDriverThread {
    private WebDriver webdriver;
    private final String operatingSystem = System.getProperty("os.name")
        .toUpperCase();
    private final String systemArchitecture = System.getProperty("os.arch");

    public WebDriver getDriver() throws Exception {
        if (null == webdriver) {
            System.out.println("");
            System.out.println("Sistema Operacional Atual:" + operatingSystem);
            System.out.println("Arquitetura Atual:" + systemArchitecture);
            System.out.println("Browser selecionado: Firefox");
            System.out.println("");
            webdriver = new FirefoxDriver(DesiredCapabilities.firefox());
        }
        return webdriver;
    }

    public void quitDriver() {
        if (null != webdriver) {
            webdriver.quit();
            webdriver = null;
        }
    }
}
```

Essa classe, por sua vez, se encarrega de manter uma referência única ao objeto WebDriver do Selenium, além das propriedades de tipo do Sistema Operacional e arquitetura do mesmo (isso para deixar nossos testes mais completos e sabermos exatamente em que tipo de ambiente os estamos executando). Por isso, o método getDriver() imprime todas estas informações antes de criar o driver do Firefox efetivamente. No final, criamos o método quitDriver() para finalizar o mesmo.

Tudo que resta agora é criar uma nova classe de testes, muito semelhante a primeira, de nome TesteBasicoWD, que ao invés de criar a própria instância de driver, buscará do pool uma disponível. Para isso, siga os passos mostrados na **Listagem 10**.

Listagem 10. Classe de testes TesteBasicoWD.

```
package br.edu.devmedia.selenium;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedCondition;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.testng.annotations.Test;

public class TesteBasicoWD extends DriverFactory {
    private void exemploGoogleQuePesquisaPor(final String stringPesquisa)
        throws Exception {
        WebDriver driver = DriverFactory.getDriver();

        driver.get("http://www.google.com");

        WebElement campoPesquisado = driver.findElement(By.name("q"));

        campoPesquisado.clear();
        campoPesquisado.sendKeys(stringPesquisa);

        System.out.println("O título da página é:" + driver.getTitle());

        campoPesquisado.submit();

        (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver objDriver) {
                return objDriver.getTitle().toLowerCase().startsWith(stringPesquisa
                    .toLowerCase());
            }
        });
        System.out.println("O título da página é:" + driver.getTitle());
    }

    @Test
    public void googleExemploQueijo() throws Exception {
        exemploGoogleQuePesquisaPor("Queijo!");
    }

    @Test
    public void googleExemploLeite() throws Exception {
        exemploGoogleQuePesquisaPor("Leite!");
    }
}
```

Modificamos nossa classe de teste básico para que estenda de `DriverFactory`. Ao invés de instanciar uma nova instância de `FirefoxDriver` no teste, estamos chamando o método `DriverFactory.getDriver()` para obter uma instância de `WebDriver` válida. Finalmente, teremos removido o `driver.quit()` de cada teste, já que isso tudo deve ser feito por nossa classe `DriverFactory` agora.

Vamos girar o nosso teste novamente usando o seguinte comando:

```
mvn clean install
```

O teste executará da mesma forma que antes. Porém, agora podemos especificar algumas threads de desempenho via comando:

```
mvn clean install -Dthreads=5
```

Repare que dessa vez abrirão dois navegadores Firefox, com ambos os testes executados em paralelo e, em seguida, ambos os navegadores fecharão novamente. A maior consequência direta desse tipo de implementação é a diminuição do tempo necessário para executar uma suíte completa de testes. Isso ocorre porque a maior parte do tempo é gasto para compilar o código e carregar os navegadores. No entanto, à medida que adicionamos mais testes, essa diminuição no tempo torna-se cada vez mais e mais evidente.

Este é, provavelmente, um bom momento para ajustar o seu arquivo `TesteBasicoWD.java` e começar a adicionar mais alguns testes que buscam termos de pesquisas diferentes, ou até acessam sites diferentes. Mexa um pouco com o número de threads e veja quantos navegadores concorrentes podemos obter funcionando ao mesmo tempo. Certifique-se de imprimir no Console os tempos de execução para ver os ganhos de velocidade que está realmente tendo. Chegará um ponto onde alcançaremos os limites de *hardware* do seu computador e adicionar mais threads vai realmente retardar o processo todo ao invés de fazê-lo mais rapidamente. Ajustar os testes de acordo com o ambiente de hardware é uma parte importante da gestão de testes em vários segmentos.

Como de praxe, manter as janelas do navegador abertas enquanto se executa todos os testes não irá funcionar em todos os casos.

Às vezes, podemos ter um site que define os cookies do lado do servidor, os quais o Selenium desconhece. Neste caso, limpar os cookies pode não ter efeito nenhum e podemos achar que fechar o navegador é a única maneira de garantir um ambiente limpo para cada teste.

Se usar o `InternetExplorerDriver`, por exemplo, provavelmente perceberemos ao usar versões um pouco mais antigas do navegador (por exemplo, IE8 e IE9) que os testes ficarão cada vez mais

Listagem 11. Interface de configuração dos drivers - DriverSetup.

```
package br.edu.devmedia.selenium;  
  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.remote.DesiredCapabilities;  
  
public interface DriverSetup {  
    WebDriver getWebDriverObject(DesiredCapabilities desiredCapabilities);  
  
    DesiredCapabilities getDesiredCapabilities();  
}
```

lentos até que cheguem a um impasse. Infelizmente, as versões mais antigas do IE não são perfeitas e têm vários problemas de vazamento de memória.

Suporte a múltiplos navegadores

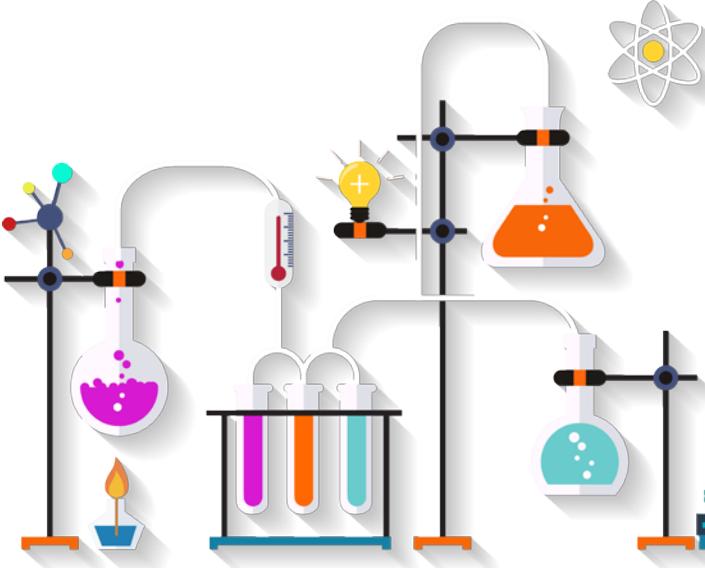
Até agora nossos testes em paralelo podem executar várias instâncias do navegador ao mesmo tempo. No entanto, ainda estamos usando apenas um tipo de condutor, o `FirefoxDriver`. Mas nosso código ainda não está adaptado para trabalhar com outros tipos de browser, isso porque o Firefox já tem uma comunicação muito próxima com o universo Selenium, inclusive disponibilizando um add-on para testes com o framework.

Para corrigir isso precisamos, inicialmente, criar uma nova interface que irá conter as assinaturas dos métodos que usaremos para criar os drivers e suas capacidades de forma dinâmica no código. Portanto, crie-a no pacote do projeto e dê o nome de `DriverSetup`. Inclua o código contido na **Listagem 11** à mesma.

Veja que ela dispõe de dois métodos: o primeiro retorna um novo objeto do tipo `WebDriver` com base nas *capabilities* (capacidades) enviadas por parâmetro; e o segundo instancia essas capacidades de acordo com as especificidades de cada navegador.

Agora precisamos implementar a estrutura de código concreta que irá implementar esta interface e garantir que os métodos sejam sobreescritos em cada browser. Para isso, faremos uso das estruturas de *Enum (enumerations)* que se encaixam muito bem nesse propósito de construir código estático e em formato de listas de valores constantes. Dê o nome da estrutura de `DriverType` e adicione o conteúdo da **Listagem 12** ao mesmo.

Veja que ela apenas implementa as configurações individuais dos navegadores (Firefox, Chrome, IE, Safari e Opera) mais usados do mercado.



Listagem 12. Código do enum de DriverType.

```
package br.edu.devmedia.selenium;

import java.util.Arrays;
import java.util.HashMap;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.ie.InternetExplorerDriver;
import org.openqa.selenium.opera.OperaDriver;
import org.openqa.selenium.remote.CapabilityType;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.safari.SafariDriver;

public enum DriverType implements DriverSetup {

    FIREFOX {
        public DesiredCapabilities getDesiredCapabilities() {
            DesiredCapabilities capabilities = DesiredCapabilities.firefox();
            return capabilities;
        }

        public WebDriver getWebDriverObject(DesiredCapabilities capabilities) {
            return new FirefoxDriver(capabilities);
        }
    },
    CHROME {
        public DesiredCapabilities getDesiredCapabilities() {
            DesiredCapabilities capabilities = DesiredCapabilities.chrome();
            capabilities.setCapability("chrome.switches",
                Arrays.asList("--no-default-browser-check"));
            HashMap<String, String> chromePreferences = new HashMap<String, String>();
            chromePreferences.put("profile.password_manager_enabled", "false");
            capabilities.setCapability("chrome.prefs", chromePreferences);
            return capabilities;
        }

        public WebDriver getWebDriverObject(DesiredCapabilities capabilities) {
            return new ChromeDriver(capabilities);
        }
    },
    IE {
        public DesiredCapabilities getDesiredCapabilities() {
            DesiredCapabilities capabilities = DesiredCapabilities.internetExplorer();
            capabilities.setCapability(CapabilityType.ForSeleniumServer
                .ENSURING_CLEAN_SESSION, true);
            capabilities.setCapability(InternetExplorerDriver
                .ENABLE_PERSISTENT_HOVERING, true);
            capabilities.setCapability("requireWindowFocus", true);
            return capabilities;
        }

        public WebDriver getWebDriverObject(DesiredCapabilities capabilities) {
            return new InternetExplorerDriver(capabilities);
        }
    },
    SAFARI {
        public DesiredCapabilities getDesiredCapabilities() {
            DesiredCapabilities capabilities = DesiredCapabilities.safari();
            capabilities.setCapability("safari.cleanSession", true);
            return capabilities;
        }

        public WebDriver getWebDriverObject(DesiredCapabilities capabilities) {
            return new SafariDriver(capabilities);
        }
    },
    OPERA {
        public DesiredCapabilities getDesiredCapabilities() {
            DesiredCapabilities capabilities = DesiredCapabilities.operaBlink();
            return capabilities;
        }

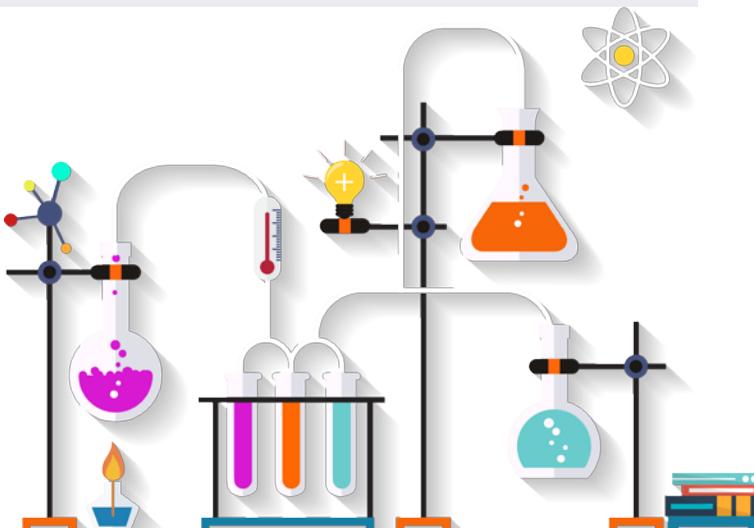
        public WebDriver getWebDriverObject(DesiredCapabilities capabilities) {
            return new OperaDriver(capabilities);
        }
    }
}
```

Para chamar qualquer configuração basta sobrescrever a instanciação do driver na classe WebDriverThread para o seguinte código:

```
webdriver = DriverType.CHROME.getWebDriverObject(DriverType.CHROME.getDesiredCapabilities());
```

Salve todo o conteúdo e execute novamente o comando do Maven para dar build na aplicação. O resultado pode ser conferido na **Figura 7**.

Dependendo das configurações do seu sistema operacional, pode se fazer necessário o download do driver do navegador específico para funcionar em conjunto com o Selenium Web Driver. Veja na seção **Links** a URL com a relação destes itens.



Dominando o Selenium Web Driver na prática

The screenshot shows a Google search results page for the query "Queijo". The search bar at the top contains "Queijo". Below it, a dropdown menu lists suggestions: "tipos de queijo", "como fazer queijo", "queijo benefícios", and "queijo caseiro". The main search results area displays two cards. The first card is for "Queijo – Wikipédia, a encyclopédia livre" with the URL <https://pt.wikipedia.org/wiki/Queijo>. It includes a snippet about cheese being made from the milk of various mammals and links to "Gorgonzola", "Queijo de Azeitão", "Queijo Serra da Estrela", and "Queijo São Jorge". The second card is for "Tudo sobre Queijo - Queijos no Brasil" with the URL www.queijosnobrasil.com.br/portal/tudo-sobre-queijo. It includes a snippet about cheese types like Prato (Lanche) Sem Lactose, Classificação dos Queijos, Como servir cortar e armazenar Queijos, Etapas da Fabricação de Queijos, História dos Queijos, Classificação dos Queijos, Maturação de Queijos, and Principais Defeitos em Queijos. To the right of these cards is a sidebar with a large image of various cheeses and a link "Mais imagens".

Figura 7. Resultado da execução dos testes no Google Chrome

O Selenium é uma ferramenta extremamente madura para lidar com o desenvolvimento de testes automatizados em vários âmbitos e te auxiliar no processo de teste das suas aplicações web.

Apesar de ser amplamente aceito, trata-se de um projeto aberto, mantido por desenvolvedores que dedicam seu tempo a contribuir e facilitar as nossas vidas. Por causa disso, o framework está em constante aprimoramento, inclusive de correções sobre seus bugs. Na seção **Links** encontra-se a página que lista todas essas melhorias para que você possa, eventualmente, se guiar quando erros acontecerem nos seus projetos.

Autor



Sueila Sousa

É tester e entusiasta de tecnologias front-end. Atualmente trabalha como analista de testes na empresa Indra, com foco em projetos de desenvolvimento de sistemas web, totalmente baseados em JavaScript e afins. Possui conhecimentos e experiências em áreas como Gerenciamento de processos, banco de dados, além do interesse por tecnologias relacionadas ao desenvolvimento e teste client side.



Links:

Página oficial do Apache Maven

<http://maven.apache.org/index.html>

Página de download do Eclipse IDE

<http://eclipse.org/downloads>

Página do Selenium Web Browser

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

Página do mvnrepository

<http://mvnrepository.com/>

Lista de drivers para browsers no Selenium

<http://www.seleniumhq.org/about/platforms.jsp>

Página de bugs do Selenium

<https://code.google.com/p/selenium/issues/list>

Você gostou deste artigo?

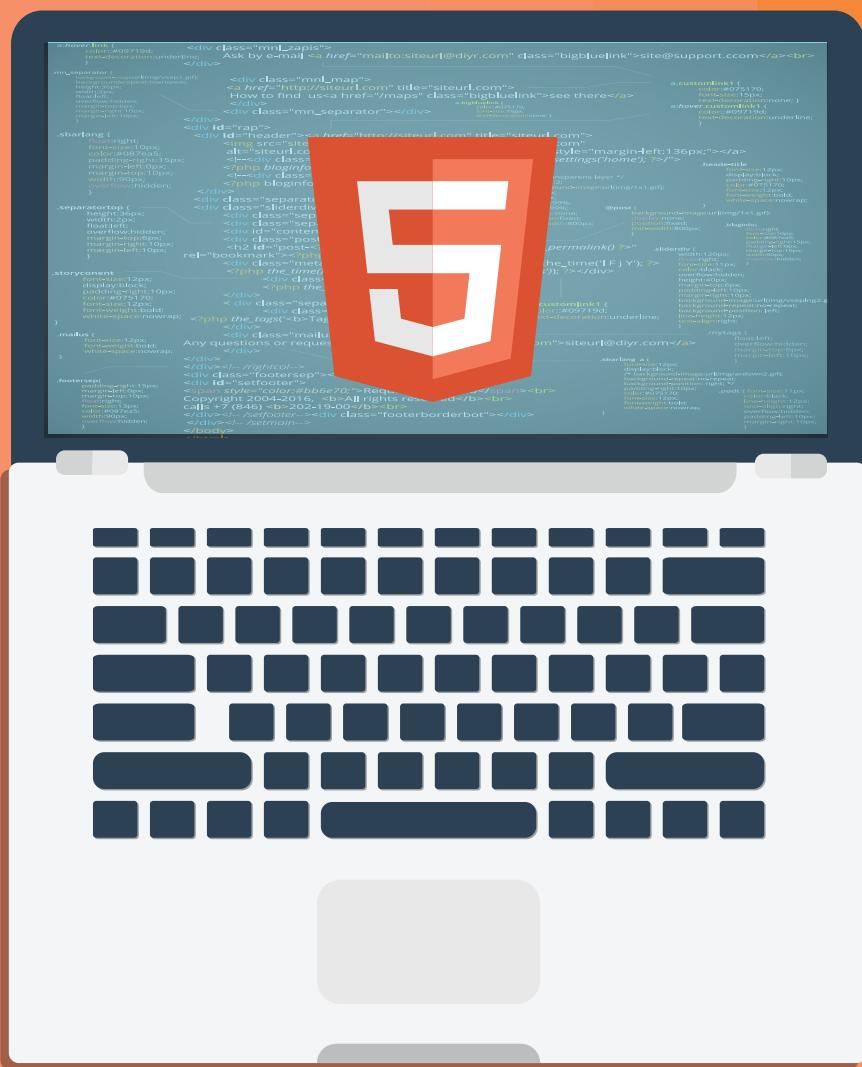
Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Guia HTML 5

Um verdadeiro manual de referência com tudo que você precisa sobre HTML!



DEVMEDIA

<http://www.devmedia.com.br/guias/guia-html/3>

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia. Somos uma equipe composta de gente que entende e gosta do que faz, assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

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

Serviços

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

1-to-1

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



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486